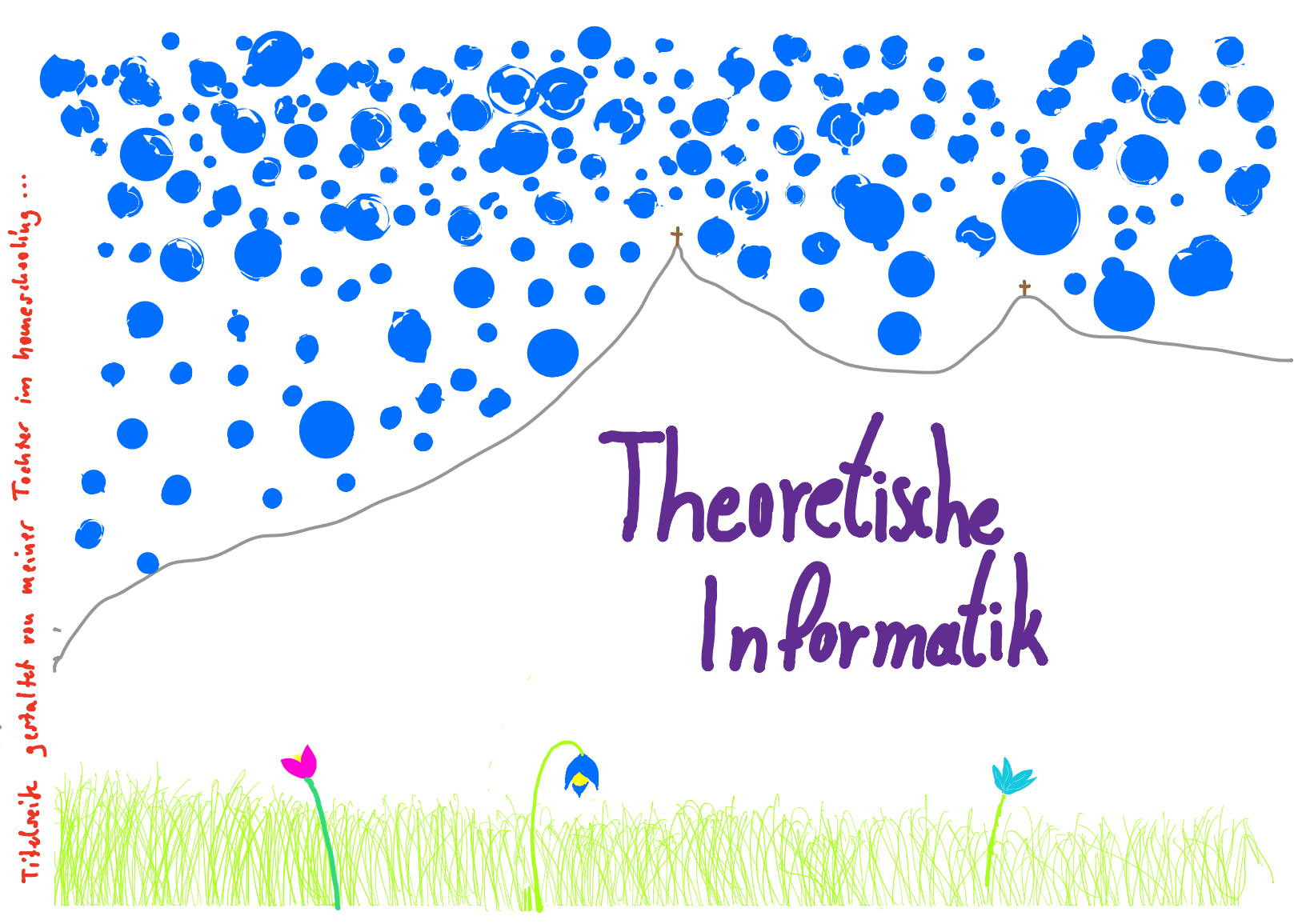

Titelbild gestaltet von meiner Tochter im Homeschooling ...
Theoretische Informatik

Theoretische Informatik:

was, warum, wozu?

Die großen Fragen der Theoretischen Informatik

- Kann ein Computer alles berechnen? Gibt es Dinge, die er nicht ausrechnen kann, prinzipiell nicht?
- Sind manche Probleme "leichter" als andere? Wie können wir das formalisieren? Was wäre ein Beispiel für ein besonders schweres Problem?
- Welche "Zutaten" sind besonders wichtig: Speicherplatz, Rechenkraft, Zufall, Parallelität, ... ?

Was sind die Grenzen der Informatik?

Historische Einordnung



David Hilbert (1862 – 1943)

- Formalisierung der Mathematik
- 23 offene Probleme (1900)

Läßt sich die Mathematik vollständig und Widerspruchsfrei durch Axiome beschreiben?

Historische Einordnung



Kurt Gödel (1906 – 1978)

1931: Gödelscher Unvollständigkeitssatz

In jedem "sinnvollen" Axiomensystem
muß es zwingend Aussagen geben,
die unentscheidbar sind.

Historische Einordnung



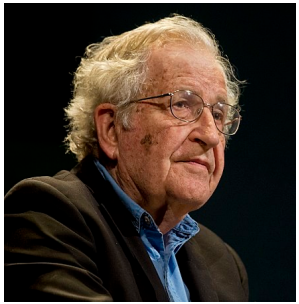
Alan Turing (1912 - 1954)

1936: Es gibt Probleme, die ein Computer niemals berechnen kann

↝ Turing-Maschine, Halbproblem

Historische Einordnung

Noam Chomsky (* 1928, Linguist)



Chomsky-Hierarchie (1956):

Axiomatische Charakterisierung von formalen Sprachen

Historische Einordnung



Stephen Cook
* 1939

1970ies:

- Cook discovers the first NP-complete problem, SAT (1971)
- Simultaneously, discovered in USSR by Levin
- Karp's list of 21 NP-complete problems



Leonid Levin
* 1948



Richard Karp, * 1935

you might know some of them from your algorithms class: knapsack, max cut, vertex cover, ...

Highlights der Vorlesung

Berechenbarkeitstheorie:

- Es gibt Funktionen, die ein Computer nicht berechnen kann. Prinzipiell nicht!
- Unvollständigkeitssatz von Gödel

Highlights der Vorlesung

Komplexitätstheorie:

- Es gibt Probleme, die (wahrscheinlich, s.u.) nicht effizient berechnet werden können.
- Das größte offene Problem der Informatik:

$$P \overset{?}{\neq} NP$$

Theoretische Informatik als Werkzeug

Automaten $\rightsquigarrow$ Rechnerarchitektur

Sprachen $\rightsquigarrow$ Programmiersprachen

Grammatiken $\rightsquigarrow$ Compiler

Komplexität $\rightsquigarrow$ Algorithmen

Wissenschaftliche Herangehensweise!

Es gibt wichtige Fragen, die zunächst nur umgangssprachlich / anschaulich formuliert werden können.

Wir müssen als erstes versuchen, diese Fragen zu formalisieren.

Dann können wir mathematische Hypothesen aufstellen, sie beweisen oder widerlegen.

Die Formalisierung ist sehr viel elementarer als ein echter Computer. Trotzdem liefern die Aussagen wichtige Erkenntnisse.

Ausblick: Grenzen der künstlichen Intelligenz?

Die Fragen und Ergebnisse, die wir in dieser Vorlesung behandeln, sind "schon alt".

Ähnliche, fundamentale Fragen stellen sich aber zur Zeit im Bereich der künstlichen Intelligenz.

Die Vorgehensweise hier ist ähnlich: Fragen stellen, auf einfache Art und Weise formalisieren, Aussagen beweisen.

Der Schritt von "Fragen stellen" zu "Formalisieren" ist oft der Knackpunkt.

Und jetzt legen wir einfach los!

Alphabete, Wörter, Sprachen

Endliche Automaten

Wollen: einfaches "Berechnungsmodell" ⇝ Automat

- Eingaben verarbeiten ⇝ Wörter und Sprachen
- etwas "berechnen" ⇝ Übergangsfunktion
- etwas speichern ⇝ Zustände
- Ausgabe produzieren

Wörter und Sprachen

Definition:

Sei Σ eine nicht-leere, endliche Menge. Wir bezeichnen sie als **Alphabet**. Die Elemente der Menge heißen **Symbole** (oder Buchstaben).

Ein **Wort** w ist eine endliche Folge von Buchstaben des Alphabets. Die Anzahl der Buchstaben heißt die **Länge des Worts** w, und wir notieren dies als $|w|$.

Manchmal schreiben wir $w = w_1 \dots w_n$ für ein Wort mit den n Buchstaben $w_1, \dots, w_n \in \Sigma$.

Das **leere Wort** ε ist ein besonderes Wort, es hat keine Buchstaben und $|\varepsilon| = 0$.

Beispiele:

- $\Sigma = \{a, \ldots, z\}$, $w = \text{informatik}$, $|w| = 10$
- $\Sigma = \{0, 1\}$, $w = 00110$, $|w| = 5$

Weiter in der Definition:

Die Menge aller Wörter der Länge n über Σ bezeichnen wir mit Σ^n. Die Menge aller Wörter über Σ bezeichnen wir mit

$$\Sigma^* := \bigcup_{n=0}^{\infty} \Sigma^n$$ (wobei $\Sigma^0 = \{\varepsilon\}$ ist). Σ^* heißt die Kleene'sche Hülle. Zusätzlich benutzen wir manchmal die Notation

$$\Sigma^+ := \bigcup_{n=1}^{\infty} \Sigma^n .$$

Eine Sprache L über Σ ist eine Teilmenge von Σ^*.

Def Die Konkatenation (Aneinanderhängung) zweier Wörter u und w besteht einfach darin, daß wir die Wörter hintereinander hängen.

Beispiel: $w = 010$, $u = 001$,

$wu = \underbrace{010}_{w}\underbrace{001}_{u}$, $uw = \underbrace{001}_{u}\underbrace{010}_{w}$

Die Konkatenation zweier Sprachen L_1, L_2 ist definiert durch $L_1 \circ L_2 := \{ uw \mid u \in L_1, w \in L_2 \}$. Die Kleene'sche Hülle einer Sprache L ist definiert als

$L^* := \{ \underbrace{x_1 \dots x_k}_{\text{Konkatenation von } k \text{ Wörtern}} \mid x_i \in L, k \in \mathbb{N}_0 \}$

Weiter in der Definition:

Wenn wir ein Wort k mal an sich selber anhängen, schreiben wir auch $w^k = \underbrace{w \dots w}_{k\text{-mal}}$

Beispiel: $w = 01$, $w^3 = 01\,01\,01$

Beispiele:

- $\Sigma = \{a, b\}$, $\Sigma^2 = \{aa, ab, ba, bb\}$
- $\Sigma = \{a, \dots, z, A, \dots, Z\}$, deutsche Sprache
- $\Sigma = \{a, b\}$, $L = \{ab, (ab)^2, (ab)^3, \dots\}$ — ababab
- $\Sigma = \{0, 1\}$, $L = \{\omega \mid \omega = 0^n 1^n \text{ für ein } n \in \mathbb{N}\}$

↓ Sprache mit unendlich vielen Worten

Bemerkung: algebraisch betrachtet hat die Operation der Konkatenation auf der Menge Σ^* die Eigenschaften:

- assoziativ $a(bc) = (ab)c$
- nicht kommutativ $ab \neq ba$
- es gibt ein neutrales Element ε $\varepsilon a = a\varepsilon = a$
- es gibt kein inverses Element

typo corrected

Weiter in der Def:

Ein Wort x heißt Teilwort eines Wortes y falls es Worte u und v gibt so daß $y = uxv$.

Falls $u = \varepsilon$, heißt x Präfix von y.

Falls $v = \varepsilon$, heißt x Suffix von y.

Beispiele:
- 01 ist Teilwort von 00111.
- 10 ist Präfix von 10 0011
- 011 ist Suffix von 001 011

Endliche (deterministische) Automaten

Literatur: Sipser Kap. 1
(Schöning Kap. 1.2,
Hromkovic Kap. 3)

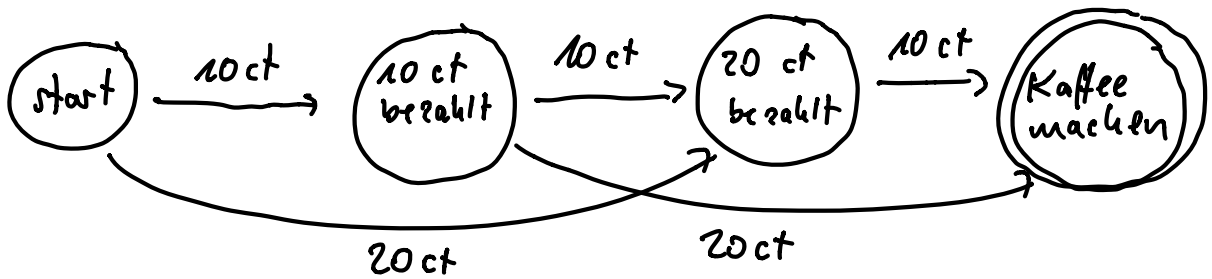
Endliche Automaten

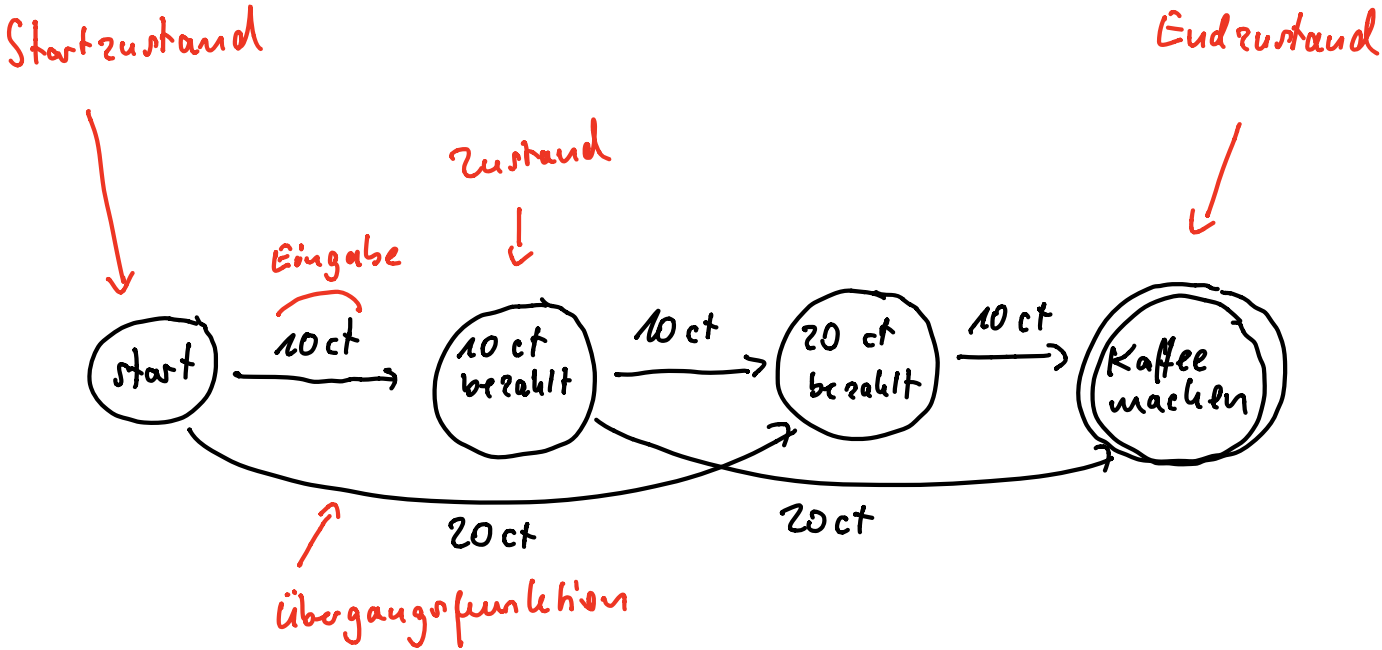
Wollen: einfaches "Berechnungsmodell" ⇝ Automat

- Eingaben verarbeiten ⇝ Wörter und Sprachen
- etwas "berechnen" ⇝ Übergangsfunktion
- etwas speichern ⇝ Zustände
- Ausgabe produzieren

Beispiel: Kaffeemaschine in der Mensa

- nimmt 10 ct und 20 ct - Münzen
- wenn man 30 ct eingeworfen hat, macht die Maschine einen Kaffee.



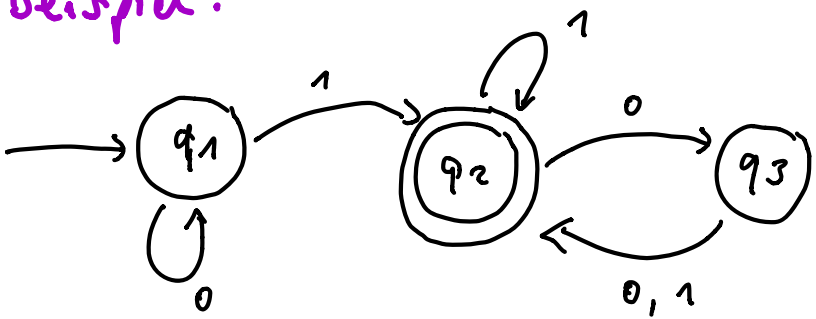

Startzustand
Endzustand
Zustand
Eingabe
start
10 ct
10 ct bezahlt
10 ct
20 ct bezahlt
10 ct
Kaffee machen
20 ct
20 ct
Übergangsfunktion

Formale Definition:

Ein (endlicher, deterministischer) Automat wird beschrieben durch ein 5-Tupel $(Q, \Sigma, \delta, q_0, F)$, wobei

- Q ist eine endliche Menge, genannt Zustände
- Σ ist eine endliche Menge, das Alphabet
- $\delta: Q \times \Sigma \to Q$ die Übergangsfunktion
- $q_0 \in Q$ der Startzustand
- $F \subseteq Q$ die Menge der akzeptierenden Zustände

Beispiel:



Zustandsdiagramm.
Die akzeptierenden Zustände werden mit zwei Kreisen gekennzeichnet, der Startzustand mit einem Pfeil.

$Q = \{q_1, q_2, q_3\}$, $\Sigma = \{0, 1\}$, q_1 Startzustand, $F = \{q_2\}$

δ gegeben wie in folgender Tabelle:

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

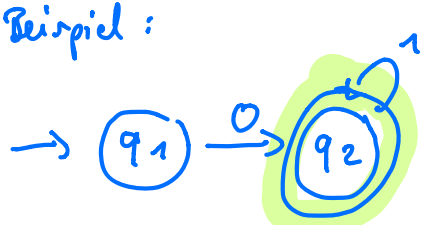
⚠ Partielle vs. totale Ü-Funktionen

Manche Autoren erlauben Übergangsfunktionen, die nur partiell definiert sind (d.h. nicht alle möglichen Übergänge sind definiert). Bei uns sind immer alle Übergänge definiert.

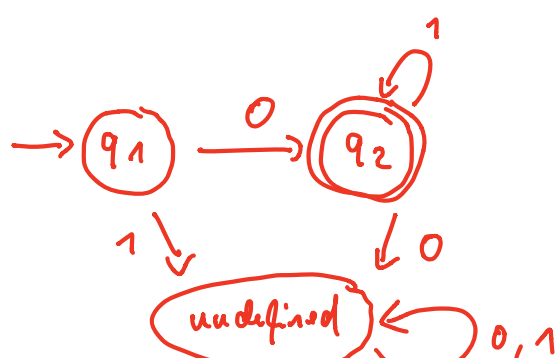
Man kann von einem zum anderen gelangen, indem man einen neuen Zustand "undefined" einführt:

Tippfehler: zweiter Kreis hat gefehlt

Beispiel:



$\hat{=}$



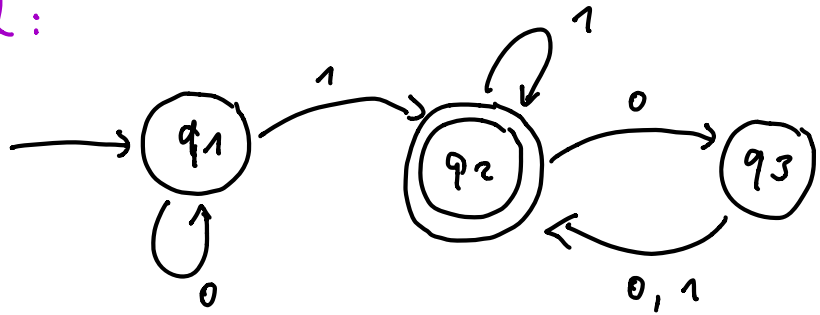
Berechnung eines Automaten

Eine Folge $s_0, \dots, s_n \in Q$ von Zuständen heißt **Berechnung** des Automaten $M = (Q, \Sigma, \delta, q_0, F)$ auf dem Wort $w = w_1 \dots w_n$ falls

- $s_0 = q_0$
- $\forall i = 0, \dots, n-1: \; s_{i+1} = \delta(s_i, w_{i+1})$

Intuition: Berechnung = eine "gültige" Folge von Zuständen, die man durch Abarbeiten des Worts w erreicht.

Beispiel:



- $w = 001$ ergibt die Zustandsfolge $q_1\, q_1\, q_1\, q_2$:

$$q_1 \xrightarrow{0} q_1 \xrightarrow{0} q_1 \xrightarrow{1} q_2 .$$

- $w = 010$

$$q_1 \xrightarrow{0} q_1 \xrightarrow{1} q_2 \xrightarrow{0} q_3$$

Weiter in der Definition:

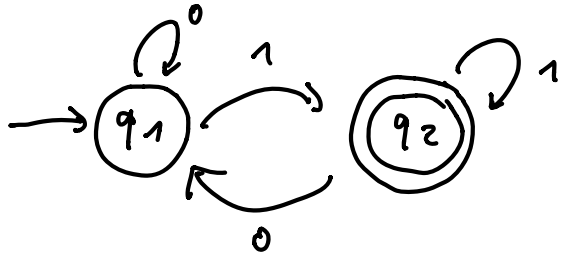
Eine Berechnung akzeptiert das Wort w, falls die Berechnung in einem akzeptierenden Zustand endet.

Die von einem endl. Automaten M (erkannte) akzeptierte Sprache $L(M)$ ist die Menge der Wörter, die von M akzeptiert werden:

$$L(M) := \{ w \in \Sigma^* \mid M \text{ akzeptiert } w \}$$

⚠ Eine Berechnung kann mehrmals in akzeptierende Zustände eintreten und wieder austreten. Wichtig ist der Endzustand, nachdem der letzte Buchstabe des Eingabewortes verarbeitet worden ist.

Beispiel:

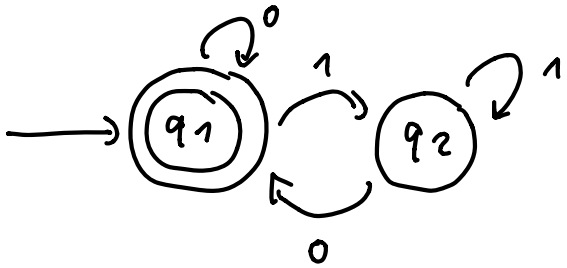


- Wort 1101 $\rightsquigarrow$ $q_1 q_1 q_2 q_2 q_1 q_1 q_2$ $\rightsquigarrow$ w wird akzeptiert
- $w = 010$ $\rightsquigarrow$ $q_1 q_1 q_1 q_2 q_1$ $\rightsquigarrow$ w wird nicht akzeptiert

Mit etwas Herumspielen findet man heraus:

$$L(M) = \{ w \in \Sigma^* \mid \text{letzter Buchstabe von } w \text{ ist } 1 \}$$

Beispiel:



Wie vorher, nur ist nun q_1 der akzeptierende Zustand.

Daher ergibt sich:

$$L(M) = \{w \in \Sigma^* \mid w = \varepsilon \text{ oder } w \text{ endet mit } 0\}.$$

Erweiterte Übergangsfunktion

Im folgenden wird es oft praktisch sein, die Übergangsfunktion auch für Wörter als Eingabe (statt nur Buchstaben) zu definieren:

Def Sei $\delta: Q \times \Sigma \to Q$ eine Übergangsfunktion. Wir definieren die erweiterte Übergangsfunktion $\delta^*: Q \times \Sigma^* \to Q$ induktiv wie folgt:

- $\delta^*(q, \varepsilon) = q$ für alle $q \in Q$
- Für $w \in \Sigma^*$, $a \in \Sigma$ setzen wir

$$\delta^*(q, wa) = \delta\left(\delta^*(q, w), a\right)$$

Zustand nach Lesen von w

nun lesen wir noch Buchstaben a

(falls $\delta^*(q, w)$ definiert ist; falls nicht, setzen wir die Funktion auf "undefined")

Reguläre Sprachen und erste Abschlußeigenschaften

Literatur: Sipser Kap. 1
(Schöning Kap. 1.2,
Hromkovic Kap. 3)

Reguläre Sprachen

Def Eine Sprache $L \subset \Sigma^*$ heißt **reguläre Sprache**, wenn es einen endlichen Automaten M gibt, der diese Sprache akzeptiert.

Die Menge aller regulären Sprachen bezeichnen wir als **REG**.

Können wir reguläre Sprachen irgendwie besser beschreiben, charakterisieren, verstehen?

- Wie sehen Sprachen aus, die von endlichen Automaten akzeptiert werden?
- Was für Eigenschaften haben sie?
- Gibt es Sprachen, die <u>nicht</u> von einem endlichen Automaten akzeptiert werden?
- Kann man die Sprachen geschickt beschreiben?

Erste Beobachtung:

Endliche Sprachen sind immer regulär.

Sehen Sie, wie Sie das beweisen könnten?

Nun wollen wir verschiedene Eigenschaften von regulären Sprachen ausnutzen.

Abgeschlossenheit bzgl. Komplementbildung

Satz: Sei L eine reguläre Sprache über Σ.
Dann ist auch $\bar{L} := \Sigma^* \setminus L$ eine reguläre Sprache.



Beweisidee (informell)

Vertausche akzeptierende und nicht akzeptierende Zustände des Automaten.

(man muß hier aufpassen, falls die Ü-Funktionen nur partiell definiert war, aber bei uns ist sie immer total definiert)

Beweis (formal):

L regulär $\Rightarrow$ es gibt Automaten $M = (Q, \Sigma, \delta, q_0, F)$, der L akzeptiert.

Definiere "Komplementautomat" $\overline{M} := (Q, \Sigma, \delta, q_0, \overline{F})$ mit

$\overline{F} := Q \setminus F$.

Dann gilt:

$w \in \overline{L} \Leftrightarrow M$ akzeptiert w nicht

$\Leftrightarrow \overline{M}$ akzeptiert w .

$\blacksquare$ Satz

Abgeschlossenheit bzgl. Vereinigung

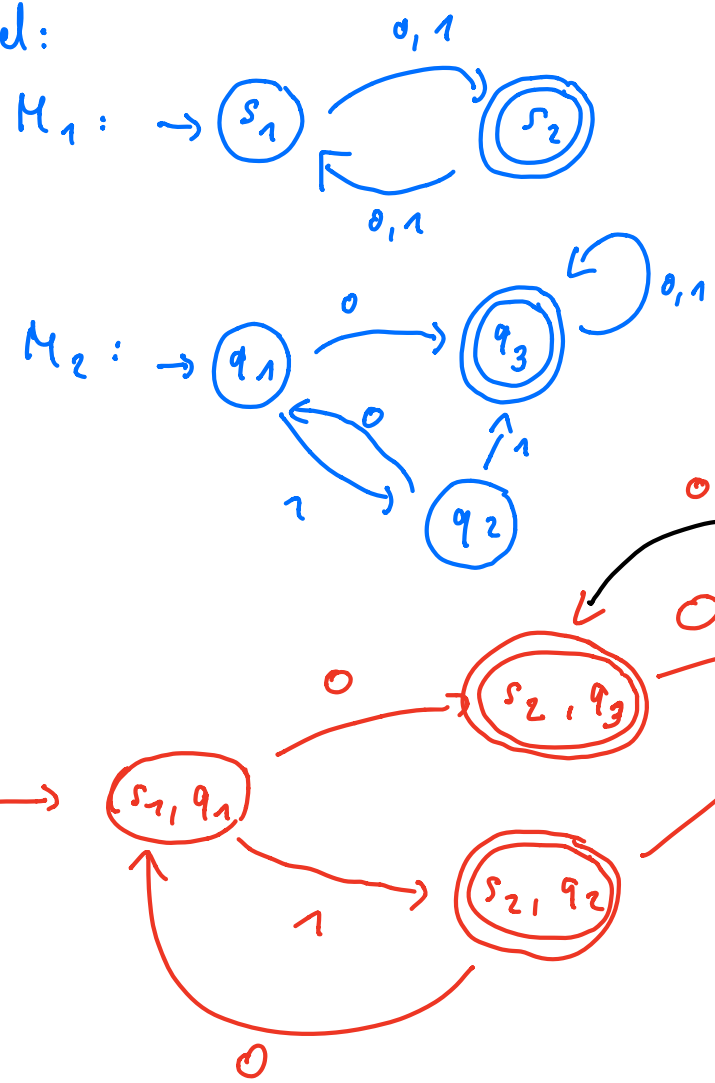
Satz: Die Menge der regulären Sprachen ist abgeschlossen bezüglich Vereinigung:

$$L_1, L_2 \in REG \Rightarrow L_1 \cup L_2 \in REG.$$

Beweisidee (informell!):

- Da $L_1, L_2 \in REG$, gibt es endliche Automaten M_1, M_2, die L_1 bzw. L_2 erkennen.
- Wenn wir nun ein Wort w als Eingabe hinlegen, müssen wir gleichzeitig M_1 und M_2 laufen lassen
  (hintereinander geht nicht, warum?.)
- Dazu konstruieren wir den "Produktautomaten", dessen Zustände immer aus einem Paar bestehen: Zustand, in dem M_1 gerade wäre, und Zustand, in dem M_2 wäre.

Beispiel:



Zustand akzeptiert falls einer der beiden "inneren" Zustände akzeptiert

Also z. B.

- $0 \in L_1 \cup L_2$
- $01 \in L_1 \cup L_2$
- $10 \notin L_1 \cup L_2$

Beweis des Satzes, formal:

Sei $M_1 = (Q_1, \Sigma_1, \delta_1, s_1, F_1)$ ein Automat, der L_1 erkennt.

$M_2 = \ldots$ L_2

Schritt 1: Konstruktion des Produktautomaten

Wir definieren den Produktautomaten $M := M_1 \times M_2$ wie folgt:

$M = (Q, \Sigma, \Delta, s, F)$ mit $Q = Q_1 \times Q_2$, $\Sigma = \Sigma_1 \cup \Sigma_2$,

neuer Startzustand: $s = (s_1, s_2)$, $F = \{(f_1, f_2) \mid f_1 \in F_1 \text{ or } f_2 \in F_2\}$,

Typo corrected

neue Übergangsfunktion: $\Delta : Q \times \Sigma \to Q$

$$\Delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a))$$

($r_1 \in Q_1$, $r_2 \in Q_2$, $a \in \Sigma$)



Δ könnte nur partiell definiert sein.

Schritt 2: Die Definition überträgt sich auf erweiterte Ü-Funktionen:

Beh: Es gilt für alle $q_1 \in Q_1$, $q_2 \in Q_2$, $w \in \Sigma^*$:

$$(\#) \quad \Delta^*((q_1,q_2),w) = (\delta_1^*(q_1,w), \delta_2^*(q_2,w))$$

Δ^*: neue Ü-Funktion (erweitert)

δ_1^*, δ_2^*: alte Funktionen (erweitert)

Beweis der Behauptung: Induktion nach $n := |w|$ (Wortlänge)

IA: $|w| = 0$. Dann ist $w = \varepsilon$ und nach Definition der erweiterten Ü-Funktion gilt $\Delta^*((q_1,q_2),\varepsilon) = (q_1,q_2)$, und $\delta_1^*(q_1,w) = q_1$, $\delta_2^*(q_2,w) = q_2$.

IS: $n \rightsquigarrow n+1$

Sei w' ein Wort der Länge n, $\sigma \in \Sigma$ ein Buchstabe. Wir betrachten $w := w'\sigma$.

Def von Δ^* und w

Zustand nach Lesen von w'

$$\Delta^*((q_1, q_2), w) =$$

$$= \Delta\left(\Delta^*((q_1, q_2), w'), \sigma\right)$$

Ind. Annahme

$$= \Delta\left(\left(\delta_1^*(q_1, w'), \delta_2^*(q_2, w')\right), \sigma\right)$$

Def von Δ

$$= \left(\delta_1\left(\delta_1^*(q_1, w'), \sigma\right), \delta_2\left(\delta_2^*(q_2, w'), \sigma\right)\right)$$

$$= \left(\delta_1^*(q_1, w), \delta_2^*(q_2, w)\right)$$

□ Beh.

Jetzt kommt noch der letzte Schritt zum Beweis des Satzes:

Schritt 3: Der Produktautomat akzeptiert $L_1 \cup L_2$.

- Nach Definition (von F) akzeptiert M ein Wort w, wenn M_1 oder M_2 das entsprechende Wort akzeptieren.
- In Schritt 2 haben wir gesehen, daß die Berechnung von M sich auf die Komponenten runterbrechen läßt.
- Daraus folgt die Aussage des Satzes.

□ Satz

Abgeschlossenheit bzgl. Schnitt und Mengendifferenz

Satz Seien L_1, L_2 zwei reguläre Sprachen. Dann sind auch $L_1 \cap L_2$ und $L_1 \setminus L_2$ reguläre Sprachen.

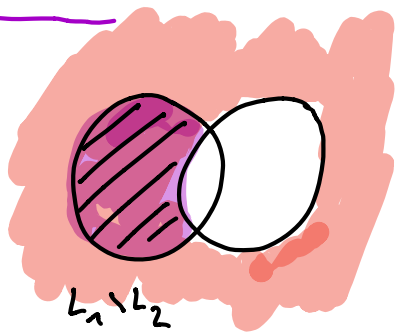
? Lesen Sie, wie die Beweise funktionieren müssen!

Beweisideen:

$L_1 \cap L_2$: Beweis funktioniert wie der für $L_1 \cup L_2$, nur definieren wir die akzeptierenden Zustände des Produktautomaten so,

$$F := \{ (q_1, q_2) \mid q_1 \in F_1 \text{ und } q_2 \in F_2 \}$$

$L_1 \setminus L_2$ $= L_1 \cap \overline{L_2}$, und wir wissen schon, daß Schnitt und Komplementbildung erlaubt sind.



Satz

Nicht-deterministische Automaten

Literatur: Sipser Kap. 1
(Schöning Kap. 1.2,
Hromkovic Kap. 3)

Nicht-deterministische Automaten

In vielen Bereichen der Informatik kann der "Zufall" ein sehr mächtiges Werkzeug sein.

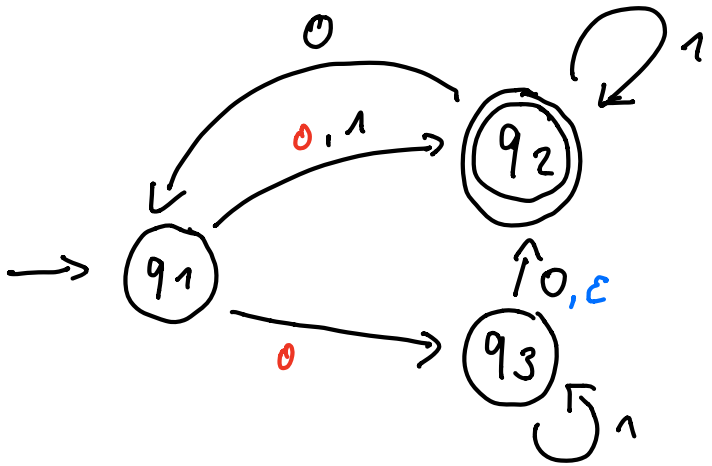
Beispiel: Randomisierte Algorithmen

Erinnern Sie sich an Quicksort? In diesem Algorithmus spielt der Zufall eine Rolle, wissen Sie noch wo?

Brauchen also auch Berechnungsmodelle für nicht-deterministische Berechnungen.

Nicht-deterministische Automaten: Idee

Bisher: Übergangsfunktion deterministisch. Jetzt wollen wir sie "nicht-deterministisch" machen und erlauben, daß eine Berechnung mehrere Wege gehen kann:



In q_1 kann der Automat bei Eingabe 0 zwei Wege einschlagen, nach q_2 oder q_3.

Von q_3 kann der Automat durch das leere Wort ε einfach nach q_2 gehen

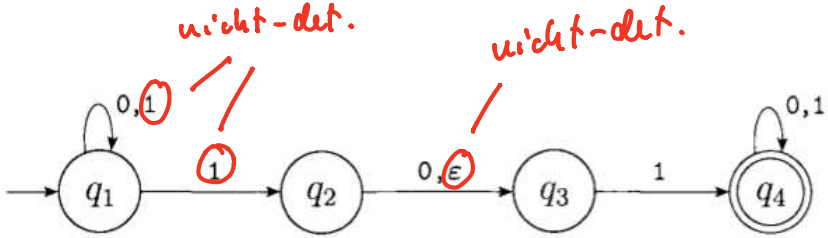
Der Automat muß sich aber nicht für einen Weg entscheiden. Er führt einfach alle möglichen Berechnungen parallel aus.

⚠ Das können ganz schön viele werden

Der Automat akzeptiert ein Wort, wenn mind. eine der parallelen Berechnungen in einem akzeptierenden Zustand endet.

Beispiel (Sipser):

Automat:



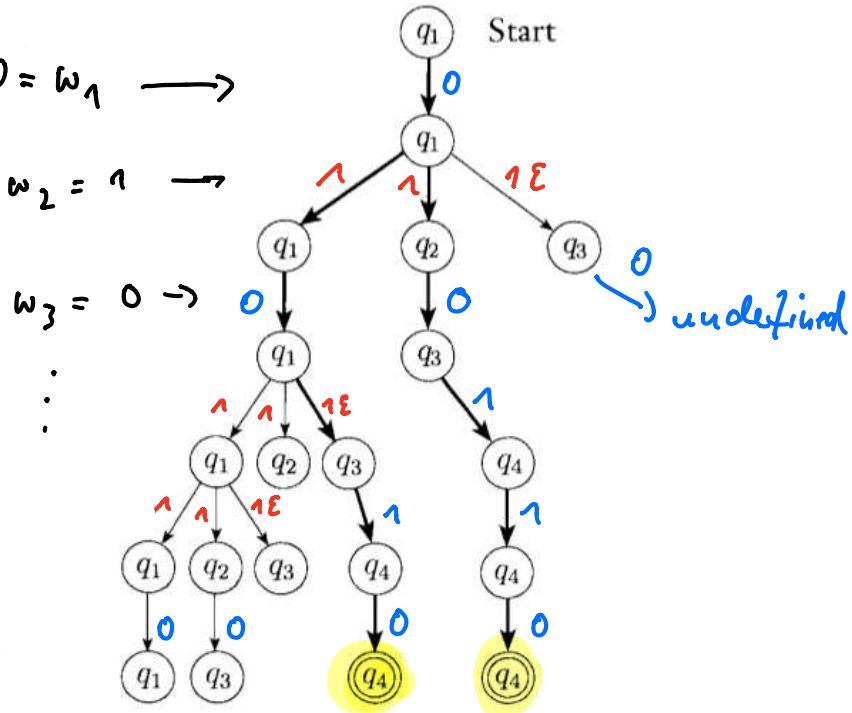
Eingabe: 010110

Berechnungsbaum:

$0 = w_1 \longrightarrow$

$w_2 = 1 \longrightarrow$

$w_3 = 0 \longrightarrow$



deterministisch

nicht-deterministisch



Nicht-determinismus wird hier nicht durch Wahrscheinlichkeitsmodelle oder "Zufall" modelliert.

(wenn man das macht, landet man eher bei Markov-Ketten aus der Stochastik).

Nicht-deterministischer Automat, formal:

typo corrected

Ein nicht-deterministischer, endlicher Automat besteht aus einem 5-Tupel $(Q, \Sigma, \delta, q_0, F)$.

- Q, Σ, q_0, F wie beim deterministischen Automat
- $\delta: Q \times \Sigma \cup \{\varepsilon\} \longrightarrow \mathcal{P}(Q)$ Übergangsfunktion

das leere Wort ε ist erlaubt.

Die Funktion nimmt also Mengen an, d.h. sie definiert die möglichen Zustände, in die man von einem Zustand durch Lesen eines Buchstaben gelangen kann.

Berechnung eines nicht-det. Aut., formal:

Sei $M = (Q, \Sigma, \delta, q_0, F)$ ein nicht-det. endl. Automat, $w = w_1 \dots w_n \in \Sigma^*$. Eine Folge von Zuständen $s_0, s_1, \dots, s_m \in Q$ heißt Berechnung von M auf w falls wir w schreiben können als $w = u_1 u_2 \dots u_m$ mit $u_i \in \Sigma \cup \{\varepsilon\}$ so daß

- $s_0 = q_0$
- Für alle $0 \leq i \leq m-1$: $s_{i+1} \in \delta(s_i, u_{i+1})$

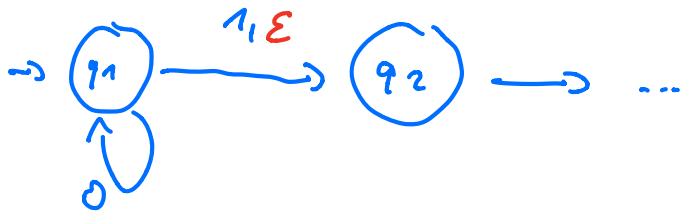
Übergänge ε, hier $u_i = \varepsilon$

Die Berechnung heißt akzeptierend, falls $s_m \in F$.

Der nicht-det. Automat M akzeptiert Wort w falls es eine akzeptierende Berechnung von M auf w gibt.

typo

⚠ **ε-Transitionen**: ein nicht-det. Automat kann bei "Lesen" des leeren Wortes ε einen Übergang machen, falls es so in der Ü-Funktion definiert ist. Beispiel:



Das heißt, daß der Automat einfach in den neuen Zustand gehen kann, ohne daß ein Buchstabe des Wortes verarbeitet wird. [Wir haben das schon in dem Beispiel vorhin gesehen]

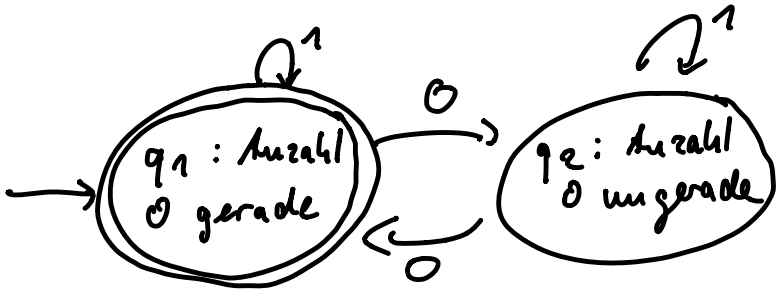
Noch ein Beispiel:

Betrachte die reguläre Sprachen $A := \{ x \in \{0,1\}^* \mid$ Anzahl 0 gerade $\}$

$B := \{ \quad \mid$ ungerade $\}$

Zugehörige Automaten:

M_A:



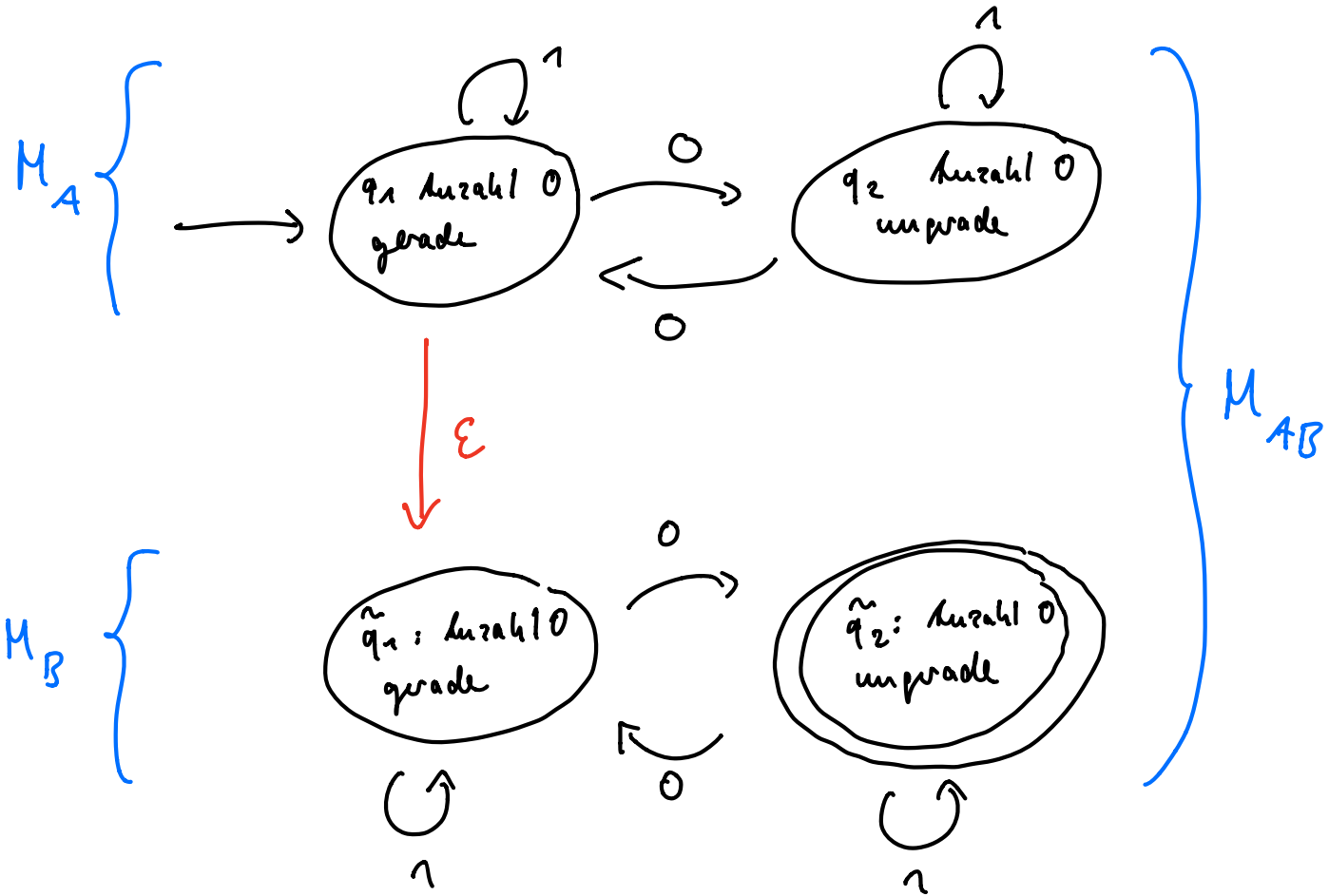
M_B: analog

? Nun betrachte die Konkatenation AB. Was müßte ein Automat tun,

- der diese Sprache erkennen soll?

Der Automat müßte bei einer Eingabe zunächst einen ersten Teil A des Worts betrachten und schauen, ob die Anzahl der 0 gerade ist. Irgendwann müßte er beschließen, daß nun der zweite Teil B des Worts anfängt und er müßte schauen, ob dort die Anzahl der 0 ungerade ist.

"Irgendwann" ⇝ nicht-deterministisch


M_A
q_1 Anzahl 0 gerade
q_2 Anzahl 0 ungerade
0
1
ε
M_B
$\tilde{q}_1$: Anzahl 0 gerade
$\tilde{q}_2$: Anzahl 0 ungerade
M_{AB}

Wichtige Frage ist nun:

können nicht-deterministische Automaten

"mehr" als deterministische

Automaten?

Sind nicht-deterministische Automaten "mächtiger" als deterministische?

Literatur: Sipser Kap. 1
(Schöning Kap. 1.2,
Hromkovic Kap. 3)

Sind nicht-det. Automaten mächtiger als det. Automaten?

Was heißt "mächtiger"?

Daß nicht-det. Automaten mehr Sprachen erkennen können als deterministische.

Das würde bedeuten, daß es einen nicht-det. Automat gibt, der eine nicht-reguläre Sprache erkennen kann.

Definition: Zwei Automaten M_1, M_2 (det. oder nicht-det.) heißen äquivalent, wenn sie die gleiche Sprache erkennen:

$$L(M_1) = L(M_2).$$

Hauptergebnis:

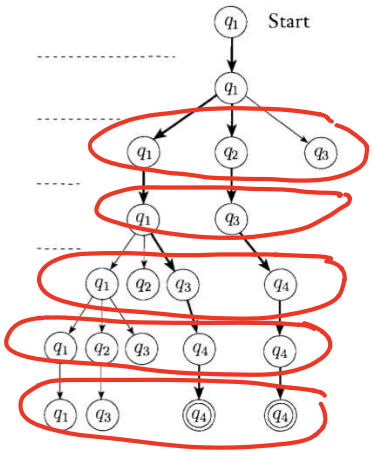
Satz: Zu jedem nicht-deterministischen endlichen Automaten gibt es einen äquivalenten deterministischen endl. Automaten.

Ist ziemlich überraschend! Das heißt, im Prinzip können nicht-det. und det. Automaten das gleiche!

⚠ Wir reden hier nicht über Effizienz, sondern nur über die Tatsache, ob etwas prinzipiell berechnet werden kann.

Beweisidee:

- Zustände des det. Automaten bestehen aus Mengen aus Q.
- Übergangsfunktion beschreibt "Schichten" im Berechnungsbaum
- Wenn $|Q| = n$, dann gibt es 2^n mögliche "Schichten"
- Wir definieren nun eine passende deterministische $\ddot{u}$-Funktion zwischen diesen Schichten.



Theorie, formal

Sei $N = (Q, \Sigma, \delta, q_0, F)$ ein nicht-det. Automat, der eine Sprache A erkennt.

Ziel: Konstruiere einen det. Automat $M = (Q', \Sigma, \delta', q_0', F')$, der auch A erkennt.

Erster Fall: es kommen in N keine ε-Transitionen vor.

Wir definieren nun die Bausteine des Automaten:

- $Q' = \mathcal{P}(Q)$ (Potenzmenge, Menge aller Teilmengen von Q)
- $q_0' = \{q_0\}$
- $F' = \{R \in Q' \mid R \text{ enthält mind. einen akzeptierenden Zustand von } N\}$

- Die neue Ü-Funktion:

$$\delta' : Q' \times \Sigma \to Q',$$

$$\delta'(R, a) = \{ q \in Q \mid q \in \delta(r, a) \text{ für ein } r \in R \}$$

R: Menge von Zuständen

a: Buchstabe

alle Zustände, die N ausgehend von Zuständen in R erreichen könnte

Für den so definierten Automaten gilt offensichtlich, daß er die gleichen Wörter akzeptiert wie N.

→

Zweiter Fall: es gibt ε-Transitionen

clarified

In diesem Fall müssen wir die $\hat{\delta}$-Funktion anpassen.

Für eine Menge $R \subseteq Q$ setzen wir

$$E(R) = \{q \mid q \text{ kann von } R \text{ durch } \varepsilon\text{-Transitionen erreicht werden}\} \cup R$$

Dann definieren wir δ' durch

$$\delta'(R, a) = \{q \mid q \in E(\delta(r, a)) \text{ für ein } r \in R\}$$

Und wir passen noch den Startzustand an:

$$q_0' = E(\{q_0\}).$$

Das tut's.

Satz

Bemerkung: Der Zustandsraum des Automaten wächst exponentiell (im worst case), wenn wir einen nicht-det. Automaten zu einem det. Automaten konvertieren. Wir bräuchten also deutlich mehr Speicherplatz ...

Korollar (einfache aber wichtige Folge des Satzes)

Eine Sprache L ist regulär $\Leftrightarrow$ Es gibt einen nicht-deterministischen Automaten, der sie akzeptiert.

Weitere Abschluß-Eigenschaften regulärer Sprachen

Literatur: Sipser Kap. 1
(Schöning Kap. 1.2,
Hromkovic Kap. 3)

Weitere Abschluß-Eigenschaften von REG

Haben schon gesehen: Die Klasse REG der regulären Sprachen ist abgeschlossen unter Vereinigung, Schnitt, Komplementbildung.

Mit Hilfe der nicht-det. Automaten können wir nun einfach weitere Abschlußeigenschaften beweisen:

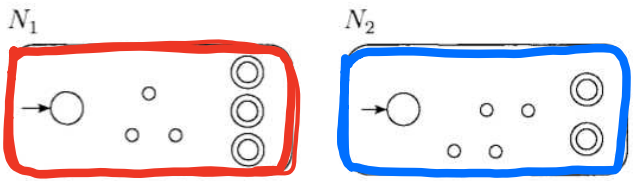
Abschluß unter Konkatenation

Satz: Die Klasse der regulären Sprachen ist abgeschlossen unter Konkatenation: $L_1, L_2 \in REG \Rightarrow L_1 L_2 \in REG$

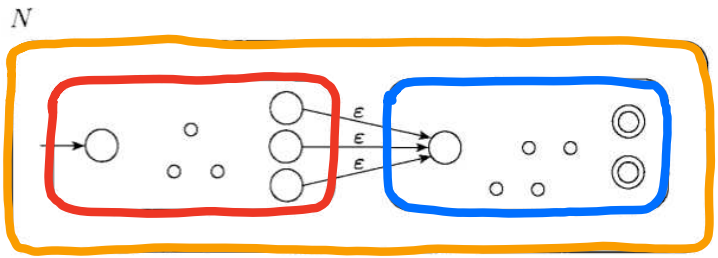
Beweis (Skizze) Wir haben das im Wesentlichen schon als Beispiel für einen nicht-det. Automaten betrachtet.

Hier ist die Konstruktion:

Automat der Sprache A



Automat der Sprache B



Nicht-det. Automat der Sprache AB

Abschluß unter *-Bildung

Satz: Die Klasse REG ist abgeschlossen unter Bildung der Kleene'schen Hülle, d.h.

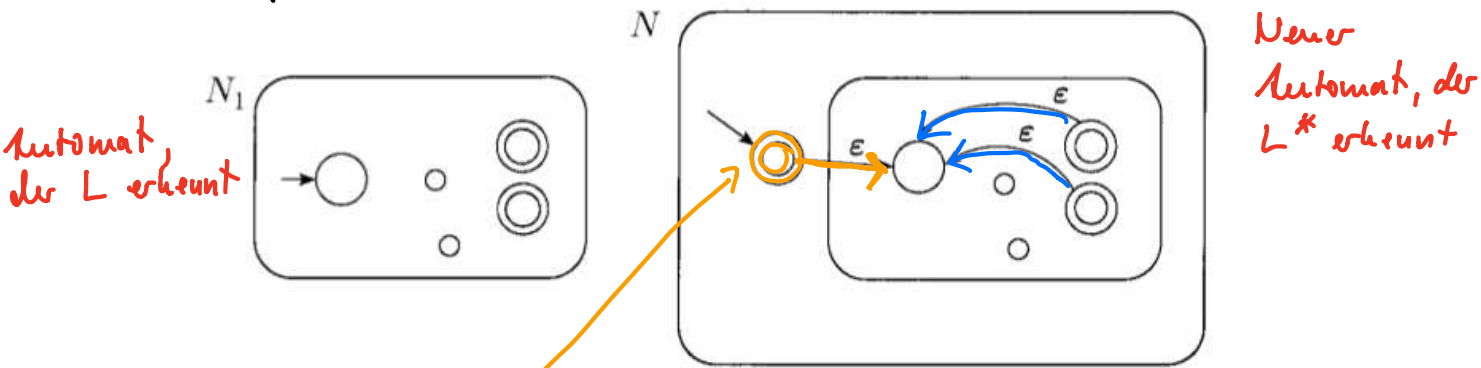
$$L \in REG \Rightarrow L^* \in REG.$$

Beweis (Skizze)

Da L regulär ist, gibt es einen Automaten N_1, der L akzeptiert.

Diesen modifizieren wir nun in zwei Hinsichten:

- Daß er auch beliebige, hintereinandergehängte Worte von L akzeptiert
- Daß er auch ε akzeptiert.



Automat, der L erkennt

Neuer Automat, der L^* erkennt

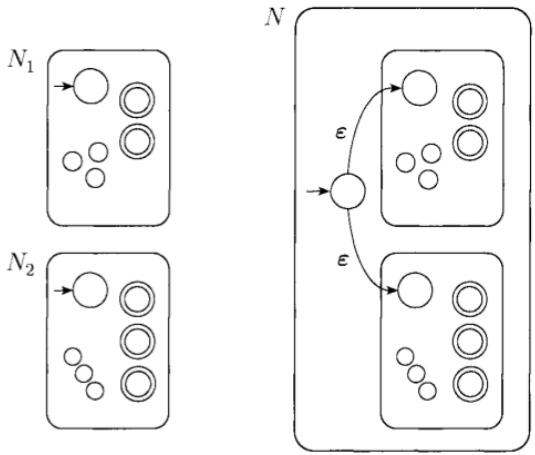
Neuer (akzeptierender) Startzustand (damit ε akzeptiert wird)

- Neue ε-Transitionen von den alten Endzuständen zum alten Startzustand (um ein weiteres Wort zu lesen)

Abschluß unter Vereinigung, nochmal ...

Wir haben schon gesehen, daß REG abgeschlossen unter Vereinigung ist. Gegeben zwei deterministische Automaten haben wir damals den auch deterministischen Produktautomaten konstruiert.

Mit nicht-deterministischen Automaten geht das nun noch viel einfacher:



Insbesondere brauchen wir nicht mehr so viele Zustände (aber diese entstehen natürlich, wenn wir den nicht-det. Automaten in einen det. Aut. konvertieren.)

Reguläre Ausdrücke

Literatur: Sipser Kap. 1
(Schöning Kap. 1.2,
Hromkovic Kap. 3)

Haben Sie in Linux/Unix schon mal benutzt ... ?

- rm *.pdf ⇝ * steht für alle strings

- grep -r '[Vv]orlesung * 20[0-2][0-9]' *

  Ein Buchstabe, entweder V oder v

  beliebiger String

  eine Ziffer

  noch eine Ziffer

  alle files im aktuellen Verzeichnis

- awk, sed, find, ...

⇝ reguläre Ausdrücke!

Reguläre Ausdrücke

... werden induktiv definiert:

Def Sei Σ ein Alphabet. Dann:

- $\emptyset$ und ε sind reguläre Ausdrücke (leere Sprache / leeres Wort)
- Alle Buchstaben aus Σ sind reg. Ausdrücke
- Falls R_1, R_2 reguläre Ausdrücke sind, dann sind auch die folgenden Ausdrücke regulär:
  - $R_1 \cup R_2$
  - $R_1 \circ R_2$
  - R_1^*

Priorität der Operatoren

Konvention: wenn wir reguläre Ausdrücke aufschreiben, gehen wir von der folgenden "Verarbeitungs-Reihenfolge" aus:

Erst $*$, dann $\circ$, dann $\cup$

Beispiele

- $0^* 1 0^*$: beliebig viele 0, dann eine 1, dann beliebig viele 0

- $\Sigma^* 1 \Sigma^*$ $\rightsquigarrow$ Wörter mit mind. einer 1

- $(0 \cup \varepsilon) 1^* = 0 1^* \cup 1^*$

  $\rightsquigarrow$ Wörter bestehend aus 0 gefolgt von bel. vielen 1 und Wörter aus bel. vielen 1

- Regular expression builder in emacs!

Induzierte Sprache $L(R)$

Def Sei R ein regulärer Ausdruck. Dann ist die von R induzierte Sprache $L(R)$ wie folgt definiert:

- $R = \emptyset \Rightarrow L(R) = \emptyset$
- $R = \varepsilon \Rightarrow L(R) = \{\varepsilon\}$
- $R = \sigma$ für ein $\sigma \in \Sigma \Rightarrow L(R) = \{\sigma\}$

- $R = R_1 \cup R_2 \Rightarrow L(R) = L(R_1) \cup L(R_2)$

  $R = R_1 \circ R_2 \Rightarrow L(R) = L(R_1) \circ L(R_2)$

  $R = R_1^* \Rightarrow L(R) = (L(R_1))^*$.

Algebraische Regeln

Seien E, F, G reguläre Ausdrücke. Dann gilt:

- $\cup$ ist assoziativ, kommutativ, hat $\emptyset$ als neutrales Element.
- $\circ$ ist assoziativ, nicht kommutativ, hat ε als neutrales El.,
- Distributivgesetze: $(E \cup F) \circ G = E \circ G \cup F \circ G$

  $E \circ (F \cup G) = E \circ F \cup E \circ G$

- $(E^*)^* = E^*$
- $\emptyset^* = \varepsilon$
- $\varepsilon^* = \varepsilon$

Beweis: einfach, lassen wir weg.

Reguläre Ausdrücke und reguläre Sprachen

Satz: Eine Sprache ist genau dann regulär, wenn sie durch einen regulären Ausdruck beschrieben wird.

Ist erst mal nicht offensichtlich!!!

Richtung 1 : regulärer Ausdruck => reguläre Sprache

Beweis: Strukturelle Induktion

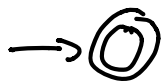
Induktionsanfang:

Falls $R = a$ für ein $a \in \Sigma$, dann gilt nach Definition daß $L(R) = \{a\}$.

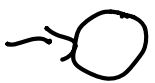
Diese Sprache wird von folgendem Automaten erkannt und ist deshalb regulär:



Falls $R = \varepsilon$, dann $L(R) = \{\varepsilon\}$, Automat



Falls $R = \emptyset$, dann $L(R) = \emptyset$, Automat



.

Induktionsschritt: Seien R_1, R_2 reguläre Ausdrücke, für die wir schon endliche Automaten konstruiert haben (= Induktionsannahme).

Dann können wir auch endliche Automaten konstruieren für

$R_1 \cup R_2$ (wie im Beweis, daß REG abgeschlossen unter $\cup$)

$R_1 \circ R_2$ (" $\circ$)

R_1^* (" *).

Also sind auch die Sprachen $L(R_1 \cup R_2)$, $L(R_1 \circ R_2)$, $L(R_1^*)$ regulär.

Richtung 1

Richtung 2: Reguläre Sprache ⇒ regulärer Ausdruck

Beweisidee:

- Sei $M = (Q, \Sigma, \delta, q_0, F)$ ein Automat, der L erkennt, $Q = \{1, \dots, n\}$.
- Betrachte beliebige zwei Zustände i, j. Wollen die Wörter finden, mit denen wir von i nach j gelangen. Das machen wir induktiv:
  - Von i nach j mit direktem Link ($k = 0$)
  - ... über $\{1\}$
  - ... über $\{1, \dots, k\}$

  → Wir beschreiben diese Wörter mit regulären Ausdrücken E_{ij}^{k}

- Dann: $L(E_{i,j}^{n}) = \{w \mid \delta^*(i, w) = j\}$, und

$$L(M) = \bigcup_{q_f \in F} L(E_{q_0, q_f}^{n})$$

n: alle Zustände erlaubt

q_f: Ende akzeptierend

q_0: Start am Anfang

Formaler Beweis:

Ziel: Konstruiere reguläre Ausdrücke $E_{i,j}^k$ so daß

(*) $$L(E_{i,j}^k) = \{ w \mid \delta^*(i,w) = j \text{ und für alle Präfixe } w' \text{ von } w \text{ mit } w' \neq \varepsilon,\ w' \neq w \text{ gilt } \delta^*(i,w') \leq k \}$$

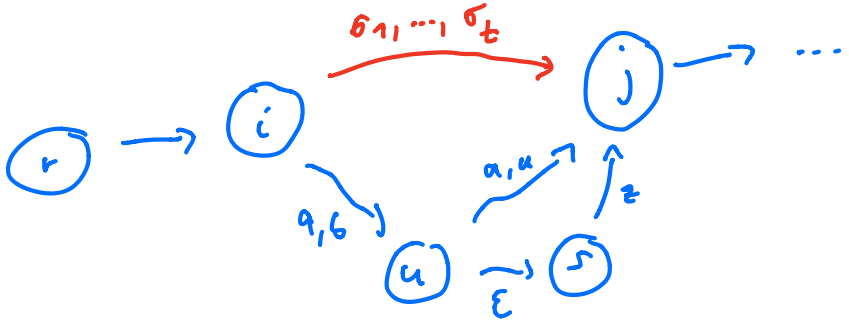
Zwischenzustände sind $\in \{1, \ldots, k\}$

Wir benutzen eine induktive Konstruktion, wobei die Induktion über k läuft.

Ind. Anfang: $k = 0$

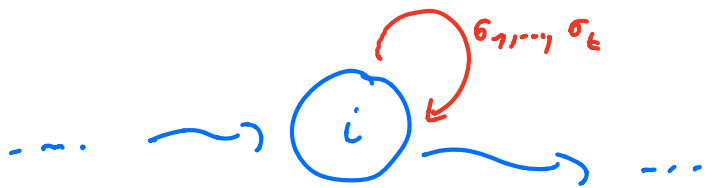
- Kein Zwischenzustand von i nach j erlaubt, nur direkte Übergänge.
- Sei $i \neq j$. Falls es einen direkten Link von i nach j gibt, seien $\sigma_1, \dots, \sigma_t$ die Buchstaben auf dem Link.

  Setze $E_{ij}^0 = \sigma_1 \cup \dots \cup \sigma_t$.



Falls es keinen direkten Link gibt, setze $E_{ij}^0 = \emptyset$.

- Sei $i = j$. Falls es einen direkten Link von i nach i gibt, seien $\sigma_1, \dots, \sigma_t$ die Buchstaben. Setze $E_{ii}^0 = \varepsilon \cup \sigma_1 \cup \dots \cup \sigma_t$
  Falls es keinen direkten Link gibt, setze $E_{ii}^0 = \varepsilon$.

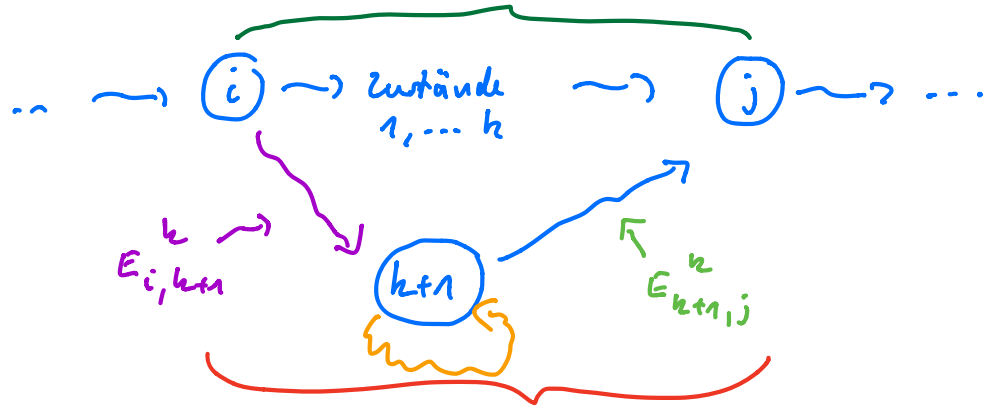


- offensichtlich ist Eigenschaft (*) erfüllt.

Induktionsannahme: haben alle E_{ij}^{r} $(0 \le r \le k)$ konstruiert. für all i, j

Induktionsschritt $k \rightsquigarrow k+1$: Wollen E_{ij}^{k+1} konstruieren.

entweder wir nehmen den "alten" Weg über Zustände aus $1, \dots, k$



oder wir nehmen neuen Zwischenzustand $k+1$

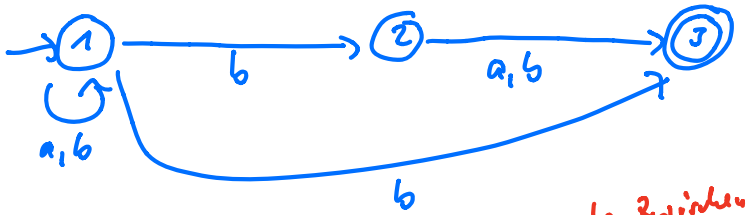
$$E_{i,j}^{k+1} = \overbrace{E_{ij}^{k}}^{\text{old}} \cup \overbrace{E_{i,k+1}^{k} \left(E_{k+1,k+1}^{k} \right)^{*} E_{k+1,j}^{k}}^{\text{new}}$$

Notation angepasst

Diese Konstruktion erfüllt offensichtlich die Eigenschaft (*).

■ Richtung 2 und ganzer Satz.

Beispiel für Richtung 2:



kein Zwischenzustand — 1 als Zwischenzustand — 1 oder 2 als Zwischenzustand — 1 oder 2 oder 3 ...

$E_{11}^{0} = \varepsilon \cup a \cup b$; $E_{11}^{1} = (\varepsilon \cup a \cup b)^* = E_{11}^{2} = E_{11}^{3}$

(von — nach)

$E_{12}^{0} = b$; $E_{1,2}^{1} = (a \cup b)^* b = E_{1,2}^{2} = E_{1,2}^{3}$

$E_{13}^{0} = b$; $E_{13}^{1} = b \cup (a \cup b)^* b$

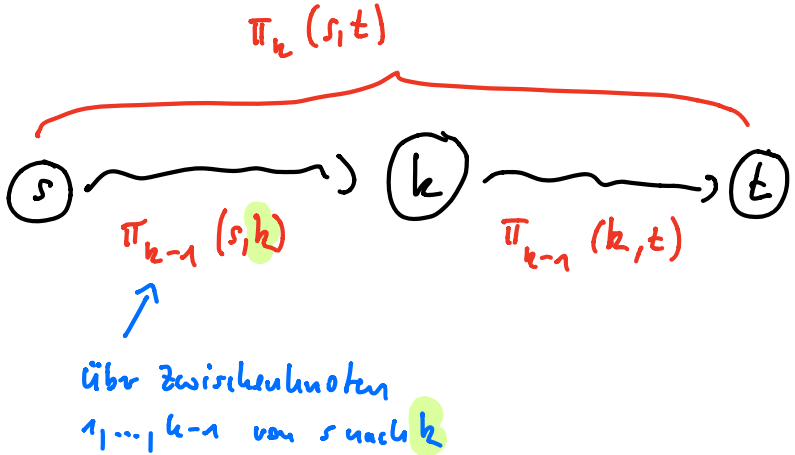
$E_{13}^{2} = E_{13}^{1} \cup E_{12}^{1} (E_{2,2}^{1})^* E_{23}^{1}$

$= (b \cup (a \cup b)^* b) \cup (a \cup b)^* b (\varepsilon)^* (a \cup b)$

$= (b \cup (a \cup b)^* b) \cup (a \cup b)^* b (a \cup b) = E_{1,3}^{3}$.

Kommt Ihnen die Beweisidee von Richtung 2 bekannt vor?

Floyd-Warshall-Alg. für all-pairs-shortest-paths:



typo

$$d_k(s,t) = \min\left\{ d_{k-1}(s,k) + d_{k-1}(k,t) \;,\; d_{k-1}(s,t) \right\}$$

Haben also bewiesen:

Satz: Eine Sprache ist genau dann regulär, wenn sie durch einen regulären Ausdruck beschrieben wird.

Das Pumping-Lemma

Literatur: Sipser Kap. 1
(Schöning Kap. 1.2,
Hromkovic Kap. 3)

Gibt es Sprachen, die nicht regulär sind?

Vermutlich ja: endliche Automaten haben nur ein endliches Gedächtnis, vielleicht reicht das nicht für sehr "komplexe" Sprachen

Beispiel: $L = \{ 0^n 1^n \mid n \in \mathbb{N} \} = \{ 01, 0011, 000111, \ldots \}$

Vermutlich müsste ein Automat, der diese Sprache erkennt, "zählen" können. Und zwar beliebig große Zahlen. Scheint mit endlich vielen Zuständen nicht möglich.

Wie könnten wir formal beweisen, daß L nicht regulär ist?

Warum ist das schwierig?

Das Pumping-Lemma

präzisiert

Satz Sei A eine reguläre Sprache (über dem Alphabet Σ). Dann gibt es eine natürliche Zahl n, so daß sich alle Wörter $s \in A$ mit Länge $|s| \geq n$ zerlegen lassen in drei Teilworte $s = xyz$ mit $x, y, z \in \Sigma^*$, so daß gilt:

(i) $|y| > 0$

(ii) $|xy| \leq n$

(iii) Für alle $i \geq 0$ ist $x y^i z \in A$.

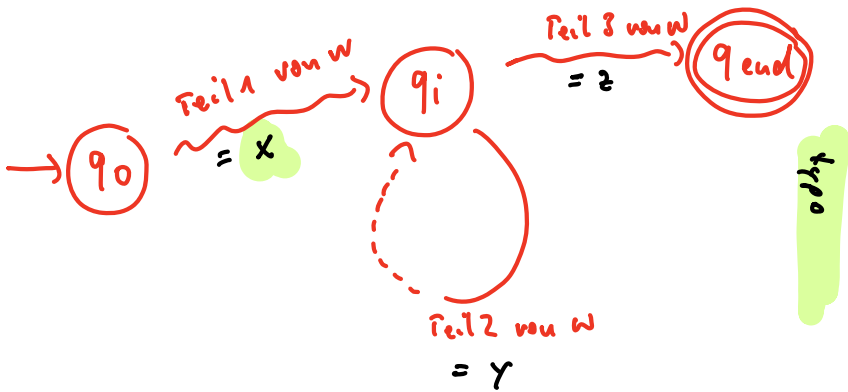
Wir wollen das Lemma später benutzen um zu zeigen, daß bestimmte Sprachen nicht regulär sind.

Beweisidee für das Pumping-Lemma

- Sei M der Automat, der A erkennt. Wähle n als die Anzahl der Zustände von M.
- Nun betrachte die Verarbeitung eines Wortes w der Länge > n. Der Automat besucht dabei n+1 Zustände (inkl. Start).

Da der Automat nur n Zustände hat, muß es einen Zustand geben, der mind. 2 mal besucht wird:

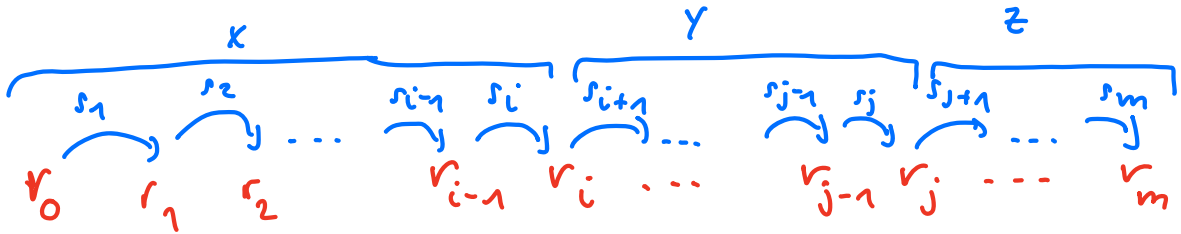
- Es gibt also eine Schleife.
- Und die können wir beliebig oft laufen.



Beweis, formal

Sei M ein deterministischer Automat, der A erkennt. Setze n = Anzahl seiner Zustände. Sei $\overbrace{s_1 \dots s_m}^{w}$ ein String der Länge $m > n$.

Sei $r_0, \dots, r_m$ die Folge der Zustände, die der Automat bei der Verarbeitung von w durchläuft. Unter den ersten $n+1$ dieser Zustände muß es mind. einen geben, der zwei mal vorkommt. Wir bezeichnen sie mit r_i und r_j:



Setze $x = s_1 \dots s_i$, $y = s_{i+1} \dots s_j$, $z = s_{j+1} \dots s_m$.

Diese Wahl erfüllt das Pumpinglemma. Insbesondere führt ein Wort $xy^i z$ immer auch zum Endzustand, wird also akzeptiert.

□ Lemma

Anwendung des Pumping-Lemmas

Wenn wir "Glück haben", können wir das Lemma benutzen, um zu zeigen, daß eine Sprache nicht regulär ist:

Um zu verstehen, wie das geht, schreiben wir das Lemma nochmal mit logischen Ausdrücken hin.

made consistent with definition

Pumping-Lemma, anders aufgeschrieben:

Lemma:

A regulär $\Rightarrow$

$\exists n \in \mathbb{N}\ \forall s$ with $|s| \geq n, s \in L\ \exists x,y,z$ with $y \neq \varepsilon, |xy| \leq n\ \forall i \in \mathbb{N}_0$: $xy^i z \in A$

($s \in \Sigma^*$)

Kontraposition davon:

$\neg(\exists n \ldots) \Rightarrow A$ nicht regulär.

Wie sieht die Verneinung der langen Aussage aus?

(*) $\forall n\ \exists s\ \forall x,y,z\ \exists i : xy^i z \notin A$

↖ das müssen wir also zeigen, um zu beweisen, daß eine Sprache nicht regulär ist.

Anwendungsbeispiel des Pumping-Lemmas:

Beh: Die Sprache $L = \{0^n 1^n \mid n \in \mathbb{N}\}$ ist nicht regulär.

Bew mit Hilfe des Pumping-Lemmas:

Betrachte ein beliebiges $n \in \mathbb{N}$. Wähle das Wort $s = 0^n 1^n$. (es gilt $|s| \geq n$.)

Sei nun $xyz = s$ eine beliebige Zerlegung (mit $|y| > 0$, $|xy| \leq n$). Wir müssen nun ein i finden, so daß $xy^i z$ nicht in L ist.

Fall 1: y besteht nur aus 0en: $s = \underbrace{0}_{x}\underbrace{0}_{y}\underbrace{\ldots 0 1 1 \ldots 1}_{z}$

Dann ist $xy^2z \notin L$ (da es mehr 0er als 1er hat).

Fall 2: y nur aus 1en, analog.

Fall 3: y hat 0en und 1en: $s = \underbrace{0 \ldots}_{x}\underbrace{0 1 1 \ldots}_{y}\underbrace{1}_{z}$

Dann ist aber $xy^2z \notin L$.

Typo

genau genommen braucht man diese Fälle nicht, da hier $|xy| > n$ gilt.

$\underbrace{0 \ldots 0}_{x}\ \underbrace{01}_{y}\ \underbrace{01}_{y}\ \underbrace{1 \ldots 1}_{z}$

Beh.



Das Pumping-Lemma kann nur benutzt werden um zu zeigen, daß eine Sprache nicht regulär ist.

Die Umkehrung gilt nicht: es gibt Sprachen, die das Pumping-Lemma erfüllen und trotzdem nicht regulär sind!

Noch ein Beispiel:

Betrachte die Sprache $L = \{w \mid$ die Teilstrings 01 und 10 kommen gleich oft auf$\}$

Erste Vermutung: gleiche Situation wie bei $L = \{0^n 1^n\}$, man müßte die Teilstrings zählen, daher L nicht regulär.

Diese Vermutung ist falsch!!! L ist regulär!!!

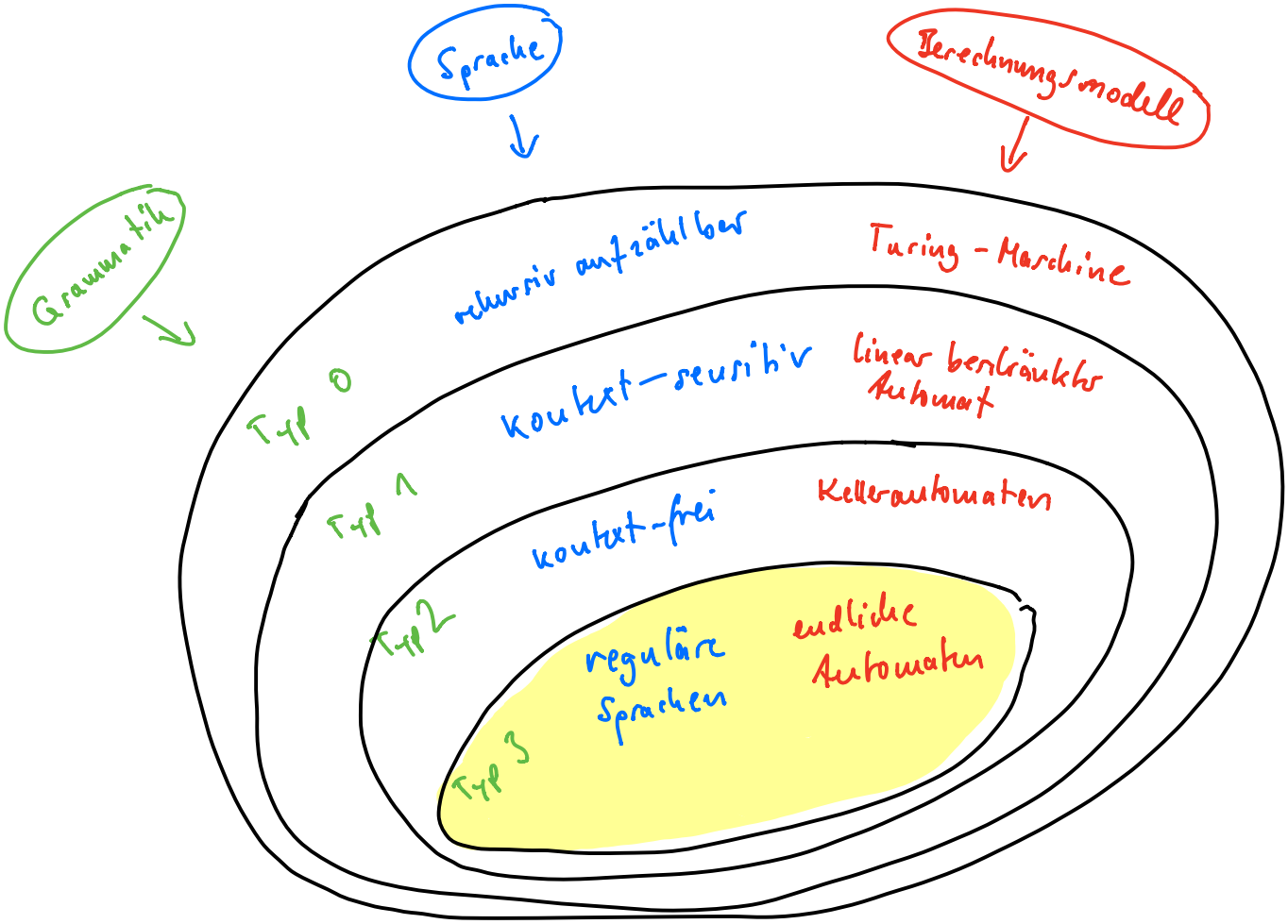
Übungsaufgabe:

(1) Beweisen Sie, daß L regulär ist.

Hinweis: betrachten Sie viele 0-1-Strings und wie sich in diesen die 01- und 10-Strings "abwechseln".

(2) Was geht bei der Anwendung des Pumping-Lemmas schief?

Ausblick jenseits regulärer Sprachen und endlicher Automaten


Sprache
Berechnungsmodell
Grammatik
rekursiv aufzählbar
Turing-Maschine
Typ 0
kontext-sensitiv
linear beschränkter Automat
Typ 1
kontext-frei
Kellerautomaten
Typ 2
reguläre Sprachen
endliche Automaten
Typ 3

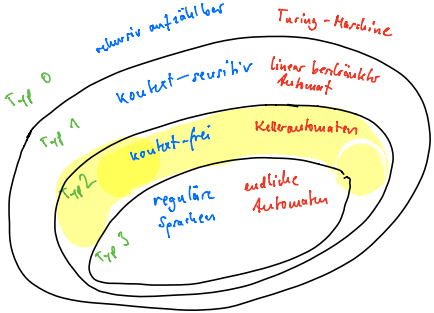
Kellerautomaten und kontextfreie Sprachen

- Automaten mit zusätzlichem Speicherplatz, einem Stack "im Keller", der beliebig groß werden kann.
- Erkennen eine größere Klasse von Sprachen, kontext-freie Sprachen.

  Beispiel: die Sprache $L = \{0^n 1^n \mid n \in \mathbb{N}\}$

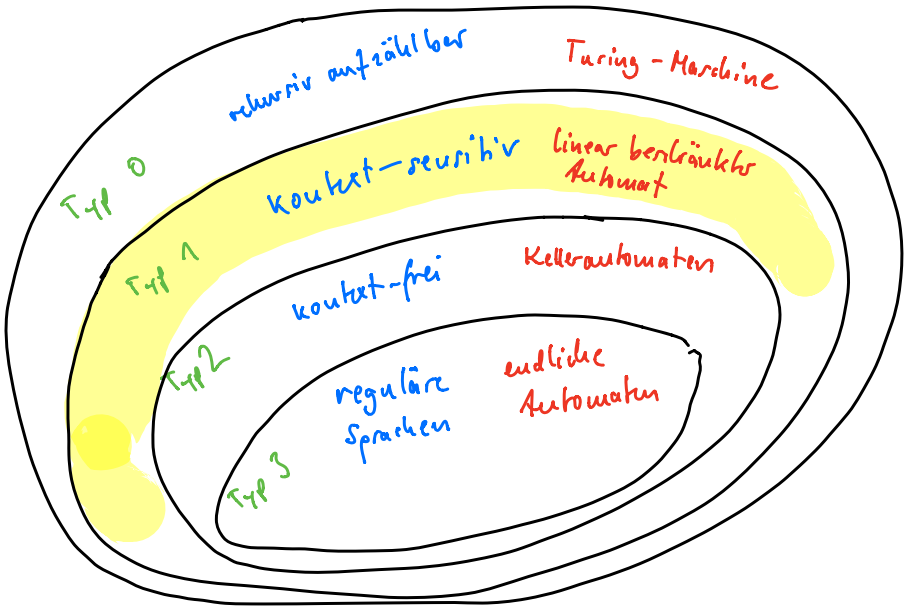
- auch für diese Automaten gibt es eine Art Pumping-Lemma, und Sprachen, die nicht davon erkannt werden.

  Beispiel: die Sprache $L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$



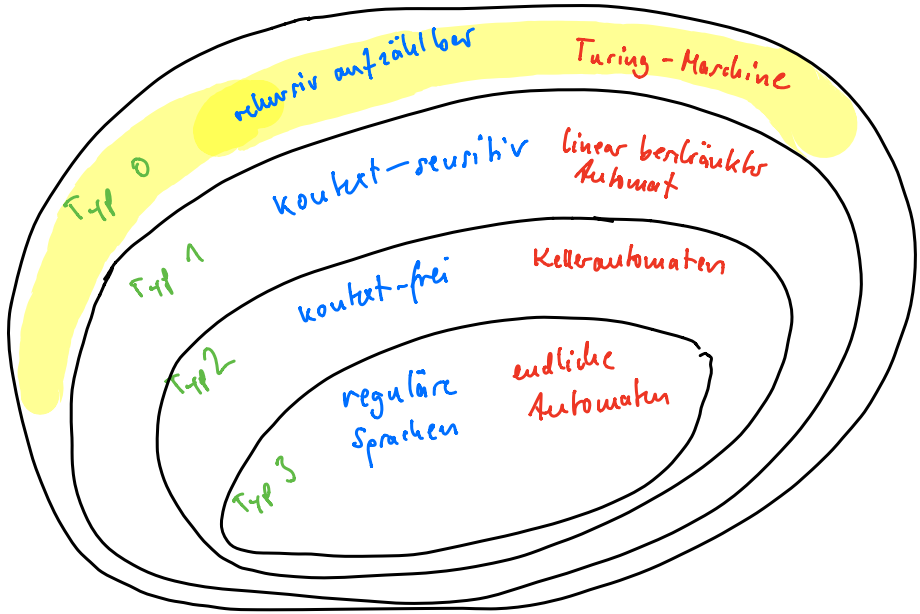
Linear beschränkter Automat

... überspringen wir zunächst komplett ...



Nun kommt als nächstes die Turingmaschine...

... das Berechnungsmodell, das einem Großteil der theoretischen Informatik zu Grunde liegt.



Kleiner Gruß meiner Tochter aus dem Homeschooling →

Vom Maiku für die Studis

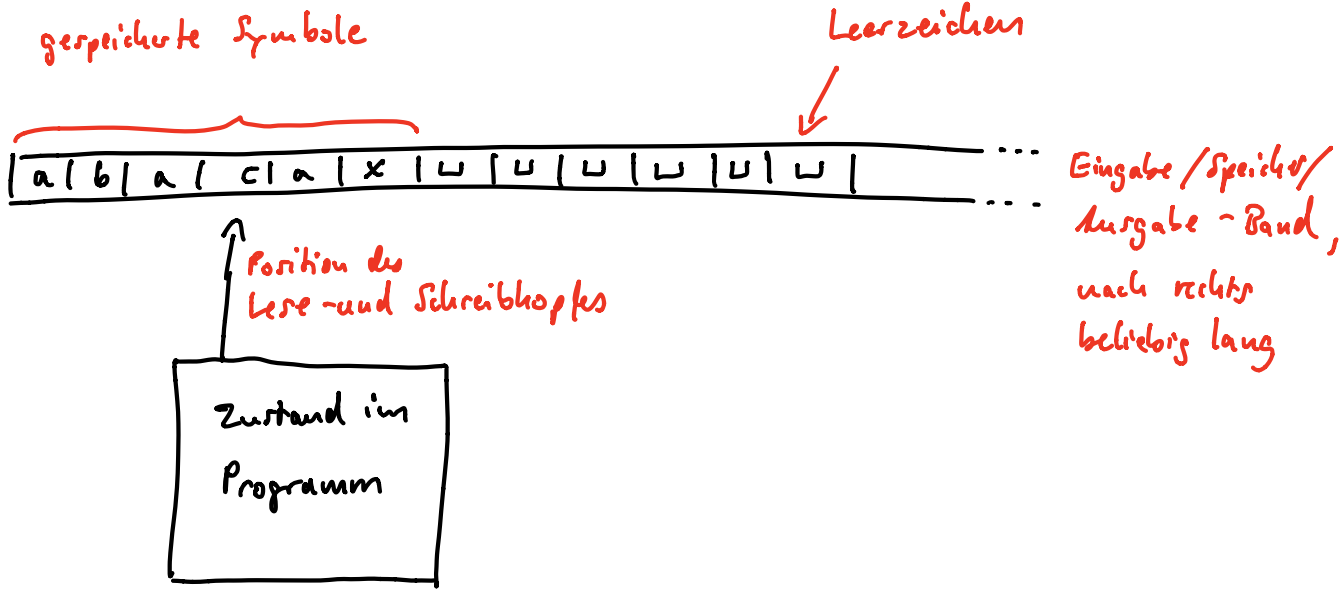
Turing - Maschinen

Literatur: Sipser Kap. 3, Hromkovic Kap. 4

Ein allgemeineres Modell eines Computers

- Kann eine Eingabe lesen
- Hat beliebig viel Speicherplatz
- Kann Dinge an beliebigen Stellen in den Speicher schreiben und wieder auslesen
- kann beliebig viele Rechenschritte machen

Turing-Maschine, intuitiv



Turing-Maschine, intuitiv

Am Anfang der Berechnung steht die Eingabe auf dem Band, der Lesekopf befindet sich im ersten Feld, und der Zustand ist der Startzustand q_0.

Turing-Maschine, intuitiv

Eine Konfiguration der TM ist beschrieben durch

- Den Inhalt des Bandes
- Die Position des Lese/Schreibkopfes
- Den derzeitigen Zustand $\in Q$

Turing-Maschine, intuitiv

In jeder Konfiguration macht die TM das folgende:

- sie liest das Symbol an der Stelle, wo der Lese/Schreibkopf gerade steht
- sie schaut in ihrer Übergangsfunktion nach, was sie im derzeitigen Zustand bei Lesen dieses Symboles tun soll:
  - sie kann im derzeitigen Feld ein neues Symbol schreiben
  - sie kann den Lese/Schreibkopf um ein Feld nach links oder rechts bewegen
  - sie kann in einen neuen Zustand übergehen.

Turing-Maschine, intuitiv

Falls die TM in einem der beiden Endzustände ankommt, hält die TM an und ist fertig. Der Inhalt des Bandes ist dann die Ausgabe der TM.

Es kann jedoch passieren, daß die TM nie in einem Endzustand ankommt. Dann rechnet sie immer weiter und kommt nie zu einem Ende. Wir sagen, sie "terminiert nicht".

Erstes Beispiel, intuitiv

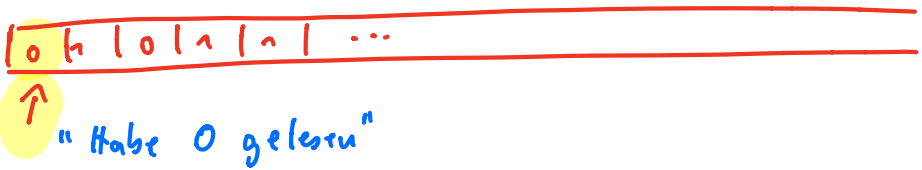
Betrachte die Sprache $L = \{ w \# w \mid w \in \{0,1\}^* \}$.

(zweimal der gleiche String w, getrennt durch Symbol $\#$).

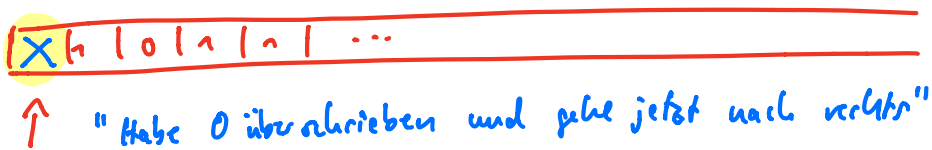
Wie kann eine TM diese Sprache erkennen?

Erstes Beispiel, intuitiv

- Lies den ersten Buchstaben der Eingabe. Merke ihn dir.

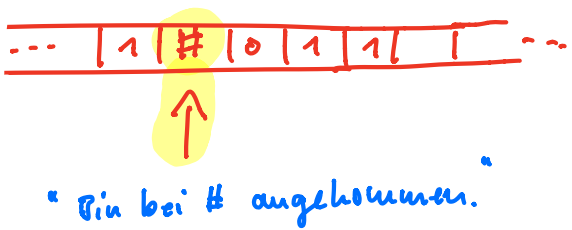


Überschreibe den Buchstaben mit einem neuen Symbol $\notin \{0, 1, \#\}$



- Geh mit dem Lesekopf so weit nach rechts, bis das Symbol # erscheint.





Erstes Beispiel, intuitiv

- Vergleiche den folgenden Buchstaben mit dem gemerkten.



  - Falls nicht gleich, reject.
  - Falls gleich: überschreibe ihn mit Y und gehe zurück, bis du wieder bei X bist.



  - Lies den folgenden Buchstaben ... usw

Formale Definition der Turing-Maschine (TM)

Def Eine Turing-Maschine (TM) ist ein 7-Tupel $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$

- Q ist eine endliche Menge von Zuständen
- Σ ist eine endliche Menge, das Eingabealphabet
- Γ ist eine endliche Menge, das Arbeitsalphabet, mit $\Sigma \subset \Gamma$ und einem "Leerzeichen" $\sqcup \in \Gamma$
- $\delta: Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$ Übergangsfunktion
- $q_0 \in Q$ Startzustand
- $q_{accept} \in Q$ akzeptierender Endzustand
- $q_{reject} \in Q$, $q_{reject} \neq q_{accept}$ verwerfender Endzustand

Bemerkungen zu dieser Definition

- Es gibt genau einen akzeptierenden und einen verwerfenden Zustand.
- Die TM beendet ihre Berechnung, sobald sie einen dieser beiden Zustände erreicht.
- Das Band der TM hat "ein linkes Ende", nach rechts ist es unbeschränkt.

Beispiel: Erkennen von Zweierpotenzen

typo

Wollen eine TM bauen, die $L = \{ 0^{2^n} \mid n \in \mathbb{N}_0 \}$ erkennt.

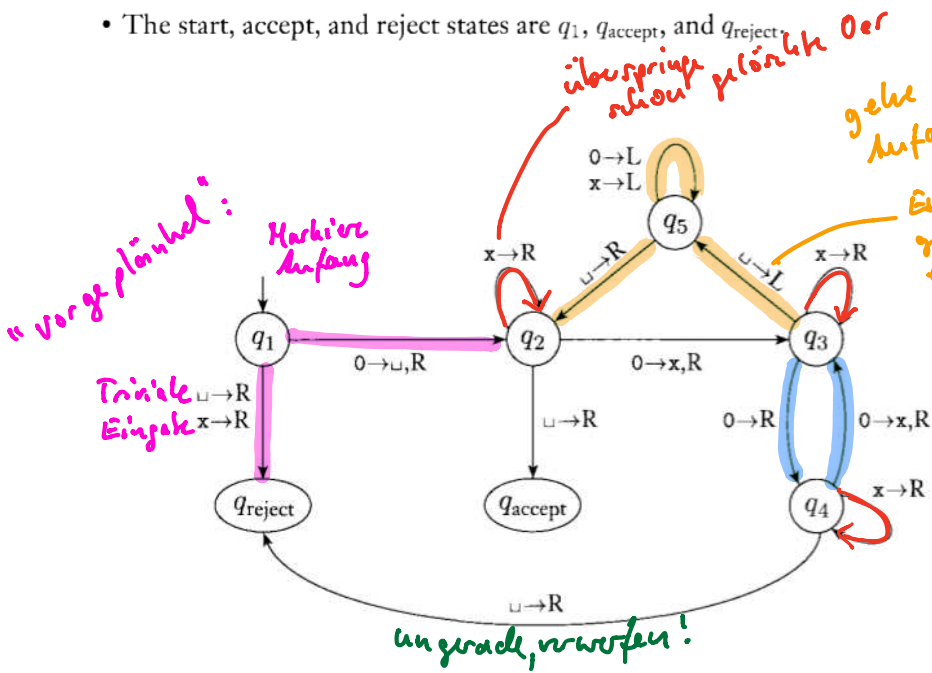
Idee:

- Lies von links nach rechts und streiche jede zweite 0 weg. Dabei:
  - Wenn es nur eine 0 gibt, akzeptiere.
  - Wenn es mehr als eine 0 gibt und die Anzahl ungerade ist, verwerfe.
  - Wenn die Anzahl gerade war, fang wieder von vorne an.

Beispiel: Erkennen von Zweierpotenzen

Formal:

- $Q = \{q_1, q_2, q_3, q_4, q_5, q_{\text{accept}}, q_{\text{reject}}\}$,
- $\Sigma = \{0\}$, and
- $\Gamma = \{0, x, \sqcup\}$.
- We describe δ with a state diagram (see Figure 3.8).
- The start, accept, and reject states are q_1, q_{accept}, and q_{reject}.



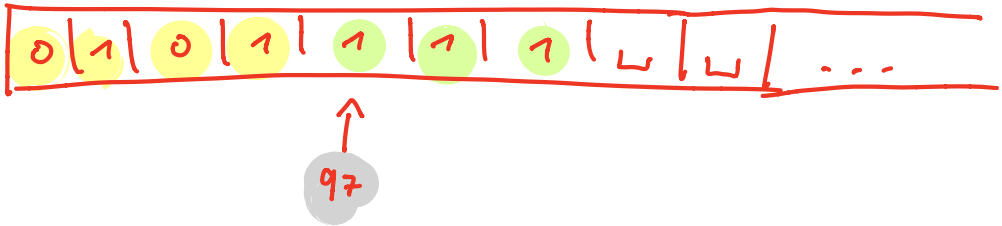
Weiter in der Definition der TM

Eine Konfiguration der TM wird beschrieben durch den Inhalt des Bandes, die Position des Lesekopfes und den derzeitigen Zustand $q \in Q$. In kompakter Notation wird das wie folgt aufgeschrieben:

"$u\ q\ v$"

- Inhalt des Speicherbandes ist der string uv
- Position des Schreibkopfes ist direkt nach u, auf dem ersten Buchstaben von v
- Zustand ist q

Beispiel:



wird beschrieben durch

$0101\,q_7\,111$

Weiter in der Definition der TM

- Die Starthonfiguration der TM M auf Eingabe w ist "$q_0 w$".
- Eine Konfiguration heißt akzeptierend / verwerfend, wenn der Zustand q_{accept} / q_{reject} ist.

Weiter in der Definition der TM

Eine Berechnung der TM M auf Eingabe w ist eine "gültige" Folge von Konfigurationen $C_0, C_1, C_2, \ldots$ so daß C_0 die Startkonfiguration ist und die Konfiguration C_{i+1} jeweils wie in der Übergangsfunktion beschrieben aus C_i hervorgeht.

Weiter in der Definition der TM

Eine Berechnung einer TM auf Eingabe w heißt akzeptierend / verwerfend, falls sie im Zustand q_{accept} / q_{reject} endet.

Eine Berechnung heißt nicht-akzeptierend, falls sie entweder in q_{reject} endet oder nie beendet wird.



Die von der TM M akzeptierte Sprache $L(M)$ ist die Menge der Wörter, die von der TM akzeptiert werden.



Bei Wörtern $\notin L$ kann die TM entweder verwerfen oder nie anhalten!

Weiter in der Definition der TM

Eine Sprache L heißt rekursiv aufzählbar (semi-entscheidbar) falls es eine TM M gibt, die L akzeptiert. Das heißt,

$w \in L \Rightarrow$ TM akzeptiert w

$w \notin L \Rightarrow$ TM verwirft oder hält nicht an

Eine Sprache L heißt (rekursiv) entscheidbar, falls es eine TM M gibt, so daß gilt:

$w \in L \Rightarrow$ TM akzeptiert w

$w \notin L \Rightarrow$ TM verwirft w

insb. hält M hier immer an

Offensichtliche Fragen sind:

1. Wie mächtig ist das Modell der TM?

- Welche Sprachen werden von einer TM erkannt?
- Wie sieht die Menge der rekursiv aufzählbaren Sprachen und der entscheidbaren Sprachen aus?
- Sind die beiden Mengen identisch oder nicht?
- Gibt es Sprachen, die weder das eine noch das andere sind?

$\rightsquigarrow$ Berechenbarkeitstheorie

Offensichtliche Fragen sind:

2. Wie "realistisch" / allgemeingültig ist das Modell der TM?

- Können TMs wirklich "das gleiche" wie normale Computer? Oder sind sie zu eingeschränkt?

- Gibt es weitere Berechnungsmodelle, oder vielleicht auch ganz komplett andere Modelle, von denen man andere Dinge beweisen kann?

  → muss alles über Sprachen gehen?

⇝ weitere Berechnungsmodelle und Church-Turing-These

Nicht-deterministische Turing-Maschinen

Literatur: Sipser Kap. 3, Hromkovich Kap. 4

Erste Kritik an Turing-Maschinen

- Ein- und Ausgabe und Speicher ist alles auf dem gleichen Band. Würden getrennte Komponenten mehr bringen?
- Wir können den Speicher nur "linear" durchlaufen, ist das vielleicht zu eingeschränkt?

Betrachten nun verschiedene Varianten von TMs und untersuchen, inwiefern sie sich von "normalen" TMs unterscheiden.

Äquivalenz

Zwei Maschinen M_1, M_2 heißen äquivalent,
falls sie die gleichen Sprachen akzeptieren: $L(M_1) = L(M_2)$.

M_1 akzeptiert w $\Leftrightarrow$ M_2 akzeptiert w

M_1 akzeptiert w nicht $\Leftrightarrow$ M_2 akzeptiert w nicht



verwirft w oder
endet nicht

$A \Rightarrow B$

$\neg B \Rightarrow \neg A$ "Kontraposition"

"Pudel" $\Rightarrow$ "Hund"

$\neg$ Hund $\Rightarrow$ $\neg$ Pudel

Mehrband - Turing - Maschinen

- Wie eine "normale" TM, nur gibt es k Bänder.
- Jedes Band hat einen eigenen Lese/Schreibkopf.
- Es gibt eine globale Zustandsmenge (insbesondere also je einen globalen Zustand q_{start}, q_{accept}, q_{reject})
- Es gibt eine globale Übergangsfunktion δ

$$\delta : Q \times \Gamma^k \longrightarrow Q \times \Gamma^k \times \{L, R\}^k$$

- δ: momentaner Zustand
- $Q \times \Gamma^k$: gelesene Buchstaben an den k Köpfen
- Q: neuer Zustand
- Γ^k: zu schreibende Buchstaben auf den k Bändern
- $\{L, R\}^k$: Bewegung der k Köpfe

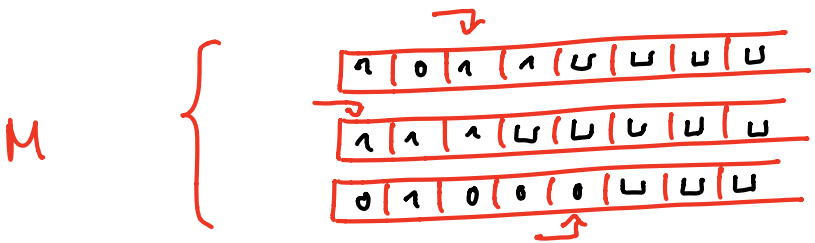
Behauptung: Mehrband-TMs sind äquivalent zu normalen (Einband-) TMs

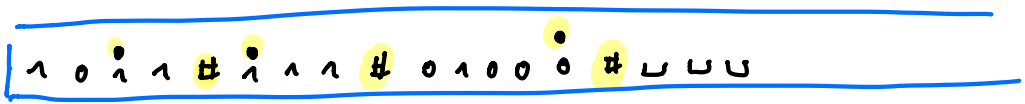
d.h. zu jeder Mehrband-TM M (multi-tape) gibt es eine Einband-TM S (single tape) so daß gilt: $L(M) = L(S)$

Beweisskizze

Gegeben eine Mehrband-TM M. Wir wollen eine Einband-TM S konstruieren, die M "simuliert".

- Speichern die Information der k Bänder von M auf einem Band von S, abgetrennt durch ein neues Symbol "#".
- Markieren die Positionen der Schreibköpfe von M mit einem Punkt über dem entsprechenden Symbol auf dem Band von S ($\dot{a}$ statt a).





Beweisskizze

Ein Berechnungsschritt von S sieht so aus:

- Wir scannen das komplette Band, um die derzeitigen Symbole unter den Köpfen zu lesen
- Wir gehen nochmal durch das Band, um alles entsprechend der k Übergangsfunktionen upzudaten.
- Um auf einem der virtuellen Bändern ein neues Symol schreiben zu können, schieben wir alles, was rechts davon steht, ein Kästchen weiter.

Es ist "intuitiv klar", daß diese Einband-TM S äquivalent zur Mehrband-TM M ist. Formalen Beweis lassen wir weg.

□

Nicht-deterministische Turing-Maschinen (NTM)

Können wie bei endlichen Automaten eine nicht-deterministische TM definieren: (= NTM)

- Zu jedem Zeitpunkt hat die TM mehrere Möglichkeiten, wie sie fortfahren kann.
- Formal gibt dann die Übergangsfunktion

$$\delta: Q \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

- Eine Berechnung der Nichtdeterministischen TM entspricht einem Pfad im Baum der möglichen Pfade, die die TM nehmen kann.

⚠ Manche Äste können unendlich lange sein, wenn die entsprechende Berechnung nie anhält.

Nicht-deterministische Turing-Maschinen (NTM)

Eine Berechnung der NTM entspricht einem möglichen Pfad im "Baum der möglichen Berechnungen".

Eine Berechnung heißt akzeptierend / verwerfend, falls sie in einem akzeptierenden / verwerfenden Zustand endet.

- Die von der NTM akzeptierte Sprache besteht aus den Wörtern, für die es eine akzeptierende Berechnung gibt:

  d.h. mind. einer der Pfade im Berechnungsbaum endet im akzeptierenden Zustand



Ein Pfad im Baum kann akzeptierend sein, während es gleichzeitig Pfade gibt, die nicht terminieren oder sogar in einem verwerfenden Zustand enden.

Wir haben gar nicht definiert, was es heißt, daß eine nicht-deterministische TM verwirft oder nicht anhält.

(Wollte man das definieren, dann so: NTM verwirft Wort w, wenn alle Pfade im reject-Zustand enden).

Notation: NTM, DTM

Um deterministische (= "normale") TMs von nicht-deterministischen zu unterscheiden, benutzen wir zwei Abkürzungen:

NTM

↑

nicht-deterministische TM

DTM

↑

deterministische TM

Behauptung: NTMs sind äquivalent zu DTMs

d.h. zu jeder NTM N gibt es eine TM D

so daß gilt: $L(N) = L(D)$.



Bei DTMs: nicht akzeptieren $\hat{=}$ verwerfen oder nicht terminieren

Bei NTMs haben wir diese beiden Begriffe gar nicht eingeführt. Wir benutzen hier einfach die logische Verneinung:

N akzeptiert w nicht $\Leftrightarrow \neg$ (N akzeptiert w)

$\Leftrightarrow \neg \exists$ Pfad: Pfad akzeptiert

$\Leftrightarrow \forall$ Pfade: $\neg$(Pfad akzeptiert)

$\Leftrightarrow \forall$ Pfade: Pfad verwirft oder endet nicht

Erinnerung: Beweis DFA $\hat{=}$ NFA

Wir haben schon bewiesen, dass deterministische und nichtdeterministische Automaten äquivalent sind.

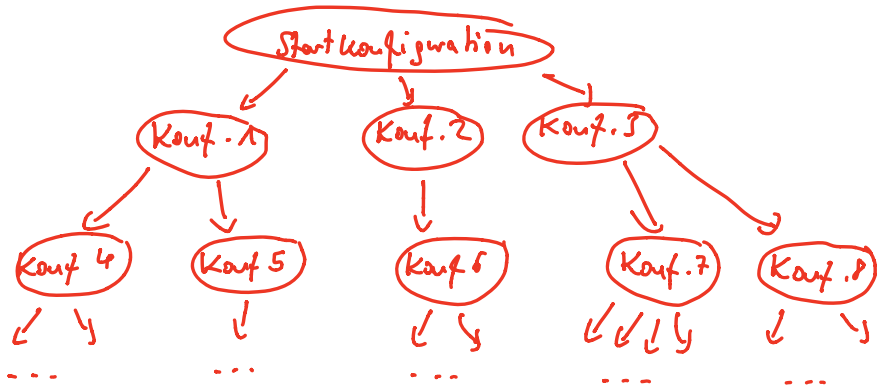
Erinnern Sie sich an den Beweis?

$\rightsquigarrow$ Potenzmengenkonstruktion

Wir verwenden nun ein ähnliches Prinzip, müssen aber aufpassen, daß nicht terminierende Berechnungen nicht alles kaputt machen.

Beweisskizze DTM $\hat{=}$ NTM:

Breitensuche (!)
im Berechnungsbaum:



D geht den Berechnungsbaum von N Schicht für Schicht durch und schaut nach, ob einer der Zustände akzeptierend ist.

Sobald D einen akzeptierenden Zustand findet, akzeptiert D auch. Offensichtlich gilt dann:

N akzeptiert w $\Leftrightarrow$ D akzeptiert w

□



NTMs sind (Berechenbarkeits-) äquivalent zu DTMs,
d.h. es werden die gleichen Sprachen erkannt.

Trotzdem können NTMs "effizienter" arbeiten.
Das wird sehr wichtig im Kapitel der
Komplexitätstheorie, wo wir vermuten,
daß NTMs nicht (komplexitäts-) äquivalent
zu DTMs sind.

Entscheidbarkeit von Sprachen

vs

Berechenbarkeit von Funktionen

Wiederholung: aufzählbare / entscheidbare Sprache

Eine Sprache L heißt rekursiv aufzählbar (semi-entscheidbar) falls es eine TM M gibt, die L akzeptiert. Das heißt,

$w \in L \Rightarrow$ TM akzeptiert w

$w \notin L \Rightarrow$ TM verwirft oder hält nicht an

Eine Sprache L heißt (rekursiv) entscheidbar, falls es eine TM M gibt, so daß gilt:

$w \in L \Rightarrow$ TM akzeptiert w

$w \notin L \Rightarrow$ TM verwirft w

Berechenbare Funktion

Def Eine Funktion $f: \Sigma^* \to \Gamma^*$ heißt (Turing-) berechenbar oder totalrekursiv, falls es eine TM gibt, die bei Eingabe von $w \in \Sigma^*$ den Funktionswert $f(w)$ ausgibt (und insbesondere anhält).

↓

Inhalt des Bandes, wenn die Maschine stoppt.

Wollen nun zeigen: Entscheidbarkeit und Berechenbarkeit sind zwei Seiten der gleichen Medaille.

Entscheidbarkeit $\rightsquigarrow$ Berechenbarkeit

Behauptung:

Eine Sprache $L \subseteq \Sigma^*$ ist entscheidbar genau dann wenn ihre charakteristische Funktion

$$\mathbb{1}_L : \Sigma^* \to \{0,1\}, \quad \mathbb{1}_L(w) = \begin{cases} 1 & w \in L \\ 0 & \text{sonst} \end{cases}$$

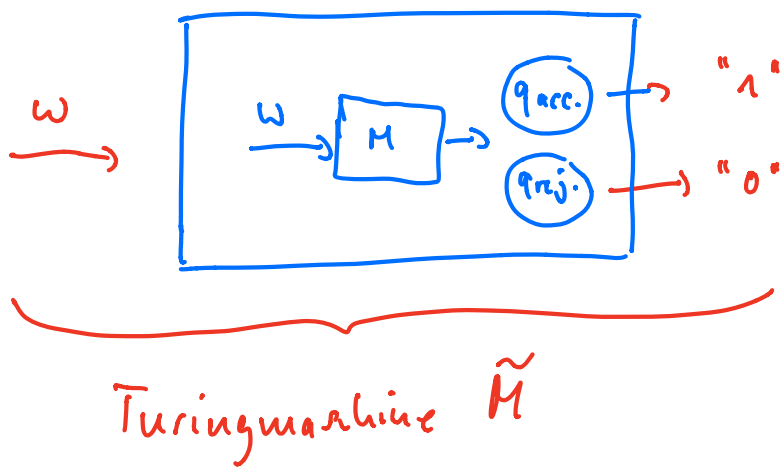
berechenbar ist.

Intuition: Ein Entscheidbarkeitsproblem kann man in ein Berechenbarkeitsproblem umwandeln.

Entscheidbarkeit ⇝ Berechenbarkeit

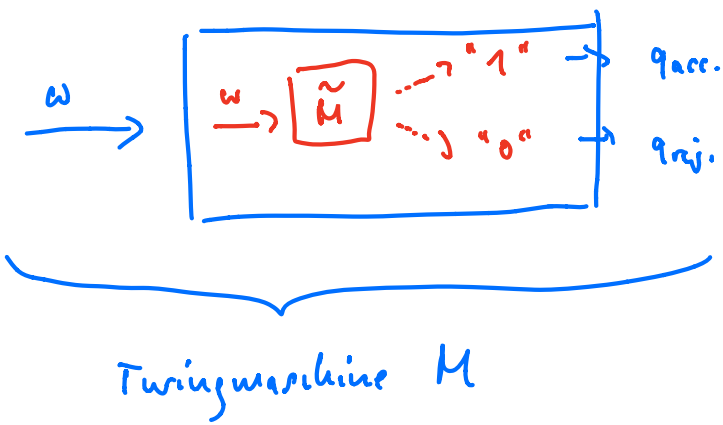
Beweis: "⇒" Wollen zeigen: L entscheidbar ⇒ $\mathcal{H}_L$ berechenbar.

Das ist einfach. Wir haben eine TM M, die w genau dann akzeptiert, wenn $w \in L$. Wir erweitern diese TM folgendermaßen zu $\tilde{M}$:
Falls M akzeptiert, schreibe 1 auf Band und lösche alles andere und stoppe. Falls M verwirft, schreibe 0 auf Band.



Entscheidbarkeit $\rightsquigarrow$ Berechenbarkeit

"$\Leftarrow$" Auch einfach. Haben $\tilde{M}$, die $\mathbb{1}_L$ berechnet. Wenn diese TM 1 zurückgibt, dann begeben wir uns in akzeptierenden Zustand. Bei 0 in den verwerfenden Zustand.



$\square$

Berechenbarkeit $\rightsquigarrow$ Entscheidbarkeit

Behauptung Eine Funktion $f: \Sigma^* \to \Gamma^*$ ist **berechenbar** genau dann, wenn es eine TM gibt, die die folgende Sprache **entscheidet**:

$$L_f = \{ w \# g \mid w \in \Sigma^*, g \in \Gamma^*, f(w) = g \}$$

$\hat{=} f(w)$

Intuition: Ein Berechenbarkeitsproblem kann man in ein Entscheidbarkeitsproblem umwandeln.

Berechenbarkeit ~ Entscheidbarkeit

Beweis "⇒" Sei f berechenbar, d.h. es gibt eine TM M_1, die bei Eingabe von $w \in \Sigma^*$ die Ausgabe $f(w)$ berechnet.

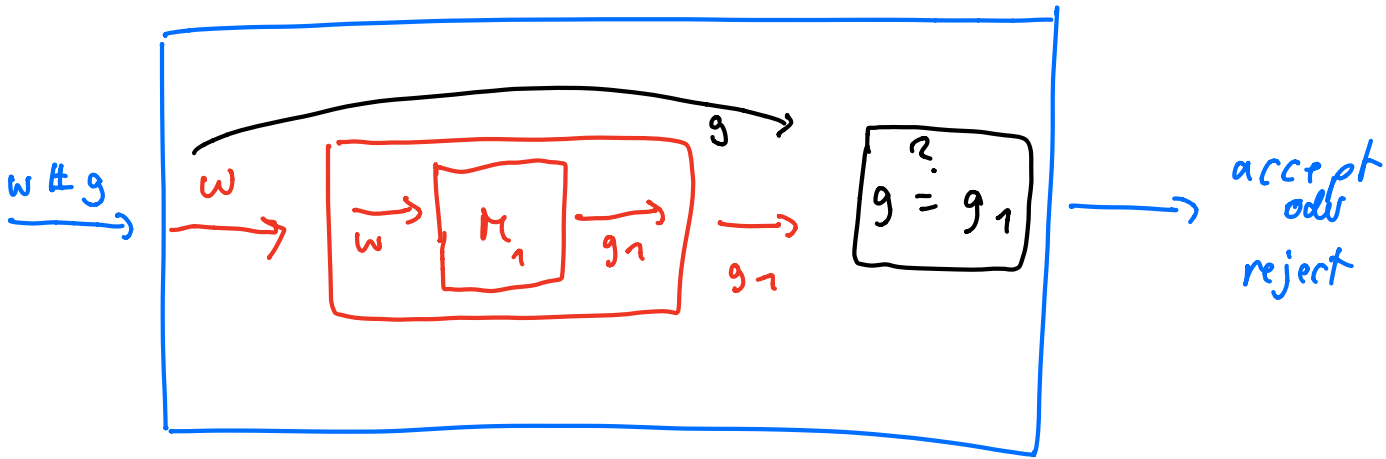
Nun bauen wir eine TM M_2.

Sie bekommt ein Wort der Form $w\#g$ als Eingabe.

Sie ruft nun die TM M_1 mit Eingabe w auf. M_1 kommt immer zum Ende. M_2 wartet auf das Ergebnis g_1 von M_1. Dann vergleicht M_2 g_1 mit dem Eingabestring g und schaut, ob es übereinstimmt.

~> Bild s. nächste Seite:

Berechenbarkeit ~ Entscheidbarkeit



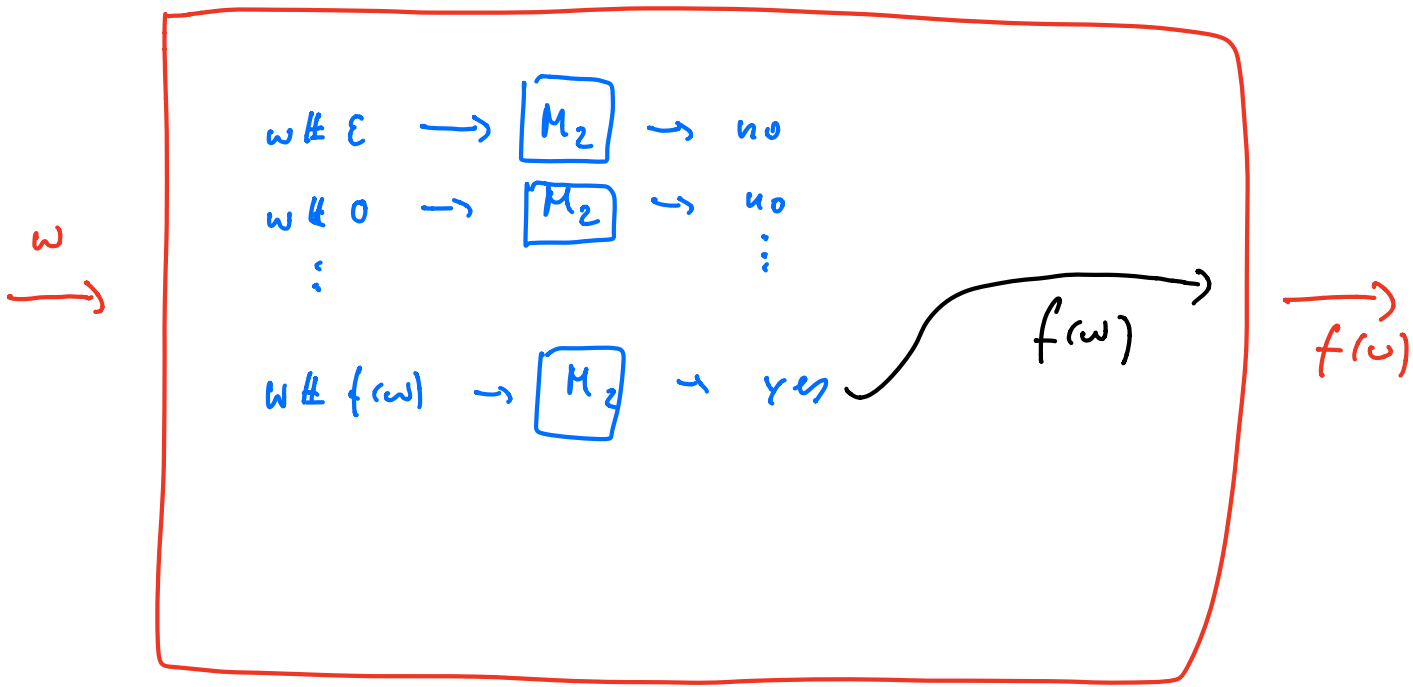
Turingmaschine M_2

Berechenbarkeit ~ Entscheidbarkeit

$\Leftarrow$ Sei M_2 eine TM, die die Sprache L_f entscheidet. Nun bauen wir daraus eine TM M_1, die f berechnet:

- Eingabe in M_1 ist das Wort w.
- Nun probieren wir der Reihe nach alle möglichen "Antworten" von $f(w)$ aus:
  - Ist $w \# \varepsilon$: "Ist $f(w) = \varepsilon$" ? $\rightsquigarrow$ Y/N
  - Ist $w \# 0 \in L_f$? $\rightsquigarrow$ Y/N
  - Ist $w \# 1 \in L_f$? $\rightsquigarrow$ Y/N
  - usw
- Das machen wir solange, bis wir die richtige Antwort "getroffen haben". (Beachte: das muß irgendwann der Fall sein).
- Sobald L_f "Ja" zurückgegeben hat, weiß M_1 die Antwort.

Berechenbarkeit $\rightsquigarrow$ Entscheidbarkeit



Turingmaschine M_1

Gödelnummer und universelle Turing-Maschine

Literatur:
Hromkovic Sec. 4.5 (Sipser 4.2); Wegener 2.4

"Programmierbare" Turingmaschine?

Bisher: Turingmaschine kann genau ein Programm ausführen.

Können wir TM auch "programmieren", so daß sie verschiedene Programme ausführen können?

Codierung einer TM

Gegeben sei eine TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$.

Wollen so eine TM durch einen binären String codieren.

(1) Sei das Arbeitsalphabet $\Gamma = \{A_1, \dots, A_r\}$. Wir codieren jeden Buchstaben durch einen binären String:

$$\text{Code}(A_j) = 1\ 0^j\ 1$$

"Start" "j" "Ende"

Dieselben Codes verwenden wir auch für Buchstaben des Eingabealphabets $\Sigma \subset \Gamma$.

Codierung einer TM

(2) Sei der Zustandsraum $Q = \{q_1, \ldots, q_m\}$ wobei immer q_1 der Startzustand, q_e der akzeptierende, q_s der verwerfende Zustand ist. Wir codieren Zustand i wie folgt:

$$\text{Code}(q_i) = \underbrace{11}_{\text{"Start"}} \; 0^i \; \underbrace{11}_{\text{"Ende"}}$$

(i: Anzahl der Nullen)

(3) Codierung der Bewegungen des Schreibkopfes:

$$\text{Code}(L) = \underbrace{111}_{\text{Start}} \; \underbrace{0}_{L} \; \underbrace{111}_{\text{Ende}}$$

$$\text{Code}(R) = \underbrace{111}_{\text{Start}} \; \underbrace{00}_{R} \; \underbrace{111}_{\text{Ende}}$$

Codierung einer TM

(4) Codierung der Übergangsfunktion: Eine einzelne Transition wird kodiert wie folgt:

$$\text{Code}\left(\underbrace{\delta(q, a) = (\tilde{q}, \tilde{a}, B)}_{\text{eine einzelne Transition}}\right) =$$

$$\#\,\text{Code}(q)\;\text{Code}(a)\;\text{Code}(\tilde{q})\;\text{Code}(\tilde{a})\;\text{Code}(B)\;\#$$

- $\text{Code}(q)$: momentaner Zustand $\in Q$
- $\text{Code}(a)$: gelesener Buchstabe $\in \Sigma$
- $\text{Code}(\tilde{q})$: neuer Zustand $\in Q$
- $\text{Code}(\tilde{a})$: zu schreibender Buchstabe $\in \Gamma$
- $\text{Code}(B)$: Bandübergang $\in \{L, R\}$

Codierung einer TM

(5) Codierung der ganzen TM M:

- Globale Daten: Anzahl m der Zustände, Anzahl r der Buchstaben im Bandalphabet
- Liste aller Transitionen

$$\text{Code}(M) = \#\, 0^m\, \#\, 0^r\, \#\#\, \text{Code}(\text{Transition } 1)\, \#\, \text{Code}(\text{Transition } 2)\, \#\ldots$$

Auf diese Weise haben wir M durch einen eindeutigen String über $\{0,1,\#\}$ beschrieben.

Codierung einer TM

(6) Können diesen String nun in einen rein binären String umwandeln, indem wir auf die drei Buchstaben die folgende Abbildung anwenden:

$$0 \longmapsto 00$$
$$1 \longmapsto 11$$
$$\# \longmapsto 01$$

neu! 

"alt": 0 1 1 # # 0 ..

(Dadurch wird die Länge des Codes verdoppelt.)

Codierung einer TM: Intuition

Intuition: Die Beschreibung der TM in Form eines Tupels $M = (\Sigma, \Gamma, \delta, q_0, q_{acc.}, q_{rej.})$ ist so etwas wie die Beschreibung in einer höheren Programmiersprache.

Der binäre Code ist aber so etwas wie die Beschreibung des Programms in Maschinensprache.

Eigenschaften der Codierung: Eindeutigkeit

Die beschriebene Codierung von Turingmaschinen ist injektiv:

$$\text{Code}(M_1) = \text{Code}(M_2) \Rightarrow M_1 = M_2$$

Jeder Code beschreibt also eindeutig genau eine TM.

Eigenschaften der Codierung: gültige und ungültige Codes können unterschieden werden

Manche binäre Strings sind "gültige Codes" für eine TM. Viele andere binäre Strings codieren keine TM.

Behauptung: Es gibt eine TM $A_{gültig}$, die für einen binären String w entscheidet, ob er eine gültige Codierung einer TM ist.

Anders formuliert: Die folgende Sprache ist entscheidbar:

$$L = \{w \in \{0,1\}^* \mid w \text{ ist Code einer TM}\}$$

Beweis Übung.

Eigenschaften der Codierung: Präfix-frei

Unsere Codierung ist Präfix-frei: kein echtes Präfix einer Codierung $\text{Code}(M)$ ist selbst schon Codierung einer anderen TM $\tilde{M}$.

Beweis Klar aufgrund der Konstruktion.

Das wird später wichtig: bei einer aneinander gehängten Zeichenkette $\text{Code}(M)w$ können wir immer entscheiden, wo $\text{Code}(M)$ aufhört und w anfängt.

Eigenschaften der Codierung: lineare Ordnung

Durch die Codes können wir eine lineare Ordnung auf der Menge aller Turingmaschinen definieren:

Def: $x, y \in \{0,1\}^+$. Wir sagen $x \leq y$, falls die natürlichen Zahlen n_x, n_y, die durch die Strings repräsentiert werden, die Eigenschaft $n_x \leq n_y$ repräsentieren.
Für $x = \varepsilon$, $y \in \{0,1\}^*$ gelt immer $\varepsilon \leq y$.

Bemerkung: Dies ist eine totale Ordnung auf der Menge der binären Strings.

↳ transitiv: $a \leq b$ und $b \leq c \Rightarrow a \leq c$
anti-symmetrisch: $x \leq y$ und $y \leq x \Rightarrow x = y$
total: $x \leq y$ oder $y \leq x$

Eigenschaften der Codierung: linear Ordnung

Def Sei $x \in \{0,1\}^*$. Für $i \in \mathbb{N}$ nennen wir x die Codierung der i-ten Turingmaschine falls gilt:

(i) $x = \text{Code}(M)$ für eine TM M

(ii) Die Menge $\{y \in \{0,1\}^* \mid y \leq x\}$ enthält genau $i-1$ Wörter, die Codierungen von Turingmaschinen sind.

Anschaulich: Wir gehen die Menge der binären Strings der Größe nach durch und numerieren die dabei auftauchenden TMs durch.

Eigenschaften der Codierung: i-k TM berechenbar

Behauptung: Es gibt eine TM A, die für ein $i \in \mathbb{N}$ die Codierung der i-ten TM berechnet.

Beweisidee: Die TM geht die binären Strings in der kanonischen Ordnung durch. Auf jeden String w wendet sie die TM $t_{gültig}$ an (siehe zwei Folien weiter vorne), um zu entscheiden, ob w die Codierung einer TM ist. Falls ja, erhöht sie einen Zähler, der bis zur Zahl i zählt.

Gödelnummer einer TM

Ausführlich:

Ein solcher Code einer TM heißt Gödelnummer der TM.

(es gibt viele Möglichkeiten, Gödelnummern zu bauen; es kommt im Folgenden nicht darauf an, wie wir den Code genau gebaut haben, sondern nur daß er existiert und einige elementare Eigenschaften wie Injektivität erfüllt).

Die Gödelnummer einer TM wird oft auch mit $\langle M \rangle$ bezeichnet. Intuitiv beschreibt die Gödelnummer das "Programm", das von M durchgeführt wird.

Gödelnummer einer TM

Man kann auch formaler definieren, wann eine Codierung eine "Gödelisierung" ist:

allgemeiner: $\mathcal{M}$ beliebige Sprache.

Def Sei $\mathcal{M}$ die Menge aller Turingmaschinen. Die Abbildung $g: \mathcal{M} \to \mathbb{N}$ heißt "Gödelisierung" falls gilt:

(i) g ist injektiv

(ii) Die Bildmenge $g(\mathcal{M})$ ist entscheidbar, d.h. wir können eine TM konstruieren, die für jedes $n \in \mathbb{N}$ entscheidet, ob $n \in g(\mathcal{M})$ gilt oder nicht.

(iii) Die Funktionen $g: \mathcal{M} \to \mathbb{N}$ und $g^{-1}: g(\mathcal{M}) \to \mathcal{M}$ sind berechenbar

Für $M \in \mathcal{M}$ heißt $g(M)$ die Gödel-Nummer von M.

Die universelle TM: Motivation

Motivation: Turing-Maschinen, wie wir sie bisher betrachtet haben, können genau ein "Programm" ausführen und genau eine Sprache entscheiden. Aber ein "echter Computer" kann doch viele Programme ausführen?!?

$\rightsquigarrow$ universelle TM

Die universelle TM: Idee

Wir konstruieren nun eine universelle TM U:

- Als Eingabe erhält U die Gödelnummer $\langle M \rangle$ einer (normalen) TM und ein Wort w, auf das die TM M angewendet werden soll.
- U soll nun die Operation von M auf w simulieren, also M auf w anwenden. Dabei soll gelten:
  - U akzeptiert genau dann, wenn M das Wort w akzeptiert.
  - U hält mit Ausgabe x auf dem Band an, genau wenn M auf w das tut.
- Bei inkorrekter Eingabe (zB wenn $\langle M \rangle$ kein gültiger Code ist), gibt U eine Fehlermeldung aus.

Die universelle TM: Konstruktion

Wir konstruieren U als eine 3-Band-TM:

Die Grundidee ist folgende:

- Auf Band 1 simulieren wir das Band von M.
- Auf Band 2 schreiben wir die Gödelnummer $\langle M \rangle$.
  Er enthält also den Programmcode.
- Auf Band 3 merken wir uns, in welchem Zustand die TM M gerade wäre.

Die universelle TM: Konstruktion

Eingabe:

Am Anfang steht die Eingabe $\langle M \rangle w$ auf dem ersten Band.
Die anderen Bänder sind leer.



Die universelle TM: Konstruktion

Vorbereitung

<M>w

- U liest die Eingabe auf dem ersten Band und teilt sie in die Teile <M> und w auf. (Hier ist wichtig, daß die Codierung Präfix-frei ist!) Falls <M> keine korrekte Codierung ist, bricht U mit einer Fehlermeldung ab.

"Syntax-check"

"Initialisierung"

- Nun kopiert U die Gödelnummer <M> auf das zweite Band und löscht sie vom ersten Band. Der Schreibkopf des ersten Bandes ist am Anfang von w. Auf das dritte Band schreiben wir die Codierung des Startzustands von M:



Die universelle TM: Konstruktion

Simulation eines Rechenschritts von M:

- U weiß auf Band 3, in welchem Zustand wir gerade sind.
- U liest den aktuellen Buchstaben von Band 1
- U schaut auf Band 2 nach dem zugehörigen Übergang
- U führt diesen auf Band 1 durch und merkt sich den neuen Zustand in Band 3.

Ausgabe: U stoppt, sobald auf Band 3 der akzeptierende oder verwerfende Zustand erreicht wird. Auf Band 1 steht dann die Ausgabe der Berechnung (falls gewünscht).

Abzählbar unendliche Mengen

Literatur: Hromkovič 5.2., Sipser 4.2.

Motivation

Wir wollen demnächst zeigen, daß es VIEL mehr Sprachen als Turingmaschinen gibt. Daran kann man dann sehen, dass es Sprachen geben muß, die nicht durch eine TM erkannt werden.

Dazu brauchen wir ein wichtiges Werkzeug: abzählbare und überabzählbare Mengen und Diagonalisierung

Endliche Mengen

Def Eine Menge $M \neq \emptyset$ heißt **endlich**, falls es ein $n_0 \in \mathbb{N}$ gibt, für das es eine bijektive Abbildung

$$g: \{1, 2, \dots, n_0\} \to M$$

gibt. Andernfalls heißt M **unendlich**.

Im endlichen Fall bezeichnen wir mit $|M| := n_0$ die **Kardinalität** oder **Mächtigkeit** der Menge.

$\mathbb{N}$ ist unendlich

Satz: Die Menge $\mathbb{N}$ ist unendlich.

Beweis: Müssen zeigen: $\forall n_0 \in \mathbb{N}\ \forall f : \{1, \dots, n_0\} \to \mathbb{N}$ ist f nicht bijektiv.

Sei also $n_0 \in \mathbb{N}$, $f : \{1, \dots, n_0\} \to \mathbb{N}$ eine Abbildung.

Sei $n_{max} := \max \{ f(i) \mid i \leq n_0 \}$. Setze $m := n_{max} + 1$.

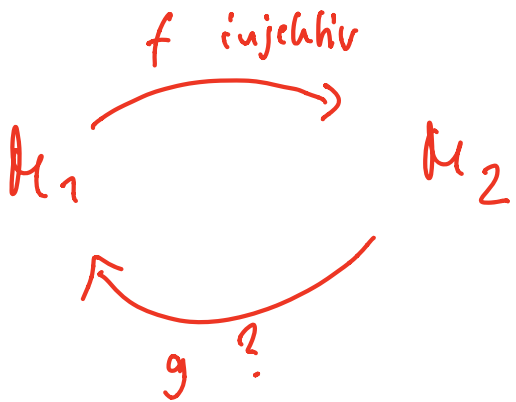
Offensichtlich ist $m \in \mathbb{N}$, aber $m \notin f(\{1, \dots, n_0\})$. Also ist f nicht surjektiv, also nicht bijektiv.

□

Vergleich von Mengen

Def Zwei Mengen M_1, M_2 heißen **gleich mächtig**, falls es eine bijektive Abbildung $M_1 \to M_2$ gibt.

M_2 heißt **mächtiger** als M_1, falls es eine injektive Abb $f: M_1 \to M_2$ und keine injektive Abb. $g: M_2 \to M_1$ gibt.



Abzählbar Mengen

Def Eine Menge M heißt **abzählbar unendlich** (kurz: **abzählbar**), wenn sie gleichmächtig wie $\mathbb{N}$ ist: es gibt eine Bijektion $f: M \to \mathbb{N}$.

Eine Menge, die weder endlich noch abzählbar ist, heißt **überabzählbar**.

Eine Menge M heißt **höchstens abzählbar**, wenn sie endlich oder abzählbar unendlich ist.

$\mathbb{N}$ ist abzählbar

Klar, oder? Wir müssen eine Bijektion $f: \mathbb{N} \to \mathbb{N}$ konstruieren. Die Identität tut's:

$f: \mathbb{N} \to \mathbb{N}$, $n \mapsto n$ ist bijektiv.

$\mathbb{N}_0$ ist abzählbar

$\mathbb{N}_0 = \{0, 1, 2, \dots\}$

Auch schnell zu sehen. Wir konstruieren eine Bijektion

$$f: \mathbb{N}_0 \to \mathbb{N}, \quad n \mapsto n+1.$$ Tut's.

(Bem: $\mathbb{N}_0$ ist gleich mächtig wie $\mathbb{N}$).

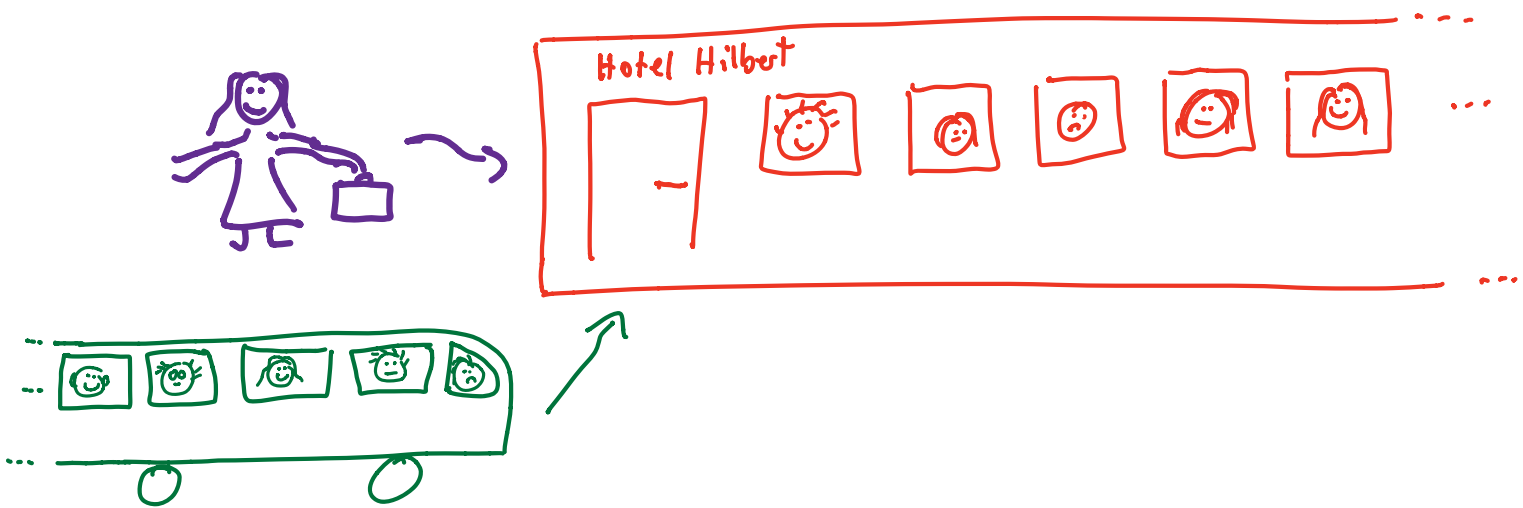
$2\mathbb{N}$ ist abzählbar

$2\mathbb{N} = \{2, 4, 6, 8, 10, \dots\}$. Können Sie die Bijektion hinschreiben?

Bemerkung: es ist egal, wie rum wir die Bijektion definieren:
Wenn es eine Bijektion $f: M \to \mathbb{N}$ gibt, dann gibt es
auch eine Bijektion $g: \mathbb{N} \to M$. Können Sie es beweisen?

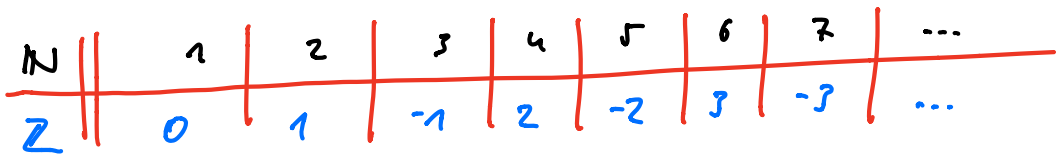
Hilberts Hotel

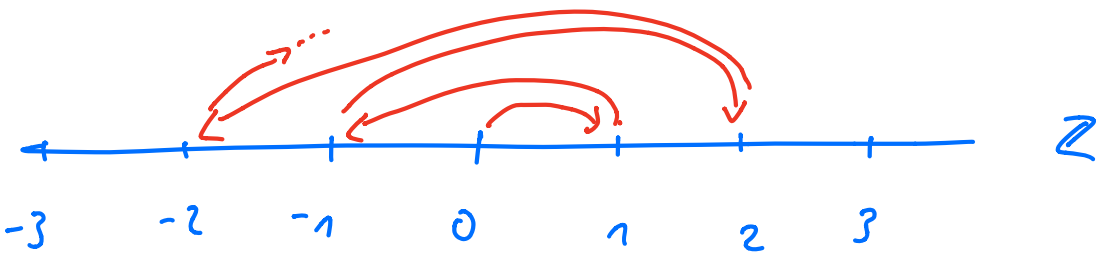
Hotel mit unendlich vielen Zimmern. Alle besetzt.
Nun kommt noch ein neuer Gast. Bringen wir sie noch unter?
Und nun kommt ein weiter Bus mit abzählbar vielen Leuten dran an. Was nun?



$\mathbb{Z}$ ist abzählbar

Konstruiere Bijektion $f: \mathbb{N} \to \mathbb{Z}$ wie folgt:





$\mathbb{Z}$ "abzählen" (in schlauer Reihenfolge)

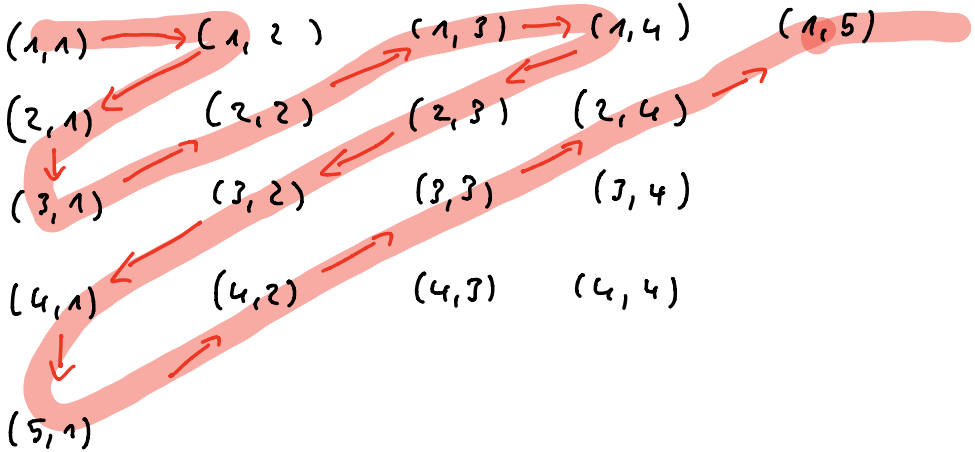
Ist $\mathbb{N}^2$ abzählbar?

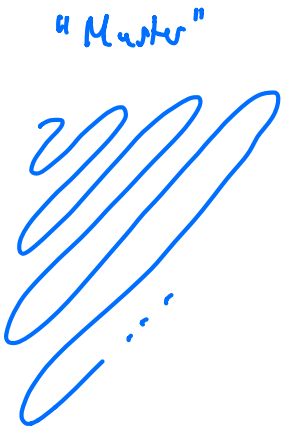
$$\mathbb{N}^2 = \mathbb{N} \times \mathbb{N} = \{ (i,j) \mid i,j \in \mathbb{N} \}$$

... überlegen Sie mal eine Weile ...

$\mathbb{N}^2$ ist abzählbar

Konstruieren Bijektion von $\mathbb{N}^2 \to \mathbb{N}$ wie folgt:



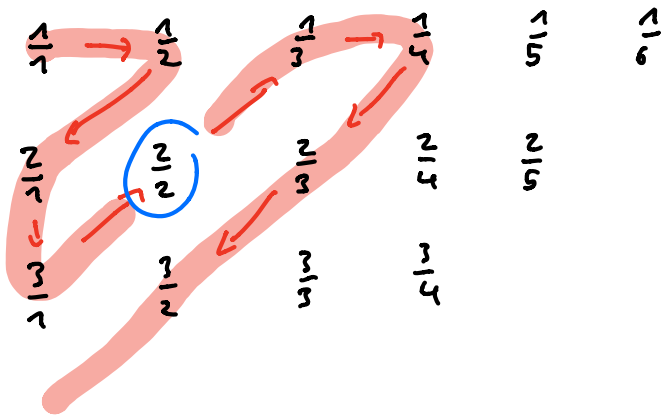


Intuitiv klar: Abbildung ist bijektiv.

$\mathbb{Q}$ ist abzählbar !!!

$\mathbb{Q}$ - Menge der rationalen Zahlen, $\mathbb{Q} = \left\{ \frac{m}{n} \mid m, n \in \mathbb{Z},\ n \neq 0 \right\}$

Schritt 1: Wir konstruieren eine Bijektion $\mathbb{N} \to \mathbb{Q}^+$ wie folgt:



Die Zahlen, die wir schon mal gesehen haben, lassen wir aus, zum Beispiel

$\frac{1}{1} = \frac{2}{2} = \frac{3}{3} \ldots$

Schritt 2 Analog für $\mathbb{Q}^-$

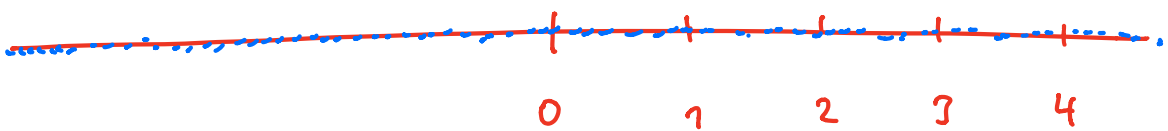
Schritt 3 Wenn M, N abzählbar sind, dann ist auch $M \cup N$ abzählbar.

Verdauen wir mal ...

$\mathbb{N} \subset \mathbb{Q}$, und $\mathbb{Q}$ "sieht viel größer aus" als $\mathbb{N}$.

Trotzdem haben wir es geschafft, eine Bijektion zu bauen.

Cool, oder?.



$\mathbb{Q}$ liegt dicht in $\mathbb{R}$, $\mathbb{N}$ ist offensichtlich nicht dicht

Achtung!

Satz Seien M, N endliche Mengen mit $|M| = |N|$, und $f: M \to N$ Abbildung. Dann sind äquivalent:

(i) f injektiv (ii) f surjektiv (iii) f bijektiv.

Bew Durch Ringschluß: (i) $\Rightarrow$ (ii) $\Rightarrow$ (iii) $\Rightarrow$ (i). Übung.

Dieser Satz gilt nicht für unendliche Mengen!

Andere Charakterisierung unendlicher Mengen

Satz Eine Menge M ist genau dann unendlich, wenn es eine echte Teilmenge $U \subset M$, $U \neq M$ und eine injektive Abbildung $f: M \to U$ gibt.

Bew lassen wir weg

$\mathbb{N}$ ist die "kleinste" unendliche Menge

Satz: Sei M eine beliebige unendliche Menge.
Dann ist $\mathbb{N}$ nicht mächtiger als M.

Bew lassen wir weg, das führt weiter in die Mengenlehre (und welche Axiome wir genau verwenden wollen).

Operationen auf abzählbaren Mengen

Satz (a) Sei A höchstens abzählbar, $f: A \to B$ bijektiv.
Dann ist auch B höchstens abzählbar.

(b) M abzählbar, $N \subset M$. Dann ist N endlich oder abzählbar.

(c) Abzählbare Vereinigungen von abzählbaren Mengen sind abzählbar:
Seien $M_1, M_2, \ldots$ abzählbare Mengen. Dann ist
$\bigcup_{i \in \mathbb{N}} M_i$ auch abzählbar.

(d) $M_1, M_2, \ldots, M_n$ abzählbar. Dann $M_1 \times M_2 \times \ldots \times M_n$ abzählbar.

⚠ Gilt nicht für unendliche Produkte! Sehen später, daß $2^{\mathbb{N}}$ überabzählbar ist.

$2^{\mathbb{N}} = \{0,1\} \times \{0,1\} \times \{0,1\} \times \ldots$
z.B. $(0, 1, 0, 1, 0, 1, \ldots)$

Beweise (a) und (b): Übung

Beweisskizze (c):

Da jede Menge M_i abzählbar ist, gibt es bijektive Abb. $f_i: \mathbb{N} \to M_i$. Wir können M_i also schreiben als

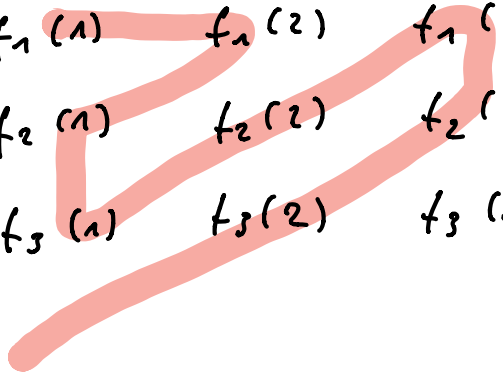
$$M_i = \{ f_i(1), f_i(2), f_i(3), \dots \}$$

Nun konstruieren wir eine "Tabelle":

$f_1(1)$ $f_1(2)$ $f_1(3)$ $f_1(4)$ ... $\rightsquigarrow$ Elemente von M_1

$f_2(1)$ $f_2(2)$ $f_2(3)$ $f_2(4)$ ... $\rightsquigarrow$ Elemente von M_2

$f_3(1)$ $f_3(2)$ $f_3(3)$ ... $\rightsquigarrow$ Elemente von M_3



Tut's!

Beweisskizze (d)

Wir zeigen: Für alle $n \in \mathbb{N}$ ist $\mathbb{N}^n$ abzählbar, durch Induktion:

IA: $n = 1$, klar $\mathbb{N}^1 = \mathbb{N}$

IS: $n \rightsquigarrow n+1$

$$\mathbb{N}^{n+1} = \underbrace{\mathbb{N}^n}_{} \times \mathbb{N}$$

abzählbar nach Indukt. voraussetzung: $\exists\, g: \mathbb{N} \to \mathbb{N}^n$ bij.

typo Farbe farch

Schreiben "Tabelle":

$$\begin{array}{ccc}
(g(1), 1) & (g(1), 2) & (g(1), 3) \\
(g(2), 1) & (g(2), 2) & (g(2), 3) \\
(g(3), 1) & (g(3), 2) & (g(3), 3)
\end{array}$$

Tut's. $\square$

Wie groß ist Σ^*?

Sei Σ ein endliches, nicht-leeres Alphabet. Wie groß ist Σ^*???

Erste Einsicht: Σ^* ist unendlich.

Beweis: Σ enthält mind. einen Buchstaben. Sei also $a \in \Sigma$. Dann enthält Σ^* die Wörter $a, a^2, a^3, \ldots$

Also finden wir eine injektive Abb. $f: \mathbb{N} \to \Sigma^*$, $f(i) = a^i$. Dann muß Σ^* unendlich groß sein.

□

Σ^* ist abzählbar unendlich.

Satz Sei Σ ein endliches Alphabet. Dann ist die Menge Σ^* abzählbar unendlich.

Beweis Wir müssen eine bij. Abb. $f: \mathbb{N} \to \Sigma^*$ finden, d.h. jedem Wort aus Σ^* eine "Nummer geben". Dazu schreiben wir die Worte von Σ^* der Länge nach geordnet auf:

$\Sigma^* =$ { Länge 0 } $\cup$ { Wort Länge 1 } $\cup$ { Wort der Länge 2 } $\cup$ ...

ε — Buchstaben — $\{aa, ab, ac, \dots\}$

1 Wort — $|\Sigma|$ Worte — $|\Sigma|^2$ viele Worte — ...

In dieser Reihenfolge numerieren wir alle Worte durch.

→

$f: \mathbb{N} \to \Sigma^*$

Müssen nicht stellen, daß diese Numerierung "bijektiv" ist, d.h.:

Wohldefiniert: Jedes $n \in \mathbb{N}$ erhält genau ein Bildwort $f(n) \in \Sigma^*$

Klar, da die Liste der Worte nie aufhört, da wir beliebig lange Worte in Σ^* haben.

Surjektiv: • Jedes Wort von Σ^* kriegt (mindestens) eine Nummer

Klar: jedes Wort hat eine endliche Länge, d.h. es taucht in einer der blauen Mengen auf. Außerdem hat jede blaue Menge nur endlich viele Elemente, so daß wir sicher sein können, daß die Numerierung dort auch ankommt.

Injektiv: • Jedes Wort von Σ^* kriegt genau eine und nicht mehrere Nummern. Klar, denn ein Wort taucht in genau einer der blauen Mengen auf. ∎

Sprachen sind höchstens abzählbar.

Satz: Sei L eine Sprache über einem endlichen Alphabet.
Dann ist L höchstens abzählbar.

Beweis: Folgt direkt aus dem vorigen Satz.

Wie viele Turingmaschinen gibt es?

Satz Die Menge aller Turingmaschinen ist abzählbar.

Beweis Wir haben gesehen, daß wir eine Turingmaschine M eindeutig durch eine Gödelnummer $\langle M \rangle \in \{0,1\}^*$ beschreiben können. Und daß $\{0,1\}^*$ selbst eine abzählbare Menge ist. Also kann es nur abzählbar viele TMs geben.

□

Überabzählbare Mengen und Diagonalisierung

Literatur: Hromkovič 5.2., Sipser 4.2.

Wiederholung

Def Eine Menge $M \neq \emptyset$ heißt **endlich**, falls es ein $n_0 \in \mathbb{N}$ gibt, für das es eine bijektive Abbildung

$$g: \{1, 2, \dots, n_0\} \to M$$

gibt. Andernfalls heißt M **unendlich**.

Def Eine Menge M heißt **abzählbar unendlich** (kurz: **abzählbar**), wenn sie gleich mächtig wie $\mathbb{N}$ ist: es gibt eine Bijektion $f: M \to \mathbb{N}$.

Eine Menge, die weder endlich noch abzählbar ist, heißt **überabzählbar**.

Welche Mengen sind überabzählbar, und wie beweist man das?

$\mathbb{R}$ ist überabzählbar

Satz: Die Menge der reellen Zahlen ist überabzählbar.

Der Beweis verwendet eine wichtige Technik, nämlich den Diagonalisierungs-Trick von Cantor:

Beweis: durch Widerspruch.

Annahme: $\mathbb{R}$ ist abzählbar, d.h. es gibt eine bijektive Abbildung $g: \mathbb{N} \to \mathbb{R}$. Wir schreiben nun die reellen Zahlen in dieser Reihenfolge hin:

$\mathbb{R}$ ist überabzählbar: Beweis

$g(1) = 0,\ 7\ 1\ 6\ 2\ 3\ 8\ 4\ \ldots$

$g(2) = 0,\ 1\ 6\ 7\ 6\ 1\ 5\ 4\ 8$

$g(3) = 0,\ 8\ 7\ 5\ 1\ 2\ 4\ 3\ 3$

$\vdots$

$g(i) = 0,\ 6\ 1\ 5\ \ldots\ \underbrace{6}_{i}\ \ldots$

Nun konstruieren wir eine neue reelle Zahl x wie folgt:

- x stimmt an der ersten Nachkommastelle nicht mit $g(1)$ überein. Sagen wir, $x = 0,3\ldots$
- x stimmt an der zweiten Nachkommastelle nicht mit $g(2)$ überein. Sagen wir, $x = 0,38\ldots$
- x stimmt an der i-ten Nachkommastelle nicht mit $g(i)$ überein, sagen wir $x = 0,38\ldots4\ldots$

$\mathbb{R}$ ist überabzählbar: Beweis

Nun gilt aber: $\forall n \in \mathbb{N}: \quad x \neq g(n)$

Also ist $x \notin g(\mathbb{N})$. Die Abbildung g war also nicht surjektiv. ↯

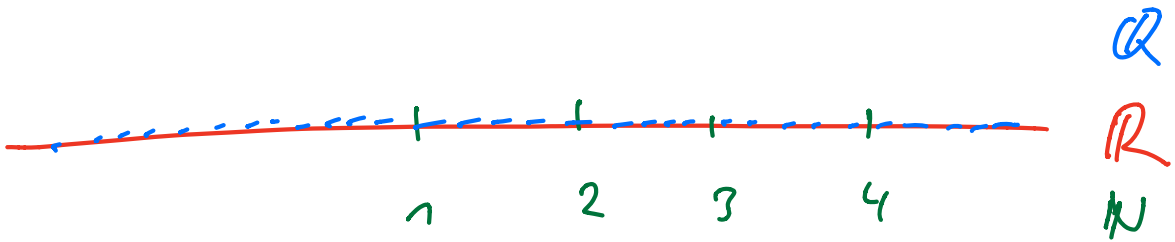
□

Diese Technik heißt "Diagonalisierung", weil wir die Dezimalstellen auf der "Diagonalen" abändern

$g(1) = 0,\ 7\ 1\ 6\ 2\ 3\ 8\ 4\ \ldots$
$g(2) = 0,\ 1\ 6\ 7\ 6\ 1\ 5\ 4\ 8$
$g(3) = 0,\ 8\ 7\ 5\ 1\ 2\ 4\ 3\ 3$
$\vdots$

$g(i) = 0,\ 6\ 1\ 5\ \ldots\ \underbrace{6}_{i}\ \ldots$

Verblüffend, oder?



$\mathbb{Q} \subset \mathbb{R}$

$\mathbb{Q}$: abzählbar

$\mathbb{R}$: überabzählbar

$2^{\mathbb{N}}$ ist überabzählbar

Satz: Die Menge $2^{\mathbb{N}} := \{0,1\}^{\mathbb{N}} := \{(a_i)_{i \in \mathbb{N}} \mid a_i \in \{0,1\}\}$ ist überabzählbar.

alle möglichen Folgen aus 0 und 1

Beweis (Skizze): Diagonalbeweis: Sei $g: \mathbb{N} \to 2^{\mathbb{N}}$ bijektiv.

$$
\begin{aligned}
g(1) &= 0\ 1\ 0\ 0\ 1\ 1\ 0\ \ldots \\
g(2) &= 1\ 1\ 0\ 1\ 1\ 1\ 0\ 0\ \ldots \\
g(3) &= 0\ 0\ 1\ 1\ 0\ 1\ 0\ 1\ \ldots
\end{aligned}
$$

Betrachte $a = (a_i)_{i \in \mathbb{N}}$ mit

$$
a_i = \begin{cases} 1 & \text{falls die } i\text{-te Ziffer von } g(i) = 0 \\ 0 & \ldots \quad 1 \end{cases}
$$

Offensichtlich $a \notin g(\mathbb{N})$, also g nicht surjektiv. $\square$

Bem: $2^{\mathbb{N}} \approx [0,1]$

Bemerkung: Was haben die Mengen $2^{\mathbb{N}}$ und das Intervall $[0,1] \subset \mathbb{R}$ gemeinsam?

Antwort: Wir können die Zahlen in $[0,1]$ als Binärfolge darstellen. Kriegen (ungefähr) eine Bijektion, müssen nur aufpassen mit Zahlen der Form $0,1010011\overline{1}\ldots$
↘ "Periode 1"

Bem: $2^{\mathbb{N}} \approx$ Indikatorfunktionen einer abzählbaren Menge

Sei M eine abzählbare Menge: $M = \{m_1, m_2, m_3, \dots\}$

Betrachte nun alle Teilmengen $U \subset M$: Können jede Teilmenge durch einen Indikatorvektor darstellen:

$$\mathbb{1}_U(i) := \begin{cases} 1 & \text{falls } m_i \in U \\ 0 & \text{sonst} \end{cases}$$

Beispiel: $U = \{m_1, m_3, m_4, m_7\}$.

$$\mathbb{1}_U = (1, 0, 1, 1, 0, 0, 1, 0, 0 \dots)$$

Es gibt also überabzählbar viele Teilmengen von M.

Potenzmengen sind mächtiger

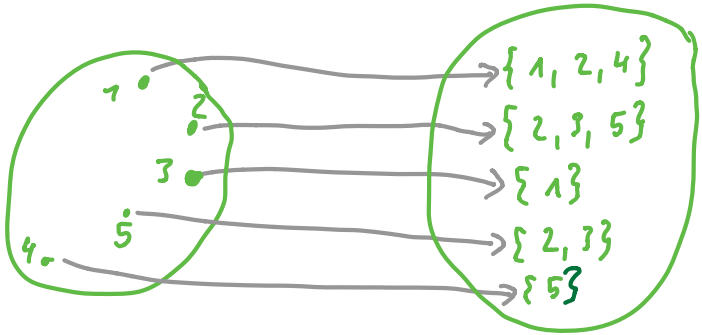
Satz Sei M eine beliebige Menge und $\mathcal{P}(M) = \{U \mid U \subset M\}$ ihre Potenzmenge. Dann gibt es keine surjektive Abbildung $\varphi : M \to \mathcal{P}(M)$. Die Potenzmenge ist also immer echt mächtiger als die Menge selbst.

Beweis Verwendet auch eine Art Diagonalstich, aber einmal mehr um die Ecke gedacht:

Potenzmengen sind mächtiger: Beweis

Ann: $e : M \to \mathcal{P}(M)$ ist surjektive Abbildung.

$M = \{1, 2, \dots, 5\}$



Betrachte $A := \{m \in M \mid m \notin e(m)\} \subset M$

nach Def: $e(m)$ ist ein Element der Potenzmenge, also eine Teilmenge E von M.

Im Beispiel:

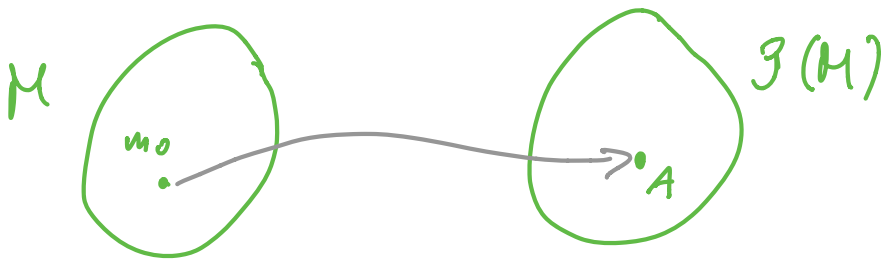
$1 \in e(1) = \{1, 2, 4\} \Rightarrow 1 \notin A$

$3 \notin e(3) = \{1\} \Rightarrow 3 \in A$

Potenzmengen sind mächtiger: Beweis

Betrachte $A := \{m \in M \mid m \notin e(m)\}$

A ist also eine Teilmenge (möglicherweise leer) von M, also $A \in \mathcal{P}(M)$. Da die Abbildung e surjektiv ist, muß es also ein $m_0 \in M$ geben mit $e(m_0) = A$.



Nun betrachten wir die Frage "Ist m_0 selbst in A?"

- Falls $m_0 \in A$, dann nach Definition $m_0 \notin e(m_0) = A$. ↯
- Falls $m_0 \notin A$, dann nach Definition $m_0 \in e(m_0) = A$. ↯

∎

Potenzmengen sind mächtiger: Konsequenz

Folge aus dem vorigen Satz: es gibt "beliebig viele" Stufen der Unendlichkeit:

$$\mathbb{N} < \mathcal{P}(\mathbb{N}) < \mathcal{P}(\mathcal{P}(\mathbb{N})) < \dots$$

$\rightsquigarrow$ Ordinalzahlen, Mengenlehre!

↳ Literatur: Ebbinghaus: Mengenlehre

Ähnlich: Friseur rasiert alle Leute, die sich nicht selbst rasieren.

Rasiert der Friseur sich selbst?

- Ja ... Kann nicht sein
- Nein ... Kann nicht sein.



~> Russells Paradox:

Exkurs: Russells Paradox

Betrachte die "Menge" $R = \{x \mid x \notin x\}$ aller Elemente, die sich nicht selbst enthalten.

- Meistens enthalten sich Mengen nicht selbst: die Menge der natürlichen Zahlen ist nicht selbst eine natürliche Zahl.
- Aber es könnte vorstellbar sein: Die "Menge aller abstrakten Ideen" könnte selber eine abstrakte Idee sein.

Ist nun $R \in R$ oder nicht? ⇝ ↯

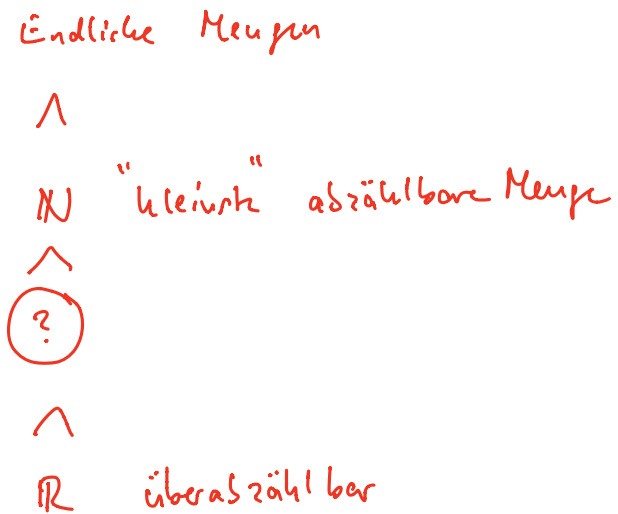
Lösung des Paradoxes: R ist keine Menge!

⇝ Zermelo – Fraenkel – Mengenlehre, Aussonderungsaxiom.

Ausblick: Kontinuumshypothese

Gibt es eine Menge, die echt mächtiger ist als $\mathbb{N}$, aber echt kleiner als $\mathbb{R}$?

Kontinuumshypothese: Nein: Es gibt keine Menge, deren Kardinalität echt größer als die von $\mathbb{N}$ und echt kleiner als die von $\mathbb{R}$ ist.



Ausblick: Kontinuumshypothese

- Zuerst betrachtet von Cantor, 1878
- Eines der Hilbert'schen 23 Probleme, 1900
- Gödel + follow-up, 1940–1960: Die Kontinuumshypothese ist unabhängig von den üblichen Axiomen der Mengenlehre Zermelo–Fraenkel + Auswahlaxiom (ZFC).

Konsequenz: können zwei Versionen der Mengenlehre bauen: eine, in der wir die Hypothese als Axiom aufnehmen; eine zweite, in der wir das Gegenteil als Axiom aufnehmen.

Die Menge aller Sprachen über $\{0,1\}$ ist überabzählbar.

Satz: Sei Σ ein endliches Alphabet. Dann ist die Menge aller Sprachen L über Σ überabzählbar groß.

Beweis Nach Definition ist eine Sprache L eine Teilmenge von Σ^*.
Wir haben schon gesehen, daß Σ^* abzählbar unendlich ist.
Wir können die Wörter also durch numerieren durch $w_1, w_2, w_3, \ldots$.
Eine Sprache können wir nun durch einen abzählbar langen Indikatorvektor beschreiben: $a_i = \begin{cases} 1 & \text{falls } w_i \in L \\ 0 & \text{sonst} \end{cases}$

Von diesen Vektoren gibt es überabzählbar viele $(2^{\mathbb{N}})$ wie schon bewiesen. □

Sprachen, die nicht rekursiv aufzählbar / semi-entscheidbar sind:

$$L \notin RE$$

Literatur: Hromkovič 5.2., Sipser 4.2.

Wiederholung: entscheidbar, semi-entscheidbar

Eine Sprache L heißt rekursiv aufzählbar / semi-entscheidbar falls es eine TM M gibt, die L akzeptiert. Das heißt,

$w \in L \Rightarrow$ TM akzeptiert w

$w \notin L \Rightarrow$ TM verwirft oder hält nicht an

Eine Sprache L heißt (rekursiv) entscheidbar, falls es eine TM M gibt, so daß gilt:

$w \in L \Rightarrow$ TM akzeptiert w

$w \notin L \Rightarrow$ TM verwirft w

Notation: R und RE

In Folgenden setzen wir ein festes, endliches Alphabet voraus, üblicherweise $\{0,1\}$.
Nun betrachten wir Sprachen über diesem Alphabet.

RE = Menge der rekursiv aufzählbaren / semi-entscheidbaren Sprachen

recursively enumerable

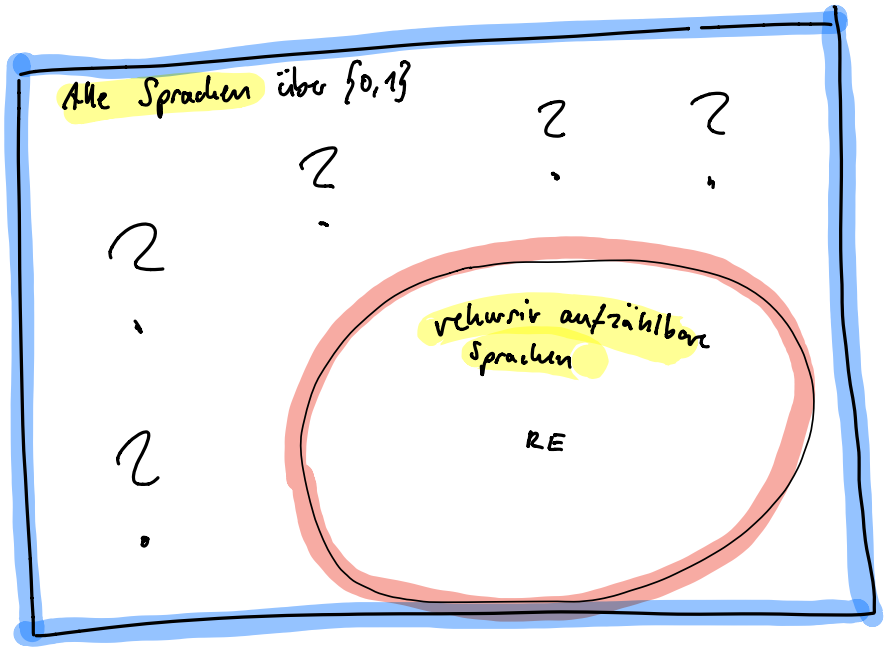
R = Menge der (rekursiv) entscheidbaren Sprachen

recursively decidable



(**REG** = Menge der regulären Sprachen)

Gibt es Sprachen außerhalb von RE?



Ja: Es gibt Sprachen, die nicht rekursiv aufzählbar sind

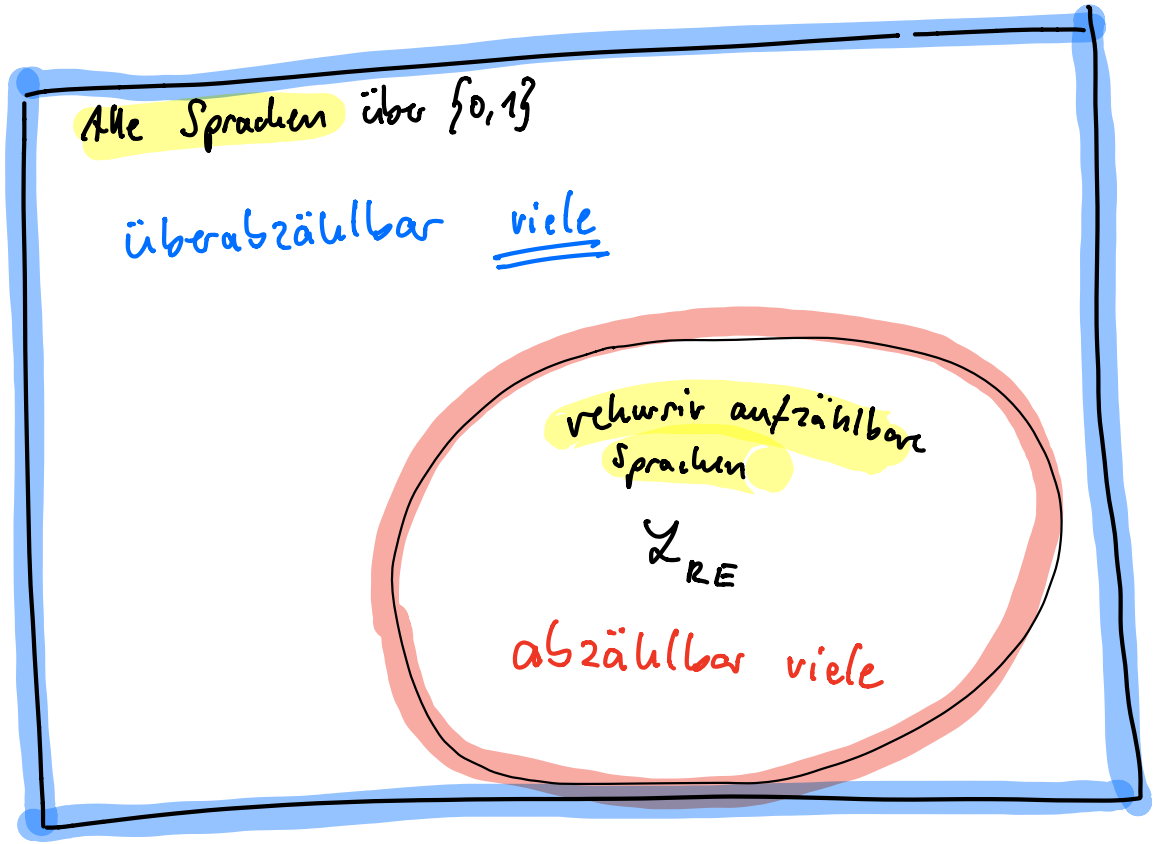
Satz Betrachte das Alphabet $\Sigma = \{0,1\}$. Es gibt Sprachen L über Σ, die nicht rekursiv aufzählbar sind.

Beweis

- Die Menge aller Turing-Maschinen (Gödel-Nummern) ist abzählbar unendlich.
- Also ist die Menge der Sprachen, die von einer TM erkannt werden, auch höchstens abzählbar.
- Die Menge aller Sprachen über $\{0,1\}$ ist aber überabzählbar.
- Also muß es (viele!) Sprachen geben, die nicht von einer TM erkannt werden!

Abzählbarkeits-argument

□


Alle Sprachen über {0,1}
überabzählbar viele
rekursiv aufzählbare Sprachen
$\mathcal{L}_{RE}$
abzählbar viele

Erste Intuition

Die "meisten" Sprachen sind nicht rekursiv aufzählbar und erst recht nicht rekursiv entscheidbar.



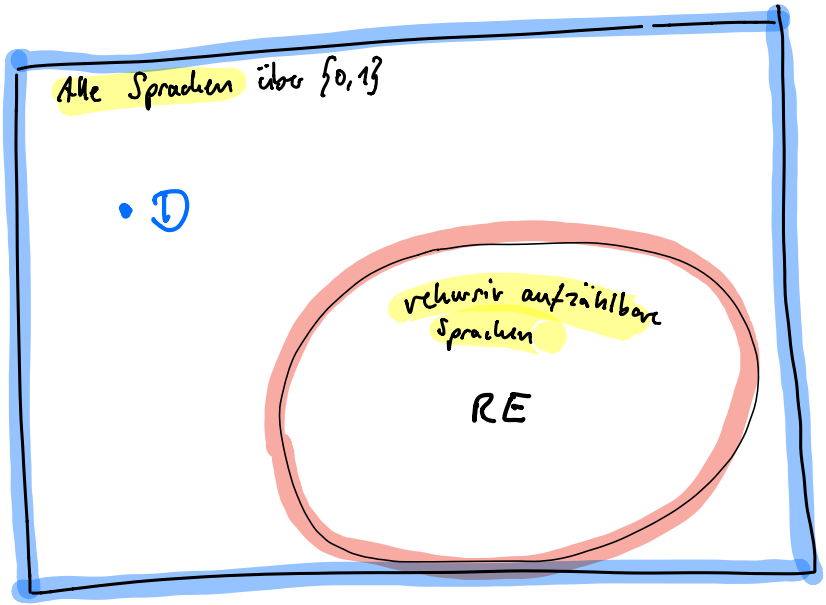
Zweite Intuition

Aber ... das ist jetzt bisschen unbefriedigend, weil es ein reiner Existenzbeweis ist.

Können wir eine Sprache $L \in RE$ konstruieren?

Starten wir mal einen ersten Versuch.

Explizit Sprache $\mathbb{D} \setminus RE$?



Bisher: "nur" Existenzbeweis, etwas unbefriedigend.

Können wir explizit eine Sprache $\mathbb{D} \setminus RE$ angeben?

Diagonalsprache

Wollen nun explizit eine Sprache L konstruieren mit $L \notin RE$.

Die Idee ist folgende:

- Seien $w_1, w_2, w_3, \ldots$ alle Wörter über dem Alphabet $\Sigma = \{0,1\}^*$
  (wir wissen, daß es nur abzählbar viele gibt, also können wir sie numerieren)
- Sei $M_1, M_2, M_3, \ldots$ die Folge aller Turingmaschinen
  (wir wissen, daß es nur abzählbar viele gibt).
- Definiere $d_{ij} = \begin{cases} 1 & \text{falls } M_i \text{ das Wort } w_j \text{ akzeptiert} \\ 0 & \text{sonst} \end{cases}$

Diagonalsprache

Nun betrachte die folgende Tabelle:

	w_1	w_2	w_3	w_4	
M_1	d_{11}	d_{12}	d_{13}	d_{14}	...
M_2	d_{21}	d_{22}	d_{23}	d_{24}	
M_3	d_{31}	d_{32}	d_{33}	d_{34}	
	$\vdots$				

In der i-ten Zeile steht der Indikatorvektor der von M_i akzeptierten Sprache.

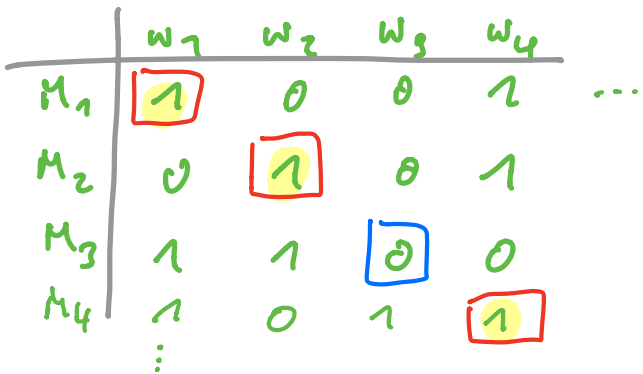
Nun definiere die Diagonalsprache D

$$D = \{ w_i \mid d_{ii} = 0 \}$$

Diagonalsprache

Nun betrachte die folgende Tabelle:

Beispiel:



Diagonalsprache

- enthält das Wort w_3
- enthält nicht die Wörter w_1, w_2, w_4

Die Sprache wird nicht von M_1 erkannt (erstes Wort falsch)
M_2 (zweites Wort falsch)
⋮

Also gibt es keine TM, die D erkennt.

Diagonalsprache $\notin$ RE

Haben also gesehen:

Satz: $D \notin RE$, d.h. die Diagonalsprache ist nicht rekursiv aufzählbar.

Beweis war wieder ein Diagonalisierungstrick.

Ein Schritt zurück:

können wir die Sprache D nicht einfach "hinschreiben"?

Sollte doch irgendwie ganz einfach gehen:

- $M_1, M_2, M_3, \ldots$ sind die Turingmaschinen in der natürlichen Ordnung
- $w_1, w_2, w_3, \ldots$ sind die binären Strings in der natürlichen Ord
- Müssen ausprobieren, was das Ergebnis von $\langle M_i, w_i \rangle$ ist.

  Falls $\langle M_i, w_i \rangle = 1$, ist $w_i \notin D$.

  $= 0$ $w_i \in D$.

- Können also für alle Wörter rausfinden, ob sie in D sind und D auf diese Weise sogar entscheiden. ?!?

Wo ist der Haken?

Ein Schritt zurück:

Können wir nicht doch eine TM bauen, die D entscheidet?

- Eingabe w
- Finde raus, welche Nummer w in der Liste aller Wörter hat:
  Für welches i_0 gilt $w = w_{i_0}$?

  Dafür gibt es eine einfache TM.
- Nun konstruiere die Twinnummer M_{i_0} (auch dafür gibt es eine TM).
- Nun wende $\langle M_{i_0}, w_{i_0} \rangle$ an und drehe die Ausgabe um.

Sollte doch klappen, oder?!?

Auflösung des Rätsels

Was passiert nun, wenn wir dieser TM das Wort eingeben, das dem Index ihrer eigenen Turingnummer entspricht?

- Sei M_D die TM, die wir gerade konstruiert haben.
- Sei i_0 der Index, den diese TM in der Liste aller TMs hat: $M_1, M_2, \ldots, \underbrace{M_{i_0}}_{=M_D}, \ldots$
- Sei w_{i_0} das i_0-te Wort in unserer Liste aller Wörter: $w_1, w_2, \ldots, w_{i_0}, \ldots$
- Was ist nun das Ergebnis von $\langle M_{i_0}, w_{i_0} \rangle$?

???

Auflösung des Rätsels

Egal was wir als $\langle M_{i_0}, w_{i_0} \rangle$ wählen, es führt zu einem Widerspruch:

- M_{i_0} akzeptiert w_{i_0} $\Rightarrow$ D sollte w_{i_0} nicht akzeptieren

  Aber $D = M_{i_0}$. ↯

- M_{i_0} akzeptiert w_{i_0} nicht $\Rightarrow$ D sollte w_{i_0} akzeptieren.

  Aber $D = M_{i_0}$. ↯

↯ ↯ ↯

Ein Schritt zurück: was heißt das jetzt?

Ist die Diagonalsprache irgendwie "besonders schwierig" (was man irgendwie vermuten würde, wenn man eine nicht berechenbare Funktion hat)?

Nein, eigentlich nicht...

Der "Raum aller möglichen TM's" ist schlichtweg zu klein.

Ein Schritt zurück: was heißt das jetzt?

Ein bisschen unbefriedigend ist es immer noch: es fühlt sich immer noch nach "Existenzbeweis" an.

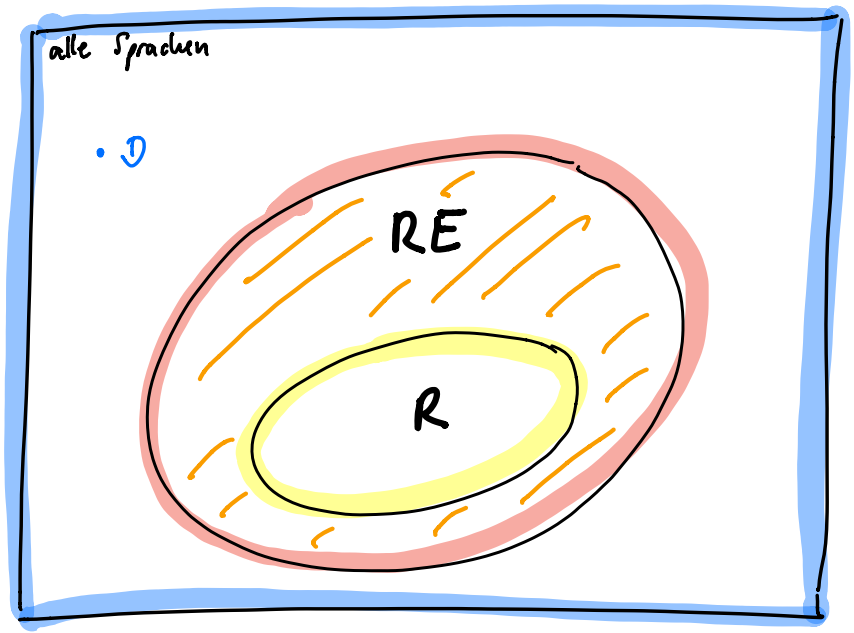
Und als Mensch könnten wir die Sprache ja auch nicht hinschreiben.

Ist alles also eine logische Spielerei? Was bedeutet das für die Praxis?

Werden später sehen: es gibt viel induktive Sprachen, die auch nicht aufzählbar sind.

Nächste Vorlesung:

Gibt es Sprachen in $RE \setminus R$?

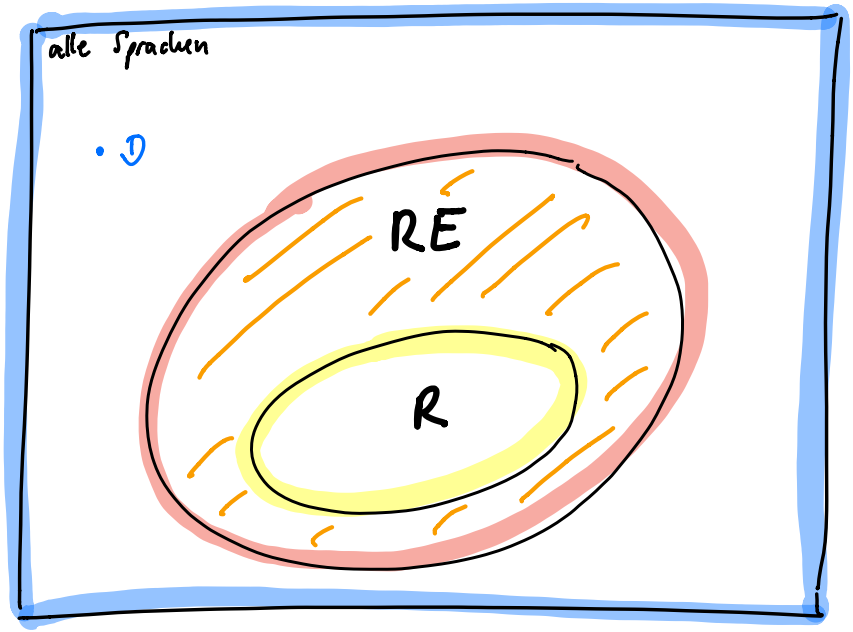


Sprachen, die
rekursiv aufzählbar / semi-entscheidbar
aber nicht entscheidbar sind:

$$L \in RE \setminus R$$

Literatur: Hromkovič 5.2., Sipser 4.2.

Gibt es Sprachen in RE \ R ?



Brauchen zuerst noch eine hilfreiche Zutat.

Sprache A_{TM}

Definiere folgende Sprache:

$$A_{TM} = \{ \langle M, w \rangle \mid M \text{ ist Code einer TM und Wort } w \text{ wird von } M \text{ akzeptiert} \} \subseteq \{0,1\}^*$$

$\langle M, w \rangle$: Wort $\in \{0,1\}^*$

Dabei steht $\langle M, w \rangle$ für das Wort, das die Gödelnummer der TM M mit Wort w konkateniert.

A_{TM} ist semi-entscheidbar

Satz: $A_{TM} \in RE$.

Bew: Um das zu zeigen, müssen wir eine TM bauen, die A_{TM} aufzählt, d.h. für $x \in A_{TM} \Rightarrow$ M akzeptiert

$x \notin A_{TM} \Rightarrow$ M verwirft oder stoppt nicht

Hierzu können wir einfach die universelle TM benutzen. Wir geben dieser TM U das Wort $\langle M, x \rangle$ als Eingabe. Wir haben U genau so konstruiert, daß U akzeptiert falls M gültiger Code und M w akzeptiert.

□

A_{TM} ist nicht entscheidbar

Satz: $A_{TM} \notin R$

Intuition: Wie würde eine TM H aussehen, die A_{TM} entscheidet?

A_{TM} akzeptiert $\iff$ H akzeptiert

A_{TM} akzeptiert nicht $\iff$ H verwirft

- A_{TM} verwirft $\Rightarrow$ H verwirft
- A_{TM} hält nicht $\Rightarrow$ H verwirft

Das sieht problematisch aus!

A_{TM} ist nicht entscheidbar

Satz: $A_{TM} \notin R$

Beweis: **Annahme:** A_{TM} ist entscheidbar.

Sei H eine TM, die A_{TM} entscheidet, d.h.

$$H(\langle M, w\rangle) \begin{cases} \text{akzeptiert} & \text{falls } M \text{ w akzeptiert} \\ \text{verwirft} & \text{falls } M \text{ w nicht akzeptiert} \end{cases}$$

	H	$\langle M_1\rangle$	$\langle M_2\rangle$	$\langle M_3\rangle$	$\langle M_4\rangle$	...
TMs	M_1	acc.	rej.	acc.	acc.	
	M_2	acc.	rej.	rej.	rej.	
	M_3	rej.	acc.	rej.	rej.	
	M_4	acc.	acc.	acc.	acc.	
	⋮					

Wörter

Kurz: H hält immer an!

Einträge: $\langle M_i, \langle M_j\rangle\rangle$

A_{TM} ist nicht entscheidbar: Beweis

Nun betrachten wir die Elemente auf der Diagonalen. Hier liefert die TM H ihren eigenen Code $\langle M_i \rangle$ als Eingabe:

H	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$ ...
M_1	acc.	rej.	acc.	acc.
M_2	acc.	rej.	rej.	rej.
M_3	rej.	acc.	rej.	rej.
M_4	acc.	acc.	acc.	acc.
⋮				

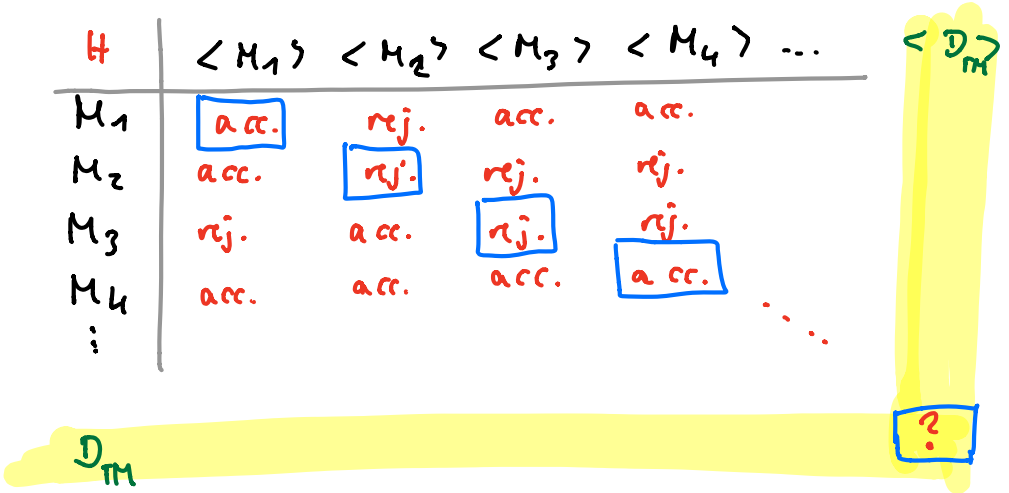
A_{TM} ist nicht entscheidbar: Beweis

Wir bauen folgende TM D_{TM}:

- D_{TM} bekommt den Code $\langle M \rangle$ einer TM als Eingabe
- D_{TM} schaut, was H auf der Eingabe $\langle M, \langle M \rangle \rangle$ machen würde (was steht auf der Diagonalen?)
- D_{TM} gibt das Gegenteil von H zurück:
  Wenn H akzeptiert, dann verwirft D_{TM}
  Wenn H verwirft, dann akzeptiert D_{TM}

A_{TM} ist nicht entscheidbar: Beweis

Mit dem üblichen Diagonalargument entsteht nun wieder ein Widerspruch, nämlich wenn wir betrachten, was das Ergebnis ist, wenn wir nun D_{TM} auf sich selbst anwenden:



Welche Ausgabe produziert D_{TM} angewandt auf $\langle D_{TM} \rangle$?

A_{TM} ist nicht entscheidbar: Beweis

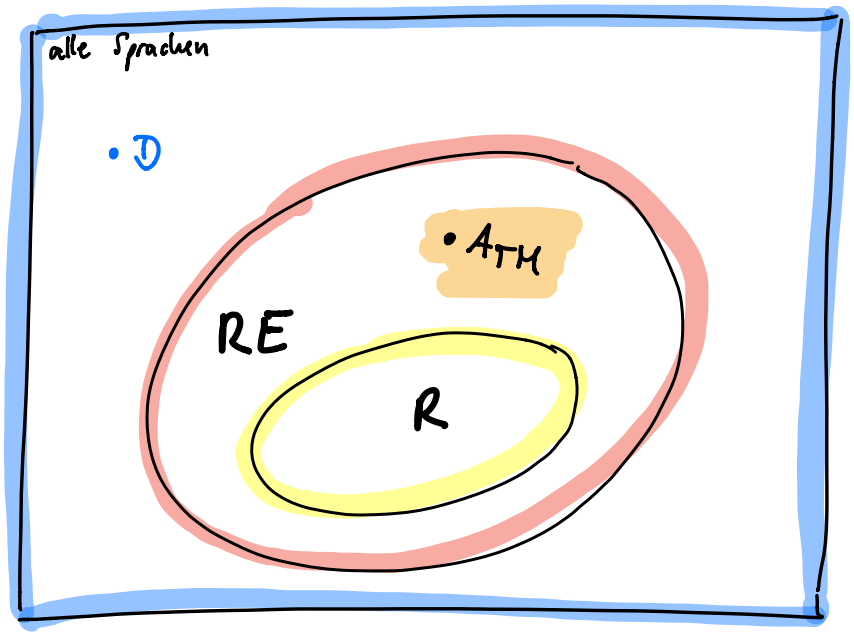
- Falls $D_{TM}(\langle D_{TM} \rangle) = \text{accept}$, dann müsste in der Tabelle von H ein "accept" stehen, aber dann müsste nach Def. von D_{TM} dort verworfen ↯
- Falls $D_{TM}(\langle D_{TM} \rangle) = \text{reject}$: analog ↯

Also kann es weder H noch D_{TM} geben, und A_{TM} kann nicht entscheidbar sein.

↯

□

Wir haben also nun eine Sprache in $RE \setminus R$ gefunden!



Eine weitere interessante Sprache durch Komplementbildung

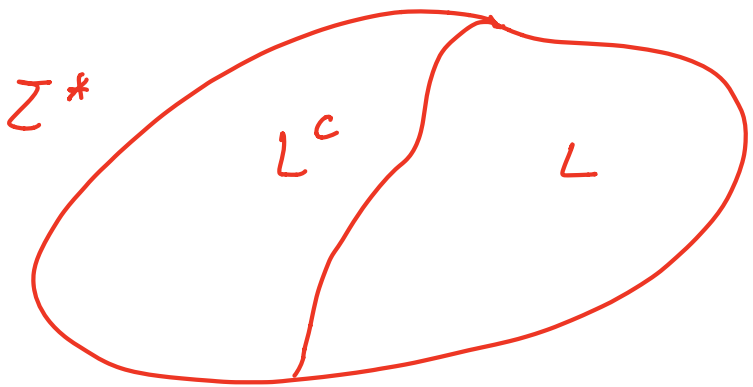
Literatur: Sipser 4.2

Komplement einer Sprache

Sei $L \in \mathcal{L}$ eine Sprache über Σ. Die Komplement-Sprache L^c ist definiert als

$$L^c := \{ w \in \Sigma^* \mid w \notin L \}$$

Manchmal auch Notation $\overline{L}$ statt L^c.



Komplement entscheidbarer Sprachen ist entscheidbar

Satz $L \in R \Rightarrow L^c \in R$

Beweis Sei M eine TM, die L entscheidet. Insbesondere stoppt sie auf allen Eingaben.

Definiere eine TM M_c, bei der der akzeptierende und verwerfende Zustand von M vertauscht wird.

Diese TM entscheidet L^c denn:

$w \in L \Rightarrow$ M akzeptiert w $\Rightarrow$ M_c verwirft w $\Rightarrow w \notin L(M_c)$

$w \notin L \Rightarrow$ M verwirft w $\Rightarrow$ M_c akzeptiert w $\Rightarrow w \in L(M_c)$

Notation: coRE

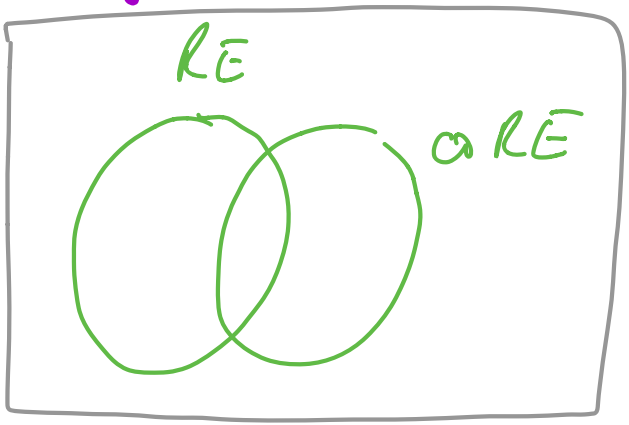
Die Menge coRE ist definiert als

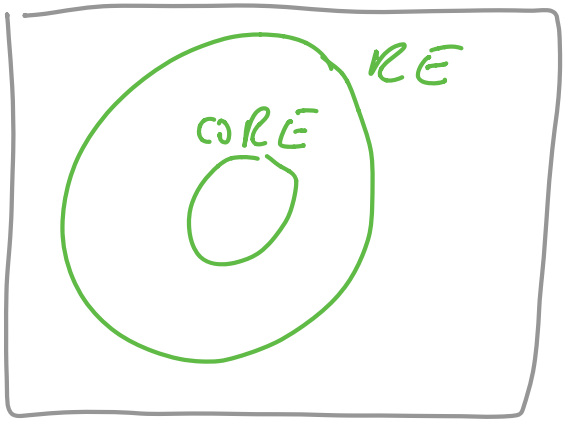
$$\text{coRE} = \{ L \mid L^c \in \text{RE} \}$$

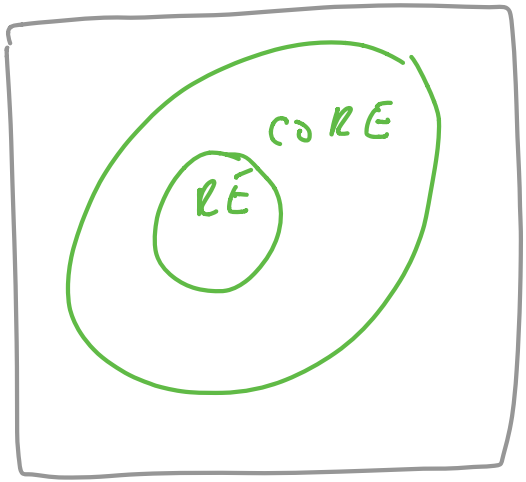
Oder anders aufgeschrieben:

$$L \in \text{coRE} :\Leftrightarrow L^c \in \text{RE}$$

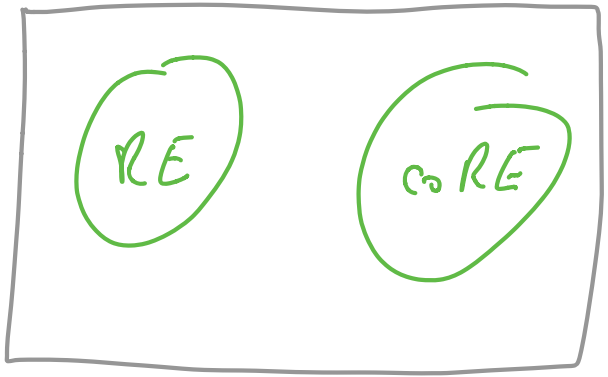
Frage: Wie verhalten sich RE und coRE zueinander?







?

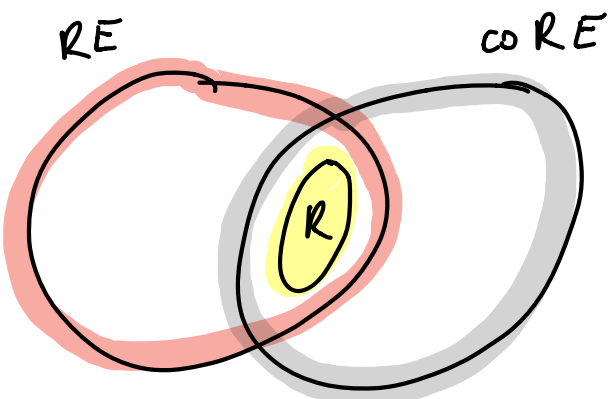


Entscheidbare Sprachen sind in RE und coRE

Beh $L \in R \Rightarrow L \in RE \cap coRE$

Beweis
- $L \in R \Rightarrow L \in RE$ ist eh klar da $R \subset RE$.
- Haben schon gesehen: $L \in R \Rightarrow L^C \in R \subset RE$.

□



Gibt es Sprachen in $(RE \cap coRE) \setminus R$?

$R = RE \cap coRE$

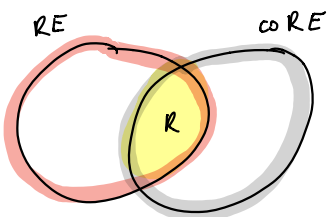
Satz $L \in R \iff L \in RE$ und $L \in coRE$

Beweis "$\Rightarrow$" schon gesehen

"$\Leftarrow$" Sei M eine TM für L und
M_c eine TM für L^c.

Definiere eine neue TM $\tilde{M}$ wie folgt:

- Lasse M und M_c parallel auf Eingabe w laufen
- Falls M akzeptiert, dann soll $\tilde{M}$ akzeptieren
  Falls M_c akzeptiert, dann soll $\tilde{M}$ verwerfen.



Dann: $w \in L \Rightarrow M$ akzeptiert $\Rightarrow \tilde{M}$ akzeptiert

$w \notin L \Rightarrow w \in L^c \Rightarrow M_c$ akzeptiert $\Rightarrow \tilde{M}$ verwirft.

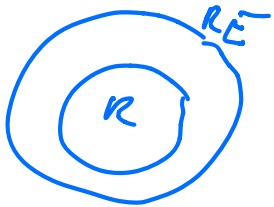
□

Komplement semi-entscheidbarer Sprachen ist nicht notwendig semi-entscheidbar

$$L \in RE \not\Rightarrow L^c \in RE$$

Ein direkter Beweis dieser Aussage wäre schwierig

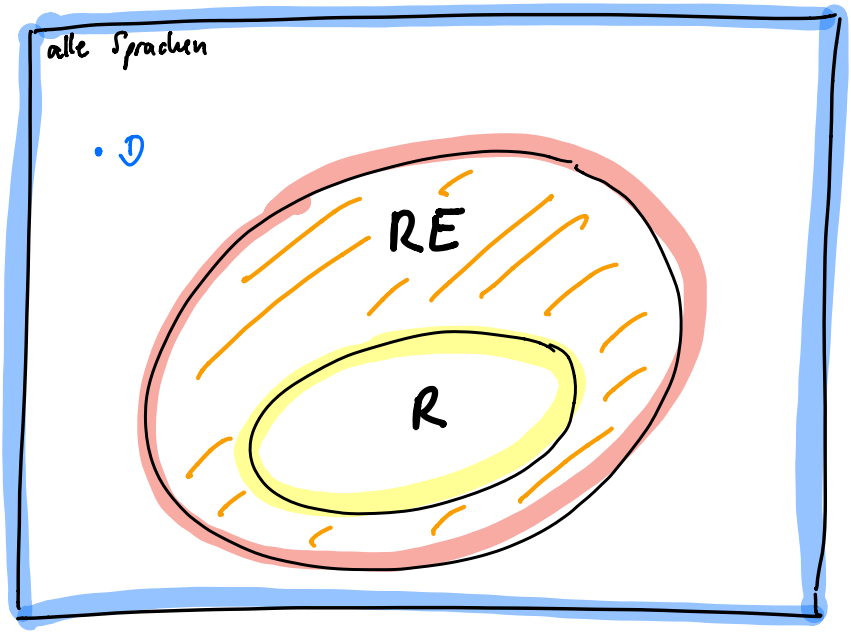
(Warum? Können nicht einfach die TM von L "invertieren" und sagen, daß das dann nicht immer L^c erkennt; es könnte ja eine andere TM geben ...)



Aber $L \in RE$ und $L^c \in RE \Rightarrow L \in R$.

Haben aber gesehen, daß es Sprachen in $RE \setminus R$ gibt. Für diese muß dann $L^c \notin RE$ gelten, also $L \notin coRE$. ∎

Zurück zu unserer Frage: gibt es weitere Sprachen $\notin$ RE?



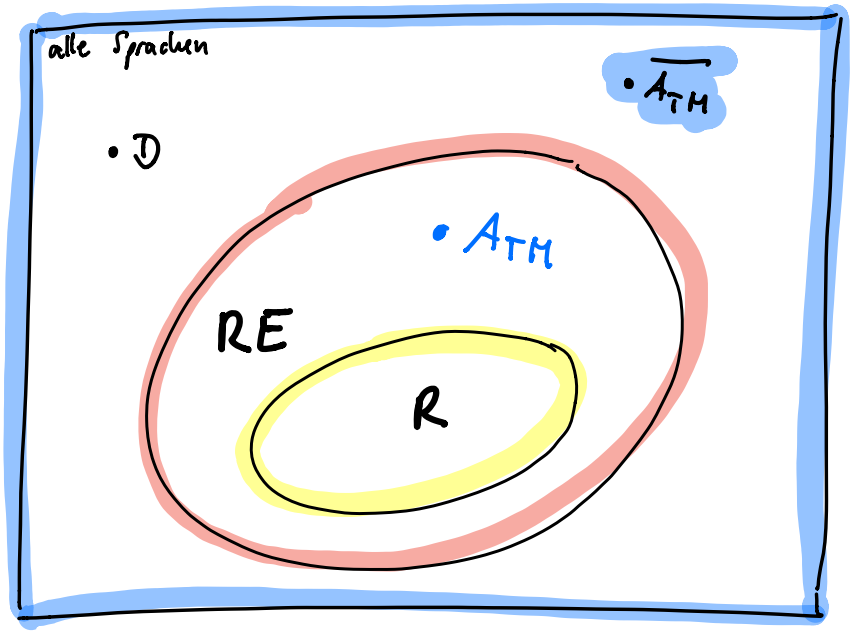
$\overline{A_{TM}}$ ist nicht semi-entscheidbar

Satz $\overline{A_{TM}} \notin RE$

Beweis Haben schon gesehen: $L \in RE$ und $L \in coRE \Rightarrow L \in R$.

Wissen daß $A_{TM} \in RE$ und $A_{TM} \notin R$. Also muß

$A_{TM} \notin coRE$ sein, d.h. $\overline{A_{TM}} \notin RE$.



$\overline{A_{TM}} = \{ \langle M, w \rangle \mid M$ akzeptiert w nicht $\}$

verwirft oder stoppt nicht

Kommt Ihnen $\overline{A_{TM}}$ bekannt vor?

$\overline{A_{TM}}$ ist sehr ähnlich zur Diagonalsprache

die wir auch schon in Vorlesung 17

gesehen haben. Nur der Beweis, daß diese Sprache

nicht in RE ist, funktioniert nun über

Komplementbildung. Und es gibt subtile Unterschiede

in den Definitionen:

Kommt Ihnen $\overline{A_{TM}}$ bekannt vor?

Hier nochmal alle Definitionen, mit ihren subtilen Unterschieden:

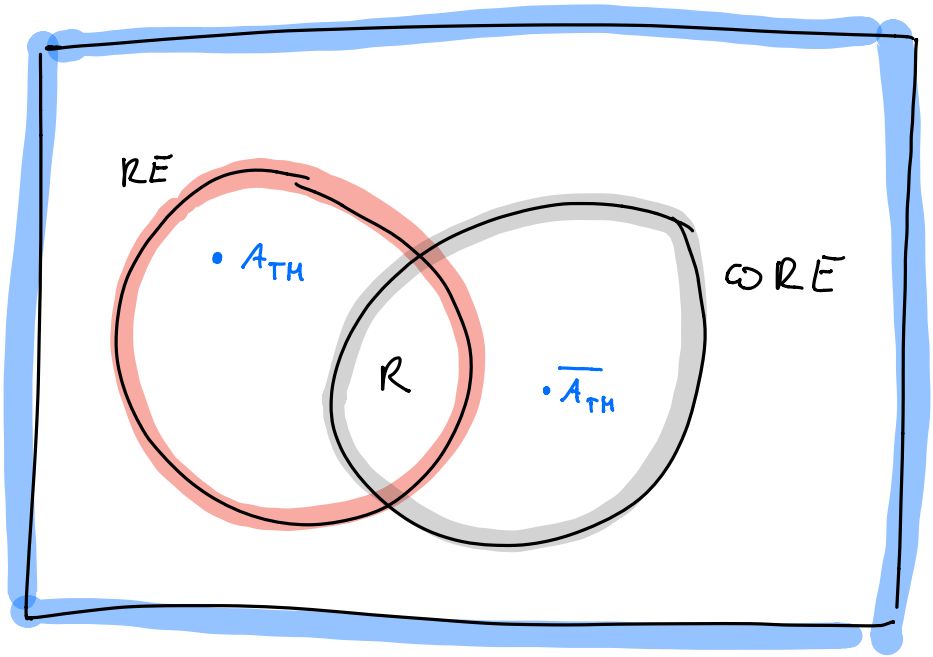
$A_{TM} = \{ \langle M, w \rangle \mid M \text{ akzeptiert } w \}$ ⇝ Vorlesung 19

$\overline{A_{TM}} = \{ \langle M, w \rangle \mid M \text{ akzeptiert } w \text{ nicht} \}$ ⇝ Vorlesung 19

$D = \{ w_i \mid M_i \text{ akzeptiert } w_i \text{ nicht} \}$ ⇝ Vorlesung 17

$D_{TM} = \{ \langle M \rangle \mid M \text{ akzeptiert } \langle M \rangle \text{ nicht} \}$ ⇝ Vorlesung 18

$\overline{D_{TM}} = \{ \langle M \rangle \mid M \text{ akzeptiert } \langle M \rangle \}$ ⇝ Vorlesung 21



Beachte: $\overline{A_{TM}} \in \text{co}RE$, denn $\overline{(\overline{A_{TM}})} = A_{TM} \in RE$.

Abbildungs-Reduktionen in der Berechenbarkeitstheorie

Literatur: Sipser 5.3 (Hromkovič 5.3)

Ziel von Reduktionen

- Wollen ein Problem P_1 lösen, indem wir es auf ein anderes Problem P_2 "reduzieren":

$$P_1 \leq P_2$$

- Das soll so geschehen, daß Aussagen über Berechenbarkeit übertragen werden können:
  - Falls P_2 "leicht" ist, erhalten wir "leichte Lösung" für P_1
  - Falls P_1 "schwer" ist, dann ist vermutlich auch P_2 "schwer"



Viele Reduktionsbegriffe

Es gibt viele Reduktionsbegriffe, die nicht immer das gleiche bedeuten!!!

Auch wir werden im Laufe der Vorlesungen mehrere davon sehen.

Definition: Abbildungs-Reduktion

Def Sprache $L_1 \subset \Sigma_1^*$ ist auf Sprache $L_2 \subset \Sigma_2^*$ **Abbildungs-reduzierbar**, falls gilt:

Es gibt eine Turing-berechenbare Funktion $f: \Sigma_1^* \to \Sigma_2^*$ so daß für alle $w \in \Sigma_1^*$ gilt:

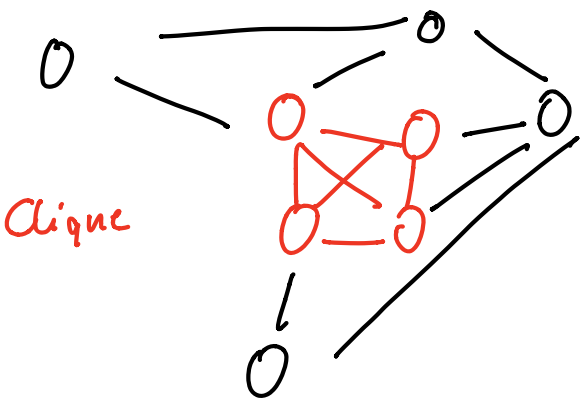
$$w \in L_1 \iff f(w) \in L_2$$

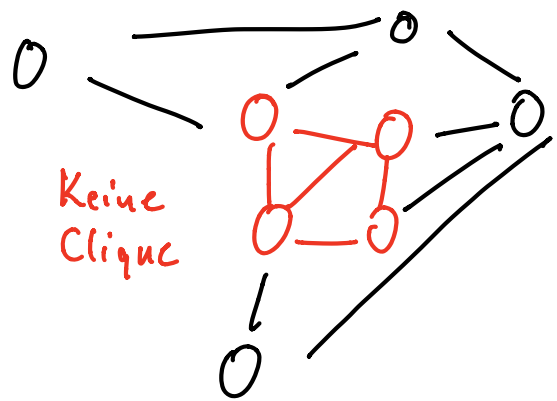
Die Funktion f heißt "**Reduktion**" von L_1 auf L_2

Schreibweise: $L_1 \leq_m L_2$. Das "m" kommt von "mapping reduction", wir lassen es auf den folgenden Folien der Einfachheit halber oft weg.

Beispiel: Clique $\leq$ Vertex cover

Problem k-Clique: Gegeben ein ungerichteter graph $G = (V, E)$ und ein $k \in \mathbb{N}$. Eine Teilmenge $V' \subseteq V$ heißt Clique, falls V' im Graph vollständig verbunden ist.



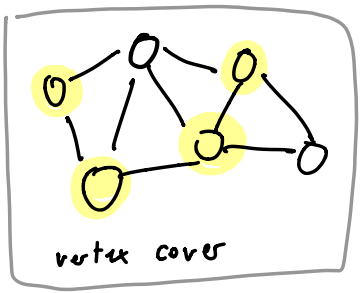


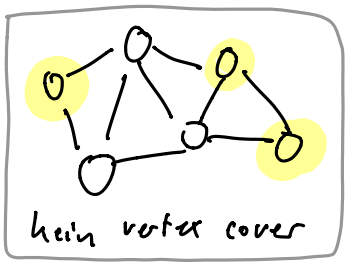
k-Clique Entscheidungsproblem: Gibt es im Graphen eine Clique der Größe k?

Beispiel: Clique $\leq$ Vertex cover

Problem k-Vertex-Cover:

Gegeben ein ungerichteter Graph $G = (V, E)$, $k \in \mathbb{N}$.
Eine Teilmenge $V' \subseteq V$ heißt "Vertex Cover", falls jede Kante des Graphen mindestens einen Endpunkt in V' hat:





k-Vertex cover Entscheidungsproblem: Gibt es ein vertex cover mit k Knoten?

Beispiel: Clique $\leq$ Vertex cover

Reduktion von Clique auf Vertex Cover:

und $k \in \mathbb{N}$ gegeben.

Sei G der gegebene Graph. (Für den wir rausfinden wollen, ob er eine k-Clique hat). Wir wollen nun einen (neuen?) Graphen $\tilde{G}$ bauen und $\tilde{k} \in \mathbb{N}$ wählen, so daß gilt:

G hat k-Clique $\Leftrightarrow$ $\tilde{G}$ hat $\tilde{k}$-Vertex-Cover.

Dazu wählen wir $\tilde{G}$ als das "Komplement" von G:

- $\tilde{G}$ hat die gleichen Knoten wie G
- $\tilde{G}$ hat Kante uv genau dann, wenn G diese Kante nicht hat.

Und wir setzen $\tilde{k} := |V| - k$.

Beispiel: Clique $\leq$ Vertex cover

Behauptung 1: Die Reduktion ist berechenbar.

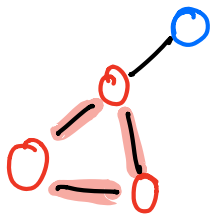
Beweis: Die Reduktion f transformiert den Graphen G in den Graphen $\tilde{G}$. Wir können offensichtlich eine TM konstruieren, die bei Eingabe von G und k die Ausgabe $\tilde{G}$ produziert (Details lassen wir weg). und $\tilde{k}$

□

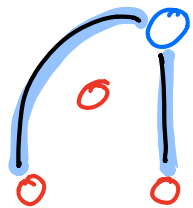
Beispiel: Clique $\leq$ Vertex cover

Behauptung 2: G hat eine Clique der Größe k $\Leftrightarrow$ $\tilde{G}$ hat einen Vertex Cover der Größe $n-k$ (wobei $n = |V|$ ist).

$n-k$: //



Clique in G



Vertex cover in $\tilde{G}$

Correct!: the graph had one edge too much

Beispiel: Clique $\leq$ Vertex cover

Beweis der Behauptung 2 $=: C$

"$\Rightarrow$" Sei $\{v_1, \dots, v_k\} \subset V$ die Knoten der Clique. Alle Kanten $\{v_i, v_j\}$ sind in G, also sind sie nicht in $\tilde{G}$. Wir wählen nun für Graph $\tilde{G}$ $\rightsquigarrow$ $V_{cover} := V \setminus \{v_1, \dots, v_k\}$. Alle Kanten in $\tilde{G}$ müssen mindestens ein Ende in V_{cover} haben, also ist diese Menge ein Vertex cover der Größe $n-k$.

Genauer: $\{u,v\} \in \tilde{E}$. Falls u oder $v \in V_{cover}$, dann ist $\{u,v\}$ von V_{cover} "abgedeckt". Falls u und v beide $\notin V_{cover}$, dann waren $u, v \in C$, also kann $\{u,v\}$ nicht in $\tilde{E}$ sein. □

"$\Leftarrow$" analog.

Behauptung 1 und 2 gemeinsam zeigen also, daß es sich um eine Abbildungs-Reduktion handelt.

Beispiel: Clique $\leq$ Vertex cover

Was bedeutet das jetzt?

- Wollen eine TM bauen, die k-Clique löst.
- Wir starten mit einer Instanz (G, k) des Problemes k-Clique.
- Wir "transformieren" diese Instanz in eine neue Instanz $(\tilde{G}, n-k)$ des Problems k-Vertex-Cover.
  Diese Transformation f kann durch eine TM berechnet werden.
- Nun können wir eine TM des k-Vertex-Cover-Problems auf $f(G,k) = (\tilde{G}, n-k)$ anwenden.
- Deren Ja/Nein-Antworten stimmen dann auch für das ursprüngliche k-Clique-Problem.

Jetzt: der Grund, warum wir das machen

Satz Falls $L_1 \leq L_2$, dann gilt:

(1) Falls L_2 (semi-) entscheidbar ist, dann ist auch L_1 (semi-) entscheidbar.

(2) Falls L_1 nicht (semi-) entscheidbar ist, dann ist auch L_2 nicht (semi-) entscheidbar.

⚠ Beachten Sie die "Richtungen"!

Beweis

Beweis (1): Sei L_2 (semi) entscheidbar, d.h. es gibt eine TM M_2, die die Sprache (semi) entscheidet.

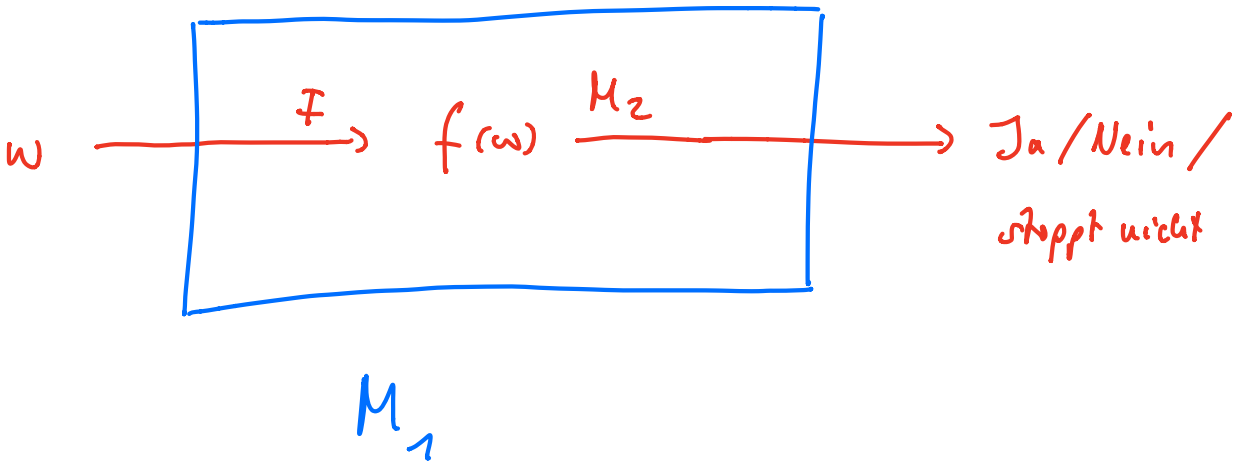
Nun bauen wir eine TM M_1, die L_1 (semi) entscheidet:

Bei Eingabe von w wendet M_1 zunächst die TM F an, die die Reduktion von L_1 auf L_2 realisiert. Auf die Ausgabe von F wenden wir nun M_2 an.

Nach Definition der Reduktion liefert M_2 dann genau das richtige Ergebnis.

Beweis

Als Bild:



Beweis

Beweis (2) Durch Widerspruch:

- Annahme: L_2 ist (semi) entscheidbar,
- Wir starten mit Instanz von L_1 und bauen die TM M_1 von der letzten Folie (geht wegen der Existenz der Reduktion).
- Dann (semi) entscheidet M_1 aber auch L_1. ↯

Also kann L_2 nicht (semi) entscheidbar sein.

□



Die anderen Richtungen gelten nicht

Sei $L_1 \leq L_2$ und L_2 nicht entscheidbar.

Was folgt dann für L_1?

Nichts!

- Wir wissen dann nur, daß wir Problem 1 nicht dadurch lösen können, daß wir es auf Problem 2 reduzieren. (Die "Reduktion" macht das Problem vielleicht nur komplizierter.)
- Aber wir wissen nicht, ob es einen anderen Weg gibt, mit dem wir Problem 1 vielleicht doch lösen könnten!

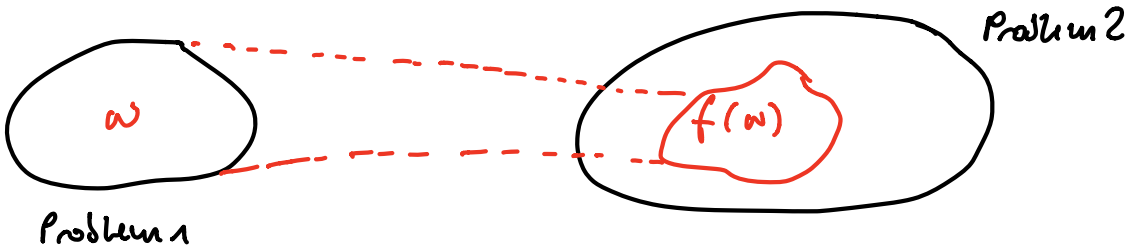


Die anderen Richtungen gelten nicht

Sei $L_1 \leq L_2$ und L_1 entscheidbar.

Was folgt dann für L_2? **Nichts!**

- Um die Reduktion zum Lösen von Problem 2 zu benutzen, müßten wir ja die Umkehrabbildung bauen (um vielleicht eine Reduktion von Problem 2 auf Problem 1 zu erhalten).
- Aber es kann Instanzen von Problem 2 geben, die nie durch die Reduktion "getroffen" werden:



Anwendung von Reduktionen:

Das Halteproblem und

viele weitere Probleme in $RE \setminus R$

Literatur: Sipser 5.3 (Hromkovič 5.3)

Ziel: Weitere Probleme in $RE \setminus R$

Wir wollen nun eine ganze Kette von Reduktionen zeigen (Definitionen kommen gleich):

$$\overline{D_{TM}} \leq A_{TM} \leq H \leq H_0 \leq K$$

Argumentationskette wird sein:

1. $\overline{D_{TM}}$ ist unentscheidbar, d.h. $\overline{D_{TM}} \notin R$:

$$\overline{D_{TM}} \leq A_{TM} \leq H \leq H_0 \leq K$$

Folgerung: alle Probleme "weiter hinten" in der Kette sind auch unentscheidbar:

$$\overline{D_{TM}} \leq A_{TM} \leq H \leq H_0 \leq K$$

2. K ist semi-entscheidbar, d.h. in RE.

$$\overline{D_{TM}} \leq A_{TM} \leq H \leq H_0 \leq K$$

Dann: Probleme "weiter vorne" in der Kette sind semi-entscheidbar:

$$\overline{D_{TM}} \leq A_{TM} \leq H \leq H_0 \leq K$$

Wiederholung: Verschiedene Sprachen mit subtilen Unterschieden

Hier nochmal alle Definitionen, mit ihren subtilen Unterschieden:

$A_{TM} = \{ \langle M, w \rangle \mid M \text{ akzeptiert } w \}$ ⇝ Vorlesung 19

$\overline{A_{TM}} = \{ \langle M, w \rangle \mid M \text{ akzeptiert } w \text{ nicht} \}$ ⇝ Vorlesung 19

$D = \{ w_i \mid M_i \text{ akzeptiert } w_i \text{ nicht} \}$ ⇝ Vorlesung 17

$D_{TM} = \{ \langle M \rangle \mid M \text{ akzeptiert } \langle M \rangle \text{ nicht} \}$ ⇝ Vorlesung 18

$\overline{D_{TM}} = \{ \langle M \rangle \mid M \text{ akzeptiert } \langle M \rangle \}$ ⇝ Vorlesung 21

Reduktion $\overline{D_{TM}} \leq A_{TM}$

Gödelnummer von M

$\overline{D_{TM}} := \{ \langle M \rangle \mid M \text{ akzeptiert } \langle M \rangle \}$

$A_{TM} := \{ \langle M, w \rangle \mid M \text{ akzeptiert } w \}$

Beh $\overline{D_{TM}} \leq A_{TM}$

Bew Reduktion ist trivial: definiere $f(w) := ww$

Ist berechenbar und offensichtlich gilt:

$w \in \overline{D_{TM}} \iff M$ akzeptiert $\langle M \rangle$

$\iff f(w) = ww \in A_{TM}$

□

Reduktion $A_{TM} \leq H$ (Halteproblem!)

$A_{TM} = \{ \langle M, w \rangle \mid M \text{ akzeptiert } w \}$

$H = \{ \langle M, w \rangle \mid M \text{ hält bei Eingabe } w \text{ an} \}$

$=:$ Halteproblem

Beh $A_{TM} \leq H$

Bew Gegeben die Eingabe $\langle M, w \rangle$. Wir bauen eine neue TM $\tilde{M}$ wie folgt:

$\tilde{M}$ macht die gleichen Berechnungen wie M.

Wenn falls M in einem nicht-akzeptierenden Zustand stoppt, dann begibt sich $\tilde{M}$ in eine Endlosschleife und stoppt nicht. Sei $\langle \tilde{M} \rangle$ der Code dieser TM. →

Den Code von $\tilde{M}$ können wir berechnen.

Nun definieren wir die Reduktionsabbildung wie folgt:

$f(\langle M, w \rangle) = \langle \tilde{M}, w \rangle$. Dann gilt:

$\langle M, w \rangle \in A_{TM}$ $\Leftrightarrow$ M akzeptiert w (def. von A_{TM})

$\Leftrightarrow$ $\tilde{M}$ stoppt bei Eingabe w

$\Leftrightarrow$ $\langle \tilde{M}, w \rangle \in H$ (def der Sprache H)

klar nach Konstruktion von $\tilde{M}$

Reduktion $H \leq H_0$

$H = \{ \langle M, w \rangle \mid M$ hält bei Eingabe w an $\}$ allgemeines Halteproblem

$H_0 = \{ \langle M \rangle \mid M$ hält bei Eingabe des leeren Wortes ε an $\}$ spezielles

Beh $H \leq H_0$

Bew Sei die Eingabe $\langle M, w \rangle$ vom Problem H gegeben.
Wir bauen eine TM $\tilde{M}$ wie folgt: Falls Eingabe von $\tilde{M}$ das Wort ε ist, dann simuliert $\tilde{M}$ die TM M auf w.
Falls die Eingabe von $\tilde{M}$ nicht ε ist, hält $\tilde{M}$ einfach an.

Sei $\langle \tilde{M} \rangle$ der Code dieser TM. ⚠ $\tilde{M}$ hängt von M und w ab, beides ist "fest einprogrammiert".

Wir definieren $f(\langle M, w\rangle) := \langle \tilde{M} \rangle$. Diese Abb. ist berechenbar und es gilt:

$\langle M, w\rangle \in H$ (def. der Sprache H)

$\Leftrightarrow$ M hält auf w an

$\Leftrightarrow$ $\tilde{M}$ hält bei Eingabe des leeren Wortes an.

$\Leftrightarrow$ $f(\langle M, w\rangle) = \langle \tilde{M} \rangle \in H_0$ (def der Sprache H_0)

"$\Rightarrow$" Denn: $\tilde{M}$ simuliert dann, was M auf w tut, und M hält ja an.

"$\Leftarrow$" Wenn $\tilde{M}$ die Eingabe ε kriegt, simuliert sie M auf w. Diese Simulation hält genau dann an, wenn M auf w anhält.

odot

Reduktion $H_0 \leq K$

$H_0 = \{ \langle M \rangle \mid M$ hält bei Eingabe des leeren Wortes ε an $\}$

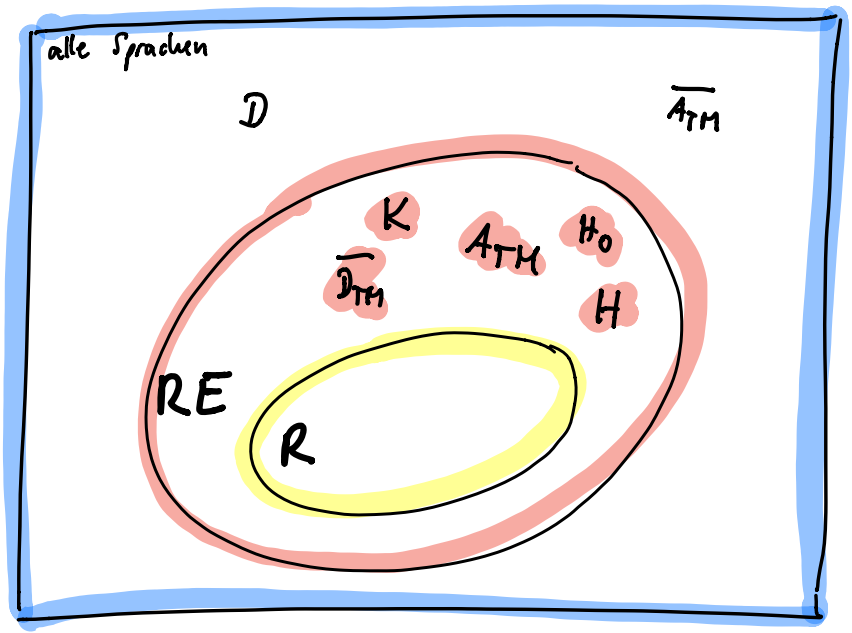
$K = \{ \langle M \rangle \mid M$ hält bei Eingabe von $\langle M \rangle$ an $\}$

Beh $H_0 \leq K$

Bew Ähnlich wie die anderen Beweise, Übung.

Erste Schlußfolgerung

Satz Die Sprachen $\overline{D_{TM}}$, A_{TM}, H, H_0, K sind alle in $RE \setminus R$



Nochmal zur Verdeutlichung: wie wendet man Reduktionen an?

Ziel 1: Wollen beweisen, daß eine Sprache L (semi)entscheidbar ist.

Vorgehen: Müssen eine Sprache $\tilde{L}$ finden mit $L \leq \tilde{L}$, so daß $\tilde{L}$ (semi) entscheidbar ist.

Ziel 2: Wollen beweisen, daß eine Sprache L nicht (semi) entscheidbar ist.

Vorgehen: Müssen eine Sprache $\tilde{L}$ finden mit $\tilde{L} \leq L$, so daß $\tilde{L}$ nicht (semi)-entscheidbar ist.

Interpretation des Halteproblems

Das Halteproblem ist ein Problem, das für die Informatik eigentlich total wichtig wäre: gegeben der Code eines Programms, können wir verifizieren, dass es immer beendet wird und nicht in einer Endlosschleife hängen bleibt?

Nein, das Halteproblem ist nicht entscheidbar!!!

(Und offensichtlich ist es rekursiv aufzählbar: wir probieren das Programm einfach aus und warten, bis es anhält. Kann lange dauern, aber wird stattfinden.)

Interpretation des Halteproblems

Warum ist das relevant? Verifikation!!!

Man würde doch gerne beweisen können, daß ein Programm, das ein Atomkraftwerk oder ein Flugzeug steuert, sich nicht aufhängt!!!

Haben wir hiermit gesehen: es gibt sehr relevante Probleme, die von einer Turingmaschine beweisbar nicht gelöst werden können.

Variante des Halteproblems

$T = \{ \langle M \rangle \mid M$ hält auf jeder Eingabe an $\}$

Man kann durch Reduktionen zeigen, daß $T \notin RE$ gilt.

Das gleiche gilt für $\overline{T}$.

Beweis: Übung

Bem: Das ist also ein Beispiel für eine Sprache, wo sowohl die Sprache selbst als auch ihr Komplement $\notin RE$ sind.

Berechnen zwei TMs das gleiche?

Seien M_i und M_j die i-te und j-te Turingmaschine.

Können wir testen, ob sie vielleicht das gleiche berechnen?

Also auf jeder Eingabe die gleiche Ausgabe erreichen (evtl. auf unterschiedlichem Weg)?

Beispiel: M_i codiert den Heapsort-Algorithmus, M_j den Mergesort-Algorithmus. Beide liefern die gleichen Ausgaben (wenn auch auf unterschiedlichem Weg).

Berechnen zwei TMs das gleiche?

Anwendung: M_i Spezifikation eines Programms,
M_j die aktuelle Implementierung.
Erfüllt die Implementierung die Spezifikation?

Definieren Sprache

$$V = \{ \langle M_1, M_2 \rangle \mid \forall w: M_1(w) = M_2(w) \}$$

Man kann mit Reduktionen zeigen, daß $V \notin RE$ gilt.
Und es gilt auch $\overline{V} \notin RE$.

Ausblick: Turing-Reduktionen

Wir haben nur bisher nur Abbildungs-Reduktionen angesehen.

Ein allgemeinerer Begriff ist die Turing-Reduktion:

Problem 1 $\leq_T$ Problem 2 falls es eine TM zur Lösung von Problem 1 gibt, die eine Turingmaschine für Problem 2 beliebig oft als "Unterprogramm" aufrufen darf.

Turing-Reduktionen sind "mächtiger" als Abbildungs-Reduktionen.

~> Sipser 6.3

Ausblick: Reduktionen in der Komplexitätstheorie

Bisher: "leichte" und "schwere" Probleme im Kontext Berechenbarkeit:

leicht = (semi) entscheidbar

schwer = nicht (semi) entscheidbar.

Müssen daher sicherstellen, daß die Reduktion f selber "leicht" (= berechenbar) ist, sonst bricht das Argument zusammen, mit dem wir dann Schlussfolgerungen ziehen wollen.

Ausblick: Reduktionen in der Komplexitätstheorie

Wir werden später in der Komplexitätstheorie noch andere Reduktionen sehen. Dort:

"leicht" = schnell berechenbar

"schwer" = nicht schnell berechenbar.

Hier brauchen wir dann, daß die Reduktion f selbst leicht d.h. "schnell berechenbar" ist.

Der Satz von Rice

Literatur: Hromkovič ; Barak Kap. 9.5

Motivation

Haben gesehen: einige "kritische" Eigenschaften von Turingmaschinen / Programmen sind nicht entscheidbar.

Frage: Sind diese Eigenschaften eher die Ausnahme oder eher die Regel?

Antwort: Satz von Rice

So gut wie alle interessanten Eigenschaften von TMs sind unentscheidbar!

Wie definieren wir "interessante Eigenschaften"?

Wir interessieren uns nur für die Ausgabe / das Ergebnis einer TM. Wie genau sie das Ergebnis berechnet hat, ist uns zunächst egal.

Wir wollen also Eigenschaften von TMs ausdrücken, die sich nur an der von ihr erkannten Sprache festmachen lassen (und nicht an der konkreten Implementierung).

Wiederholung: Äquivalenz von THS

Zwei THS H_1, H_2 heißen äquivalent, falls gilt:

$$L(H_1) = L(H_2)$$

Nicht-triviale Sprache

Def Sei L eine Sprache, deren Wörter alle Codes von TMs sind: $L \subseteq \{ w \in \{0,1\}^* \mid w \text{ ist Code einer TM} \}$.

Die Sprache L heißt **nicht-trivial**, falls gilt:

$L \neq \emptyset$ und

Es gibt TMs, deren Code in L enthalten ist.

$L \neq \{ w \in \{0,1\}^* \mid w \text{ ist Code einer TM} \}$.

Es gibt TMs, deren Code nicht in L enthalten ist.

Semantisches Entscheidungsproblem

Def Sei L eine Sprache, deren Wörter alle Codes von TMs sind: $L \subseteq \{ w \in \{0,1\}^* \mid w \text{ ist Code einer TM} \}$.

Die Sprache L heißt semantisch, falls gilt:

Wenn M_1 und M_2 äquivalente TMs sind, dann sind entweder beide in L enthalten oder beide nicht in L enthalten:

$$\langle M_1 \rangle \in L \iff \langle M_2 \rangle \in L$$

... es hängt nicht an der Implementierung, sondern nur an der Ausgabe der TM ...

Satz von Rice

Satz: Jedes semantische, nichttriviale Entscheidungsproblem ist unentscheidbar.



Beweisidee

Sei L eine nichttriviale, semantische Sprache.

Prinzipielle Herangehensweise: wollen das spezielle Halteproblem H_0 auf L reduzieren, es ist aber ein bisschen komplizierter.

Sei M_{leer} eine TM, die kein einziges Wort akzeptiert:

$$L(M_{leer}) = \emptyset \quad .$$

Können Sie so eine TM hinschreiben?

Wie genau wir reduzieren, hängt nun davon ab, ob $\langle M_{leer} \rangle \in L$ gilt oder nicht:

Beweisidee

Vorgehensweise wird sein:

- Falls $\langle M_{leer} \rangle \in L$, dann konstruieren wir eine Reduktion $\overline{H_0} \leq_m L$. ≙ Fall 1

- Falls $\langle M_{leer} \rangle \notin L$, dann konstruieren wir eine Reduktion $H_0 \leq_m L$. ≙ Fall 2

Da sowohl H_0 als auch $\overline{H_0}$ unentscheidbar sind, muß dann auch L unentscheidbar sein.

Erinnerung H_0

H_0 = spezielles Halteproblem

$$H_0(\langle M \rangle) = \begin{cases} 1 & \text{falls TM } M \text{ hält bei Eingabe } \varepsilon \text{ an} \\ 0 & \text{sonst} \end{cases}$$

Beweis Satz von Rice

Gegeben: L nicht trivial, semantisch

Da L nicht-trivial ist, gibt es

eine TM M_0 mit $\langle M_0 \rangle \notin L$, und

eine TM M_1 mit $\langle M_1 \rangle \in L$.

Beweis Satz von Rice, Fall 1

Wir betrachten den Fall 1 : $\langle M_{leer} \rangle \in L$

Wollen eine Reduktion $\overline{H_0} \leq L$ konstruieren

Im Folgenden sei also M_L eine TM, die L entscheidet.

Rest der Reduktion:

- Eingabe $\langle N \rangle$, wollen entscheiden, ob N auf ε stoppt.
- Transformieren $\langle N \rangle$ in einen anderen String $\langle M \rangle$, den wir dann in M_L füttern. Es soll gelten:

$$N \in \overline{H_0} \iff M \in L$$

Beweis Satz von Rice, Fall 1

Turingmaschine R zur Lösung von $\overline{H_0}$ auf Eingabe $\langle N \rangle$:

// Eingabe: $\langle N \rangle$ Code einer TM

// Gewünschte Ausgabe: ob N auf ε stoppt

1. Konstruiere den Gödel-Code einer TM M, die folgende Berechnung durchführt:

   Turingmaschine M auf Eingabe z:

   - Simuliere $\langle N, \varepsilon \rangle$
   - Return $H_0(z)$

   ⚠ die blaue TM wird hier nicht ausgeführt!

   Reduktion: Berechnung der neuen Instanz $f(N) := M$

2. Return $H_L(\langle M \rangle)$

   Aufruf der TM H_L mit der transformierten Instanz M als Eingabe

Beweis Satz von Rice, Fall 1

Turingmaschine R zur Lösung von $\overline{H_0}$ auf Eingabe $\langle N \rangle$:

// Eingabe: $\langle N \rangle$ Code einer TM

// Gewünschte Ausgabe: ob N auf ε stoppt

1. Konstruiere den Gödel-Code einer TM M, die folgende Berechnung durchführt:

   Turingmaschine M auf Eingabe z:
   - Simuliere $\langle N, \varepsilon \rangle$ — N hart einprogrammiert
   - Return $M_0(z)$ — feste TM mit $\langle M_0 \rangle \notin L$

2. Return $M_L(\langle M \rangle)$

   M_L: TM der Sprache L

   $\langle M \rangle$: Code der blauen TM

Beweis Satz von Rice, Fall 1

- Sei $N \notin \overline{H_0}$, d.h. $N \in H_0$, d.h. N hält bei Eingabe ε an.
- In diesem Fall ist die blaue TM M äquivalent zur TM H_0 (denn die beiden liefern dann genau das gleiche Ergebnis).
- Da L semantisch ist, gilt daher $M \in L \Leftrightarrow H_0 \in L$.
- Wir wissen aber, daß $H_0 \notin L$ gilt. Daher gibt die TM R in Zeile 2 $H_L(M) = H_L(H_0) =$ 0 zurück: $R(N) = 0$.

Beweis Satz von Rice, Fall 1

- Sei $N \in \overline{H_0}$, d.h. $N \notin H_0$, d.h. N hält bei Eingabe ε nicht an.
- In diesem Fall ist die blaue TM M äquivalent zur TM M_{leer}. (denn die beiden liefern dann genau das gleiche Ergebnis).
- Da L semantisch ist, gilt daher $M \in L \Leftrightarrow M_{leer} \in L$.
- Wir wissen aber, daß $M_{leer} \in L$ gilt. Daher gibt die TM R in Zeile 2 $M_L(M) = M_L(M_{leer}) = 1$ zurück: $R(N) = 1$.

Beweis Satz von Rice, Fall 1

Haben also gesehen:

- $N \notin \overline{H_0} \Rightarrow f(N) := M \notin L$
- $N \in \overline{H_0} \Rightarrow f(N) := M \in L$

} Reduktion tut was sie soll

- Die Reduktion ist offensichtlich berechenbar (beweisen wir nicht formal)

Beweis Satz von Rice, Fall 2

Fall 2 geht nun ganz analog, wir müssen nur zwei Stellen anpassen:

- Wir reduzieren $H_0 \leq L$
- Statt der Sprache H_0 verwenden wir in der blauen TM die Sprache H_1.

Das Ganze sieht dann so aus:

Beweis Satz von Rice, Fall 2

Turingmaschine R zur Lösung von H_0 auf Eingabe $\langle N \rangle$:

// Eingabe: $\langle N \rangle$ Code einer TM

// Gewünschte Ausgabe: ob N auf ε stoppt

1. Konstruiere den Gödel-Code einer TM M_1, die folgende Berechnung durchführt:

   Turingmaschine M auf Eingabe z:

   - Simuliere $\langle N, \varepsilon \rangle$
   - Return $M_1(z)$

2. Return $M_L(\langle M \rangle)$

Beweis Satz von Rice, Fall 2

Dann: $N \in H_0$, d.h. $\langle N, \varepsilon \rangle$ stoppt $\Rightarrow$

$\Rightarrow$ blaue TM äquivalent zu H_1

$\Rightarrow H_L(H) = H_L(H_1) = 1$ nach Wahl von H_1

$N \notin H_0$, d.h. $\langle N, \varepsilon \rangle$ stoppt nicht

$\Rightarrow$ blaue TM äquivalent zu H_{leer}, und im Fall 2 gilt $H_{leer} \notin L$

$\Rightarrow H_L(H) = H_L(H_{leer}) = 0$



Betrachten wir nochmal den Beweis

- Wo haben wir benutzt, daß L nicht-trivial ist?
- Wo haben wir benutzt, daß L semantisch ist?

Achtung: Umkehrung ist i.A. falsch

Theorem von Rice sagt intuitiv:

semantisch $\Rightarrow$ unentscheidbar

Die Umkehrung gilt aber nicht: es gibt unentscheidbare Sprachen, die nicht semantisch sind.

Was bedeutet das jetzt alles?

Haben gesehen: allgemeine "Verifikation" von Programmen ist unmöglich! Können wir uns dann überhaupt noch erlauben, sicherheitskritische Funktionen von Computern ausführen zu lassen?

Autopilot bei Flugzeugen?

Steuerung eines Atomkraftwerks?

Was bedeutet das jetzt alles?

Nochmal: was genau haben wir im Theorem von Rice bewiesen?

Wenn wir keinerlei Einschränkungen dahingehend vornehmen, welche Sprachen und TMs wir anschauen, dann sind wichtige Eigenschaften von TMs / Programmen / Algorithmen nicht berechenbar.

Was bedeutet der jetzt alles?

Betrachte folgendes Programm:

```
HelloWorld ()
    print "Hello World"
end
```

Für dieses Programm sollten wir doch beweisen können, daß es anhält?.

Was bedeutet das jetzt alles?

Es gibt Hoffnung: vielleicht müssen wir die "Sorte von Programmen" einschränken, und können dann wieder Aussagen machen. Zum Beispiel:

> Programme, die keine Schleifen und Variablen benutzen, halten an.

Müssen also Einschränkungen vornehmen: Programm darf nur bestimmte Dinge tun, bestimmte Stellen des Speichers beschreiben, erfüllt Invarianzen ...

$\rightsquigarrow$ Software-Verifikation

Was bedeutet der jetzt alles?

Beachte: Warum gelt der Beweis des Satzes von Rice nicht, wenn wir nicht alle Programme / TMs zulassen?

Im Moment wissen wir nicht, was die blaue TM im Code für Eigenschaften hat, und ob der gleiche Beweis funktionieren würde, wenn wir diese TM einschränken würden.

Weitere unentscheidbare Probleme

Intuition zu Aufzählbarkeit / Entscheidbarkeit

Wann ist ein Problem / eine Sprache rekursiv aufzählbar? Entscheidbar?

Intuition: RE aber nicht R

$H = \{ \langle M, w \rangle \mid M \text{ hält auf } w \}$

Frage: Ist $\langle M, w \rangle \in H$?

Versuch einer Antwort mit folgendem Algorithmus:

- Simuliere M auf w. Wenn $\langle M, w \rangle \in H$, dann hält die Simulation in einem akzeptierenden Zustand. Gut ☺
  ⇝ rekursiv aufzählbar (und so kann man es beweisen)
- Wenn $\langle M, w \rangle \notin H$, hält die Simulation nicht an, wir können aber nicht verwerfen. Schlecht ☹
  ⇝ vermutlich nicht entscheidbar
  (Hier: Bauchgefühl! Es könnte einen besseren Algorithmus geben! Beweis muß anders gehen)

Intuition: RE aber nicht R

Vorgehensweise:

- finde Algorithmus, der aufzählt
- mit diesem Alg. kann auch der Beweis geführt werden.

- versuche viele Algorithmen.
- Wenn alle das nicht knacken, vermute "nicht entscheidbar".
- Beweis geht nicht über einen nicht funktionierenden Algorithmus

Intuition: nicht mal in RE

$T = \{ \langle M \rangle \mid M \text{ hält auf } \underline{\text{jeder}} \text{ Eingabe an} \}$

Frage: Ist $\langle M \rangle \in T$?

Versuch einer Antwort mit folgendem Algorithmus:

- Probiere M der Reihe nach auf allen Wörtern aus.

Problem: dieser Algorithmus wird nie fertig, kommt also nie in einen akzeptierenden Zustand. Schlecht :(

⇝ vermutlich nicht rekursiv aufzählbar

(Hier: Bauchgefühl! Er könnte einen besseren Algorithmus geben! Beweis muß anders gehen)

Intuition: nicht mal in RE

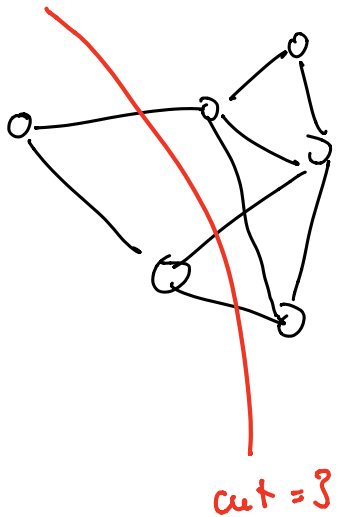
Vorgehensweise:

- versuche viele Algorithmen
- wenn keiner funktioniert, versuch "$\notin$ RE"
- Beweis geht nicht über einen nicht funktionierenden Alg!

Achtung, Bauchgefühl kann trügen!

Mir fällt kein gutes Beispiel aus der Berechenbarkeit ein, aber aus der Komplexitätstheorie:

Cut in einem Graphen: Teile Knotenmenge in zwei Gruppen und zähle die Kanten zwischen den zwei Gruppen



Mincut Problem: leicht
(in $O(n^3)$)

Maxcut Problem: schwer
(NP-vollständig, siehe später)

Aber das Bauchgefühl sagt zunächst bei beiden: schwer...

Nicht berechenbar jenseits von Rice?

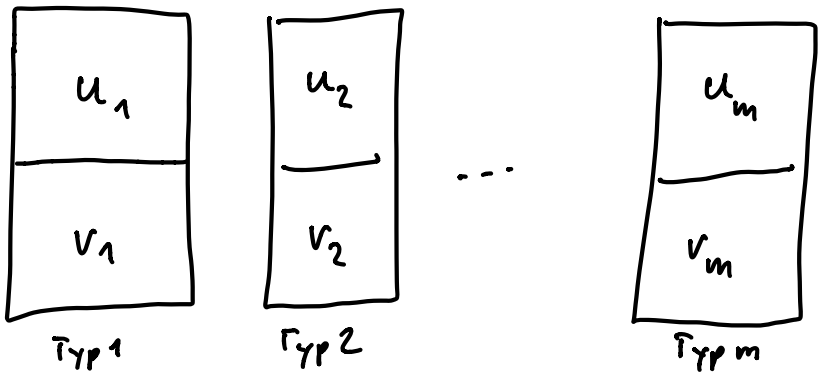
Satz von Rice: Interessante Eigenschaften von Computerprogrammen sind nicht berechenbar.

Aber gibt es auch "andere Probleme", die nichts mit Programmen zu tun haben und auch unentscheidbar sind?

Ja, ganz viele! Wir schauen uns ein paar davon an:

Post'sches Korrespondenzproblem

Gegeben: m verschiedene "Typen" von Dominosteinen.



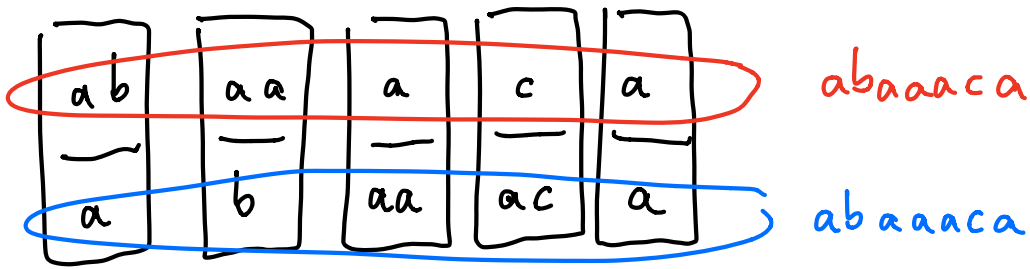
$u_1, \ldots, u_m$ und $v_1, \ldots, v_n$ sind feste Wörter über einem Alphabet Σ.

Von jedem Typ Dominostein haben wir beliebig viele Steine zur Verfügung.

Post'sches Korrespondenzproblem

Frage: gibt es eine endliche Folge von Dominosteinen, so daß der "Wort oben" identisch ist mit dem "Wort unten"?

Beispiel:



Wir dürfen jeden Typ Dominostein beliebig oft verwenden.

Post'sches Korrespondenzproblem

Satz Das Post'sche Korrespondenzproblem ist unentscheidbar!

Beweis: siehe Hromkovič Kap. 5.1.

Warum könnte es schwierig sein?

In diesem Fall gibt es unendlich viele Konstellationen, die wir ausprobieren könnten:

- Wenn eine Anordnung das Problem löst: schön. → vermutlich rekursiv aufzählbar
- Aber was, wenn wir schon lange gesucht haben und keine finden? → vermutlich nicht entscheidbar

Die Frage ist hier: gibt es eine "strukturelle Eigenschaft", anhand derer wir sehen können, daß wir die Suche irgendwann abbrechen können?

Matrix Mortality Problem (Paterson, 1970)

Gegeben: endliche Menge von $n \times n$-Matrizen mit ganzzahligen Einträgen: $M_1, M_2, \ldots, M_k$.

Frage: Gibt es ein Produkt solcher Matrizen, das die Nullmatrix ergibt? (Jede Matrix darf beliebig oft vorkommen).

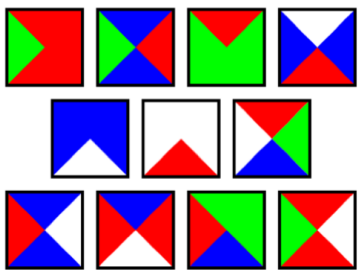
Dieses Problem ist unentscheidbar.

Referenz: M. Paterson: Unsolvability in 3×3-Matrices. 1970

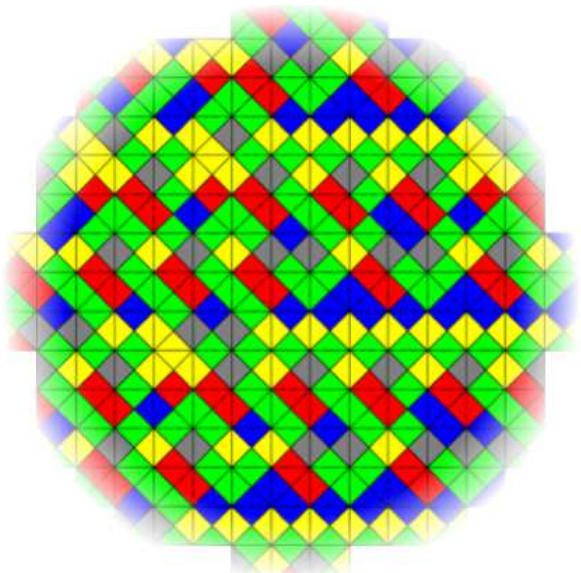
Wang - Fliesen (Hao Wang, 1961; Berger 1966)

Gegeben: Menge von "Wang - Fliesen": siehe unten

Frage: Können wir damit die Ebene "kacheln", so daß immer nur passende Farben aneinander grenzen?



Einzelne Fliesen



Kachelung der Ebene

Abb: Wikipedia

Wang - Fliesen (Hao Wang, 1961; Berger 1966)

Die Frage ist unentscheidbar.

Beweis funktioniert durch Reduktion des Haltproblems:

→ R. Berger: Undecidability of the Domino Problem, 1966

Game of Life (Conway, 1970)

Startkonfiguration beliebig, iteriere die folgenden Regeln:

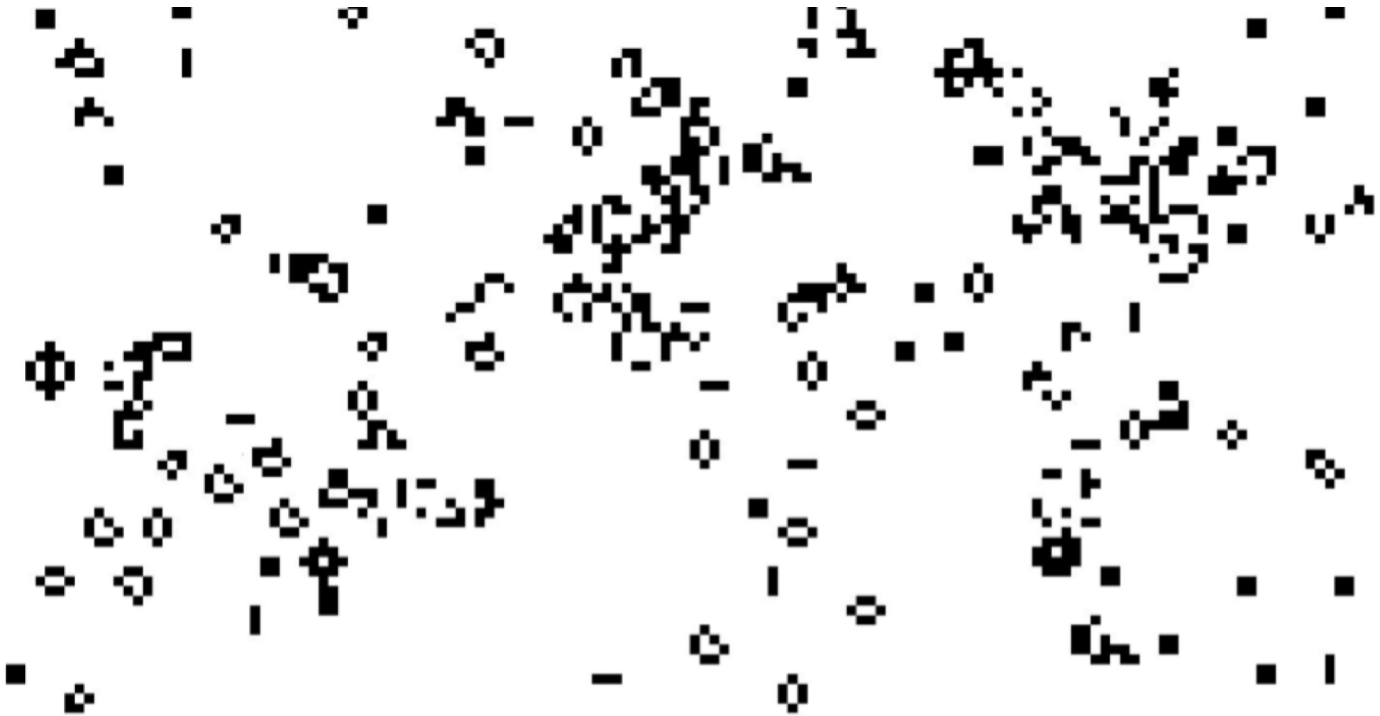
(1) Jede schwarze Zelle mit zwei oder drei schwarzen Nachbarn bleibt schwarz, die anderen schwarzen Zellen werden weiß.

(2) Jede weiße Zelle mit drei schwarzen Nachbarn wird schwarz, die anderen weißen Zellen bleiben weiß.

Frage: läßt sich eine bestimmte Zielkonfiguration aus der Startkonfiguration erreichen?

Bem: Ist ein spezielles Beispiel eines zellulären Automaten, sehen wir vielleicht später noch in der Vorlesung.

Game of Life



Kolmogorov - Komplexität

Ausflug in die Datenkomprimierung:

Betrachten wir die folgenden beiden Strings:

0 1 0 1 0 1 0 1 0 1 0 1 0 1

0 1 1 0 1 0 1 0 0 1 0 1 1 0 1 0 1 0

Den blauen String können wir sehr kompakt beschreiben: sieben mal 01

Den roten String können wir nicht so einfach beschreiben.

Kolmogorov - Komplexität

Gegeben: • feste "Programmiersprache" (z.B. Turing-Maschine)

• ein String s

Kolmogorov-Komplexität von s: Länge des kürzesten Programms (in der gegebenen Sprache), das den String s als Ausgabe produziert.

Kolmogorov - Komplexität

Intuition:

0 1 0 1 0 1 0 1 0 1 0 1 0 1

0 1 1 0 1 0 1 0 0 0 1 0 1 1 0 1 0 1 0

- Der blaue string hat niedrige Komplexität:

  for $i = 1, \ldots, 7$, print "01", end

- Der rote string hat hohe Komplexität, mir fällt nichts besseres ein als

  print " 0 1 1 0 1 0 1 0 0 0 1 0 1 1 0 1 0 1 0 "

Kolmogorov - Komplexität

Vielleicht überraschend:

Die Kolmogorov-Komplexität ist nicht berechenbar!

Wieso? Können wir nicht alle Programme "ausprobieren", ausführen vom kürzesten?

Nein. Vielleicht hält eines davon nicht an, und dann kommen wir nicht weiter.

Kolmogorov-Komplexität

Literatur dazu: Sipser Kap. 6.4

(Hromkovič Kap. 5.6)

Wer es ganz genau wissen will:

Li / Vitanyi: Introduction to Kolmogorov Complexity

Vielleicht am Ende der Vorlesung noch mehr dazu.

Viele mehr ...

Es gibt noch viel mehr unberechenbare Funktionen,

unentscheidbare Probleme.

Vergleich von Modellen der Berechenbarkeit

Literatur: Barak Kap. 8

Motivation: wo sind wir gerade?

1. Teil der Vorlesung: Endliche Automaten

→ zu eingeschränkt

2. Teil der Vorlesung: Turing-Maschinen

→ weitaus mächtiger

→ können aber nicht alles, was wir gerne hätten!

(Halteproblem etc)

Jetzt: Gibt es ein "sinnvolles Berechnungsmodell", das mehr kann als Turingmaschinen?

Andere Berechnungsmodelle

Wir wollen andere "Programmiersprachen" / "Computermodelle" betrachten und mit THs vergleichen.

Intuition: Ein Berechnungsmodell ist etwas, das eine Eingabe annimmt und (manchmal) eine Ausgabe "berechnet".

(Formale Definition kommt gleich)

Turing-Vollständigkeit und -Äquivalenz

Intuition: Ein Berechnungsmodell heißt Turing-vollständig (Turing complete), falls es jede Turing-berechenbare Funktion berechnen kann.

d.h. wir können eine TM auf das andere Modell "reduzieren"

Ein Computermodell heißt Turing-äquivalent, wenn es genau die gleichen Funktionen wie eine TM berechnen kann.

d.h. die Reduktion geht auch in die andere Richtung

Turing-Vollständigkeit und -Äquivalent

Formaler:

Sei $\mathcal{F}$ die Menge aller partiellen Funktionen von $\{0,1\}^*$ nach $\{0,1\}^*$.

Ein **Berechnungsmodell** ist eine Abbildung

$$\mathcal{M}: \{0,1\}^* \to \mathcal{F}$$

$\approx$ Gödelnummer — Funktion, die dadurch "implementiert" wird

Ein **Programm** $P \in \{0,1\}^*$ **$\mathcal{M}$-berechnet** eine Funktion $F \in \mathcal{F}$ falls $\mathcal{M}(P) = F$.

Turing - Vollständigkeit und -Äquivalenz

Ein Berechnungsmodell $\mathcal{M}$ heißt **Turing-Vollständig** wenn es eine Turing-berechenbare Abbildung

$\text{encode}_{\mathcal{M}} : \{0,1\}^* \to \{0,1\}^*$ gibt, so daß

binäre Darstellung der neuen Maschine

dessen berechnete Funktion — $\mathcal{M}(\text{encode}(\langle M \rangle))$

Gödelnr. einer TM

identisch zu der von der TM M berechneten Funktion ist.

Turing-Vollständigkeit und -Äquivalenz

Ein Berechnungsmodell M heißt **Turing-äquivalent**, falls

- es Turing-vollständig ist — "Reduktion $TM \leq M$"
- und es zusätzlich eine M-berechenbare Abb. — "Reduktion $M \leq TM$"

$$\text{decode}_M : \{0,1\}^* \to \{0,1\}^*$$

gibt, so daß für jedes $P \in \{0,1\}^*$

$N = \text{decode}_M(P)$ die Gödelnr. einer TM ist und diese TM die gleiche Funktion wie $M(P)$ berechnet.

Ziel in den nächsten Vorlesungen

Betrachten verschiedene Berechnungsmodelle, die erstaunlicherweise alle Turing-vollständig / äquivalent sind.

WHILE - und FOR - Programme

Literatur: Schöning 2.3.

Motivation

Turing-Maschinen sind ja ganz nett ...

aber gibt es auch Modelle der Berechenbarkeit,

die näher an echten Programmiersprachen sind?

WHILE-Programme: Alphabet

While-Programme sind strings über einem Alphabet, das folgende Dinge enthält:

Variablen: $x_0, x_1, x_2, \ldots$	abzählbar viele
Konstanten: $0, 1, 2, \ldots$	abzählbar viele
Key words: while, do, end	genau diese drei
Symbole: $:=, \neq, ;, +, -, [,]$	genau diese sieben

WHILE-Programme: Syntax

über dem Alphabet der vorigen Folie

While-Programme sind Strings, die induktiv wie folgt definiert sind:

Def (1) Ein "einfacher Befehl" ist eine der folgenden drei Anweisungen:

einfache arithmetische Operationen

$x_i := x_j + x_k$ oder

$x_i := x_j - x_k$ oder

$x_i := c$ ⇝ Wertzuweisung

wobei $i, j, k, c \in \mathbb{N}$.

(2) Ein WHILE-Programm P ist entweder ein einfacher Befehl oder hat die Form

(a) while $x_i \neq 0$ do P_1 end ⇝ eine Schleife

oder (b) $[P_1 ; P_2]$ ⇝ Hintereinanderausführung

für $i \in \mathbb{N}$, P_1, P_2 while-Programme.

WHILE-Programme: Syntax vs Semantik



Intuitiv ist natürlich klar, was hier gemeint sein soll. Aber formal bestehen die Programme erst mal aus Strings, wir haben noch nicht definiert, wie Ein- und Ausgabe "verarbeitet werden"

WHILE-Programme: Semantik

Eingabe: besteht aus natürlichen Zahlen $a_0, \dots, a_{s-1} \in \mathbb{N}$ und wird in den Variablen $x_0, \dots, x_{s-1}$ gespeichert.

Ausgabe: Falls das Programm anhält, dann ist die Ausgabe der Inhalt der Variable x_0 bei Beendigung des Programms.

Variablenbelegung: Jedes Programm darf beliebig viele, aber nur endlich viele Variablen benutzen. Sei $\ell(P)$ die maximale Zahl an benutzten Variablen. Dann können wir die Variablenbelegung zu jedem Zeitpunkt als einen Vektor schreiben: zum Beispiel ist die Startbelegung $(a_0, a_1, \dots a_{s-1}, 0 \dots 0)$

WHILE-Programme: Semantik

Sei S die Startbelegung der Variablen. Wir definieren nun eine partielle Funktion $\phi_P(S)$, die uns sagt, welche Ausgabe ein Programm P bei Eingabe von S produziert. Das ist ein bisschen mühsam, aber anders geht es nicht:

Sei $S = (\sigma_0, \sigma_1, \sigma_2, \dots)$ eine Variablenbelegung

(wir geben nicht an, wie lange sie ist, halt "lang genug")

WHILE-Programme: Semantik

Semantik einfacher Ausdrücke:

die σ sind nat. Zahlen, mit denen wir rechnen können

Falls $P =$ "$x_i := x_j + x_k$", dann gilt

$$\phi_P(s) = (\sigma_0, \sigma_1, \dots, \sigma_{i-1}, \sigma_j + \sigma_k, \sigma_{i+1}, \dots)$$

Falls $P =$ "$x_i := x_j - x_k$", dann gilt

$$\phi_P(s) = (\sigma_0, \sigma_1, \dots, \sigma_{i-1}, \tilde{\sigma}_i, \sigma_{i+1}, \dots) \text{ mit}$$

$$\tilde{\sigma} = \max\{\sigma_j - \sigma_k, 0\}$$

Falls $P =$ "$x_i = c$", dann

$$\phi_P(s) = (\sigma_1, \dots, \sigma_{i-1}, c, \sigma_{i+1}, \dots)$$

WHILE-Programme: Semantik

Semantik der Hintereinander-Ausführung:

Falls $P = \text{“}[P_1 ; P_2]\text{”}$, dann sei

$$\Phi_P(s) = \begin{cases} \Phi_{P_2}(\Phi_{P_1}(s)) & \text{falls definiert} \\ \text{undefiniert} & \text{sonst} \end{cases}$$

Im folgenden bezeichnen wir mit $\Phi_P^{(r)}(s)$ die r-fache Hintereinander-Ausführung von P, gestartet auf s:

$$s \xrightarrow{P} s_1 \xrightarrow{P} s_2 \xrightarrow{P} \cdots \xrightarrow{P} s_r .$$

WHILE-Programme: Semantik

Semantik von Schleifen:

Falls $P =$ "while $x_i \neq 0$ do P_1 end", sei $r \in \mathbb{N}$ die kleinste Zahl so daß entweder

$\Phi_{P_1}^{(r)}(\vec{x})$ nicht terminiert oder so daß die i-te Variable in $\Phi_{P_1}^{(r)}(\vec{x})$ gleich 0 ist. Dann setzen wir

$$\Phi_P(\vec{x}) = \begin{cases} \Phi_{P_1}^{(r)}(\vec{x}) & \text{falls das Programm geendet hat} \\ \text{undefined} & \text{sonst.} \end{cases}$$

WHILE-Programme: Semantik

Lemma: Für jedes while-Programm P ist Φ_P wohldefiniert.

Beweis per struktureller Induktion, lassen wir weg (ist einfach aber langweilig).

Syntactic Sugar

Auch jetzt ist die "Programmiersprache" noch ziemlich eingeschränkt und umständlich. Können wir die "gängigen Konstrukte" aus normalen Programmiersprachen auch in der einfachen while-Sprache ausdrücken?

Die Antwort ist ja, und ich zeige nur wenige Beispiele:

Syntactic Sugar: Arithmetische Operationen

Alle arithmetischen Operationen auf natürlichen Zahlen können durch WHILE-Programme gerechnet werden:

Beispiel Multiplikation: $x_i := x_j \cdot x_k$

```
x_i := 0
count := x_j
while (count ≠ 0) do
    x_i := x_i + x_k
    count := count - 1
end
```

FOR - Schleife

FOR-Schleifen können durch while-Schleifen dargestellt werden:

Übung

(siehe auch nächstes Video zur FOR-Programmiersprache)

Syntactic Sugar: if - Bedingungen

Wollen folgendes Statement:

IF x = 0 THEN P END

Können dies wie folgt erreichen:

// Sei x eine Variable, die schon definiert ist, und y nicht.

```
Y := 1
FOR x DO y := 0 END
FOR y DO P END
```

- Überlegen Sie, warum das wirklich klappt...
- Können Sie ein if - then - else - statement bauen?

Übung...

Syntactic Sugar : Stacks und Arrays

Übung

Syntactic Sugar

usw

Man kann zeigen, daß man mit while-Programmen alle gängigen Konstrukte von Standard-Programmiersprachen (Java, C++, ...) nachbauen kann.

WHILE vs. Turing ?

While-Programme sind so "mächtig" wie normale Programmiersprachen. Wie verhalten sie sich im Vergleich zu Turing-Maschinen?

WHILE vs. FOR vs. GOTO

Literatur: Schöning 2.3. (Hofmann 6.1)

FOR - Programme

Alphabet: Wie bei while-Programmen (nur statt "while" nehmen wir "for")

Def Ein **FOR-Programm** ist entweder ein einfacher Befehl (wie im WHILE-Programm) oder hat die Form

(a) $\quad$ for x_i do P_1 end

oder (b) $\quad [P_1 ; P_2]$

für ein $i \in \mathbb{N}$ und For-Programme P_1, P_2.

FOR-Programme: Semantik

Die Semantik sollte intuitiv klar sein:

for x_i do P end

soll bedeuten:

falls $x_i = 0$, wird nichts getan

falls $x_i > 0$, dann wird P entsprechend oft ausgeführt.

Formal muß man alles definieren, analog zu while-Programmen, lassen wir hier weg.

FOR-Programme: Semantik

Ergänzung

for x_i do P end

Wichtig: Das ist so zu verstehen, daß der Wert x_i einmal am Anfang der FOR-Schleife ausgelesen wird, und so oft wird die Schleife dann ausgeführt.

Insbesondere hat es keinen Einfluß mehr auf die Anzahl der FOR-Durchläufe, wenn x_i innerhalb der FOR-Schleife verändert wird.

FOR-Programme terminieren immer

Ergänzung

Behauptung: FOR-Programme wie eben definiert terminieren immer.

Beweis Strukturelle Induktion, Übung!



In vielen "echten" Programmiersprachen kann man FOR-Schleifen hinschreiben, die nicht terminieren. Dort wird dann die Zählvariable innerhalb der Schleife verändert. Das wurde für die hier dargestellte FOR-Programmiersprache ausgeschlossen.

Beispiel:

```
for (i = 1, i < 10, i++)
    print ("Test")
    i = i - 1
end
```

FOR-Programme terminieren immer

Ergänzung

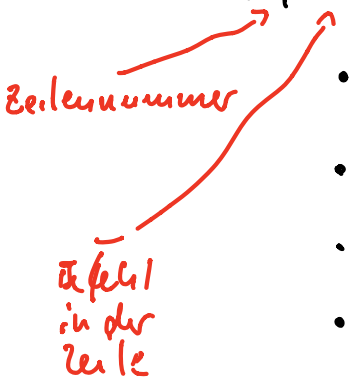
Behauptung: FOR-Programme wie eben definiert terminieren immer.

Beweis Strukturelle Induktion, Übung!

GOTO - Programme

Alphabet: wie bei while-Programmen, aber zusätzlich sind alle Zeilen numeriert

Def Ein Goto-Programm besteht aus einer endlichen Folge $(1, s1), (2, s2), \dots, (m, sm)$, wobei jedes s_t die Form hat



- $x_i = x_j + x_k$
- $x_i = x_j - x_k$
- $x_i = c$
- if $x_i \neq 0$ then goto Line l
- HALT → Beende das Programm

Die Semantik sollte intuitiv klar sein...

WHILE- und FOR - Berechenbarkeit

Def Eine (partielle) Funktion $f: \mathbb{N}^r \to \mathbb{N}$ heißt

WHILE - berechenbar, falls es ein WHILE-Programm gibt, das die Funktion berechnet.

... FOR - berechenbar ...

... GOTO- berechenbar ...

Wir wollen nun diese Begriffe vergleichen, und dann noch mit Turing-Berechenbarkeit zusammenbringen. Wir machen das alles ziemlich skizzenhaft, und lassen ganz viele Details weg ...

WHILE vs FOR vs GOTO?

Welche dieser Programme sind "mächtiger"?

Zwei Richtungen sollten hoffentlich klar sein,
sehen Sie, welche das sind?

"FOR $\leq$ WHILE"

↳ intuitive Schreibweise ...

Satz FOR-Programme können durch WHILE-Programme simuliert werden.

Beweis ist einfach, sehen Sie, wie es geht?

FOR x_i DO P END

WHILE $\not\equiv$ FOR !



For-Programme terminieren immer, while-Programme nicht. Das sollte uns schon zu denken geben...

Es stellt sich heraus, daß es Funktionen gibt, die while-berechenbar, aber nicht for-berechenbar sind!

$\rightsquigarrow$ Ackermann-Funktion, siehe später.

"WHILE ≤ GOTO"

Satz While-Programme können durch Goto-Programme simuliert werden.

Beweis ist einfach, sehen Sie, wie es geht?

"GOTO ≤ WHILE"

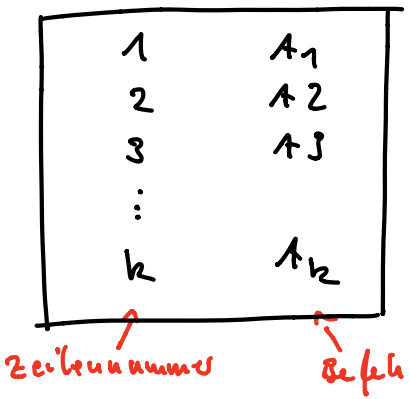
Vielleicht nicht so offensichtlich, es gilt auch die andere Richtung:

Satz GOTO-Programme können durch while-Programme (mit höchstens einer Schleife) simuliert werden.

Beweisskizze: nächste Seite

Beweisskizze GOTO ≤ WHILE

GOTO-Programm: ⇝ **WHILE-Programm:**




```
count = 1
While count ≠ 0 do
    if count = 1 do Ã1 end
    if count = 2 do Ã2 end
    :
    if count = k do Ãk end
end
```

A_i	$\tilde{A}_i$
$x_i := x_\ell + x_j$	$x_i := x_\ell + x_j$; count = count+1
if $x_j = c$ goto t	if $x_j = c$ then count = t else count = count +1 end
HALT	count = 0

Erklärung der Beweisskizze

- Das while-Programm hat einen Counter, der im Wesentlichen der Zeilennummer des goto-Programms entspricht.
- Einfache Anweisungen werden einfach ausgeführt und der Counter wird erhöht.
- Bei goto-Befehlen wird der Counter auf die Zielzeilennummer gesetzt. Die while-Schleife "rechnet" daraufhin diese Zeile (und läuft im Zweifelsfall wieder von vorne an), bis die richtige Zeile gefunden wurde.
- Die while-Schleife wird nur abgebrochen, wenn ein Goto-Programm HALT erreichen will.

"WHILE = WHILE(1) = GOTO"

Wichtige Schlussfolgerungen:

f ist WHILE-berechenbar

$\Leftrightarrow$ f ist WHILE-berechenbar mit while-Programmen, die höchstens eine Schleife haben

$\Leftrightarrow$ f ist goto-berechenbar.

WHILE vs. Turing?

... nächste Vorlesung!

WHILE vs. Turing-Berechenbarkeit

"Turing ≈ while"

Satz WHILE-Programme sind Turing-äquivalent

d.h. wir können Turingmaschinen durch while-Programme simulieren und umgekehrt.

Beide Richtungen schauen wir nun nacheinander an.

"while ≤ Turing"

Sei ein while-Programm gegeben. Wollen zeigen, daß wir dieses Programm durch eine TM simulieren können.

Idee dazu: while-Programme sind induktiv definiert. Wir benutzen daher strukturelle Induktion, um die Aussage zu zeigen.

Gegeben ein while-Programm mit m Variablen. Wir wollen eine TM mit m Bändern konstruieren, so daß der Inhalt von Variable i auf Band i steht.

Erinnerung: Definition von WHILE

Def (1) Ein "einfacher Befehl" ist eine der folgenden drei Anweisungen:

$x_i := x_j + x_k$ oder

$x_i := x_j - x_k$ oder

$x_i := c$

wobei $i, j, k, c \in \mathbb{N}$.

(2) Ein WHILE-Programm P ist entweder ein einfacher Befehl oder hat die Form

(a) while $x_i \neq 0$ do P_1 end

oder (b) $[P_1 ; P_2]$

für $i \in \mathbb{N}$, P_1, P_2 while-Programme.

Strukturelle Induktion: Skizze

Induktionsanfang: Wir können einfache Objekte mit Turingmaschinen simulieren.

Ist intuitiv klar. Wir haben schon gesehen, daß wir arithmetische Operationen durch TMs realisieren können.

Strukturelle Induktion: Skizze

Induktionsschritt 1: Gegeben P_1, P_2 While-Programme.

Induktionsannahme: Haben TMs M_1, M_2, die P_1 und P_2 simulieren.

Zu zeigen: Dann können wir auch die **Hintereinander-ausführung** $[P_1; P_2]$ durch eine TM simulieren.

Intuitiv ist das klar.

Strukturelle Induktion: Skizze

Induktionsschritt: Gegeben ein while-Programm

Induktionsannahme: Können P durch eine TM simulieren.

Zu zeigen: Können dann auch folgende Schleife durch eine TM simulieren

while $x_i \neq 0$ do P end

Intuitiv ist das auch klar.

□

"Turing ≤ while": Skizze

Sei M eine Turingmaschine. Wollen zeigen, daß man ihre Berechnung durch ein while-Programm simulieren kann.

Intuition:

- Derzeitiger Zustand q wird in einer Variable gespeichert
- Inhalt des Lesekopfes (und evtl. was links und rechts davon steht) wird auch in einer Variable gespeichert.
- Übergangsfunktion wird durch eine if-else-Abfrage realisiert: if (Zustand = q AND Bandsymbol = r) ...
  else if (...)
  ...

→

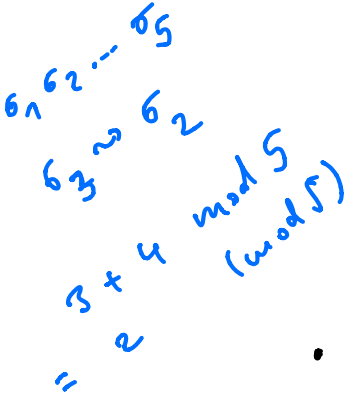
"Turing ≤ while": Skizze

- Überschreiben von Symbolen auf dem Band wird durch eine modulo-arithmetische Operation simuliert

  Alphabet hat k Buchstaben. Dann:

  ersetze Symbol i durch Symbol j ≙

  Inhalt_neu = Inhalt_alt + (j − i) mod k



- Bauen eine while-Schleife: solange die TM nicht in einem Endzustand ist, läuft das Programm. Dazu brauchen wir noch eine Variable, "zuende":

```
while (zuende = 0)
...
end
```

⚠ For-Schleifen würden hier nicht genügen!

Nur Skizzen!

Die Argumente von vorne sind nur skizziert. Für einen formalen Beweis müssten wir sehr viel mehr ausholen.

Insb. müssen wir auch zeigen, dass die Abbildung von TM auf While-Programme auch berechenbar ist...

Konsequenz

Alles, was wir mit "normalen" Programmiersprachen (Java, C++, ...) machen können, können wir im Prinzip auch durch TMs erreichen. Das Berechenbarkeitsmodell der TM ist also sehr passend, und die Ergebnisse (Beschränkungen) übertragen sich!

Andere Modelle

Es gibt noch andere Modelle, die an die Architektur von Computern angelehnt sind, zum Beispiel die Random Access Machine (Registermaschine). Hier kann man, im Gegensatz zu Turingmaschinen, auf beliebige Teile des Speichers zugreifen (Lesen / Schreiben).

Auch dieses Modell ist äquivalent zu Turingmaschinen.

In den nächsten Vorlesungen: rekursive Funktionen und λ-Kalkül als eher mathematische Modelle der Berechenbarkeit.

Primitiv rekursive Funktionen und μ-rekursive Funktionen

Literatur: Schöning 2.4

Hofmann: Theoretische Informatik 8.1.4

Motivation

Ganz anderes Berechnungsmodell als bisher,

sehr viel mehr mathematisch motiviert.

Etwa vor 100 Jahren entwickelt (1920er Jahre),

also noch vor Turingmaschinen.

Primitiv rekursive Funktionen

Wir betrachten Funktionen, die eine feste Anzahl k an natürlichen Zahlen als Eingabe kriegen und eine nat. Zahl als Ausgabe erzeugen:

$$f: \mathbb{N}^k \to \mathbb{N}$$

Eine Funktion mit k Eingaben nennt man auch eine **k-stellige Funktion.**



In diesem Kapitel sind die nat. Zahlen so zu verstehen, daß 0 darin enthalten ist.

Primitiv rekursive Funktionen

Definition: Die Menge der **primitiv rekursiven** Funktionen wird induktiv definiert wie folgt:

"Induktionsanfang"

(1) Die **konstante**, 0-stellige Funktion 0 ist primitiv rekursiv.

(2) Die **Nachfolgerfunktion** $s: \mathbb{N} \to \mathbb{N}$, $s(k) = k+1$ ("successor") ist primitiv rekursiv.

(3) Die **Projektionsfunktionen** der Form

$$f_i: \mathbb{N}^k \to \mathbb{N}, \quad (x_1, x_2, \dots, x_k) \longmapsto x_i$$

sind primitiv rekursiv.

→

Primitiv rekursive Funktionen

Weiter in der Definition:

"Induktionsregeln"

(4) Die Komposition von prim. rek. Funktionen ist prim. rek:

Seien $f: \mathbb{N}^k \to \mathbb{N}$, $g_1, \dots, g_k : \mathbb{N}^m \to \mathbb{N}$ prim. rek., dann auch

$$h(x_1, \dots, x_m) := f\left(g_1(x_1, \dots, x_m), \dots g_k(x_1, \dots, x_m)\right)$$

(5) Primitive Rekursion: Seien $g: \mathbb{N}^k \to \mathbb{N}$, $h: \mathbb{N}^{k+1} \to \mathbb{N}$ prim.rek.
Dann ist auch die folgende Funktion $f: \mathbb{N}^{k+1} \to \mathbb{N}$ prim. rek:

$$f(0, x_1, \dots, x_k) = g(x_1, \dots, x_k)$$

$$f(s(n), x_1, \dots, x_k) = h(f(n, x_1 \dots x_k), x_1, \dots, x_k)$$

$s(n)$: Nachfolger (Intuition: $s(n) = n+1$)

$f(n, x_1 \dots x_k)$: hier findet die Rekursion statt

Primitiv rekursive Funktionen: Beispiel

Beispiel: **Addition** $add: \mathbb{N}^2 \to \mathbb{N}$ ist primitiv rekursiv

$$add(0, x) = x$$

$$add(s(n), x) = s(add(n, x))$$

$s(n) = n+1$

s: Nachfolger

Intuition: $(n+1) + x = (n+x)+1$

Genauso definiert man auch die Addition, wenn man die natürlichen Zahlen mit Hilfe der Peano-Axiome definiert und dann die Addition definiert.

Primitiv rekursive Funktionen $\hat{=}$ FOR

Intuition: Die Rekursion ist so etwas wie eine for-Schleife:

$$f(0, x_1, \dots, x_k) = g(x_1, \dots, x_k)$$ Initialisierung

$$f(\underbrace{n+1}_{=:s(n)}, x_1, \dots, x_k) = h(f(n, \dots), x_1, \dots, x_k)$$

Ergebnis nach $n+1$ Schleifendurchläufen

Berechnung der Schleife basierend auf Ergebnis des vorigen Schleifendurchlaufs

Primitiv rekursive Funktionen ≙ FOR

Satz: Die Menge der primitiv rekursiven Funktionen stimmt genau mit der Menge der FOR-berechenbaren Funktionen überein.

Beweisskizze:

Richtung 1: FOR-berechenbar => primitiv rekursiv berechenbar

Beweis durch strukturelle Induktion:

Jede Variable des FOR-Programms wird durch eine Variable der Funktion beschrieben.

->

Primitiv rekursive Funktionen $\hat{=}$ FOR

Induktionsanfang: Einfache Befehle im For-Programm lassen sich als primitiv rekursive Funktion schreiben:

$$x_i = x_j + x_k \; j \qquad x_i = x_j - x_k \; j \qquad x_i = c$$

intuitiv klar, wir haben schon gesehen, daß Addition primitiv rekursiv ist. Das gleiche gilt für die Subtraktion, und Konstante Funktionen sind eh prim. rek.

Primitiv rekursive Funktionen $\hat{=}$ FOR

Induktionsschritt: Hintereinanderausführung

FOR-Programm: $[P_1; P_2]$

Induktionsannahme: Können P_1, P_2 durch Funktionen g_1, g_2 beschreiben.

Dann können wir auch die Hintereinanderausführung als Funktion schreiben: $g_2(g_1)$

Das ist dann wieder primitiv rekursiv.

Primitiv rekursive Funktionen $\hat{=}$ FOR

Induktionsschritt 2: FOR-Schleife

FOR x_i DO P END

Induktionsvoraussetzung: können P durch eine Funktion g_P beschreiben.

Dann beschreiben wir die FOR-Schleife wie folgt:

$f(0, x_1, \dots, x_k) = \dots$ Initialisierung der Variablen vor der FOR-Schleife

$f(n+1, x_1, \dots, x_k) = g_P(f(n, x_1, \dots, x_n))$

g_P: Berechnung in der for-Schleife

$f(n, x_1, \dots, x_n)$: Zustand nach vorigem Durchlauf

Tut's.

Primitiv rekursive Funktionen $\hat{=}$ FOR

Richtung 2: Primitiv rekursiv => FOR-berechenbar

Funktioniert ähnlich wie die erste Richtung. ∎

Erweiterter Rekursionsbegriff ?

Wissen schon: FOR-Berechenbarkeit ist nicht Turing-äquivalent.

Können wir einen anderen Begriff von Rekursiven Funktionen definieren, der Turing-äquivalent ist!

μ-rekursive Funktionen

Wir führen nun noch einen zweiten Rekursionsoperator ein, die "μ-Rekursion" (μ wie Minimum):

Gegeben eine $(k+1)$-stellige Funktion $f : \mathbb{N}^{k+1} \to \mathbb{N}$ wollen wir über ihr erstes Argument eine Art Minimum bilden:

"
$$\min \{ n \mid f(n) = 0 \}$$
"

Das ist die Intuition, formal sieht es so aus:

μ-rekursive Funktionen

Def Die Menge M der μ-rekursiven Funktionen wird induktiv wie folgt definiert:

- alle primitiv rekursiven Funktionen sind in M enthalten.
- alle Funktionen sind enthalten, die durch Anwendung des μ-Operators aus primitiv rekursiven Funktionen gebaut werden können:

  Sei $f: \mathbb{N}^{k+1} \to \mathbb{N}$ eine (möglicherweise nur partiell definierte) Funktion.

  Dann ist $\mu f: \mathbb{N}^k \to \mathbb{N}$ definiert durch

$$(\mu f)(x_1, \dots, x_k) = \min \{ m \in \mathbb{N}_0 \mid f(m, x_1 \dots, x_k) = 0 \text{ und für alle } u < m \text{ ist } f(u, x_1 \dots, x_k) \text{ definiert} \}$$

  Falls die Menge leer ist, ist $(\mu f)(x_1, \dots, x_k)$ undefiniert.

μ-rekursive Funktionen

Weiter in der Definition:

Insbesondere müssen μ-rekursive Funktionen nicht total sein, sie können auch nur partiell definiert sein.

Das heißt, daß eine Funktion $f: \mathbb{N}^k \to \mathbb{N}$ nicht jedem Element ihres Definitionsbereiches einen Wert zuweisen muß.

μ-rekursive Funktionen: Intuition

Intuition:

suchen das kleinste m, so daß $f(m,\dots) = 0$ gilt

$$(\mu f)(x_1,\dots,x_k) = \min\{\, m \in \mathbb{N}_0 \mid f(m,x_1\dots,x_k) = 0 \text{ und für alle } u < m \text{ ist } f(u,x_1\dots,x_k) \text{ definiert}\,\}.$$

suchen aber nur solche m an, für die die kleineren Werte $f(u,\dots)$ alle definiert sind.



Es kann sein, daß diese Menge leer ist. In dem Fall ist dann die Funktionsvorschrift μf an dieser Stelle nicht definiert, und μf wird eine partielle Funktion.

μ-rekursive Funktionen: Intuition

Was ist der substantielle Unterschied zu primitiver Rekursion?

- Ein Minimum über eine feste Anzahl von Variablen können wir auch primitiv rekursiv definieren.
- Der Knackpunkt ist hier, daß wir ein Minimum über die Menge $\{n \mid f(n,\ldots) = 0\}$ bilden, von der wir vorher gar nicht wissen, wie viele Elemente sie enthalten könnte.

μ-rekursive Funktionen: Intuition

Das klingt schon nach dem Unterschied zwischen FOR (feste Anzahl)

und WHILE (beliebig hohe Anzahl)

Auch die Tatsache, daß die Funktion evtl. nur partiell

definiert sein darf, riecht nach WHILE (Endlosschleife)

μ-rekursive Funktionen $\hat{=}$ WHILE

Satz Die Menge der μ-rekursiven Funktionen stimmt genau mit der Menge der WHILE-berechenbaren Funktionen überein.

Beweisskizze: Prinzip ist wie vorher bei primitiv rekursiven Funktionen durch strukturelle Induktion.

- Wissen: primitiv rekursiv $\hat{=}$ FOR
- Fehlt nur noch: μ-Operator $\hat{=}$ while-Schleife

μ-rekursive Funktionen $\hat{=}$ WHILE

While-Schleife $\Rightarrow$ μ-Operator, Intuition:

- Betrachte das Programm

  WHILE $x_i \neq 0$ DO Q END

  wobei wir annehmen, daß Q durch eine μ-rek. Funktion dargestellt werden kann.

- Sei $h(m, \dots)$ der Wert der Variable x_i, nachdem die While-Schleife m mal durchlaufen wurde.

- Durch den μ-Operator können wir herausfinden, wann die Variable x_i zum ersten mal 0 geworden ist.

- Dann können wir eine primitiv rekursive Funktion definieren, die genau so oft hintereinander ausgeführt wird.

μ-rekursive Funktionen $\hat{=}$ WHILE

μ-rekursiv $\Rightarrow$ while-Schleife: ähnlich

□

Wo sind wir?

- primitiv rekursiv $\hat{=}$ FOR
- μ-rekursiv $\hat{=}$ WHILE
- FOR $\leq$ WHILE aber nicht umgekehrt.

Also muß es Funktionen geben, die μ-rekursiv, aber nicht primitiv rekursiv sind. Kann man davon eine explizit hinschreiben? $\rightsquigarrow$ Ackermann-Funktion

typo

Die Ackermann-Funktion

Literatur: Schöning, Hofmann

Ackermann-Funktion: Definition

Beispiel für eine Funktion, die μ-rekursiv aber nicht primitiv rekursiv ist.

Die Funktion ist ganz einfach definiert: $a : \mathbb{N}_0^2 \to \mathbb{N}_0$

Basis-fälle

$$a(0, y) = y + 1 \quad \text{für } y \geq 0$$

$$a(x, 0) = a(x-1, 1) \quad \text{für } x > 0$$

Rekursion

$$a(x, y) = a(x-1, a(x, y-1)) \quad \text{für } x, y > 0.$$

(Erfunden von Mathematiker Ackermann, 1920-er Jahre)

Ackermann-Funktion, erste Intuition

Die Ackermann-Funktion wächst wahnsinnig schnell:

$$a(1,y) = y + 2$$

$$a(2,y) = 2y + 3$$

$$a(3,y) = 2^{y+3} - 3$$

$$a(4,y) = \underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{y \text{ mal den Exponenten bilden}} - 3$$

Ackermann-Funktion ist while-berechenbar



Intuitiv ist es klar, daß man diese Funktion durch ein rekursiv definiertes Programm "berechnen" kann

```
function a(x,y)
    if x=0 , then return y+1
    elseif y=0 then return a(x-1, 1)
    else
            return a(x-1, a(x, y-1))
    endif
end function
```

Aber bisher haben wir noch nicht überlegt, ob while-Programme auch Rekursionen simulieren können ?!?

Sind rekursive Programme while-berechenbar?

Vermutlich haben Sie schon in einer der Programmier-Vorlesungen gesehen: Hinter den Kulissen werden rekursive Funktionsaufrufe mit Hilfe eines Stacks und einer while-Schleife im iterativen Code umgesetzt.

Die intuitive Antwort ist also "ja".

Wir schauen uns als ein spezielles Beispiel eine etwas einfachere Variante für die Ackermann-Funktion an:

Ackermann-Funktion ist while-berechenbar

Die Ackermann-Funktion ist while-berechenbar

- Haben schon gesehen: While-Programme können Stacks implementieren.
- Nun simulieren wir die Rekursion der Ackermann-Funktion mit einem Stack und einer while-Schleife:

- Die Idee ist, daß wir uns auf dem Stack "merken", welche rekursiven Aufrufe wir als nächstes machen müssen. In jedem Rekursionsschritt holen wir neue Argumente vom Stack, und legen die rekursiven Aufrufe wieder auf den Stack drauf, bis wir im Basisfall ankommen:

Ackermann-Funktion ist while-berechenbar

```
function ackermann (x, y)
    Push(x), Push(y)
    while (Stack > 1)
        y = Pop;  x = Pop;
        if x=0 then push(y+1)
        elsif y=0 then push(x-1); push(1)
        else push(x-1), push(x), push(y-1)
        endif
    end while
    result = POP
end function
```

→ hole die nächsten Elemente vom Stack

Fallunterscheidung wie in Ackermann, lege die rekursiven Aufrufe wieder auf den Stack

Tut's.

Ackermann-Funktion ist nicht FOR-berechenbar

Warum ist $a(n,m)$ nicht FOR-Berechenbar?

Begründung 1 (Skizze)

- Kann zeigen: zur Berechnung von $a(n,m)$ braucht man mindestens $n+2$ FOR-Schleifen.
- Für festes n ist es also durch FOR-Schleifen darstellbar.
- Aber für beliebiges n bräuchte man eine unbeschränkte Anzahl an for-Schleifen. Das ist aber im FOR-Modell nicht enthalten.

Ackermann-Funktion ist nicht FOR-berechenbar

Warum ist $a(n,m)$ nicht FOR-Berechenbar?

Begründung 2 (Skizze):

- Zeige Monotonie: $\forall x \le x' \; \forall y \le y' : a(x,y) \le a(x',y')$
- Zeige: Für jedes FOR-Programm gibt es eine Konstante c_p, so daß die Summe aller Variablen bei Beendigung des Programms durch die Ackermann-Funktion beschränkt ist:

  maximale Summe $\leq a(c_p, n)$ (Summe bei Start)

  wichtig, echte Ungleichung

  Beweis: Induktion

- Falls man die Ackermann-Fkt mit FOR-Programmen berechnen könnte, führt das zu einem Widerspruch:

  $a(n,n) < a(n,n)$ ↯

Rückblick und Ausblick

Hilbert hatte 1926 vermutet: alle "berechenbaren" Funktionen sind primitiv rekursiv. Wurde von Ackermann 1928 widerlegt.

Es gibt natürlich noch viele weitere Möglichkeiten, wie man Rekursion einsetzen / definieren könnte.

Aber alle davon sind äquivalent zu while-Programmen.

Church - Turing - These

Geschichte

1930er Jahre:

- Turing erfindet die Turing-Maschine
- Church erfindet den λ-Kalkül
- Gödel (und andere) definieren rekursive Funktionen

Innerhalb weniger Jahre wird bewiesen, daß alle drei Modelle äquivalent (im Sinne der Berechenbarkeit) sind.

Church-Turing-These

In den 1930er-Jahren wurde dann eine Hypothese formuliert, die in heutigen Worten etwa wie folgt lautet:

> Alles, was "intuitiv" berechnet werden kann, kann auch von einer Turingmaschine berechnet werden.

≈ in den Gesetzen der Physik

> Jede "effektiv berechenbare" Funktion kann von einer TM berechnet werden

Church - Turing - These

Etwas moderne Formulierungen

Jeder "Algorithmus" kann mit Hilfe einer Turingmaschine implementiert werden.

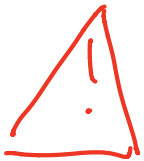
Every finite physical System can be simulated to any specified degree on a universal Turing machine.

Andere "klassische" Berechnungsmodelle

Es gab viele Versuche, alternative Berechnungsmodelle aufzustellen, die einerseits physikalisch plausibel sind (z.B. kann man nicht unendlich viele Berechnungen in einem Schritt durchführen), und andererseits mächtiger als Turingmaschinen sind.

Hat nicht geklappt.

Hypothese, keine Tatsache!



Die Church-Turing-These ist eine Vermutung über die Beschaffenheit der Welt.

Sie ist nicht beweisbar oder widerlegt, da sie ja auch nicht in Form einer mathematischen Aussage beschrieben wird.

Was sagt die Physik?

Physikalische Grenzen?

Was genau sind die physikalischen Grenzen der Berechenbarkeit (d.h. was kann man prinzipiell bauen)?

~> viel Diskussion und Literatur

Quanten-Computer sind Turing-äquivalent

Jeder Quantencomputer kann durch einen normalen Computer simuliert werden (allerdings mit exponentiell mehr Zeit).

Wenn Rechenzeit aber nicht das Kriterium ist, dann sind beide Modelle äquivalent.

David Deutsch: Quantum theory, the Church-Turing-Principle and the universal quantum computer. 1985

Andere physikalische Modelle?

⇝ "Hypercomputing" (z.B. accelerating machine, relativistic machine)

Hier kann man Modelle definieren, die formal mächtiger als TMs sind. Die Frage ist aber, ob sie wirklich gebaut werden können.

Und was ist mit

künstlicher Intelligenz?

"Neuronale Netze" sind oft Turing-äquivalent

→ unter bestimmten Voraussetzungen

Literatur: • Perez, Marinkovic, Barcelo: On the Turing completeness of modern neural network architectures. ICLR 2019

• What graph neural networks cannot learn: depth vs width. ICLR 2020

hängt sehr an den Modellen, ihren Parametern, usw.

Details jenseits der Vorlesung.

Und ich habe es nicht im Detail gelesen und kann nicht beurteilen, ob ich es glaube oder nicht. Hängt sehr stark daran, wie man es formuliert, vermute ich.

Diskussion um die These

Es gibt nach wie vor viel Diskussion über die Church-Turing-These.

- Manche Argumente sagen, daß "Funktionen auf Strings" einen eingeschränkten (zu mathematischen) Verständnis von Berechnungen sind.

- Manche sagen, daß das traditionelle Konzept eines Algorithmus zu eingeschränkt ist. (z.B. im Kontext von maschinellem Lernen, das ja nicht statisch ist sondern von (großen) Trainingsdaten abhängt).

Mein eigenes Bauchgefühl

Ich würde nicht darauf wetten, daß alles, was sich derzeit im Gebiet der künstlichen Intelligenz tut, auf Turing-Berechenbarkeit zurückführen läßt.

Aber am Ende ist das vielleicht auch gar keine so interessante Frage mehr? Bzw. trifft den Kern der Probleme nicht mehr?

Die offensichtliche Frage am Horizont ist, ob es prinzipiell "künstliche Intelligenz" geben kann.

(Physikalisch spricht nichts dagegen, zumindest werden wir finden, daß Menschen intelligent sind).

Vorlesung: Ende der Berechenbarkeitstheorie

Haben gesehen:

- viele Modelle für "Berechnung"
- alle sind eingeschränkt: wir können nicht alles berechnen, was wir wollen, selbst mit beliebigen Resourcen

  Beispiel: Halteproblem, Tiling-Problem
- viele Modelle sind äquivalent!

Wie geht es jetzt in der Vorlesung weiter?

In der Praxis haben wir beschränkte Ressourcen!

- ⇝ Zeit
- ⇝ Speicherplatz

⇝ Komplexitätstheorie

Der Unvollständigkeitssatz von Gödel

Literatur: Schöning Kap. 2.9
(Sipser 6.2)

Gödel'scher Unvollständigkeitssatz

→ verifizierbar, mächtig genug, wenn die wichtigsten Eigenschaften der Nat. Zahlen und deren Operatoren + und · beschreiben

Satz Jedes Beweissystem für die Menge der wahren arithmetischen Formeln ist notwendigerweise unvollständig: es gibt immer wahre arithmetische Formeln, die nicht beweisbar sind.

Formeln und logische Aussagen über die natürlichen Zahlen, mit den Operatoren + und · und ∃ und ∀ Quantoren

Wir wollen zunächst alle Begriffe einführen und dann eine (moderne Variante) des Beweises betrachten.

Arithmetische Formeln

Über den natürlichen Zahlen wollen wir die folgenden Dinge zur Verfügung haben:

- Rechenoperationen $+$ und $\cdot$
- Variablen
- $=$
- Logische Operationen $\neg$, $\vee$, $\wedge$
- Quantoren $\exists$, $\forall$

Aus diesen Dingen können wir "Formeln" (Aussagen) hinschreiben, die wahr oder falsch sein können. Formal formuliert das folgendermaßen:

Arithmetische Formeln

Def Ein **Term** wird induktiv wie folgt definiert:

- Jedes $n \in \mathbb{N}$ ist ein Term.
- Jede Variable x_i $(i \in \mathbb{N})$ ist ein Term.
- Wenn t_1, t_2 Terme sind, dann auch $(t_1 + t_2)$ und $(t_1 \cdot t_2)$

Eine **Formel** wird induktiv wie folgt definiert:

- Wenn t_1, t_2 Terme sind, dann ist $(t_1 = t_2)$ eine Formel.
- F, G Formeln, dann sind auch $\neg F$, $(F \wedge G)$ und $(F \vee G)$ Formeln.
- Wenn x eine Variable und F eine Formel ist, dann sind auch $\exists x\, F$ und $\forall x\, F$ Formeln.

Beispiel einer Formel:

$$\forall x \, \exists y \, \forall z \, \big(((x \cdot y) = z) \wedge ((1+x) = x \cdot y) \big)$$

Für arithmetische Formeln können wir nun definieren, wann die Formel wahr ist. Ich lasse die formale Definition weg, es ist intuitiv klar, was gemeint ist.

Beispiel: • $\forall x \, \exists y \, ((x+y) = x \cdot (x+1))$ ist wahr

• $\forall x \, \exists y \, ((x=0) \vee (x \cdot y) = 1)$ ist falsch.

Intuition, wie definiert man, wann eine Formel wahr ist?

- eine Formel $(t_1 = t_2)$ ist wahr, falls für alle möglichen Variablenbelegungen die Gleichheit gilt.
- logische Operatoren "wie üblich":
  - $\neg F$ ist wahr $\Leftrightarrow$ F ist nicht wahr
  - $F \wedge G$ ist wahr $\Leftrightarrow$ F und G sind beide wahr
  - $\vee$ analog
- $\exists x\, F$ ist wahr wenn es ein $n \in \mathbb{N}$ gibt, so dass wenn man x in F durch n ersetzt, dann F wahr ist.
- $\forall x\, F$ analog

Arithmetisch repräsentierbare Funktionen

Def Eine Funktion $f: \mathbb{N}^k \to \mathbb{N}$ ist **arithmetisch repräsentierbar**, falls es eine arithmetische Formel $F(x_1, \dots, x_k, y)$ gibt, so daß gilt:

$$f(n_1, n_2, \dots, n_k) = m \iff F(n_1, \dots, n_k, m) \text{ ist wahr.}$$

- Intuition: die Formel F ist sowas wie ein Verifizierer für die Funktion.
- Gegeben eine Funktion f (die beliebig kompliziert oder sogar zufällig gewählt sein kann), ist es nicht offensichtlich, wie man F konstruieren kann, wir haben ja nur wenige Operatoren über $\mathbb{N}$ zur Verfügung!

Intuition: Beweissystem

- Ein Beweis besteht aus einer Folge von Formeln, die entweder Axiome sind, oder anhand der logisch erlaubten Schlüsse aus den Axiomen hergeleitet werden können.
- Ein Beweis soll "entscheidbar" oder verifizierbar sein, d.h. durch ein vorher festgelegtes Verfahren kann man nachprüfen, ob der Beweis richtig ist oder nicht.
- Ein Beweis soll nur dann "wahr" sein, wenn die durch ihn bewiesene Aussage wirklich wahr ist ("Korrektheit des Systems")

Intuition: Beweissystem

Ein Beweissystem heißt widerspruchsfrei, wenn es keine Formel gibt, für die man sowohl F als auch $\neg F$ beweisen kann.

Ein Beweissystem ist vollständig, wenn man für jede Formel F entweder F oder $\neg F$ beweisen kann.

Intuition: Beweissystem

- Bei der Wahl eines Beweissystems legt man fest, welche Axiome gelten sollen und welche logischen Schlussregeln man anwenden darf.

- Damit der Gödelsche Unvollständigkeitssatz gilt, muss das verwendete Beweissystem hinreichend stark sein, so dass man zumindest die üblichen Aussagen über $\mathbb{N}$, die Operationen $+$ und $\cdot$ usw zeigen kann. (Sehen wir nachher genauer).

- Wir legen uns im folgenden nicht auf ein Beweissystem fest, das ist ja der Witz: sobald es "mächtig genug" ist, muss es unvollständig sein.

Beweisidee des Satzes von Gödel

Im folgenden werden wir nun den Beweis des Satzes von Gödel skizzieren.

Ich teile ihn in mehrere Schritte auf.

Schritt 1: Jede while-berechenbare Funktion ist "arithmetisch repräsentierbar" (kann durch eine Formel ausgedrückt werden).

Beweis von Schritt 1:

- Den beweist man durch strukturelle Induktion über den Aufbau von while-Programmen.
- Man nutzt hier stark aus, daß man jede endliche Zahlenfolge durch eine geschickt gewählte Modulo-Formel erzeugen kann.

  Hier gehen viele Eigenschaften der nat. Zahlen ein, und alle Operatoren sind essentiell: $+, \cdot, \forall, \exists, \ldots$.

Schritt 2 Die Menge WA der wahren arithmetischen Formeln ist nicht rekursiv aufzählbar.

Beweis von Schritt 2:

(1) Wenn WA rekursiv aufzählbar wäre, dann wäre die Menge schon entscheidbar:

- zu einer Formel F ist entweder F oder ¬F wahr,
  eines von beiden muss also in der Aufzählung vorkommen
- aber sobald das eine vorkommt,
  wissen wir auch, dass die negierte Formel falsch ist

Weiter in Schritt 2:

(2) • Wir nehmen nun eine Sprache L, die rekursiv aufzählbar aber nicht entscheidbar ist, zum Beispiel H vom Halteproblem.

• Diese Sprache reduzieren wir nun auf WA: $L \leq WA$

Die Reduktion funktioniert so: L rekursiv aufzählbar
$\Rightarrow$ L while-berechenbar $\overset{\text{Schritt 1}}{\Rightarrow}$ durch arithmetische Formel repräsentierbar so daß: $w \in L \Leftrightarrow$ Formel wahr.

• Da L nicht entscheidbar ist, ist nun auch WA nicht entscheidbar, und wegen (1) dann auch nicht rekursiv aufzählbar.

Schritt 3: Jedes Beweissystem für die Menge der wahren arithmetischen Formeln ist unvollständig

Beweis von Schritt 3

- Angenommen das Beweissystem ist vollständig.
- Dann können wir die wahren Formeln rekursiv aufzählen, indem wir alle potentiellen Beweise durchgehen und prüfen, ob sie wahr sind. Für die wahren Beweise lassen wir nur dann die Aussage ausgeben, die gerade bewiesen wurde.
- Aber in Schritt 2 haben wir gesehen, daß WA nicht rekursiv aufzählbar ist. ↯



Was sind die "Schlüsselzutaten" im Theorem?

- Arithmetische Formeln sind mächtig: Jede while-berechenbare Funktion kann als Formel geschrieben werden. = Schritt 1

  Hier gehen ganz entscheidend die Eigenschaften der nat. Zahlen ein (und daß das Beweissystem hinreichend stark ist, so daß wir diese benutzen können).

- Wenn es ein vollständiges Beweissystem gäbe, dann könnten wir es benutzen, um die wahren Formeln rekursiv aufzuzählen. = Schritt 3

- Aber die wahren Aussagen sind nicht rekursiv aufzählbar = Schritt 2

  Hier geht ein , daß es Sprachen gibt, die rekursiv aufzählbar aber nicht entscheidbar sind, wie das Halteproblem.

Was sind die "Schlüsselzutaten" im Beweis?

- Der Beweis würde schiefgehen, wenn wir über $\mathbb{N}$ nur die Addition, aber nicht die Multiplikation hätten (in Schritt 1)

  Die entstehenden Formeln sind dann nicht mächtig genug, alle while-Berechnungen zu beschreiben.

- Wir brauchen wirklich while-Berechenbarkeit, mit for würde es nicht funktionieren (in Schritt 3).

  Denn wir müssen ja eine Sprache aus RE \ R verwarten, und dafür reicht FOR nicht aus.

Fazit

- Es ist erstaunlich, wie eng man arithmetische Formeln mit Turingmaschinen verknüpfen kann.
- Der Beweis vom Gödelschen Satz ist für uns nicht außer Reichweite! Cool, oder?!?

Disaster der Informatik

- Die Tatsache, daß es nicht berechenbare Funktionen gibt ist ganz eng verbandelt damit, daß man die natürlichen Zahlen nicht "perfekt" axiomatisieren kann.

Disaster der Mathematik

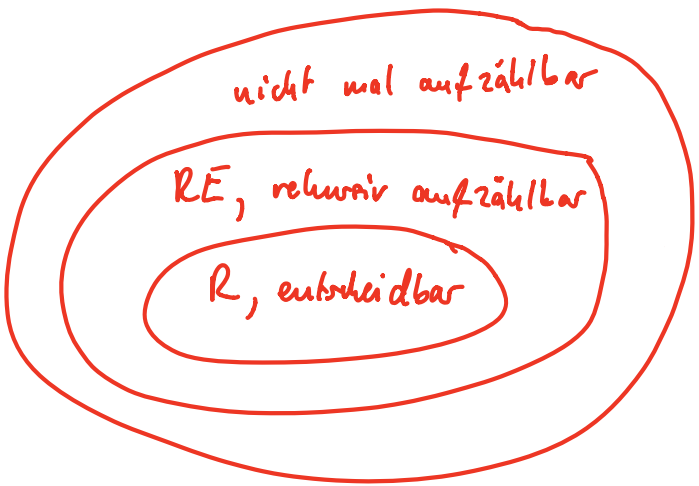
Komplexitätstheorie

Literatur: Sipser, Hromkovič, Arora/Barak

Motivation

Bisher Berechenbarkeitstheorie: Was kann man prinzipiell berechnen, wenn man beliebig viel Zeit und Speicherplatz hat?

$\rightsquigarrow$ prinzipielle Beschränkungen



Motivation

Nun wollen wir uns mit etwas realistischerem Blick anschauen, wie es sich mit den entscheidbaren Sprachen verhält.

Welche davon sind **effizient entscheidbar?**



Rechenzeit

Speicherplatz

Was erwartet uns?

- Einteilung der Probleme in "effizient" lösbare Probleme (P)
  "nicht effizient lösbare" Probleme (NP, NPC, und jenseits)
- Speicherplatz vs Rechenzeit
- $P \stackrel{?}{\neq} NP$, das größte Rätsel der theoretischen Informatik
- besonders schwere Probleme (NPC)
- je nach Zeit dann noch Hierarchien, Zufall, ...
  - Hierarchien: "schwerer" als NP
  - Zufall: hilft er oder nicht?

Komplexitätsmaße

TIME und SPACE

Literatur: Hromkovič 6.2, Sipser 7.1

(Barak / Arora 1)

Motivation

- Wollen messen, wie "schnell" / mit welchem Zeitaufwand ein Problem lösbar ist.
- Soll unabhängig von der gewählten Architektur sein, wollen daher nicht in Sekunden messen
- Unabhängige Einheit: wie viele Rechenschritte brauchen wir
  - auf einer bestimmten Instanz
  - im best / worst case über alle Instanzen
  - im Mittel ?.

Zeitkomplexität

Def

Sei eine deterministische Einband-Turingmaschine M über dem Eingabealphabet Σ gegeben, die immer anhält.

Dann ist die Zeitkomplexität $\text{Time}_M(x)$ der Berechnung von M auf x die Anzahl der Rechenschritte, die M bis zum Anhalten durchführt.

Die Zeitkomplexität von M ist eine Funktion (!)

$$\text{Time}_M : \mathbb{N} \to \mathbb{N}$$

definiert durch

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

worst case bei Eingabelänge n

Speicherplatzkomplexität

Def Sei eine deterministische Einband-Turingmaschine M über dem Eingabealphabet Σ gegeben, die immer anhält.
Sei C eine Konfiguration von M.

Die Speicherplatzkomplexität der Berechnung von M auf x, $\mathrm{Space}_M(x)$ ist die maximale Anzahl an nicht-leeren Zellen auf dem Ein/Ausgabeband während der Berechnung von M auf x:

Die Speicherplatzkomplexität von M ist eine Funktion
$\mathrm{Space}_M : \mathbb{N} \to \mathbb{N}$, definiert durch

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

worst case

Speicherplatzkomplexität



Eine Zelle gilt als nicht-leer, sobald die TM einmal auf dieser Zelle operiert hat: in jedem Rechenschritt muß die TM ja ein Symbol in die derzeitige Zelle des Bandes schreiben. Sobald das einmal geschehen ist, gilt die Zelle nicht mehr leer.

(Manchmal könnte man zum Beispiel eine sehr lange Zahl n dadurch codieren, daß man n leere Zellen läßt und am Ende ein spezielles Symbol anbringt).



Beachten Sie auch: die TM kann auf dem Band nicht an beliebige Stellen springen, sie kann sich nur "linear" fortbewegen.

Ergänzung im Nachhinein

Obere / untere Schranken

Sei L eine Sprache (f eine Funktion).

Wir sagen, daß die Funktion $g: \mathbb{N} \to \mathbb{N}$,

$g(n)$ eine **obere Schranke** für die Zeitkomplexität von L (von f)

ist, falls es eine deterministische TM M gibt, die L entscheidet (f berechnet)

so daß $\text{Time}_M(n) \in O(g(n))$ ist.

↳ O-Notation siehe separates Video!

Wir sagen, daß $g(n)$ eine **untere Schranke** für die Zeitkomplexität ist, falls für **jede** deterministische TM M, die L entscheidet (f berechnet), gilt:

$$\text{Time}_M(n) \in \Omega(g(n)).$$

Obere / untere Schranken, nochmal

Obere Schranke $g(n)$:

$\exists M : Time_M(n) \in O(g)$

$\exists M : \max_{|w|=n} T_M(w) \in O(g)$

$\exists M \quad \exists c > 0 \quad \forall w \text{ mit } |w| = n : T_M(w) \leq c \cdot g(n)$

Untere Schranke $g(n)$:

$\forall M : Time_M(n) \in \Omega(g)$

$\forall M : \max_{|w|=n} T_M(n) \in \Omega(g)$

$\forall M \quad \exists c > 0 \quad \exists w \text{ mit } |w| = n : T_M(w) \geq c \cdot g(n)$

Ausführlich

L hat Zeitkomplexität höchstens $g(n)$, wenn es eine deterministische TM M gibt, deren Rechenzeit bei jedem Eingabewort der Länge n nicht länger als $\text{const} \cdot g(n)$ ist.

Es gibt einen (schnellen) Algorithmus, der beim schlechtesten Wort nicht länger braucht als $\text{const} \cdot g(n)$

↳ gleiche Konstante für alle n

Anschaulich

L hat Zeitkomplexität mindestens $g(n)$,

wenn jede deterministische TM M

beim schwierigsten Wort mindestens Zeit const. $g(n)$ braucht.

Jeder Algorithmus (und sei er noch so schlau) braucht

für ein schwieriges Wort mindestens Zeit const. $g(n)$

Details der Turingmaschine sind fast egal

Im Moment hängt unsere Definition der Komplexität noch von der Turingmaschine M ab, die wir benutzen:

Aber wie wir gleich sehen werden, ist der genaue Aufbau der TM "fast egal":

Details der Turingmaschine sind fast egal

Alphabet:

Beh Sei $f: \{0,1\}^* \to \{0,1\}^*$ eine Funktion, die von einer TM M mit Arbeitsalphabet Σ in Zeit $T(n)$ berechnet werden kann. Dann kann die Funktion auch von einer TM $\tilde{M}$ mit Arbeitsalphabet $\{0,1,\square,\triangleright\}$ in

$$\tilde{T}(n) = 4 \cdot \log|\Sigma| \cdot T(n)$$

berechnet werden.

Beweisidee: Kodiere jeden Buchstaben von Σ mit $\log|\Sigma|$ vielen binären Bits.

d.h. $f \in \underset{\Omega}{O}(T(n)) \iff f \in \underset{\Omega}{O}(\tilde{T}(n))$

Details der Turingmaschine sind fast egal

Anzahl Bänder

Satz Sei M eine k-Band-TM, die die Funktion $f: \{0,1\}^* \to \{0,1\}^*$ in Zeit $T(n)$ berechnet. Dann gibt es eine Einband-TM, die die gleiche Funktion in $5 \cdot k \cdot T(n)^2$ Zeit berechnet.

Bew ü (~ Barak/Arora Kap. 1)

usw

Details der Turingmaschine sind fast egal

Ähnliche Konzepte für die Bewegung des Schreibkopfes, Band einseitig oder zweiseitig, usw.

Details der Turingmaschine sind fast egal

Solange wir uns (wie im folgenden) dafür interessieren,
ob eine Laufzeit polynomiell oder exponentiell ist,
führen alle sinnvollen TM-Architekturen zu den gleichen
Komplexitätsklassen...

Bis auf eine entscheidende Ausnahme:

Deterministisch vs Nicht-deterministisch

... bis auf deterministisch / nicht-deterministisch



Es ist aber nicht egal, ob wir deterministische oder nicht deterministische Turingmaschinen betrachten:

Die Verwendung einer nicht deterministischen TM kann die Berechnung "exponentiell schneller" machen.

Umgekehrt haben wir gesehen, daß wir jede nicht-det. TM mit einer deterministischen TM simulieren können, aber die Laufzeit kann hierbei exponentiell größer werden.

... bis auf deterministisch / nicht-deterministisch

Kap. Berechenbarkeit, Nicht-det. TM äquivalent zu det. TM

Ü: schauen Sie den Beweis dazu an und achten Sie darauf, was bei der Konstruktion mit der Laufzeit passiert.

Können Sie ein Beispiel bauen, wo die Laufzeit exponentiell größer wird?



Wir werden im Folgenden also peinlich genau darauf achten, ob wir gerade deterministische oder nicht-det. TMs verwenden.

Nächster Schritt

Definition der Komplexitätsklasse P

$\rightsquigarrow$ nächste Vorlesung

Oblivious Turing machines

Dieses Kapitel habe ich in der Vorlesung dann doch nicht behandelt, Sie können es überspringen

Literatur: Arora / Barak 2.3

"Oblivious" (unwissende) TM

Im Beweis wollen wir eine ganz spezielle TM verwenden, eine "oblivious TM" ("unwissende" TM).

Anschaulich: bei einer oblivious TM hängen die Bewegungen des Lese/Schreibkopfes nicht vom Inhalt der Eingabe ab, sondern nur von deren Länge.

Def. (def.) Eine TM M heißt "oblivious", falls gilt: für jede Eingabe $x \in \{0,1\}^*$ und jedes $i \in \mathbb{N}$ hängt die Position des Kopfs von M zum Zeitpunkt i nur von $|x|$ und i ab.

"Oblivious" (unwissende) TM

Intuition:

- Kopfbewegung ist deterministisch vorbestimmt.
- In jedem Schritt liest die TM den aktuellen Buchstaben und schreibt einen neuen (in Abh. vom Zustand), aber wohin sie den Kopf bewegt ist vorher bestimmt.
- Übergangsfunktion $\delta : Q \times \Gamma \to Q \times \Gamma$

  (statt $\delta : Q \times \Gamma \to Q \times \Gamma \times \{l, r\}$)

Hört sich nach einer starken Einschränkung an, ist es aber nicht:

Oblivious ≈ normale TM

Man kann zeigen, daß oblivious TMs "äquivalent" zu normalen TMs sind, in folgendem Sinne:

Satz: Sei $L \in DTIME(T(n))$, d.h. es gibt eine det. TM M, die L in Zeit $T(n)$ entscheidet. Dann gibt es eine oblivious TM $\tilde{M}$, die L in Zeit $O(T(n)^2)$ entscheidet.

Oblivious ≈ normale TM : Beweis

Beweisidee : Betrachte die Berechnung der normalen TM M.
Wir wollen diese Berechnung durch eine oblivious
Berechnung ersetzen, wo der Kopf sich stur
deterministisch bewegt.

Beobachte zunächst:

Bis Schritt i kann die TM M der Band höchstens bis Zelle i
berührt haben.

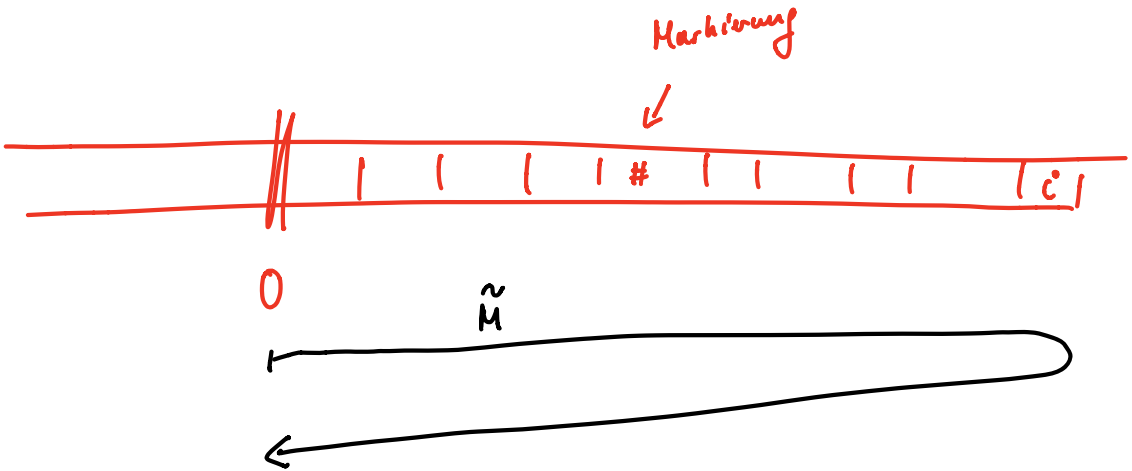
Oblivious ≈ normale TM: Beweis

Die TM $\tilde{M}$ simuliert Schritt i von M wie folgt:

- eine Markierung auf dem Band markiert die Stelle, wo der Kopf von M stehen würde.
- $\tilde{M}$ geht das ganze Band von Pos. 1 bis Pos. i entlang. (vorbestimmt)
  Wenn es auf die Markierung trifft, kommt es in einen speziellen Zustand und überschreibt das Symbol auf dem Band so, wie M es getan hätte.
- Falls M nach rechts gegangen wäre, verschiebt $\tilde{M}$ die Markierung auf dem Hinweg um eins nach rechts.
  (Fall links: Rückweg)

Oblivious ≈ normale TM: Beweis

Anschaulich: der eine (!) Rechenschritt i der TM M wird durch $2i$ Schritte der TM $\tilde{M}$ simuliert:



Oblivious ≈ normale TM: Beweis

Wir lassen ganz viele Details weg. Aber wir können uns davon überzeugen, daß die Simulation $\tilde{M}$ wirklich das gleiche Ergebnis produziert wie M.

Oblivious ≈ normale TM: Beweis

Rechenzeit von $\tilde{M}$:

Wenn M $T(n)$ Schritte braucht, dann braucht $\tilde{M}$ $c \cdot (T(n))^2$ viele Schritte, denn Schritt i von M braucht Aufwand $c \cdot i$ in $\tilde{M}$. Damit:

$$T_{\tilde{M}}(n) = c \cdot \sum_{i=1}^{T(n)} i = \tilde{c} \cdot (T(n))^2$$

(Ich lasse hier viele Details weg, zum Beispiel braucht $\tilde{M}$ mehrere Bänder, um sich manche Dinge zu merken).

□

Stärkere Schranke: $T \log T$

Es gibt sogar noch eine stärkere obere Schranke:

statt $O(T(n)^2)$ kann man auch in

$O(T(n) \log(T(n)))$ Zeit simulieren. Ist aber kompliziertes,

lassen wir weg.

Wiederholung:

O - Notation

Motivation

- Wollen das Wachstum von Funktionen messen:

  "Exponentialfunktion wächst schneller als Polynom"

- Interessieren uns aber nur für die grobe Größenordnung

O-Notation: Definitionen

Seien $f, g : \mathbb{N} \to \mathbb{N}$ Funktionen.

großes O

$$f \in O(g) \iff \exists c > 0 \; \exists n_0 > 0 \; \forall n > n_0 : \; 0 < f(n) \leq c \cdot g(n)$$

$$\iff \limsup_{n \to \infty} \frac{f(n)}{g(n)} < \infty$$

"f ist von Größenordnung höchstens g"

kleines o

$$f \in o(g) \iff \forall c > 0 \; \exists n_0 > 0 \; \forall n > n_0 : \; 0 < f(n) < c \cdot g(n)$$

$$\iff 0 \leq \limsup_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

"f hat Größenordnung echt kleiner als g"

O-Notation: Definitionen

$f \in \Omega(g) \iff \exists c > 0 \; \exists n_0 > 0 \; \forall n > n_0 : f(n) \geq c \cdot g(n) > 0$

großes Omega

$\iff 0 < \liminf_{n \to \infty} \frac{f(n)}{g(n)} \leq \infty$

$\iff g \in O(f)$

kleines Omega

$f \in \omega(g) \iff \forall c > 0 \; \exists n_0 > 0 \; \forall n > n_0 : f(n) > c \cdot g(n) > 0$

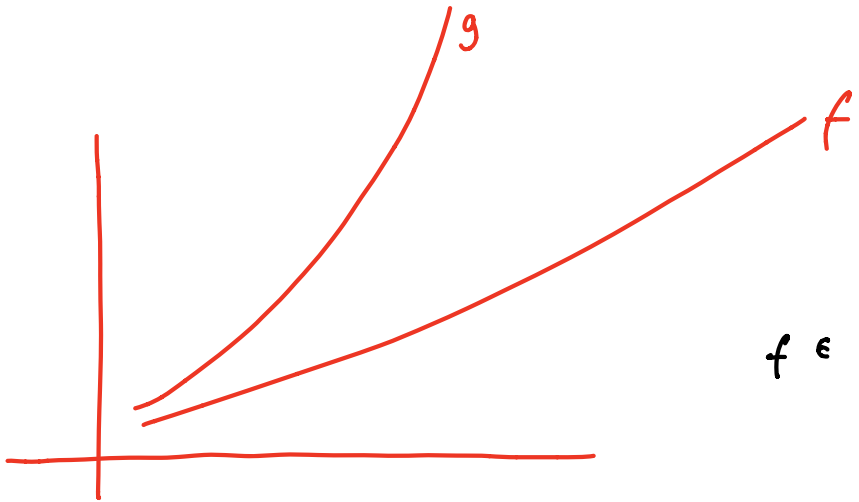
$\iff \liminf \frac{f(n)}{g(n)} = \infty$

$\iff g \in o(f)$

Theta

$f \in \Theta(g) \iff f \in O(g) \text{ und } f \in \Omega(g)$

Intuition



$f \in O(g)$, $g \in \Omega(f)$

$f \in O(g)$: f wächst höchstens so schnell wie g

$f \in o(g)$: f wächst echt langsamer als g

$f \in \Omega(g)$: f wächst mindestens so schnell wie g

$f \in \omega(g)$: f wächst echt schneller als g

Beispiele

$f(n) = n^2 + 3n - 7$;

Dann gilt zum Beispiel:

$f \in O(n^{10})$, denn $\limsup \frac{f(n)}{g(n)} = 0 < \infty$

$f \in o(n^{10})$, denn $\limsup f(n)/g(n) = 0$

$f \in w(\log(n))$, denn $\liminf f(n)/g(n) = \infty$

$f \in \Theta(n^2)$, denn $\liminf f(n)/g(n) = \limsup f(n)/g(n) = 1$

$f \notin o(n^2)$, denn $\limsup f(n)/g(n) = 1 > 0$

Wichtige Spezialfälle

- f Polynom mit Grad d, dann $f \in \Theta(n^d)$
- Jedes Polynom f erfüllt $f \in o(\exp(n))$ und $f \in \omega(\log(n))$
- Jede Funktion, die von einer Konstante $b > 0$ beschränkt ist, erfüllt $f \in O(1)$ (wobei 1 die konstante Funktion $n \mapsto 1$ ist).
- Basis des Logarithmus ist egal: $\log_a(n) \in \Theta(\log_b(n))$
- Für jede Konstante $c > 0$ gilt $O(c \cdot g) = O(g)$

Erste einfache Regeln

- $f \in O(g_1 + g_2)$ und $g_1 \in O(g_2) \Rightarrow f \in O(g_2)$
- $f_1 \in O(g_1)$, $f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(g_1 + g_2)$
- $f \in O(g_1)$, $g_1 \in O(g_2) \Rightarrow f \in O(g_2)$

Komplexitätsklasse P

Literatur: Hromkovič, (Sipser, Barak)

Komplexitätsklasse TIME(T(n))

Def Sei $T: \mathbb{N} \to \mathbb{N}$. Die Komplexitätsklasse **TIME(T(n))** ist die Menge aller Sprachen $L \subseteq \{0,1\}^*$, die in worst case Laufzeit $O(T(n))$ entschieden werden können (von einer deterministischen TM):

$$\text{TIME}(T(n)) = \{ L \subseteq \{0,1\}^* \mid \exists \text{ det. TM } M, \text{ die } L \text{ entscheidet und } \text{Time}_M(n) \in O(T(n)) \}$$

Tippo

Alternative Bezeichnung manchmal: **DTIME(T(n))** ($D \hat{=}$ deterministisch)

Komplexitätsklasse SPACE ($s(n)$)

Def Sei $s: \mathbb{N} \to \mathbb{N}$. Die Komplexitätsklasse $SPACE(s(n))$ ist die Menge aller Sprachen $L \subseteq \{0,1\}^*$, die in worst case mit Speicherplatzbedarf $O(s(n))$ entschieden werden können:

$$SPACE(s(n)) = \{ L \subseteq \{0,1\}^* \mid \exists \text{ det. TM } M, \text{ die } L \text{ entscheidet und } space_M(n) \in O(s(n)) \}.$$

Alternative Bezeichnung: $DSPACE(s(n))$

Komplexitätsklassen P und PSPACE

Def $P = \bigcup_{d \in \mathbb{N}} TIME(n^d)$ P ≙ polynomiell

d.h. P enthält alle Sprachen, die in polynomieller Laufzeit von einer deterministischen TM entschieden werden können.

Analog für Speicherplatz:

$PSPACE = \bigcup_{d \in \mathbb{N}} SPACE(n^d)$

Achtung: Entscheidungsprobleme!

Traditionell werden die Komplexitätsklassen für Entscheidungsprobleme über Sprachen definiert (ist ein Wort w in L oder nicht?), nicht über Funktionen.

In der Praxis löst man aber oft andere Probleme (Suchprobleme, Optimierungsprobleme, ...).

Mit einem Trick kann man manchmal Ergebnisse zwischen beiden Problemtypen hin und herschieben.

Beispiel:

Achtung: Entscheidungsproblem!

Optimierungsproblem: Finde den kürzesten Weg zwischen s und t

Entscheidungsproblem: Gibt es einen Weg zwischen s und t der Länge höchstens k^2.

Wir kommen später nochmal auf diesen Unterschied zurück.

Komplexität eines Algorithmus analysieren: Theorie vs "Praxis"

Betrachte das folgende Entscheidungsproblem:
Gegeben ein Graph $G = (V, E)$, ist er verbunden oder nicht?

Wollen nun die Laufzeit eines korrekten Algorithmus für dieses Problem analysieren, zum Beispiel Breitensuche.

Formal müssen wir nun folgende Dinge tun:

- Eingabe binär kodieren und deren Anzahl Bits zählen in n
- TM konstruieren, die den Algorithmus "implementiert" und schauen, wie viele Rechenschritte diese TM macht

Komplexität eines Algorithmus analysieren: Theorie vs "Praxis"

In der Praxis macht man es oft einfacher:

Länge der Eingabe: # Knoten + # Kanten

Rechenschritte des Algorithmus:

- Arithmetische Operationen kosten eine "Einheit" (auch wenn die Zahlen wenden so groß, dass wir deren Größe methodisch müssen) → wissen wir vorher, dass jede Zahl in, z.B., 100 Bits kodiert werden kann?
- Schreiben / Lesen einer Zahl aus dem Speicher kostet eine "Einheit".

Beispiele für Probleme in P

... haben Sie in der Algorithmenvorlesung gesehen!

- Sortieren von n Zahlen $\rightsquigarrow$ $O(n \log n)$
  ($\rightsquigarrow \Omega(n \log n)$ für vergleichsbasierte Algorithmen)

- Kürzeste Wege in Graphen $\rightsquigarrow$ Je nach Algorithmus, zum Beispiel $O(n^3)$ für Bellman-Ford, $O(n^2)$ für Dijkstra, ...

Komplexitätsklasse NP

Literatur: Hromkovič, (Sipser, Barak)

Motivation

Bisher haben wir nur deterministische TMs angeschaut.

Nun: nicht-deterministische TMs

Wiederholung:

- viele Rechenwege "gleichzeitig"
- wenn einer davon erfolgreich ist, nehmen wir den

Sieht deutlich "effizienter" aus...

Definition $\mathrm{TIME}_N(w)$

Sei N eine nicht-deterministische TM über dem Eingabealphabet $\{0,1\}$. Für ein Wort $w \in \{0,1\}^*$ ist die Zeitkomplexität

$\mathrm{TIME}_N(w)$

von N auf w die Länge der kürzesten akzeptierenden Berechnung, die w entscheidet (und 0, falls es keine akzeptierende Berechnung gibt).

- Notation wie bei deterministischen TMs, hängt ja noch von der TM N (bzw. M) ab.
- ⚠ kürzeste Berechnung, ⚠ akzeptierende Berechnung

Definition $TIME_N(n)$

Sei N eine nicht-det. TM über $\{0,1\}^*$. Dann ist

$$\underset{N}{TIME}(n) = \max\{ TIME_N(w) \mid |w| = n,\ w \in L(N) \}$$

Falls es kein w gibt, das Länge n hat und in L ist, setzen wir

$$\underset{N}{TIME}(n) = 0.$$



Definition hier nur über Wörter in $L(N)$, das ist anders als bei deterministischen TMs.

Obere Schranken

Sei $t : \mathbb{N} \to \mathbb{N}$ eine Funktion. Wir sagen, eine nicht-det TM N ist

schwach $t(n)$-beschränkt falls für alle $n \in \mathbb{N}$

$$\text{TIME}_N(n) \in O(t(n))$$

(⚠ Aussage nur über akzeptierende Berechnungen).

N ist stark-$t(n)$-beschränkt, wenn jede Berechnung auf w (nicht nur die akzeptierenden)

höchstens $t(|w|)$ Schritte macht.

Komplexitätsklasse NTIME(T(n))

Def Sei $T: \mathbb{N} \to \mathbb{N}$. Die Komplexitätsklasse NTIME(T(n)) ist die Menge aller Sprachen $L \subseteq \{0,1\}^*$, die in worst case Laufzeit $T(n)$ von einer nicht-deterministischen TM akzeptiert werden können:

Typo

$$NTIME(T(n)) = \{ L \in \{0,1\}^* \mid \exists \text{ nicht-det TM } N \text{ mit } L(N) = L \text{ und } Time_N(n) \in \mathcal{O}(T(n)) \}$$

$$= \{ L \in \{0,1\}^* \mid \exists \text{ nicht-det TM } N, \text{ die schwach-}t(n)\text{-beschränkt ist, mit } L(N) = L \}$$

Komplexitätsklasse NP

Def $NP = \bigcup_{d \in \mathbb{N}} NTIME(n^d)$

"nicht-det polynomielle Zeit"

⚠ NP steht NICHT für "nicht polynomiell" ...

(und das trifft es auch nicht richtig, denn es gibt Klassen jenseits von NP ...)

Beispiel: CLIQUE $\in$ NP

Clique problem: Given an undirected graph $G = (V, E)$ with n vertices and a parameter $k \leq n$. Decide whether G contains a clique of size k.

Proposition: Clique $\in$ NP

Proof (1) We construct a non-deterministic TM N:

- In the beginning, the NTM "randomly selects" k vertices of the graph: $v_1, \dots, v_k$
- Then it tests whether these vertices form a clique.

(2) We prove that the running time of the NTM is polynomial in n:

Theorem: Clique $\in$ NP

(Proof continued:)

- Within one path, the running time is $O(k^2)$ (checking the existence of k^2 edges)
- We now consider all possible computation paths of the TM. If one of them accepts, then we have found a clique. Then the running time of this algorithm is $O(k^2)$, and because we know that $k \leq n$, the running time is also $O(n^2)$, definitely polynomial in n.
- If none of the paths find a clique, the TM also stops after $O(n^2)$ steps with all paths rejecting.

Komplexitätsklasse NSpace (s(n))

Analog: $SPACE_N(w)$ = benötigter Speicherplatz derjenigen akzeptierenden Berechnung von N auf w, die am wenigsten Platz benötigt. (und ∞, falls es keine akzeptierende Berechnung gibt).

$$SPACE_N(n) = \max\{ SPACE_N(w) \mid |w| = n,\ w \in L(N)\}$$

$$NSPACE(s(n)) = \{ L \in \{0,1\}^* \mid \exists \text{ nicht-det TM } N \text{ mit } L(N) = L \text{ und } SPACE_N(n) \in O(s(n)) \}$$



NTMs gibt es nicht "in echt"

- Irgendwie klar, aber um es nochmal deutlich zu sagen:

  NTMs sind ein theoretisches Konstrukt.

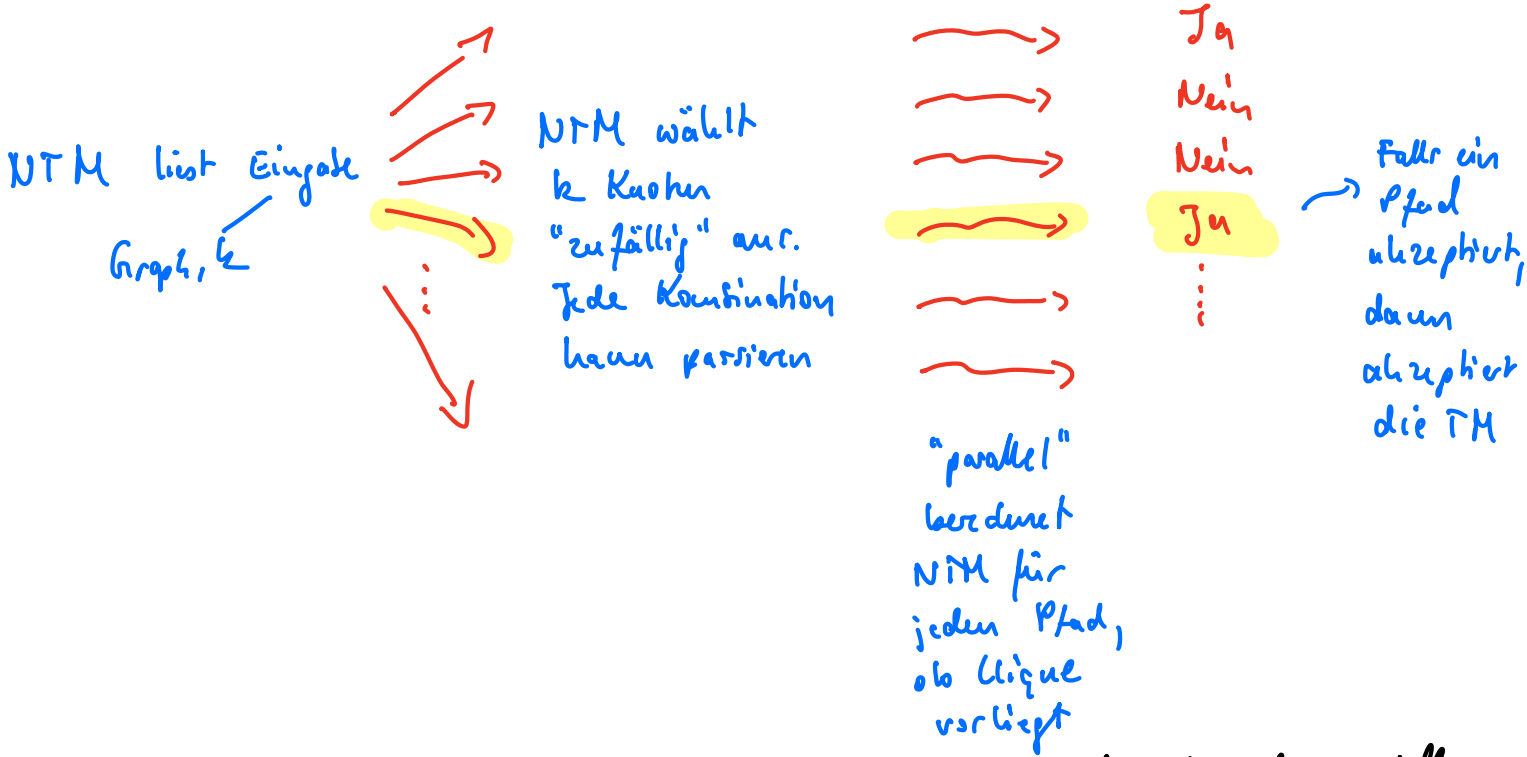
- Es ist erstaunlich hilfreich dabei, relevante "echte" Dinge zu charakterisieren (wie "schwer" Probleme sind).

- Aber natürlich kann man sie nicht bauen!!!

NP: Charakterisierung durch Verifikation

Literatur: Hromkovič, Sipser

Nochmal: CLIQUE und die NTM dafür



Laufzeit polynomiell, wenn es einen akzeptierenden Pfad mit polynomieller Laufzeit gibt.

Andere Sichtweise: Verifizierer

- NTM "rät" die richtige Lösung (die k Knoten der Clique).
  (Das ist der nichtdeterministische Teil).

- Dann gibt es einen deterministischen Algorithmus, der die geratene Lösung "verifiziert".
  (deterministisch, typischerweise mit kurzer Laufzeit)

NTM $\hat{=}$ Verifizierung

Haben am Clique-Beispiel gesehen, daß die Arbeitsweise der NTM so ähnlich wie ein Verifizierer sein kann.

Geht das immer?

Antwort Ja!

Das schauen wir jetzt genauer an.

p - Verifizierer, formal

Def Sei $p: \mathbb{N} \to \mathbb{N}$ eine Funktion.

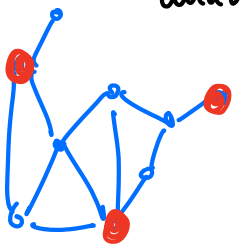
Eine deterministische TM M heißt p-Verifizierer für eine Sprache $L \subseteq \{0,1\}^*$, wenn folgendes gilt:

(1) Für alle $w \in L$ gibt es einen String $c \in \{0,1\}^*$ (certificate, witness, Zeuge) mit Länge $|c| \leq p(|w|)$ (Länge abhängig von der Länge von w) so daß M bei Eingabe von (w, c) nach höchstens $p(|w|)$ vielen Schritten akzeptiert.

(2) Falls $w \notin L$, dann gilt für alle $c \in \{0,1\}^*$, daß M bei Eingabe von (w, c) nach höchstens $p(|w|)$ Schritten stopt und dann verwirft.

p - Verifizierer, Beispiele

Independent set: Gegeben Graph G, $k \leq n$, gibt es eine unabhängige Knotenmenge der Größe k?



↳ keine Kanten zwischen irgendeinem der Knoten

Verifizierer kriegt eine Knotenmenge und muß nur noch testen, ob sie wirklich unabhängig ist. Ist nach $O(k^2)$ Schritten fertig damit (für jedes Paar von Knoten, schaue ob eine Kante existiert oder nicht)

p - Verifizierer, Beispiele

Traveling salesman: Gegeben gewichteter Graph, $c > 0$.

Gibt es eine Tour im Graphen durch alle Städte, die Länge $\leq c$ hat?

Verifizierer: Kriegt eine Tour, soll sagen ob die Länge $\leq c$ ist. Diese Verifikation braucht nur linear (in der Anzahl Knoten) viel Zeit.



Wieder betrachten wir nur Entscheidungsprobleme.

"NP (=) Polynomialzeit-Verifizierbar"

Satz $L \in NP$ $\Leftrightarrow$ es gibt einen polynomialzeit-Verifizierer für L.

P-Verifizierer, wobei $p: \mathbb{N} \to \mathbb{N}$ ein Polynom ist

Beweisidee "⇒"

"⇒" • Sei $L \in NP$, d.h. es gibt eine NTM N so daß $L(N) = L$ und $TIME_N(n) = p(n)$ für ein Polynom p.

- Wir können alle Berechnungspfade der NTM in einem Baum aufschreiben. OBdA sei der Baum binär. (denn max. Knotengrad beschränkt ... überlegen Sie warum)
- Ein Pfad durch den Baum kann durch einen binären String c beschrieben werden, der für jeden Knoten sagt, welche 'Abzweigung' der Pfad hier gewählt hat.
- Falls der Eingabewort $w \in L$ ist, gibt es einen Pfad der Länge höchstens $p(|w|)$, der in einem akzeptierenden Zustand endet.
- Sei nun c_N der String, der diesen Pfad beschreibt.

→

Beweisidee "⇒"

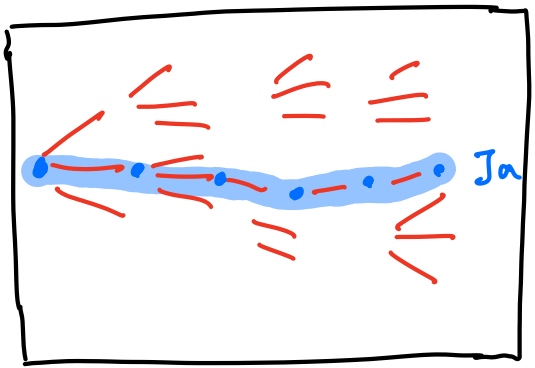
- Der Verifizierer V bekommt nun w und c_w als Eingabe. Er simuliert den Berechnungspfad, den N genommen hat. Die Berechnung von V ist deterministisch, da der String c_w immer sagt, wie man fortfahren muß.
- Die Berechnung von V auf w endet nach $p(|w|)$ vielen Schritten in einem akzeptierenden Zustand, falls $w \in L$.
- Falls $w \notin L$, dann gibt es keinen Pfad im Baum von N, der akzeptiert. Nach $p(|w|)$ Schritten darf der Verifier also abbrechen und gibt "nicht verifiziert" zurück. Das tut er, egal welches c man ihm mit eingegeben hat.

□

Beweisidee "⇒"

NTM für $w \in L$:

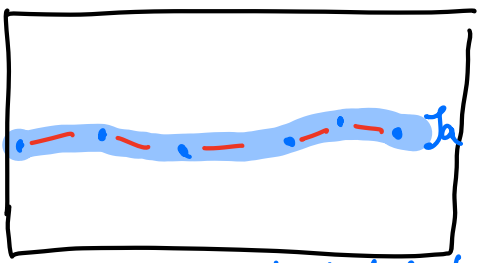
$w \rightsquigarrow$



String c beschreibt Weg durch den Baum

Verifier V:

$(w, c) \rightsquigarrow$

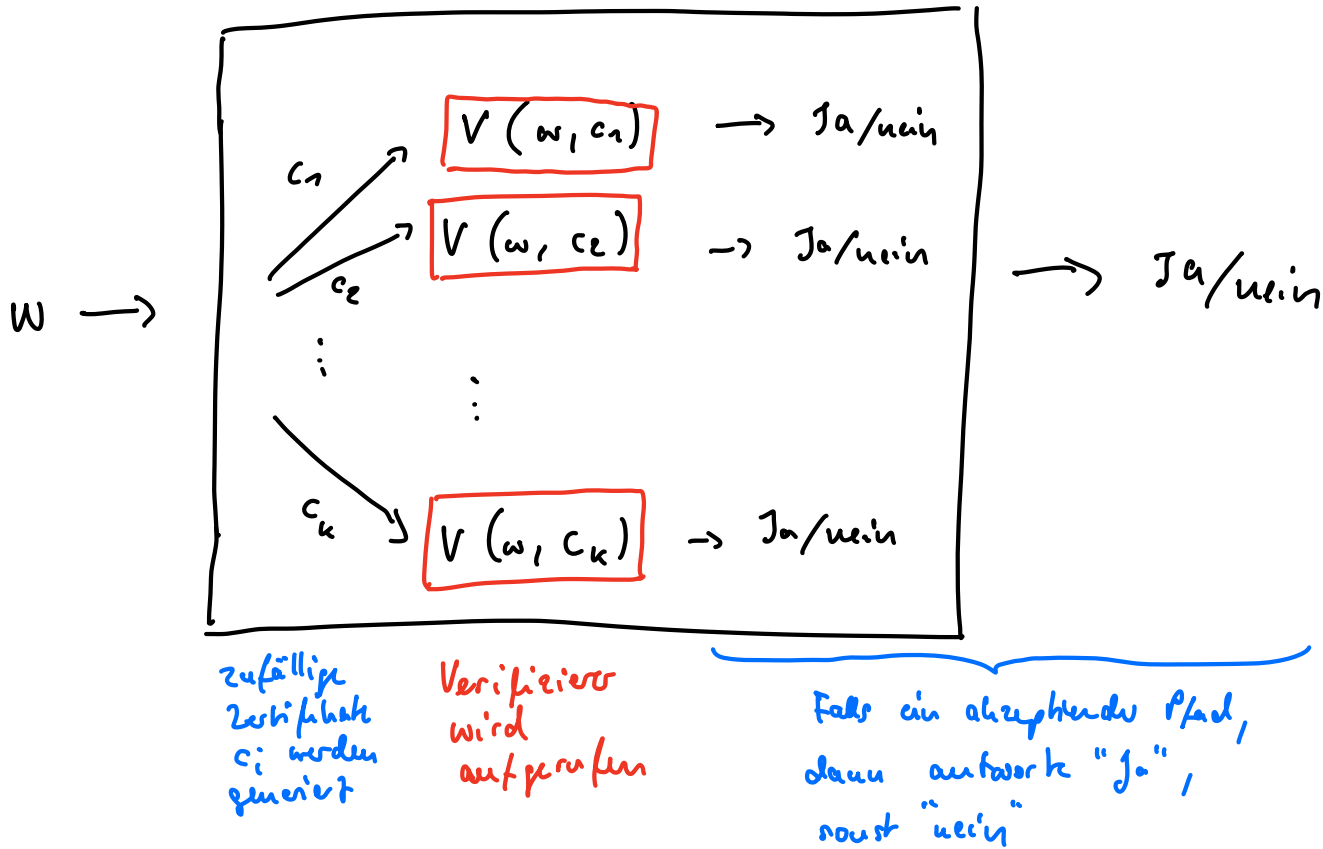


Gehe den deterministischen durch c beschriebenen Pfad aus dem Baum, überzeuge dich davon

Beweisidee "<="

- Sei V ein Verifier für L.
- Wir konstruieren eine NTM, die Eingabe w akzeptiert falls $w \in L$.
- Die NTM probiert hierzu einfach alle möglichen Strings c der Länge $p(|w|)$ mit Hilfe von V aus (nicht-deterministisch).
- Falls $w \in L$, dann gibt es laut Def. von L ein c, so dass V mit Eingabe w, c akzeptiert. Dann ist dann auch ein akzeptierender Berechnungspfad von N.
- Falls $w \notin L$, dann gibt es keinen akzeptierenden Pfad.

Beweisidee "<="


w
$V(w, c_1)$
$V(w, c_2)$
$V(w, c_k)$
c_1
c_2
c_k
Ja/nein
Ja/nein
Ja/nein
Ja/nein
zufällige Zertifikate c_i werden generiert
Verifizierer wird aufgerufen
Falls ein akzeptierender Pfad, dann antworte "Ja", sonst "nein"

Wie viel hilft der Nicht-Determinismus?

Literatur: Wegener Kap. 3.2
(Hromkovic)

Frage: beliebig großes speed-up?

Haben gesehen: Wenn wir eine NTM verwenden, erzielen wir im Vergleich zu einer kanonischen DTM oft einen exponentiellen Speedup.

Können wir durch Verwendung von nicht-deterministischen TMs beliebige speed-ups in Rechenzeit erzielen?

Die Antwort ist nein:

Nur exponentieller Speedup

Satz Für jede Sprache $L \in NP$ gibt es ein Polynom $p(n)$ und eine deterministische TM M_1, so daß M die Sprache L in Zeit $2^{p(n)}$ akzeptiert.

Beweisidee

- $L \in NP \Rightarrow$ es gibt NTM N und Polynom $q(n)$ so daß N die Sprache L in Zeit $q(n)$ erkennt.

- Betrachte den Berechnungsbaum von N. Der Verzweigungsgrad an jedem Knoten ist höchstens $d = 3 \cdot |Q| \cdot |\Gamma|$ (nächster Zustand, gelesener Buchstabe, Bewegung Schreibkopf)

und wir betrachten alle Pfade bis Länge $q(n)$. Es gibt also höchstens $d^{q(n)}$ viele Pfade.

- Diese können wir nun auch alle hintereinander ausführen, was dann Zeit $\underbrace{d^{q(n)}}_{\text{\# Pfade}} \cdot \underbrace{q(n)}_{\text{Länge jedes Pfades}}$ verbraucht.

- Basiswechsel + ausdrücken als $2^{p(n)}$: tut's. □

NP ⊂ DEXPTIME

Definition: Die Komplexitätsklasse DEXPTIME enthält diejenigen Sprachen, die durch eine deterministische TM in exponentieller Laufzeit entschieden werden können:

$$\text{DEXPTIME} := \bigcup_{d \in \mathbb{N}} \text{TIME}(2^{n^d})$$

Haben also gerade bewiesen:

$$\text{NP} \subset \text{DEXPTIME}.$$

Nächste Fragen

- Gilt $\subsetneq$ oder $\subseteq$? ⇝ jetzt
- Gibt es Probleme "jenseits von DEXPTIME"? ⇝ später
- Und wie ist das Verhältnis zwischen P und NP? ⇝ jetzt

$$P \overset{?}{=} NP$$

Wie verhält sich P zu NP?

- Offensichtlich gilt $P \subseteq NP$.
- Haben gesehen: $NP \subseteq EXP$
- Aber gilt nun auch $P \subsetneq NP$?

$P \stackrel{?}{\neq} NP$

Größte Frage der Informatik: gilt $P \stackrel{?}{\neq} NP$?

D.h. gibt es Probleme, die wirklich so schwer sind, daß kein Polynomieller Algorithmus sie lösen kann?

Antwort bis heute unbekannt!!!

Was ist die Evidenz?

Evidenz für $P \neq NP$

Es gibt sehr viele Probleme in NP, für die bisher kein Polynomzeit-Algorithmus bekannt ist.

Und viele Leute arbeiten seit 50 Jahren daran, das Problem zu lösen ...

Die meisten Leute glauben, daß also $P \neq NP$ gilt als $P = NP$.

Möglichkeit: $P \neq NP$ unabhängig von Axiomen

Es ist denkbar, daß die Frage, ob $P \neq NP$ gilt oder nicht, unabhängig von den üblichen Axiomen der Mathematik ist (so wie die Kontinuumshypothese).

Was würde das für die Praxis bedeuten? Unklar ... zumindest mir.

Falls doch $P = NP$ gilt?

Wenn man das konstruktiv beweisen könnte, dann hätte man auf einen Schlag effiziente Algorithmen für ganz viele Probleme, die bisher in der Praxis unlösbar waren:

- Kryptographie bricht zusammen.
- Quantencomputer brauchen wir dann auch nicht mehr wirklich...
- Beweise zu finden ist dann so einfach wie Beweise zu verifizieren.

Es würde allerdings alles am Grad des Polynoms hängen. Denn Algorithmen der Ordnung n^{100} sind immer noch nicht praktisch...

1 Million Dollar

Das $P \neq NP$ - Problem wurde in die Liste der "Millenium" - Probleme der Mathematik aufgenommen.

Falls sie es lösen, kriegen sie 1 Mio Dollar...

Neuere Literatur zu $P \neq NP$

- Richard Lipton's blog, auch als Buch:

  Richard Lipton: The $P = NP$ question and Gödel's lost letter. Springer

- Aaronson: $P \stackrel{?}{\neq} NP$, Open problems in mathematics, 2016

- Siehe auch die ausführliche Diskussion in Kap. 16 von Papadimitriou's Buch.

Boolesche Formeln und SAT

Literatur: Hromkovič, Sipser
Arora/Barak Kap. 2

Boolesche Formel

Eine **boolesche Formel** (logische Formel) besteht aus booleschen Variablen, die durch die Operatoren $\wedge$, $\vee$, $\neg$ verknüpft werden.

true/false

$\wedge$: Konjunktion, logisches "und"

$\vee$: Disjunktion, logisches "oder"

$\neg$: Negation, Verneinung

Beispiel: $x_1 \vee \neg x_2 \wedge x_3 \vee (x_5 \wedge x_6 \wedge \neg x_p)$

"Literale": Variable oder negierte Variable

Schreibweise Negation: $\neg x_i$ oder $\overline{x_i}$

Erfüllbare Formel

Eine boolesche Formel ist **erfüllbar**, wenn es eine Belegung der Variablen mit den Werten "wahr" oder "falsch" gibt, so daß die ganze Formel wahr wird.

Beispiele:

- $x_1 \wedge x_2 \wedge \neg x_3$ ist erfüllbar mit
  $x_1 = \text{true}$, $x_2 = \text{true}$, $x_3 = \text{false}$, denn
  $\text{true} \wedge \text{true} \wedge \neg(\text{false}) \rightsquigarrow \text{true}$

- $x_1 \wedge \neg x_1$ ist nicht erfüllbar

Konjunktive Normalform (CNF)

- Eine "Klausel" ist eine Formel, die nur aus Disjunktionen zwischen Literalen besteht:

  Bsp: • $x_1 \vee \neg x_2 \vee x_3 \vee x_4 \vee \neg x_5$ ist eine Klausel

  • $\neg(x_1 \vee x_2)$ ist keine Klausel

- Eine logische Formel ist in konjunktiver Normalform ("CNF"), wenn sie die Form

$$\text{Klausel}_1 \wedge \text{Klausel}_2 \wedge \ldots \wedge \text{Klausel}_k$$ hat.

Beispiel:

"und" zwischen den Klammern

"oder" in den Klammern

$$(x_1 \vee \neg x_2) \wedge (\neg x_3 \vee x_4 \vee x_6) \wedge (\neg x_1 \vee \neg x_2)$$

Boolesche Funktion als CNF

Jede beliebige Boolesche Funktion kann durch eine CNF-Formel ausgedrückt werden (mit exponentieller Länge).

Beh: Sei $f: \{0,1\}^\ell \to \{0,1\}$ eine beliebige Funktion.
Dann gibt es eine boolesche Formel φ in CNF mit ℓ Variablen und Länge $\leq \ell \cdot 2^\ell$ so daß für alle $u \in \{0,1\}^\ell$ gilt: $\varphi(u) = f(u)$.

Die Länge einer booleschen Formel steht hier für die Anzahl der $\vee/\wedge$-Zeichen in der Formel.

Boolesche Funktion als CNF: Beweis

Beweis

Schritt 1 Betrachte ein Wort $v \in \{0,1\}^\ell$. $z_1, \dots, z_\ell$ Variablen

Wir konstruieren zunächst eine Klausel c_v mit ℓ Variablen so daß gilt:

$$c_v(u) = \begin{cases} 1 & \text{falls } u \neq v \\ 0 & \text{falls } u = v \end{cases}$$

Das ist einfach: für $v = (v_1, \dots, v_\ell)$ wählen wir die Klausel

$$c_v = (a_1 \vee a_2 \vee \dots \vee a_\ell) \text{ mit}$$

$$a_i = \begin{cases} z_i & \text{falls } v_i = 0 \\ \overline{z_i} & \text{falls } v_i = 1 \end{cases}$$

Beispiel:

$v = (1,1,0,0)$, wähle $c_v = \overline{z_1} \vee \overline{z_2} \vee z_3 \vee z_4$

Dann $c_v(u) = 0 \iff u = v$

Boolesche Funktion als CNF: Beweis



Schritt 2

Sei $V := \{v \in \{0,1\}^\ell \mid f(v) = 0\}$

Definiere nun die Formel $\varphi = \bigwedge_{v \in V} c_v$.

Notation:

$\bigwedge_{i=1\dots n} q_i = q_1 \wedge q_2 \wedge \dots \wedge q_n$

Nun gilt:

- φ ist in CNF
- $\varphi(z) = 0 \iff f(z) = 0$
- Die Länge von φ ist $\leq \ell \cdot 2^\ell$ denn
  - Jedes c_v hat Länge $= \ell$
  - $|V| \leq 2^\ell$

∎

SAT

Satisfyability Problem (SAT)

Gegeben eine boolesche Formel in konjunktiver Normalform.

Frage: Ist die Formel erfüllbar, d.h. gibt es eine Belegung der Variablen, so daß die Formel wahr wird?

- Wird uns noch oft begegnen
- Ist intuitiv "schwer"
- Sehr relevant in der Praxis!

SAT

Beh: $SAT \in NP$

Beweisskizze Sei eine SAT-Formel mit n Literalen über $m \leq n$ Variablen gegeben. Sei eine Belegung der m Variablen gegeben. Dann können wir ganz einfach in polynomieller Zeit verifizieren, ob diese Belegung die Formel erfüllt oder nicht. Also ist $SAT \in NP$.

3 SAT, k-SAT

Es gibt viele Varianten von SAT, zum Beispiel:

k-SAT: wie SAT, nur daß die Klauseln höchstens k Literale enthalten dürfen.

3 SAT: der Spezialfall von $k = 3$, wird später wichtig.

Bsp: $(x_1 \vee \neg x_2 \vee x_4) \wedge (x_5 \vee x_1 \vee x_3)$

Offensichtlich ist k-SAT $\in$ NP, für alle Werte k.

Schwierige (?) Probleme in NP

Motivation

Wollen ein Gefühl dafür entwickeln, was für Entscheidungsprobleme sich so in NP tummeln.

Klar: $P \subseteq NP$, also sind die "einfachen" Probleme in NP.

Wie sieht "das schwierige Ende" aus?

SAT

Satisfyability Problem (SAT)

Gegeben eine Formel in konjunktiver Normalform.

Frage: Ist die Formel erfüllbar, d.h. gibt es eine Belegung der Variablen, so daß die Formel wahr wird?

3 SAT : Wie SAT, nur hat jede Klausel in der Formel höchstens drei Literale.

Max cut / Min cut

Betrachte Graphen $G = (V, E)$. Ein cut im Graphen ist eine Zerlegung der Knoten in zwei Mengen: $V = V_1 \dot\cup V_2$.

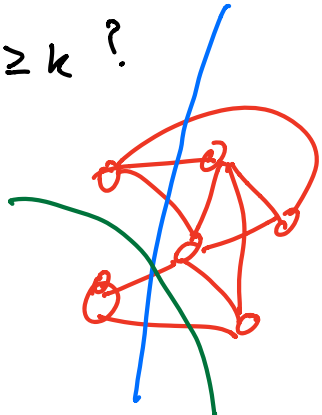
Der Wert eines cuts ist die Anzahl der Kanten zwischen V_1 und V_2.

Mincut: gibt es einen cut im Graphen mit Wert $\leq k$?

$\rightsquigarrow$ einfach

Max cut: gibt es einen cut mit Wert cut $\geq k$?

$\rightsquigarrow$ schwierig



Beide sind offensichtlich in NP (warum?)

Integer programs

Betrachte ein System von linearen Gleichungen der Form

$$a_{11} x_1 + a_{12} x_2 + \dots + a_{1n} x_n = b_1$$

$$\vdots$$

$$a_{m1} x_1 + \dots + a_{mn} x_n = b_m$$

Hierbei sind a_{ij}, b_i gegeben. Gesucht sind $x_1, \dots, x_n \in \mathbb{Z}$.

Falls die Lösungen $x_1, \dots, x_n \in \mathbb{Z}$ sein sollen, dann heißt das Problem ein **Linear Integer Problem** und ist schwer.

Quadratic 0-1- equations

Nun erlauben wir auch gemischte Terme der Form

$$a_{ijk}\, x_j\, x_k \quad .$$

Nehmen wir an, alle Variablen sollen 0 oder 1 sein.

Systeme von solchen quadratischen Gleichungen sind schwer zu lösen.

Offensichtlich in NP.

Vertex Cover

Eine Knotenüberdeckung eines Graphen $G = (V, E)$ ist eine Teilmenge $V' \subseteq V$ von Knoten, so daß jede Kante $e \in E$ mindestens einen Endknoten in V' hat.

Vertex cover problem: gegeben Graph $G = (V, E)$, $k \in \mathbb{N}$.
Gibt es ein Vertex cover der Größe $\leq k$?

offensichtlich ist vertex cover in NP (warum?)

Subset Sum Problem

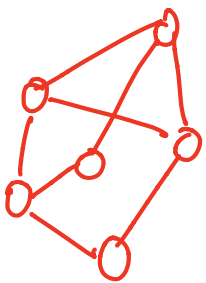
Gegeben n natürliche Zahlen $x_1, \dots, x_n \in \mathbb{N}$ und $b \in \mathbb{N}$.

Gibt es eine Indexmenge $I \subseteq \{1, \dots, n\}$ so daß

$$\sum_{i \in I} x_i = b \quad ?$$

Hamiltonian Cycle

Gegeben ein ungerichteter Graph $G = (V, E)$, gibt es einen Zyklus im Graphen, der jeden Knoten genau ein mal besucht und dann zum Startpunkt zurück kehrt?



Traveling salesman problem, TSP

Gegeben ein ungerichteter Graph, was ist die Länge des kürzesten Hamiltonschen Cycles in G?

Entscheidungsproblem: Gibt es in G einen Zyklus der Länge $\leq k$?

Leichte Probleme in NP

Da $P \subset NP$ liegt, liegen natürlich auch viele leichte Probleme in NP. Fast alles, was Sie in der Algorithmenvorlesung gesehen haben.

Polynomzeit-Reduktionen

Literatur: Hromkovič 6.1., Sipser 7.4

Motivation

Wie in der Berechenbarkeitstheorie auch wollen wir Probleme aufeinander reduzieren, um deren "Schwierigkeit" vergleichen zu können.

Wie dort auch muß die Reduktion selbst "einfach" sein. In diesem Kontext: in polynomieller Zeit berechenbar.

Polynomielle Reduktion

Def Seien $L_1 \subseteq \Sigma_1^*$, $L_2 \subseteq \Sigma_2^*$ zwei Sprachen.

L_1 ist auf L_2 polynomiell reduzierbar, $L_1 \leq_p L_2$,
falls es eine TM M gibt, die in polynomieller Zeit
eine Abbildung $f: \Sigma_1^* \to \Sigma_2^*$ berechnet,
so daß gilt:

$$w \in L_1 \iff f(w) \in L_2.$$

Auf den folgenden Folien lassen wir den Index "p" in "$\leq_p$" oft
weg und schreiben einfach "$\leq$".

Intuition

Sehr ähnlich wie in der Berechenbarkeitstheorie, nur daß wir nun zusätzlich fordern, daß die Reduktion in polynomieller Zeit passieren muß.

Clique $\leq_p$ Vertex cover

... haben wir schon früher konstruiert.

Aber Sie müssen sich noch überlegen, warum die Reduktion in polynomieller Zeit funktioniert. Übung!

3 SAT $\leq$ Clique

Gegeben: Formel der Form

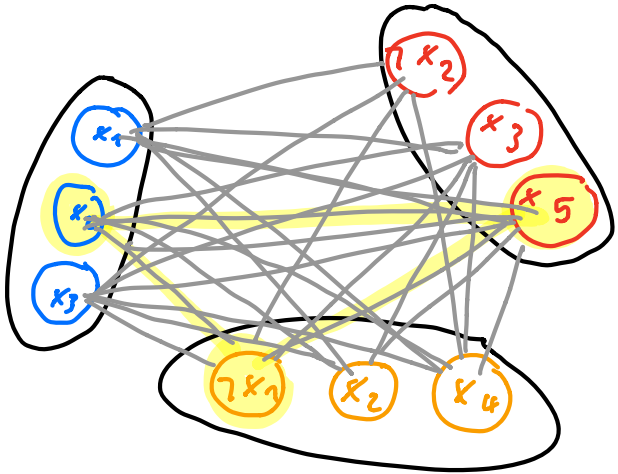
$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \dots \wedge (a_k \vee b_k \vee c_k)$$

mit k Klauseln und höchstens $n = 3 \cdot k$ Literalen und Variablen.

Wollen daraus einen Graphen konstruieren, auf dem wir dann das k-Clique-Problem lösen wollen. Das funktioniert wie folgt:

3 SAT ≤ Clique

$$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee x_3 \vee x_5) \wedge (\neg x_1 \vee x_2 \vee x_4)$$



- Jede Dreiergruppe von Literalen entspricht einer Dreiergruppe von Knoten
- Kanten:
  - nicht innerhalb der Gruppen
  - nicht zwischen entgegengesetzten Literalen z.B. x_1 und $\neg x_1$
  - sonst überall

Damit die Formel wahr wird, muß in jeder Knotengruppe mind. ein Literal wahr sein.

Knoten sind verbunden, wenn sie gleichzeitig wahr sein können.

3SAT ≤ Clique

Behauptung: Diese Konstruktion stellt eine Polynomzeit-Reduktion zwischen 3SAT und k-Clique dar:

Beweis **(1) Polynomzeit:**

Offensichtlich:
- Ursprüngliche Formel mit Länge n
- Graph hat n Knoten und $\leq n^2$ Kanten
- Konstruktion einer Kante benötigt nur das Wissen über die beiden Endknoten und die Klausel, in der beide sind.

$\rightsquigarrow O(n^2)$

3 SAT ≤ Clique

(2) Formel erfüllbar $\Leftrightarrow$ es gibt Clique der Größe k

"$\Rightarrow$" Sei eine Variablenzuweisung gegeben, die Φ erfüllt. In jeder Klausel muss dann mind. ein Literal wahr sein. Nun markieren wir dieses Literal in jeder Klausel (falls es mehrere Literale gibt, nehmen wir eine davon). Diese Knotenmenge stellt eine k-Clique dar:

- k Knoten
- keine widersprüchlichen Literale (sonst könnte Formel nicht wahr sein), also alle durch Kanten verbunden.

"$\Leftarrow$" analog. ∎

SAT $\leq$ 3SAT

Sei eine Formel Φ in CNF gegeben (eine Instanz von SAT). Wollen die Formel so umschreiben, daß eine Instanz von 3SAT daraus wird.

Sei eine Klausel aus Φ mit Länge > 3 gegeben, sagen wir

$$c = l_1 \vee \dots \vee l_k$$

Wir führen einige neue Variablen ein: $y_1, \dots, y_{k-3}$

Dann ersetzen wir c durch folgende Teilformel:

$$\psi_c := (l_1 \vee l_2 \vee y_1) \wedge (\bar{y}_1 \vee l_3 \vee y_2) \wedge \dots \wedge (\overline{y_{k-4}} \vee l_{k-2} \vee y_{k-3}) \wedge (\overline{y_{k-3}} \vee l_{k-1} \vee l_k)$$

Das machen wir für alle Klauseln aus Φ, die zu lang sind. Dabei kommt eine 3SAT-Instanz heraus, nennen wir sie Ψ.

index typo

SAT $\leq$ 3SAT

Beh Das ist eine poly. Reduktion von SAT auf 3SAT.

(0) Es kommt eine gültige 3SAT-Instanz raus: ✓

(1) Polynomzeit: klar, $O(n)$

(2) Φ erfüllbar $\Leftrightarrow$ Ψ erfüllbar

"$\Rightarrow$" Sei Φ erfüllbar, d.h. es gibt eine Belegung, die alle Klauseln wahr machen. Insb. auch für die Klausel c. Es muss also mind. ein Literal in c wahr sein, sagen wir, l_i ist wahr.

Nun setzen wir $y_1, \dots, y_{i-2}$ auf "wahr" und $y_i, y_{i+1}, \dots, y_{k-3}$ auf "falsch".

Dann ist die Teilformel Ψ_c wahr:

index typo

index typo

wahr wg. (*) wahr wg. (#) wahr wg. (**)

$$\psi_C := (\ell_1 \vee \ell_2 \vee y_1) \wedge (\bar{y}_1 \vee \ell_3 \vee y_2) \wedge \dots \wedge (\bar{y}_{k-4} \vee \ell_{k-1} \vee y_{k-3}) \wedge (\bar{y}_{k-3} \vee \ell_{k-1} \vee \ell_k)$$

(#) Sei ℓ_3 wahr, d.h. $i = 3$.

(*) Dann setzen wir y_1 = wahr, dadurch erhält jede Klausel vor dieser Klausel eine wahre Variable, ist also erfüllt.

(**) Dann setzen wir $y_2 = \dots = y_{k-1}$ auf falsch, also

$\bar{y}_2 = \dots = \bar{y}_{k-3}$ = wahr.

Dadurch erhält jede Klausel hinten eine wahre Variable

Dieselbe Konstruktion machen wir für alle anderen Klauseln in ϕ.

Tut's.

(2) „⇐"

OBdA betrachten wir nur ψ_c.

Sei ψ_c erfüllbar, betrachte eine erfüllende Belegung der Variablen. In jeder der Klauseln von ψ_c muß also mind. ein Literal wahr sein:

$$\psi_c := (l_1 \vee l_2 \vee y_1) \wedge (\bar{y}_1 \vee l_3 \vee y_2) \wedge \dots \wedge (\bar{y}_{k-4} \vee l_{k-1} \vee y_{k-3}) \wedge (\bar{y}_{k-3} \vee l_{k-1} \vee l_k)$$

Müssen zeigen: dafür ist mind. eines der l_i - Literale wahr.

Das kann aber nicht anders sein: ψ_c besteht aus $k-2$ vielen Klauseln, aber es gibt nur $k-3$ viele y - Variablen. Also muß in mind. einer der Klauseln ein l_i wahr sein.

Aber dann ist auch c wahr. ☺

□

3SAT $\leq$ SAT

Offensichtlich gilt auch die andere Richtung: 3SAT $\leq$ SAT.

Als Reduktion können wir einfach die Identitätsabbildung nehmen. Denn jede 3SAT-Instanz ist auch eine SAT-Instanz. ☺

$SAT \overset{\leq}{\geq} k\text{-}SAT$ für $k \geq 3$

Mit analogen Beweisen wie eben kann man zeigen, daß die Aussage $SAT \leq k\text{-}SAT$ auch für alle $k \geq 3$ gilt.

Für $k = 2$ bricht der Beweis allerdings zusammen, sehen Sie warum ?!?.

Reduktionen sind transitiv

Satz Polynomzeit-Reduktionen sind transitiv:

$$A \leq_p B, \quad B \leq_p C \Rightarrow A \leq_p C$$

Beweis Übung

Korollar: $SAT \leq$ vertex cover

denn haben schon gesehen: $SAT \leq 3SAT$ und

$3SAT \leq$ Clique und Clique $\leq$ vertex cover.

Reduktion auf einfaches Problem

Beh Sei $L_1 \leq_p L_2$ und $L_2 \in P$. Dann ist auch $L_1 \in P$.

Beweis Klar, oder ?

2-SAT $\in$ P

Erstaunlicherweise kann man durch eine Reduktion auf ein einfaches Graphenproblem zeigen, daß 2-SAT $\in$ P gilt...

→ Übungsblatt

Leichte Probleme

Reduktionen bieten also wieder eine Methode, mit der man beweisen kann, daß bestimmte Probleme "leicht" sind.

Hier: "leicht" = "$\in P$"

Die Strategie ist: Um zu zeigen, daß ein Problem L_1 "leicht", d.h. $\in P$, ist, muß ich es auf ein anderes Problem L_2 reduzieren, von dem ich schon weiß, daß es in P ist:

$$L_1 \leq L_2$$

Was heißt "schwierig"?

Wie könnten wir nun "schwierig" definieren?

Erster Versuch: schwierig $\hat{=}$ "nicht in polynomieller Zeit lösbar"
$\hat{=}$ $\notin P$

... wird so nicht klappen, warum?..?

$\rightsquigarrow$ nächste Vorlesung: NP - schwere Probleme

NP-vollständige Probleme

Literatur: Hromkovič 6.6., Sipser 7.4

Motivation

Wollen Probleme / Sprachen einteilen in leicht und schwer berechenbar.

- "Leicht" $\hat{=}$ in polynomieller Zeit lösbar
- "Schwer": am liebsten würden wir sagen "nicht in poly. Zeit lösbar". Aber das hilft nicht weiter, weil wir nicht wissen, ob $P \neq NP$ gilt.

Elegante Lösung: NP-Vollständigkeit.

Wollen sagen können, daß ein Problem zu den schwersten Problemen in NP gehört.

NP- vollständige Sprachen

Def Eine Sprache $L \subseteq \{0,1\}^*$ heißt **NP-schwer / NP-hart**, falls für alle Sprachen $L' \in NP$ gilt daß $L' \leq_p L$.

Intuition: Wenn wir L lösen können, dann können wir damit auch jedes andere Problem L' aus NP lösen.

Eine Sprache L heißt **NP-vollständig**, falls gilt:

(1) $L \in NP$

(2) L ist NP-schwer.

Die Menge aller NP-vollständigen Sprachen wird mit **NPC** ("NP complete") bezeichnet.

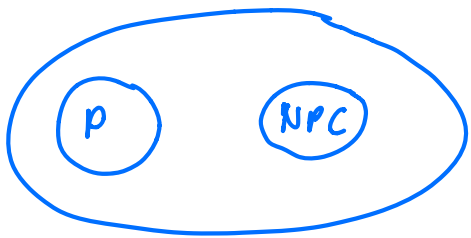
NP-vollständige Sprachen

Anschaulich: wenn ein Problem in NPC ist, dann gehört es zu den schwersten Problemen in NP.

Verhältnis zw. P und NPC : Fall $P \neq NP$

Satz Falls $P \neq NP$, dann gilt $P \cap NPC = \emptyset$.

Beweis Offensichtlich durch Kontrapositions: Falls $P \cap NPC \neq \emptyset$, dann gibt es ein $L \in P \cap NPC$, das wir in poly. Zeit berechnen können. Da $L \in NPC$, können wir dann auch jedes andere Problem $L' \in NP$ in poly Zeit berechnen, also $P = NP$.



Falls $P \neq NP$

Die Probleme in NPC sind dann echt schwerer als die in P.

Verhältnis zw. P und NPC : Fall $P \neq NP$

Eine interessante Frage ist dann: gibt es Probleme "zwischen" P und NP?

Falls $P \neq NP$, ist die Antwort "ja". ("Ladner's theorem")

d.h. $\exists L : L \notin P, L \notin NPC$



es könnte Reduktionen geben von

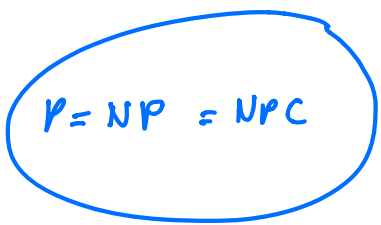
$L \leq L_2$

aber nicht $L_2 \leq L$

Verhältnis zw. P und NPC : Fall P = NP

Satz Falls es eine Sprache L gibt, so daß $L \in NPC$ und $L \in P$, dann gilt $P = NP$.

Beweis folgt direkt aus der Definition von NPC. Wir können dann jedes Problem $L' \in NP$ auf L reduzieren und so in poly. Zeit lösen.



Falls $P = NP$

Alle Probleme in NP sind gleich leicht (poly. Zeit).

"Erste" Sprache $L \in NPC$?

Wollen rausfinden, ob NPC überhaupt eine nicht-triviale Klasse ist: gibt es ein L, das beweisbar in NPC liegt?

Sieht echt schwierig aus: müssen beweisen, daß sich <u>jedes</u> Problem $L' \in NP$ auf L reduzieren läßt. Wie kann das gehen?

Weitere Sprachen $L' \in NPC$

Angenommen wir hätten ein $L \in NPC$. Dann können wir leicht andere Sprachen in NPC finden, durch Reduktion:

Beh $L_1 \in NPC$ und $L_1 \leq_p L_2$
$\Rightarrow L_2 \in NP$-hart

Bew Einfach, folgt direkt aus den Definitionen:
$L_1 \in NPC \Rightarrow$ können jedes $L \in NP$ reduzieren: $L \leq_p L_1$
Wg. Transitivität von "$\leq_p$" und $L_1 \leq L_2$ dann auch $L \leq_p L_2$. Also L_2 NP-hart. ∎

Um $L_2 \in NPC$ zu zeigen, fehlt dann nur noch $L_2 \in NP$, was oft leicht zu sehen ist.

Satz von Cook & Levin

Satz $SAT \in NPC$.

1971 bewiesen, unabhängig von Cook (Kanada) und Levin (UdSSR). Super-Cool!!!

Wie können wir den Satz beweisen?!?

Müssen zeigen: $L \in NP \Rightarrow$ es gibt Reduktion $L \leq SAT$.

Können nur verwenden: $L \in NP$, d.h. es gibt eine TM M, die L verifiziert, d.h. für jedes Wort $w \in L$ gibt es ein Zertifikat c so daß M die Eingabe $\langle w, c \rangle$ akzeptiert (und für $w \notin L$ akzeptiert sie nie).

Müssen also irgendwie die Berechnung einer TM durch eine boolesche Formel beschreiben...

$\rightarrow$ nächste Vorlesung.

Beweis des Satzes von Cook & Levin

Literatur: Schöning (& Arora/Barak Kap. 2)
(Hromkovič, Sipser)

Satz von Cook & Levin

Satz $SAT \in NPC$.

Beweisstrategie: erste Idee

Wollen folgendes erreichen:

Sei $L \in NP$. Wollen zeigen: $L \leq_p SAT$, d.h. es gibt Abbildung $f_L: \{0,1\}^* \to$ CNF-Formeln so daß
- $f_L(w)$ erfüllbar $\Leftrightarrow$ $w \in L$
- f_L polynomiell berechenbar

Wissen: $L \in NP$, d.h. es gibt TM M (den Verifier) so daß

$$w \in L \Leftrightarrow \exists u : M(w,u) = 1$$

wobei $|u| \leq p(|w|)$ polynomiell ist.

Was müssen wir beweisen?

(1) $SAT \in NP$, haben wir schon gesehen, offensichtlich.

(2) Sei L eine beliebige Sprache in NP.
Dann gibt es eine polynomielle Reduktion

$$L \leq_p SAT.$$

Müssen also so eine Reduktion konstruieren.
Sieht echt schwierig aus!

Beweisidee, erster Versuch

Betrachte $L \in NP$, sei M der zugehörige Verifier.

Wollen Reduktion:

Wort w	$\longmapsto$	CNF-Formel φ_w
$w \in L$, d.h.	$\Leftrightarrow$	Formel erfüllbar, d.h.
$\exists u : M(w,u) = 1$	$\Leftrightarrow$	$\exists \tilde{u} : \varphi_w(\tilde{u}) = 1$

$u \in \{0,1\}^{p(n)}$, p Polynom

Für festes w betrachte Funktion $g_{M,w} : \{0,1\}^{p(n)} \to \{0,1\}$,

$g_{M,w} : u \mapsto M(u,w)$

u: Zertifikat; $M(u,w)$: Ergebnis der TM

$\varphi_w =$ — Nun baue die entsprechende CNF-Formel, die genau dann wahr ist, wenn $M(u,w) = 1$ gilt (wie in voriger Vorlesung).

Beweisidee, erster Versuch

Damit hätten wir eigentlich erreicht, was wir wollen:

$w \longmapsto \varphi_w$ so daß $w \in L \iff \varphi_w$ erfüllbar

Wo ist der Haken?

Antwort: Die CNF φ_w kann exponentiell in $p(n)$ werden, ist also nicht in polynomieller Zeit berechenbar.

Mist. Aber die Idee war schon mal gut!

Beweisidee, zweiter Versuch

Wir wollen vorgehen wie im ersten Versuch, aber etwas vorsichtiger: wollen sicherstellen, daß φ_w nicht zu groß werden kann.

Wollen eine CNF konstruieren, die bei Eingabe eines $w \in L$ die akzeptierende Berechnung $M(w,c)$ des Verifizierers nachvollzieht.

Wir machen das jetzt "zu Fuß", und nur als Skizze:

Wir beschreiben die Konfiguration der TM M während der Berechnung durch folgende boolesche Variablen:

$\text{zust}_{t,z}$ $\quad (t = 0, \dots, p(n),\ z \in Q)$

$\text{zust}_{t,z} = 1$ falls M sich nach t Schritten im Zustand z befindet

$\text{pos}_{t,i}$ $\quad (t = 0, \dots, p(n),\ i = -p(n), \dots, +p(n))$

$\text{pos}_{t,i} = 1$ falls sich M nach t Schritten an Position i auf dem Band befindet

$\text{band}_{t,i,a}$ $\quad (t = 0, \dots, p(n),\ i = -p(n), \dots, p(n),\ a \in \Gamma)$

$\text{band}_{t,i,a} = 1$ falls Inhalt des Bandes an Position i zur Zeit t der Buchstabe a ist.

Wir führen eine Hilfsfunktion ein:

$G(x_1, \dots, x_m) = 1$:$\Leftrightarrow$ für genau einen Index $i \in \{1, \dots m\}$ ist $x_i = 1$, für alle anderen $= 0$.

Dies kann mit einer CNF-Formel mit $O(m^2)$ vielen Literalen ausgedrückt werden:

$$G = \underbrace{\left(\bigvee_{i=1}^{m} x_i \right)}_{\text{mind. ein } x_i \text{ ist wahr}} \wedge \underbrace{\left(\bigwedge_{j=1}^{m-1} \bigwedge_{l=j+1}^{m} \neg (x_j \wedge x_l) \right)}_{\text{höchstens ein } x_i \text{ ist wahr (keine zwei gemeinsam)}}$$

Durch "Ausmultiplizieren" erhalten wir eine CNF mit $O(m^2)$ vielen Literalen.

Überprüfung der Anfangsbedingungen:

Zum Zeitpunkt 0 muß sich der Kopf am Anfang des Bandes befinden, der Zustand muß q_0 sein, und das Wort w muß am Anfang des Bandes stehen.

Das können wir durch eine Formel ausdrücken, die genau dann erfüllbar ist, wenn eine gültige Anfangsbedingung vorliegt:
Sei $w = w_1 \dots w_n$ das eingegebene Wort.

$$A = \underbrace{zust_{0,q_0}}_{\text{Startzustand } q_0} \wedge \underbrace{pos_{0,1}}_{\text{Schreibkopf auf Pos. 1}} \wedge \underbrace{\bigwedge_{j=1}^{n} band_{0,j,w_j}}_{\text{erste } n \text{ Zellen sind Eingabe } w} \wedge \underbrace{\bigwedge_{j=n+1}^{p(n)} band_{0,j,\sqcup}}_{\text{restliche Zellen sind leer}}$$

Beachte: A ist CNF mit $O(p(n))$ vielen Variablen.

Randbedingungen

Zu jedem Zeitpunkt t ist die TM in genau einem Zustand, der Kopf ist an genau einer Stelle, und in jeder Zelle steht genau ein Buchstabe:

$$R = \bigwedge_t \Big(G(\text{zust}_{t,z_1}, \dots \text{zust}_{t,z_k}) \wedge G(\text{pos}_{t,-p(n)}, \dots, \text{pos}_{t,p(n)}) \wedge \bigwedge_i G(\text{band}_{t,i,a_1}, \dots, \text{band}_{t,i,a_\ell}) \Big)$$

Für jedes t: const. viele Variablen

Für jedes t: $O(p(n))$ Variablen
$\Rightarrow$ $G(\dots)$ braucht $O(p(n)^2)$ Variablen
über alle t
$\Rightarrow O(p(n)^3)$ Variablen

Für jedes t_1 $O(p)$ Variablen

Beachte: R ist CNF mit $O(p(n)^3)$ vielen Variablen

Übergang zwischen Konfigurationen

- Bandinhalt darf sich nur dort ändern, wo der Kopf steht
- Kopf darf sich nur um eins nach links / rechts bewegen
- Übergänge müssen durch die Übergangsfunktion "gedeckt" sein.

$\ddot{U} :=$ ...

... CNF mit $O(p(n)^2)$ vielen Variablen

Endzustand

Am Ende der Berechnung muß ein Endzustand erreicht werden.

$E := z_{p(u), q_{accept}}$

CNF mit $O(1)$ vielen Variablen

Gesamtformel ist CNF mit poly. Größe

Wir definieren nun die boolesche Formel

$$\varphi_w = A \wedge R \wedge \ddot{U} \wedge E$$

Jede der Teilformeln $A, R, \ddot{U}, E$ kann als CNF geschrieben werden, und insgesamt brauchen wir $O(p(n)^3)$ viele Variablen.

$w \in L \Rightarrow$ Formel erfüllbar

Sei nun $w \in L$. Dann gibt es ein Zertifikat u der Länge $\leq p(n)$, bei dem die TM $M(w,u)$ eine gültige, akzeptierende Berechnung durchführt. In diesem Fall gibt es also eine Belegung aller Variablen, die die Formel wahr macht.

Also ist die konstruierte Formel φ_w erfüllbar.

Formel erfüllbar $\Rightarrow$ $w \in L$

Sei die konstruierte Formel φ_w erfüllbar.

Dann liegt eine gültige Berechnung der TM M vor.

Insbesondere gibt es Tableaus, bei denen lesen wir die korrekten Übergänge kriegen. Wir können also sogar ein Wort u konstruieren, das als Zertifikat dient.

Damit haben wir für das gegebene w ein u gebaut, so daß $M(w,u) = 1$ gilt, also $w \in L$.



Weitere Sprachen in NPC

Wiederholung

Eine Sprache L ist NP-hart, falls es für alle anderen Sprachen $L' \in NP$ eine Polynomzeit-Reduktion $L' \leq L$ gibt.

Eine Sprache L ist NP-vollständig, falls $L \in NP$ und L NP-hart ist. Wir schreiben $L \in NPC$.

Satz von Cook: $SAT \in NPC$.

Kennen wir schon weitere Probleme in NPC?

Weitere Probleme in NPC?

Was ist die Strategie, um zu beweisen, daß eine Sprache L in NPC ist?

Behauptung: Sei $L \in NPC$ und $L \leq L_1$.
Dann ist L_1 NP-hart.

Beweis: Offensichtlich:

$L \in NPC \Rightarrow \forall \tilde{L} \in NP : \tilde{L} \leq L \leq L_1$

Da "$\leq$" transitiv ist, gilt dann auch
$\forall \tilde{L} \in NP : \tilde{L} \leq L_1$. Also ist L_1 NP-hart. ∎

Weitere Probleme in NPC ?

Insbesondere gilt wieder:

Falls $L_1 \leq L_2 \leq L_3 \leq \ldots$ und L_1 NP-hart, dann sind auch alle anderen Sprachen aus der Folge NP-hart.

Um zu zeigen, daß $L \in NPC$ ist, müssen wir also eine Sprache $L_1 \in NPC$ finden, so daß $L_1 \leq L$ gilt. (Und dann müssen wir noch $L \in NP$ zeigen).

Schon bekannte Beispiele

Haben schon gesehen:

- $SAT \leq 3SAT \leq$ Clique $\leq$ Vertex Cover

  Da $SAT \in NPC$ gilt, sind also alle anderen Probleme in dieser Reihe NP-hart.

- Haben auch gesehen: alle diese Probleme sind in NP.

Also sind auch 3SAT, Clique, Vertexcover $\in NPC$.

Noch ein Beispiel: ILP

Integer-Linear-Programming / "ILP": Gegeben ein lineares Ungleichungssystem der Form $Ax \leq b$, wobei A eine $m \times n$-Matrix mit ganzzahligen Einträgen ist, und b ein ganzzahliger Vektor der Länge m.

Gibt es einen Vektor $x \in \{0,1\}^n$, so daß das Gleichungssystem erfüllt ist?

Erste Beobachtung:

offensichtlich gilt $ILP \in NP$ (warum?)

3SAT ≤ ILP

Konstruktion der Reduktion

Sei eine 3CNF-Formel gegeben mit n Variablen und k Klauseln.

- Wir bauen nun die **Matrix A** wie folgt: $A = (a_{ij})_{ij} \in \mathbb{Z}^{k \times n}$

$$a_{ij} = \begin{cases} 1 & \text{falls Variable } j \text{ in Klausel } i \text{ negiert auftaucht} \\ -1 & \text{un negiert} \\ 0 & \text{gar nicht} \end{cases}$$

a_{ij}: i = Klausel, j = Variable

A kodiert also die CFN in eindeutiger Art und Weise.

- Wir definieren den **Vektor b** mit $b_i = -1 + \sum_{j=1}^{n} \max\{0, a_{ij}\}$

b_i = "zählt" wie viele Variablen j negiert in Klausel i auftauchen

3SAT ≤ ILP

Beispiel: $\phi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee x_4)$

$k = 2$ Klauseln, $n = 4$ Variablen

$$A = \begin{pmatrix} -1 & -1 & 1 & 0 \\ 1 & 0 & -1 & -1 \end{pmatrix}$$

← Jede Zeile entspricht einer Klausel. Sie hat immer höchstens 3 Einträge $\neq 0$.

↑ jede Spalte entspricht einer Variable.

$$b = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Wollen entscheiden ob es x mit $Ax \leq b$ gibt.

3SAT ≤ ILP

- Betrachten Vektor $x = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$.

  Erfüllt die CNF und auch das Ungleichungssystem (wobei 1 = true, 0 = false ist).

- Betrachte den Vektor $x = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$

  Erfüllt die CNF und auch das Ungleichungssystem (wobei 1 = true, 0 = false ist).

→

3SAT $\leq$ ILP

- Betrachte den Vektor $x = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}$. Erfüllt Klausel 1, aber nicht Klausel 2.

Im Gleichungssystem erfüllt x Zeile 1, aber nicht Zeile 2

$$A \cdot x = \begin{pmatrix} -1 & -1 & 1 & 0 \\ 1 & 0 & -1 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} -2 \\ 1 \end{pmatrix} \quad \begin{matrix} \leq \\ \not\leq \end{matrix} \quad \overbrace{\begin{pmatrix} 0 \\ 0 \end{pmatrix}}^{b}$$

Vermutung: $(Ax)_i \leq b_i \iff$ Belegung x erfüllt Klausel i.

3SAT ≤ ILP

(0) Wir erhalten eine Instanz von ILP: offensichtlich

(a) Die Reduktion braucht nur polynomiell viel Zeit: offensichtlich

(2) Formel von 3SAT erfüllbar ⇔ ILP hat Lösung:

Wir identifizieren eine Belegung A von 3SAT mit einem Vektor x in ILP:

x_i = True $\hat{=}$ 1, x_i = false $\hat{=}$ 0.

Setze $y := Ax$.

Nun können wir zeigen:

$y_i \leq b_i$ ⇔ Summe der erfüllten Literale in Klausel i ist ≥ 1

⇔ Klausel i erfüllt

(Ü).

21 NP-vollständige Probleme von Karp

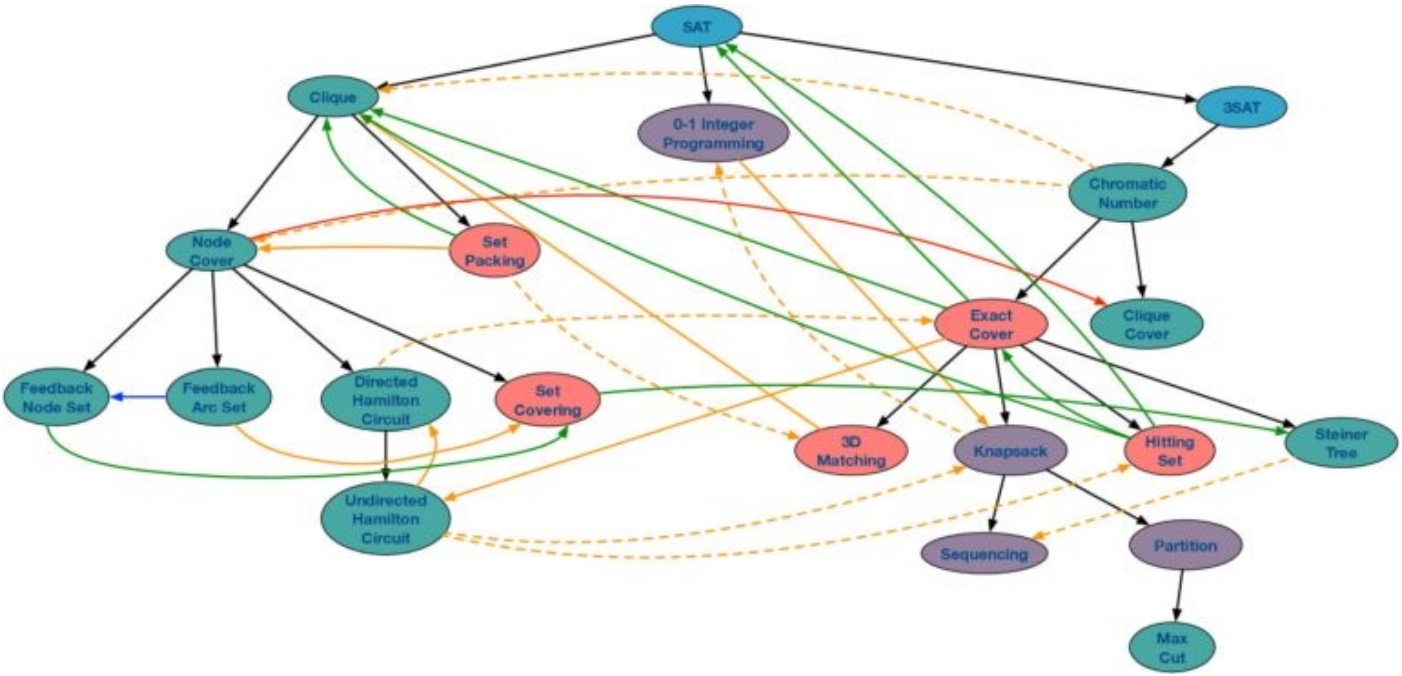


Abbildung: J. Schreiber/P. Lahkanpal/J. O'Harrow

1971: Theorem of Cook

1972: Karp's 21 Problems

Aktueller Stand

Es gibt tausende von Problemen, von denen bewiesen wurde, daß sie NP-hart sind. Viele davon sind extrem relevant für die Praxis!!!

Manche davon können erstaunlicherweise in der Praxis trotzdem oft gut gelöst werden. Manche andere aber leider nicht.

Wie kann das sein?

→ nächste Vorlesung

Gibt es Probleme "zwischen P und NPC"?

Literatur: Arora / Barak Kap. 3

Motivation

Erstaunlicherweise sind fast alle bekannten, „schwierigen" Probleme gleich in NPC. Die leichten sind alle in P.

Aber gibt es nur „dazwischen"?

(Kann es nur der Fall sein falls P ≠ NP).



Kandidaten für Probleme zwischen P und NP

Es gibt wenige interessante Probleme, für die man zur Zeit weder beweisen kann, daß sie in P sind, noch daß sie in NPC sind. Die beiden folgenden werden immer genannt:

Graph-Isomorphismus: Gegeben zwei Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ mit gleich vielen Knoten und Kanten. Gibt es eine Abbildung $f: V_1 \to V_2$ so daß $(u,v) \in E_1 \iff (f(u), f(v)) \in E_2$?

Faktorisierung: Gegeben drei nat. Zahlen N, L, U, entscheide ob N einen Primfaktor p im Intervall $[L, U]$ hat.

Mehr Zeit ⇒ mehr Macht

Wollen zeigen: Turingmaschinen, die mehr Zeit zur Verfügung haben, können auch mehr Sprachen erkennen:

Satz (Zeithierarchie)

Seien f, g zwei (konstruierbare) Funktionen, so daß

$f(n) \log f(n) = o(g(n))$ d.h. f ist "deutlich kleiner" als g.

Dann gilt $\mathrm{DTIME}(f(n)) \subsetneq \mathrm{DTIME}(g(n))$

d.h. es gibt Sprachen, die zwar in Zeit $O(g(n))$, aber nicht in Zeit $O(f(n))$ berechnet werden können.

Ein ähnliches Resultat gilt auch für NTIME.

<u>Beweis</u> ein hübsches Diagonalisierungs-Argument!

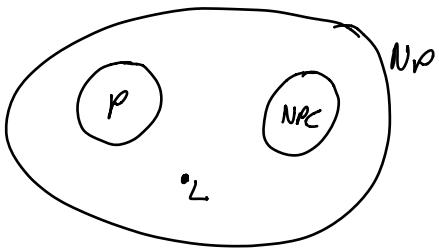
Siehe Arora/Barak Kap. 3 $\square$

"NP-Intermediate" Sprachen

Satz von Ladner (1975)

Falls $P \neq NP$ gilt, dann gibt es eine Sprache

$L \in NP \setminus (P \cup NPC)$.



Beweis: Analog über Diagonalisierung. Ähnliche Tricks wie beim Halteproblem. Man konstruiert eine Sprache, die angibt, "wie schwierig" sie selbst ist.

Arora / Barak Kap. 3.3.

Komplexitätsklassen zw. P und NPC

Falls $P \neq NP$ ... gibt es viele, siehe zum Beispiel

Buch von Papadimitriou.

Jenseits von NP

Literatur: Arora/Barak 2.6.

Motivation

Bisher: P, NP, NPC ... aber alles noch in NP.

Wo gibt es außerhalb von NP?

Die Klasse coNP

Sei $L \subseteq \{0,1\}^*$ eine Sprache. Das Komplement $\overline{L}$ oder L^c ist definiert als $\overline{L} = \{0,1\}^* \setminus L$.

d.h. Wort ist in $\overline{L}$ $\Leftrightarrow$ Wort ist nicht in L

$$\text{coNP} := \{ L \mid \overline{L} \in \text{NP} \}.$$

Erinnerung: wir hatten schon mal coRE betrachtet, gleiches Prinzip...

⚠ coNP $\neq$ Komplement von NP!

co NP: nochmal

$\overline{L} \in NP$ $\Leftrightarrow$ es gibt Zertifikat u für alle ja-Instanzen von $\overline{L}$

$\Leftrightarrow$ es gibt Zertifikat für alle nein-Instanzen von L

Alternative Charakterisierung

Satz Sei $L \subseteq \{0,1\}^*$. Dann ist $L \in \text{co}NP$ $\Leftrightarrow$
es gibt ein Polynom p und eine deterministische TM M mit
polynomieller Laufzeit, so daß für jedes $x \in \{0,1\}^p$:

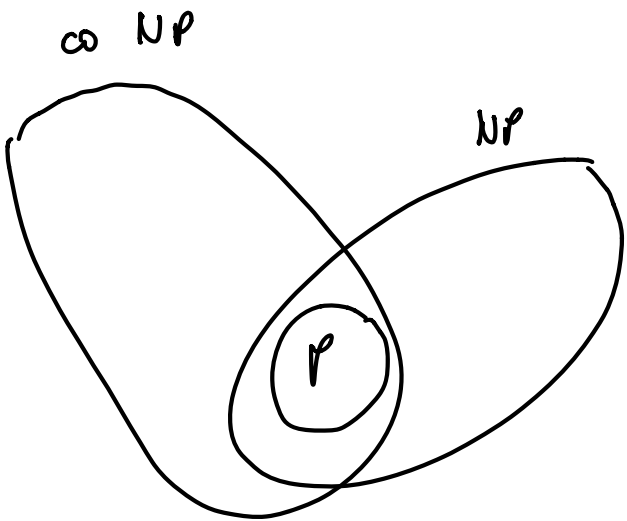
$$x \in L \Leftrightarrow \forall u \in \{0,1\}^p : M(x,u) = 1.$$

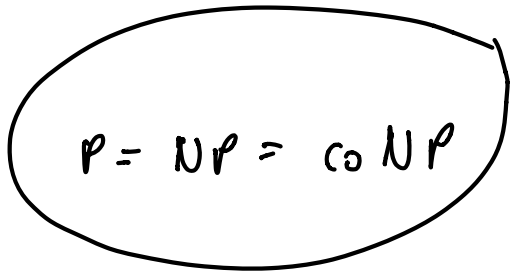
⚠ Bei NP hatten wir hier einen Existenzquantor, bei $\text{co}NP$
einen $\forall$-Quantor:

... $x \in L \Leftrightarrow \exists u \in \{0,1\}^p : M(x,u) = 1.$

Bew Wir nehmen den Verifier von L und "drehen ihn um".

$NP \stackrel{?}{\neq} co\,NP$?





Satz: $P = NP \Rightarrow NP = co\,NP$

Vermutung: $\overline{3SAT} \notin NP$, d.h. $NP \neq co\,NP$

Und jenseits davon?

Betrachte das Problem

keine zwei Knoten adjazent

Independent Set := $\{(G,k) \mid$ Graph G hat eine unabhängige Knotenmenge der Größe mind. $k\}$

Ist offensichtlich in NP: Zeuge ist das independent set

Exact-Indep.-Set := $\{(G,k) \mid$ die größte unabh. Menge in G hat genau k Knoten$\}$

Sieht nicht nach NP aus, denn was wäre ein Zeuge? Müssen ja nachprüfen, daß es keine größeren unabh. Mengen gibt.

Die Klasse Σ_2

Welche Quantoren brauchen wir?

- NP: $\exists$ Zeuge u so daß ...
- co NP: $\forall$ Zeugen u gilt: ...
- Σ_2P: $\exists$ Zeuge u $\forall$ großen Neugen ...

Die Klasse Σ_2

Def Die Klasse Σ_2 besteht aus allen Sprachen L, für die es eine det. TM M mit polynomieller Laufzeit und ein Polynom q gibt, so daß gilt:

$$x \in L \iff \exists u \in \{0,1\}^{q(n)} \; \forall v \in \{0,1\}^{q(n)} : M(x,u,v) = 1$$

Bemerkung: $NP \subset \Sigma_2$, $coNP \subset \Sigma_2$.

Beispiel: Exakt-Ind.-Set $\in \Sigma_2$

Die Klassen Σ_i

Def Sei $i \in \mathbb{N}$. Eine Sprache L ist in Σ_i, falls es eine det. TM mit poly. Laufzeit und ein Polynom q gibt, so daß gilt:

$$x \in L \iff \exists u_1 \in B \; \forall u_2 \in B \; \exists u_3 \in B \; \forall u_4 \in B \; \ldots : M(x, u_1, u_2, \ldots, u_i) = 1.$$

wobei $B = \{0,1\}^{q(|x|)}$

Bem $NP = \Sigma_1$

Polynomielle Hierarchie

Die Klassen $\Sigma_1, \Sigma_2, \ldots$ nennt man die Polynomielle Hierarchie.

Es ist ein offenes Problem, ob die Klassen strikt ineinander enthalten sind oder nicht:

$$\Sigma_1 \overset{?}{\subsetneq} \Sigma_2 \overset{?}{\subsetneq} \Sigma_3 \subsetneq \ldots$$

- Falls es ein i gibt, so daß $\Sigma_i = \Sigma_{i+1}$, dann "bricht die Hierarchie zusammen": $\Sigma_i = \Sigma_{i+1} = \Sigma_{i+2} = \ldots$
- Falls $P = NP$ gilt, dann kollabiert die ganze Hierarchie auf P.

Alternativen zur worst-case-Analyse

Motivation

Haben gesehen: viele relevante Probleme sind NP-hart / NP-vollständig.

Aber trotzdem können viele von ihnen in der Praxis erstaunlich gut gelöst werden!

Zum Beispiel SAT, ausgerechnet: es gibt ziemlich gute SAT-Solver, in der Praxis wird SAT als nicht (mehr) problematisch betrachtet.

Wie kann das sein? Ist ein Fehler in der Theorie?

Bisherige Theorie zu "streng"?

- Bisherige Aussagen betreffen immer den worst case.
  Aber vielleicht taucht dieser in der Praxis gar nicht oft auf?

- Bisherige Aussagen betreffen immer exakte Lösungen. Vielleicht reichen approximative Lösungen oft aus und sind einfacher zu finden?

Diese Vorlesung: Ausblick in viele verschiedene Richtungen...

Limits von worst case

Die ganze Theorie bisher beruht auf einer worst-case-Analyse: Für jedes n gibt es mind. eine Instanz, die schwer ist.

Aber vielleicht sind ja viele andere Instanzen gar nicht so schwer zu lösen??

Oft sind die schweren Instanzen ganz besonders konstruiert, aber "die meisten" Instanzen sind gar nicht so schlimm.

Beispiel: Was sind die worst case Instanzen für Sortieralgorithmen?

Average Case Analyse

Sei I_n eine Menge von Instanzen der Länge n (da wir es meist mit diskreten Objekten zu tun haben, ist I_n endlich).

Sei P eine Wahrscheinlichkeitsverteilung über I_n.

Sei A ein Algorithmus (eine Turing-Maschine), die das gestellte Problem löst. Sei $T(x)$ die Zeit, die der Algorithmus auf Instanz x braucht. Dann ist die mittlere Laufzeit (average time) von Algorithmus A auf Instanzen der Länge n definiert als

$$E(T(x)) = \sum_{x \in I_n} p(x) \cdot T(x)$$

↓ Erwartungswert über die Menge aller Instanzen

Average Case Analyse

Beispiel Quicksort: In Ihrer Algorithmen-Vorlesung haben Sie vielleicht gesehen:

Quicksort hat worst case Laufzeit $\Theta(n^2)$ und

average case Laufzeit $\Theta(n \log n)$.

Hier wurde der average über alle Permutationen einer Folge der Länge n gebildet, und jede Permutation war gleich wahrscheinlich.

Average case analysis

Trotzdem ist diese Analyse oft unbefriedigend, da das Ergebnis sehr stark davon abhängt, über welche Instanzen und mit welcher Wahrscheinlichkeitsverteilung wir den Mittelwert bilden

Bsp: Was ist eine geeignete Verteilung über Graphen?

- Erdös-Renyi? Stochastic Blockmodel?
- Preferential attachment?
- Random geometric graphs?

Average case analysis

Oft zeigt man auch Aussagen von folgendem Typ:

Wenn wir zufällig eine Instanz unseres Problems ziehen (aus einer vorher definierten Wahrscheinlichkeitsverteilung), dann ist sie mit hoher Wahrscheinlichkeit in kurzer Zeit lösbar.

Komplexitätstheorie für average case

Man kann nun versuchen, eine Theorie für average case complexity zu bauen (Levin, Impagliazzo, 1980er/90er Jahre).

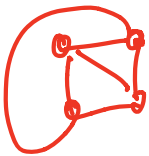
Das ist aber mühsam und nicht elegant (finde ich), und hat sich meines Wissens nicht ernsthaft durchgesetzt.

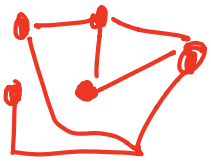


Literatur dazu: Arora/Barak Buch Kap. 18

Einschränkung auf bestimmte Instanzen

Oft kommen in der Praxis gar nicht beliebige Instanzen eines Problems vor. Und diejenigen, die vorkommen, sind vielleicht nicht grade die worst cases.





Beispiel:

- Falls Graphen planar sind, kann man das graph isomorphism problem in linear Zeit lösen

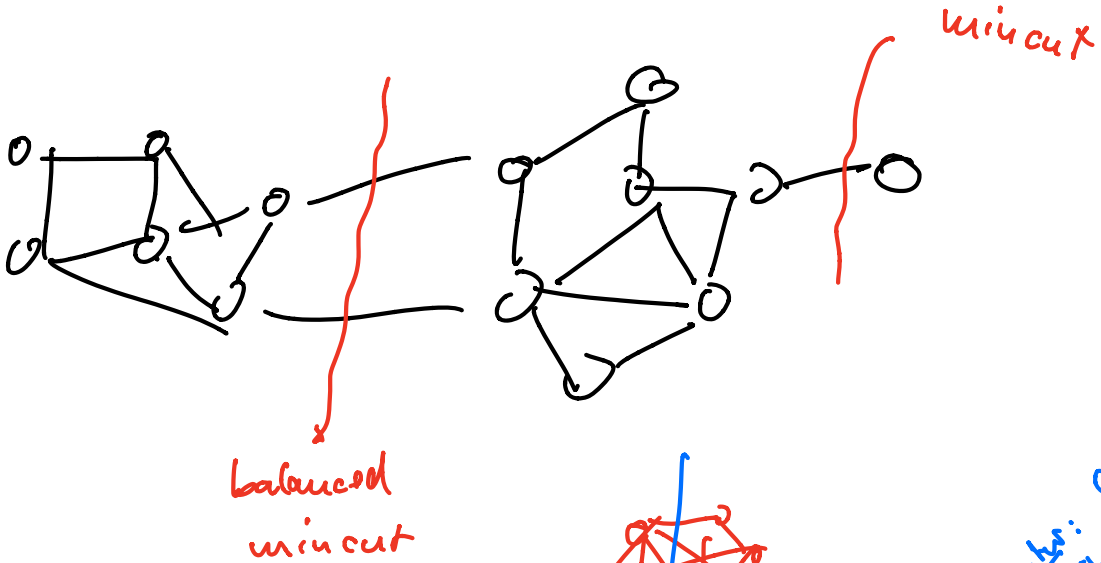
Zeichnung in Ebene, so daß Kanten sich nicht überschneiden

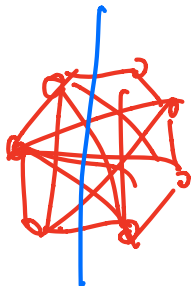
sind zwei Graphen identisch bis auf Umnummerierung der Knoten?

Einschränkung auf bestimmte Instanzen

"Clustering is hard only if it does not matter":

Clustering als ein balanced mincut problem ($\in$ NPC):



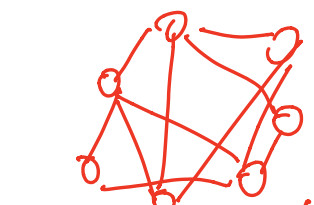


Literatur: Clustering is difficult only when it does not matter.
A. Daniely, N. Linial, M. Saks, 2012

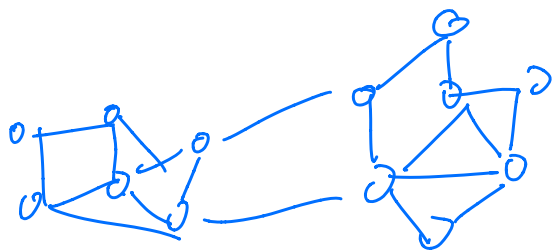
Einschränkung auf bestimmte Instanzen

"Clustering is hard only if it does not matter":

- Die "schweren" Beispiele sind Graphen, die keine gute Clusterstruktur haben. Aber die interessieren uns eh nicht.
- In den "interessanten Fällen", wo eine starke Clusterstruktur vorliegt, sollte sie auch leicht zu finden sein.
- Stability / perturbation analysis: Wenn die optimale Lösung stabil unter Perturbationen ist, dann ist sie auch leicht zu finden.



uninteressant, schwer



interessant, einfach

Smoothed Analysis

Spielman / Teng, ≈ 2000

- Der Simplex-Algorithmus, mit dem man Lineare Programme löst, hat exponentielle worst case Komplexität.
- Aber in der Praxis ist er erstaunlich schnell!
- Smoothed analysis, Intuition:
  - Wir betrachten Instanzen der Länge n
  - Wir addieren ein kleines Rauschen. (zum Beispiel auf allen Koeffizienten im Linearen Programm).
  - Dann betrachten wir den worst case über die mittlere Laufzeit (Mittelwert über alle kleinen Perturbationen):

Smoothed analysis

Sei I der Raum aller Instanzen des LP.

Worst case: $\max_{x \in I} \text{Time}(x)$

Average case: $\underset{x \in I}{\text{mean}} \text{Time}(x)$

Smoothed case: für festes ε: $\max_{x \in I} \underset{\text{Rauschen } r}{\text{mean}} \text{Time}(x + \varepsilon r)$

ε Größe des Rauschens

r Rauschen (eine Zufallsvariable)

Smoothed analysis ist also irgendwie "zwischen average case und worst case".

Smoothed analysis

Wurde erfolgreich auf einige wichtige Probleme angewandt.

- Linear programming / Simplex algorithm ⇝ Spielman/Teng
- k-means clustering ⇝ Röglin et al, 2011
- Knapsack
- ...

Aktives Forschungsgebiet.

Ausweg 2: Approximative Lösungen

Viele Probleme aus der Praxis sind Optimierungsprobleme.

- TSP: Finde kürzeste Tour im Graphen

Es ist aber oft gar nicht wichtig, die absolut beste Lösung zu finden. Eine Lösung, die z.B. 1.1 mal so lang ist wie die optimale Lösung wäre ja auch ganz ok.

Gibt es Probleme, die zwar schwer exakt lösbar sind, aber leicht "approximierbar" sind?

Approximierungs-Algorithmus

Sei x eine Instanz eines Optimierungsproblems (zum Beispiel, ein TSP-Graph). Sei $Opt(x)$ der Wert der besten Lösung (Länge der kürzesten Tour). Sei $A(x)$ der Wert, den ein bestimmter Algorithmus zurück gibt. Wir sagen, der Approximationsfaktor des Algorithmus A ist

$$\alpha_A := \max_x \frac{A(x)}{Opt(x)}$$

wobei das Maximum über alle Instanzen gebildet wird. Ein Algorithmus ist eine $f(n)$-Approximation, falls der Faktor auf Eingaben der Größe n höchstens $f(n)$ ist.

Approximation für Vertex cover

Vertex cover: gegeben ungerichteter Graph, finde die kleinste Teilmenge V' von Knoten, so daß jede Kante des Graphen mind. ein Ende in V' hat.

- Vertex Cover ist im Allgemeinen in NPC (eines von Karps 21 Problemen). :(
- Aber man kann leicht einen Algorithmus bauen, der vertex cover bis auf Faktor 2 approximiert und schnell ist :)

  ⇝ Algorithmen-Vorlesung

→

Approximation für Vertex cover

- Man kann beweisen, daß es keine konstante-Faktoren-Approximation besser eines Faktors von 1,3 geben kann. (falls $P \neq NP$).

- Vertex cover ist APX-vollständig, d.h. es kann keine beliebig gute Approximation geben (außer $P = NP$).

  ↓ PTAS, siehe nachher

Approximations-Algorithmen für TSP

- TSP auf beliebigen Metriken kann nicht in polynomieller worst case Laufzeit für auch konstanten Faktor approximiert werden. :(
- Wenn wir TSP auf Euklidische Räume einschränken, bleibt es NP-vollständig :(
- Aber: Wir können TSP auf Euklidischen Räumen leicht approximieren:
  - Faktor-2-Approximierung: → Algorithmenvorlesung
  - Faktor $(1+\varepsilon)$-Approximierung: Arora 1996. :) ↳ PTAS, next slide

Komplexitätsklasse APX

Die Klasse APX ist die Menge aller Optimierungsprobleme in NP, für die es einen Algorithmus mit polynomieller Laufzeit gibt, der einen konstanten Approximationsfaktor hat.

Ein Optimierungsproblem ist APX-hart, falls es eine PTAS-Reduktion von jedem Problem in APX zu diesem Problem gibt. Es ist APX-vollständig, falls es selbst zusätzlich in APX liegt.

Komplexitätsklasse APX

- MaxCut ist APX-hart
- TSP auf metrischen Räumen ist APX-hart

PTAS: Polynomial time approx. scheme

Ein Algorithmus heißt PTAS für ein Optimierungsproblem falls es für jeden Wert von ε eine Lösung produziert, die höchstens Faktor $(1+\varepsilon)$ von der optimalen Lösung entfernt ist, und höchstens polynomielle Laufzeit hat.

Die Laufzeit des PTAS hängt typischerweise von ε ab (je kleiner ε, desto höher die Laufzeit).

Intuition:

Wenn man ein PTAS hat, dann "ist alles gut" ☺

Jenseits von theoretischen Garantien

... gibt es natürlich ganz viele Ansätze, schwere Probleme doch zu lösen.

- greedy algorithms
- local search
- Heuristiken

Oft kann man darüber nichts beweisen, aber man kann natürlich experimentell auswerten, ob die Algorithmen oft gut funktionieren.

Aktives Forschungsgebiet

- Komplexitätstheorie ist als erste Näherung schon ganz gut, und kann vieles erklären, was wir im "Alltag" beobachten.
- Trotzdem ist es immer noch eine viel diskutierte Frage, wie man Komplexitätstheorie "passender" machen könnte.
- Es gibt sehr viele Theorien, aber keine davon ist offensichtlich überzeugend.

Zufällige Algorithmen und Komplexität

Literatur: Barak Kap. 19, 20
Sipser 10.2
Arora / Barak Kap. 7 und 20

Motivation

In der Komplexitätstheorie bisher kommt Zufall nur in einer etwas merkwürdigen Art und Weise (nicht-det. TM) vor.

Man schaut immer alle Berechnungspfade an und fragt sich nur, ob einer existiert, der akzeptiert.

Es gibt viele zufällige Algorithmen, bei denen Zufall eine wichtige Rolle spielt.

Frage: Wie modellieren wir das sinnvollerweise?

Quicksort

Rekursiv führe die folgenden Schritte aus:

- Wähle ein Pivot-Element p, typischerweise zufällig (uniform) aus den gegebenen Elementen
- Spalte die derzeitige Menge auf in zwei Teile, die Elemente $\leq p$ und $> p$.

  Vermutung: wenn p zufällig gewählt ist, dann sind die zwei entstehenden Mengen ähnlich groß, mit hoher Wahrscheinlichkeit.

- Am Ende setze die entstehenden Mengen "richtig" zusammen.

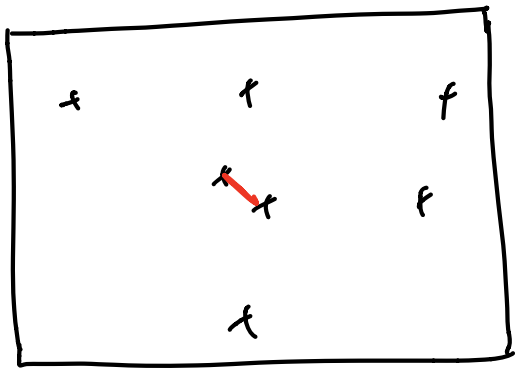
Quicksort

Satz Der Algorithmus liefert immer das korrekte Ergebnis zurück. Die Laufzeit ist zufällig, mit Erwartungswert $O(n \log n)$ und nie länger als $O(n^2)$.

Closest pair of points

Gegeben sind n Punkte in der Ebene $\mathbb{R}^2$.

Frage: welche zwei sind am nächsten beisammen?



Closest pair of points

Deterministische Algorithmen:

- Brute force: alle paarweisen Abstände ausrechnen: $O(n^2)$
- Divide and conquer Ansatz: Raum in immer kleinere Teile spalten $\rightsquigarrow O(n \log n)$

Gut, aber es geht noch besser!

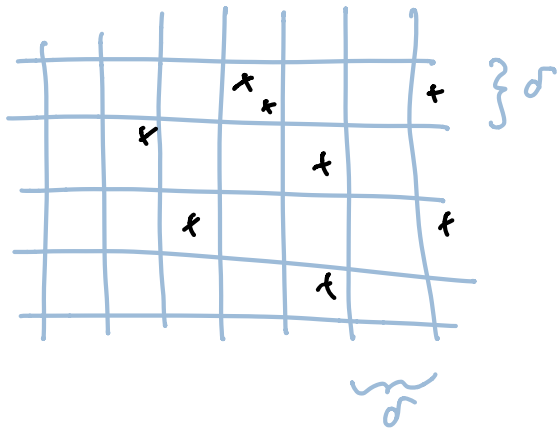
Closest pair of points

Zufälliger Algorithmus (Rabin, 1976)

(1) Wähle $\sqrt{n}$ Punkte zufällig aus und berechne deren paarweise Abstände $O(\sqrt{n}^2) = O(n)$. Sei δ der kleinste Abstand dabei.

(2) Teile den Raum in ein Gitter der Seitenlänge δ

(3) Da der kleinste Abstand $\leq \delta$ sein muß, wissen wir, daß das closest pair im selben oder benachbarten Gitterwürfel liegen muß.



(4) Führe brute force Suche über alle benachbarten Würfel durch.

Closest pair of points

Satz Der Algorithmus liefert immer das korrekte Ergebnis zurück. Mit hoher Wahrscheinlichkeit ist seine Laufzeit in $O(n)$.

Beweis ~> Algorithmenvorlesung

Cool, $O(n)$!!! Besser kann es nicht gehen!

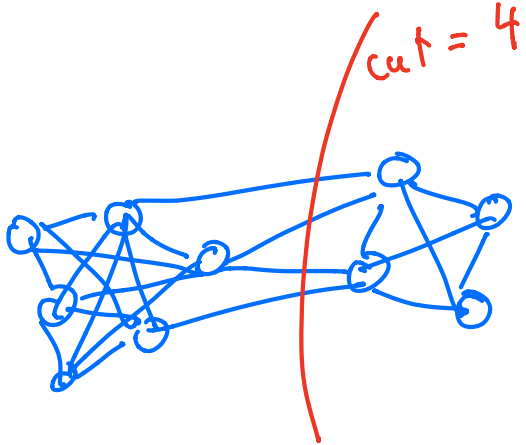
Randomized Mincut

Gegeben: Graph $G = (V, E, w)$ mit nicht-negativen Kantengewichten w, ungerichtet.

Gesucht: Partition $V = V_1 \dot\cup V_2$ so daß

$$\text{cut}(V_1, V_2) := \sum_{\substack{i \in V_1 \\ j \in V_2}} w_{ij}$$

minimiert wird.



Randomized Mincut

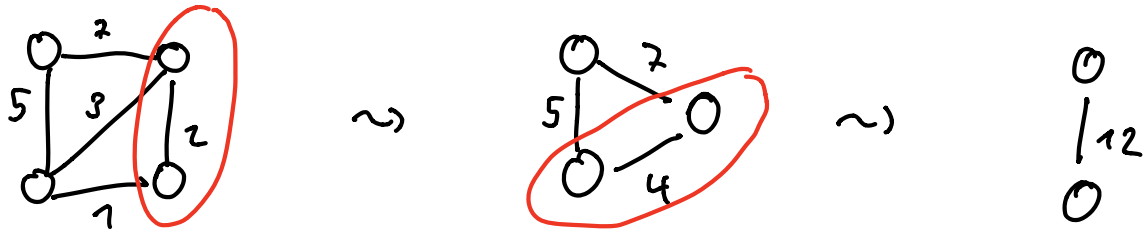
Deterministische Algorithmen:

- Mincut-Maxflow Algorithmen, je nach Implementierung
  $O(n\,m\,\log(n^2/m))$ wobei $n = |V|$, $m = |E|$.
  $\approx \tilde{O}(n^3)$
  # Knoten
  # Kanten
- Stoer/Wagner contraction algorithm: $O(n \cdot m + n^2 \log n)$
- ...

Randomized Mincut

Zufälliger Algorithmus von Karger/Stein, 1993/1996

Kantenkontraktion:



Algorithmus:

Solange der Graph mehr als eine Kante hat

- wähle eine davon zufällig aus (uniform) und kontrahiere sie

Die zwei übrigen Kanten repräsentieren die Partition.

Randomized Mincut

Intuition:

- wollen Kanten kontrahieren, die nicht "im mincut" liegen.
- Da dort nur wenige Kanten sind (mincut!), ist die Wahrscheinlichkeit groß, sie nicht zu erwischen.

Satz Eine verbesserte Version dieses Algorithmus (rekursiv) hat deterministische Laufzeit $O(n^2(\log n)^3)$ und findet einen mincut mit Wahrscheinlichkeit mindestens $1 - 1/n!$

Cool! $\approx \tilde{O}(n^2)$ statt $\approx \tilde{O}(n^3)$

Monte Carlo Algorithmen

- **Ergebnis** des Algorithmus hängt vom Zufall ab (und ist nicht immer das gewünschte Ergebnis, aber typischerweise mit hoher W.keit das richtige Ergebnis, oder mit hoher W.keit nahe dran).

- **Laufzeit** kann durch eine deterministische Funktion beschrieben werden.

Beispiel: Mincut von Karger/Stein

Las Vegas Algorithmen

- Algorithmus liefert immer das korrekte Ergebnis
- Laufzeit hängt vom Zufall ab. Typischerweise analysiert man dann deren Erwartungswert.

Beispiel:

- Quicksort
- Closest pair of points

Komplexitätstheorie für zufällige Algorithmen

- Brauchen zunächst ein passendes Rechenmodell

  ⇝ probabilistische Turing-Maschine

- Dann können wir Komplexitätsklassen definieren

  ⇝ BPP, ZPP

Probabilistische Turingmaschine

Def Eine probabilistische Turingmaschine, PTM, ist eine Turingmaschine mit zwei Übergangsfunktionen δ_0 und δ_1. In jedem Schritt wählen wir mit W.keit $1/2$ entweder den Übergang wie in δ_0 beschrieben oder wie in δ_1 beschrieben.

Intuition: Zu jedem Zeitpunkt gibt es zwei Möglichkeiten mit je W.keit $1/2$.

Alle diese Zufallsentscheidungen passieren unabhängig.

Mit $M(x)$ bezeichnen wir die Zufallsvariable, die das Ergebnis der Berechnung (akzeptieren oder verwerfen) enthält. Wir sagen, M hat Laufzeit $T(n)$, falls M für jede Eingabe der Länge n immer innerhalb von $T(n)$ Schritten anhält.

egal wie die Zufallsbits ausgefallen sind

PTM vs NTM

Sowohl in PTM als NTM gibt es einen Baum mit allen möglichen Berechnungswegen. Bei der PTM weisen wir jedem Pfad eine W.keit zu, bei der NTM nicht.

- Eine NTM akzeptiert, falls es mind. einen Pfad im Baum gibt, der akzeptiert.
- In einer PTM interessieren wir uns dafür, "wie viele Pfade" (mit welcher W.keit) zum Ende akzeptieren.

Laufzeit einer PTM

Def Sei $T: \mathbb{N} \to \mathbb{N}$, $L \subset \{0,1\}^*$. Wir sagen, eine PTM M **entscheidet L in Zeit $T(n)$** falls

- für jede Eingabe x der Länge n die PTM M nach $T(n)$ Schritten anhält (egal was die zufälligen Entscheidungen waren).
- und die **Wahrscheinlichkeit**, daß M das **richtige** Ergebnis liefert, mind. $2/3$ ist:

$$\forall x: \quad P(M(x) = L(x)) \geq 2/3 \qquad \text{wobei}$$

$$L(x) = \begin{cases} 1 & x \in L \\ 0 & \text{sonst} \end{cases}$$

Wichtig: Für jedes x kommt mit W.keit $2/3$ das richtige raus.
(Wir haben keine W.keitsverteilung über dem Eingaberaum!)
Also immer noch sogar wie worst case über x.

Warum 2/3 ?

... in der Definition der Laufzeit?

- Welche Konstante man hier wählt ist fast egal, solange sie nicht 1/2 ist. Denn:
  Durch unabhängiges Wiederholen der zufälligen Berechnung kann man die Fehlerwahrscheinlichkeit beliebig klein kriegen:

Wahrscheinlichkeit erhöhen

Satz Sei A ein zufälliger Algorithmus, der mit W.keit $p > 0$ das richtige Ergebnis liefert. Dann können wir daraus einen Algorithmus $\tilde{A}$ bauen, der mit W.keit $1 - \varepsilon$ das richtige Ergebnis liefert (für $0 < \varepsilon < 1$): Wir wiederholen A einfach mindestens $b := \log(1/\varepsilon)/p$ mal unabh. voneinander und nehmen das beste Ergebnis. z.B. Sortieren

Bew $P(\tilde{A} \text{ falsch}) = P(\text{alle } b \text{ Wiederholungen falsch})$

$= (1-p)^b \leq \varepsilon$

Wir lösen nach b auf und benutzen $\log(1-p) \approx -p$. Dann

$$b \geq \frac{\log \varepsilon}{\log(1-p)} \approx \frac{\log \varepsilon}{-p} = \frac{\log(1/\varepsilon)}{p}$$

□

Wahrscheinlichkeit erhöhen

Wir können also zum Beispiel $\varepsilon = \frac{1}{n}$ setzen. Dann ist der Alg. $\tilde{A}$ korrekt mit W.keit mind. $1 - \frac{1}{n}$, und hat Laufzeit

$$T_{\tilde{A}}(n) = b \cdot T_A(n) = \frac{\log(1/\varepsilon)}{p} \cdot T_A(n)$$

$$= \frac{\log(n)}{p} \cdot T_A(n)$$

Eine ähnliche Analyse kann man machen, wenn man am Ende das majority vote über die Entscheidungen der Algorithmen macht.

(Siehe Kap. W.2, Boole)

BPTIME und BPP

Def $\mathrm{BPTIME}(T(n))$ ist die Klasse der Sprachen, die von einer PTM in Zeit $O(T(n))$ entschieden werden können.

bounded error probabilistic

Def $\mathrm{BPP} := \bigcup_{c \in \mathbb{N}} \mathrm{BPTIME}(n^c)$ sind die Sprachen, die von einer PTM in polynomieller Zeit entschieden werden können.

bounded error probabilistic polynomial time

BPP $\hat{=}$ Monte-Carlo-Algorithmen

- Laufzeit deterministisch beschränkt
- Richtiges Ergebnis mit "hoher" W.keit.

Beispiele:

- Sortieren
- Mincut

P vs BPP?

- Offensichtlich gilt $P \subset BPP$. Aber gilt vielleicht auch $P = BPP$?
  Anders gefragt: Macht der Zufall eine TM "echt stärker"?
- Antwort nicht bekannt!
- Aber viele Leute vermuten: $P = BPP$!
  Falls das stimmt, dann gibt uns der Zufall nichts, was wir nicht ohne Zufall schon könnten!!!

Das schauen wir uns gleich genauer an: De-Randomisierung.

ZTIME und ZPP

Def $ZTIME(T(n))$ enthält diejenigen Sprachen L, für die es eine PTM M gibt, deren erwartete Laufzeit $O(T(n))$ ist und die für jede Eingabe x das richtige Ergebnis liefert.

zero error

$$ZPP := \bigcup_{c \in \mathbb{N}} ZTIME(n^c).$$

ZPP $\hat{=}$ Las Vegas Algorithmen

- ZPP ist die Klasse für Las-Vegas-Algorithmen:
  - immer richtiges Ergebnis
  - Erwartete Laufzeit polynomiell.

- Bekannt: $ZPP \subset BPP$. Andere Richtung unbekannt.

- Auch unbekannt: $P \overset{?}{=} ZPP$?

De-Randomisierung

Man kann einen zufälligen Algorithmus immer deterministisch machen, indem man alle möglichen zufälligen Sequenzen simuliert:

- Sei M eine PTM, die $r(n)$ zufällige Bits benutzt, $t(n)$ Zeit benötigt, und die korrekte Antwort mit W.keit $p > \frac{1}{2}$ gibt.
- Nun simuliere alle $2^{r(n)}$ Bitsequenzen (deterministisch) und nimm das Mehrheitsergebnis.
- Das stimmt dann mit dem wahren Ergebnis überein, und die Laufzeit des (nun deterministischen) Alg ist $2^{r(n)} t(n)$.

Durch exponentiell viel Zeit können wir also den Zufall immer "entwerfen". Aber geht es vielleicht auch in polynomieller Zeit?

De-Randomisierung

Idee für polynomielle De-Randomisierung:

benutze Pseudo-Zufallszahlengenerator:

- Polynomiell viele Sequenzen, die kein "Gegner" von echten Zufallssequenzen unterscheiden könnte
- Simuliere mit diesen polynomiell vielen Sequenzen.
- Wenn es gut läuft, verhält sich der Algorithmus dann so wie auf "echten" Zufallsvariablen aber nur mit polynomiellem Overhead.

$\rightsquigarrow$ Literatur

Take home messages

aus der

Komplexitätstheorie

Komplexitätstheorie

- Wollen Probleme in Schwierigkeitsklassen einordnen
- Rechenzeit
- Speicherplatz

(Wir haben uns hauptsächlich mit Zeit beschäftigt.)

P : die leichten Probleme

- in polynomieller Worst Case Laufzeit lösbar
- enthält viele Probleme, die in der Praxis wichtig sind: kürzeste Wege, sortieren, ...
- Obwohl "Polynom" immer noch zu schwer für die Praxis sein könnte (n^{100} zum Beispiel), sind viele Probleme aber P mit Polynomen mit kleinem Grad lösbar: n^2, n^3, n^5.

P: die leichten Probleme

Es gibt aber auch einige, für die man nur Algorithmen mit hohen Exponenten kennt:

- PTAS mit kleinem ε
- Max-Bisection ist approximierbar mit Faktor 0.8 in Zeit $O(n^{10^{100}})$. Austrin et. al, ACM Trans. Alg, 2016
- Sampling a point from a convex body, $O(n^{19})$
  Dyer, Frieze, Kannan, JACM 1991
- Recognizing "map graphs" (≈ planar graphs), $O(n^{120})$
  Thorup, FOCS 1998

NP und NPC: potentiell schwere Probleme

- In polynomieller Zeit verifizierbar (nicht notwendig lösbar)
- NPC: die schwersten Probleme in NP
- Enthält viele wichtige Probleme: SAT, TSP, ...

$$P \overset{?}{=} NP$$

Das größte offene Problem der Informatik,
mit möglicherweise weitreichenden Folgen

Verbreitete Irrglauben

NP (oder wahlweise, NPC) sind die schwersten Probleme, die es gibt!

Falsch:

- Es gibt viele Probleme jenseits von NP, bis hin zu Problemen, die nicht mal mehr entscheidbar / berechenbar sind

Verbreitete Irrglauben

Wenn ein Problem NP-vollständig ist, dann gibt es keine Chance, es zu lösen.

Falsch! Im worst case geht es nicht in polynomieller Zeit, aber vielleicht auf vielen anderen (manchmal sogar fast allen) Instanzen schon, und vielleicht gibt es Approximationen oder Heuristiken, die "gut genug" funktionieren.

Verbreitete Irrglauben

Wenn der Suchraum exponentiell viele Elemente hat, dann ist das Problem NP-hart.

Falsch! Oft gibt es Strukturen, die man ausnutzen kann.

Beispiel: mincut

Viele weitere Themen

Es gibt ganz viele weitere Themen, die wichtig sind und die wir ja nicht angesprochen haben

↝ Buch Arora / Barak

Mein persönliches Fazit

- Komplexitätstheorie ist auch für Anwenderinnen wichtig.
- Die Theorie ist toll, elegant, macht Spaß.
- Hat aber viele Grenzen, an denen man nicht so wichtig weiter kommt.
- Forschung in Komplexitätstheorie ist mittlerweile sehr technisch geworden, die Einstiegshürde ist ziemlich hoch.
- Es gibt viele wichtige Fragen, die von Komplexitätstheorie gar nicht gefragt werden (können)

  ⇝ Maschinelles Lernen, KI