

Theoretische Informatik

Allgemeine Informatik
Medizinische Informatik
Technische Informatik
Wintersemester 2019/20

Prof. Dr. Wolfgang Mauerer
wolfgang.mauerer@oth-regensburg.de

1. Überblick und Einführung

- 1.1 Administrativa
- 1.2 Warum Theorie?

2. Formale Sprachen und Automaten

- 2.1 Endliche Automaten
- 2.2 Reguläre Sprachen I
- 2.3 Worterzeugung und Grammatiken
- 2.4 Chomsky-Hierarchie
- 2.5 Reguläre Sprachen und endliche Automaten
- 2.6 Nicht-Deterministische

endliche Automaten

- 2.7 Grammatiken und NEAs
- 2.8 Äquivalenz von NEAs und DEAs
- 2.9 Reguläre Ausdrücke
- 2.10 Das Pumping-Lemma
- 2.11 Automatenminimierung
- 2.12 Abschlusseigenschaften regulärer Sprachen
- 2.13 Kontextfreie Sprachen
- 2.14 Kellerautomaten
- 2.15 CYK-Algorithmus

3. Berechenbarkeitstheorie

- 3.1 Turing-Maschinen

- 3.2 LBAs und der Satz von Kuroda
- 3.3 Berechenbarkeit und Church-Turing-These
- 3.4 Varianten von Turing-Maschinen
- 3.5 Berechnungskomplexität
- 3.6 Alternative Berechnungsmodelle
- 3.7 Universelle Turing-Maschinen
- 3.8 Das Halteproblem

4. Komplexitätstheorie

- 4.1 Definitionen
- 4.2 Komplexitätsklassen
- 4.3 Struktur von NP

OTH Regensburg

- ▶ Professor für theoretische Informatik
- ▶ Leiter Labor für Digitalisierung
- ▶ Forschungsinteressen
 - ▶ Statistik und maschinelles Lernen in der SW-Entwicklung
 - ▶ (Echtzeit-)Systeme gemischter Kritikalität
 - ▶ Innovation durch Theorie (Quantencomputing, Zufallserzeugung, ...)
- ▶ Hacken *und* Beweisen

Siemens Corporate Research

- ▶ Linux (Kernel, Low-Level)
- ▶ Harte Echtzeit (MRT, Simatic, Android, ...)
- ▶ Statistik und maschinelles Lernen für SW-Architektur: github.com/lfd
- ▶ Früher: MPL (QIT, QED)

Scheinkriterien

- ▶ Klausur (90min) am Ende des Semesters
- ▶ Keine Hilfsmittel
- ▶ Material der gesamten Vorlesung relevant (orientiert sich an Übungsaufgaben).

IT₁

- ▶ Mo, 10:00–13:15, K017

IN₁

- ▶ Fr, 10:00–13:15, U212

Übungsaufgaben

- ▶ Neues Übungsblatt jede Woche am Montag.
- ▶ Bearbeitung *vor*, in und *nach* den Übungen
- ▶ Abgabe in *fixen* Zweiergruppen erlaubt
 - ▶ Elektronisch im GRIPS-System
 - ▶ Deadline: Montag eine Woche später, 23:59 Uhr
- ▶ Mindestens 50% der Aufgaben müssen abgegeben und positiv votiert worden sein!
 - ▶ Positiv votieren = Sie gehen nachvollziehbar davon aus, dass Sie die Aufgabe korrekt bearbeitet haben
 - ▶ Lösung muss nicht perfekt sein
 - ▶ *Jeder* Student muss *einzel*n votieren (auch bei Bearbeitung in Zweiergruppe)

IM

<i>Gruppe</i>	<i>Tag</i>	<i>Uhrzeit</i>	<i>Raum</i>	<i>Betreuer</i>
1	Mi	15:15–16:45	K002	Emanuel Erben
2	Mi	15:15–16:45	K008	Lukas Schmidbauer
3	Do	15:15–16:45	K008	Lukas Schmidbauer

IN

<i>Gruppe</i>	<i>Tag</i>	<i>Uhrzeit</i>	<i>Raum</i>	<i>Betreuer</i>
1	Di	08:15–09:45	K017	Emanuel Erben
2	Di	11:45–13:15	K006	Florian Loher
3	Di	13:30–15:00	K013	Florian Loher

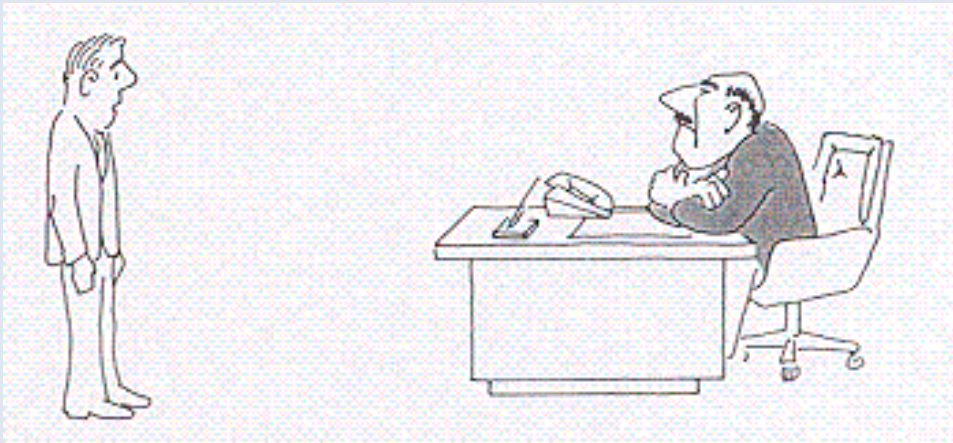
Start Übungsbetrieb

- Der Übungsbetrieb startet in KW₄₁

- ▶ Definiert im *Modulhandbuch*
- ▶ 6 SWS $\Rightarrow$ 240h Arbeitsaufwand
- ▶ Ca. 90h Vorlesung und Übungen, ca. 150h Eigenstudium
- ▶ Selbständige Beschäftigung erwünscht und *notwendig!*

- ▶ Vorlesung mit seminaristischen Unterrichtselementen.
- ▶ Aktive Teilnahme und Rückfragen erwünscht.

Probleme (nicht)lösen



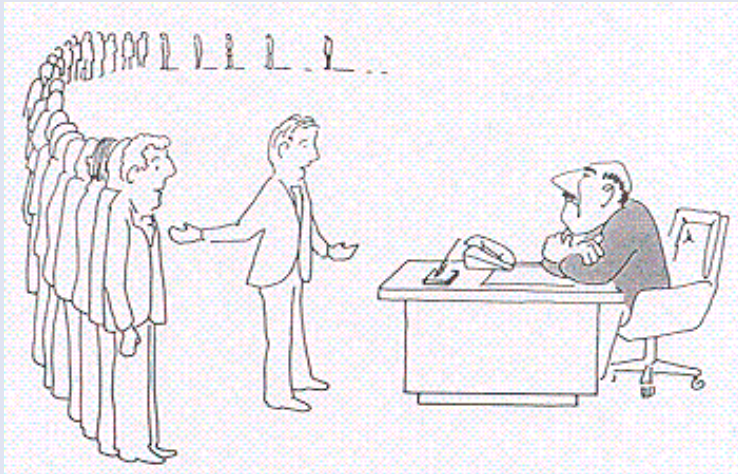
Bildquelle: Garey and Johnson, *Computers and Intractability*

Probleme (nicht)lösen



Bildquelle: Garey and Johnson, *Computers and Intractability*

Probleme (nicht)lösen



Bildquelle: Garey and Johnson, *Computers and Intractability*

E. W. Dijkstra

»In der Informatik geht es genauso wenig um Computer, wie in der Astronomie um Teleskope.«

Taschenbuch der Informatik

»Wissenschaft, die sich mit den *theoretischen Grundlagen*, den Mitteln und Methoden sowie mit der Anwendung der elektronischen Datenverarbeitung (EDV) beschäftigt, d.h. mit allen Aspekten der Informationsverarbeitung unter Einsatz von Computern einschließlich ihres Einflusses auf die Gesellschaft.«

Etymologie

- ▶ Informatik = *Information* und *Mathematik*
- ▶ Dipl. Ing. ≠ Dipl. Inf.
- ▶ Verstehen *und* anwenden!

Was ist Theorie?

- ▶ (Vereinfachtes) Modell der Realität
- ▶ Erklärung einer möglichst großen Klasse von Phänomenen mit einer möglichst geringen Anzahl von Annahmen
- ▶ Quantitative, verifizierbare Prognosen

Theorie-Missbrauch

- ▶ »Meiner Theorie nach wird Deutschland Weltmeister!«
- ▶ »Theorien zur Scheidung von ⟨Promi xyz⟩«
- ▶ »Das funktioniert in der Theorie«
- ▶ I. Kant (1793): *Über den Gemeinspruch: Das mag in der Theorie richtig sein, taugt aber nicht für die Praxis*

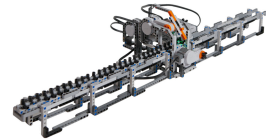
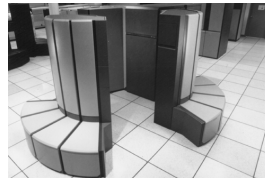
Was ist Theorie?

- ▶ (Vereinfachtes) Modell der Realität
- ▶ Erklärung einer möglichst großen Klasse von Phänomenen mit einer möglichst geringen Anzahl von Annahmen
- ▶ Quantitative, verifizierbare Prognosen

Was ist Theorie *nicht*?

- ▶ Politisch
- ▶ Unfehlbar
- ▶ Frei von Kontroversen und Skeptizismus

Welcher Rechner kann mehr als andere?



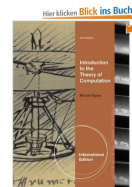
Themengebiete

- ▶ Formale Sprachen und Automatentheorie
- ▶ Berechenbarkeitstheorie
- ▶ Komplexitätstheorie

Fragestellungen

- ▶ Welche Aufgaben kann ein Computer lösen, welche nicht?
- ▶ Kann jeder Computer die gleichen Problem lösen?
- ▶ Wie kann man einen Computer mathematisch definieren?
- ▶ *Was ist ein Computer?*
- ▶ Welche Funktionen sind berechenbar, welche nicht?
- ▶ Was ist formal ein Algorithmus?
- ▶ Rechenzeit/Speicherplatz?
- ▶ Klassifikation von Problemen?
- ▶ Welche Teile der Informatik sind »zeitlos« gültig (und nicht nach zwei Handygenerationen veraltet)?

- ▶ Dirk W. Hoffmann, *Theoretische Informatik*, 2. Auflage, Carl Hanser Verlag, 2011
- ▶ Michael Sipser, *Introduction to the Theory of Computation*, 3rd edition, Cengage Learning, 2012
- ▶ Uwe Schöning, *Theoretische Informatik – kurz gefasst*, 5. Auflage, Spektrum, 2008



VORLESUNGSSKRIPT
THEORETISCHE INFORMATIK
PROF. DR. WOLFGANG MAUERER



Copyright © 2014-2015, Wolfgang Mauerer
Lizenziert unter CC BY-SA 3.0, siehe <http://creativecommons.org/licenses/by-sa/3.0/de/legalcode>.
NEUNTE AUFLAGE



Version vom 26. Januar 2015 – DRAFT

Vorlesungsskript

- ▶ Unter Dauerkonstruktion
- ▶ Download (PDF) im GRIPS-System

Taschenbuch der Informatik

»Ein Algorithmus ist eine Vorschrift zur Lösung eines Problems, die für eine Realisierung in Form eines Programms auf einem Computer geeignet ist.«

Eigenschaften

- ▶ Algorithmus gibt eindeutig Abfolge der Verarbeitungsschritte vor
- ▶ Algorithmus ist *unabhängig* von einer spezifischen Notation
- ▶ *Eigenschaften* eines Algorithmus nicht notwendigerweise im Detail bekannt/kennbar

Sprachen in der Ausbildung

- ▶ Cobol, Fortran; PL/1, APL; C, Pascal; Modula 2, Java, C#
- ▶ Konkrete Sprache: (Stilabhängiges) Hilfsmittel
- ▶ Ausweg: Pseudocode
 - ▶ Zielgruppe: Menschen, nicht Maschinen $\Rightarrow$ Irrelevante Details eliminieren
 - ▶ Siehe <http://en.wikipedia.org/wiki/Pseudocode>
 - ▶ Im Zweifelsfall: Gesunder Menschenverstand!

```
1: procedure COLLATZ( $n$ )  
2:   if  $n \equiv 0 \bmod 2$  then  
3:      $n \leftarrow \frac{n}{2}$   
4:   else  
5:      $n \leftarrow 3n + 1$   
6:   end if  
7:   Gib  $n$  aus  
8:   return COLLATZ( $n$ )  
9: end procedure
```

```
1: procedure COLLATZ( $n$ )  
2:   if gerade( $n$ ) then  
3:      $n \leftarrow \frac{n}{2}$   
4:   else  
5:      $n \leftarrow 3n + 1$   
6:   end if  
7:   PRINT( $n$ )  
8:   return COLLATZ( $n$ )  
9: end procedure
```

Solide? Hype?

Block Chain! AI! Machine Learning!

ARTICLES

<https://doi.org/10.1038/s42256-018-0002-3>

nature
machine intelligence

Corrected: Author Correction

Learnability can be undecidable

Shai Ben-David¹, Pavel Hrubeš², Shay Moran³, Amir Shpilka⁴ and Amir Yehudayoff  ^{5*}

The mathematical foundations of machine learning play a key role in the development of the field. They improve our understanding and provide tools for designing new learning paradigms. The advantages of mathematics, however, sometimes come with a cost. Gödel and Cohen showed, in a nutshell, that not everything is provable. Here we show that machine learning shares this fate. We describe simple scenarios where learnability cannot be proved nor refuted using the standard axioms of mathematics. Our proof is based on the fact the continuum hypothesis cannot be proved nor refuted. We show that, in some cases, a solution to the ‘estimating the maximum’ problem is equivalent to the continuum hypothesis. The main idea is to prove an equivalence between learnability and compression.

1. Überblick und Einführung

- 1.1 Administrativa
- 1.2 Warum Theorie?

2. Formale Sprachen und Automaten

- 2.1 Endliche Automaten
- 2.2 Reguläre Sprachen I
- 2.3 Worterzeugung und Grammatiken
- 2.4 Chomsky-Hierarchie
- 2.5 Reguläre Sprachen und endliche Automaten
- 2.6 Nicht-Deterministische

endliche Automaten

- 2.7 Grammatiken und NEAs
- 2.8 Äquivalenz von NEAs und DEAs
- 2.9 Reguläre Ausdrücke
- 2.10 Das Pumping-Lemma
- 2.11 Automatenminimierung
- 2.12 Abschlusseigenschaften regulärer Sprachen
- 2.13 Kontextfreie Sprachen
- 2.14 Kellerautomaten
- 2.15 CYK-Algorithmus

3. Berechenbarkeitstheorie

- 3.1 Turing-Maschinen

- 3.2 LBAs und der Satz von Kuroda
- 3.3 Berechenbarkeit und Church-Turing-These
- 3.4 Varianten von Turing-Maschinen
- 3.5 Berechnungskomplexität
- 3.6 Alternative Berechnungsmodelle
- 3.7 Universelle Turing-Maschinen
- 3.8 Das Halteproblem

4. Komplexitätstheorie

- 4.1 Definitionen
- 4.2 Komplexitätsklassen
- 4.3 Struktur von NP

Formale Sprachen und Automaten

Warum Sprachen?

Grundproblem: Gehört ein Wort/Satz zu einer bestimmten Klasse?

- ▶ Syntaktisch korrektes C++/Java/C#-Programm
- ▶ Netzwerkpaket
- ▶ Bankleitzahl
- ▶ ...

Zutaten

- ▶ *Grammatik*: Beschreibt eine Sprache
- ▶ *Wortproblem*: Gegeben Sprache L (durch Grammatik G beschrieben), und Wort p : Gilt $p \in L$ oder $p \notin L$?

Beispiel: Compiler (siehe Tafel)

Definition: Alphabet

Ein *Alphabet* Σ ist eine endliche Menge von Symbolen

Beispiele

- ▶ $\Sigma_1 = \{0, 1\}$ Ziffern des Binärsystems
- ▶ $\Sigma_2 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ Dezimalziffern
- ▶ $\Sigma_3 = \{0, \dots, 9, A, B, C, D, E, F\}$ Hexadezimalziffern
- ▶ $\Sigma_4 = \{a, b, \dots, z\}$ Kleinbuchstaben
- ▶ $\Sigma_5 = \{\vee, \wedge, \neg\}$ Aussagenlogische Symbole

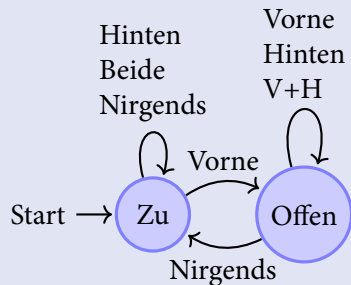
Definition: Zeichen, Wort, Sprache

- ▶ Jedes Element $\sigma \in \Sigma$ ist *Zeichen* des Alphabets
- ▶ Jedes Element $\omega \in \Sigma^*$ ist *Wort* über Σ
- ▶ Σ^n : Menge aller Wörter der Länge $n \in \mathbb{N} \setminus 0$
- ▶ *Leeres Wort* ϵ : $\epsilon \in \Sigma^*$, $\epsilon \notin \Sigma^n$
- ▶ Länge eines Wortes (Anzahl der Buchstaben): $|w|$. Achtung: $|\epsilon| = 0$
- ▶ Jede Teilmenge $L \subseteq \Sigma^*$ ist eine *formale Sprache*

Sprachverarbeitung

- ▶ Definition formaler Sprachen?
- ▶ Erkennen von Wörtern (Lösung Wortproblem)?

Automatische Tür



Zustandsübergänge

	Nirgends	Vorne	Hinten	V+H
Zu	Zu	Offen	Zu	Zu
Offen	Zu	Offen	Offen	Offen

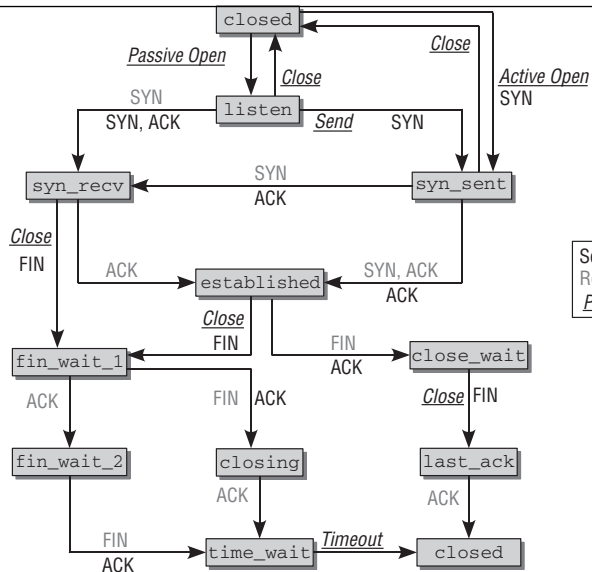
Anwendungen

Gleiche oder ähnliche Konzepte in vielen Gebieten:

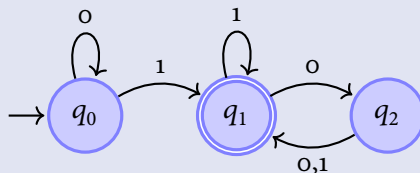
- ▶ Steuerungsaufgaben
- ▶ Netzwerkprotokolle (TCP, UDP)
- ▶ Schnelle Mustererkennung in Daten
- ▶ Verallgemeinerung: Markov-Ketten
 - ▶ Schriftklassifikation (OCR), Bilderkennung
 - ▶ Sprachverarbeitung
 - ▶ Hochfrequenz-Aktienhandel

Problemklassen

- ▶ Reine Zustandsübergänge
- ▶ »Übersetzen« von Eingaben in Ausgaben
- ▶ Erkennen von Sprachen (Fokus dieser Vorlesung)



Sending: Black
Receiving: Gray
Protocol: Underlined

Endlicher Automat M_1 

Komponenten

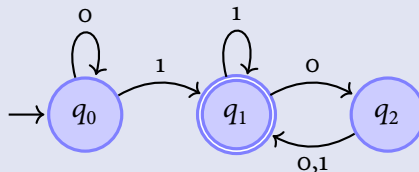
- ▶ Zustände: q_0, q_1, q_2
- ▶ Startzustand q_0
- ▶ Endzustand q_1
- ▶ Kanten: Transitionen

Aufgabe

- ▶ Lies $w \in \{0, 1\}^*$ (Buchstabe für Buchstabe), durchlaufe Transitionen
- ▶ Ende Eingabe: Akzeptierender Zustand?

Beispiel Eingaben »1101«, »00« & Charakterisierung: Siehe Tafel

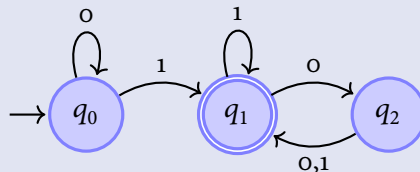
Endlicher Automat M_1



Definition: Deterministischer endlicher Automat (DEA)

Ein endlicher Automat besteht aus folgenden Komponenten:

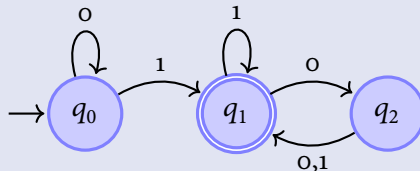
- ▶ Zuständen, in denen sich der Automat befinden kann
- ▶ Alphabet, aus dem die Eingabe generiert wird
- ▶ Eine Festlegung der Übergänge
- ▶ Einen ausgezeichneten Zustand, in dem der Automat startet
- ▶ Keinen, einen oder mehrere Endzustände

Endlicher Automat M_1 **Definition: Deterministischer endlicher Automat (DEA)**

Ein endlicher Automat ist ein 5-Tupel $(Q, \Sigma, \delta, q_0, F)$, wobei:

- ▶ Zuständen, in denen sich der Automat befinden kann
- ▶ Alphabet, aus dem die Eingabe generiert wird
- ▶ Eine Festlegung der Übergänge
- ▶ Einen ausgezeichneten Zustand, in dem der Automat startet
- ▶ Keinen, einen oder mehrere Endzustände

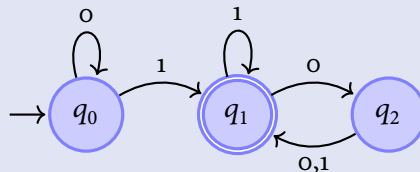
Endlicher Automat M_1



Definition: Deterministischer endlicher Automat (DEA)

Ein endlicher Automat ist ein 5-Tupel $(Q, \Sigma, \delta, q_0, F)$, wobei:

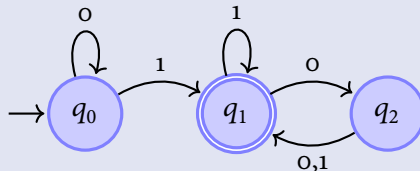
- ▶ Q eine endliche Zustandsmenge
- ▶ Alphabet, aus dem die Eingabe generiert wird
- ▶ Eine Festlegung der Übergänge
- ▶ Einen ausgezeichneten Zustand, in dem der Automat startet
- ▶ Keinen, einen oder mehrere Endzustände

Endlicher Automat M_1 **Definition: Deterministischer endlicher Automat (DEA)**

Ein endlicher Automat ist ein 5-Tupel $(Q, \Sigma, \delta, q_0, F)$, wobei:

- ▶ Q eine endliche Zustandsmenge
- ▶ Σ ein endliches Alphabet,
- ▶ Eine Festlegung der Übergänge
- ▶ Einen ausgezeichneten Zustand, in dem der Automat startet
- ▶ Keinen, einen oder mehrere Endzustände

Endlicher Automat M_1

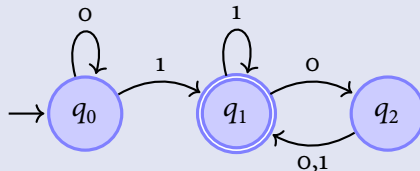


Definition: Deterministischer endlicher Automat (DEA)

Ein endlicher Automat ist ein 5-Tupel $(Q, \Sigma, \delta, q_0, F)$, wobei:

- ▶ Q eine endliche Zustandsmenge
- ▶ Σ ein endliches Alphabet,
- ▶ $\delta : Q \times \Sigma \rightarrow Q$ die Übergangsfunktion,
- ▶ Einen ausgezeichneten Zustand, in dem der Automat startet
- ▶ Keinen, einen oder mehrere Endzustände

Endlicher Automat M_1

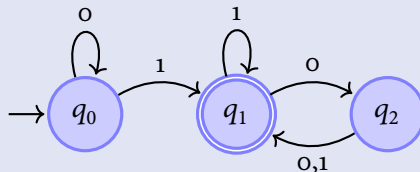


Definition: Deterministischer endlicher Automat (DEA)

Ein endlicher Automat ist ein 5-Tupel $(Q, \Sigma, \delta, q_0, F)$, wobei:

- ▶ Q eine endliche Zustandsmenge
- ▶ Σ ein endliches Alphabet,
- ▶ $\delta : Q \times \Sigma \rightarrow Q$ die Übergangsfunktion,
- ▶ $q_0 \in Q$ der Startzustand und
- ▶ Keinen, einen oder mehrere Endzustände

Endlicher Automat M_1

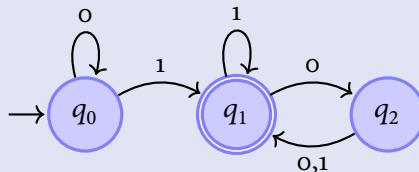


Definition: Deterministischer endlicher Automat (DEA)

Ein endlicher Automat ist ein 5-Tupel $(Q, \Sigma, \delta, q_0, F)$, wobei:

- ▶ Q eine endliche Zustandsmenge
- ▶ Σ ein endliches Alphabet,
- ▶ $\delta : Q \times \Sigma \rightarrow Q$ die Übergangsfunktion,
- ▶ $q_0 \in Q$ der Startzustand und
- ▶ $F \subseteq Q$ die Menge der akzeptierenden Zustände ist.

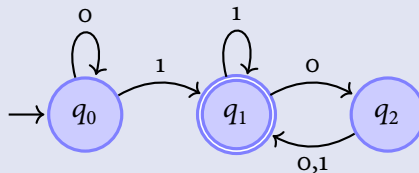
Endlicher Automat M_1



Terminologie

- ▶ M_1 akzeptiert bestimmte Wörter (beispielsweise 1101, 110000, 1100, 0101010101, 1)
- ▶ M_1 erkennt eine Sprache (welche?)
- ▶ Eine Maschine akzeptiert (im nicht-pathologischen Fall) mehrere Wörter, aber nur eine Sprache

Endlicher Automat M_1



Anmerkungen

- $\mathcal{L}(M_1) = A$ mit

$A = \{w \mid w \text{ enthält mindestens eine Eins, und das Wort endet auf eine gerade Anzahl von Nullen oder mit 1.}\}$

- Kurzform: M_1 erkennt A

Formale Definition Beispielautomat: Siehe Tafel

Weitere Beispiele: Siehe Tafel

Konstruktion endlicher Automaten

Gegeben sei die Sprache

$$L = \{w | w \text{ beginnt mit } 1 \text{ und endet mit } 0 \}$$

Gesucht: DEA M , der L akzeptiert.

Konstruktion: Siehe Tafel

Konstruktion endlicher Automaten II

Gegeben sei die Sprache

$$L = \{(ab)^n | n \in \mathbb{N}(n > 0)\}$$

Gesucht: DEA M , der L akzeptiert.

Konstruktion: Siehe Tafel

Formalisierung Rechengang

Sei M ein DEA, und sei $w = w_1 w_2 \cdots w_n$ die Eingabe mit $w_i \in \Sigma$ und $|w| = n$. M akzeptiert die Eingabe w , wenn eine Sequenz aus Zuständen $r_0, r_1, \dots, r_n$ aus Q existiert, die folgende Bedingungen erfüllt:

- ▶ Der Automat beginnt im Startzustand
- ▶ Der Automat verhält sich in jedem Schritt gemäß der Übergangsfunktion
- ▶ Die Erkennung endet in einem Endzustand

Formalisierung Rechengvorgang

Sei M ein DEA, und sei $w = w_1 w_2 \cdots w_n$ die Eingabe mit $w_i \in \Sigma$ und $|w| = n$. M akzeptiert die Eingabe w , wenn eine Sequenz aus Zuständen $r_0, r_1, \dots, r_n$ aus Q existiert, die folgende Bedingungen erfüllt:

- ▶ $r_0 = q_0$
- ▶ Der Automat verhält sich in jedem Schritt gemäß der Übergangsfunktion
- ▶ Die Erkennung endet in einem Endzustand

Formalisierung Rechenvorgang

Sei M ein DEA, und sei $w = w_1 w_2 \cdots w_n$ die Eingabe mit $w_i \in \Sigma$ und $|w| = n$. M akzeptiert die Eingabe w , wenn eine Sequenz aus Zuständen $r_0, r_1, \dots, r_n$ aus Q existiert, die folgende Bedingungen erfüllt:

- ▶ $r_0 = q_0$
- ▶ $\delta(r_i, w_{i+1}) = r_{i+1}$ für $i = 0, \dots, n-1$
- ▶ Die Erkennung endet in einem Endzustand

Formalisierung Rechenvorgang

Sei M ein DEA, und sei $w = w_1 w_2 \cdots w_n$ die Eingabe mit $w_i \in \Sigma$ und $|w| = n$. M akzeptiert die Eingabe w , wenn eine Sequenz aus Zuständen $r_0, r_1, \dots, r_n$ aus Q existiert, die folgende Bedingungen erfüllt:

- ▶ $r_0 = q_0$
- ▶ $\delta(r_i, w_{i+1}) = r_{i+1}$ für $i = 0, \dots, n-1$
- ▶ $r_n \in F$

Formalisierung Rechenvorgang II

Konfiguration eines DEA M ist durch aktuellen Zustand r und verbleibende Eingabe w eindeutig festgelegt.

- ▶ Startkonfiguration: $q_0 w_1 w_2 \cdots w_n$
- ▶ Übergang: $q_0 w_1 w_2 \cdots w_n \rightarrow q_1 w_2 \cdots w_n \Leftrightarrow \delta(q_0, w_1) = q_1$

Alternative Definition Akzeptanz

- ▶ Induktive Fortsetzung von δ auf $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$:
 - ▶ $\hat{\delta}(q, \epsilon) = q$
 - ▶ Für $n > 0$: $\hat{\delta}(q, w_1 w_2 \dots w_n) = \delta(\hat{\delta}(q, w_1 \dots w_{n-1}), w_n)$
- ▶ DEA akzeptiert $w = w_1 w_2 \cdots w_n \Leftrightarrow \hat{\delta}(q_0, w) \in F$

Beispiel: Siehe Tafel.

Alternative Definition Akzeptanz

- ▶ Induktive Fortsetzung von δ auf $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$:
 - ▶ $\hat{\delta}(q, \epsilon) = q$
 - ▶ Für $n > 0$: $\hat{\delta}(q, w_1 w_2 \dots w_n) = \delta(\hat{\delta}(q, w_1 \dots w_{n-1}), w_n)$
- ▶ DEA akzeptiert $w = w_1 w_2 \dots w_n \Leftrightarrow \hat{\delta}(q_0, w) \in F$

Akzeptierte Sprache eines DEA

$$\begin{aligned}\mathcal{L}(M) &= \{w_1 w_2 \dots w_n \mid \hat{\delta}(q_0, w_1 w_2 \dots w_n) \in F\} \\ &= \{w \mid \hat{\delta}(q_0, w) \in F\}\end{aligned}$$

Definition: Reguläre Sprache

Eine Sprache, die von einem deterministischen endlichen Automaten erkannt wird, bezeichnet man als *reguläre Sprache*

Definition: Reguläre Operationen

Zusammensetzen von Wörtern:

- ▶ *Konkatenation*: Für $u, w \in \Sigma^*$ gibt $u \cdot w = uw$ das durch Hintereinanderschreiben zusammengesetzte Wort an.
- ▶ *Vereinigung*: Für $u, w \in \Sigma^*$ gibt $u + w = u|w$ (u oder w) die Vereinigung der Wörter an.
- ▶ *Sternoperation*: $A^* \equiv \{x_1 x_2 x_3 \cdots x_k \mid k \geq 0 \wedge x_i \in A \forall i\}$

Reguläre Operationen führen reguläre Sprachen in reguläre Sprachen über (noch zu beweisen).

Probleme/Fragestellungen

- ▶ DEAs: Mehr auf analytischen als generativen Aspekt fokussiert.
- ▶ Einfachere Methoden als DEAs, um reguläre Sprache zu generieren?
- ▶ Charakterisierung regulärer Sprachen (Möglichkeiten und Grenzen)
- ▶ Zunächst: Alternativen zur Worterzeugung

Beispielsprache I: Gerade Zahlen $\in \mathbb{N}$

- ▶ Symbole (Ziffern): $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- ▶ Sprache $L \subseteq \Sigma^*$:

$$L = \{u | u \in \Sigma^* \wedge \exists n \in \mathbb{N} : u = 2n\}$$

Beispielsprache II: Primzahlen

- ▶ Symbole wie bei Beispiel I
- ▶ Sprache $L \subseteq \Sigma^*$:

$$L = \{n | n \in \Sigma^* \wedge n \text{ prim} \}$$

Fortschritt?

- ▶ Sprachcharakterisierung erleichtert
- ▶ Aber: Sprache von regulärem Automaten erkennbar?
- ▶ Ausweg: Explizite Grammatiken

Spracherzeugung

- ▶ *Grammatik*: Explizite Erzeugungsmöglichkeit für formale Sprache
- ▶ *Struktur*: Explizite Ableitung von Wörtern aus Regeln

Beispiel-Grammatik

$\langle \text{Satz} \rangle \rightarrow \langle \text{Subjekt} \rangle \langle \text{Prädikat} \rangle \langle \text{Objekt} \rangle$

$\langle \text{Subjekt} \rangle \rightarrow \langle \text{Artikel} \rangle \langle \text{Adjektiv} \rangle \langle \text{Substantiv} \rangle$

$\langle \text{Artikel} \rangle \rightarrow \text{Der} \mid \text{Die} \mid \text{Das}$

$\langle \text{Adjektiv} \rangle \rightarrow \text{kleine} \mid \text{eloxierte} \mid \text{flinke}$

$\langle \text{Substantiv} \rangle \rightarrow \text{Eisbär} \mid \text{Mond} \mid \text{Hypertricher}$

$\langle \text{Prädikat} \rangle \rightarrow \text{mag} \mid \text{isst} \mid \text{induziert}$

$\langle \text{Objekt} \rangle \rightarrow \text{Kekse} \mid \text{Schokolade} \mid \text{Raum-Zeit-Verschiebungen}$

Beispiel: Siehe Tafel.

Grammatik: Komponenten

- ▶ Nicht-Terminal-Symbole: $\langle \text{Satz} \rangle$, $\langle \text{Subjekt} \rangle$, ...
- ▶ Terminal-Symbole: Das, Schokolade, Raum-Zeit-Verschiebung, ...
- ▶ Produktionen: $\text{lhs} \rightarrow \text{rhs}$
- ▶ Startsymbol: $\langle \text{Satz} \rangle$

Algorithmus: Worterzeugung

1. Beginne mit Startsymbol
2. Ersetze alle Nicht-Terminal-Symbole gemäß den Produktionen
3. Wenn weiterhin Nicht-Terminal-Symbole vorhanden: Weiter bei 2
4. Gib Satz/Wort zurück

Beispiele: Siehe Tafel (arithmetische Ausdrücke, $a^n b^n c^n$)

Definition: Grammatik

Eine Grammatik G ist ein 4-Tupel (V, Σ, P, S) bestehend aus

- ▶ der endlichen Variablenmenge V (Nicht-Terminal-Symbole)
- ▶ dem endlichen Terminalalphabet Σ mit $V \cap \Sigma = \emptyset$
- ▶ der endlichen Menge $P \subseteq (V \cup \Sigma)^+ \setminus \Sigma^+ \times (V \cup \Sigma)^*$ von Produktionen (Regeln)
- ▶ dem Startsymbol $S \in V$

Spezifikation Produktion

- ▶ Hier: Allgemeinste Form von Produktionen
- ▶ Einschränkung bei Produktionen bestimmt maßgeblich Sprachklasse!

Anmerkungen Chomsky-Hierarchie

- ▶ *Phrasenstrukturgrammatiken (Typ 0)*: Jede Grammatik per Definitionem enthalten (*aber*: Sprachen, die nicht durch Grammatik zu beschreiben sind!) *Achtung*: Nomenklaturverwirrung in der Literatur.
- ▶ *Kontextsensitive Grammatiken (Typ 1)*: Anwendung einer Produktion führt niemals zu Verkürzung
- ▶ *Reguläre Grammatik (Typ 3)*: Rechtslineare Ableitung, da Produktionen nur nach rechts hin aufgebaut werden können

Chomsky-Hierarchie

Einteilung der Menge aller Sprachen in verschiedene (absteigend) mächtige Klassen.

Beschränkungen für alle Regeln $l \rightarrow r$:

- ▶ *Phrasenstrukturgrammatiken (Typ 0)*: Keine Regeleinschränkungen.
- ▶ *Kontextsensitive Grammatiken (Typ 1)*: Es gilt $|l| \leq |r|$.
- ▶ *Kontextfreie Grammatiken (Typ 2)*: $l \in V$.
- ▶ *Reguläre Grammatik (Typ 3)*: $r \in \Sigma \cup \Sigma V$.

Für Typ n -Grammatiken gelten jeweils die Beschränkungen von Typ $n - 1$ -Grammatiken.

Wenn G eine Typ n -Grammatik ist, bezeichnet man $\mathcal{L}(G)$ als Typ n -Sprache (dito: kontextfreie Sprache etc.)

Problem: ϵ -Regeln

- ▶ Wegen $|l| \leq |r|$ kann in kontextsensitiven (und damit kontextfreien und regulären) Sprachen keine Regel der Form $N \rightarrow \epsilon$ auftreten.
- ▶ Was tun, wenn $\epsilon \in \mathcal{L}(G)$ erwünscht?

ϵ -Sonderregel

$S \rightarrow \epsilon$ explizit zugelassen, wenn $\epsilon \in \mathcal{L}(G)$ erwünscht

Algorithmus zum Umschreiben von Grammatiken: Siehe Tafel

(Entscheidungs-)Probleme

- ▶ *Wortproblem*: Gegeben Wort $w \in \Sigma^*$ und Sprache $L \subseteq \Sigma^*$: Gilt $w \in L$ oder $w \notin L$?
- ▶ *Leerheitsproblem*: Enthält eine Sprache L kein Wort, d.h. gilt $L = \emptyset$?
- ▶ *Endlichkeitsproblem*: Besitzt L nur endlich viele Elemente, d.h. $|L| \neq \infty$?
- ▶ *Schnittproblem*: Gilt für zwei Sprachen L_1, L_2 , dass $L_1 \cap L_2 = \emptyset$?
- ▶ *Äquivalenzproblem*: Sind zwei Sprachen L_1, L_2 gleich, d.h. gilt $L_1 = L_2$?

Ableitbarkeit

Sei $l \in (V \cup \Sigma)^+ \setminus \Sigma^+$ und $r \in (V \cup \Sigma)^*$. Dann ist r aus l

- ▶ *direkt ableitbar*, wenn es eine Regel $(u, v) \in P$ und $\alpha, \beta \in (V \cup \Sigma)^*$ gibt, so dass

$$l = \alpha u \beta, r = \alpha v \beta.$$

Formal: $l \Rightarrow r$.

- ▶ *indirekt ableitbar*, wenn es durch endlich viele Ableitungsschritte aus l erzeugbar ist.

Formal: $l \xRightarrow{*} r$ (reflexive-transitive Hülle der Ableitungsrelation $\Rightarrow$, siehe Tafel)

Erzeugte Sprache $\mathcal{L}(G)$

$$\mathcal{L}(G) \equiv \{w \in \Sigma^* \mid S \xRightarrow{*} w\}$$

EBNF: Vereinfachungen

- ▶ $A \rightarrow \beta_1, A \rightarrow \beta_2, \dots, A \rightarrow \beta_n$ wird ersetzt durch $A \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$
- ▶ $A \rightarrow \alpha[\beta]\gamma$ steht für die beiden Regeln

$$A \rightarrow \alpha\gamma$$

$$A \rightarrow \alpha\beta\gamma$$

- ▶ $A \rightarrow \alpha\{\beta\}\gamma$ steht für die Regeln

$$A \rightarrow \alpha\gamma$$

$$A \rightarrow \alpha B\gamma$$

$$B \rightarrow \beta$$

$$B \rightarrow \beta B$$

Häufig bei der Definition von Programmiersprachen anzutreffen!

Scheme

```
...
<conditional> --> (if <test> <consequent> <alternate>)
<test> --> <expression>
<consequent> --> <expression>
<alternate> --> <expression> | <empty>

<assignment> --> (set! <variable> <expression>)

<derived expression> -->
  (cond <cond clause>+)
  | (cond <cond clause>* (else <sequence>))
  | (case <expression>
      <case clause>+)
  | (case <expression>
      <case clause>*
      (else <sequence>))
  | (and <test>*)
  | (or <test>*)
  | (let (<binding spec>*) <body>)
  | (let <variable> (<binding spec>*) <body>)
  | (let* (<binding spec>*) <body>)
  | (letrec (<binding spec>*) <body>)
  | (begin <sequence>)
  | (do (<iteration spec>*)
        (<test> <do result>)
        <command>*)
  | (delay <expression>)
  | <quasiquote>
...

```

Java

```
...
Block:
    { BlockStatements }

BlockStatements:
    { BlockStatement }

BlockStatement:
    LocalVariableDeclarationStatement
    ClassOrInterfaceDeclaration
    [Identifier :] Statement

LocalVariableDeclarationStatement:
    { VariableModifier } Type VariableDeclarators ;

Statement:
    Block
    ;
    Identifier : Statement
    StatementExpression ;
    if ParExpression Statement [else Statement]
...
    while ParExpression Statement
    do Statement while ParExpression ;
    for ( ForControl ) Statement
    try Block (Catches | [Catches] Finally)
    try ResourceSpecification Block [Catches] [Finally]

StatementExpression:

```

Siehe Tafel

Definition: Eindeutige Grammatik

Eine Grammatik G heißt eindeutig, wenn alle Ableitungen eines Wortes $w \in \mathcal{L}(G)$ immer zu genau einem Syntaxbaum führen. Eine nicht eindeutige Grammatik heißt mehrdeutig.

- ▶ Mehrdeutigkeit: Eigenschaft Grammatik, nicht notwendigerweise Sprache
- ▶ Mehrdeutige Grammatik: häufig in eine eindeutige Grammatik G' mit $\mathcal{L}(G) = \mathcal{L}(G')$ überführbar
- ▶ Es existieren Sprachen, die ausschließlich durch mehrdeutige Grammatiken erzeugt werden können (*inhärent mehrdeutige Sprachen*)

Beispiel: Siehe Tafel

Reguläre Sprachen

Grammatik $G(V, \Sigma, P, S)$ ist regulär, wenn für alle Regeln $l \rightarrow r$ gilt:

- ▶ $|l| \leq |r|$
- ▶ $l \in V$ und $r \in (\Sigma \cup \Sigma V)$

Anwendungen: Omnipräsent

- ▶ In praktisch allen Programmiersprachen »verbaut«: Perl, Grep, Java (`java.util.regex`), Go (`regexp`), D (`std.regex`), C# (`System.Text.RegularExpressions`), PHP (`preg_*`), ...
- ▶ Systemadministration, Konfigurationsdateien, Eingabechecker, Datenmanipulation, ...
- ▶ Achtung: »Reguläre Ausdrücke« häufig über obige Definition hinaus erweitert!

Beispiel: Siehe Tafel

Behauptung

- ▶ Reguläre Sprachen und endliche Automaten erkennen äquivalente Sprachklassen
- ▶ Konstruktion: Zustände des DEA als Nicht-Terminale auffassen, Grammatik entsprechend Übergängen gestalten

Beispiel: Muster »abc« in String finden (siehe Tafel)

Theorem: Endliche Automaten und reguläre Grammatiken

Zu jedem deterministischen endlichen Automaten M gibt es eine reguläre Grammatik G mit $\mathcal{L}(G) = \mathcal{L}(M)$.

Einschränkung

Obiges Theorem gilt nur für *eine* Richtung!

Beweis: Siehe Tafel

Status Quo: Reguläre Sprachen

- ▶ Reguläre Sprachen auf unterschiedliche Arten erzeugbar
- ▶ Erkannte Sprachen jedes endlichen Automaten durch reguläre Grammatik beschreibbar
- ▶ Noch zu zeigen: Vollständige Äquivalenz der Ansätze
- ▶ Mehr Verständnis über endliche Automaten erforderlich!

Theorem: Endliche Automaten und reguläre Grammatiken

Zu jedem deterministischen endlichen Automaten M gibt es eine reguläre Grammatik G mit $\mathcal{L}(G) = \mathcal{L}(M)$.

Umkehrung: G regulär $\Rightarrow \exists$ DEA für $\mathcal{L}(G)$

- ▶ DEA: Transitionsfunktion gibt Übergang eindeutig vor
- ▶ Grammatik: Mehrere Ableitungen aus einer Regel möglich
- ▶ *Nichtdeterministische Auswahl* aus einer Menge von Regeln
- ▶ 🖱️ DEAs auf Nichtdeterminismus erweitern!

DEA auf nicht-deterministische Übergänge erweitern

Determinismus

- ▶ $\delta : Q \times \Sigma \rightarrow Q$ *eindeutige* Abbildung von Tupel (Q, Σ) auf Folgezustand
- ▶ Eindeutige Rechenschritte!

Nichtdeterminismus

- ▶ Neue Übergangsfunktion:

$$\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$$

- ▶ Tupel Q, Σ wird auf *mehrere* potentielle Folgezustände abgebildet ($\mathcal{P}(Q)$: Potenzmenge der Zustände)
- ▶ Mehrdeutige Rechenschritte
- ▶ Unphysikalisches (aber nützliches) Modell!

Beispiele: Siehe Tafel

- ▶ Wörter $x \in \{0, 1\}^*$, die mit 00 enden oder gleich 0 sind
- ▶ Wörter $x \in \{0, 1\}^*$, deren k -letzte Ziffer 0 ist

Definition: Nicht-Deterministischer endlicher Automat (NEA)

Ein nicht-deterministischer endlicher Automat besteht aus folgenden Komponenten:

- ▶ Zuständen, in denen sich der Automat befinden kann
- ▶ Alphabet, aus dem die Eingabe generiert wird
- ▶ Eine Festlegung der *verschiedenen möglichen* Übergänge
- ▶ Einen *oder mehrere* Startzustände
- ▶ Einen oder mehrere Endzustände

Unterschied DEA/NEA

- ▶ Mehrere alternative Übergänge mit gleichem Symbol möglich
- ▶ Mehrere Startzustände möglich

Definition: Nicht-Deterministischer endlicher Automat (NEA)

Ein nicht-deterministischer endlicher Automat ist ein 5-Tupel $M = (Q, \Sigma, \delta, E, F)$, wobei:

- ▶ Zuständen, in denen sich der Automat befinden kann
- ▶ Alphabet, aus dem die Eingabe generiert wird
- ▶ Eine Festlegung der *verschiedenen möglichen* Übergänge
- ▶ Einen *oder mehrere* Startzustände
- ▶ Einen oder mehrere Endzustände

Unterschied DEA/NEA

- ▶ Mehrere alternative Übergänge mit gleichem Symbol möglich
- ▶ Mehrere Startzustände möglich

Definition: Nicht-Deterministischer endlicher Automat (NEA)

Ein nicht-deterministischer endlicher Automat ist ein 5-Tupel $M = (Q, \Sigma, \delta, E, F)$, wobei:

- ▶ Q eine endliche Zustandsmenge
- ▶ Alphabet, aus dem die Eingabe generiert wird
- ▶ Eine Festlegung der *verschiedenen möglichen* Übergänge
- ▶ Einen *oder mehrere* Startzustände
- ▶ Einen oder mehrere Endzustände

Unterschied DEA/NEA

- ▶ Mehrere alternative Übergänge mit gleichem Symbol möglich
- ▶ Mehrere Startzustände möglich

Definition: Nicht-Deterministischer endlicher Automat (NEA)

Ein nicht-deterministischer endlicher Automat ist ein 5-Tupel $M = (Q, \Sigma, \delta, E, F)$, wobei:

- ▶ Q eine endliche Zustandsmenge
- ▶ Σ ein endliches Alphabet,
- ▶ Eine Festlegung der *verschiedenen möglichen* Übergänge
- ▶ Einen *oder mehrere* Startzustände
- ▶ Einen oder mehrere Endzustände

Unterschied DEA/NEA

- ▶ Mehrere alternative Übergänge mit gleichem Symbol möglich
- ▶ Mehrere Startzustände möglich

Definition: Nicht-Deterministischer endlicher Automat (NEA)

Ein nicht-deterministischer endlicher Automat ist ein 5-Tupel $M = (Q, \Sigma, \delta, E, F)$, wobei:

- ▶ Q eine endliche Zustandsmenge
- ▶ Σ ein endliches Alphabet,
- ▶ $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$ die Übergangsfunktion,
- ▶ Einen *oder mehrere* Startzustände
- ▶ Einen oder mehrere Endzustände

Unterschied DEA/NEA

- ▶ Mehrere alternative Übergänge mit gleichem Symbol möglich
- ▶ Mehrere Startzustände möglich

Definition: Nicht-Deterministischer endlicher Automat (NEA)

Ein nicht-deterministischer endlicher Automat ist ein 5-Tupel $M = (Q, \Sigma, \delta, E, F)$, wobei:

- ▶ Q eine endliche Zustandsmenge
- ▶ Σ ein endliches Alphabet,
- ▶ $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$ die Übergangsfunktion,
- ▶ $E \subseteq Q$ die Menge der Startzustände
- ▶ Einen oder mehrere Endzustände

Unterschied DEA/NEA

- ▶ Mehrere alternative Übergänge mit gleichem Symbol möglich
- ▶ Mehrere Startzustände möglich

Definition: Nicht-Deterministischer endlicher Automat (NEA)

Ein nicht-deterministischer endlicher Automat ist ein 5-Tupel $M = (Q, \Sigma, \delta, E, F)$, wobei:

- ▶ Q eine endliche Zustandsmenge
- ▶ Σ ein endliches Alphabet,
- ▶ $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$ die Übergangsfunktion,
- ▶ $E \subseteq Q$ die Menge der Startzustände
- ▶ $F \subseteq Q$ die Menge der akzeptierenden Zustände ist.

Unterschied DEA/NEA

- ▶ Mehrere alternative Übergänge mit gleichem Symbol möglich
- ▶ Mehrere Startzustände möglich

Interpretation

- ▶ Massiv paralleler Computer (`fork/exec`)
- ▶ Rechnung unter Auflistung aller Möglichkeiten (*Achtung*: Keine Wahrscheinlichkeiten!)
- ▶ »Orakel«, das automatisch den (bzw. einen) korrekten Pfad wählt

Erweiterung: ϵ -NEA

- ▶ Modifikation: $\delta : Q \times \Sigma_* \rightarrow \mathcal{P}(Q)$
- ▶ $\Sigma_* \equiv \Sigma \cup \epsilon$
- ▶ »Leere« Übergänge möglich

Interpretation

- ▶ Massiv paralleler Computer (`fork/exec`)
- ▶ Rechnung unter Auflistung aller Möglichkeiten (*Achtung*: Keine Wahrscheinlichkeiten!)
- ▶ »Orakel«, das automatisch den (bzw. einen) korrekten Pfad wählt

Erweiterung: ϵ -NEA

- ▶ Modifikation: $\delta : Q \times \Sigma_* \rightarrow \mathcal{P}(Q)$
- ▶ $\Sigma_* \equiv \Sigma \cup \epsilon$
- ▶ »Leere« Übergänge möglich

Transitionsfunktion

- ▶ Übergang: $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$
- ▶ Zustand auf *Menge* von Zuständen abgebildet
- ▶ Verallgemeinerung auf $\hat{\delta} : \mathcal{P}(Q) \times \Sigma^* \rightarrow \mathcal{P}(Q)$ mittels

$$\begin{aligned}\hat{\delta}(Q', \epsilon) &= Q' \text{ für alle } Q' \subseteq Q \\ \hat{\delta}(Q', w_1 w_2 \cdots w_n) &= \bigcup_{q \in Q'} \hat{\delta}(\delta(q, w_1), w_2 \cdots w_n)\end{aligned}$$

- ▶ Illustration: Siehe Tafel

Akzeptierte Sprache eines NEA

$$\mathcal{L}(M) = \{w \in \Sigma^* \mid \hat{\delta}(E, w) \cap F \neq \emptyset\}$$

Illustration: Unterschied DEA/NEA

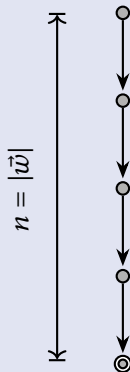
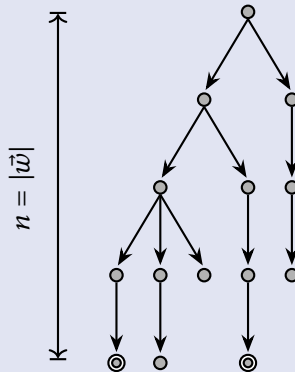


Illustration: Unterschied DEA/NEA



Beispiel Berechnungsbaum NEA: Siehe Tafel

Unterschiede DEA/NEA

- ▶ NEA muss nicht vollständig sein
 - ▶ Zustände erlaubt, die für ein oder mehrere $\sigma \in \Sigma$ keine ausgehende Kante besitzen
 - ▶ $\delta(q, \sigma) = \perp$ erlaubt
 - ▶ Übergangsfunktion *partiell* definiert
- ▶ Verarbeitung in Zweig kann stoppen ($\emptyset \in \mathcal{P}(Q)$)

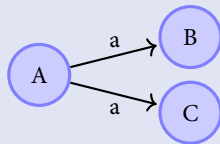
Umwandlung reguläre Grammatik $G \rightarrow$ NEA M

Regeln I

- ▶ Nicht-Terminalsymbole werden zu Zuständen
- ▶ $\langle A \rangle \rightarrow a \langle A \rangle \mid b \langle A \rangle$ wird zu



- ▶ $\langle A \rangle \rightarrow a \langle B \rangle \mid a \langle C \rangle$ wird zu



Regeln II

- ▶ Existiert Regel $\langle A \rangle \rightarrow \epsilon$, wird A zu Endzustand
- ▶ Regeln der Form $\langle A \rangle \rightarrow a$ werden zu

$$\langle A \rangle \rightarrow a \langle A \rangle'$$

$$\langle A \rangle' \rightarrow \epsilon$$

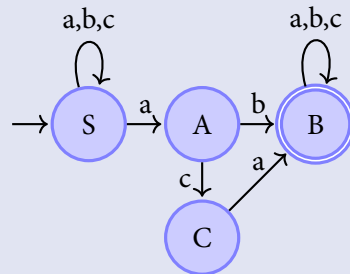
- ▶ Startsymbol S wird zu Anfangszustand

Formaler Beweis: Siehe Literatur

Grammatik

$$\langle S \rangle \rightarrow a\langle S \rangle \mid b\langle S \rangle \mid c\langle S \rangle \mid a\langle A \rangle$$
$$\langle A \rangle \rightarrow b\langle B \rangle \mid c\langle C \rangle$$
$$\langle B \rangle \rightarrow a\langle B \rangle \mid b\langle B \rangle \mid c\langle B \rangle \mid \epsilon$$
$$\langle C \rangle \rightarrow a\langle B \rangle$$

Äquivalenter NEA



Erkannte/Erzeugte Sprache

Alle Wörter $w \in \{a, b, c\}^*$, die »ab« oder »aca« enthalten

Grammatik

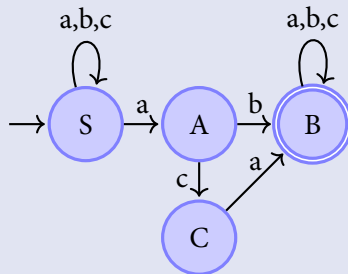
$$\langle S \rangle \rightarrow a\langle S \rangle \mid b\langle S \rangle \mid c\langle S \rangle \mid a\langle A \rangle$$

$$\langle A \rangle \rightarrow b\langle B \rangle \mid c\langle C \rangle$$

$$\langle B \rangle \rightarrow a\langle B \rangle \mid b\langle B \rangle \mid c\langle B \rangle \mid \epsilon$$

$$\langle C \rangle \rightarrow a\langle B \rangle$$

Äquivalenter NEA



Erkannte/Erzeugte Sprache

Alle Wörter $w \in \{a, b, c\}^*$, die »ab« oder »aca« enthalten

Status Quo

- ▶ $\text{DEA} \rightarrow \text{reguläre Grammatik}$
- ▶ $\text{DEA} \rightarrow \text{NEA}$ (trivial)
- ▶ $\text{Reguläre Grammatik} \rightarrow \text{NEA}$
- ▶ **Noch zu zeigen:** $\text{NEA} \rightarrow \text{DEA}$

Illustration: Siehe Tafel

NEAs und DEAs

- ▶ Intuition: Nichtdeterminismus ist mächtiger als Determinismus
- ▶ *Aber*: Satz von Rabin und Scott garantiert Äquivalenz von DEAs und NEAs

Satz von Rabin und Scott

Jede von einem NEA akzeptierbare Sprache L ist auch von einem DEA akzeptierbar.

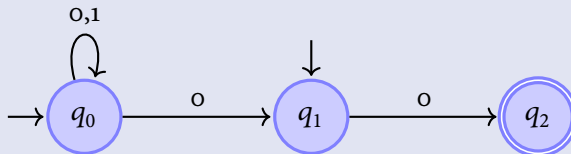
NEAs und DEAs

- ▶ Intuition: Nichtdeterminismus ist mächtiger als Determinismus
- ▶ *Aber*: Satz von Rabin und Scott garantiert Äquivalenz von DEAs und NEAs

Satz von Rabin und Scott

Jede von einem NEA akzeptierbare Sprache L ist auch von einem DEA akzeptierbar.

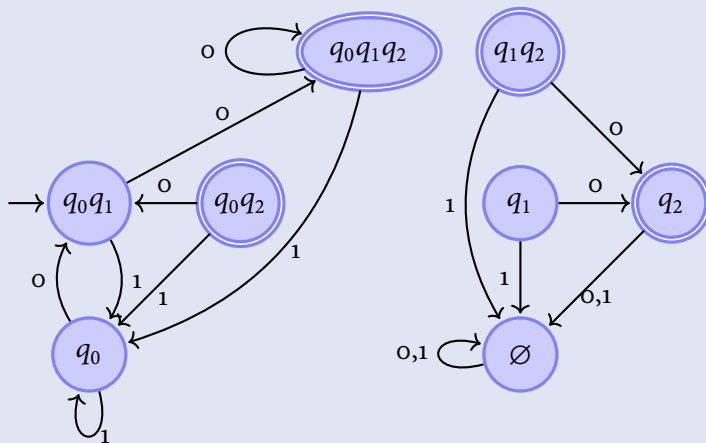
NEA: $x \in \{0, 1\}^*$, $x = 0$ oder x endet auf 00



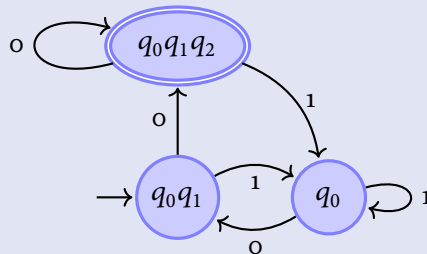
Rabin/Scott: Grundidee

- ▶ Alle möglichen Zustandsmengen als *einen* Zustand auffassen
- ▶ Transitionen zwischen Zustandsmengen angeben.
- ▶ Neue Endzustände: Menge enthält mindestens einen alten Endzustand

Äquivalenter DEA



Äquivalenter DEA ohne nicht-erreichbare Zustände



Bisheriger Stand

- ▶ Konvertierungsalgorithmus $NEA \rightarrow DEA$ vorhanden
- ▶ Rückrichtung trivial!
- ▶ Keine Effizienzverschlechterung ($|w| = n \Leftrightarrow n$ Bearbeitungsschritte)
- ▶ Potentiell exponentieller Zuwachs an Zuständen

Äquivalenz DEA/NEA: Formale Darstellung

Sei $M = (Q, \Sigma, \delta, E, F)$ ein NEA. Dann existiert ein DEA $M' = (Q', \Sigma, \delta', q'_0, F')$ mit

- ▶ $Q' \equiv \mathcal{P}(Q)$ Potenzmenge von Q
- ▶ $\delta'(Z, a) \equiv \bigcup_{q \in Z} \delta(q, a)$ mit $Z \in Q' = \mathcal{P}(Q)$
- ▶ $q'_0 = E$ (Interpretation als kombinierter Einzelzustand!)
- ▶ $F' = \{Z \subseteq Q' \mid Z \cap F \neq \emptyset\}$

der die gleiche Sprache wie M akzeptiert, also

$$\mathcal{L}(M) = \mathcal{L}(M').$$

Beweis der Äquivalenz: Siehe Tafel

Status Quo II: Ringschluss vollendet!

- ▶ $\text{DEA} \rightarrow \text{reguläre Grammatik}$
- ▶ $\text{DEA} \rightarrow \text{NEA}$ (trivial)
- ▶ $\text{Reguläre Grammatik} \rightarrow \text{NEA}$
- ▶ $\text{NEA} \rightarrow \text{DEA}$

Exponentieller Zustandszuwachs

- ▶ $\exists$ NEAs, deren Konvertierung in DEAs exponentiell viele Zustände ($2^{|Q|}$) benötigt
- ▶ Beispiel: $L = \{w \in \{0, 1\}^n \mid k\text{-letztes Zeichen ist } 0\}$ (Beweis: Siehe Tafel)

Schlussfolgerungen

- ▶ Potenzmengenkonstruktion $\text{NEA} \rightarrow \text{DEA}$ kann optimal sein
- ▶ NEAs können Sprachen im Allgemeinen *exponentiell kompakter* als DEAs darstellen
- ▶ Worst Case tritt so gut wie nie auf

Exponentieller Zustandszuwachs

- ▶ $\exists$ NEAs, deren Konvertierung in DEAs exponentiell viele Zustände ($2^{|Q|}$) benötigt
- ▶ Beispiel: $L = \{w \in \{0, 1\}^n \mid k\text{-letztes Zeichen ist } 0\}$ (Beweis: Siehe Tafel)

Schlussfolgerungen

- ▶ Potenzmengenkonstruktion NEA $\rightarrow$ DEA kann optimal sein
- ▶ NEAs können Sprachen im Allgemeinen *exponentiell kompakter* als DEAs darstellen
- ▶ Worst Case tritt so gut wie nie auf

Beispiele für reguläre Ausdrücke

- ▶ Postleitzahl & Ort: $(0-9)^+_-(A-Z|a-z)^+$
 - ▶ $()$: Gruppierung
 - ▶ $A-Z$: Zeichengruppe (entspricht $A | B | C | \dots | X$)
 - ▶ $\xi_{\{m,n\}}$: Zwischen m und n Wiederholungen des regulären Ausdrucks ξ
- ▶ Beispielsprache aller Wörter $x \in \{0, 1\}^*$, die mit 00 enden oder gleich 0 sind: $0|((0|1)^*00)$
- ▶ eMail-Adressen matchen (*vereinfachter* Ansatz): $(A-Z|a-z|0-9|.|_|-)^+@(A-Z|a-z|0-9|.|_|-)^+.(A-Z|a-z)\{2,6\}$

Software-Bibliotheken

http://en.wikipedia.org/wiki/Comparison_of_regular_expression_engines

Beispiele für reguläre Ausdrücke

- ▶ Postleitzahl & Ort: $(0-9)^+_-(A-Z|a-z)^+$
 - ▶ $()$: Gruppierung
 - ▶ $A-Z$: Zeichengruppe (entspricht $A | B | C | \dots | X$)
 - ▶ $\xi\{m,n\}$: Zwischen m und n Wiederholungen des regulären Ausdrucks ξ
- ▶ Beispielsprache aller Wörter $x \in \{0,1\}^*$, die mit 00 enden oder gleich 0 sind: $0|((0|1)^*00)$
- ▶ eMail-Adressen matchen (*vereinfachter* Ansatz): $(A-Z|a-z|0-9|.|_|-)^+@(A-Z|a-z|0-9|.|_|-)^+.(A-Z|a-z)\{2,6\}$

Software-Bibliotheken

`http://en.wikipedia.org/wiki/Comparison\_of\_regular\_expression\_engines`

Reguläre Ausdrücke: Syntax

Sei Σ ein beliebiges Alphabet. Dann wird die Menge aller *regulären Ausdrücke* R_Σ gegeben durch:

- ▶ Die leere Menge und das leere Wort
- ▶ Ein beliebiges Symbol aus Σ
- ▶ Die Konkatenation zweier regulärer Ausdrücke
- ▶ Die Auswahl zwischen zwei regulären Ausdrücken
- ▶ Die Sternoperation angewendet auf einen regulären Ausdruck
- ▶ Gruppierung regulärer Ausdrücke

Reguläre Ausdrücke: Syntax

Sei Σ ein beliebiges Alphabet. Dann wird die Menge aller *regulären Ausdrücke* R_Σ induktiv gebildet durch:

- ▶ $\emptyset, \epsilon \in R_\Sigma$
- ▶ Ein beliebiges Symbol aus Σ
- ▶ Die Konkatenation zweier regulärer Ausdrücke
- ▶ Die Auswahl zwischen zwei regulären Ausdrücken
- ▶ Die Sternoperation angewendet auf einen regulären Ausdruck
- ▶ Gruppierung regulärer Ausdrücke

Reguläre Ausdrücke: Syntax

Sei Σ ein beliebiges Alphabet. Dann wird die Menge aller *regulären Ausdrücke* R_Σ induktiv gebildet durch:

- ▶ $\emptyset, \epsilon \in R_\Sigma$
- ▶ $\Sigma \in R_\Sigma$
- ▶ Die Konkatenation zweier regulärer Ausdrücke
- ▶ Die Auswahl zwischen zwei regulären Ausdrücken
- ▶ Die Sternoperation angewendet auf einen regulären Ausdruck
- ▶ Gruppierung regulärer Ausdrücke

Reguläre Ausdrücke: Syntax

Sei Σ ein beliebiges Alphabet. Dann wird die Menge aller *regulären Ausdrücke* R_Σ induktiv gebildet durch:

- ▶ $\emptyset, \epsilon \in R_\Sigma$
- ▶ $\Sigma \in R_\Sigma$
- ▶ $r_1, r_2 \in R_\Sigma \Rightarrow r_1 r_2 \in R_\Sigma$
- ▶ Die Auswahl zwischen zwei regulären Ausdrücken
- ▶ Die Sternoperation angewendet auf einen regulären Ausdruck
- ▶ Gruppierung regulärer Ausdrücke

Reguläre Ausdrücke: Syntax

Sei Σ ein beliebiges Alphabet. Dann wird die Menge aller *regulären Ausdrücke* R_Σ induktiv gebildet durch:

- ▶ $\emptyset, \epsilon \in R_\Sigma$
- ▶ $\Sigma \in R_\Sigma$
- ▶ $r_1, r_2 \in R_\Sigma \Rightarrow r_1 r_2 \in R_\Sigma$
- ▶ $r_1, r_2 \in R_\Sigma \Rightarrow (r_1 \mid r_2) \in R_\Sigma$
- ▶ Die Sternoperation angewendet auf einen regulären Ausdruck
- ▶ Gruppierung regulärer Ausdrücke

Reguläre Ausdrücke: Syntax

Sei Σ ein beliebiges Alphabet. Dann wird die Menge aller *regulären Ausdrücke* R_Σ induktiv gebildet durch:

- ▶ $\emptyset, \epsilon \in R_\Sigma$
- ▶ $\Sigma \in R_\Sigma$
- ▶ $r_1, r_2 \in R_\Sigma \Rightarrow r_1 r_2 \in R_\Sigma$
- ▶ $r_1, r_2 \in R_\Sigma \Rightarrow (r_1 \mid r_2) \in R_\Sigma$
- ▶ $r \in R_\Sigma \Rightarrow r^* \in R_\Sigma$
- ▶ Gruppierung regulärer Ausdrücke

Reguläre Ausdrücke: Syntax

Sei Σ ein beliebiges Alphabet. Dann wird die Menge aller *regulären Ausdrücke* R_Σ induktiv gebildet durch:

- ▶ $\emptyset, \epsilon \in R_\Sigma$
- ▶ $\Sigma \in R_\Sigma$
- ▶ $r_1, r_2 \in R_\Sigma \Rightarrow r_1 r_2 \in R_\Sigma$
- ▶ $r_1, r_2 \in R_\Sigma \Rightarrow (r_1 \mid r_2) \in R_\Sigma$
- ▶ $r \in R_\Sigma \Rightarrow r^* \in R_\Sigma$
- ▶ $r \in R_\Sigma \Rightarrow (r) \in R_\Sigma$

Reguläre Ausdrücke: Syntax

Sei Σ ein beliebiges Alphabet. Dann wird die Menge aller *regulären Ausdrücke* R_Σ induktiv gebildet durch:

- ▶ $\emptyset, \epsilon \in R_\Sigma$
- ▶ $\Sigma \in R_\Sigma$
- ▶ $r_1, r_2 \in R_\Sigma \Rightarrow r_1 r_2 \in R_\Sigma$
- ▶ $r_1, r_2 \in R_\Sigma \Rightarrow (r_1 \mid r_2) \in R_\Sigma$
- ▶ $r \in R_\Sigma \Rightarrow r^* \in R_\Sigma$
- ▶ $r \in R_\Sigma \Rightarrow (r) \in R_\Sigma$

Semantik

- ▶ $\mathcal{L}(\emptyset) = \emptyset, \mathcal{L}(\epsilon) = \{\epsilon\}$
- ▶ $\mathcal{L}(a \in \Sigma) = \{a\}$
- ▶ $\mathcal{L}(rs) = \mathcal{L}(r)\mathcal{L}(s)$
- ▶ $\mathcal{L}(r \mid s) = \mathcal{L}(r) \cup \mathcal{L}(s)$
- ▶ $\mathcal{L}(r^*) = \mathcal{L}(r)^*$
- ▶ $\mathcal{L}((r)) = \mathcal{L}(r)$

Endliche Sprachen

Alle endlichen Sprachen sind durch reguläre Ausdrücke beschreibbar:

- ▶ Sei $A = \{\vec{w}_1, \vec{w}_2, \dots, \vec{w}_n\}$ *endliche* Menge der *endlichen* Wörter
- ▶ $(\vec{w}_1 | \vec{w}_2 | \vec{w}_3 | \dots | \vec{w}_n)$ regulärer Ausdruck für A

Satz von Kleene

Die Menge der durch reguläre Ausdrücke beschreibbaren Sprachen ist genau die Menge der regulären Sprachen.

Beweis: Hinrichtung (über regulärer Ausdruck $\rightarrow$ NEA): siehe Tafel; Rückrichtung: Siehe Literatur (außer Hoffmann)

Endliche Sprachen

Alle endlichen Sprachen sind durch reguläre Ausdrücke beschreibbar:

- ▶ Sei $A = \{\vec{w}_1, \vec{w}_2, \dots, \vec{w}_n\}$ *endliche* Menge der *endlichen* Wörter
- ▶ $(\vec{w}_1|\vec{w}_2|\vec{w}_3|\dots|\vec{w}_n)$ regulärer Ausdruck für A

Satz von Kleene

Die Menge der durch reguläre Ausdrücke beschreibbaren Sprachen ist genau die Menge der regulären Sprachen.

Beweis: Hinrichtung (über regulärer Ausdruck $\rightarrow$ NEA): siehe Tafel; Rückrichtung: Siehe Literatur (außer Hoffmann)

Status Quo III

- ▶ $DEA \rightarrow \text{reguläre Grammatik}$
- ▶ $DEA \rightarrow NEA$ (trivial)
- ▶ $\text{Reguläre Grammatik} \rightarrow NEA$
- ▶ $\text{Reguläre Ausdrücke} \leftrightarrow \text{Reguläre Grammatik}$
- ▶ $NEA \rightarrow DEA$

Alt: Vorlage

```
\section{Einleitung}  
In dieser \textbf{Einleitung},  
die \textit{sehr \textbf{schnell}} neu} ...
```

Neu: Konvertiert

```
<abschnitt>Einleitung</abschnitt>  
In dieser <fett>Einleitung</fett>,  
die <kursiv>sehr <fett>schnell</fett> neu</kursiv> ...
```

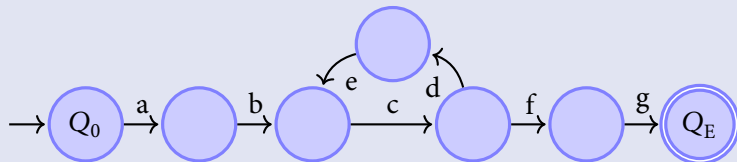
RegExp-Versuche

```
\{ (.*) \}  
\{ ([^\{\}]*) \}  
\{ ([^{}]*\{[^{}]*\}[^{}]*) \}
```

Generelles Problem

- ▶ Ist eine Grammatik G regulär?
- ▶ Kann eine Sprache L mit regulären Ausdrücken beschrieben werden?

DEA mit 7 Zuständen



Beispiele

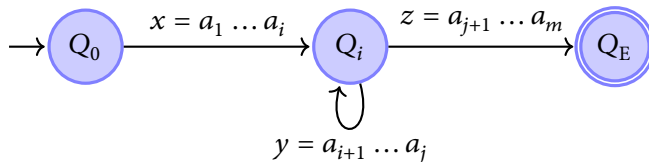
- ▶ $abcfg \Rightarrow$ Kein Zyklus ($|w| = 5$)
- ▶ $abccdefg \Rightarrow$ Einfacher Zyklus ($|w| = 8$)
- ▶ $abcedcdcfg \Rightarrow$ Zweifacher Zyklus ($|w| = 11$)

Zyklen

Zustandsanzahl: $|Q| = n \Rightarrow$ Zyklus für Wörter w mit $|w| \geq n$

Komponenten

1. L regulär, DEA M mit $L = \mathcal{L}(M)$.
2. M habe $|Q| = n$ Zustände.
3. $w \in L$ mit $|w| \geq n$, $w = a_1 a_2 \dots a_m$
4. Q_i Zustand von A nach Lesen der ersten i Symbole von w .
5. $|w| = m \geq n = |Q|$
6. Es gibt Zahlen $i, j : 0 \leq i < j \leq n : Q_i = Q_j$



Pumping-Lemma (Schleifenlemma, Bar-Hillel)

Sei L eine reguläre Sprache. Dann gibt es eine Zahl n , so dass sich alle Wörter $w \in L$, $|w| \geq n$ zerlegen lassen in $w = xyz$, so dass

1. $|y| \geq 1$
2. $|xy| \leq n$
3. Für alle $i \in \mathbb{N}_0$: $xy^iz \in L$

Beweis: Siehe Tafel

Dyck-Sprache D_1

Ist $L \equiv \{a^n b^n \mid n \in \mathbb{N}^+\}$ eine reguläre Sprache?

Beispiele: ab, aabb, aaabbb, aaaabbbb, ...

Beweis: Siehe Tafel (Widerspruchsbeweis)

Quadratzahlen

$L \equiv \{a^n \mid \exists m \in \mathbb{N} : n = m^2\}$. Beispiele: a, aaaa, aaaaaaaaaa, ...

Beweis

- ▶ Annahme: L ist regulär
- ▶ Sei n die Konstante des Pumping-Lemmas
- ▶ Wähle $z = a^{n^2} \in L \Rightarrow |z| > n$
- ▶ Pumping-Lemma:
 - ▶ $z = uvw$, $1 \leq |v| \leq |uv| \leq n$
 - ▶ $uv^2w \in L$
- ▶ $n^2 = |z| = |uvw| < |uv^2w| \leq n^2 + n < n^2 + 2n + 1 = (n+1)^2$
- ▶ Daraus folgt: $|uv^2w|$ ist eine Quadratzahl zwischen n^2 und $(n+1)^2$
- ▶ Widerspruch!

Äquivalenzrelation für Sprache L

Es gilt $xR_L y$ genau dann, wenn für alle Wörter $z \in \Sigma^*$ gilt

$$xz \in L \Leftrightarrow yz \in L.$$

- ▶ Anfügen beliebiger Teilstrings: Keine Änderung Mitgliedschaft in Äquivalenzklasse
- ▶ $z = \epsilon : x \in L \Leftrightarrow y \in L$
- ▶ Anzahl erzeugter Äquivalenzklassen: *Index* von R_L

Satz von Myhill und Nerode

Eine Sprache L ist genau dann regulär, wenn der Index von R_L endlich ist.

Beweis: Siehe Tafel.

Beispiel

$$L \equiv \{x \in \{0, 1\}^* \mid x \text{ endet mit } 00\}$$

Äquivalenzklassen

- ▶ $[\epsilon] = \{x \mid x \text{ endet nicht mit } 0\}$
- ▶ $[0] = \{x \mid x \text{ endet mit } 0, \text{ aber nicht mit } 00\}$
- ▶ $[00] = \{x \mid x \text{ endet mit } 00\}$

Beispiel Automat: Siehe Tafel

Satz: Minimalautomat

Äquivalenzklassenautomat M_0 ist minimal, d.h. besitzt die kleinstmögliche Anzahl von Zuständen

Folgerungen aus vorhergehendem Beweis:

- ▶ Sei M ein Automat mit $\mathcal{L}(M) = L$.
- ▶ R_M : Verfeinerung von R_L sein, da ansonsten Klassen fehlen: $R_M \subseteq R_L = R_{M_0}$
- ▶ Anzahl Zustände von M größer oder gleich Anzahl Zustände von M_0
- ▶ Zustandszahl von M und L gleich: Identisch bis auf *Isomorphie*, d.h. Umbenennung der Zustände

Automat ist *nicht* minimal, wenn es zwei Zustände q, q' gibt:

$$\hat{\delta}(q, x) \in F \Leftrightarrow \hat{\delta}(q', x) \in F$$

$\Rightarrow (q, q')$ können verschmolzen werden. Algorithmus:

1. Tabelle aller Zustandspaare (q, q') mit $q \neq q'$ aufstellen
2. Alle Paare (q, q') markieren mit $q \in F$ und $q' \notin F$ oder umgekehrt
3. Für alle unmarkierten Paare (q, q') und alle $a \in \Sigma$: Testen, ob

$$(\delta(q, a), \delta(q', a))$$

bereits markiert ist. Wenn ja, *Ausgangspaar* (q, q') markieren

4. Schritt 3 solange wiederholen, bis Tabelle invariant
5. Alle nicht-markierten Paare zu je einem Zustand verschmelzen

Beispiel: Siehe Tafel

Abschlusseigenschaften regulärer Sprachen

Die regulären Sprachen sind abgeschlossen unter

- ▶ Vereinigung
- ▶ Schnitt
- ▶ Komplement
- ▶ Produkt
- ▶ Stern

Beweis: Siehe Tafel

Neuen Beweistechnik: Nicht-Regularität einer Sprache

- ▶ Zwei Sprachen L_1 (regulär), L_2 (unbekannt)
- ▶ $L_3 = L_1 \circ L_2$ (für geeignete Operation $\circ$) berechnen
- ▶ L_3 nicht regulär $\rightarrow L_2$ nicht regulär

Abschlusseigenschaften regulärer Sprachen

Die regulären Sprachen sind abgeschlossen unter

- ▶ Vereinigung
- ▶ Schnitt
- ▶ Komplement
- ▶ Produkt
- ▶ Stern

Beweis: Siehe Tafel

Neuen Beweistechnik: Nicht-Regularität einer Sprache

- ▶ Zwei Sprachen L_1 (regulär), L_2 (unbekannt)
- ▶ $L_3 = L_1 \circ L_2$ (für geeignete Operation $\circ$) berechnen
- ▶ L_3 nicht regulär $\rightarrow L_2$ nicht regulär

Entscheidbare Eigenschaften regulärer Sprachen

- ▶ *Wortproblem*: Gegeben Wort $w \in \Sigma^*$ und Sprache $L \in \Sigma^*$: Gilt $w \in L$ oder $w \notin L$?
- ▶ *Leerheitsproblem*: Enthält eine Sprache L kein Wort, d.h. gilt $L = \emptyset$?
- ▶ *Endlichkeitsproblem*: Besitzt L nur endlich viele Elemente, d.h. $|L| \neq \infty$?
- ▶ *Schnittproblem*: Gilt für zwei Sprachen L_1, L_2 , dass $L_1 \cap L_2 = \emptyset$?
- ▶ *Äquivalenzproblem*: Sind zwei Sprachen L_1, L_2 gleich, d.h. gilt $L_1 = L_2$?

Entscheidbare Eigenschaften regulärer Sprachen

- ▶ *Wortproblem*: ✓ (DEA verwenden)
- ▶ *Leerheitsproblem*: ✓ (DEA: Kein Pfad von Start- zu (einem) Endzustand)
- ▶ *Endlichkeitsproblem*: ✓ ($|\mathcal{L}(M)| = \infty \Leftrightarrow \exists$ Zyklus, der von q_0 erreichbar ist und in F endet.)
- ▶ *Schnittproblem*: ✓ (Konstruktion Automat, siehe Tafel)
- ▶ *Äquivalenzproblem*: ✓ (Minimalautomaten (bis auf Isomorphie) identisch)

Kontextfreie Sprachen

Erinnerung I: Kontextfreie Sprache (Typ 2)

- ▶ Gegeben durch Grammatik $G = (V, \Sigma, P, S)$
- ▶ Beschränkungen für alle Regeln $l \rightarrow r \in P$:
 - ▶ $|l| \leq |r|$
 - ▶ $l \in V$

Erinnerung II

Eine kontextfreie Grammatik G kann ϵ -frei gemacht werden

Beispiel 1: Palindrome

- ▶ Alle Wörter, die von vorne und hinten gelesen identisch sind, also

$$L_P \equiv \{w \in \Sigma^* \mid w = v0v^R \cup w = v1v^R\}$$

- ▶ Kontextfreie Grammatik für Palindrome über $\Sigma = \{0, 1\}$:

$$\langle S \rangle \rightarrow 0 \mid 1 \mid 0\langle S \rangle 0 \mid 1\langle S \rangle 1$$

Korollar

$\mathcal{L}_3 \subset \mathcal{L}_2$, d.h. reguläre Sprachen sind eine *echte* Teilmenge der kontextfreien Sprachen.
Beweis:

- ▶ Nach Konstruktion: $\forall L \in \mathcal{L}_3 : L \in \mathcal{L}_2$
- ▶ Binäre Palindrome sind *nicht regulär*, d.h. $L_P \notin \mathcal{L}_3$
- ▶ Binäre Palindrome sind *kontextfrei*, d.h. $L_P \in \mathcal{L}_2$
- ▶ Ergo $\mathcal{L}_3 \subset \mathcal{L}_2$

Beispiel 1: Palindrome

- ▶ Alle Wörter, die von vorne und hinten gelesen identisch sind, also

$$L_P \equiv \{w \in \Sigma^* \mid w = v0v^R \cup w = v1v^R\}$$

- ▶ Kontextfreie Grammatik für Palindrome über $\Sigma = \{0, 1\}$:

$$\langle S \rangle \rightarrow 0 \mid 1 \mid 0\langle S \rangle 0 \mid 1\langle S \rangle 1$$

Korollar

$\mathcal{L}_3 \subset \mathcal{L}_2$, d.h. reguläre Sprachen sind eine *echte* Teilmenge der kontextfreien Sprachen.
Beweis:

- ▶ Nach Konstruktion: $\forall L \in \mathcal{L}_3 : L \in \mathcal{L}_2$
- ▶ Binäre Palindrome sind *nicht regulär*, d.h. $L_P \notin \mathcal{L}_3$
- ▶ Binäre Palindrome sind *kontextfrei*, d.h. $L_P \in \mathcal{L}_2$
- ▶ Ergo $\mathcal{L}_3 \subset \mathcal{L}_2$

Beispiel 2: Dyck-Sprache D_1

- ▶ Korrekt geklammerte Ausdrücke: $()$, $(())$, $((()))$, ...
- ▶ Grammatik:

$$\langle S \rangle \rightarrow (\langle S \rangle) \mid \epsilon$$

Definition: Chomsky-Normalform

Eine Grammatik G mit $\epsilon \notin \mathcal{L}(G)$ ist in *Chomsky-Normalform* gegeben, wenn alle Regeln eine der beiden Formen

$$A \rightarrow BC$$

$$A \rightarrow a$$

haben, wobei $A, B, C \in V$ und $a \in \Sigma$.

Satz

Zu jeder kontextfreien Grammatik G mit $\epsilon \notin \mathcal{L}(G)$ gibt es eine Grammatik G' in Chomsky-Normalform mit $\mathcal{L}(G) = \mathcal{L}(G')$.

Sei $G = (V, \Sigma, P, S)$ eine ϵ -freie kfG

Konstruktion, Schritt 1

Umformung in $G' = (V', \Sigma, P', S)$ mit zwei Regelformen

1. $A \rightarrow a$ mit $A \in V', a \in \Sigma$
2. $A \rightarrow x$ mit $A \in V', x \in (V' \cup \Sigma)^+, |x| \geq 2$

Konvertierungsalgorithmus:

- ▶ *Zyklen eliminieren:* Gibt es $B_1, B_2, \dots, B_k$ mit $B_1 \rightarrow B_2 \rightarrow B_3 \rightarrow \dots \rightarrow B_k \rightarrow B_1$, dann ersetze alle B_i durch B
- ▶ *Variablen sortieren & transformieren:* $V' = \{A_1, \dots, A_n\}$ wird so numeriert, dass $A_i \rightarrow A_j \Rightarrow i < j$ gilt.

Dann Elimination dieser Regeln:

- ▶ Iteriere von $k = n - 1$ bis $k = 1$
- ▶ Für alle $A_k \rightarrow A_{k'}$ mit $k' > k$ und $A_{k'} \rightarrow x_1 | x_2 | x_3 | \dots | x_m$: Neue Regel einführen: $A_k \rightarrow x_1 | x_2 | \dots | x_m$

Sei $G = (V, \Sigma, P, S)$ eine kfG aus Schritt 1

Konstruktion, Schritt 2

1. Neue Regel $B \rightarrow a$ für jedes Terminalzeichen a hinzufügen: $\forall a \in \Sigma: V_{\text{neu}} = V \cup \{B\}, P_{\text{neu}} = P \cup \{(B \rightarrow a)\}$
2. Alle a auf der rechten Regelseite durch B ersetzen (außer Regel bereits in Form $A \rightarrow a$). Ergibt Regeln der Form:

$$A \rightarrow a \text{ oder } A \rightarrow B_1 B_2 \dots B_k, k \geq 2$$

3. Für alle Regeln $A \rightarrow B_1 B_2 \dots B_k$ mit $k \geq 3$: Neue Variablen $C_1, C_2, \dots, C_{k-2}$ einführen mit

$$\begin{aligned} A &\rightarrow B_1 C_1, C_1 \rightarrow B_2 C_2, \\ C_2 &\rightarrow B_3 C_3, \dots, C_{k-2} \rightarrow B_{k-1} B_k \end{aligned}$$

Beispiel

$$S \rightarrow AB$$

$$S \rightarrow ABA$$

$$A \rightarrow aA$$

$$A \rightarrow a$$

$$B \rightarrow Bb$$

$$B \rightarrow b \mid \epsilon$$

Umwandlung: Siehe Tafel

Beispiel 2

$$S \rightarrow ab \mid aSb$$

Greibach-Normalform

Eine ϵ -freie kontextfreie Grammatik G ist in Greibach-Normalform, wenn alle Regeln die Form

$$\langle A \rangle \rightarrow a \langle B \rangle_1 \langle B \rangle_2 \langle B \rangle_3 \dots \langle B \rangle_k, k \geq 0$$

haben ($\langle B \rangle_i \in V, a \in \Sigma$)

- ▶ Zu jeder ϵ -freien kontextfreien Grammatik G gibt es eine Grammatik G' in Greibach-Normalform mit $\mathcal{L}(G') = \mathcal{L}(G)$ (Beweis: Siehe Literatur)
- ▶ Reguläre Sprachen sind in Greibach-Normalform mit $k = 0$ oder $k = 1$.

Probleme/Fragestellungen

- ▶ Ist eine gegebene Sprache kontextfrei?
- ▶ $\Rightarrow$ Aussage über Struktur von kfS notwendig.

Klammerpaare

$S \rightarrow SS \mid (S) \mid ()$

Beispiele: $()$, $(())$, $((()))$, $((()))$, ...

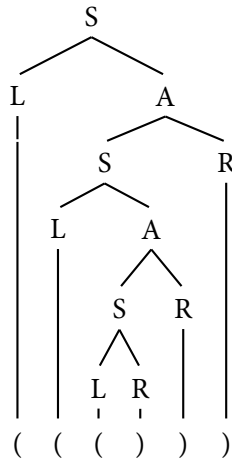
CNF-Darstellung

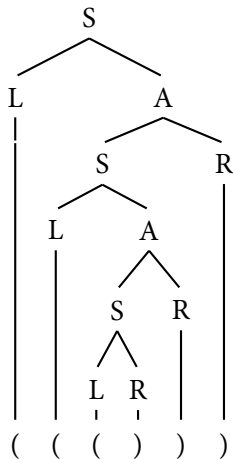
$S \rightarrow SS \mid LA \mid LR$

$L \rightarrow ($

$R \rightarrow)$

$A \rightarrow SR$

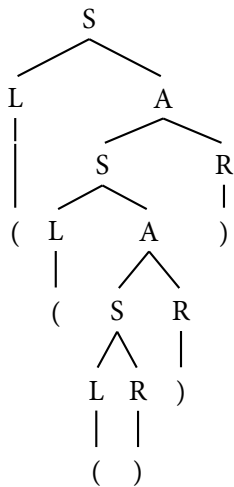




Wort

$$((()))$$

Teilbaum

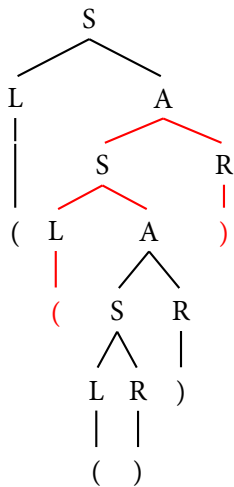


Wort

$((((()))$

Teilbaum



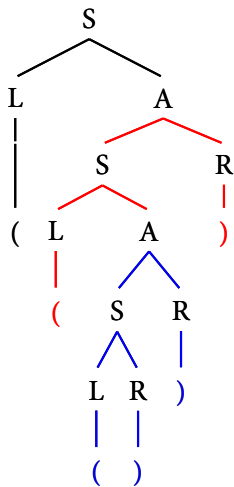


Wort

$(((())))$

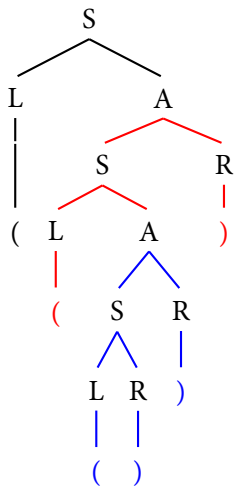
Teilbaum





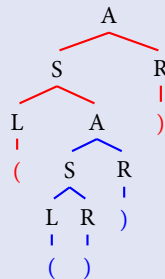
Wort

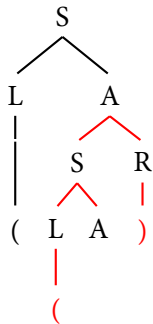
Teilbaum



Wort

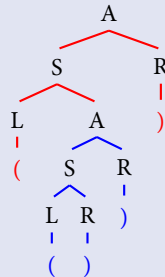
Teilbaum

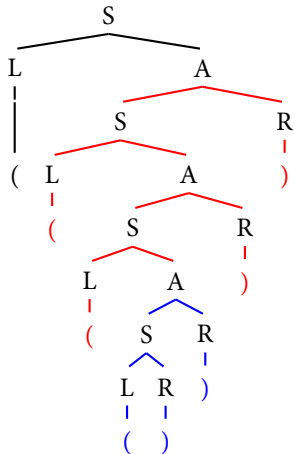




Wort

Teilbaum

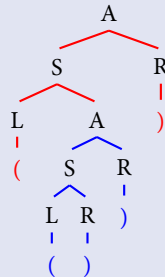


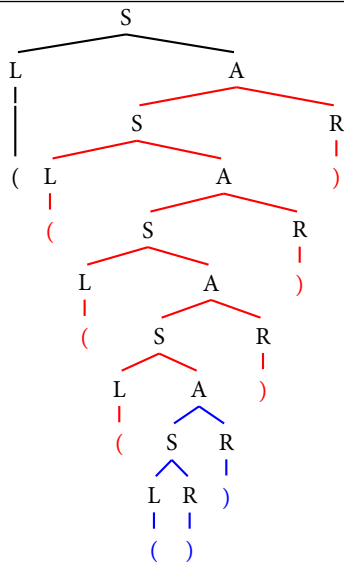


Wort

$$(((())))$$

Teilbaum

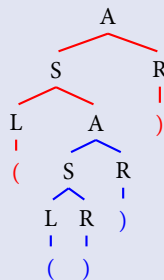




Wort

((((())))))))

Teilbaum



Fragen

- ▶ Wann tritt Nicht-Terminalzeichen sicher mehrfach in Pfad auf?
- ▶ Struktur aufgepumpter und fixer Anteile?

Satz: Pfadlänge in Binärbäumen

Sei $k \in \mathbb{N}$ und B ein Binärbaum. Mindestens ein Pfad der Länge $\geq k$ existiert in B , wenn die Blattanzahl $|B| \geq 2^k$ ist.

Satz: Pfadlänge in Binärbäumen

Sei $k \in \mathbb{N}$ und B ein Binärbaum. Mindestens ein Pfad der Länge $\geq k$ existiert in B , wenn die Blattanzahl $|B| \geq 2^k$ ist.

$k=0$

S

Induktionsschritt: $k \rightarrow k+1$



Sei $|B_1| \geq 2^{k_1}$, $|B_2| \geq 2^{k_2}$.

Sei oBdA. $|B_1| \geq |B_2|$.

$|B| \leq 2 \cdot 2^{k_1} = 2^{k_1+1}$.

Pfad der Länge k_1 in B_1 existiert nach IV,

Pfad der Länge $k_1 + 1$ nach Konstruktion.

Satz: Pfadlänge in Binärbäumen

Sei $k \in \mathbb{N}$ und B ein Binärbaum. Mindestens ein Pfad der Länge $\geq k$ existiert in B , wenn die Blattanzahl $|B| \geq 2^k$ ist.

$k=0$

S

Induktionsschritt: $k \rightarrow k+1$



Sei $|B_1| \geq 2^{k_1}$, $|B_2| \geq 2^{k_2}$.

Sei oBdA. $|B_1| \geq |B_2|$.

$|B| \leq 2 \cdot 2^{k_1} = 2^{k_1+1}$.

Pfad der Länge k_1 in B_1 existiert nach IV,

Pfad der Länge $k_1 + 1$ nach Konstruktion.

Satz: Pfadlänge in Binärbäumen

Sei $k \in \mathbb{N}$ und B ein Binärbaum. Mindestens ein Pfad der Länge $\geq k$ existiert in B , wenn die Blattanzahl $|B| \geq 2^k$ ist.

Mehrfache Nicht-Terminalsymbole

$G = (V, \Sigma, P, S)$ mit $|V|$ Nicht-Terminalsymbolen.

- ▶ Pfad mit Länge $|V|$ macht Wiederholung erforderlich ($|V| + 1$ Knoten).
- ▶ Garantiert für Wörter mit Länge $\geq 2^{|V|}$.



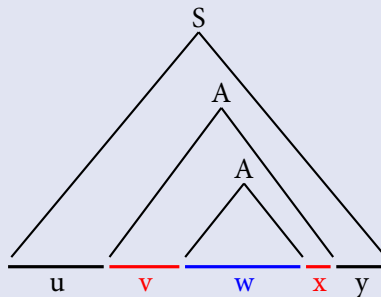
Fragen

- ▶ ✓ Wann tritt Nicht-Terminalzeichen sicher mehrfach in Pfad auf?
- ▶ Struktur aufgepumpter und fixer Anteile?

Ableitung

1. $G = (V, \Sigma, P, S)$, in CNF. Wähle Wort z mit $|z| \geq 2^{|V|}$, d.h. binärer Syntaxbaum mit $\geq 2^{|V|}$ Blättern.
2. Pfad der Länge $\geq |V|$ vorhanden $\Rightarrow$ enthält $|V| + 1$ Nonterminale $\Rightarrow$ Mehrfaches Auftreten von A (Suche von unten nach oben: A max. $|V|$ Schritte von Blättern entfernt)

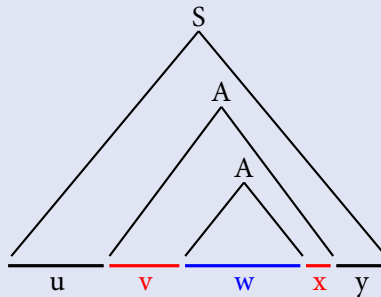
Syntaxbaum



Ableitung

3. A muss in Regel $A \rightarrow BC$ existieren, da ansonsten kein weiteres A erzeugt werden kann (CNF!) $\Rightarrow |vx| \geq 1$.
4. Oberes A max. $|V|$ Schritte von Blättern entfernt $\Rightarrow |vwx| \leq 2^{|V|}$.

Syntaxbaum



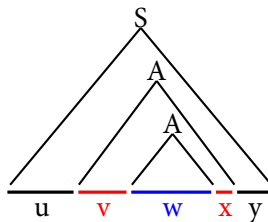
Fragen

- ▶ ✓ Wann tritt Nicht-Terminalzeichen sicher mehrfach in Pfad auf?
- ▶ ✓ Struktur aufgepumpter und fixer Anteile?

Pumping-Lemma für kfS

Sei L eine kontextfreie Sprache. Dann gibt es eine Konstante n , so dass sich alle Wörter $z \in L$, $|z| \geq n$ zerlegen lassen in $z = uvwxy$, so dass

1. $|vwx| \leq n$ (Mittelteil begrenzt)
2. $|vx| \geq 1$ (Mindestens eine aufgepumpte Zeichenreihe nicht leer)
3. Für alle $i \geq 0 : uv^iwx^iy \in L$



Grammatik

Ist $L \equiv \{a^i b^i c^i \mid i \in \mathbb{N}^+\}$ eine kontextfreie Sprache?

Beispiele

abc, aabbcc, aaabbbccc, ...

Behauptung: Nein! ✗

- ▶ Annahme: Sprache ist kontextfrei, Pumping-Lemma gilt.
- ▶ Pumping-Lemma anwenden.
- ▶ Aufgepumptes Wort nicht in Sprache $\Rightarrow$ Pumping-Lemma gilt nicht $\Rightarrow$ *Widerspruch!* $\Rightarrow$ Sprache kann nicht kontextfrei sein.

Grammatik

Ist $L \equiv \{a^i b^i c^i \mid i \in \mathbb{N}^+\}$ eine kontextfreie Sprache?

Beispiele

abc, aabbcc, aaabbbccc, ...

Beweis

- ▶ Pumping-Lemma: $\exists n \in \mathbb{N}$, so dass $z = a^i b^i c^i$, $|z| \geq n$ als $uvwxy$ darstellbar mit $|vx| \geq 1$, $|vwx| \leq n$. Wähle $i = n$.
 - ▶ $uv^0wx^0y = uwy \in L$, ebenso $uvwxy \in L$.
 - ▶ vx muss gleiche Anzahl a,b,c enthalten (wegen $|vx| \geq 1$ mindestens einen).
 - ▶ $|vwx| \leq n$: vwx kann nicht gleichzeitig a und c enthalten:
aaa, aab, abb, bbb, bbc, bcc, ccc
 - ▶ vx kann also nicht gleiche Anzahl a,b,c enthalten.
- ▶ Widerspruch!

Regulär

- ▶ RegExps vs. kontextfreie Sprache
- ▶ Ad-hoc-Lösung vs. Parser (Yacc, ANTLR, XText, ...)

Kontextfrei

- ▶ Wortproblem kfS: $\mathcal{O}(n^3)$, kontextsensitiv: $\mathcal{O}(2^n)$
- ▶ $\Rightarrow$ Für IT-Anwendungen kontraproduktiv!
- ▶ Anwendung: Linguistik
(Baumadjunktionsgrammatiken)

Praktische Anwendung

- ▶ Manche Programmiersprachen-Aspekte nicht-kontextfrei
- ▶ Zweistufiges Verfahren:
 - ▶ Kontextfreie Grammatik zur Syntaxanalyse, bsp. Wort $a^n b^j c^k$
 - ▶ Semantische Analyse, bsp. $n == j == k \Rightarrow$ effektiv $a^n b^n c^n$ sichergestellt

Regulär

- ▶ RegExps vs. kontextfreie Sprache
- ▶ Ad-hoc-Lösung vs. Parser (Yacc, ANTLR, XText, ...)

Kontextfrei

- ▶ Wortproblem kfS: $\mathcal{O}(n^3)$, kontextsensitiv: $\mathcal{O}(2^n)$
- ▶ $\Rightarrow$ Für IT-Anwendungen kontraproduktiv!
- ▶ Anwendung: Linguistik
(Baumadjunktionsgrammatiken)

Praktische Anwendung

- ▶ Manche Programmiersprachen-Aspekte nicht-kontextfrei
- ▶ Zweistufiges Verfahren:
 - ▶ Kontextfreie Grammatik zur Syntaxanalyse, bsp. Wort $a^n b^j c^k$
 - ▶ Semantische Analyse, bsp. $n == j == k \Rightarrow$ effektiv $a^n b^n c^n$ sichergestellt

Abschlusseigenschaften kontextfreier Sprachen

Die kontextfreien Sprachen sind abgeschlossen unter

- ▶ Vereinigung
- ▶ Produkt
- ▶ Stern

Sie sind *nicht* abgeschlossen unter

- ▶ Schnitt
- ▶ Komplement

Beweis: Siehe Tafel

Probleme bei DEAs und NEAs

- ▶ Sprachen der Form $a^n b^n$ oder ww^R nicht-regulär
 - ▶ Kein DEA/NEA zur Entscheidung konstruierbar
 - ▶ Alternatives Maschinenmodell?
- ▶ Intuitiver Grund: Kein »Gedächtnis« für bislang gelesene Eingabe
- ▶ Idee: NEA um (mehr als 1 implizites Bit) Speicher erweitern

Kellerautomat

Ein *Kellerautomat* (*Pushdown Automaton*, *PDA*) besteht aus

- ▶ Zuständen, in denen sich der Automat befinden kann
- ▶ einer Menge von Zeichen, die der Automat verarbeitet
- ▶ einer Menge von Zeichen, die im Kellerspeicher stehen können
- ▶ einer Regel für Zustandsübergänge
- ▶ einem Anfangszustand
- ▶ einem anfänglichen Zeichen im Keller

Illustration: Siehe Tafel

Kellerautomat

Ein *Kellerautomat* (*Pushdown Automaton*, *PDA*) ist gegeben durch ein 6-Tupel $(Q, \Sigma, \Gamma, \delta, q_0, \#)$:

- ▶ Zuständen, in denen sich der Automat befinden kann
- ▶ einer Menge von Zeichen, die der Automat verarbeitet
- ▶ einer Menge von Zeichen, die im Kellerspeicher stehen können
- ▶ einer Regel für Zustandsübergänge
- ▶ einem Anfangszustand
- ▶ einem anfänglichen Zeichen im Keller

Illustration: Siehe Tafel

Kellerautomat

Ein *Kellerautomat* (*Pushdown Automaton*, *PDA*) ist gegeben durch ein 6-Tupel $(Q, \Sigma, \Gamma, \delta, q_0, \#)$:

- ▶ Q : endliche Zustandsmenge
- ▶ einer Menge von Zeichen, die der Automat verarbeitet
- ▶ einer Menge von Zeichen, die im Kellerspeicher stehen können
- ▶ einer Regel für Zustandsübergänge
- ▶ einem Anfangszustand
- ▶ einem anfänglichen Zeichen im Keller

Illustration: Siehe Tafel

Kellerautomat

Ein *Kellerautomat* (*Pushdown Automaton*, *PDA*) ist gegeben durch ein 6-Tupel $(Q, \Sigma, \Gamma, \delta, q_0, \#)$:

- ▶ Q : endliche Zustandsmenge
- ▶ Σ : endliches Bandalphabet
- ▶ einer Menge von Zeichen, die im Kellerspeicher stehen können
- ▶ einer Regel für Zustandsübergänge
- ▶ einem Anfangszustand
- ▶ einem anfänglichen Zeichen im Keller

Illustration: Siehe Tafel

Kellerautomat

Ein *Kellerautomat* (*Pushdown Automaton, PDA*) ist gegeben durch ein 6-Tupel $(Q, \Sigma, \Gamma, \delta, q_0, \#)$:

- ▶ Q : endliche Zustandsmenge
- ▶ Σ : endliches Bandalphabet
- ▶ Γ : endliches Kelleralphabet
- ▶ einer Regel für Zustandsübergänge
- ▶ einem Anfangszustand
- ▶ einem anfänglichen Zeichen im Keller

Illustration: Siehe Tafel

Kellerautomat

Ein *Kellerautomat* (*Pushdown Automaton*, *PDA*) ist gegeben durch ein 6-Tupel $(Q, \Sigma, \Gamma, \delta, q_0, \#)$:

- ▶ Q : endliche Zustandsmenge
- ▶ Σ : endliches Bandalphabet
- ▶ Γ : endliches Kelleralphabet
- ▶ $\delta : Q \times (\Sigma \cup \epsilon) \times \Gamma \rightarrow \mathcal{P}_f(Q \times \Gamma^*)$, wobei $\mathcal{P}_f(X)$ die Menge aller *endlichen* Teilmengen der Menge X angibt
- ▶ einem Anfangszustand
- ▶ einem anfänglichen Zeichen im Keller

Illustration: Siehe Tafel

Kellerautomat

Ein *Kellerautomat* (*Pushdown Automaton, PDA*) ist gegeben durch ein 6-Tupel $(Q, \Sigma, \Gamma, \delta, q_0, \#)$:

- ▶ Q : endliche Zustandsmenge
- ▶ Σ : endliches Bandalphabet
- ▶ Γ : endliches Kelleralphabet
- ▶ $\delta : Q \times (\Sigma \cup \epsilon) \times \Gamma \rightarrow \mathcal{P}_f(Q \times \Gamma^*)$, wobei $\mathcal{P}_f(X)$ die Menge aller *endlichen* Teilmengen der Menge X angibt
- ▶ $q_0 \in Q$: Anfangszustand
- ▶ einem anfänglichen Zeichen im Keller

Illustration: Siehe Tafel

Kellerautomat

Ein *Kellerautomat* (*Pushdown Automaton, PDA*) ist gegeben durch ein 6-Tupel $(Q, \Sigma, \Gamma, \delta, q_0, \#)$:

- ▶ Q : endliche Zustandsmenge
- ▶ Σ : endliches Bandalphabet
- ▶ Γ : endliches Kelleralphabet
- ▶ $\delta : Q \times (\Sigma \cup \epsilon) \times \Gamma \rightarrow \mathcal{P}_f(Q \times \Gamma^*)$, wobei $\mathcal{P}_f(X)$ die Menge aller *endlichen* Teilmengen der Menge X angibt
- ▶ $q_0 \in Q$: Anfangszustand
- ▶ $\# \in \Gamma$: Ursprüngliches Kellersymbol

Illustration: Siehe Tafel

Interpretation Transitionsfunktion

$$\delta(q, a, A) \ni (q', B_1 \dots B_k)$$

1. M ist in Zustand q und liest Zeichen a . Oberstes Zeichen im Keller: A
2. Übergang:
 - ▶ M kann in den Zustand q' übergehen
 - ▶ Kellerzeichen A dann durch $B_1 \dots B_k$ ersetzt (B_1 steht oben).
 - ▶ Möglich: $B_1 B_2 \dots B_k = AB_2 \dots B_k$, d.h. *push-Operation*
 - ▶ Möglich: $k = 0$, d.h. *pop-Operation*

Nicht-Determinismus

- ▶ Mehrere simultane Übergänge möglich
- ▶ Spontane Übergänge (mit $a = \epsilon$) möglich

Akzeptanz

- ▶ *Kein* akzeptierender Endzustand!
- ▶ Akzeptanzkriterien für Wörter $x \in \Sigma^*$:
 1. Wort komplett gelesen
 2. Keller leer
- ▶ Äquivalent zu expliziten Endzuständen (ohne Beweis)

Konfiguration

Konfiguration eines PDA gegeben durch Drei-Tupel (Q, Σ^*, Γ^*)

- ▶ $q \in Q$: Momentaner Zustand
- ▶ $w' \in \Sigma^*$: Noch zu lesender Anteil der Eingabe
- ▶ $\gamma \in \Gamma^*$: Aktueller Kellerinhalt

Rechenschritt: Konfigurationsübergang

Die Relation $\vdash: Q \times \Sigma^* \times \Gamma^* \rightarrow Q \times \Sigma^* \times \Gamma^*$ gilt, wenn Konfiguration $k' \in (Q, \Sigma^*, \Gamma^*)$ aus Konfiguration $k \in (Q, \Sigma^*, \Gamma^*)$ durch einfache Anwendung der δ -Funktion hervorgeht:
 $k \vdash k'$

Konfiguration

Konfiguration eines PDA gegeben durch Drei-Tupel (Q, Σ^*, Γ^*)

- ▶ $q \in Q$: Momentaner Zustand
- ▶ $w' \in \Sigma^*$: Noch zu lesender Anteil der Eingabe
- ▶ $\gamma \in \Gamma^*$: Aktueller Kellerinhalt

Rechenschritt: Konfigurationsübergang

Die Relation $\vdash: Q \times \Sigma^* \times \Gamma^* \rightarrow Q \times \Sigma^* \times \Gamma^*$ ist definiert durch:

$$(q, w_1 w_2 \dots w_n, A_1 \dots A_m) \vdash (q', w_2 \dots w_n, B_1 \dots B_k A_2 \dots A_m)$$

wenn $\delta(q, w_1, A_1) \ni (q', B_1 \dots B_k)$

Konfiguration

Konfiguration eines PDA gegeben durch Drei-Tupel (Q, Σ^*, Γ^*)

- ▶ $q \in Q$: Momentaner Zustand
- ▶ $w' \in \Sigma^*$: Noch zu lesender Anteil der Eingabe
- ▶ $\gamma \in \Gamma^*$: Aktueller Kellerinhalt

Rechenschritt: Konfigurationsübergang

Die Relation $\vdash: Q \times \Sigma^* \times \Gamma^* \rightarrow Q \times \Sigma^* \times \Gamma^*$ ist definiert durch:

$$(q, w_1 w_2 \dots w_n, A_1 \dots A_m) \vdash (q', w_1 w_2 \dots w_n, B_1 \dots B_k A_2 \dots A_m)$$

wenn $\delta(q, \epsilon, A_1) \ni (q', B_1 \dots B_k)$

Beispiele

1. $L_1 \equiv \{a_1 a_2 \dots a_n \$ a_n a_{n-1} \dots a_1 \mid a_i \in \Sigma \setminus \{\$ \}\}$
2. $L_2 \equiv \{a_1 a_2 \dots a_n a_n a_{n-1} \dots a_1 \mid a_i \in \Sigma\}$

Maschinen & Ableitungen: Siehe Tafel

Erkannte Sprache eines PDA

Sei $\vdash^*$ die reflexiv-transitive Hülle von $\vdash$. Die durch einen PDA M akzeptierte Sprache $\mathcal{L}(M)$ ist gegeben durch

$$\mathcal{L}(M) = \{w \in \Sigma^* \mid (q_0, w, \#) \vdash^* (q, \epsilon, \epsilon) \text{ für ein } q \in Q\}$$

Deterministischer Kellerautomat (DPDA)

- ▶ PDA *ohne* nicht-deterministische Übergänge:

$$|\delta(q, a, A)| + |\delta(q, \epsilon, A)| \leq 1 \quad \forall q \in Q, a \in \Sigma, A \in \Gamma$$

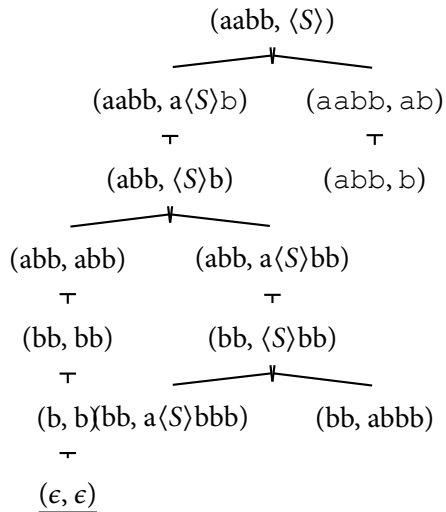
- ▶ Erkannte Sprache: *Deterministisch kontextfrei*
 - ▶ Sehr eng verwandt: LR(k)-Familie
 - ▶ Details: Siehe Vorlesung »Compilerbau«
- ▶ Ohne Beweis: $\mathcal{L}_3 \subset \mathcal{L}_{\text{def. kF}} \subset \mathcal{L}_2$
- ▶ Achtung: DPDA benötigt iA Endzustand q_f

Satz: Kellerautomaten und kfS

Eine Sprache L ist genau dann kontextfrei, wenn L von einem nichtdeterministischen Kellerautomaten erkannt wird.

Beweis:

- ▶ Vorwärtsrichtung siehe Tafel
- ▶ Rückrichtung siehe Literatur (formal relativ aufwendig)



Parser für kfS

- ▶ Annahme: Grammatik in CNF
- ▶ Ableitung von Wörtern:
 - ▶ Einzelner Buchstabe ($x = a$): Regel der Form $A \rightarrow a$
 - ▶ Mehrere Buchstaben ($x = a_1 a_2 \dots a_n$): Regel $A \rightarrow BC$.
A: Anfangsstück $a_1 a_2 \dots a_k$
B: Endstück $a_{k+1} \dots a_n$

Illustration: Siehe Tafel

Problemreduktion

$x \in \mathcal{L}(G)$ auf »kleinere« Teilprobleme reduziert:

- ▶ $a_1 a_2 \dots a_k \in \mathcal{L}(B)$
- ▶ $a_{k+1} \dots a_n \in \mathcal{L}(C)$
- ▶ *Achtung:* k und benötigte Regel sind unbekannt!
- ▶ Lösung: Alle Möglichkeiten »durchprobieren«

Nomenklatur

$x_{i,j}$ ist Teilwort von x ,

- ▶ das an Position i (Eins-basierte Indizierung) beginnt
- ▶ das Länge j besitzt, d.h. $|x_{i,j}| = j$

Beispiele: $x = abcdef \rightarrow x_{2,2} = bc, x_{1,4} = abcd, x_{1,6} = x$

CYK: Grundidee

- ▶ Ableitungen für alle Teilwörter mit Länge 1 finden
- ▶ Ableitungen für Teilwörter der Länge $j \leq n = |x|$ finden
 - ▶ j zerlegbar: $(1, j-1); (2, j-2); \dots; (j-1, 1)$
 - ▶ Bisherige Ergebnisse für Teilwörter w' mit $w' \leq j-1$ verwendbar!
- ▶ Mögliche Startpositionen von Teilwort mit Länge j :
 $1, 2, \dots, n-j+1$

Beispiel

- ▶ $w = abcdef, |w| = 6$. Betrachte Teilwörter Länge $j = 4$:
 - ▶ $i = 1$: $abcd \rightsquigarrow (a, bcd); (ab, cd); (abc, d)$
 - ▶ $i = 2$: $bcde \rightsquigarrow (b, cde); (bc, de); (bcd, e)$
 - ▶ $i = 3 = 6 - 4 + 1$: $cdef \rightsquigarrow (c, def); (cd, ef); (cde, f)$

Cocke-Younger-Kasami-Algorithmus III

```
1: procedure CYK( $\vec{w}, G$ )
2:    $n = |\vec{w}|$ 

3:   for  $i \leftarrow 1, n$  do
4:     for  $(A \rightarrow w_i) \in P$  do
5:       Erweitere  $T_{i,1}$  um den Eintrag  $A$ 
6:     end for
7:   end for

8:   for  $j \leftarrow 2, n$  do
9:     for  $i \leftarrow 1, n - j + 1$  do
10:      for  $k \leftarrow 1, j - 1$  do
11:        Erweitere  $T_{i,j}$  um den Eintrag  $A$ , wenn es
12:        eine Regel  $(A \rightarrow BC) \in P$  gibt, für die gilt:
13:        1.)  $B \in T_{i,k}$ 
14:        2.)  $C \in T_{i+k, j-k}$ 
15:      end for
16:    end for
17:  end for
```

▷ j : Länge Teilwort

▷ i : Startposition

▷ k : Aufteilung

CYK: Allgemeines

- ▶ Drei verschachtelte Schleifen
- ▶ Jede Schleife durchläuft etwa n Iterationen
- ▶ Gesamtaufwand: In etwa n^3 Iterationen (präzisere Notation später)
- ▶ Prinzip: *Dynamisches Programmieren*
 - ▶ Gesamtlösung aus Teillösungen ermitteln
 - ▶ Teillösungen in Tabelle vorrätig halten
- ▶ Mehr zum Entwurfsprinzip: Vorlesung *Algorithmen und Datenstrukturen*

Entscheidbare Eigenschaften kontextfreier Sprachen

- ▶ *Wortproblem:* ✓
 - ▶ CYK-Algorithmus verwenden
- ▶ *Leerheitsproblem:* ✓
 - ▶ Gegeben CNF-Grammatik
 - ▶ Regeln markieren, die Terminale erzeugen (beispielsweise $\langle A \rangle \rightarrow a$).
 - ▶ Dann alle Regeln markieren, die markierte Regel enthalten (beispielsweise $\langle B \rangle \rightarrow \langle A \rangle \langle C \rangle$).
 - ▶ Wenn $\langle S \rangle$ markiert wird: Sprache nicht leer
- ▶ *Endlichkeitsproblem:* ✓
 - ▶ Pumping-Lemma zu Hilfe nehmen
- ▶ *Schnittproblem:* ✗
- ▶ *Äquivalenzproblem:* ✗

Argumentation für nicht-Entscheidbarkeit: Siehe nächster Teil

1. Überblick und Einführung

- 1.1 Administrativa
- 1.2 Warum Theorie?

2. Formale Sprachen und Automaten

- 2.1 Endliche Automaten
- 2.2 Reguläre Sprachen I
- 2.3 Worterzeugung und Grammatiken
- 2.4 Chomsky-Hierarchie
- 2.5 Reguläre Sprachen und endliche Automaten
- 2.6 Nicht-Deterministische

endliche Automaten

- 2.7 Grammatiken und NEAs
- 2.8 Äquivalenz von NEAs und DEAs
- 2.9 Reguläre Ausdrücke
- 2.10 Das Pumping-Lemma
- 2.11 Automatenminimierung
- 2.12 Abschlusseigenschaften regulärer Sprachen
- 2.13 Kontextfreie Sprachen
- 2.14 Kellerautomaten
- 2.15 CYK-Algorithmus

3. Berechenbarkeitstheorie

- 3.1 Turing-Maschinen

- 3.2 LBAs und der Satz von Kuroda
- 3.3 Berechenbarkeit und Church-Turing-These
- 3.4 Varianten von Turing-Maschinen
- 3.5 Berechnungskomplexität
- 3.6 Alternative Berechnungsmodelle
- 3.7 Universelle Turing-Maschinen
- 3.8 Das Halteproblem

4. Komplexitätstheorie

- 4.1 Definitionen
- 4.2 Komplexitätsklassen
- 4.3 Struktur von NP

Berechenbarkeitstheorie

Nachteile Kellerautomat

- ▶ Kein wahlfreier Zugriff auf Stapelspeicher
- ▶ Kein wahlfreier Zugriff auf Eingabewort
- ▶ Keine »Markierungen« in Eingabe

Erweitertes Maschinenmodell

- ▶ Eingabe- und Speicherband vereinigen
- ▶ Wahlfreie Zugriffe erlauben

(Nicht-)Deterministische Turing-Maschine

Eine *Turing-Maschine* ist gegeben durch ein 7-Tupel $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$. Dabei ist

- ▶ Q eine endliche Zustandsmenge
- ▶ Σ ein endliches Eingabealphabet
- ▶ $\Gamma \supset \Sigma$ ein endliches Arbeitsalphabet
- ▶ δ die Transitionsfunktion mit Signatur
 - ▶ Deterministisch: $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, N\}$
 - ▶ Nicht-Deterministisch: $\delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R, N\})$
- ▶ $q_0 \in Q$ der Startzustand
- ▶ $\square \in \Gamma - \Sigma$ das »Blank«-Symbol
- ▶ $F \subseteq Q$ die Menge der Endzustände

Illustration: Siehe Tafel

Konfiguration

Interpretation Übergangsfunktion:

- ▶ Deterministisch: $\delta(q, a) = (q', b, x)$
- ▶ Nicht-Deterministisch: $\delta(q, a) \ni (q', b, x)$

Vorgehen:

1. M befindet sich im Zustand q und liest Zeichen a
2. M geht in Zustand q' über, schreibt das Zeichen b *auf den Platz von a* , und führt die Kopfbewegung $x \in \{L, R, N\}$ aus

Beispiel: Siehe Tafel

Konfiguration TM

Eine Konfiguration einer TM ist gegeben durch ein Wort $k \in \Gamma^* Q \Gamma^*$. Sei $k = \alpha q \beta$:

- ▶ q : Aktueller Zustand
- ▶ α : Wort links des Schreib/Lese-Kopfes
- ▶ $\beta = \beta_0 \beta_1 \cdots \beta_n$: Wort rechts des S/L-Kopfes. Kopf steht auf Feld mit Beschriftung β_0

Startkonfiguration $q_0 \vec{w}$

- ▶ $w \in \Sigma^*$ steht auf Band, M in Zustand q_0
- ▶ S/L-Kopf steht auf erstem Buchstaben von w
- ▶ Restliches Band mit Blanks $\square$ befüllt

Konfigurationsübergänge I

Die zweistellige Relation $\vdash$ gibt *Konfigurationsübergänge* bei einer Turing-Maschine an:

$$a_1 \dots a_m q b_1 \dots b_n = \begin{cases} a_1 \dots a_m \textcolor{red}{q}' c b_2 \dots b_n & \delta(q, b_1) = (q', c, N), \\ & m \geq 0, n \geq 1 \\ a_1 \dots a_m \textcolor{red}{q}' b_2 \dots b_n & \delta(q, b_1) = (q', c, R), \\ & m \geq 0, n \geq 2 \\ a_1 \dots a_{m-1} \textcolor{red}{q}' a_m c b_2 \dots b_n & \delta(q, b_1) = (q', c, L), \\ & m \geq 1, n \geq 1 \end{cases}$$

Konfigurationsübergänge II

- Sonderfall 1: $n = 1$, M läuft nach rechts:

$$a_1 \dots a_m q b_1 \vdash a_1 \dots a_m c q' \square \text{ für } \delta(q, b_1) = (q', c, R)$$

- Sonderfall 2: $m = 0$, M läuft nach links

$$q b_1 \dots b_n \vdash q' \square c b_2 \dots b_n \text{ für } \delta(q, b_1) = (q', c, L)$$

Turing-Maschine für $L \equiv \{0^n 1^n \mid n \in \mathbb{N}\}$

Grundidee: Alternierend 0 in X und 1 in Y ändern (jeweils in Gruppen)

- ▶ 0 in X ändern
- ▶ Nach rechts fahren, erste 1 in Y ändern
- ▶ Nach links fahren bis zum ersten X
- ▶ Schleife mit davon rechts stehender 0 neu starten
- ▶ Wenn keine Null rechts steht: Prüfen, ob noch Einsen vorhanden sind
 - ▶ Nein: Akzeptieren
 - ▶ Ja: Nicht akzeptieren

Übergangsfunktion δ

	0	1	X	Y	$\square$
q_0	(q_1, X, R)	$\emptyset$	$\emptyset$	(q_3, Y, R)	$\emptyset$
q_1	$(q_1, 0, R)$	(q_2, Y, L)	$\emptyset$	(q_1, Y, R)	$\emptyset$
q_2	$(q_2, 0, L)$	$\emptyset$	(q_0, X, R)	(q_2, Y, L)	$\emptyset$
q_3	$\emptyset$	$\emptyset$	$\emptyset$	(q_3, Y, R)	$(q_f, \square, R)$
q_f	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Maschinendefinition

$$M = (\{q_0, q_1, q_2, q_3, q_f\}, \{0, 1\}, \{0, 1, X, Y, \square\}, \delta, q_0, \{q_f\})$$

Übergangsfunktion δ

	o	1	X	Y	$\square$
q_0	(q_1, X, R)	$\emptyset$	$\emptyset$	(q_3, Y, R)	$\emptyset$
q_1	(q_1, o, R)	(q_2, Y, L)	$\emptyset$	(q_1, Y, R)	$\emptyset$
q_2	(q_2, o, L)	$\emptyset$	(q_0, X, R)	(q_2, Y, L)	$\emptyset$
q_3	$\emptyset$	$\emptyset$	$\emptyset$	(q_3, Y, R)	$(q_f, \square, R)$
q_f	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Beispiel: 0011 ✓

$q_0 0011 \vdash Xq_1 011 \vdash X0q_1 11 \vdash Xq_2 0Y1 \vdash q_2 X0Y1 \vdash Xq_0 0Y1$
 $\vdash XXq_1 Y1 \vdash XXYq_1 1 \vdash XXq_2 YY \vdash Xq_2 XYY \vdash XXq_0 YY$
 $\vdash XXYq_3 Y \vdash XXYYq_3 \square \vdash XXYY\square q_f \square$

Übergangsfunktion δ

	0	1	X	Y	$\square$
q_0	(q_1, X, R)	$\emptyset$	$\emptyset$	(q_3, Y, R)	$\emptyset$
q_1	$(q_1, 0, R)$	(q_2, Y, L)	$\emptyset$	(q_1, Y, R)	$\emptyset$
q_2	$(q_2, 0, L)$	$\emptyset$	(q_0, X, R)	(q_2, Y, L)	$\emptyset$
q_3	$\emptyset$	$\emptyset$	$\emptyset$	(q_3, Y, R)	$(q_f, \square, R)$
q_f	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Beispiel: 0010 ✗

$$\begin{aligned}
 q_0 0010 &\vdash Xq_1 010 \vdash X0q_1 10 \vdash Xq_2 0Y0 \vdash q_2 X0Y0 \vdash Xq_0 0Y0 \\
 &\vdash XXq_1 Y0 \vdash XXYq_1 0 \vdash XXY0q_1 \square
 \end{aligned}$$

Modell und reale Welt

- ▶ Unendliches Band & Nicht-Determinismus: Unrealistisch (endliche Festplatten, endlicher RAM-Speicher, ...)
- ▶ Eingeschränktes Modell: Endliches Band

Linear beschränkte Turing-Maschine (LBA)

Eine nicht-deterministische TM M heißt linear beschränkt (*linear bounded automaton*), wenn für alle $a_1 a_2 \dots a_{n-1} a_n \in \Sigma^+$ und für alle Konfigurationen $\alpha q \beta$ mit $q_0 \hat{a}_1 a_2 \dots a_{n-1} \hat{a}_n \vdash \alpha q \beta$ gilt, dass $|\alpha \beta| = n$.

- ▶ Bandteil mit Eingabe wird nicht verlassen (Beschränkung auf $n!$)
- ▶ Endemarkierung für rechte Grenze (links trivial): $\Sigma \rightarrow \Sigma' \equiv \Sigma \cup \{\hat{a} \mid a \in \Sigma\}$

Satz von Kuroda

Die von linear beschränkten, nicht-deterministischen Turing-Maschinen akzeptierbaren Sprachen sind genau die kontextsensitiven Sprachen $\mathcal{L}_1$

Beweis Vorwärtsrichtung (Rückwärtsrichtung: Siehe Literatur) :

- ▶ Bei Eingabe $\vec{w} = x_1 x_2 \cdots x_n$: Nicht-deterministisch Regel $(u \rightarrow v) \in P$ wählen
- ▶ Vorkommen von v auf Band suchen
- ▶ Gefunden $\Rightarrow$ Teilwort durch u ersetzen (wenn $|u| < |v|$: Restwort nach links verschieben)
- ▶ Lediglich S auf Band: Anhalten, ansonsten zurück zu Schritt 1
- ▶ Gegeben G mit $\mathcal{L}(G) \in \mathcal{L}_1$:

$$\begin{aligned} x \in \mathcal{L}(G) &\Leftrightarrow \exists \text{ Ableitung } S \Rightarrow \cdots \Rightarrow x \\ &\Leftrightarrow \exists \text{ Rechnung von } M, \text{ die Ableitung umgekehrt simuliert} \\ &\Leftrightarrow x \in \mathcal{L}(M) \end{aligned}$$

Satz (Erweiterung Satz von Kuroda)

Die durch allgemeine Turing-Maschinen akzeptierbaren Sprachen sind genau die Phrasenstruktursprachen $\mathcal{L}_0$

Beweis: Vorwärtsrichtung analog Satz von Kuroda (TM *ohne* Beschränkung),
Rückwärtsrichtung: Siehe Literatur

Status Quo

- ▶ Reguläre Sprachen $\mathcal{L}_3 \Leftrightarrow \text{DEA, NEA}$
- ▶ Kontextfreie Sprachen $\mathcal{L}_2 \Leftrightarrow \text{PDA}$
- ▶ Kontextsensitive Sprachen $\mathcal{L}_1 \Leftrightarrow \text{LBA}$
- ▶ Phrasenstruktur-Sprachen $\mathcal{L}_0 \Leftrightarrow \text{Turing-Maschine}$

Turing-Berechenbarkeit

Ein Problem, das (in Form der Berechnung einer Funktion) auf einer Turing-Maschine gelöst werden kann, bezeichnet man als *Turing-berechenbar*.

Church-Turing-These

Die durch *formale Definition der Turing-Berechenbarkeit* erfasste Klasse von Funktionen stimmt genau mit der Klasse der im *intuitiven Sinne berechenbaren* Funktionen überein.

Alternative Maschinenmodelle

- ▶ While- und Goto-Programme (einfache Programmiersprachen)
- ▶ μ -rekursive Funktionen (nicht behandelt)
- ▶ Registermaschinen (moderner »Computer«)
- ▶ Genaue Definitionen später!

Turing-Berechenbarkeit

Ein Problem, das (in Form der Berechnung einer Funktion) auf einer Turing-Maschine gelöst werden kann, bezeichnet man als *Turing-berechenbar*.

Church-Turing-These

Die durch *formale Definition der Turing-Berechenbarkeit* erfasste Klasse von Funktionen stimmt genau mit der Klasse der im *intuitiven Sinne berechenbaren* Funktionen überein.

Alternative Maschinenmodelle

- ▶ While- und Goto-Programme (einfache Programmiersprachen)
- ▶ μ -rekursive Funktionen (nicht behandelt)
- ▶ Registermaschinen (moderner »Computer«)
- ▶ Genaue Definitionen später!

Endlosschleifen

- ▶ Turing-Maschinen können in Endlosschleife geraten!
- ▶ Bei DEA, NEA nicht möglich

Definition: Akzeptanz

- ▶ Eine Turing-Maschine M *akzeptiert* eine Sprache L , wenn M alle $x \in L$ akzeptiert, d.h. in einem Zustand $q \in F$ hält.
- ▶ Achtung: $x \notin L$ mit M hält *nicht* für x ist möglich!

Definition: Entscheidbarkeit

Eine Turing-Maschine M *entscheidet* eine Sprache L , wenn M die Sprache L akzeptiert und für alle $x \notin L$ nach endlich vielen Schritten in einem Zustand $q \notin F$ hält.

Nomenklatur

- ▶ L ist *semi-entscheidbar* (*recognisable*, »rekursiv aufzählbar«, »recursively enumerable«) $\Leftrightarrow \exists M$, die L akzeptiert
- ▶ L ist *entscheidbar* (*decidable*, »rekursiv«), $\Leftrightarrow \exists M$, die L entscheidet

Turing-Berechenbarkeit II

- ▶ Eine partielle Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ ist *Turing-berechenbar*, wenn es eine Turing-Maschine M gibt, die bei Eingabe von $x_1, x_2, \dots, x_k$ die Ausgabe $y = f(x_1, x_2, \dots, x_k)$ liefert und hält:

$$q_0 x_1 x_2 \cdots x_k \vdash^* qy \text{ mit } q \in F \wedge y \in \Gamma^* \wedge y = f(x_1, \dots, x_k)$$

- ▶ Falls M nicht hält, gilt $f(x_1, \dots, x_k) = \perp$

Offene Fragen

- ▶ Systematische Informationkodierung
- ▶ Ressourcenverbrauch (Zeit und Bandplatz)
- ▶ »Orthodoxere« Berechnungsmodelle

Binäres Alphabet

- ▶ Standardalphabete: $\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, \square\}$
- ▶ Andere endliche Alphabete: Kodierung durch Abzählen
 - ▶ i -ten Buchstaben durch Binärzahl $\text{bin}(i)$ darstellen
 - ▶ Maximale Anzahl Binärziffern bekannt $\Rightarrow$ von Links mit Nullen auffüllen
 - ▶ Alternativ: Trennzeichen verwenden
- ▶ Hinweis: Unäre Kodierung ungeeignet
 - ▶ Exponentiell lange Zahlen im Vergleich zu Binärkodierung
 - ▶ Dito für jede andere »vernünftige« Kodierung

Turing-Berechenbarkeit III

- ▶ Eine partielle Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ ist *Turing-berechenbar*, wenn es eine Turing-Maschine M gibt, die bei Eingabe von $x_1, x_2, \dots, x_k$ die Ausgabe $y = f(x_1, x_2, \dots, x_k)$ liefert und hält:

$$q_0 \text{ bin}(x_1) \square \text{bin}(x_2) \square \dots \square \text{bin}(x_k) \overset{*}{\vdash} \square \dots \square q \text{ bin}(y) \square \dots \square$$

mit $q \in F \wedge y \in \Gamma^* \wedge y = f(x_1, \dots, x_k)$

- ▶ Falls M nicht hält, gilt $f(x_1, \dots, x_k) = \perp$

Robuste Definition

- ▶ Turing-Maschinen sind robust gegenüber (sinnvollen) Modifikationen der genauen Arbeitsdefinition
- ▶ Erweiterungen: Gleiche Berechnungsmächtigkeit, einfachere Handhabbarkeit

Varianten (Illustration: Tafel)

- ▶ k Spuren, *ein* Schreib/Lesekopf
- ▶ k Bänder, k Schreib/Leseköpfe

Robuste Definition

- ▶ Turing-Maschinen sind robust gegenüber (sinnvollen) Modifikationen der genauen Arbeitsdefinition
- ▶ Erweiterungen: Gleiche Berechnungsmächtigkeit, einfachere Handhabbarkeit

Varianten (Illustration: Tafel)

- ▶ k Spuren, *ein* Schreib/Lesekopf
 - ▶ Folge von k Zeichen: Neues Symbol
 - ▶ Eingabe- und Bandalphabet ersetzen
 - ▶ Äquivalenz: Trivial
- ▶ k Bänder, k Schreib/Leseköpfe

Robuste Definition

- ▶ Turing-Maschinen sind robust gegenüber (sinnvollen) Modifikationen der genauen Arbeitsdefinition
- ▶ Erweiterungen: Gleiche Berechnungsmächtigkeit, einfachere Handhabbarkeit

Varianten (Illustration: Tafel)

- ▶ k Spuren, *ein* Schreib/Lesekopf
 - ▶ Folge von k Zeichen: Neues Symbol
 - ▶ Eingabe- und Bandalphabet ersetzen
 - ▶ Äquivalenz: Trivial
- ▶ k Bänder, k Schreib/Leseköpfe
 - ▶ *Unabhängige* Kopfoperationen, Eingabe auf Band 1
 - ▶ Transitionsfunktion: $\delta : Q \setminus F \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, N, R\}^k$
 - ▶ Äquivalenz zu zeigen

Ressourcenfragen

- ▶ Wieviel Bandplatz verbraucht?
- ▶ Wie viele Rechenschritte notwendig?

Zeitkomplexität $T_M(\vec{x})$

$T_M(\vec{x}) \equiv$ Anzahl der Schritte von M mit
Eingabe $\vec{x} \in \Sigma^*$

Platzkomplexität $S_M(\vec{x})$

$S_M(\vec{x}) \equiv$ Anzahl *verschiedener* Zellen, die M
bei Eingabe $\vec{x} \in \Sigma^*$ besucht

Zeitkomplexität $T_M(\vec{x})$

$T_M(\vec{x}) \equiv$ Anzahl der Schritte von M mit
Eingabe $\vec{x} \in \Sigma^*$

Platzkomplexität $S_M(\vec{x})$

$S_M(\vec{x}) \equiv$ Anzahl *verschiedener* Zellen, die M
bei Eingabe $\vec{x} \in \Sigma^*$ besucht

Worst-Case-Betrachtung

- ▶ Worst-Case-Laufzeit (*time complexity*):

$$T_M(n) \equiv \max_{x \in \Sigma^*, |\vec{x}| \leq n} T_M(\vec{x})$$

- ▶ Worst-Case-Platzverbrauch (*space complexity*):

$$S_M(n) \equiv \max_{x \in \Sigma^*, |\vec{x}| \leq n} S_M(\vec{x})$$

Platz- und Zeitbeschränkung

Gegeben $t, s : \mathbb{N} \times \mathbb{N}$. Eine Turing-Maschine M heißt $t(n)$ -zeitbeschränkt und $s(n)$ -platzbeschränkt, wenn für alle $n \in \mathbb{N}$ gilt:

$$T_M(n) \leq t(n) \wedge S_M(n) \leq s(n)$$

Äquivalenz TM und Mehrband-TM

Seien $c_1, c_2 \in \mathbb{N}$. Jede $t(n)$ -zeit- und $s(n)$ -platzbeschränkte k -Band-Turingmaschine M_k kann durch eine $c_1 \cdot t(n) \cdot s(n)$ -zeitbeschränkte und $c_2 \cdot s(n)$ -platzbeschränkte Turingmaschine M simuliert werden.

Beweis: Siehe Tafel

Folgerungen

- ▶ Maximal ein Feld pro Zeitschritt besucht $\Rightarrow s(n) \leq t(n)$
- ▶ $\Rightarrow c_1 \cdot t(n) \cdot s(n) \leq c_1 \cdot t(n)^2$
- ▶ 1-TM versus k -TM: Höchstens quadratische Verlangsamung $\Rightarrow$ Irrelevant

Äquivalenz TM und Mehrband-TM

Seien $c_1, c_2 \in \mathbb{N}$. Jede $t(n)$ -zeit- und $s(n)$ -platzbeschränkte k -Band-Turingmaschine M_k kann durch eine $c_1 \cdot t(n) \cdot s(n)$ -zeitbeschränkte und $c_2 \cdot s(n)$ -platzbeschränkte Turingmaschine M simuliert werden.

Beweis: Siehe Tafel

Folgerungen

- ▶ Maximal ein Feld pro Zeitschritt besucht $\Rightarrow s(n) \leq t(n)$
- ▶ $\Rightarrow c_1 \cdot t(n) \cdot s(n) \leq c_1 \cdot t(n)^2$
- ▶ 1-TM versus k -TM: Höchstens quadratische Verlangsamung $\Rightarrow$ Irrelevant

Äquivalenz DTM und NTM

Zu jeder nicht-deterministischen Turing-Maschine M gibt es eine äquivalente deterministische Turing-Maschine M'

Illustration: Siehe Tafel

Konsequenzen

- ▶ NTM und DTM: Gleiche Berechnungsmacht, drastisch unterschiedliche Ressourcenaufwände (exponentielle Verlangsamung)
- ▶ DTM und k -Band-DTM: Gleiche Berechnungsmacht, unsubstantielle Laufzeitunterschiede
- ▶ ☞ Unterschiedliche Einsatzzwecke:
 - ▶ Praktische Effizienz: An *Laufzeit* DTM gemessen
 - ▶ Prinzipielle Lösbarkeit: An *Entscheidbarkeit* durch NTM gemessen

Äquivalenz DTM und NTM

Zu jeder nicht-deterministischen Turing-Maschine M gibt es eine äquivalente deterministische Turing-Maschine M'

Illustration: Siehe Tafel

Konsequenzen

- ▶ NTM und DTM: Gleiche Berechnungsmacht, drastisch unterschiedliche Ressourcenaufwände (exponentielle Verlangsamung)
- ▶ DTM und k -Band-DTM: Gleiche Berechnungsmacht, unsubstantielle Laufzeitunterschiede
- ▶ ☞ Unterschiedliche Einsatzzwecke:
 - ▶ Praktische Effizienz: An *Laufzeit* DTM gemessen
 - ▶ Prinzipielle Lösbarkeit: An *Entscheidbarkeit* durch NTM gemessen

while: Syntax

$$\langle const \rangle \rightarrow 0 \mid 1 \mid 2 \mid \dots$$

$$\langle var \rangle \rightarrow x_0 \mid x_1 \mid x_2 \mid \dots$$

$$\langle stmts \rangle \rightarrow \langle stmt \rangle \mid \langle stmt \rangle ; \langle stmts \rangle$$

$$\langle stmt \rangle \rightarrow \langle assignment \rangle \mid \langle loop \rangle \mid \langle while \rangle$$

$$\langle assignment \rangle \rightarrow \langle var \rangle := \langle expr \rangle$$

$$\langle loop \rangle \rightarrow \text{loop } \langle var \rangle \text{ do } \langle stmts \rangle \text{ end}$$

$$\langle while \rangle \rightarrow \text{while } x_i \neq 0 \text{ do } \langle stmts \rangle \text{ end}$$

$$\langle expr \rangle \rightarrow \langle var \rangle \mid \langle const \rangle \mid \langle expr \rangle + \langle expr \rangle \mid \langle expr \rangle - \langle expr \rangle$$

Anmerkungen

- ▶ `loop`: Anzahl Schleifendurchläufe *fest* bestimmt
- ▶ `while`: Kontrollvariable darf in $\langle stmts \rangle$ geändert werden
- ▶ `if` durch `loop` simulierbar. Beispiel: `if  $x_0=0$  then P end` äquivalent zu

```
 $x_1 := 1;$   
loop  $x_0$  do  $x_1 := 0$  end;  
loop  $x_1$  do P end;
```

while: Semantik I

- ▶ Speichervektor $\vec{v} = (x_0, x_1, \dots, x_n, x_{n+1}, \dots, x_m)$
 - ▶ Eingabe in $x_0, \dots, x_n$
 - ▶ Ausgabe in $x_{n+1}, \dots, x_m$
- ▶ Semantik gegeben durch $\delta : \mathbb{N}^{n+1} \times x \rightarrow \mathbb{N}^{m-n}$ mit $x \in L_{\text{while}}$
- ▶ Induktive Definition über Programmstruktur

while: Semantik II

► Wertzuweisung

$$\delta(\vec{v}, x_i := \langle expr \rangle) \equiv \vec{v}[x_i \leftarrow \langle expr \rangle]$$

► Konkatenation

$$\delta(\vec{v}, P_1; P_2) \equiv \delta(\delta(\vec{v}, P_1), P_2)$$

► loop-Schleife

$$\delta(\vec{v}, \text{loop } x_i \text{ do } P \text{ end}) \equiv \delta(\vec{v}, \underbrace{P; P; P; \dots; P}_{x_i\text{-fach}}) \equiv \delta(\vec{v}, P^{x_i})$$

while: Semantik III

► Terminierungsmenge

$$T_i \equiv \{k \in \mathbb{N} \mid \delta(\vec{v}, P^k) = (x_0, \dots, x_{i-1}, 0, x_{i+1}, \dots) \text{ mit } x_l \in \mathbb{N}_0\}$$

► Aussage: Wie oft muss P wiederholt werden, damit x_i Null wird?

► Kleinste Wiederholungsanzahl: $\min(T_i)$

► Semantik `while`-Schleife

$$\delta(\vec{v}, \text{while } x_i \text{ do } P \text{ end}) \equiv \begin{cases} \perp & \text{falls } T_i = \emptyset \\ \delta(\vec{v}, P^{\min(T_i)}) & \text{falls } T_i \neq \emptyset \end{cases}$$

► $\delta(\perp, P) = \perp$

- Eine nicht-terminierende Schleife $\Rightarrow$ Gesamtes Programm terminiert nicht

While-Berechenbarkeit

- ▶ Eine partielle Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ ist *While-berechenbar*, wenn es ein While-Programm P gibt, das bei Eingabe von $x_1, x_2, \dots, x_k$ die Ausgabe $y = f(x_1, x_2, \dots, x_k)$ liefert und terminiert:

$$\delta((x_1, x_2, \dots, x_k, 0), P) = (x_1, \dots, x_k, y)$$

- ▶ Falls P nicht terminiert, gilt $f(x_1, \dots, x_k) = \perp$

Satz: While-Programme und Turing-Maschinen

Jedes While-Programm kann auf einer Turing-Maschine simuliert werden.

Beweis: While-Sprachelemente auf Mehrband-Turingmaschine abbilden.

Goto-Sprachelemente

- ▶ Wertzuweisungen: $x_i := x_j \pm c$
- ▶ Unkonditionale Sprünge: `goto  $M_i$`
 - ▶ i -te Zeile implizit mit Label M_i versehen
- ▶ Bedingte Sprünge: `if  $x_i = c$  then goto  $M_i$`
- ▶ Stop-Anweisung: `halt`
 - ▶ Letzte Programmanweisung implizit `halt`

Satz

Funktion While-berechenbar $\Leftrightarrow$ Funktion Goto-berechenbar

Beweis $\Rightarrow$: Jede While-Schleife `while  $x_i \neq 0$  do  $P$  end` äquivalent zu

M_1 : if $x_i = 0$ then goto M_k ;

P ;

M_{k-1} : goto M_1 ;

M_k : ...

SatzFunktion While-berechenbar $\Leftrightarrow$ Funktion Goto-berechenbarBeweis » $\Leftarrow$ «: Goto-Programm $M_1 : A_1;$ $M_2 : A_2;$

...

 $M_n : A_n;$

simulieren durch

 $y := 1;$ while $y \neq 0$ do if $y = 1$ then A_1 end; if $y = 2$ then A_2 end;

...

 if $y = n$ then A_n end;

end

Zuordnung Sprachelemente

<code>goto M_j</code>	 <code>$y := j$</code>
<code>if x_i goto M_j</code>	 <code>$y := y + 1$; if x_i then $y := j$ end;</code>
<code>halt</code>	 <code>$y := 0$</code>
<code>A (sonstige Anweisungen)</code>	 <code>A; $y := y + 1$</code>

Satz: Turing-Maschinen und Goto-Programme

Jede deterministische Turing-Maschine kann durch ein Goto-Programm simuliert werden.

Beweis:

- ▶ $Q = \{q_1, q_2, \dots, q_n\}$ und $\Gamma = \{a_1, \dots, a_N\}$
 - ▶ Mengen eindeutig numeriert
 - ▶ $\text{idx}(a_i) = i, i \geq 1$
- ▶ Wähle $b > |\Gamma| = N$

Konfiguration der TM

$$\underbrace{a_{i_m} \dots a_{i_1}}_{\text{Linker Bandteil } \vec{x}} \quad q_l \quad \underbrace{a_{j_1} \dots a_{j_n}}_{\text{Rechter Bandteil } \vec{y}}$$

Konfiguration der TM

$$\underbrace{a_{i_m} \dots a_{i_1}}_{\text{Linker Bandteil } \vec{x}} \quad q_l \quad \underbrace{a_{j_1} \dots a_{j_n}}_{\text{Rechter Bandteil } \vec{y}}$$

☞ $\vec{x}, \vec{y}$ bijektiv repräsentierbar durch b -adische Zahlen *ohne* Null:

$$x \equiv \underbrace{\langle i_m i_{m-1} \dots i_2 i_1 \rangle}_{\text{Big Endian}} = \sum_{k=1}^m i_k b^{k-1}$$

$$y \equiv \underbrace{\langle j_1 j_2 \dots j_{n-1} j_n \rangle}_{\text{Little Endian}} = \sum_{k=1}^n j_k b^{k-1}$$

$$\delta(q_i, \sigma) = (q_j, \sigma', L)$$

- ▶ Erstes Zeichen in y ersetzen

$$y \leftarrow y \div b$$

$$y \leftarrow \text{idx}(\sigma') + b \times y$$

- ▶ Linksbewegung ausführen

$$y \leftarrow (x \bmod b) + b \times y$$

$$x \leftarrow x \div b$$

- ▶ Zustand ändern: $q \leftarrow j$

$$\delta(q_i, \sigma) = (q_j, \sigma', R)$$

- ▶ Erstes Zeichen in y ersetzen

$$y \leftarrow y \div b$$

$$y \leftarrow \text{idx}(\sigma') + b \times y$$

- ▶ Rechtsbewegung ausführen

$$x \leftarrow b \times x + (y \bmod b)$$

$$y \leftarrow y \div b$$

- ▶ Zustand ändern: $q \leftarrow j$

Beispiel: Siehe Tafel

1. Variablen x, y, q initialisieren
2. Transitionsfunktion simulieren:

```
 $M_2$ :  $a := y \bmod b$ ; // Aktuelles Zeichen in a  
    if  $q=1$  and  $a=1$  then goto  $M_{1.1}$ ;  
    if  $q=1$  and  $a=2$  then goto  $M_{1.2}$ ;  
    ...  
    if  $q=n$  and  $a=N$  then goto  $M_{n.N}$ ;
```

```
 $M_{1.1}$ :  $\delta(q_1, a_1)$  simulieren  
    goto  $M_2$ ;  
 $M_{1.2}$ :  $\delta(q_1, a_2)$  simulieren  
    goto  $M_2$ ;  
...  
 $M_{n.N}$ :  $\delta(q_n, a_N)$  simulieren  
    goto  $M_2$ ;
```

3. Ergebnis nach x_0 schreiben

Nachteil TM

- ▶ Feste Programmierung: δ und Q fest gegeben
- ▶ Eingabe bestimmt Ablauf, *nicht* Algorithmus
- ▶ Unpraktisch (aber: kein fundamentaler Unterschied zu Computer)
- ▶ Lösung: *Universelle Turingmaschine*

Universelle Turing-Maschine

- ▶ Zwei Eingaben
 - ▶ Eingabedaten ω
 - ▶ Abgearbeitetes Programm f
- ▶ Maschine berechnet $f(\omega)$
- ▶ Problem: Codierung einer TM auf Band?

Kodierung einer Turing-Maschine

- ▶ Maschine M oBdA gegeben durch

$$M = (\{q_0, \dots, q_n, q_f\}, \{0, 1\}, \{0, 1, \square\}, \delta, q_0, \square, \{q_f\})$$

- ▶ Transitionsfunktion $\delta(q, a) = (q', b, \{L, R, N\})$ interpretiert als 5-Tupel

$$(q, a, q', b, \{L, R, N\})$$

mit $a, b \in \{0, 1\}$ und $q, q' \in Q$

- ▶ Kodierung obiger Komponenten als Zahlen
 - ▶ Für alle Übergänge durchführen
 - ▶ Zahlen konkatenieren $\Rightarrow$ Größere Zahl
 - ▶ Maschine M durch Zahl charakterisiert

Gödelisierung: Unäre Kodierung

- ▶ Q, Σ, Γ und $\{L, R, N\}$ eindeutig (dezimal) numerieren
 - ▶ Abbildungen $Q \rightarrow \mathbb{N}, \Sigma \rightarrow \mathbb{N}, \Gamma \rightarrow \mathbb{N}$ und $\{L, R, N\} \rightarrow \mathbb{N}$
 - ▶ Unär kodieren $\#$ trennt Elemente, 00 trennt Übergänge
 - ▶ $1 \rightarrow 1, 2 \rightarrow 11, 3 \rightarrow 111, \dots$
- ▶ Effizienz irrelevant!

Beispiel: Siehe Tafel

Gödelisierung

Gödelisierung: Funktion $c : M \rightarrow \mathbb{N}$ mit folgenden Eigenschaften:

- ▶ c ist injektiv
- ▶ Bildmenge $c(M)$ ist entscheidbar
- ▶ $c : M \rightarrow \mathbb{N}$ und $c^{-1} : c(M) \rightarrow M$ sind berechenbar

Gödelisierung: Unäre Kodierung

- ▶ Q, Σ, Γ und $\{L, R, N\}$ eindeutig (dezimal) numerieren
 - ▶ Abbildungen $Q \rightarrow \mathbb{N}, \Sigma \rightarrow \mathbb{N}, \Gamma \rightarrow \mathbb{N}$ und $\{L, R, N\} \rightarrow \mathbb{N}$
 - ▶ Unär kodieren $\circ$ trennt Elemente, oo trennt Übergänge
 - ▶ $1 \rightarrow 1, 2 \rightarrow 11, 3 \rightarrow 111, \dots$
- ▶ Effizienz irrelevant!

Beispiel: Siehe Tafel

Gödelisierung

Gödelisierung: Funktion $c : M \rightarrow \mathbb{N}$ mit folgenden Eigenschaften:

- ▶ c ist injektiv
- ▶ Bildmenge $c(M)$ ist entscheidbar
- ▶ $c : M \rightarrow \mathbb{N}$ und $c^{-1} : c(M) \rightarrow M$ sind berechenbar

Funktionen vs. Mengen

- ▶ Berechenbarkeit auf Funktionen zugeschnitten
- ▶ Analogon für Sprachen? ➡ Mengen mit Funktionen verbinden

Charakteristische Funktion

Die charakteristische Funktion $\chi_A : \Sigma^* \rightarrow \{0, 1\}$ ist definiert durch

$$\chi_A(\omega) = \begin{cases} 1 & \text{für } \omega \in A \\ 0 & \text{für } \omega \notin A. \end{cases}$$

Eine Menge $A \subseteq \Sigma^*$ heißt *entscheidbar*, falls die charakteristische Funktion χ_A berechenbar ist.

Hinweis: Analoge Definition für Semi-Entscheidbarkeit.

Sei $\hat{M}$ eine beliebige TM. Dann ist M_ω definiert durch

$$M_\omega \equiv \begin{cases} M & \text{wenn } \omega \text{ Codewort von } M \\ \hat{M} & \text{sonst.} \end{cases}$$

Satz: Spezielles Halteproblem

Gegeben sei die Selbstanwendbarkeitsmenge

$$K \equiv \{\omega \in \{0, 1\}^* \mid M_\omega \text{ angesetzt auf } \omega \text{ hält}\}$$

Die Menge K ist nicht entscheidbar.

Beweis: Siehe Tafel

Definition: Reduzierbarkeit

Seien $A \subseteq \Sigma^*$ und $B \subseteq \Gamma^*$ Sprachen. Dann heißt A auf B reduzierbar ($A \leq B$), wenn es eine totale berechenbare Abbildung $f : \Sigma^* \rightarrow \Gamma^*$ gibt, so dass $\forall x \in \Sigma^*$:

$$x \in A \Leftrightarrow f(x) \in B.$$

Satz: Reduzierbarkeit und Entscheidbarkeit

Wenn $A \leq B$ und B entscheidbar ist $\Rightarrow A$ entscheidbar.

Satz: Reduzierbarkeit und Entscheidbarkeit

Wenn $A \leq B$ und B entscheidbar ist $\Rightarrow A$ entscheidbar.

Beweis:

- ▶ Es gelte $A \leq B \Rightarrow$ Abbildung f existiert
- ▶ Charakteristische Funktion χ_B berechenbar, da B entscheidbar
- ▶ Komposition $\chi_B \circ f$ berechenbar:

$$\chi_A(x) = \begin{cases} 1 & \text{für } x \in A \\ 0 & \text{für } x \notin A \end{cases} = \begin{cases} 1 & \text{für } f(x) \in B \\ 0 & \text{für } f(x) \notin B \end{cases} = \chi_B(f(x))$$

- ▶ χ_A ist berechenbar $\Rightarrow A$ entscheidbar. □

Satz: Allgemeines Halteproblem

Gegeben sei die Sprache

$$H \equiv \{w\#x \mid M_w \text{ angewandt auf } x \text{ hält} \}$$

H ist nicht entscheidbar.

Beweis:

- ▶ Anwendung Kontraposition Reduzierbarkeitslemma
- ▶ Wenn $K \leq H$: K unentscheidbar $\Rightarrow H$ unentscheidbar
- ▶ Wähle $f(w) = w\#w : w \in K \Leftrightarrow f(w) \in H$

□

Satz von Rice

Sei $\mathcal{R}$ die Klasse aller Turing-berechenbaren Funktionen. Sei $\mathcal{S} \subset \mathcal{R}$, $\mathcal{S} \neq \emptyset$. Dann ist die Sprache

$$C(\mathcal{S}) = \{\omega \mid \text{Die von } M_\omega \text{ berechnete Funktion liegt in } \mathcal{S}\}$$

nicht berechenbar.

Beweis: Siehe Tafel

Satz von Rice

Sei E eine nicht-triviale funktionale Eigenschaft von Turing-Maschinen, und sei M eine Turing-Maschine. Dann ist das Problem »Besitzt M die Eigenschaft E ?« nicht entscheidbar.

Eigenschaften E

- ▶ Nicht-trivial: Mindestens eine Maschine besitzt Eigenschaft *nicht*
- ▶ Funktional: Relevante Eigenschaft der Maschine
 - ▶ *Nicht gültig*: Gelbe TM, freundliche TM, Überschall-TM, ...
 - ▶ *Gültig*: TM berechnet durch 2 teilbare Zahl, TM gibt mindestens zweimal gleiches Zeichen aus, ...

Beweis: Siehe Tafel

Implikationen

- ▶ Halteproblem/Satz von Rice gelten für alle Turing-vollständigen Berechnungsmechanismen!
- ▶ Praxisrelevante nicht-entscheidbare Probleme
 - ▶ Statische Verifikation
 - ▶ Optimierende Compiler
 - ▶ Korrektheitsbeweise

PCP (Post's Correspondence Problem)

Gegeben sei eine endliche Menge von Wortpaaren

$$(x_1, y_1), \dots, (x_n, y_n)$$

mit $x_i, y_i \in \Sigma^+$. Gibt es eine Indexfolge $i_1, i_2, \dots, i_k \in \{1, 2, \dots, n\}$, so dass gilt

$$x_{i_1} x_{i_2} \dots x_{i_k} = y_{i_1} y_{i_2} \dots y_{i_k}?$$

Beispiel: Siehe Tafel

PCP (Post's Correspondence Problem)

Gegeben sei eine endliche Menge von Wortpaaren

$$(x_1, y_1), \dots, (x_n, y_n)$$

mit $x_i, y_i \in \Sigma^+$. Gibt es eine Indexfolge $i_1, i_2, \dots, i_k \in \{1, 2, \dots, n\}$, so dass gilt

$$x_{i_1} x_{i_2} \dots x_{i_k} = y_{i_1} y_{i_2} \dots y_{i_k}?$$

Beweis (Struktur)

- ▶ Für jede Turing-Maschine eine PCP-Instanz konstruieren
- ▶ Verbindung zu haltenden Maschinen herstellen
- ▶ Halteproblem durch PCP lösen
- ▶ Widerspruch $\Rightarrow$ PCP nicht entscheidbar

Folgerungen (Reduktionen) aus dem PCP

Folgende Probleme sind unentscheidbar:

- ▶ Schnittproblem für kontextfreie Sprachen
- ▶ Mehrdeutigkeitsproblem für kontextfreie Grammatiken
- ▶ Leerheitsproblem für kontextsensitive Sprachen

Beweis: Siehe Tafel

1. Überblick und Einführung

- 1.1 Administrativa
- 1.2 Warum Theorie?

2. Formale Sprachen und Automaten

- 2.1 Endliche Automaten
- 2.2 Reguläre Sprachen I
- 2.3 Worterzeugung und Grammatiken
- 2.4 Chomsky-Hierarchie
- 2.5 Reguläre Sprachen und endliche Automaten
- 2.6 Nicht-Deterministische

endliche Automaten

- 2.7 Grammatiken und NEAs
- 2.8 Äquivalenz von NEAs und DEAs
- 2.9 Reguläre Ausdrücke
- 2.10 Das Pumping-Lemma
- 2.11 Automatenminimierung
- 2.12 Abschlusseigenschaften regulärer Sprachen
- 2.13 Kontextfreie Sprachen
- 2.14 Kellerautomaten
- 2.15 CYK-Algorithmus

3. Berechenbarkeitstheorie

- 3.1 Turing-Maschinen

- 3.2 LBAs und der Satz von Kuroda
- 3.3 Berechenbarkeit und Church-Turing-These
- 3.4 Varianten von Turing-Maschinen
- 3.5 Berechnungskomplexität
- 3.6 Alternative Berechnungsmodelle
- 3.7 Universelle Turing-Maschinen
- 3.8 Das Halteproblem

4. Komplexitätstheorie

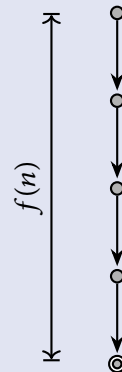
- 4.1 Definitionen
- 4.2 Komplexitätsklassen
- 4.3 Struktur von NP

Komplexitätstheorie

Definition: Laufzeitmessung für DTM

- ▶ Gegeben: DTM M , darauf entscheidbare Sprache L .
Länge der Eingabe: $n \equiv |w|, w \in L$.
- ▶ Laufzeit bzw. Zeitkomplexität: $f : \mathbb{N} \rightarrow \mathbb{N}$, f gibt maximale Anzahl Schritte von M an.
- ▶ Äquivalente Aussagen
 - ▶ » M läuft in in Zeit $f(n)$.«
 - ▶ » M ist eine $f(n)$ -Turing-Maschine.«

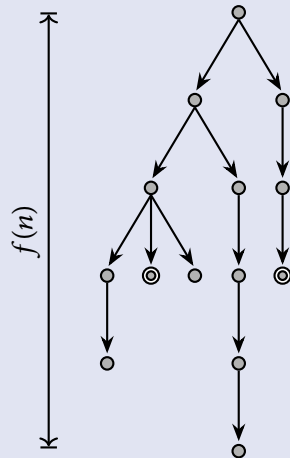
Illustration



Definition: Laufzeitmessung für NTM

- ▶ Gegeben: NTM M , darauf entscheidbare Sprache L .
Länge der Eingabe: $n \equiv |w|, w \in L$.
- ▶ Laufzeit bzw. Zeitkomplexität: $f : \mathbb{N} \rightarrow \mathbb{N}$, f gibt maximale Anzahl Schritte von M an.
- ▶ Äquivalente Aussagen
 - ▶ » M läuft in in Zeit $f(n)$.«
 - ▶ » M ist eine $f(n)$ -Turing-Maschine.«

Illustration



Probleme

- ▶ Exakte Laufzeitbestimmung: Komplexe Ausdrücke.
- ▶ »Einmaleffekte«: Initialisierung, Buchhaltung, ...
- ▶ Konstanten können (je nach Ressourcenmodell) von HW abhängen.
- ▶ Statische Beiträge irrelevant für große Eingaben.

Asymptotik

- ▶ $f(n) = 6n^3 + 2n^2 + 20n + 45$
- ▶ $f(1000) = 6 \times 10^9 + 2 \times 10^6 + 20 \times 1000 + 45$
- ▶ Quadratischer Term $\approx$ 0.001 kubischer Term
- ▶ Wesentlich: $f(n) = \mathcal{O}(n^3)$
- ▶ Vorsicht bei »=«! Keine Gleichheit im üblichen Sinn.

Definition: Landau-Symbol

Gegeben $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$. Man sagt $f(n) = \mathcal{O}(g(n))$, wenn $\exists c, n_0 \in \mathbb{N}^+$, so dass $\forall n \geq n_0$

$$f(n) \leq c \cdot g(n).$$

Asymptotische Notation (zur Referenz)

Bezeichnung	$C = \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$	Grob gesprochen
$f(n) = \mathcal{O}(g(n))$	$C < \infty$	$f \leq g$
$f(n) = \Omega(g(n))$	$C > 0$	$f \geq g$
$f(n) = \Theta(g(n))$	$0 < C < \infty$	$f = g$
$f(n) = o(g(n))$	$C = 0$	$f < g$
$f(n) = \omega(g(n))$	$C = \infty$	$f > g$

Beispiel: Siehe Tafel

Komplexitätsklassen

- ▶ Gruppierung vergleichbar schwieriger Algorithmen.
- ▶ Üblicherweise: Zeit- und Platzbedarf, andere Ressourcen als Maß möglich.

Zeitkomplexitätsklasse TIME

Sei $t : \mathbb{N} \rightarrow \mathbb{R}^+$. Die Zeitkomplexitätsklasse $\text{TIME}(t(n))$ enthält alle Sprachen, die von einer $\mathcal{O}(t(n))$ -DTM entschieden werden.

Polynome als Klassifikationsfunktion

- ▶ Detailunterschiede bei Berechnungsmodellen: Irrelevant
- ▶ Für $t(n)$ mit $t(n) \geq n$ gibt es für jede $\mathcal{O}(t(n))$ -Mehrband-TM eine gleichwertige $\mathcal{O}(t^2(n))$ -Einband-TM).

Komplexitätsklassen

- ▶ Gruppierung vergleichbar schwieriger Algorithmen.
- ▶ Üblicherweise: Zeit- und Platzbedarf, andere Ressourcen als Maß möglich.

Zeitkomplexitätsklasse TIME

Sei $t : \mathbb{N} \rightarrow \mathbb{R}^+$. Die Zeitkomplexitätsklasse $\text{TIME}(t(n))$ enthält alle Sprachen, die von einer $\mathcal{O}(t(n))$ -DTM entschieden werden.

Polynome als Klassifikationsfunktion

- ▶ Detailunterschiede bei Berechnungsmodellen: Irrelevant
- ▶ Für $t(n)$ mit $t(n) \geq n$ gibt es für jede $\mathcal{O}(t(n))$ -Mehrband-TM eine gleichwertige $\mathcal{O}(t^2(n))$ -Einband-TM).

Komplexitätsklassen

- ▶ Gruppierung vergleichbar schwieriger Algorithmen.
- ▶ Üblicherweise: Zeit- und Platzbedarf, andere Ressourcen als Maß möglich.

Zeitkomplexitätsklasse TIME

Sei $t : \mathbb{N} \rightarrow \mathbb{R}^+$. Die Zeitkomplexitätsklasse $\text{TIME}(t(n))$ enthält alle Sprachen, die von einer $\mathcal{O}(t(n))$ -DTM entschieden werden.

Polynome als Klassifikationsfunktion

- ▶ Detailunterschiede bei Berechnungsmodellen: Irrelevant
- ▶ Für $t(n)$ mit $t(n) \geq n$ gibt es für jede $\mathcal{O}(t(n))$ -Mehrband-TM eine gleichwertige $\mathcal{O}(t^2(n))$ -Einband-TM).

Definition: Komplexitätsklasse P

$$P \equiv \bigcup_k \text{TIME}(n^k)$$

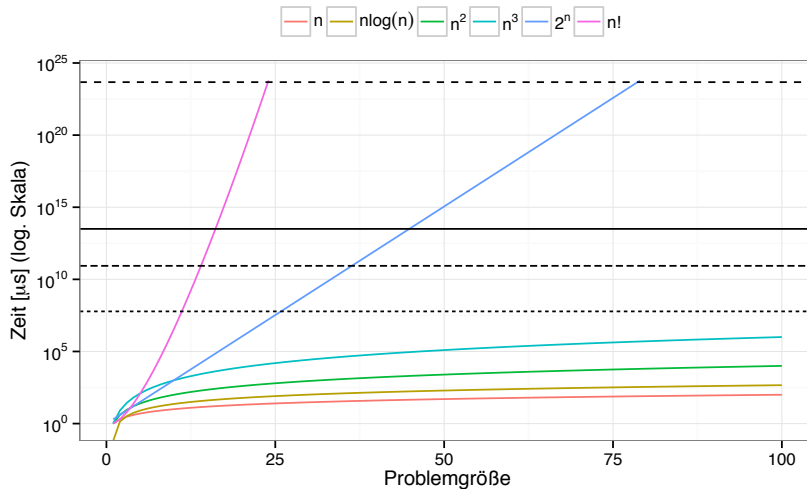
Eigenschaften

- ▶ Invariant gegenüber polynomialen Beschleunigungen/Verlangsamungen.
- ▶ Klasse der praktisch lösbaren Probleme (keine strikte Trennschärfe!).
- ▶ Enthält $\text{TIME}(1)$, $\text{TIME}(\log n)$, $\text{TIME}(n \log n)$, ...

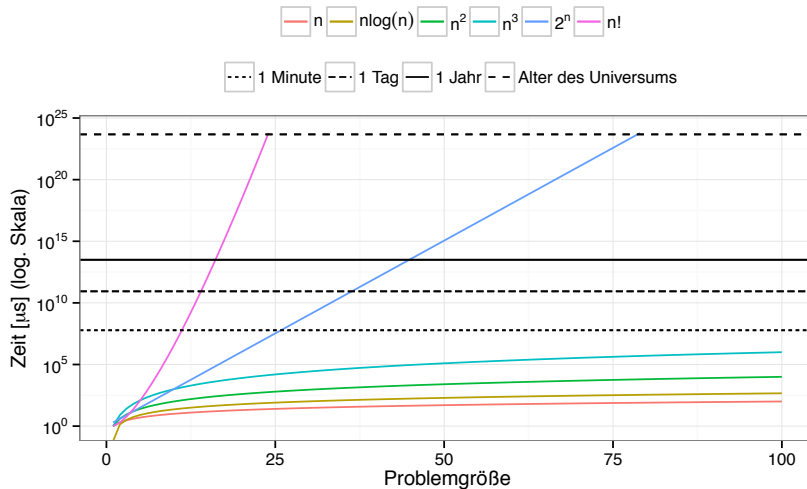
Eigenschaften

- ▶ Fast Fourier-Transformation
- ▶ Sortieralgorithmen (Merge Sort, Quicksort, Bubblesort, ...)
- ▶ Binäre Suche
- ▶ Primzahl-Test
- ▶ Kontextfreie Sprachen
- ▶ ...

Warum polynomiale Skalierung?



Warum polynomiale Skalierung?



P und NP

$$P \equiv \bigcup_k \text{TIME}(n^k) \quad (1)$$

$$NP \equiv \bigcup_k \text{NTIME}(n^k) \quad (2)$$

Interpretation von NP

- ▶ NP bedeutet *nicht* nicht-polynomial!
- ▶ NTIME: Äquivalent von TIME für nicht-deterministische Turingmaschine
- ▶ Achtung: Enthaltensein in NP bedeutet nicht notwendigerweise, dass kein effizienter Algorithmus *existiert*.

P und NP

$$P \equiv \bigcup_k \text{TIME}(n^k) \quad (1)$$

$$NP \equiv \bigcup_k \text{NTIME}(n^k) \quad (2)$$

Interpretation von NP

- ▶ NP bedeutet *nicht* nicht-polynomial!
- ▶ NTIME: Äquivalent von TIME für nicht-deterministische Turingmaschine
- ▶ Achtung: Enthaltensein in NP bedeutet nicht notwendigerweise, dass kein effizienter Algorithmus *existiert*.

Alternative Definition von NP (leicht vereinfacht)

Eine Problem ist in NP, wenn die Korrektheit der Lösung in polynomialer Zeit auf einer DTM überprüft werden kann.

1. Lösung effizient auf NTM raten
2. Korrektheit effizient auf DTM überprüfen

Beweis: Siehe Vorlesung »Moderne theoretische Informatik«

Beispiel: COMPOSITES

$$A \equiv \{x \mid x = p \cdot q; q, p \in \mathbb{N} \wedge p, q > 1\}$$

- ▶ *Schwierig*: Zwei Faktoren p, q finden
- ▶ *Einfach*: $p \cdot q$ berechnen und Resultat mit x vergleichen

Struktur von NP

- ▶ Es gibt besonders schwere Probleme in NP
- ▶ Alle anderen Probleme in NP sind auf besonders schwere Varianten effizient zurückführbar

Grundproblem Reduzierbarkeit

- ▶ Gegeben: Problem B mit Lösung, Problem A .
- ▶ Gesucht: Methode, die mit Lösung von B Lösung von A ermittelt.
- ▶ Vorgehensweise:
 - ▶ Eingabe von A auf Eingabe von B abbilden.
 - ▶ Sicherstellen, dass »Ja« (»Nein«)-Instanzen auf »Ja« (»Nein«)-Instanzen abgebildet werden
- ▶ Abbildung muss *effizient* zu berechnen sein!

Definition: Polynomial Zeit-berechenbare Funktion

$f : \Sigma^* \rightarrow \Sigma^*$ ist polynomial Zeit-berechenbar, wenn eine polynomial-Zeit DTM M existiert, die für jede Eingabe ω mit $f(\omega)$ als Bandinhalt hält.

Grundproblem Reduzierbarkeit

- ▶ Gegeben: Problem B mit Lösung, Problem A .
- ▶ Gesucht: Methode, die mit Lösung von B Lösung von A ermittelt.
- ▶ Vorgehensweise:
 - ▶ Eingabe von A auf Eingabe von B abbilden.
 - ▶ Sicherstellen, dass »Ja« (»Nein«)-Instanzen auf »Ja« (»Nein«)-Instanzen abgebildet werden
- ▶ Abbildung muss *effizient* zu berechnen sein!

Definition: Polynomial Zeit-berechenbare Funktion

$f : \Sigma^* \rightarrow \Sigma^*$ ist polynomial Zeit-berechenbar, wenn eine polynomial-Zeit DTM M existiert, die für jede Eingabe ω mit $f(\omega)$ als Bandinhalt hält.

Definition: Polynomial Zeit-berechenbare Funktion

$f : \Sigma^* \rightarrow \Sigma^*$ ist polynomial Zeit-berechenbar, wenn eine polynomial-Zeit DTM M existiert, die für jede Eingabe w mit $f(w)$ als Bandinhalt hält.

Definition: Polynomial-Zeit Abbildungs-Reduzierbarkeit

Sprache A ist *polynomial Zeit-Abbildungs-Reduzierbar* (*polynomial time mapping reducible, many-to-one-reducible*) auf Sprache B ,

$$A \leq_p B,$$

wenn $\exists f : \Sigma^* \rightarrow \Sigma^*$, f polynomial Zeit-berechenbar, so dass $\forall w$:

$$w \in A \Leftrightarrow f(w) \in B.$$

Satz

Wenn $A \leq_p B$ und $B \in \mathbf{P}$, dann ist $A \in \mathbf{P}$.

Algorithmus

- ▶ Gegeben: Eingabe $\omega \in A$, TM M für B .
- ▶ Berechne $f(\omega)$
- ▶ Führe M auf $f(\omega)$ aus; gibt Resultat zurück.

Anmerkungen

- ▶ Komposition zweier Polynome ist weiterhin Polynom
- ▶ Funktioniert auch für NTMs

Definition: NP-Vollständigkeit

Eine Sprache B ist NP-vollständig, wenn gilt:

- ▶ $B \in \text{NP}$
- ▶ $\forall A \in \text{NP} : A \leq_p B$

NPC ist die Klasse aller NP-vollständigen Probleme.

Satz II

Wenn $B \in \text{NPC}$ und $B \leq_p C$, dann gilt

$$C \in \text{NPC}$$

Satz I

$$B \in \text{NPC} \wedge B \in \text{P} \Rightarrow \text{P} = \text{NP}$$

Satz III (Cook-Levin)

$$\text{SAT} \in \text{NPC}$$

NP-Vollständigkeit: Konsequenzen

- ▶ Alle NPC-Probleme sind gleich schwierig
- ▶ NPC-Probleme sind *nicht* in P, wenn $P \neq NP$
- ▶ $\exists$ Algorithmus in P für NP-vollständiges Problem $\Leftrightarrow P = NP$
- ▶ Starke aktuelle Vermutung:

$$NP \neq P$$

- ▶ Preisgeld für Beweis: 1 Million US-Dollar
 - ▶ Siehe *Millenium Prize*
 - ▶ Theoretische Informatik *ist* kommerziell interessant :)

Graphen

Soziale Netzwerke, Internet, Compiler, ...

- ▶ Graphhomomorphismus (Netzwerke strukturell identisch?)
- ▶ Längster Pfad
- ▶ Graphfärbbarkeit (Registerallokation)

Optimierung

Produktionsplanung, Scheduling, ...

- ▶ Bin Packing (Verteilung in Container)
- ▶ Travelling Salesman
- ▶ Quadratic Assignment (elektronische Komponenten platzieren)

Formale Sprachen

Compiler, Big Data, Maschinelles Lernen, ...

- ▶ Längste gemeinsame Teilsequenz
- ▶ String-String-Korrektur

Spiele und Puzzles

Daddeln

- ▶ Super Mario Bros (schnellstmöglicher Lauf)
- ▶ Mastermind (Lösung finden)
- ▶ Lemmings (Level lösen)

Siehe beispielsweise dimacs.rutgers.edu/~graham/pubs/papers/cormodelemmings.pdf oder arxiv.org/pdf/1203.1895v1.pdf

Konsequenzen

- ▶ Viele praktische Probleme können (sehr wahrscheinlich) nicht effizient gelöst werden
- ▶ Approximationsverfahren wichtig
 - ▶ Manche Probleme können beweisbar nicht approximiert werden
 - ▶ Tritt sehr selten auf
- ▶ Details: Siehe Vorlesung »Algorithmen und Datenstrukturen«

Problem des Handlungsreisenden

Travelling Salesman Problem (TSP)

- ▶ Gegeben: n Städte und die Entfernungen dazwischen
- ▶ Gesucht: Rundreise durch alle Städte mit minimaler Gesamtlänge
- ▶ Optimierungsproblem!

Formulierung als Entscheidungsproblem

- ▶ Gegeben: n Städte, Entfernungen dazwischen und ein Wert k
- ▶ Gesucht: Gibt es eine Rundreise durch alle Städte mit Länge $\leq k$

Zusammenhang

Entscheidungsproblem nicht in P $\Rightarrow$ Optimierungsproblem nicht in P

Erfüllbarkeit logischer Ausdrücke: SAT

- ▶ Boole'sche Variablen: $x_0, x_1, \dots, x_n$
- ▶ Verknüpfungen: *Oder* ($\vee$), *Und* ($\wedge$), *Negation* ($\neg$), Kurzschreibweise: $\bar{x}_i \equiv \neg x_i$
- ▶ Klauseln: *Disjunktion* (Oder-Verknüpfung) von Variablen und negierten Variablen. Beispiel:

$$K_1 = (x_3 \vee x_7 \vee \bar{x}_2)$$

- ▶ Boolescher Ausdruck: *Konjunktion* (Und-Verknüpfung) von Klauseln. Beispiel:

$$B = K_1 \wedge K_2 \wedge K_3 = (x_3 \vee x_7 \vee \bar{x}_2) \wedge (x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_4 \vee x_6)$$

Konjunktive Normalform

- ▶ Konjunktion von Disjunktionen: *Konjunktive Normalform* (KNF, CNF)
- ▶ Jede Klausel enthält maximal i Literale $\Rightarrow$ i -CNF

Erfüllbarkeit

- ▶ Ausdruck B erfüllt, wenn Belegung der Variablen $x_1, \dots, x_n$ existiert, die B wahr macht
- ▶ Beispiele:
 - ▶ $B = (x_0 \vee x_1 \vee x_2)$: ✓
 - ▶ $B = (x_0 \vee x_1) \wedge (\bar{x}_0 \vee x_2)$: ✓
 - ▶ $B = (x_0 \vee x_1) \wedge \bar{x}_0 \wedge \bar{x}_1$: ✗

Sprache SAT

$\text{SAT} \equiv \{K \mid K \text{ ist erfüllbarer Boole'scher Ausdruck in KNF}\}$

Praktische Anwendungen

- ▶ Konstringierte Planungsprobleme (Stunden-, Ablaufplanung, ...)
- ▶ Produktlinien in der Softwareentwicklung
- ▶ Konfigurationsoptionen-Konsistenz (bsp. Linux-Kernel)

SAT-Solver: Siehe www.satcompetition.org

Satz von Cook und Levin

Für jedes Problem $A \in \text{NP}$ gilt

$$A \leq_p \text{SAT}$$

Beweis (Struktur)

- ▶ Für jede Sprache $A \in \text{NP}$: Reduktion für Maschine $M(A)$ konstruieren
- ▶ Für jedes Wort $\omega \in A$:
 - ▶ Boole'sche Formel ϕ konstruieren
 - ▶ ϕ simuliert M mit Eingabe ω
 - ▶ Maschine akzeptiert $\Leftrightarrow$ Formel erfüllbar
 - ▶ Maschine akzeptiert nicht $\Leftrightarrow$ Keine erfüllende Belegung existiert
- ▶ Hinweis: Zahllose technische Details (siehe bsp. Sipser Theorem 7.37)

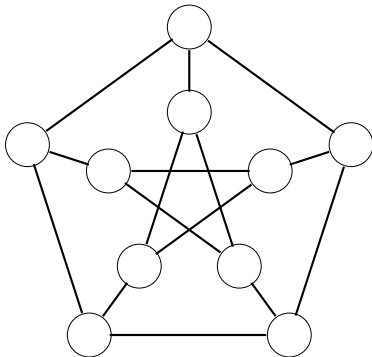
Satz von Cook und Levin

Für jedes Problem $A \in \text{NP}$ gilt

$$A \leq_p \text{SAT}$$

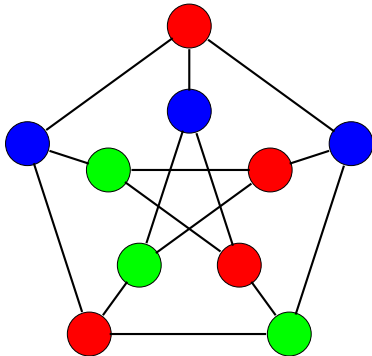
Henne und Ei

- ▶ NP-Vollständigkeit von SAT *ohne* Reduktion auf andere Probleme in NP beweisbar.
- ▶ Startpunkt für NP-Vollständigkeits-Beweise



Problemstellung

- ▶ Jeder Knoten: Eine Farbe (rot, grün, blau)
- ▶ Zwei durch Kante verbundene Knoten: Unterschiedliche Farbe
- ▶ Existiert gültige Farbzweisung?



Problemstellung

- ▶ Jeder Knoten: Eine Farbe (rot, grün, blau)
- ▶ Zwei durch Kante verbundene Knoten: Unterschiedliche Farbe
- ▶ Existiert gültige Farbzweisung?

Definition Drei-Färbbarkeit

$$\begin{aligned} 3\text{-COLOUR} = \{G = (V, E) \mid G \text{ ist ungerichteter Graph} \\ \wedge c : V \rightarrow \{r, g, b\}, \\ \forall (u, v) \in E : c(u) \neq c(v)\} \end{aligned}$$

Komplexität von 3-COLOUR

- ▶ SAT ist mindestens so schwierig wie 3-COLOUR: $3\text{-COLOUR} \leq_p \text{SAT}$
- ▶ 3-COLOUR ist NP-vollständig: $\text{SAT} \leq_p 3\text{-COLOUR}$

Bildquelle: A. Borodin, Universität Toronto

Beweis: 3-COLOUR $\leq_p$ SAT

Knotenfarbe über Variablen festlegen

$r_i = 1$: Rot, $g_i = 1$: Grün, $b_i = 1$: Blau.

1.) (Mindestens) eine Farbe pro Knoten

$$\alpha_1 = \bigwedge_{i=1}^n (r_i \vee g_i \vee b_i)$$

Beweis: 3-COLOUR $\leq_p$ SAT

Knotenfarbe über Variablen festlegen

$r_i = 1$: Rot, $g_i = 1$: Grün, $b_i = 1$: Blau.

2.) Kein Knoten hat zwei Farben

$$\begin{aligned}\alpha_2 &= \bigwedge_{i=1}^n \left[(\overline{r_i \wedge g_i}) \wedge (\overline{g_i \wedge b_i}) \wedge (\overline{r_i \wedge b_i}) \right] \\ &= \bigwedge_{i=1}^n \left[(\bar{r}_i \vee \bar{g}_i) \wedge (\bar{g}_i \vee \bar{b}_i) \wedge (\bar{r}_i \vee \bar{b}_i) \right]\end{aligned}$$

Beweis: 3-COLOUR $\leq_p$ SAT

Knotenfarbe über Variablen festlegen

$r_i = 1$: Rot, $g_i = 1$: Grün, $b_i = 1$: Blau.

3.) Knoten an gemeinsamer Kante: Verschiedene Farben

$$\begin{aligned}\alpha_3 &= \bigwedge_{(v_i, v_j \in E)} \left[(\overline{r_i} \wedge \overline{r_j}) \wedge (\overline{g_i} \wedge \overline{g_j}) \wedge (\overline{b_i} \wedge \overline{b_j}) \right] \\ &= \bigwedge_{(v_i, v_j \in E)} \left[(\bar{r}_i \vee \bar{r}_j) \wedge (\bar{g}_i \vee \bar{g}_j) \wedge (\bar{b}_i \vee \bar{b}_j) \right]\end{aligned}$$

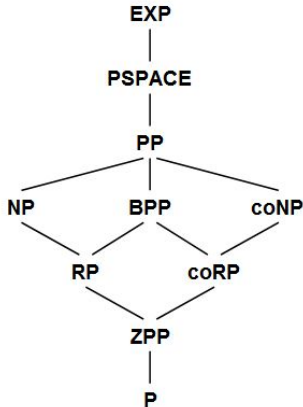
Beweis: 3-COLOUR $\leq_p$ SAT

Knotenfarbe über Variablen festlegen

$r_i = 1$: Rot, $g_i = 1$: Grün, $b_i = 1$: Blau.

Überprüfung durch SAT

- ▶ Zu überprüfender Ausdruck: $K = \alpha_1 \wedge \alpha_2 \wedge \alpha_3$
 - ▶ Erfüllbar: Färbung existiert
 - ▶ Nicht erfüllbar: Färbung existiert nicht
- ▶ ☞ 3-COLOUR ist Spezialfall von SAT!



Aktuelle Forschungsfragen

- ▶ Wie schlimm ist NP wirklich?
- ▶ Können probabilistische Maschinen mehr als deterministische Maschinen?
- ▶ Sind nicht-klassische Maschinen schneller als klassische?

