Zusammenfassung von Theoretische Informatik I

Behandelte Themen

- Sprachklassen (Chomsky Hierarchie): regulär = erkennbar = rechtslinear, deterministisch kontextfrei, kontextfrei, kontextsensitiv, Typ 0
- Automatenmodelle zur Beschreibung von Sprachen:: NEAs, DEAs, ε-NEAs, Wort-NEAs, Kellerautomaten, det. Kellerautomaten
- Andere Mechanismen, um Sprachen endlich zu beschreiben: reguläre Ausdrücke, verschiedene Arten von Grammatiken
- Eigenschaften von Sprachfamilien: Abschlusseigenschaften, Entscheidbarkeit und Komplexität von Problemen
- Konstruktionen und Beweistechniken: Potenzmengenkonstruktion, Produktautomat, Quotientenautomat, Nerode-Rechtskongruenz, verschiedene Pumping Lemmas, Normalformen von Grammatiken, etc.

Zusammenfassung Abschlusseigenschaften

	$L_1 \cap L_2$	$L_1 \cup L_2$	$\overline{L}$	$L_1 \cdot L_2$	L^*
Typ 0	✓	✓	✗	✓	✓
kontextsensitiv	✓	✓	✓	✓	✓
kontextfrei	✗	✓	✗	✓	✓
det. kontextfrei	✗	✗	✓	✗	✗
regulär	✓	✓	✓	✓	✓

Zusammenfassung Entscheidungsprobleme

	Wortproblem	Leerheitsprob..	Äquivalenzprob.
Typ 0 Grammatik	unentsch.	unentsch.	unenscht.
Typ 1 Grammatik	nicht polyzeit	unentsch.	unentsch.
Typ 2 Grammatik / PDA	polyzeit	polyzeit	unentsch.
det. PDA	linearzeit	polyzeit	entsch. [2001]
NEA / reg. Ausdruck / Typ 3 Grammatik	linearzeit	linearzeit	nicht polyzeit
DEA	linearzeit	linearzeit	polyzeit

Ausblick auf Theoretische Informatik II

Themen von Theoretische Informatik II:

- Entscheidbarkeit und Berechenbarkeit
  Welche Probleme sind algorithmisch entscheidbar und welche nicht?

  Beispiele für unentscheidbare Probleme:
  Äquivalenzproblem für PDAs, Wortproblem für Typ 0 Grammatiken

- Komplexität
  Wenn ein Problem entscheidbar / berechenbar ist, wieviel Zeit/Speicherplatz benötigt man mindestens?

  Beispiel:
  Das Äquivalenzproblem für NEAs kann man (wahrscheinlich) nicht in polynomieller Zeit entscheiden

- Sprachen und Grammatiken der Typen 0 und 1

Entscheidbarkeit / Berechenbarkeit

Wir haben die Entscheidbarkeit zahlreicher Probleme nachgewiesen, z.B.

- Wortproblem für kontextfreie Grammatiken;
- Äquivalenzproblem für DEAs.

Methode:

- Angabe eines Algorithmus in Pseudocode (z.B. CYK-Algorithmus)
- Beschreibung des Verfahrens mit Nachweis, dass alle Schritte effektiv sind:

  Berechnen von $\sim_{\mathcal{A}}$ über Folge $\sim_0, \sim_1, \ldots$ für beide Automaten

  Konstruktion der Quotientenautomaten

  Test auf Isomorphie

Genug Information für Implementierung in konkreter Programmiersprache!

Aber wie beweist man, dass für ein Problem kein Algorithmus existiert?

Dazu muss man zunächst die Frage beantworten:

Was ist ein Algorithmus?

Mögliche Antworten:

- Programmiersprachen: C, Pascal, Java, Lisp, Prolog, Assembler, etc.
- Mathematische Formalismen: Turingmaschine, Registermaschine (RAM), WHILE Programme, μ-berechenbare Funktionen, λ-Kalkül, Abstract State Machines (ASMs), etc.

Interessant: alle diese Modelle sind gleich mächtig.

Wir wählen möglichst einfaches Modell, um Beweise zu erleichtern:

Turingmaschine:

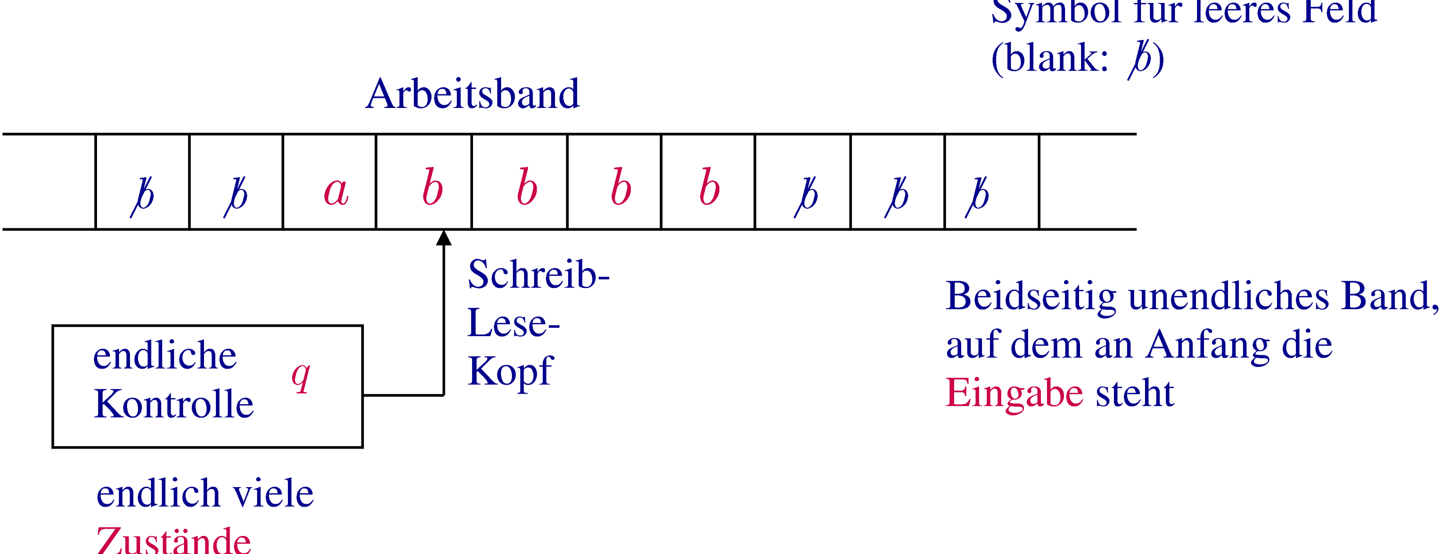
- entwickelt 1936 von Alan Turing, um Berechenbarkeit zu studieren
- endliche Kontrolle wie bei NEAs und PDAs
  + unendliches Arbeitsband (ohne Zugriffsbeschränkung von PDAs)

Church-Turing These:

> Die (intuitiv) berechenbaren Funktionen sind genau die mit Turingmaschinen (und damit auch mit Java-Programmen etc.) berechenbaren Funktionen.

These, kein Theorem, da „intuitiv berechenbar“ kein formaler Begriff ist.

Turing-Maschine



- Kopf in jedem Schritt um max. ein Feld nach links oder rechts.
- Akzeptieren/Verwerfen über Endzustände

Papadimitriou:

„It is amazing how little we need to have everything.“

Definition Turingmaschine

Eine Turingmaschine über dem Eingabealphabet Σ hat die Form $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, \Delta, F)$, wobei

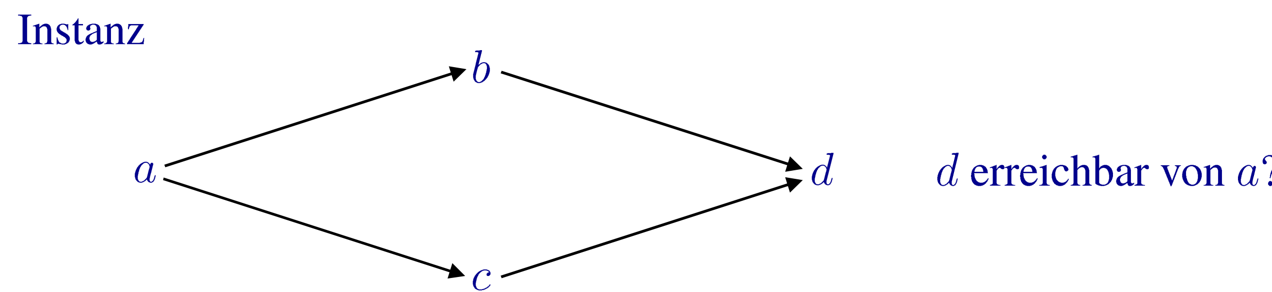
- Q endliche Zustandsmenge ist,
- Σ das Eingabealphabet ist,
- Γ das Arbeitsalphabet ist mit $\Sigma \subseteq \Gamma$, $\not{b} \in \Gamma \setminus \Sigma$,
- $q_0 \in Q$ der Anfangszustand ist,
- $F \subseteq Q$ die Endzustandsmenge ist und
- $\Delta \subseteq Q \times \Gamma \times \Gamma \times \{r, l, n\} \times Q$ die Übergangsrelation ist.

Zur formalen Definition von Entscheidbarkeit betrachten wir Entscheidungsprobleme als formale Sprachen:

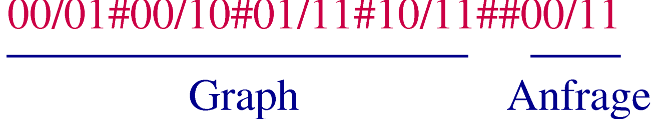
- Probleminstanz wird als Wort w dargestellt.
- $w \in L$ gdw. w eine „ja-Instanz“ ist.

Beispiel: Erreichbarkeitsproblem in gerichteten Graphen

Instanz



dargestellt als Wort



Turingmaschinen können formale Sprachen erkennen:

$$L(M) = \{w \in \Sigma^* \mid M \text{ gestartet auf } w \text{ hält in Endzustand}\}$$

Beachte: wenn $w \notin L(M)$, dann entweder

- M gestartet auf w hält an, aber nicht in Endzustand oder
- M gestartet auf w hält niemals an.

Ein Algorithmus, der ein Problem entscheidet, soll natürlich immer anhalten

Definition Entscheidbarkeit

Ein Problem $L \subseteq \Sigma^*$ heisst entscheidbar gdw. es eine Turingmaschine M gibt, die auf allen Eingaben anhält und mit $L(M) = L$.

Zusammenhang zur Chomsky-Hierarchie:

- Typ 0

  Eine Sprache ist vom Typ 0 (mit Grammatik generierbar) gdw. sie von einer Turingmaschine erkannt wird.

  TMs können also verwendet werden, um Eigenschaften von Typ 0-Sprachen zu studieren (Abschluss etc).

- Typ 1

  Eine Sprache ist vom Typ 1 (mit kontextsensitiver Grammatik generierbar) gdw. sie von einem linear beschränkten Automaten erkannt wird.

  Linear beschränkter Automat:

  Turingmaschine, die nur den von der Eingabe belegten Teil des Bandes nutzen darf

Theorem

Die Sprache $L = \{a^n b^n c^n \mid n \geq 0\}$ wird von LBA erkannt.

LBA, der L erkennt, geht wie folgt vor:

- ersetze erstes a durch a', erstes b durch b' und erstes c durch c'; prüfe dabei, ob Eingabe von der Form $a^*b^*c^*$, verwerfe wenn nicht
- laufe zurück zum zweiten a, ersetze es durch a', das zweite b durch b' und das zweite c durch c'
- dies wird solange wiederholt, bis kein a oder kein b oder kein c mehr gefunden wird; bei fehlendem b oder c: verwerfe (zu viele a's)
- bei fehlendem a prüfe, ob das Band nur noch Symbole a', b', c' enthält, verwerfe wenn nicht (zu viele b's oder c's)
- Akzeptiere

Komplexitätstheorie

Klassifikation von Entscheidungsproblemen in Komplexitätsklassen gemäss Menge von Ressourcen, die zum Entscheiden des Problems benötigt werden.

Wichtige Komplexitätsklassen z.B.:

- P oder PTime

  Menge der Probleme, die mit polynomial zeitbeschränkter deterministischer Turingmaschine entschieden werden können

- NP

  Menge der Probleme, die mit polynomial zeitbeschränkter nicht-deterministischer Turingmaschine entschieden werden können

- PSpace

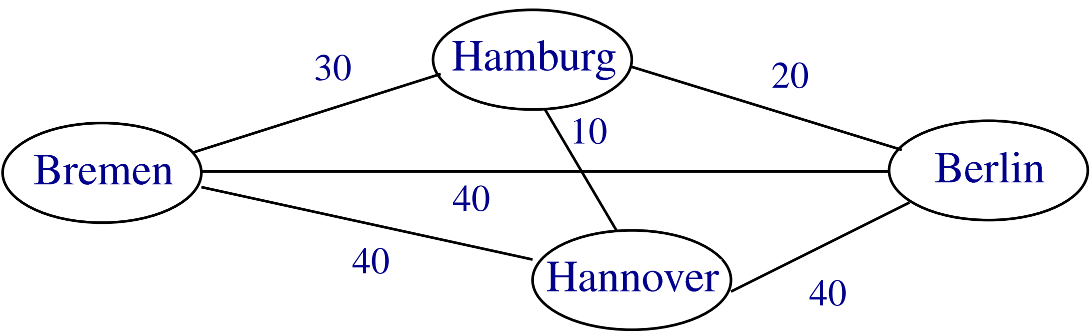
  Menge der Probleme, die mit polynomial platzbeschränkter Turingmaschine entschieden werden können

Besonders interessant ist der Zusammenhang von P und NP:

- P wird oft als Menge der effizient lösbaren Probleme betrachtet
- Es gibt sehr große Zahl natürlicher Probleme, die offensichtlich in NP sind für die aber bisher niemand beweisen konnte, dass sie nicht in P sind
- Man weiß noch nicht mal, ob $P \neq NP$ gilt (also ob nicht-deterministische TMs mächtiger sind als deterministische)
- Das ist sehr unbefriedigend!
- Man kann aber für fast alle der o.g. Probleme zeigen: wenn das Problem in P ist, dann P = NP
- Dies wird als sehr starkes Indiz dafür gewertet, dass das Problem nicht in P ist.

Beispiel: Das Travelling Salesman Problem

Eingabe: Ein Routennetz mit Kosten, z.B.:



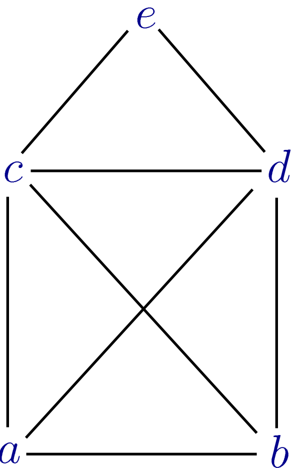
und eine Kostengrenze x, z.B. 120

Frage: Kann man von Bremen aus eine Rundreise organisieren, die alle Städte einschließt und Gesamtkosten $\leq x$ hat?

Ist ein typisches NP-vollständiges Problem:

- ist in NP, wahrscheinlich nicht in P denn dann P = NP
- hat stark kombinatorischen Charakter

Beispiel: zwei verwandte Probleme auf ungerichteten Graphen



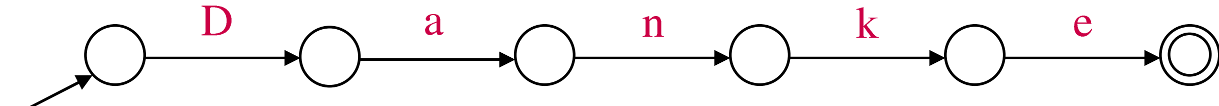
Problem 1: Eulerkreis

Gegeben ungerichteten Graph G, entscheide ob es geschlossenen Kreis in G gibt, der jede Kante genau einmal besucht

Problem 2: Hamiltonkreis

Gegeben ungerichteten Graph G, entscheide ob es geschlossenen Kreis in G gibt, der jeden Knoten genau einmal besucht

Eulerkreis ist in P, Hamiltonkreis ist NP-vollständig



für Eure Aumerksamkeit!