Berechenbarkeit + Komplexität

Vorlesung: Di 1415 – 1545

Tutorien Mo, Di 8–10 Mo, Di 10–12

im Rubikon anmelden, Beginn 26.4.

schr. Prüfungstermine: 22.7 und 18.10. (offen)

rubikon.informatik.uni-ulm.de

Was sind berechenbare Funktionen?

Adäquate Definition

(Turing, Church, Kleene, Herbrand, Gödel...)
ab 1936

verschiedene alternative Def. von „berechenbar"
haben sich als äquivalent herausgestellt

Deshalb:

> Der intuitive Berechenbarkeitsbegriff wird adäquat erfasst durch die (z.B.) Turing-Berechenbarkeit.

Church'sche These (Church-Turing-These)

$f: A \to B$ sinnvoll nur wenn A, B abzählbar, z.B. $\mathbb{N}$, Σ^*

Beispiele: $f: \mathbb{N} \to \{0,1\}$

$$f(x) = \begin{cases} 1, & x \text{ (als Dezimalzahl) entspricht Anfangsabschnitt von } \pi \\ 0, & \text{sonst} \end{cases}$$

Bsp: $f(314) = 1 \quad f(55) = 0 \quad f(3146) = 0$

f ist berechenbar: π kann approximiert werden: $a_n \to \pi \quad (n \to \infty)$

$$|a_n - \pi| \leq \varepsilon_n$$

$$g(x) = \begin{cases} 1, & x \text{ kommt irgendwo in } \pi \text{ vor} \\ 0, & \text{sonst} \end{cases}$$

$g(14) = 1 \quad g(890123) = ?$

... könnte nicht berechenbar sein.

$$h(x) = \begin{cases} 1, & \text{irgendwo in } \pi \text{ kommt} \geq x\text{-mal 7'er vor} \\ 0, & \text{sonst} \end{cases}$$

~~ein~~

$h(1) = 1 \quad h(23) = ?$

Entweder kommen beliebig lange 7er Folgen in π vor, dann $h(x) = 1 \quad \forall x$.

Oder: $\exists n_0$ es gibt 7er-Folge bis zur Länge n_0, aber nicht $n_0 + 1$

Dann gilt: $h(x) = \begin{cases} 1, & x \leq n_0 \\ 0, & x > n_0 \end{cases}$

Unendliche viele potenzielle Algorithmen:

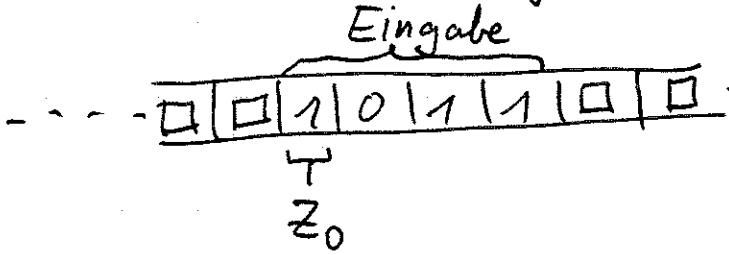
$A_1, A_2, A_3, \ldots$ Einer davon berechnet die Funktion h.

$i(x) = \begin{cases} 1, & \text{Riemansche Hyp. gilt} \\ 0, & \text{sonst} \end{cases}$

ist berechenbar.

Def. der Turing-Maschine M:

endl. Zustandsmenge $Z = \{z_0, \ldots, z_e\}$



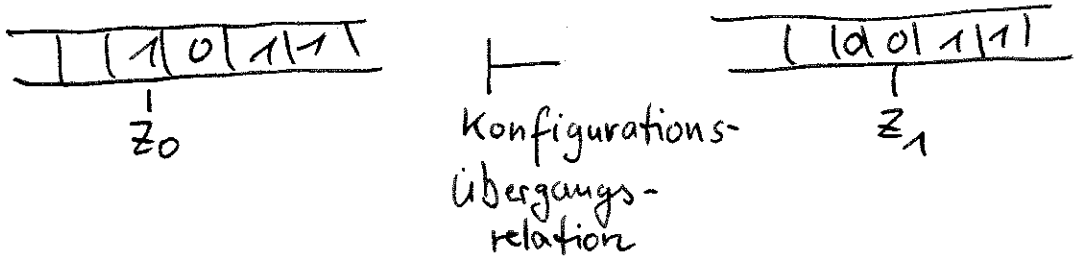
Startkonfig.

Arbeitsalphabet $\Gamma \supseteq \Sigma \cup \{\square\}$

↑ Eingabealphabet, z.B. $\Sigma = \{0,1\}$

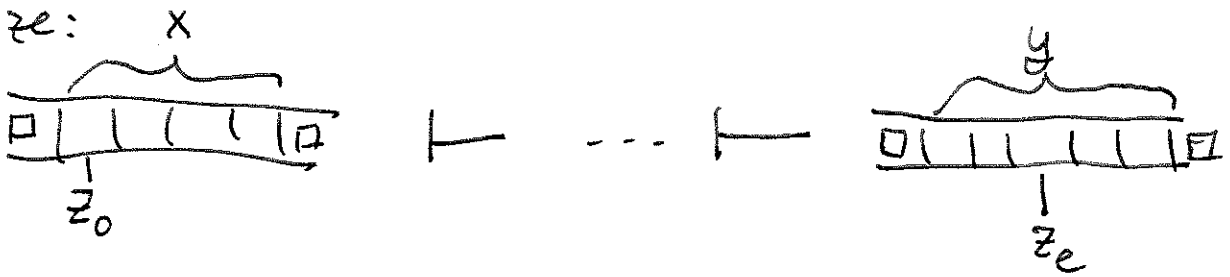
Übergangsfunktion: $\delta : Z \times \Gamma \longrightarrow Z \times \Gamma \times \{L, R, N\}$

z.B: $\delta(z_0, 1) = (z_1, 0, R)$



Def: $f: \Sigma^* \to \Sigma^*$ heißt Turing-berechenbar falls es eine TM M gibt, die bei jeder Eingabe $x \in \Sigma^*$ nach endlich vielen Schritten stoppt mit Ausgabe $y = f(x)$

Skizze:



Statt Σ^* auch $\mathbb{N}$ möglich (Binärzahlen)

Bsp: $n \mapsto n+1$ ist T-berechenbar

$$\delta(z_0, 0) = (z_0, 0, R)$$
$$\delta(z_0, 1) = (z_0, 1, R)$$
$$\delta(z_0, \square) = (z_1, \square, L)$$
$$\delta(z_1, 0) = (z_e, 1, N)$$
$$\delta(z_1, 1) = (z_1, 0, L)$$
$$\delta(z_1, \square) = (z_e, 1, N)$$

$$f_\pi(x) = \begin{cases} 1, & * \text{ Anfangsabschnitt von } \pi \\ 0, & \text{sonst} \end{cases}$$

ist berechenbar. Ähnlich mit f_e.

Gilt, dass für jede reelle Zahl $r \in \mathbb{R}$ die Funktion f_r berechenbar ist?

Es gibt nur abzählbar viele berechenbare f_r's. Aber $r \in \mathbb{R}$ überzählbar.

$\Rightarrow$ Es gibt nicht-berechenbare Funktionen.

Aufgaben / Rückschau zur letzten Vorlesung

Aufgabe: Gib eine Turingmaschine an, die die Funktion $f: \mathbb{N} \to \mathbb{N}$ mit $f(x) = 2x$ berechnet. (Natürliche Zahlen als Binärzahlen darstellen.)

Antwort: Methode: Hinten eine Null anfügen.

Eingabealphabet sei $\Sigma = \{0, 1\}$; Arbeitsalphabet: $\Gamma = \{0, 1, \square\}$; Startzustand: z_0; Endzustand: z_e.

Überführungsfunktion:

$$\begin{cases} \delta(z_0, 0) = (z_0, 0, R) \\ \delta(z_0, 1) = (z_0, 1, R) \\ \delta(z_0, \square) = (z_e, 0, N) \end{cases}$$

Frage: Lässt sich der Begriff der Turing-Berechenbarkeit auch auf partielle Funktionen erweitern?

Antwort: Ja, dies wird in der Berechenbarkeitstheorie so getan, dass man $f(x) = \text{undefiniert}$ assoziiert mit der Eigenschaft, dass die betreffende f-berechnende Maschine bei Eingabe x nicht stoppt.

Übliche Notationen:

$f(x) = \perp$ ... f an der Stelle x ist undefiniert.

$M(x)\downarrow$ ... M, bei Eingabe x, stoppt.

$M(x)\uparrow$ ... M, bei Eingabe x, stoppt nicht.

Vorteil: Durch die Ausweitung auf partielle Funktionen kann man jeder Turingmaschine M eine durch sie berechnete Funktion f_M zuordnen — die ggf. partiell sein kann.
Wenn man auf totalen Funktionen bestehen würde beim Berechenbarkeitsbegriff, so könnte man das nicht. Man müsste erst entscheiden, ob f_M eine totale Funktion ist (was algorithmisch gar nicht geht — siehe später).

Sprechweisen: f ist total berechenbar, bedeutet: f ist eine totale Funktion und es gibt eine immer stoppende Turingmaschine, die f berechnet.

f ist <u>partiell berechenbar</u>, bedeutet:
es gibt eine Turingmaschine, die f berechnet (im erweiterten Sinne), aber f könnte partiell sein (d.h. an den betreffenden Eingaben x stoppt M nicht): $f_M(x) = \perp$ gdw. $M(x) \uparrow$

<u>Aufgabe</u>: Gib eine Turingmaschine an, die die überall undefinierte Funktion Ω (also $\Omega(x) = \perp$ für alle x) berechnet!

<u>Antwort</u>: Zum Beispiel so:

$$\delta(z_0, 0) = (z_0, 0, N)$$
$$\delta(z_0, 1) = (z_0, 1, N)$$

Es wird nie der Endzustand erreicht.

<u>Bemerkung</u>: Im Sinne dieser erweiterten Definition von „berechenbar" ist dann die Funktion

$$f(x) = \begin{cases} 1, & x \text{ kommt irgendwo in } \pi \text{ vor,} \\ \perp, & \text{sonst} \end{cases}$$

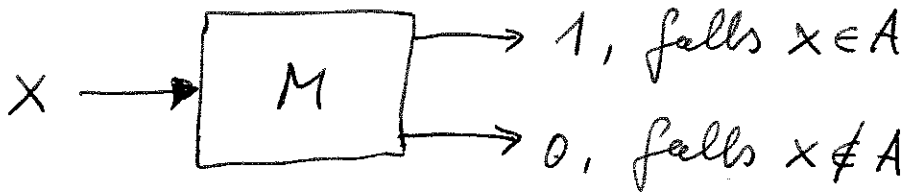
berechenbar!

Definition: Eine Teilmenge A (von $\mathbb{N}$ bzw. von Σ^*) heißt **entscheidbar** (im Sinne der Turing'schen Definition von berechenbar), falls es eine immer stoppende Turingmaschine M gibt, so dass für alle x gilt:

$$f_M(x) = \begin{cases} 1, & x \in A \\ 0, & x \notin A \end{cases}$$

Anders ausgedrückt: Die charakteristische Funktion von A (welche eine totale Funktion ist, per Definition) ist berechenbar.

Skizze:



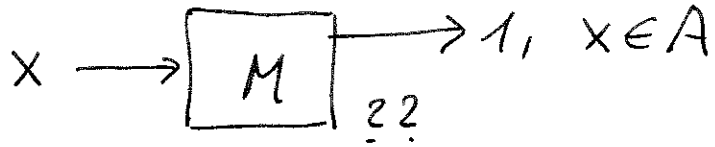
Definition: Die Menge A heißt **semi-entscheidbar**, falls es eine Turingmaschine M gibt, so dass für alle x gilt:

$$f_M(x) = \begin{cases} 1, & x \in A \\ \perp, & x \notin A \end{cases}$$

Gleichwertig:

$$x \in A \quad \text{gdw.} \quad M(x)\downarrow$$

Skizze:



Bemerkung: Dies entspricht der in der Vorlesung Formale Grundlagen gegebenen Definition, dass eine Turingmaschine die Sprache A akzeptiert.
Notation: $A = T(M)$.
In der Vorlesung wurde gezeigt, dass dies äquivalent ist zu: A ist Typ 0, d.h. A kann von einer (nicht weiter eingeschränkten) formalen Grammatik erzeugt werden, also: $A = L(G)$.

Also:

A ist Typ 0 gdw. A ist semi-entscheidbar.

Das Komplement von A ist $\mathbb{N} \setminus A$ (bzw $\Sigma^* \setminus A$).

Notation: $\overline{A}$.

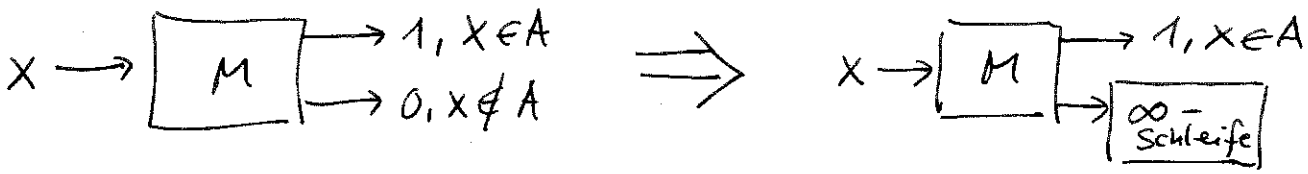
Bemerkung: Wenn A entscheidbar ist, dann auch $\overline{A}$.

Dies gilt aber im Allgemeinen nicht für die Semi-Entscheidbarkeit.
(Die Menge der Typ 0-Sprachen ist nicht komplement-abgeschlossen).

Es gilt (unmittelbar nach Definition):

A entscheidbar $\Longrightarrow$ A ist semi-entscheidbar.
$\Longrightarrow$ $\overline{A}$ ist semi-entscheidbar.

Umformung:

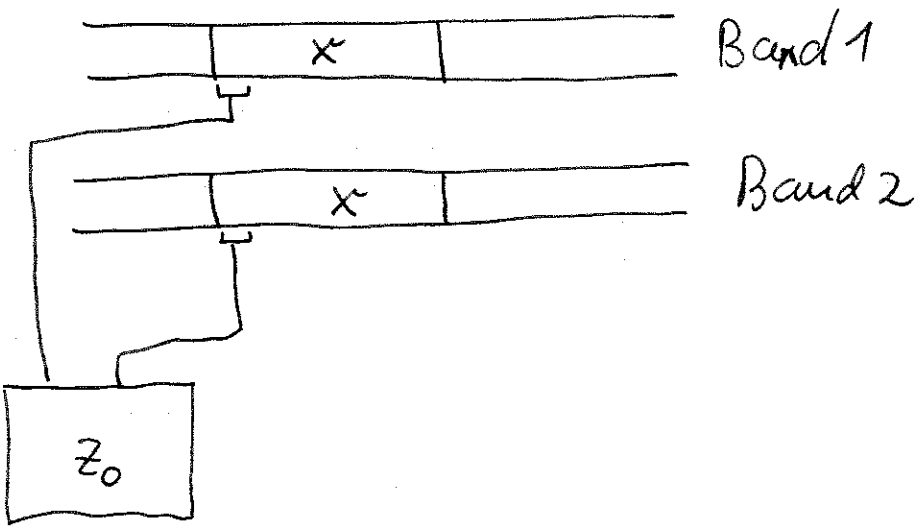


Es gilt auch die Umkehrung:

Satz: Eine Menge A ist entscheidbar
gdw.
A ist semi-entscheidbar und $\overline{A}$ ist semi-entscheidbar.

Beweis: Es verbleibt noch die Richtung von unten nach oben zu zeigen. Nach Voraussetzung gibt es also Turingmaschinen M_1 und M_2 mit folgenden „Berechnungsfähigkeiten" (symbolisch):

Wir werden später zeigen, dass (normale) Turingmaschinen und Mehrband-Turingmaschinen äquivalente Berechnungsformalismen sind.
Wir argumentieren nun, dass A entscheidbar ist durch eine (immer stoppende) 2-Band-Turing-maschine. (Damit kann A dann auch durch eine reguläre 1-Band-TM ~~berechnet~~ entschieden werden.)
Die Eingabe x wird zu Beginn auf beide Bänder kopiert, so dass dann dies die Ausgangssituation ist:



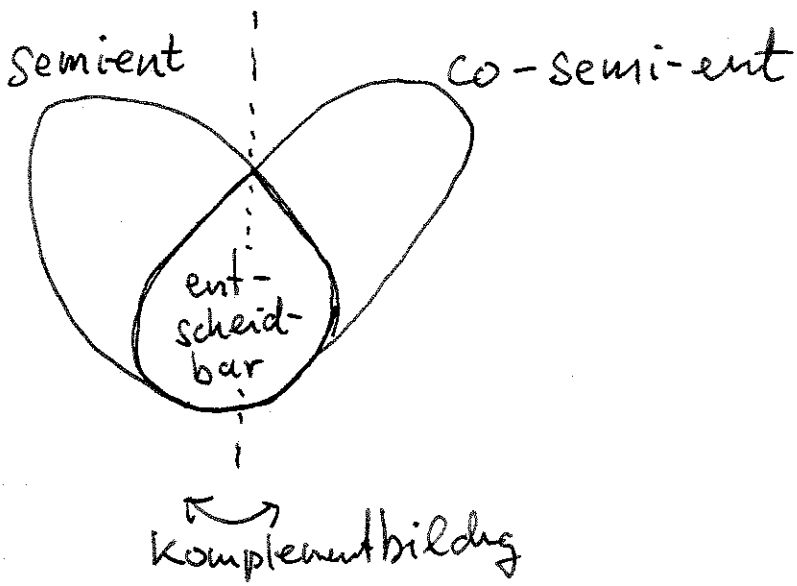
Die Überführungsfunktion δ der 2-Band-TM

$$\delta : Z \times \Gamma \times \Gamma \longrightarrow Z \times \Gamma \times \Gamma \times \{R,L,N\} \times \{R,L,N\}$$

führt dann auf Band 1 die Maschine M_1 aus und auf Band 2 die Maschine M_2 (simultan). Hierbei ist $Z = Z_1 \times Z_2$.

Sollte M_1 stoppen, so gibt die Maschine 1 aus
Sollte M_2 stoppen, so gibt die Maschine 0 aus
(und stoppt dann in beiden Fällen).

Somit: diese 2-Band-TM ist eine immer stoppende Maschine und berechnet die charakteristische Funktion von A.
A ist also entscheidbar. □



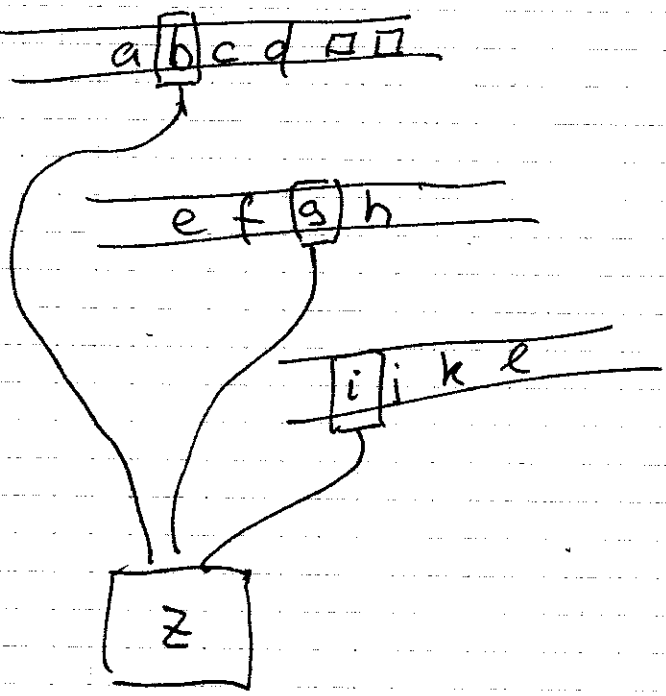
Mehrband-Turingmaschinen können auf mehreren Bändern gleichzeitig agieren. Die δ-Funktion hat jetzt die Form:

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

k = Anzahl der Bänder

Mehrband-TM können durch 1-Band-TM simuliert werden.

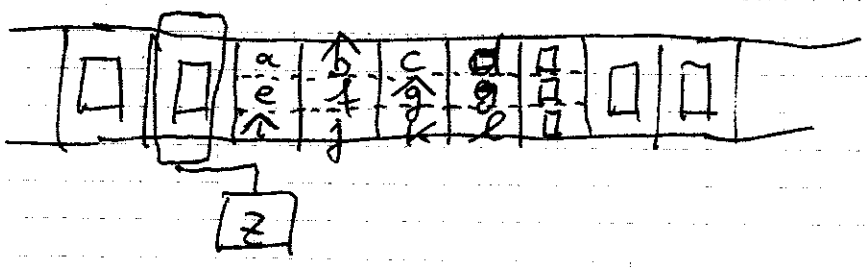
Idee:



3-Band-TM mit Arbeitsalph Γ

Bei der (simulierenden) 1-Band-TM ein erweitertes Arbeitsalphabet $\Gamma \cup (\Gamma \cup \hat{\Gamma})^3$

$$\hat{\Gamma} = \{\hat{a} \mid a \in \Gamma\}$$



Ein Schritt der Mehrband-TM wird durch einen Hintereinander-Durchgang der 1-Band-TM simuliert. Ein Buchstabe der Form â deutet an, dass der betreffende Schreibleskopf auf den Buchstaben a steht.

Vorschau:

2 Berechenbarkeitsbegriffe, basierend auf Progr-Sprachen

GOTO-Berechenbarkeit

WHILE-Berechenbarkeit

1 ~~Ber~~ math. Berechenbarkeitsbegriff:

μ-rekursiven Funktion.

Aufgabe: Gib eine 2-Band-TM an, die bei Eingabe $x \in \{0,1\}^*$ auf Band 1, dieses x auf Band 2 kopiert. (Band 2 war zu Beginn leer.)

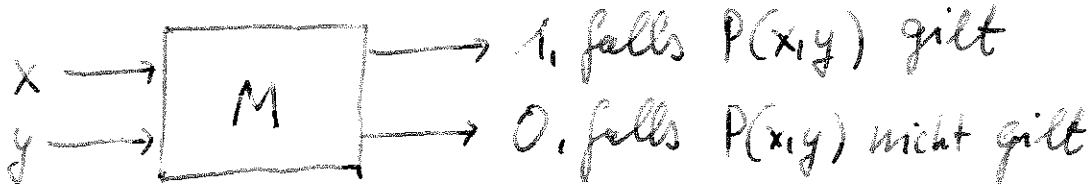
Antwort:

$$\delta(z_0, 0, \square) = (z_0, 0, 0, R, R)$$
$$\delta(z_0, 1, \square) = (z_0, 1, 1, R, R)$$
$$\delta(z_0, \square, \square) = (z_e, \square, \square, N, N)$$

Aufgabe: Man zeige, dass eine Sprache A semi-entscheidbar ist genau dann, wenn man die Mitgliedschaft von x in A durch einen Existenzquanter ausdrücken kann:

$$x \in A \iff \exists y : P(x,y)$$

Hierbei ist P ein entscheidbares, 2-stelliges Prädikat. Das heißt, es gibt einen immer-stoppenden Algorithmus M mit:



Beweis: (Von rechts nach links)

Gegeben sei ein solches entscheidbares Prädikat P bzw. der Algorithmus zur Berechnung von P.

Dann ist Folgendes ein Semi-Entscheidungsverfahren für A:

Bei Eingabe x,
überprüfe für $y = 1, 2, 3, 4, \ldots$
ob $P(x, y)$ gilt.
Falls ein solches y gefunden wird,
stoppe und gib 1 aus.

(Von links nach rechts)

Gegeben sei ein Semi-Entscheidungsalgorithmus S für A. Das heißt, bei Eingabe x stoppt S genau dann, wenn $x \in A$, sonst nicht.

Definiere ein 2-stelliges Prädikat $P(x, y)$ wie folgt: $P(x, y)$ gilt gdw. S bei Eingabe x stoppt in $\leq y$ Schritten

Offensichtlich ist P entscheidbar und es gilt:

$$x \in A \iff \exists y : P(x, y)$$

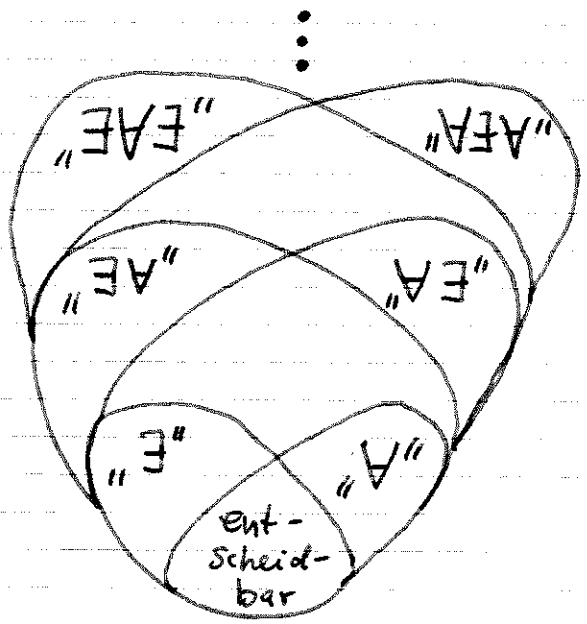
□

Bemerkung (nicht im Skript):

Wenn wir die semi-entscheidbaren Sprachen kurz durch „$\exists$" bezeichnen, dann kann man entsprechend die co-semi-entscheidbaren Sprachen (also diejenigen, deren Komplemente semi-entscheidbar sind) mit „$\forall$" bezeichnen, denn:

$$\neg\, \exists y : P(x,y) \iff \forall y : \underbrace{\overline{P}(x,y)}_{\text{ebenfalls entscheidbar.}}$$

Solcherart kann man eine menschliche Hierarchie von Sprachen erhalben:



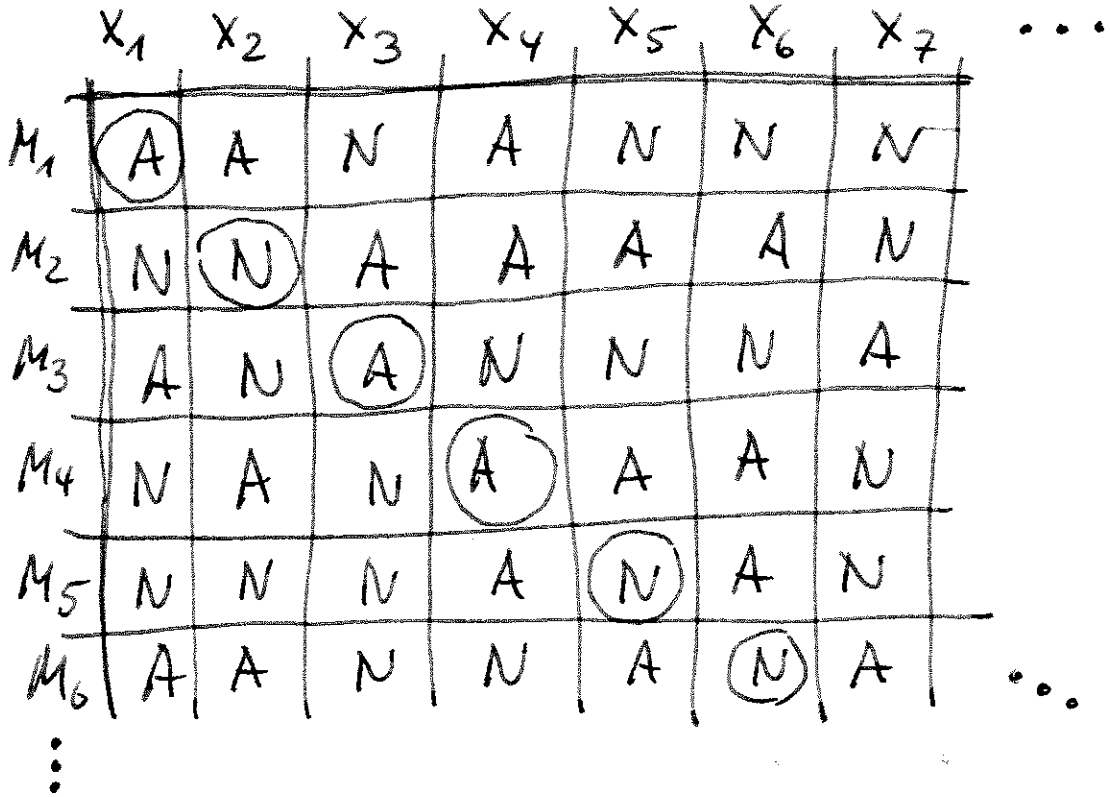
die so genannte arithmetische Hierarchie.

Aufgabe: Wir erinnern, dass gilt:

A ist semi-entscheidbar gdw. $A = T(M)$ für eine Turingmaschine M.

Ähnlich wie bei dem Beweis, dass R überabzählbar ist, zeige dass es eine Sprache D gibt, die nicht semi-entscheidbar ist. (Das zugrunde liegende Alphabet sei $\Sigma = \{0,1\}$.)

Antwort: Die Menge aller Turingmaschinen (über dem Eingabealphabet Σ) kann abgezählt werden, da eine Turingmaschine durch einen endlichen Text beschrieben werden kann. Sei also $M_1, M_2, M_3, \ldots$ eine Aufzählung aller dieser Turingmaschinen und sei $A_1 = T(M_1)$, $A_2 = T(M_2)$, $A_3 = T(M_3), \ldots$ die Menge aller semi-entscheidbaren Mengen. Ferner sei $\Sigma^* = \{\varepsilon, x_1, x_2, x_3, x_4, \ldots\}$. Nun kann man in einer unendlichen Matrix beschreiben, ob Turingmaschine M_i die Eingabe x_j akzeptiert (A) oder nicht akzeptiert (N):



Entlang der Diagonalen definieren wir nun die gesuchte „Diagonalsprache" $D = \{x_i \mid x_i \notin T(M_i)\}$. Es werden also genau diejenigen Wörter in die Sprache D aufgenommen, bei denen in der Diagonale ein N steht.

D kann nicht semi-entscheidbar sein. Wenn doch, dann gäbe es einen Index i, so dass $D = T(M_i)$. Bei der Eingabe x_i kann dies nicht stimmen, denn wegen der Def. von D gilt:

$$x_i \in T(M_i) \Leftrightarrow x_i \in D$$

Widerspruch. ∎

Mehrband-TM kann ihre verschiedenen Bänder so wie Integer-Variablen (in Binärdarstellung) verwenden. Man kann Programme angeben, die einfache Rechenoperationen durchführen, etwa:

$$x := x+1$$

$$x := 2 \cdot x$$

$$x := y$$

$$x := c \qquad (c \text{ Konstante})$$

Mittels Schleifen kann man z.B. auch die Addition (und ähnlich auch die Multiplikation) simulieren:

```
z: Teste, ob x den Wert 0 hat,
                wenn ja: springe nach z'
   x := x - 1
   y := y + 1
   Springe zu Zustand z
z':
```

Die Werte der Variablen x und y werden addiert. Das Ergebnis ist danach in der Variablen y.

Formale Definition eines Berechenbarkeitsbegriffs mit Hilfe einer einfachen Programmiersprache namens GOTO:

Eine Funktion $f: \mathbb{N}^k \to \mathbb{N}$ heißt <u>GOTO-berechenbar</u>, falls es ein GOTO-Programm (siehe unten) gibt, das f berechnet, und zwar stehen zu Beginn die Werte $n_1, \ldots, n_k$ in den Variablen $x_1, \ldots, x_k$ sowie 0 in den restlichen Variablen (insgesamt haben wir: die Variablen $x_0, x_1, \ldots, x_k, x_{k+1}, \ldots, x_m$). Sofern $f(n_1, \ldots, n_k) = n_0$, genau dann stoppt das Programm und n_0 steht in der Variablen x_0.

Ein <u>GOTO-Programm</u> sieht folgendermaßen aus: Marken gefolgt von jeweils einer Anweisung:

$$M_1 : A_1$$
$$M_2 : A_2$$
$$\vdots$$
$$M_l : A_l$$

Als Anweisung A_i ist zugelassen:

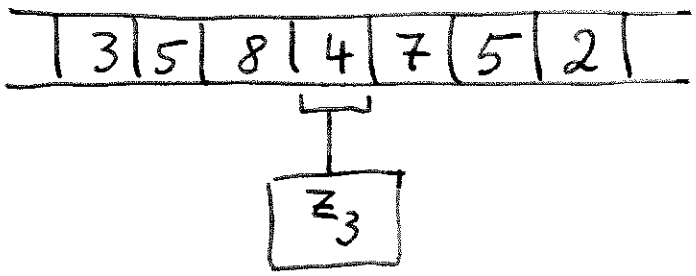
Wertzuweisungen:	$x_i := x_j \pm c$
unbedingter Sprung:	GOTO M_i
bedingter Sprung:	if $x_i = c$ then GOTO M_j
Stoppanweisung:	HALT

Die obige Diskussion zeigt, dass Mehrband-TMen GOTO-Programme simulieren können. Daher haben wir folgende gegenseitige Simulation – bis jetzt:

Wir machen als Nächstes plausibel, dass:

Zunächst beobachten wir, dass man mittels GOTO-Programmen außer der Addition auch Multiplikation $x \cdot y$, Exponentiation: x^y, ganzzahlige Division: x div y, Rest bei der Division: x mod y, Binomialkoeff. $\binom{x}{y}$, Fakultät $x!$ und viele andere berechnen kann.

Gegeben sei eine 1-Band-TM, die eine Funktion berechnet oder eine Sprache (semi-) entscheidet. Wir betrachten eine Konfiguration im Verlauf einer Rechnung. Der Einfachheit halber nehmen wir an: $\Sigma = \{1, 2, \ldots, 9\}$



Das GOTO-Programm simuliert Konfigurationen mit Hilfe dreier Variablen x, y, z. In diesem Fall ist $x = 3584$ (Bandinhalt links bis zum Schreib-Lesekopf)

$y = 257$ (Bandinhalt rechts – in umgedrehter Reihenfolge)

$z = 3$ (Nummer des aktuellen Zustands)

Wird nun ein Konfigurationsübergang ausgeführt mittels $\delta(z_3, 4) = (z_7, 9, R)$

so führt das GOTO-Programm folgende

Anweisungen durch:

$u := x \bmod 10$

if $z=3$ and $u=4$ then

$$\left[\begin{array}{l} z := 7 \\ x := 10 * (x^{+5}) + (y \text{ MOD } 10) \\ y := y \text{ div } 10 \end{array}\right]$$

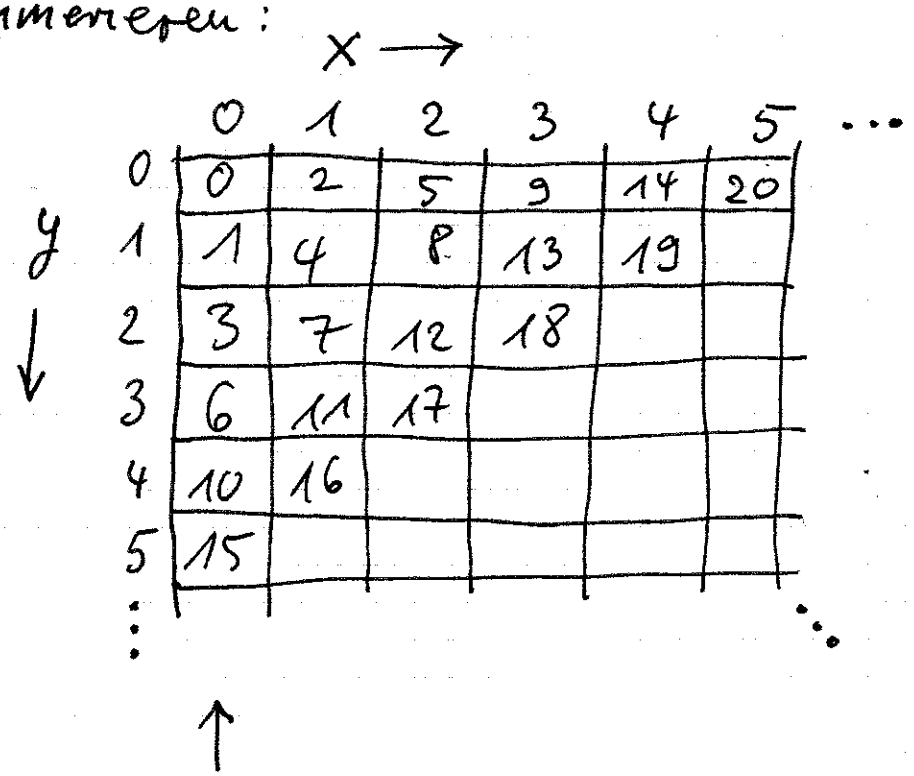
In dieser Weise kann der gesamte Rechenablauf einer TM durch ein GOTO-Programm nachempfunden werden.

Vorschau: WHILE-Programme (strukturiert)
Sind ebenfalls äquivalent zu GOTO.

Aufgabe: Im späteren Verlauf der Vorlesung wird eine berechenbare Funktion benötigt, die $\mathbb{N}^2$ bijektiv auf $\mathbb{N}$ abbildet. Gib eine solche Fkt. $c: \mathbb{N}^2 \to \mathbb{N}$ an!

(Wir wollen fortan mit $\mathbb{N}$ die Menge der natürlichen Zahlen einschließlich der 0 verstehen: $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$)

Antwort: Gemäß des „ersten Cantorschen Diagonalverfahrens" läßt sich $\mathbb{N}^2$ wie folgt durchnummerieren:



$c(x,y)$ sind die Einträge in der Tabelle

↑ der y-te Eintrag in dieser Spalte ($y = 0, 1, 2, \ldots$)

ist $\sum_{i=0}^{y} i = \frac{y(y+1)}{2}$

Dieser Eintrag bei $(0,y)$ befindet sich am Anfang der y-ten Diagonalen.
Der Eintrag (x,y) befindet sich an x-ter Stelle in der $(x+y)$-ten Diagonalen.

Somit:

$$c(x,y) = \sum_{i=0}^{x+y} i + x = \frac{(x+y)(x+y+1)}{2} + x$$

Die Berechnung von $c(x,y)$ setzt sich zusammen aus Funktionen wie Addition, Multiplikation, Division durch 2, die klar berechenbar sind.

Bemerkung: Auch die Umkehrfunktionen von $c(x,y)$ sind berechenbar, also $f(c(x,y)) = x$, $g(c(x,y)) = y$ bzw: $c(f(n), g(n)) = n$.

Da c bijektiv ist, kann man f so berechnen:

$f(n)$ = diejenige Zahl $x \leq n$, die zusammen mit einer Zahl $y \leq n$ ergibt, dass $c(x,y) = n$

$g(n)$ dito.

WHILE-Berechenbarkeit:

Eine Funktion $f: \mathbb{N}^k \to \mathbb{N}$ heißt **WHILE-berechenbar**, falls es ein WHILE-Programm gibt (Def. siehe unten) mit Programmvariablen

$x_0, x_1, \ldots, x_k, x_{k+1}, \ldots, x_n \quad (n \geq k)$

so dass zur Berechnung von $f(n_1, \ldots, n_k)$ die Variablen $x_1, \ldots, x_k$ mit den Werten $n_1, \ldots, n_k$ (und die restlichen Variablen mit 0) gestartet werden, und sofern $f(n_1, \ldots, n_k)$ definiert ist und den Wert m hat, so stoppt das WHILE-Programm mit dem Wert m in der Variablen x_0.

Hierbei sind WHILE-Programme induktiv wie folgt definiert:

- Jede Wertzuweisung $x_i := x_j \pm c$ ist ein WHILE-Programm
- Ist P ein WHILE-Programm, so ist auch IF $x_i = c$ THEN P END ein WHILE-Programm.

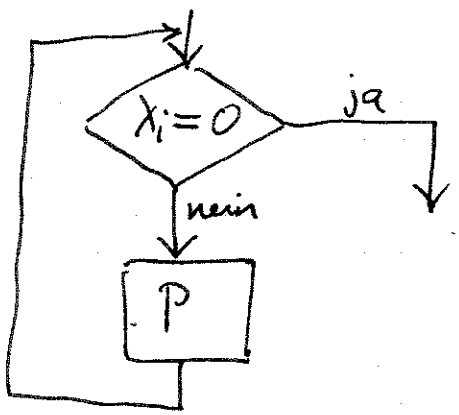
- Sind P, Q WHILE-Programme, dann ist auch P;Q ein WHILE-Programm.
- Ist P ein WHILE-Programm, dann ist auch WHILE $x_i \neq 0$ DO P END ein WHILE-Programm.

Hiermit haben wir die Syntax von WHILE-Programmen definiert. Da diese Programme in klar definierter Weise Rechnungen vollziehen sollen, müssen wir auch über die Semantik (die Art der Ausführung von solchen Programmen) reden.

- Bei einer Wertzuweisung $x_i := x_j \pm c$ wird der Wert der Variablen x_i neu überschrieben, nachdem $x_j \pm c$ berechnet wurde (es kann auch $i=j$ sein).

- Bei IF $x_i = c$ THEN P END wird zuerst überprüft, ob der aktuelle Wert von x_i gleich c ist. Wenn ja, wird P ausgeführt.

- Wenn das Programm die Form $P;Q$ hat, dann wird zuerst P ausgeführt, dann auf den Variablen-Werten, die P hinterlässt, wird Q ausgeführt.
- Bei WHILE $x_i \neq 0$ DO P END wird P ~~solange~~ immer wieder ausgeführt, solange x_i einen Wert $\neq 0$ hat, wie folgendes Flussdiagramm illustriert:



GOTO-Programme können WHILE-Programme simulieren. Klar sollte das sein in Bezug auf:

- Wertzuweisungen
- IF-Anweisungen
- Hintereinanderausführung zweier Programme bei $P;Q$

Am Interessantesten (aber ebenso einfach) ist die Simulation einer WHILE-Schleife durch ein GOTO-Programm:

WHILE $x_i \neq 0$ DO P END

$\Downarrow$

M: IF $x_i = 0$ THEN GOTO M'

(Simulation von P)

GOTO M

M':

Nun wird gezeigt, dass jedes GOTO-Programm durch ein WHILE-Programm simuliert werden kann.
Gegeben sei ein GOTO-Programm:

M_1 : A_1

M_2 : A_2

$\vdots$

M_n : A_n

wobei die A_i sein können:

- Wertzuweisung: $x_i := x_j \pm c$
- Sprung: GOTO M_i
- bedingter Sprung: IF $x_i = c$ THEN GOTO M_j
- HALT-Anweisung: HALT

Das simulierende WHILE-Programm verwendet eine besondere Variable z. Der Wert von z ist die Nummer der GOTO-Anweisung, die aktuell auszuführen ist. Zu Beginn wird z auf 1 gesetzt. Bei der HALT-Anweisung wird z auf 0 gesetzt. Das GOTO-Programm wird wie folgt simuliert:

```
z := 1;
WHILE z ≠ 0 DO
    if z=1 then A'1 END;
    if z=2 then A'2 END;
        ⋮
    if z=n then A'n END
END
```

Hierbei gilt:

$$A_i' = \begin{cases} X_j := X_\ell \pm c \,;\; Z := Z+1 & \text{falls } A_i = X_j := X_\ell \pm c \\ Z := k & \text{falls } A_i = \text{GOTO } M_k \\ \hline Z := Z+1; \\ \text{if } X_j = c \text{ then } Z := k & \text{falls } A_i = \text{if } X_j = c \text{ then GOTO } M_k \\ Z := 0 & \text{falls } A_i = \text{HALT} \end{cases}$$

Bei dieser Simulation von GOTO nach WHILE entsteht nur <u>eine einzige</u> While-Schleife.

Wir fassen zusammen: Folgende Berechenbarkeitskonzepte sind zueinander äquivalent:

1-Band-TM $\Leftrightarrow$ Mehrband-TM $\Leftrightarrow$ GOTO $\Leftrightarrow$ WHILE

Bzgl. der gegenseitigen Simulation von GOTO und WHILE ergibt sich: Jedes WHILE-Programm kann so umgeformt werden, dass es nur noch aus 1 WHILE-Schleife besteht.

Gegeben sei ein WHILE-Programm, das > 1 WHILE-Schleifen enthält. Man forme das WHILE-Programm um in ein äquivalentes GOTO-Programm, dann wieder zurück von GOTO nach WHILE. Am Ende hat man ein äquivalentes WHILE-Programm, das nur noch 1 WHILE-Schleife enthält.

Nun eine Abschweifung:

Wir schränken die „Berechnungsfähigkeit" von WHILE-Programmen ein: von WHILE-Schleifen zu LOOP-Schleifen: (WHILE ist nicht mehr zugelassen):

Syntax: LOOP x_i DO P END

Semantik: Wiederhole das Programm P soooft, wie oft der Wert von x_i zu Beginn angibt:

$\underbrace{P_j P_j P_j \cdots_j P}_{\text{(Wert von } x_i\text{)-mal}}$

Also nochmals zusammen gefasst:

LOOP-Berechenbarkeit wird definiert durch:

- Wertzuweisung
- IF-Anweisung
- Hintereinanderausführung: P;Q
- LOOP-Schleife

Tatsächlich könnte man nachträglich die IF-Anweisung wieder fallen lassen (der Einfachheit wegen):

Statt `IF X=0 THEN P END`

kann simuliert werden durch:

```
y:=1;
LOOP x DO y:=0; END;
LOOP y DO P END
```

Die Besonderheit von LOOP-Programmen: sie Stoppen immer!

Daher gilt:

f LOOP-berechenbar $\Rightarrow$ f ist total + (WHILE, GOTO, etc.)-berechenbar

Gilt auch die Umkehrung?

Antwort: Nein

Das heißt, es gibt totale + berechenbare Funktionen, die nicht LOOP-berechenbar sind. Ein solches Beispiel ist die Ackermann-Funktion $a: \mathbb{N}^2 \to \mathbb{N}$:

$$a(0,y) = y+1$$

$$a(x,0) = a(x-1,1) \quad \text{falls } x>0$$

$$a(x,y) = a(x-1, a(x,y-1)) \quad \text{falls } xy>0$$

Lemma: $a(1,y) = y+2$

Beweis durch Induktion nach y.

$y=0$: $a(1,0) = a(0,1) = 1+1 = 2$ ✓

$y \to y+1$: $a(1,y+1) = a(0, a(1,y)) = a(1,y)+1 = y+2+1$ ✓

Lemma: $a(2,y) = 2y+3$

Beweis → Übung

Lemma: $a(3,y) = 2^{y+3} - 3$

Beweis: $(y=0)$ $a(3,0) = a(2,1) = 2\cdot 1 + 3 = 5$

$2^{0+3} - 3 = 8 - 3 = 5$ } ✓

$(y \to y+1)$ $a(3,y+1) = a(2, a(3,y)) = 2 \cdot a(3,y) + 3$

$= 2 \cdot (2^{y+3} - 3) + 3$

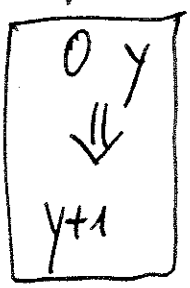
$= 2^{y+4} - 6 + 3 = 2^{(y+1)+3} - 3$ ✓

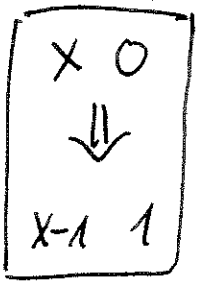
Beispiel: Berechnung von $a(2,2)$.

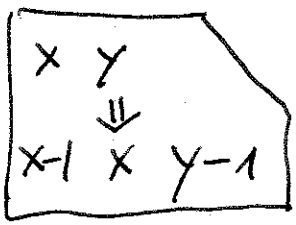
Eine Zahlenfolge $n_1 n_2 \dots n_k$ stellt $a(n_1, a(n_2, \dots, a(n_{k-1}, n_k)\dots))$ dar:

$2\,2 = 121$
$= 1120$
$= 1111$
$= 11010$
$= 11001$
$= 1102$
$= 113$
$= 1012$
$= 10011$
$= 100010$
$= 100001$
$= 10002$
$= 1003$
$= 104$
$= 15$
$= 0414$
$= 0013$
$= 00012$
$= 000011$
$= 0000010$
$= 00000001$
$= 000002$
$= 00003$
$= 0004$
$= 005$
$= 006 = 7$ also $a(2,2) = 7$

Regeln:







Die Ackermann-Funktion $a : \mathbb{N}^2 \to \mathbb{N}$

$$a(0, y) = y + 1 \qquad (1)$$

$$a(x, 0) = a(x-1, 1) \qquad (2)$$

$$a(x, y) = a(x-1, a(x, y-1)) \qquad (3)$$

ist <u>wohldefiniert</u>, d.h. für alle $x, y \in \mathbb{N}$ endet die Anwendung der Definition (1), (2), (3) nach endlich vielen Schritten. (Also ist a eine <u>totale Fkt.</u>)

<u>Beweis</u> durch Induktion nach x:

($x = 0$) Wegen (1) ist $a(0, y)$ für alle y wohldefiniert, nämlich $= y + 1$.

($X \to X+1$) Falls $y = 0$ so ist wegen (2):

$a(x+1, 0) = a(x, 1)$ ... wohldefiniert nach I.V.

Falls $y > 0$, so gilt mit (3), wiederholt:

$$\begin{aligned} a(x+1, y) &= a(x, a(x+1, y-1)) \\ &= a(x, a(x, a(x+1, y-2))) \\ &\vdots \\ &= a(x, a(x, \ldots, a(x+1, 0) \cdots)) \\ &= a(x, a(x, \cdots, a(x, 1) \cdots)) \end{aligned}$$

(2)

Jeder dieser $a(x, \dots)$ - Berechnungen ist nach I.V. wohldefiniert. Somit ist $a(x+1, y)$ wohldefiniert.

Lemma A: $y < a(x, y)$.

Beweis durch Induktion nach x.

$(x=0)$ $y < y+1 \underset{(1)}{=} a(0, y)$ ✓

$(x \to x+1)$ Nach IV gelte $y < a(x, y)$ für alle y.

Zu zeigen: $y < a(x+1, y)$ für alle y.

Dies zeigen wir per Induktion nach y:

$(y=0)$ $0 < \underbrace{1 < a(x, 1)}_{\text{nach IV für } x \text{ mit } y=1} \underset{(2)}{=} a(x+1, 0)$ ✓

$(y \to y+1)$ Nach IV gelte $y < a(x+1, y)$.

Zu zeigen ist $y+1 < a(x+1, y+1)$.

Nach IV für x mit $a(x+1, y)$ für y ein und erhalten: $a(x+1, y) < a(x, a(x+1, y))$

$\underset{(3)}{=} a(x+1, y+1)$. Nach IV bzgl y gilt $y < a(x+1, y)$. Eingesetzt:

$y < a(x+1, y) < a(x+1, y+1) \Rightarrow y+1 < a(x+1, y+1)$. ✓

Lemma B: $a(x,y) < a(x,y+1)$

Beweis:

Falls $x=0$: $a(0,y) \underset{(1)}{=} y+1 < y+2 \underset{(1)}{=} a(0,y+1)$ ✓

Falls $x>0$: Lemma A mit $x-1$ und $a(x,y)$ eingesetzt:

$$a(x,y) < a(x-1, a(x,y)) \underset{(3)}{=} a(x,y+1)$$ ✓

Lemma C: $a(x,y+1) \leq a(x+1,y)$

Beweis: Induktion nach y:

$(y=0)$ $a(x,0+1) \underset{(2)}{=} a(x+1,0)$ ✓

$(y \to y+1)$ Wegen Lemma A ist $y+1 < a(x,y+1)$

Es folgt: $y+2 \leq a(x,y+1) \underset{(IV)}{\leq} a(x+1,y)$.

Somit: $a(x,y+2) \underset{\text{Lemma B und obige Zeile}}{\leq} a(x,a(x+1,y)) \underset{(3)}{=} a(x+1,y+1)$ ✓

Lemma D: $a(x,y) < a(x+1,y)$

Beweis: $a(x,y) < a(x,y+1) \leq a(x+1,y)$

(Lemma B) (Lemma C)

Nun haben wir also strenge Monotonie im 1. und im 2. Argument.

Ordne jedem LOOP-Programm P (egal wieviel-stellig die hierbei berechnete Funktion ist) eine Funktion $f_P : \mathbb{N} \to \mathbb{N}$ zu.

n_i ... Anfangswerte der Programvariablen

n_i' ... Endwerte der Programvariablen nach Ablauf von P

$$(n_0, \ldots, n_k) \xrightarrow{P} (n_0', \ldots, n_k')$$

$$f_P(n) := \max \left\{ \sum_{i=0}^{k} n_i' \;\middle|\; \sum_{i=0}^{k} n_i \leq n \right\}$$

... größtmögliche Summe der erreichbaren Variablenwerte, wenn P gestartet wird mit Anfangswerten, deren Summe $\leq n$ ist.

<u>Lemma E</u>: Für jedes LOOP-Programm P gibt es eine Konstante k, so dass für alle n:

$$f_P(n) < a(k, n)$$

Beweis: Durch Induktion über den Aufbau des LOOP-Programms P.

Man muss folgende Fälle unterscheiden:

- P besteht aus einer Wertzuweisung
- P hat die Form $P = Q;R$ wobei auf Q und R die I.V. anwendbar ist.
- P hat die Form LOOP x_i DO Q END wobei auf Q die I.V. anwendbar ist.

Im ersten Fall, der Wertzuweisung, kann man sich auf $x_i := x_j \pm 1$ beschränken (ansonsten mehrere Wertzuweisungen verwenden). Dann gilt:

$$f_P(n) = 2n+1 < a(2,n) = 2n+3 \quad \checkmark$$

P habe die Form $Q;R$ und es gebe nach I.V. Konstanten k_1, k_2 so dass $\forall n$:

$$f_Q(n) < a(k_1,n) \qquad f_R(n) < a(k_2,n)$$

Es folgt (mehrfach: Monotonie):

$$
\begin{aligned}
f_P(n) &\leq f_R(f_Q(n)) \\
&< a(k_2, f_Q(n)) \\
&< a(k_2, a(k_1, n)) \\
&\leq a(l, a(l+1, n)) \quad , l = \max(k_1, k_2) \\
&\underset{(3)}{=} a(l+1, n+1) \\
&\leq a(l+2, n) \qquad \text{Lemma B}
\end{aligned}
$$

Falls P die Form hat LOOP x_i DO Q END
Dann gibt es eine Konstante k so dass

$$\forall n: \quad f_Q(n) < a(k, n)$$

Wir können OBdA annehmen, dass x_i in Q ~~keine Zuweisung erhält~~ nicht vorkommt. Bei der Maximumsbildung in der Definition von $f_P(n)$ sei m die optimale Wahl für Variablenwert x_i. Also ist m die Anzahl der Schleifendurchläufe der LOOP-Schleife. Ferner können wir x_i bei der Betrachtung von Q weglassen: $f_Q(n) \leq f_Q(n-m)+m$

Somit erhalten wir (sei $m>0$):

$$f_P(n) \leq \underbrace{f_Q(f_Q(\ldots f_Q}_{m\text{-mal}}(n-m)\ldots)) + m$$

$$\underbrace{< \ldots <}_{m\text{-mal}}\ a(k, a(k, \ldots, a(k, n-m)\ldots)) + m$$

Es folgt: $f_P(n) \leq a(\underbrace{k, a(k, \ldots, a(k}_{m\text{-mal}}, n-m)\ldots))$

$$\overset{(3)}{=} a(k+1, n-1)$$

$$< a(k+1, n) \quad \checkmark$$

(Bem: Der Fall $m=0$: $f_P(n) = n < n+1 = a(0,n)$)

Satz: Die Ackermann-Funktion ist nicht LOOP-berechenbar.

Beweis: Angenommen a ist LOOP-berechenbar ~~durch ein Programm P~~. Dann ist auch $g(n) = a(n,n)$ LOOP-berechenbar durch ein LOOP-Programm P. Es gilt ~~für alle~~ $g(n) \leq f_P(n)$.

Nach Lemma E gibt es Konstante k, so dass

$$\forall n: \quad f_P(n) < a(k,n)$$

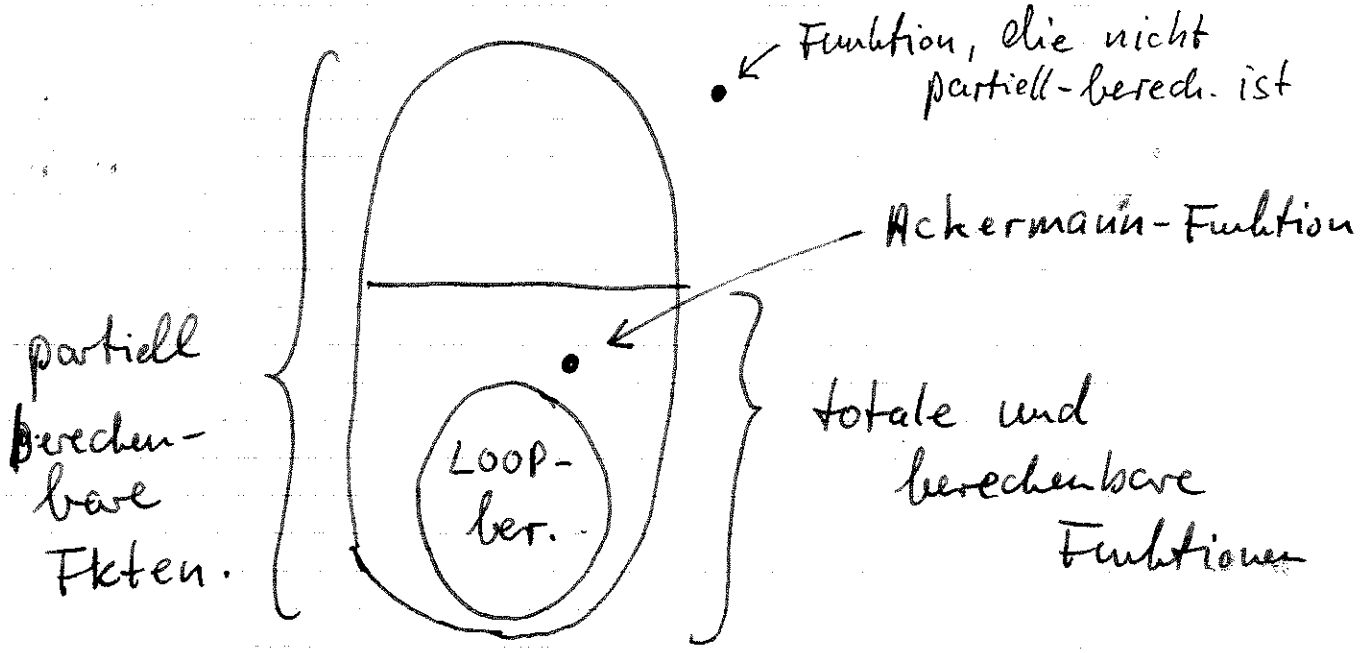
Wenn man $n=k$ einsetzt, ergibt sich:

$$g(k) \leq f_p(k) < a(k,k) = g(k)$$

Dies ist ein Widerspruch.

Also ist a nicht LOOP-berechenbar. □

Skizze:



Vorschau:

f in Polynomialzeit berechenbar $\Rightarrow$ f LOOP-berechenbar.

Zwischenspiel:

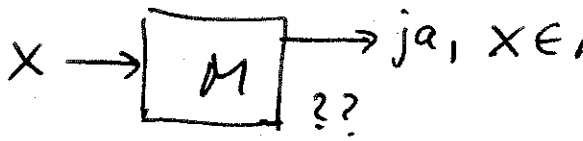
Rekursive Aufzählbarkeit.

$A \subseteq \mathbb{N}$ (oder $A \subseteq \Sigma^*$) heißt <u>rekursiv aufzählbar</u>,
wenn $A = \emptyset$, oder
wenn es totale + berechenbare Fkt f gibt,
so dass A der Wertebereich von f ist
("A wird von f aufgezählt")

$$A = \{ f(0), f(1), f(2), \ldots \}$$

<u>Satz:</u> A ist rekursiv aufzählbar
gdw.
A ist semi-entscheidbar.

<u>Beweis:</u> ($\Leftarrow$) Für einen Algorithmus M gelte:



Sei $A \neq \emptyset$. Dann gibt es ein $a \in A$. Wir fixieren ein solches Element a aus A.

Folgender (immer stoppender) Algorithmus M' berechnet eine totale Funktion f:

Eingabe n

Interpretiere n als $n = c(x,y)$

Sei also $x := d(n)$; $y = e(n)$

(wobei d, e die Umkehrfkt. von c sind)

Simuliere für x Schritte eine

Rechng von M bei Eingabe y.

Wenn M dabei stoppt und „ja"

ausgibt, so output(y)

Andernfalls: output(a)

Nun ist klar, dass die Ausgabewerte dieses Algorithmus bei Eingabe $n = 0, 1, 2, \ldots$ genau die Elemente von A sind.

($\Rightarrow$) A werde durch die Funktion f aufgezählt, also $A = W(f)$.

Ein Semi-Entscheidungsverfahren für A lautet:

Eingabe: x

for $n := 0, 1, 2, \ldots$ do

if $f(n) = x$ then output „ja".

□

Aufgabe: Die Menge $LOOP_k$, $k \in \mathbb{N}$, enthält alle LOOP-berechenbaren Funktionen, die sich unter Verwenden von $\leq k$ LOOP-Schleifen berechnen lassen.

Man zeige: $add: \mathbb{N}^2 \to \mathbb{N}$ ist $LOOP_1$-berechenbar

$mult: \mathbb{N}^2 \to \mathbb{N}$ ist $LOOP_2$-berechenbar, aber nicht $LOOP_1$-berechenbar.

Antwort: add ist $LOOP_1$-berechenbar:

(Die Eingabe erfolgt in x_1, x_2, die Ausgabe durch x_0).

LOOP x_1 DO $x_2 := x_2 + 1$ END;

$x_0 := x_2$

mult ist $LOOP_2$-berechenbar:

LOOP x_1 DO $\underbrace{x_0 := x_0 + x_2}$ END

Berechnung mit Hilfe eines $LOOP_1$-Programms

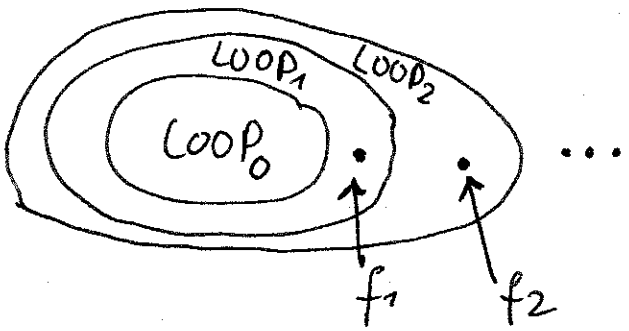
mult ist nicht $LOOP_1$-berechenbar (informal):

$LOOP_1$-Programme können nur Funktionen berechnen deren Ausgabe höchstens linear mit der Eingabe wächst. Allerdings: $mult(x,x) = x^2$

Man kann zeigen, dass:

$$LOOP_0 \subsetneq LOOP_1 \subsetneq LOOP_2 \subsetneq LOOP_3 \subsetneq \cdots \bigcup_k LOOP_k = LOOP$$

Sei $f_1, f_2, f_3, \ldots$ eine Folge von einstelligen Funktionen mit $f_k \in LOOP_k \setminus LOOP_{k-1}$.



Aufgabe: Man zeige, dass $g(k,n) := f_k(n)$ nicht LOOP-berechenbar sein kann.

Antwort: Angenommen, g wäre doch LOOP-berechenbar. Dann gibt es eine Zahl $l \in \mathbb{N}$, so dass g $LOOP_l$-berechenbar ist. Dann ist auch $h(n) := g(l+1, n)$ $LOOP_l$-berechenbar, denn h entsteht aus g mittels Einsetzung der Konstanten $l+1$ in die Funktion g und benötigt keine weitere LOOP-Schleife. Nun ist aber $h = f_{l+1}$ und kann nicht $LOOP_l$-berechenbar sein. Widerspruch. □

Die primitiv-rekursiven Funktionen

Dies ist eine induktiv definierte Menge von Funktionen (von $\mathbb{N}^k$ nach $\mathbb{N}$):

(1) Alle Konstanten Funktionen sind p.r.
(Beispiel: $(x,y,z) \mapsto 5$ ist p.r.)

(2) Alle identischen Abbildungen (Projektionen) sind p.r.
(Beispiel: $(x,y) \mapsto y$ ist p.r.)

(3) Die Nachfolgerfunktion $S(n) = n+1$ ist p.r.

(4) Jede Funktion, die durch Einsetzung (Komposition) von p.r. Funktionen darstellbar ist, ist selber p.r.
(Beispiel: Wenn $f: \mathbb{N} \to \mathbb{N}$ und $g: \mathbb{N}^2 \to \mathbb{N}$ p.r. sind, dann auch:

$$f(f(x)),\ g(f(x), f(y)),$$
$$f(g(f(x), x)),\ g(g(x,y), g(u,v)),$$
usw.)

(5) Jede Funktion f, die sich durch primitive Rekursion definieren lässt, basierend auf Funktionen g und h, die bereits p.r. sind, ist auch p.r.

Dies bedeutet Folgendes:

$$f(0, \dots) = g(\dots)$$

$$f(n+1, \dots) = h(f(n, \dots), n, \dots)$$

Die Funktionen gemäß (1), (2), (3) heißen auch die Basisfunktionen.

Bem: Eine Funktion f, die die Gleichungen gemäß primitiver Rekursion erfüllt, ist eindeutig definiert (Induktion nach n) + total.

Beispiele von prim. rek. Funktionen:

Die Additionsfunktion $add : \mathbb{N}^2 \to \mathbb{N}$:

$$add(0, x) = \underbrace{x}_{\text{identische Abbildung, p.r. nach Def. (2)}}$$

$$add(n+1, x) = \underbrace{S}_{\text{die Nachfolgerfkt ist p.r. nach (3)}}(add(n, x))$$

Die Multiplikation:

$$mult(0, x) = \underbrace{0}_{\text{konstante Fkt ist p.r. nach (1)}}$$

$$mult(n+1, x) = add(mult(n, x), x)$$

Die Funktion $h(n) = \frac{n(n+1)}{2}$ ist prim. rek. denn:

$$h(0) = 0$$

$$h(n+1) = \frac{(n+1)(n+2)}{2} = \frac{n(n+1)}{2} + \frac{2(n+1)}{2}$$

$$= h(n) + n + 1$$

$$= \underbrace{S(add}_{\text{Einsetzung (4)}}(h(n), n))$$

Damit ergibt sich, dass auch die Paarungsfunktion $c(x,y) = \frac{(x+y)(x+y+1)}{2} + x$

primitiv rekursiv ist (mittels Einsetzung (4)):

$$c(x,y) = add(h(add(x,y)), x)$$

$c(x,y)$	$x \rightarrow$ 0	1	2	3	4
$y \downarrow$ 0	0	2	5	9	14
1	1	4	8	13	
2	3	7	12		
3	6	11			
4	10				

Die folgenden technischen Betrachtungen dienen dazu, nachzuweisen, dass die beiden Umkehrfunktionen $e, f : \mathbb{N} \to \mathbb{N}$ ebenfalls primitiv rekursiv sind.

Es gilt:

$$e(n) = \max\{x \leq n \mid \exists y \leq n : c(x,y) = n\}$$

$$f(n) = \max\{y \leq n \mid \underbrace{\exists x \leq n : c(x,y) = n}_{\text{beschränkter } \exists\text{-Quantor}}\}$$

beschränkter max-Operator

Sei $P(x)$ ein Prädikat ($\hat{=}$ Funktion, die nur die Funktionswerte 0 und 1 annehmen kann).

Die Funktion q entsteht durch Anwendung des <u>beschränkten max-Operators</u> aus P und ist wie folgt definiert:

$$q(n) = \max \{ x \leq n \mid P(x) \}$$

Falls kein $x \leq n$ mit $P(x)$ existiert, so setzen wir $q(n) = 0$.

Man kann q durch primitive Rekursion definieren:

$$q(0) = 0$$

$$q(n+1) = \begin{cases} q(n), & \text{falls } P(n+1) = 0 \\ n+1, & \text{falls } P(n+1) = 1 \end{cases}$$

$$= q(n) + P(n+1) * (n+1-q(n))$$

Somit: Wenn P primitiv rekursiv ist, dann ist es auch q.

Ähnlich: Das Prädikat R entsteht aus dem Prädikat P mittels <u>beschränktem Existenzquantor</u>, falls gilt:

$$R(n) \quad \text{gdw.} \quad \exists x \leq n : P(x)$$

Falls P primitiv rekursiv ist, so ist es auch das Prädikat R, denn man kann R mittels primitiver Rekursion, basierend auf P (und anderer bereits p.r. Funktionen) definieren:

$$R(0) = P(0)$$

$$R(n+1) = R(n) \vee P(n+1)$$

$$= P(n+1) + R(n) - P(n+1) * R(n)$$

Die Funktion $c : \mathbb{N}^2 \to \mathbb{N}$ ist 2-stellig. Man kann jedoch auch Tripel, Quadrupel, usw. von nat. Zahlen codieren durch

$$c(x_1, c(x_2, x_3)),$$

$$c(x_1, c(x_2, c(x_3, x_4))), \quad \text{usw.}$$

Entsprechende Umkehrfunktionen kann man dann wie folgt berechnen:

$$x_1 = e(n), \quad x_2 = e(f(n)), \quad x_3 = f(f(n))$$

bzw. $x_1 = e(n)$, $x_2 = e(f(n))$, $x_3 = e(f(f(n)))$, $x_4 = f(f(f(n)))$ usw.

Notation: Um die in einem LOOP-Programm vorkommenden Werte $x_0, x_1, \ldots, x_k$ zu codieren, verwenden wir:

$$n = \langle x_0, x_1, \ldots, x_k \rangle = c(x_0, c(x_1, \ldots c(x_{k-1}, x_k)))$$

Für die Umkehrfunktionen:

$$x_0 = d_0(n) = e(n)$$

$$x_1 = d_1(n) = e(f(n))$$

$$\vdots$$

$$x_k = d_k(n) = \underbrace{f(f(\ldots f(n)\ldots))}_{k\text{-mal}}$$

Satz: Es gilt, dass $f: \mathbb{N}^l \to \mathbb{N}$ LOOP-berechenbar ist genau dann, wenn f primitiv rekursiv ist.

Beweis: Sei $f: \mathbb{N}^l \to \mathbb{N}$ LOOP-berechenbar. Dann gibt es ein LOOP-Programm P (mit Programmvariablen $x_0, x_1, \ldots, x_k$ $(k \geq l)$), das f berechnet.

Durch Induktion über den Aufbau von P zeigen, dass $g_P: \mathbb{N} \to \mathbb{N}$ primitiv rekursiv ist, wobei

$$g_P: n = \langle n_0, n_1, \ldots, n_k \rangle \xmapsto{P} \langle n_0', n_1', \ldots, n_k' \rangle = g_P(n)$$

Wenn dies gezeigt ist, dann folgt auch, dass f primitiv rekursiv ist, denn:

$$f(x_1, \ldots, x_l) = d_0(g_P(\langle 0, x_1, \ldots, x_l, 0, \ldots, 0 \rangle))$$

- Falls P die Form hat $x_i := x_j \pm c$ (Wertzuweisung) so gilt:

$$g_P(n) = \langle d_0(n), \ldots, d_{i-1}(n), d_j(n) \pm c, d_{i+1}(n), \ldots, d_k(n) \rangle$$

(Einsetzung nach (4))

- Falls P die Form hat $P = Q;R$ so existieren nach Ind. vor. entsprechende Funktionen g_Q und g_R. Somit gilt: $g_P(n) = \underbrace{g_R(g_Q(n))}_{\text{Einsetzung (4)}}$.

- Falls P die Form hat $P = \text{LOOP } x_i \text{ DO } Q \text{ END}$ so definiere zunächst per prim. Rekursion eine 2-stellige Funktion h:

$$h(0,n) = n$$
$$h(t+1,n) = g_Q(h(t,n))$$

Aus dieser Definition folgt, dass

$$h(t,n) = \underbrace{g_Q(g_Q(\cdots g_Q(n)\cdots))}_{t\text{-mal}}$$

Mit anderen Worten: Dies ist die Wirkung von $\underbrace{Q;Q;\cdots;Q}_{t\text{-mal}}$ auf die Programmvariablen $n = \langle x_0, x_1, \ldots, x_k \rangle$. Damit ergibt sich:

$$g_P(n) = h(d_i(n), n)$$

Beweis der Umkehrung (Induktion über dem Aufbau der prim. rekursiven Funktionen):

- Die Basisfunktionen gemäß (1), (2), (3) sind ~~primitiv rekursiv~~ LOOP-berechenbar.
- Falls eine P.r. Funktion durch Einsetzung aus anderen prim.rek. Funktionen (die nach Ind.vor. bereits LOOP-berechenbar sind) entsteht, so kann man die fragliche Funktion durch entsprechendes Hintereinanderausführen dieser Funktionen berechnen.
- Falls f durch eine primitive Rekursion definiert wird, also

$$f(0, \ldots) = g(\ldots)$$
$$f(n+1, \ldots) = h(f(n, \ldots), n, \ldots)$$

so können g und h nach Ind. vor. bereits durch LOOP-Programme berechnet werden.

Nun kann f durch ein LOOP-Programm folgendermaßen berechnet werden:

$y := g(\ldots)$; $m := 0$;

LOOP n DO $y := h(y, m, \ldots)$;

$m := m+1$ END.

□

Vorschau: Einen weiteren Operator, den μ-Operator, einführen, mit dessen Hilfe die Menge der prim. rek. Funktionen erweitert wird (es können auch partielle Funktionen entstehen).

Durch Erweiterung des obigen Beweises zeigen, dass μ-rekursiv dasselbe bedeutet wie WHILE-berechenbar.

μ-rekursive Funktionen (Kleene)

Zusätzlich zu den Definitionen bei primitiv rekursiv ein weiteres Definitionsprinzip für Funktionen hinzunehmen (den μ-Operator):

Sei $f: \mathbb{N}^{n+1} \to \mathbb{N}$ bereits primitiv (bzw. μ-) rekursiv. Dann ist auch folgende Funktion $g: \mathbb{N}^n \to \mathbb{N}$ μ-rekursiv:

$$g(\vec{x}) = \min \{ k \mid f(k, \vec{x}) = 0 \}$$

Sofern Minimum nicht existiert: undefiniert

Kurznotation: $g = \mu f$

Satz: f ist μ-rekursiv gdw. f ist WHILE-berechenbar.

Den Beweis von „primitiv-rekursiv ⇔ LOOP-berech." erweitern um den μ-Operator bzw. die WHILE-Schleife.

Es gelte $g = \mu f$. Nach Ind. vor. kann man f bereits durch ein LOOP (bzw. WHILE-) Programm

berechnen. Dann berechnet folgendes Programm die Funktion g:

$x_0 := 0$

$y := f(x_0, x_1, \ldots, x_n)$

WHILE $y \neq 0$ DO

$x_0 := x_0 + 1;$

$y := f(x_0, x_1, \ldots, x_n)$

END

Umkehrung: Sei P ein WHILE-Programm der Form

$$P = \text{WHILE } x_i \neq 0 \text{ DO } Q \text{ END}$$

wobei für Q bereits nach Ind. voraus. eine Darstellung als prim. rek. bzw. μ-rekursive Funktion existiert. Wie im Beweis von „prim.rek $\hat{=}$ LOOP" gibt es eine prim.rek. Funktion $h(n,x)$, die den Zustand der Programmvariablen $x = \langle x_0, \ldots, x_k \rangle$ nach n Ausführungen von Q wiedergibt. Wir setzen

$$g_P(x) = h(\underbrace{\mu(d_i h)(x)}, x)$$

minimale Anzahl von Q-Wiederholungen, bis $x_i = 0$, sonst undefiniert.

□

Kleene-Normalform für μ-rekursive Funktionen:

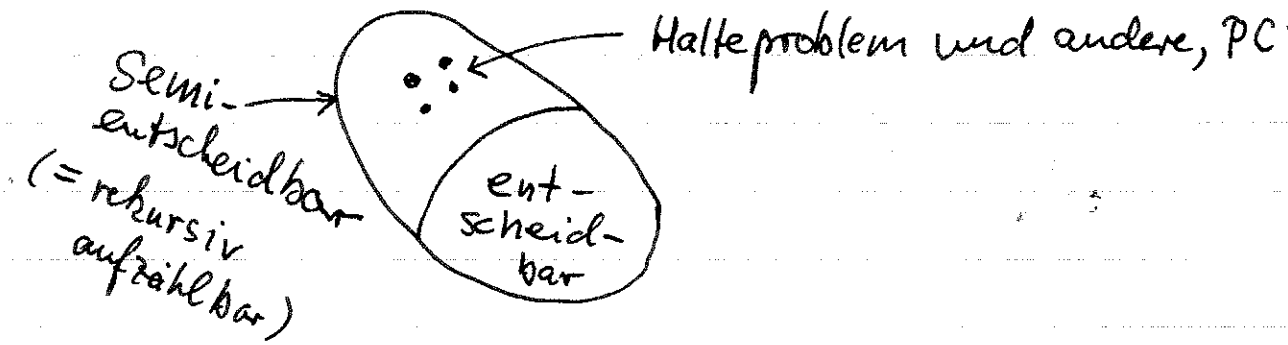
Für jede μ-rekursive Funktion f gibt es primitiv rekursive Funktionen p und q, so dass

$$f(\vec{x}) = p(\vec{x}, \mu q(\vec{x}))$$

Beweis: Die μ-rek. Funktion f ist WHILE-berechenbar. Man kann das While-Programm so umschreiben, dass nur 1 WHILE-Schleife vorhanden ist. Rück-Umformung in μ-rekursive Funktionsdarstellung liefert die angegebene Darstellung. □

Halteproblem, Unentscheidbarkeit

Ziel: einige unentscheidbare Probleme identifizieren. Es zeigt sich, dass diese Liste von Problemen semi-entscheidbar aber nicht entscheidbar ist!



Def: Das Halteproblem ist folgendes Entscheidungsproblem, dargestellt als Menge:

$$H = \{ (M, x) \mid M \text{ angesetzt auf } x \text{ hält} \}$$

Hierbei ist M ein Algorithmus, ein Berechnungsverfahren, z.B. eine Turingmaschine, While-Programm oder GOTO-Programm.

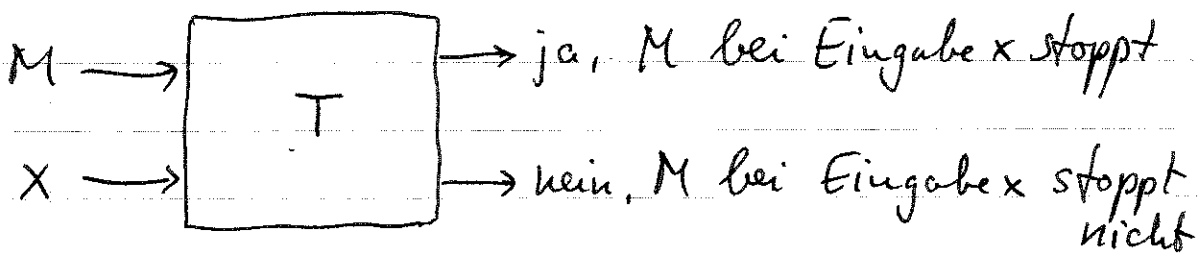
Satz: Das Halteproblem H ist nicht entscheidbar.

Beweis (indirekt):

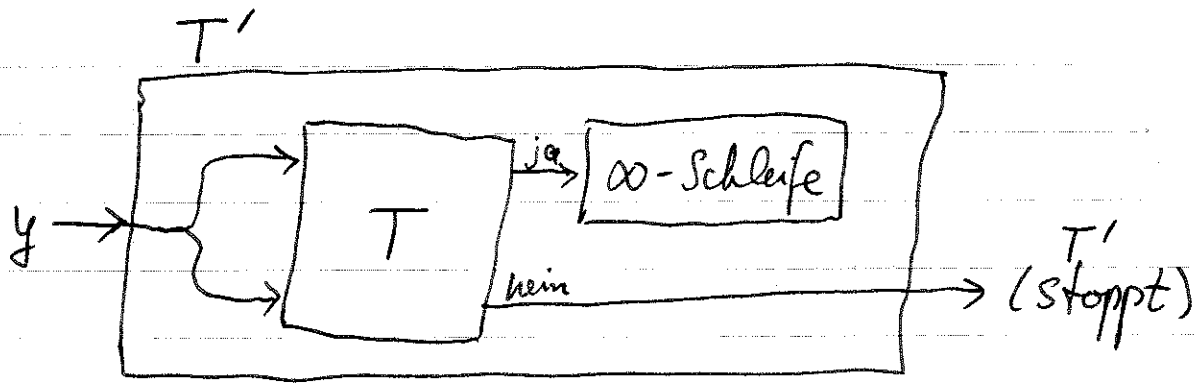
Angenommen, H wäre doch entscheidbar. Das heißt, diese totale Funktion ist berechenbar:

$$(M,x) \longmapsto \begin{cases} 1, & M \text{ auf Eingabe } x \text{ stoppt} \\ 0, & M \text{ auf Eingabe } x \text{ stoppt nicht.} \end{cases}$$

Bildhaft, gibt es also einen immer stoppenden Algorithmus T der folgenden Art:



Wir benutzen T innerhalb eines weiteren Algorithmus T':



Nun gilt:

T angesetzt auf T' hält

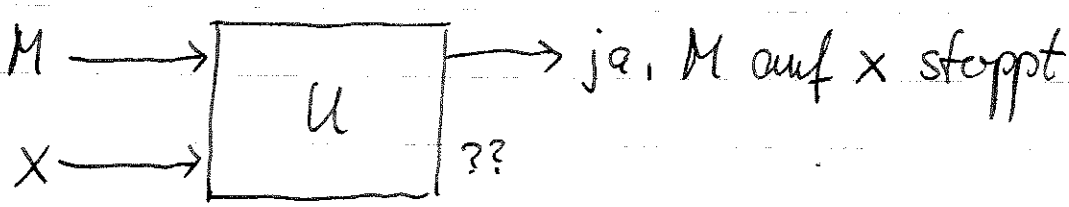
$\Leftrightarrow$ T angesetzt auf (T', T') gibt 0 aus

$\Leftrightarrow$ $(T', T') \notin H$

$\Leftrightarrow$ T' angesetzt auf T' hält nicht.

Dies ist ein Widerspruch. Dieser zeigt, dass H nicht entscheidbar sein kann.

Bem: Das Problem H ist aber durchaus semi-entscheidbar:



Einen solchen Algorithmus U zu programmieren, ist durchaus möglich: U führt den Rechenablauf von M, bei Eingabe x, Schritt für Schritt wie ein Interpreter aus. Sollte M auf x nicht stoppen, dann stoppt U auch nicht.

Ein solcher Algorithmus U, programmiert als Turingmaschine, heißt universelle Turingmaschine.
(Eine solche universelle Turingmaschine hat Turing bereits in seinem Artikel von 1936 angegeben.)
Um weitere Probleme als unentscheidbar nachzuweisen, beziehen wir uns zurück auf die Unentscheidbarkeit von H.
Sehr nützliches Konzept: Reduzierbarkeit.

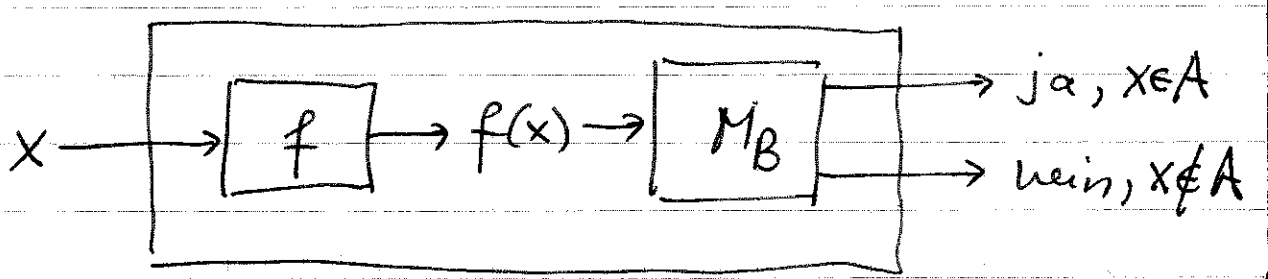
Def: Seien A, B Mengen (Entscheidungsprobleme).
Dann heißt A auf B reduzierbar
(symbolisch: $A \leq B$), falls es eine totale + berechenbare Funktion f gibt, so dass

$$\forall x: \quad x \in A \iff f(x) \in B.$$

Eine unmittelbare Konsequenz aus der Definition ist:

Lemma: Es gelte $A \preceq B$. Ferner sei B (semi-) entscheidbar.
Es folgt, dass auch A (semi-) entscheidbar ist.

Beweis: Die Reduktion von A nach B werde durch die totale + berechenbare Funktion f vermittelt. Ein Entscheidungsverfahren M_B für B kann für die Entscheidung von A verwendet werden:



Analog: aus der Semi-Entscheidbarkeit von B folgt die Semi-Entscheidbarkeit von A. □

Bem: Anwendung des Lemmas in der logisch äquivalenten Kontraposition:

Es gelte $A \leq B$. Ferner sei A nicht entscheidbar.
Es folgt, dass auch B nicht entscheidbar ist.

(Die Rolle von A übernimmt z.B. das Halteproblem H.)

Def: $H_0 := \{ M \mid M$ ist eine Turingmaschine.
M angesetzt auf leeres Band
stoppt.
(Analog: ein While-Programm oder
GOTO-Programm mit allen Programm-
Variablen auf 0 gesetzt, stoppt.) $\}$

--- das Halteproblem bei leerem Band
(bzw. das Halteproblem ohne eine Eingabe)

Wir zeigen: $H \leq H_0$.

Damit folgt, dass H_0 nicht entscheidbar ist.

Finde also eine berechenbare Funktion

$$f: (M, x) \longmapsto M'$$

so dass gilt:

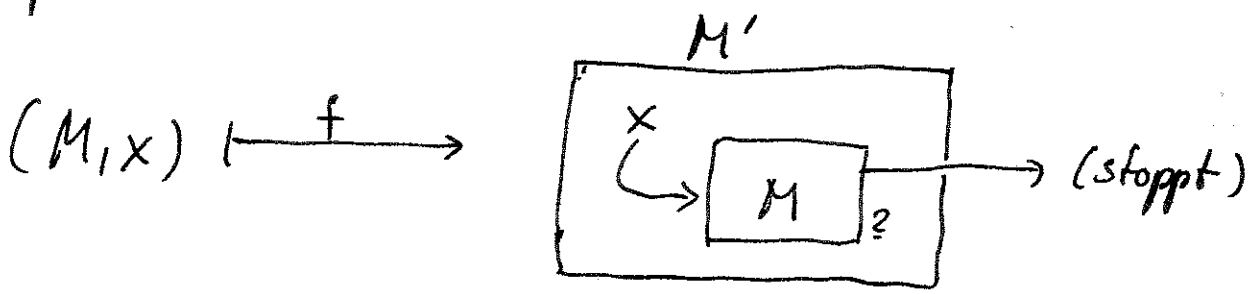
$$(M,x) \in H \iff f(M,x) = M' \in H_0$$

$$\left(\begin{array}{c} M \text{ angesetzt auf} \\ x \text{ stoppt} \end{array}\right) \iff \left(\begin{array}{c} M' \text{ angesetzt auf} \\ \text{leeres Band stoppt} \end{array}\right)$$

Die aus M und x zu konstruierende Maschine M' arbeit wie folgt:

M' {
- Auf leerem Band angesetzt schreibt M' zunächst x aufs Band.
- Danach verhält sich M' wie M, angesetzt auf x.

Graphisch:



f erfüllt die Bedingung einer Reduktion, denn:

$(M,x) \in H \iff$
M angesetzt auf x hält $\iff$

$(f(M,x) =) M'$ angesetzt auf leeres Band hält.

Somit ist bewiesen:

> Satz: Das Halteproblem auf leerem Band ist unentscheidbar.

Das Halteproblem auf leerem Band wird nur deshalb benötigt, um den folgenden Satz zu beweisen.

Satz von Rice, 1951:

Sei R die Menge aller partiell berechenbaren Funktionen. Sei $\emptyset \subsetneq \mathcal{S} \subsetneq R$ eine beliebige nicht-triviale Teilmenge. Dann ist das Entscheidungsproblem

$$C(\mathcal{S}) = \{ M \mid \text{die vom Algorithmus } M \text{ berechnete Funktion liegt in } \mathcal{S} \}$$

unentscheidbar!

Dies sind alles Spezialfälle des Satzes von Rice:

Es ist unentscheidbar, festzustellen ob bei gegebenem Algorithmus M:

- M immer stoppt (also eine totale Fkt berechnet)
- M eine konstante Funktion berechnet
- M eine monoton steigende Funktion berechnet
- M die Funktion $x \mapsto x^2$ berechnet
- M eine partielle Funktion berechnet, also für manche Eingaben nicht stoppt.

usw. usw.

Kurz formuliert: Gegeben ein Algorithmus M, dann ist es unentscheidbar, irgendeine Eigenschaft der von M berechneten Funktion festzustellen.

Beweis des Satzes von Rice:

$\Omega \in \mathcal{R}$ ist die überall undefinierte Fkt.

Entweder $\Omega \in \mathcal{S}$ (Fall 1) oder $\Omega \notin \mathcal{S}$ (Fall 2).

Fall 1: Da $\mathcal{S} \neq \mathcal{R}$ gibt es $q \in \mathcal{R} \setminus \mathcal{S}$; diese Funktion werde durch Maschine Q berechnet.

Definiere Funktion $f : M \longmapsto M'$

wobei M' wie folgt arbeitet:

> Angesetzt auf eine Eingabe x:
> wird diese zunächst ignoriert und M'
> verhält sich wie M angesetzt auf
> leeres Band. Falls dies stoppt, so
> starte Q auf Eingabe x.

Nun gilt:

$M \in H_0 \Rightarrow$ M auf leerem Band stoppt

$\Rightarrow$ M' auf x berechnet wie Q auf x, also berechnet $q(x)$ $\forall x$

$\Rightarrow$ die von M' berechnete Fkt liegt nicht in $\mathcal{S}$

$\Rightarrow \underbrace{f(M)}_{M'} \notin C(\mathcal{S})$

Sowie: $M \notin H_0 \Rightarrow$ M auf leerem Band stoppt nicht

$\Rightarrow$ M' ($= f(M)$) stoppt nie „berechnet" also Ω

$\Rightarrow$ die von $f(M)$ (berechnete Fkt.) liegt in $\mathcal{S}$

$\Rightarrow$ $f(M) \in C(\mathcal{S})$

Insgesamt zeigt dies: $\overline{H_0} \leq C(\mathcal{S})$.

Es gilt: H_0 unentscheidbar $\Rightarrow$ $\overline{H_0}$ unentscheidbar

$\Rightarrow$ $C(\mathcal{S})$ unentscheidbar.

Fall 2: Analog zeigt man $H_0 \leq C(\mathcal{S})$. □

Post's Correspondence Problem (PCP)

(Emil Post, 1947)

Gegeben: $K = \binom{x_1}{y_1}\binom{x_2}{y_2}\cdots\binom{x_k}{y_k}$

wobei $x_i, y_i \in \Sigma^+$ (Σ ein Alphabet)

Beispiel: $K = \binom{a}{aba}\binom{ab}{bb}\binom{baa}{aa}$

1. 2. 3.

Gefragt: Gibt es Folge von Indizes

$i_1, i_2, \ldots, i_n \in \{1, 2, \ldots, k\}$ so dass:

$$x_{i_1} x_{i_2} \cdots x_{i_n} = y_{i_1} y_{i_2} \cdots y_{i_n} \quad ?$$

Obiges Beispiel: Indexfolge 1, 3, 2, 3 ergibt:

$$x_1 x_3 x_2 x_3 = a|baa|ab|baa$$
$$y_1 y_3 y_2 y_3 = a|ba|aa|bb|aa$$

Solche Indexfolge heißt dann **Lösung** des PCP.

Anderes Beispiel: $\binom{001}{0}\binom{01}{011}\binom{01}{101}\binom{10}{001}$

Besitzt Lösung, 1. 2. 3. 4.

aber kürzeste Lösung besteht aus

66 Indizes: 2, 4, 3, 4, 4, ..., 1, 3

x-Folge: 01|10|01|10|10| ... |001|01

y-Folge: 011|001|101|001|001 |0|101

2. 4. 3. 4. 4. 1. 3.

Bem: Das PCP ist semi-entscheidbar

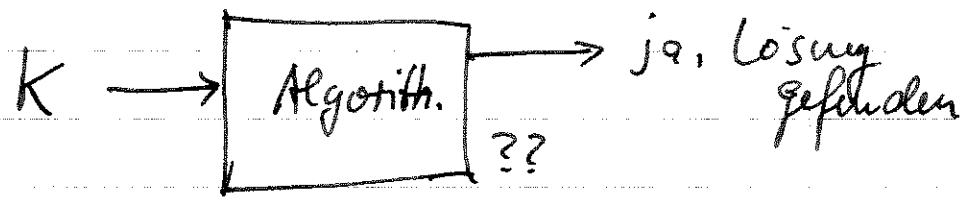
Eingabe $\binom{x_1}{y_1}\binom{x_2}{y_2}\cdots\binom{x_k}{y_k} = K$

for $n := 1, 2, 3, 4, \ldots$ do

Teste alle Indexfolgen $(i_1, i_2, \ldots, i_n)$

ob $x_{i_1} x_{i_2} \cdots x_{i_n} = y_{i_1} y_{i_2} \cdots y_{i_n}$

Skizze:



Wir zeigen, dass PCP unentscheidbar ist

(Bem: Das ist das erste unentscheidbare Problem, bei dem es nicht der Fall ist, dass Algorithmen M als Eingabe vorkommen.)

Beweisstrategie: Zeigen, dass:

$$H \leq MPCP \text{ und } MPCP \leq PCP$$

MPCP (modifiziertes PCP) wird nur aus technischen Gründen benötigt:

Problem MPCP:

gegeben: K wie beim PCP

gefragt: Gibt es eine Lösung $i_1, i_2, \ldots, i_n$ (wie beim PCP) so dass $i_1 = 1$?

D.h: PCP-Lösungen mit $i_1 \neq 1$ werden nicht als Lösung des MPCP betrachtet.

Lemma: $MPCP \leq PCP$

Beweis: Gegeben $K = \left(\begin{smallmatrix} x_1 \\ y_1 \end{smallmatrix}\right)\left(\begin{smallmatrix} x_2 \\ y_2 \end{smallmatrix}\right) \dots \left(\begin{smallmatrix} x_k \\ y_k \end{smallmatrix}\right)$

über Alphabet Σ.

Verwende 2 neue Symbole \$ und # ($\notin \Sigma$)

Notation: Für $w = a_1 a_2 \dots a_n \in \Sigma^+$

schreiben:

$$\overline{w} = \# a_1 \# a_2 \# \dots \# a_n \#$$

$$\acute{w} = a_1 \# a_2 \# \dots \# a_n \#$$

$$\grave{w} = \# a_1 \# a_2 \# \dots \# a_n$$

Hier ist die gesuchte Reduktion:

$$K = \left(\begin{smallmatrix} x_1 \\ y_1 \end{smallmatrix}\right)\left(\begin{smallmatrix} x_2 \\ y_2 \end{smallmatrix}\right) \dots \left(\begin{smallmatrix} x_k \\ y_k \end{smallmatrix}\right) \quad \xmapsto{f}$$

$$\left(\begin{smallmatrix} \overline{x_1} \\ \grave{y}_1 \end{smallmatrix}\right)\left(\begin{smallmatrix} \acute{x}_1 \\ \grave{y}_1 \end{smallmatrix}\right)\left(\begin{smallmatrix} \acute{x}_2 \\ \grave{y}_2 \end{smallmatrix}\right) \dots \left(\begin{smallmatrix} \acute{x}_k \\ \grave{y}_k \end{smallmatrix}\right)\left(\begin{smallmatrix} \$ \\ \#\$ \end{smallmatrix}\right)$$

Beispiel: Aus $K = \left(\begin{smallmatrix} a \\ aba \end{smallmatrix}\right)\left(\begin{smallmatrix} ab \\ bb \end{smallmatrix}\right)\left(\begin{smallmatrix} baa \\ aa \end{smallmatrix}\right)$

wird das neue PCP:

$$\binom{\#a\#}{\#a\#b\#a}\binom{a\#}{\#a\#b\#a}\binom{a\#b\#}{\#b\#b}\dots\binom{b\#a\#a\#}{\#a\#a}\binom{\$}{\#\$}$$

Die Lösung 1,3,2,3 des ursprünglichen PCPs welche auch eine Lösung von MPCP ist, da sie mit 1 beginnt, ist im ursprünglichen (M)PCP:

x-Folge: a | b a a | a b | b a a

y-Folge: a b a | a a | b b | a a

Nach der Transformation bzw. Reduktion lautet die Lösung:

x-Folge: # a # | b # a # a # | a # b # | b # a # a # | $

y-Folge: # a # b # a | # a # a | # b # b | # a # a | # $

Der Trick besteht darin, dass das PCP nach der Reduktion, wenn überhaupt, mit dem 1. Paar beginnen ~~kann~~ muss, da es das einzige ist, bei dem x und y mit # beginnen.

Dies zeigt:

$$\left(\begin{array}{l}K \text{ besitzt eine} \\ \text{Lösung mit } i_1 = 1\end{array}\right) \text{ gdw. } \left(\begin{array}{l}\text{das transformierte PCP } f(K) \\ \text{besitzt (irgend) eine Lösung}\end{array}\right)$$

Also: $MPCP \leq PCP$. $\square$

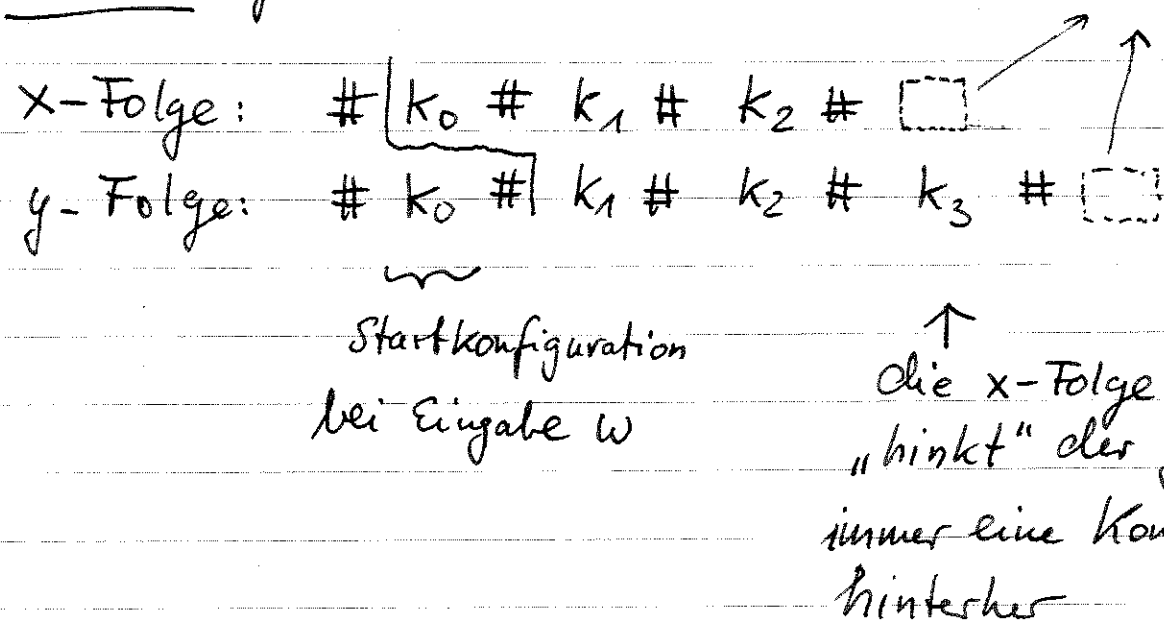
Lemma: $H \leq MPCP$

Beweis: Gegeben eine Turingmaschine M mit einer Eingabe w. Wir müssen eine Abbildung f finden, so dass $(M,w) \xmapsto{f} \binom{x_1}{y_1}\binom{x_2}{y_2}\cdots\binom{x_k}{y_k}$ und es gilt:

$$\left(\begin{array}{l}M \text{ auf Eingabe } w \\ \text{stoppt}\end{array}\right) \Leftrightarrow \left(\begin{array}{l}\text{das PCP } f(M,w) \text{ hat} \\ \text{eine Lösung, die mit} \\ \text{dem ersten Paar } \binom{x_1}{y_1} \\ \text{beginnt}\end{array}\right)$$

Die Idee besteht darin, die PCP-Paare so anzugeben, dass durch Hintereinanderfügen der $\binom{x_i}{y_i}$-Paare ein Rechenablauf von M, bei Eingabe w nachgebildet werden kann.

Die folgende Skizze zeigt die Intention:



Das erste PCP-Paar, das die Sonderrolle spielt, dass mit diesem begonnen werden muss, stellt die Startkonfiguration $k_0 = z_0 w$ und den ersten „Überhang" der y-Folge gegenüber der x-Folge her:

$$\binom{\#}{\#\, z_0\, w\, \#}$$

Die weiteren PCP-Paare:

$\binom{a}{a}$ ... dient nur zum Kopieren gleicher Symbole ~~bei~~ von Konfiguration k_i in die Konfiguration k_{i+1}

$\begin{pmatrix} z\,a \\ z'\,b \end{pmatrix}$ falls $\delta(z,a) = (z', b, N)$

$\begin{pmatrix} z\,a \\ b\,z' \end{pmatrix}$ falls $\delta(z,a) = (z', b, R)$

$\forall c$: $\begin{pmatrix} c\,z\,a \\ z'\,c\,b \end{pmatrix}$ falls $\delta(z,a) = (z', b, L)$

... sowie noch einige Paare für die Sonderfälle, wenn z ganz an den Rand einer Konfiguration geraten ist.

Ferner noch einige Paare, die dafür sorgen, dass, wenn ein Endzustand erreicht wurde, der „Überhang" der y-Folge gegenüber der x-Folge schrittweise abgebaut wird und somit eine Lösung des MPCP ermöglicht wird (Skript Seite 47).

□

z.B $\begin{pmatrix} a\,z\,e \\ z\,e \end{pmatrix}, \begin{pmatrix} z\,e\,a \\ z\,e \end{pmatrix}$

Das (leerer) Schnittproblem bei kontextfreien Grammatiken (kf Schnitt):

Gegeben: kf. Grammatiken G_1, G_2

Gefragt: Ist das Schnitt $L(G_1) \cap L(G_2)$ leer?

Das komplementäre Problem $\overline{\text{kf Schnitt}}$ stellt die Frage, ob es ein Wort w gibt mit $w \in L(G_1)$ und $w \in L(G_2)$.

Es gilt: $PCP \leq \overline{\text{kf Schnitt}}$, daher ist das Schnittproblem für kf. Grammatiken unentscheidbar.

Wir führen dies an einem Beispiel vor.

PCP-Problem: $K = \left(\begin{matrix} a \\ aba \end{matrix}\right)\left(\begin{matrix} ab \\ bb \end{matrix}\right)\left(\begin{matrix} baa \\ aa \end{matrix}\right)$

1. 2. 3.

Aus K konstruieren wir 2 Grammatiken G_1, G_2.

In G_1 werden die x-Folgen dargestellt,

in G_2 werden die y-Folgen dargestellt

(zusammen mit den Indizes, die der jeweiligen Folge zugrunde(liegt).

Das Terminalalphabet ist:

$$\{ a, b, 1, 2, 3 \}$$

$\underbrace{\quad}_{\text{das Alphabet des PCPs.}}$ $\underbrace{\quad}_{\text{die Indizes des PCPs}}$

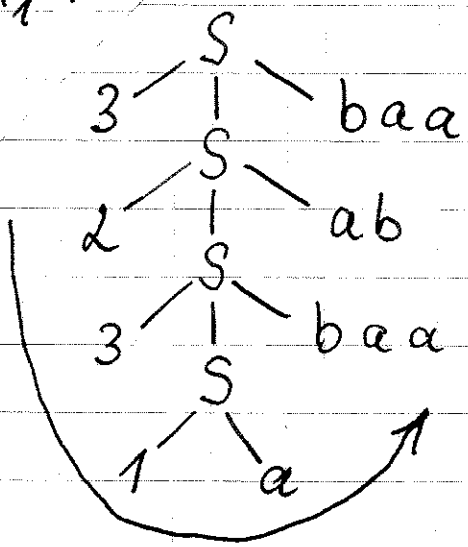
G_1: $S \to 1a \mid 2ab \mid 3baa \mid 1Sa \mid 2Sab \mid 3Sbaa$

G_2: $S \to 1aba \mid 2bb \mid 3aa \mid 1Saba \mid 2Sbb \mid 3Saa$

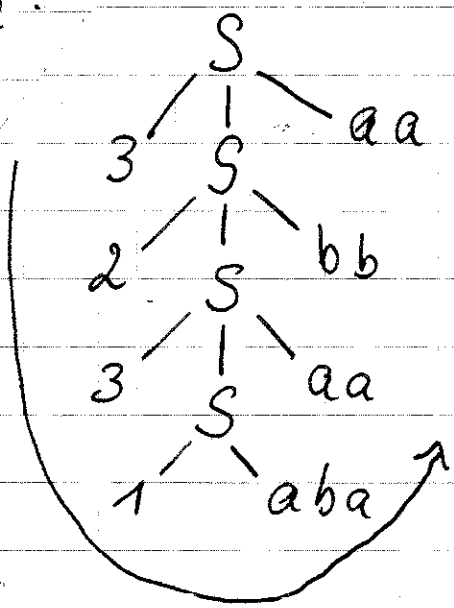
Die Lösung 1,3,2,3 des PCP ergibt folgendes gemeinsames Wort in $L(G_1)$ und $L(G_2)$: 3231abaaabbaa

Dies bestätigen die folgenden beiden Syntax-bäume:

Ableitung in G_1:



Ableitung in G_2:



Das Beispiel kann so verallgemeinert werden, dass damit die Reduktion von PCP nach $\overline{\text{kf Schnitt}}$ gezeigt werden kann. (s. Skript).

Bemerkungen: Die im Beweis konstruierten Grammatiken G_1, G_2 sind sogar deterministisch kontextfrei (→ Vorlesung Formale Grundlagen).

Die determ. kf. Sprachen sind effektiv unter Komplement abgeschlossen, d.h. es gibt Konstruktionsvorschrift f, so dass

$$\underbrace{G}_{\text{det. kf. Grammatik}} \longmapsto \underbrace{f(G)}_{\text{det. kf. Grammatik}} \quad \text{so dass} \quad \overline{L(G)} = L(f(G))$$

Ähnlich: Die kf. Sprachen sind effektiv unter Vereinigung abgeschlossen: Es gibt Transformation g so dass:

$$\underbrace{G_1, G_2}_{\text{kf. Grammatiken}} \longmapsto \underbrace{g(G_1, G_2)}_{\text{kf. Grammatik}}$$

so dass: $L(G_1) \cup L(G_2) = L(g(G_1, G_2))$

Mit diesen Fakten lässt sich nun

kf Schnitt bei det. kf. $\leq$ Äquivalenzproblem bei kf. Grammatiken

Gegeben: G_1, G_2 gefragt: $L(G_1) = L(G_2)$?

Zeigen:

$$L(G_1) \cap L(G_2) = \emptyset \iff L(G_1) \subseteq L(f(G_2))$$

$$\iff L(G_1) \cup L(f(G_2)) = L(f(G_2))$$

$$\iff L(g(G_1, f(G_2))) = L(f(G_2))$$

Die gesuchte Reduktionsfunktion ist also:

$$(G_1, G_2) \longmapsto (g(G_1, f(G_2)), f(G_2))$$

Damit ist das Äquivalenzproblem für kf. Grammatiken ebenfalls unentscheidbar!

Mit Hilfe des PCP lässt sich die Unentscheidbarkeit der Prädikatenlogik (→ Vorlesung Logik) nachweisen:

PCP $\leq$ Gültigkeitsproblem für Formeln der Prädikatenlogik.

Wir führen dies wieder am Beispiel durch.

Sei $K = \left(\begin{smallmatrix} a \\ aba \end{smallmatrix}\right)\left(\begin{smallmatrix} ab \\ bb \end{smallmatrix}\right)\left(\begin{smallmatrix} baa \\ aa \end{smallmatrix}\right)$ das gegebene PCP.

Die zu konstruierende Formel F der Prädikatenlogik verwendet folgende Signatur:

c .... Konstante (0-stell. Funktionssymbol)

f_a, f_b .... 1-stellige Funktionssymbole

P .... 2-stelliges Prädikat

Sei z.B. $x = aaba$ ein Wort. Wir verwenden die Abkürzung $f_x()$ für

$f_a(f_b(f_a(f_a())))$

Das PCP K wird auf folgende Formel F abgebildet:

$$F = P(f_a(c), f_{aba}(c)) \wedge P(f_{ab}(c), f_{bb}(c)) \wedge P(f_{baa}(c), f_{aa}(c))$$

$$\wedge \forall u \forall v \left(P(u,v) \rightarrow P(f_a(u), f_{aba}(v)) \vee P(f_{ab}(u), f_{bb}(v)) \vee P(f_{baa}(u), f_{aa}(v)) \right)$$

$$\longrightarrow \quad \exists z \; P(z,z)$$

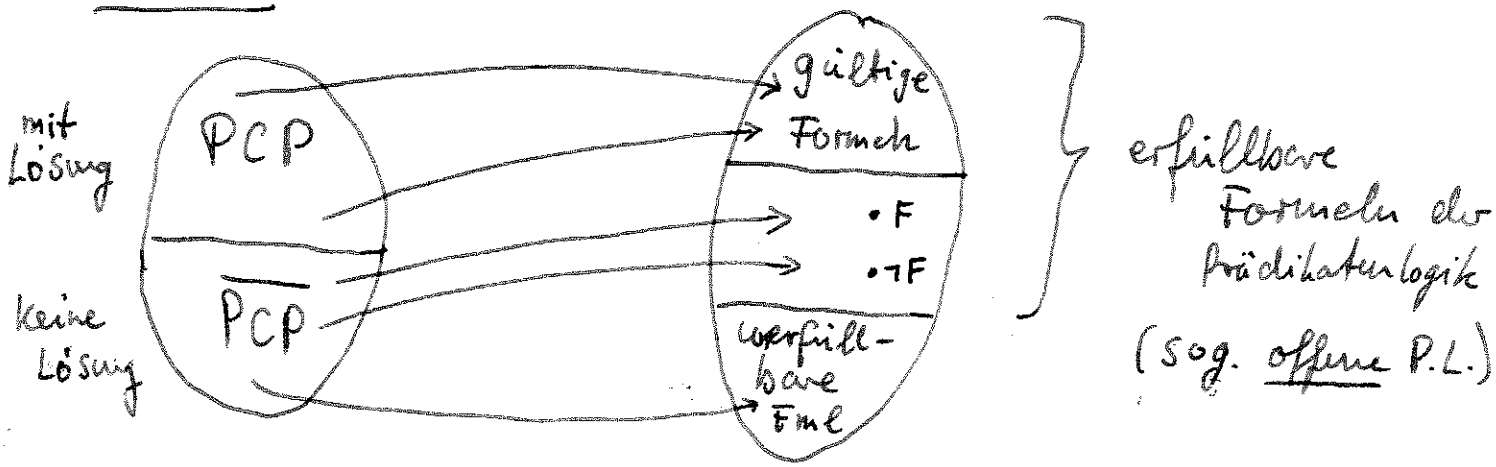
Es gilt:

K besitzt eine Lösung (in diesem Fall Indexfolge 1,3,2,3 mit Lösungswort abaaabbaa) genau dann, wenn gilt:

F ist gültig.

In der letzten Vorlesung wurde eine Reduktion PCP $\leq$ Gültigkeit von präd. log. Formeln angegeben.

Skizze:



Bem: Es gibt Formeln F so, dass weder F noch ¬F gültig ist.

In der offenen Prädikatenlogik werden den vorkommenden Prädikaten, Funktionen keine von vornherein festen Interpretationen zugewiesen.

Nun betrachten wir eine Variante, bei der den prädikatenlog. Formeln eine feste Interpretation (eine sog. Struktur → Vorlesung Logik) zugewiesen wird.

Alle vorkommenden Zahlenwerte (von Variablen, Konstanten, Funktionen) sind natürliche Zahlen.

Das heißt, als Grundmenge wird $\mathbb{N}$ festgelegt.

Die einzig vorkommenden Funktionen sind $+$ (für Addition, 2-stellig) und $*$ (für Multiplikation, 2-stellig). Als Prädikat ist $=$ zugelassen (Gleichheit).

Mit diesen Grundoperationen können weitere Funktionen bzw. Prädikate leicht definiert werden:

„$x < y$“ : $\exists z \neg(z=0) \wedge (x+z=y)$

„x ist Teiler von y“: $\exists z\,(x*z=y)$

„x mod $y = z$“ : $\exists q\,(x = q*y+z) \wedge (z<y)$

„x ist Primzahl“ : $\forall y \forall z\,((x=y*z) \rightarrow ((y=1) \vee (z=1))) \wedge (x \neq 1)$

usw.

Definiere (in Logik-Vorlesung heißt diese Menge $Th(\mathbb{N})$):

$WA = \{ F \mid F$ ist eine solche prädikatenlogische Formel, die unter der Interpretation $(\mathbb{N}, +, *, =)$ wahr ist $\}$

unter dieser festen Interpretation,

Bem: Jede Formel F ist nun entweder wahr oder falsch. Wenn F falsch ist, dann ist $\neg F$ wahr (und umgekehrt). Dies ist anders als in der offenen Prädikatenlogik.

In der Terminologie der Reduzierbarkeit heißt das: $WA \leq \overline{WA}$ bzw. $\overline{WA} \leq WA$.

Die Reduktionsfunktion ist hierbei $F \longmapsto \neg F$.

Diese Tatsache hat eine besondere Konsequenz:

Wenn $\overline{WA}$ (bzw. WA) semi-entscheidbar sein sollte, dann folgt, dass auch WA (bzw. $\overline{WA}$) semi-entscheidbar ist.

Aus WA semi-entsch. und $\overline{WA}$ semi-entsch. folgt, dass WA sogar entscheidbar ist.

Wir werden mit ähnlichen Methoden wie letztes Mal zeigen, dass $\boxed{PCP \leq WA.}$

Damit folgt, dass WA nicht entscheidbar ist, und nach der obigen Diskussion sogar nicht semi-entscheidbar ist.

Diese Aussage: WA ist nicht semi-entscheidbar
bzw. WA ist nicht rekursiv aufzählbar
kann inhaltlich so umgedeutet werden:

Die Theorie der natürlichen Zahlen (andere Bezeichnung für WA) ist nicht axiomatisierbar.

Nochmals anders ausgedrückt: jeder „Versuch", die Theorie der natürlichen Zahlen zu axiomatisieren (bekannt sind z.B. die Axiome von Peano) ist nicht „ausdrucksstark" genug, um daraus alle wahren Sätze der Zahlentheorie (also WA) herleiten zu können.

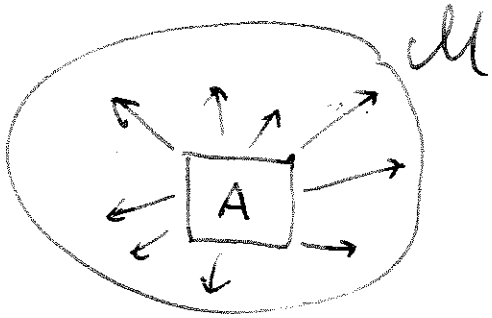
Diese Formulierung ist der berühmte

Gödelsche Unvollständigkeitssatz

Gödel selbst hat diesen Satz (wie wir beim Halteproblem) in einer indirekten Art und Weise bewiesen.
(Dies erinnert wieder an das Argument wie beim Barbier, der alle im Dorf rasiert, die sich nicht selbst rasieren.)

Nochmals zur Axiomatisierbarkeit. Eine Menge M von Formeln wird als axiomatisierbar bezeichnet, wenn es eine Menge von Formeln

$A \subseteq M$ gibt; die Axiome, wobei A entscheidbar sein sollte, so dass man alle Formeln in M durch Anwenden geeigneter Ableitungsregeln aus A herleiten kann:



Indem man systematisch alle Axiome auflistet und desweiteren alle Formeln, die sich in 1, in 2, in 3, usw. Schritten ableiten lassen, so erhält man ein rekursives Aufzählungsverfahren für M. Deshalb gilt:

M axiomatisierbar $\Rightarrow$ M rekursiv aufzählbar
(= semi-entscheidbar)

(Man kann „axiomatisierbar" und „rek. aufzählbar" gleichsetzen).

Wie kann man PCP $\leq$ WA beweisen?

Beim PCP geht es um Strings, die hintereinandergehängt werden, so dass schließlich bei der x-Folge und der y-Folge dasselbe Wort entsteht.

Bei WA geht es um arithmetische Rechenoperationen, die mit + und * (desweiteren: Teilbarkeit, div, mod...) ausgedrückt werden können.

Man muss das Strings-aneinanderfügen durch arithmetische Rechenoperationen simulieren (man spricht von „arithmetisieren", auch von „Gödelisieren").

Haben wir als Beispiel das PCP

$$K = \underset{1.}{\begin{pmatrix} 1 \\ 101 \end{pmatrix}}, \underset{2.}{\begin{pmatrix} 10 \\ 00 \end{pmatrix}}, \underset{3.}{\begin{pmatrix} 011 \\ 11 \end{pmatrix}}$$

Angenommen, wir stellen Strings $\in \{0,1\}^*$ als natürliche Zahlen (z.B. im Dezimalsystem) dar. Nehmen wir an, $x \in \mathbb{N}$ stellt bereits einen String als nat. Zahl dar. Dann bedeutet das Hintenanfügen von "101"

an den String, den x repräsentiert, die Rechenoperation $1000 * x + 101$.

Auf diese Weise können wir den x_i's zuordnen:

$f_1(u) := 10 * u + 1$ denn $x_1 = 1$

$f_2(u) := 100 * u + 10$ denn $x_2 = 10$

$f_3(u) := 1000 * u + 011$ denn $x_3 = 011$

Analog kann man es mit den y_i's machen:

$g_1(u) := 1000 * u + 101$ denn $y_1 = 101$

$g_2(u) := 100 * u + 00$ denn $y_2 = 00$

$g_3(u) := 100 * u + 11$ denn $y_3 = 11$

Die Tatsache, dass obiges PCP die Lösung 1,3,2,3 hat, kann jetzt so ausgedrückt werden:

$$f_3(f_2(f_3(f_1(0)))) = g_3(g_2(g_3(g_1(0))))$$

Allgemeiner kann man die Aussage, dass (speziell bei 3 x_i's und 3 y_i's) eine Lösung existiert, so ausdrücken:

$\exists n$ $\exists$ Zahlen $u_0, \dots, u_n, v_0, \dots, v_n$ so dass:

$$(u_0 = 0) \wedge (v_0 = 0) \wedge (u_n = v_n) \wedge \forall i < n:$$

$$\Big[(f_1(u_i) = u_{i+1}) \wedge (g_1(v_i) = v_{i+1}) \Big]$$

$$\vee \Big[(f_2(u_i) = u_{i+1}) \wedge (g_2(v_i) = v_{i+1}) \Big]$$

$$\vee \Big[(f_3(u_i) = u_{i+1}) \wedge (g_3(v_i) = v_{i+1}) \Big]$$

Das verbleibende technische Problem ist, dass ein $\exists$-Quantor sich nicht auf eine Folge von nat. Zahlen, sondern nur auf einzelne Zahlen beziehen darf. Dieses Problem kann man so lösen, dass man eine Methode findet, beliebig lange Zahlenfolgen durch eine einzelne (oder zwei) Zahlen zu codieren. Dies gelingt mit Hilfe des chinesischen Restsatzes. Es verbleiben noch einige technische Probleme, um dies dann konkret durchzuführen (→ Skript, Seite 53).

Zusammenfassung:

Die Menge der gültigen Formeln der offenen Prädikatenlogik ist

- nicht entscheidbar (Church, Turing 1936)
  Hier gezeigt durch Reduktion vom PCP aus.

- semi-entscheidbar bzw. axiomatisierbar
  („Gödelscher Vollständigkeitssatz" 1929 – Gödel zeigt dies speziell für den sog. Hilbert-Kalkül.
  In der Vorlesung Logik: Algorithmus von Gilmore bzw. prädikatenlogischer Resolutionskalkül)

Die Menge der wahren prädikatenlogischen Formeln mit der speziellen Interpretation $(\mathbb{N}, +, *)$, die sog. Zahlentheorie oder Arithmetik, ist

- weder entscheidbar, noch semi-entscheidbar noch axiomatisierbar.

  („Gödelscher Unvollständigkeitssatz" von 1931)
  In der Vorlesung gezeigt durch Reduktion von PCP aus, sowie Eigenschaften von WA und (semi)-entsch.

Zum Begriff der
„Unvollständigkeit" beim Gödelschen Satz.

Sei A eine Menge von Axiomen, dieses seien Formeln mit Quantoren und den Operatoren $+, *$ sowie den Gleichheitszeichen $=$, welche in der Interpretation $(\mathbb{N}, +, *)$ wahr sind. Sei $Cons(A)$ die Menge aller Konsequenzen der Formeln in A, alles was sich aus den Axiomen A an weiteren wahren Formeln ableiten lässt. Dann sagt der Gödelsche Unvollständigkeitssatz, das $Cons(A)$ niemals gleich WA sein kann (sonst wäre WA rekursiv aufzählbar), sondern es muss gelten

$$Cons(A) \subsetneq WA$$

Jeder Kalkül basierend auf A bleibt <u>unvollständig</u>, es bleiben immer Formeln $F \in WA$ übrig, deren Wahrheit vermittels A nicht bewiesen werden kann: $F \in WA \setminus Cons(A)$.

Kurt Gödel (geb. 1906 in Brünn, Österreich.
gest. 1978 in Princeton, USA)

Dissertation, 1929: „Vollständigkeitssatz"

Habilitation, 1931: „Unvollständigkeitssatz".

Emigration in die USA, 1940
Wurde Professor in Princeton
enge Freundschaft mit Einstein
Paranoia, Unterernährung

„Hilberts Programm" – basierend auf berühmten Vortrag im Jahr 1900 auf Mathematiker-Kongress in Paris. Hilbert formuliert 23 offene Probleme, u.a. Frage nach vollständiger Axiomatisierung der Zahlentheorie. (Gödel zeigt: die gibt es nicht.)

Hilberts 10. Problem: Finde Methode (Algorithmus) zur Lösung Diophantischer Gleichungen.

„Lösen Diophantischer Gleichungen" ist gleichwertig mit Bestimmen des Wahrheitsgehalts von solchen Formeln in WA, die nur Existenzquantoren enthalten (und keine Negationen), z.B.:

$$\exists x \exists y \exists z \underbrace{(x = y * z \wedge \text{----})}_{\text{Diophantisches Gleichungssystem}}$$

$\overset{\parallel}{=}$ nur natürlich-zahlige Lösungen von Interesse

1970 zeigt Y. Matiyasevich, dass dieses Problem unentscheidbar ist (durch Reduktion vom Halteproblem von GOTO-Programmen bzw. Registermaschinen aus).

Komplexitätstheorie

Fixiere ein bestimmtes Berechnungsmodell: Mehrband-Turingmaschine.

Sei $TIME(f(n))$ die Menge aller Entscheidungsprobleme A, für die es eine Mehrband-TM M gibt so dass für alle Eingaben x gilt:

$$\underbrace{time_M(x)}_{\substack{\text{Anzahl Rechen-}\\ \text{schritte von } M,\\ \text{bei Eingabe } x}} \leq f(|x|)$$

Typische Funktionen $f(n) = \underbrace{n,\ n^2,\ n^3}_{\text{Polynomial}},\ \underbrace{2^n, n^n}_{\text{exponential}}$ usw.

$$\mathbb{P} := \bigcup_{\substack{p \text{ ist}\\ \text{Polynom, also}\\ p(n) = a_k n^k + \dots + a_1 \cdot n^1 + a_0}} TIME(p(n))$$

(k ... Grad des Polynoms, $a_k \neq 0$)

$$= \bigcup_{k \in \mathbb{N}} TIME(k \cdot n^k)$$

Man setzt die Klasse P gleich mit „effizient berechenbar".

Bsp: Nicht-polynomial sind Funktionen wie

$n^{\log n}$, $(\log n)^{\log n}$, 2^n, 3^n, $n!$, n^n, 2^{2^n}, $2^{2^{\cdot^{\cdot^{2}}}}$ n-mal

Polynome haben im Vergleich zu Exponentialfunktionen ein gemäßigtes Wachstum:

Time complexity function	Size n					
	10	20	30	40	50	60
n	.00001 second	.00002 second	.00003 second	.00004 second	.00005 second	.00006 second
n^2	.0001 second	.0004 second	.0009 second	.0016 second	.0025 second	.0036 second
n^3	.001 second	.008 second	.027 second	.064 second	.125 second	.216 second
n^5	.1 second	3.2 seconds	24.3 seconds	1.7 minutes	5.2 minutes	13.0 minutes
2^n	.001 second	1.0 second	17.9 minutes	12.7 days	35.7 years	366 centuries
3^n	.059 second	58 minutes	6.5 years	3855 centuries	2×10^8 centuries	1.3×10^{13} centuries

Figure 1.2 Comparison of several polynomial and exponential time complexity functions.

Noch drastischeres Argument: Bei polynomialer Komplexität nimmt lösbare Problemgröße um bestimmten Faktor zu; bei exponentieller Komplexität nur um additive Konstante.

Size of Largest Problem Instance Solvable in 1 Hour

Time complexity function	With present computer	With computer 100 times faster	With computer 1000 times faster
n	N_1	$100\ N_1$	$1000\ N_1$
n^2	N_2	$10\ N_2$	$31.6\ N_2$
n^3	N_3	$4.64\ N_3$	$10\ N_3$
n^5	N_4	$2.5\ N_4$	$3.98\ N_4$
2^n	N_5	$N_5+6.64$	$N_5+9.97$
3^n	N_6	$N_6+4.19$	$N_6+6.29$

Figure 1.3 Effect of improved technology on several polynomial and exponential time algorithms.

Verallgemeinerte Church'sche These:

Alle Berechnungsmodelle sind untereinander polynomial äquivalent. Das bedeutet: die Definition der Klasse P ist robust gegenüber der Wahl des Berechnungsmodells.

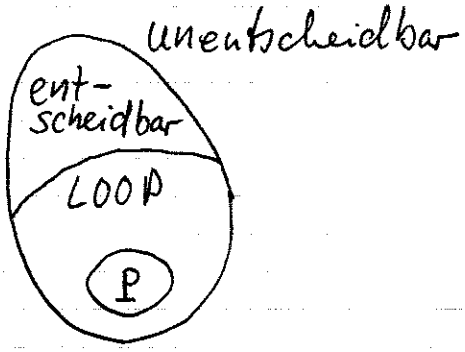
Vergleich mit LOOP-Berechenbarkeit:

Polynome lassen sich durch + und * - Berechnungen durch ein LOOP-Programm berechnen. Beim Umformen einer Mehrband-TM, die polynomial zeitbeschränkt ist (also $time_M(x) \leq p(|x|)$) entsteht ein While-Programm (oBdA mit nur einer While-Schleife), so dass die Anzahl der While-Schleifendurchgänge durch ein Polynom $q(n)$ begrenzt ist. Dieser Zahlenwert $q(n)$, $n = |x|$, kann zuvor durch ein LOOP-Programm berechnet werden. Man kann dann die While-Schleife durch eine entsprechende LOOP-Schleife ersetzen.

Daher sind alle Problemstellungen in **P** auch LOOP-Berechenbar.

Das Argument trifft auf alle Zeit-Beschränkungen $f(n)$ wie z.B. 2^n, 2^{2^n}, $2^{2^{\cdot^{\cdot^{2}}}}\}n$ zu, sofern

sie selber LOOP-berechenbar sind. Die LOOP-berechenbaren Funktionen umfassen also noch weit mehr als nur P.



Nichtdeterministische Turingmaschinen:

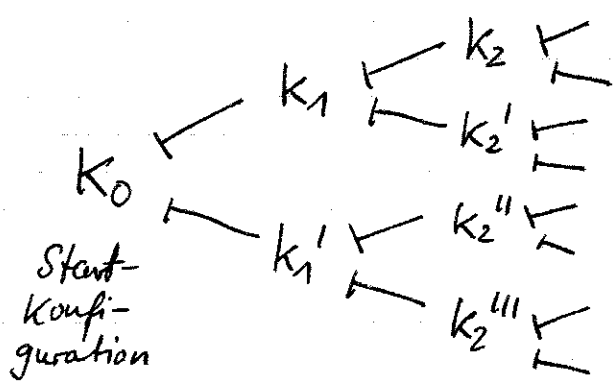
$$\delta : Z \times \Gamma \longrightarrow \mathcal{P}(Z \times \Gamma \times \{L, R, N\})$$

Z: Zustände, Γ: Arbeitsalphabet

d.h. mehrere Aktionen sind simultan möglich. OBdA: Es genügt 2 Alternativen zu betrachten.

Beispiel: $\delta(z, a) = \{(z', b, R), (z'', c, L)\}$

Der Rechenablauf einer nichtdet. TM sieht typischerweise so aus:



Nach t Schritten können 2^t verschiedene Konfigurationen auftreten.

Eine Eingabe x gilt als <u>akzeptiert</u> ($x \in T(M)$), falls es nur einen Rechenablauf gibt, der x akzeptiert (also in einen akzeptierenden Endzustand übergeht). Die Definition von Rechenzeit bei nichtdet. TM ist ebenso „großzügig" wie die Definition des Akzeptierens:

$$ntime_M(x) := \begin{cases} \min(\text{Länge einer akzeptierenden Rechnung, bei Eingabe } x), & x \in T(M) \\ 0, & x \notin T(M) \end{cases}$$

$$NTIME(f(n)) := \{ A \mid A \text{ wird von einer nichtdet. TM } M \text{ akzeptiert } (A = T(M)) \text{ so dass } \forall x: ntime_M(x) \leq f(|x|) \}$$

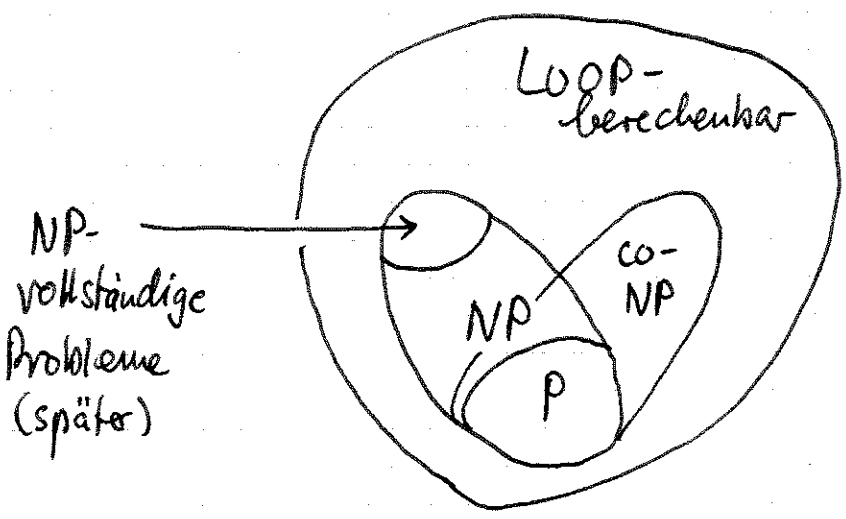
$$NP := \bigcup_{\substack{p \text{ ist} \\ \text{Polynom}}} NTIME(p(n))$$

Aus den Definitionen ergibt sich $P \subseteq NP$.
Offen ist das sog. P-NP-Problem, nämlich,
ob $P = NP$ oder $P \neq NP$ (also $P \subsetneq NP$)
gilt. Die meisten Forscher glauben, dass $P \neq NP$
gilt. Es gibt aber auch Befürworter von $P = NP$:

Zitat aus D. Knuth: The Art of Computer Programming, Vol 4B.

* At the present time very few people believe that P = NP [see *SIGACT News* **43**, 2 (June 2012), 53–77]. In other words, almost everybody who has studied the subject thinks that satisfiability cannot be decided in polynomial time. The author of this book, however, suspects that $N^{O(1)}$-step algorithms do exist, yet that they're unknowable. Almost all polynomial time algorithms are so complicated that they lie beyond human comprehension, and could never be programmed for an actual computer in the real world. Existence is different from embodiment.

Die Welt von P und NP unter der Annahme
$P \neq NP$ sowie $NP \neq co\text{-}NP$:



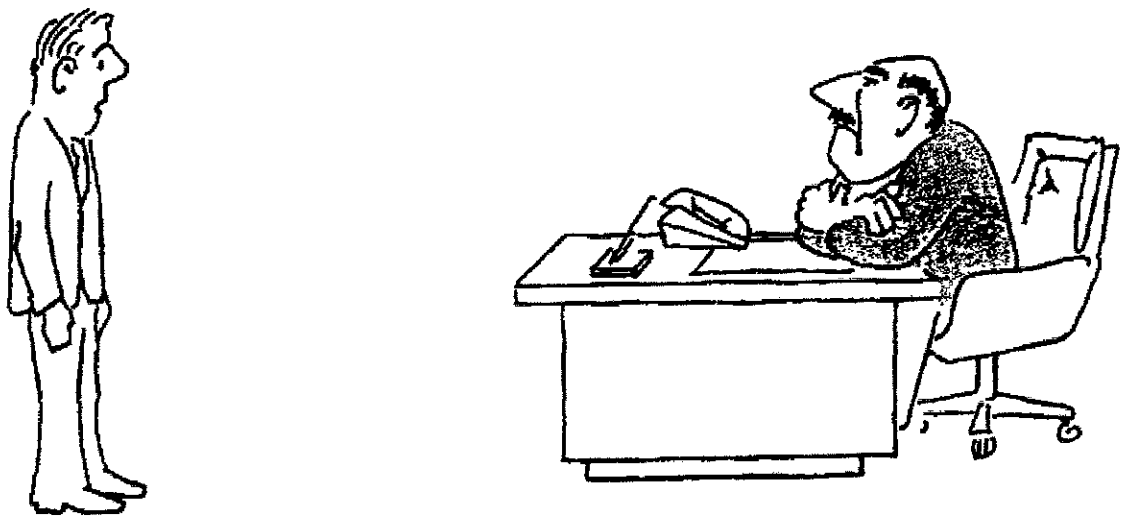
Intuition:
P verhält sich zu NP wie „entscheidbar" zu „semi-entscheidbar"

Die Vermutung $NP \neq co\text{-}NP$ $(= \{A \mid \bar{A} \in NP\})$ wird begründet durch die asymmetrische Definition des Akzeptierens und von $ntime_M(x)$ bei nichtdeterministischen Turingmaschinen.

Die Klasse P ist dagegen komplementabgeschlossen: $P = co\text{-}P$.

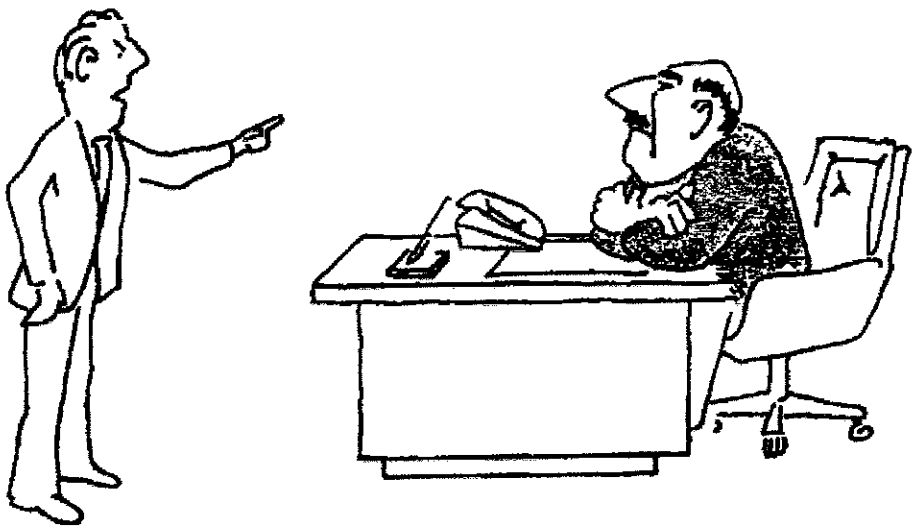
NP-Vollständigkeit

Für viele Probleme in NP sind keine polynomialen Algorithmen bekannt. Ohne die NP-Vollständigkeitstheorie stünden wir so da (Bilder aus Garey/Johnson):



"I can't find an efficient algorithm, I guess I'm just too dumb."

Ideal wäre Folgendes:



"I can't find an efficient algorithm, because no such algorithm is possible!"

Leider steht ein solcher Beweis seit ca. 50 Jahren, seit P, NP definiert wurden, aus.

Den Reduzierbarkeitsbegriff ($\leq$) aus der Berechenbarkeitstheorie übertragen in die Welt von P und NP:

Def: Es gilt $A \leq_p B$, falls es Funktion f gibt, die in polynomialer Zeit berechenbar ist, so dass für alle x gilt:

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

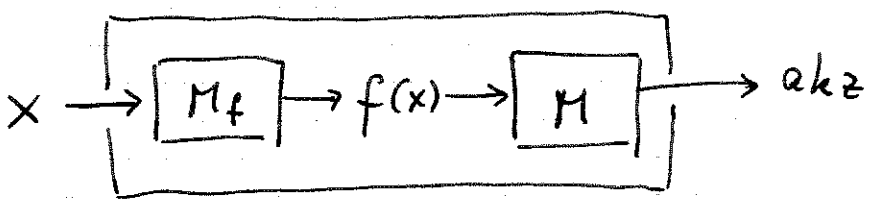
Beobachtung (ähnlich wie in Berechenbarkeitstheorie):

> Falls $A \leq_p B$ und falls $B \in (N)P$, so folgt auch, dass $A \in (N)P$.

Logisch äquivalente Kontraposition:

> Falls $A \leq_p B$ und falls $A \notin (N)P$, so folgt auch, dass $B \notin (N)P$.

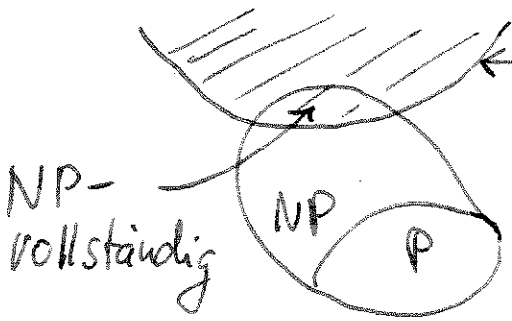
Beweis: Die Maschine M_f zur Berechnung der Reduktionsfunktion f habe Laufzeit $p(n)$. Ferner sei $B \in (N)P$ mit Laufzeit $(n)time_M(x) \leq q(|x|)$ akzeptierbar, wobei $B = T(M)$. Dann wird A durch Kombination dieser beiden Maschinen akzeptiert:



Diese Maschine hat Laufzeit $\leq \underbrace{p(n) + q(p(n))}_{\text{ebenfalls polynomial}}$ □

Def: Ein Problem A heißt **NP-hart**, falls für alle Probleme $L \in NP$ gilt: $L \leq_p A$.

Wenn darüber hinaus A selber in NP liegt, so heißt A **NP-vollständig**.

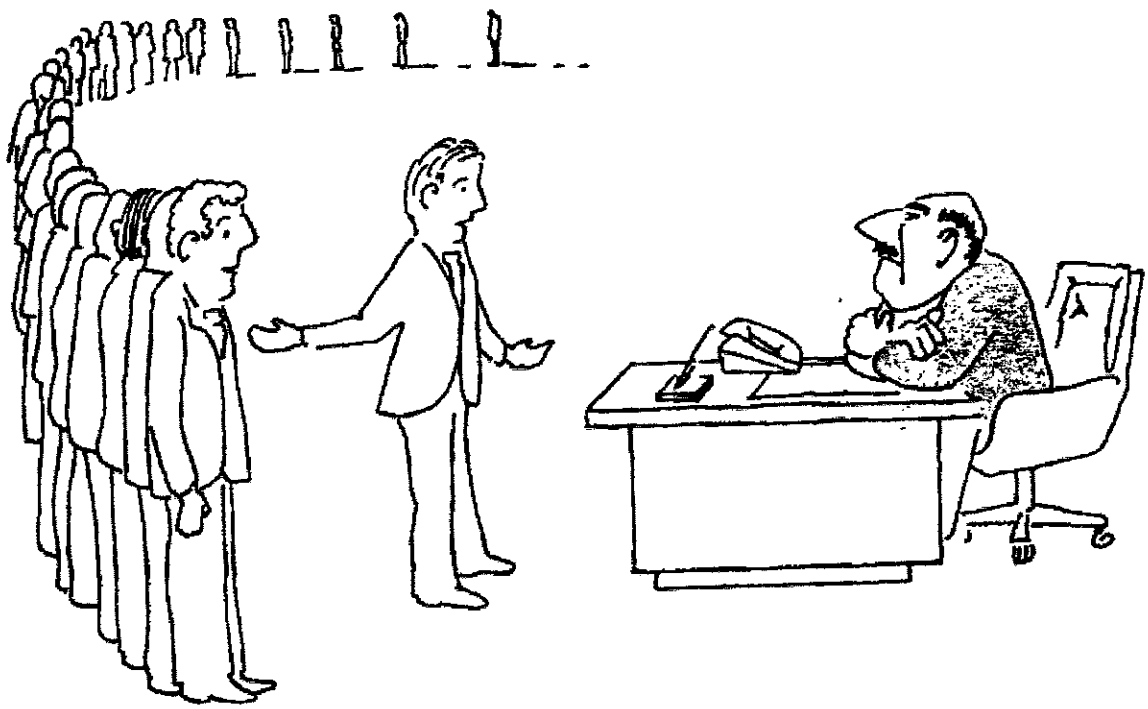


Der Status eines einzigen NP-vollständigen Problems A entscheidet das P-NP-Problem

$A \in P \Rightarrow P = NP$

$A \notin P \Rightarrow P \neq NP$

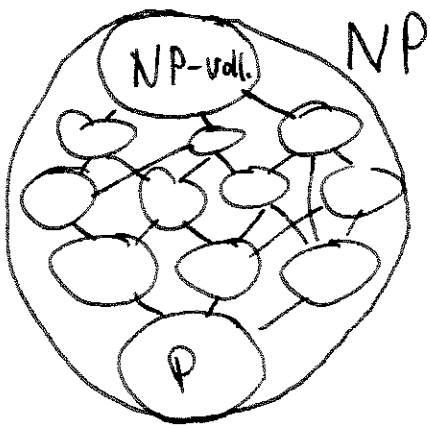
An polynomialen Algorithmen für NP-vollständigen Problemen haben sich viele Forscher vergeblich versucht.



"I can't find an efficient algorithm, but neither can all these famous people."

Die Relation $\leq_p$ definiert eine partielle Ordnung. Für je zwei NP-vollständige Probleme A und B gilt: $A \leq_p B$ und $B \leq_p A$. Dasselbe gilt für je zwei Probleme in P. Sofern $P \neq NP$, so gibt es unendlich viele Äquivalenzklassen zwischen P (dem kleinsten Element in NP) und NP-vollst. (dem größten Element in NP):

Falls $P \neq NP$:



der einzige „natürliche" Kandidat für ein Problem in diesem Zwischenbereich ist GRAPH ISOMORPHIE

$= \{(G_1, G_2) \mid G_1 \text{ und } G_2 \text{ sind isomorph}\}$

Es folgt eine Auflistung einiger bekannter NP-vollständiger Probleme (vgl. Garey/Johnson: Computers and Intractability – A Guide to the Theory of NP-Completeness):

- Das Erfüllbarkeitsproblem der Aussagenlogik SAT =
  { F | F ist erfüllbare Boole'sche Formel }

Auch Varianten bzw. Einschränkungen des SAT-Problems sind NP-vollständig:

KNF-SAT : wie SAT, Formel muss in KNF vorliegen

k-KNF-SAT (kurz: k-SAT) : wie KNF-SAT, jede Klausel hat $\leq k$ Literale
($k \geq 3$)

- Das Cliquenproblem bei Graphen: CLIQUE =
  {(G, k) | G ist Graph, $k \in \mathbb{N}$, G enthält einen vollständigen Teilgraphen bestehend aus k Knoten (einen K_k) }

- Das Hamiltonkreis-Problem bei Graphen: HAM =
  { G | G besitzt Hamiltonkreis, also Kreis auf dem jeder Knoten genau einmal vorkommt }

Eng mit diesem Problem verwandt ist das TRAVELLING SALESMAN - Problem

Gegeben ein Graph (es kann auch ein vollständiger Graph sein). Jede Kante hat eine Bewertung (eine Entfernung). Ferner ist Zahl $k \in \mathbb{N}$ gegeben. Stelle fest, ob es eine Rundreise (einen Hamiltonkreis) gibt, dessen Gesamt-Entfernung $\leq k$ ist.

- Das Färbungsproblem bei Graphen: COLOR = $\{(G,k) \mid G$ ist ein Graph, der sich mit k Farben einfärben lässt.$\}$

  Dies bedeutet: es gibt $f: V \to \{1,2,\ldots,k\}$ so dass für alle Kanten $\{x,y\} \in E$ gilt: $f(x) \neq f(y)$. Sogar der Spezialfall der 3-Färbbarkeit ist bereits NP-vollständig.

- SUBSET SUM, eine einfache Version des Rucksack-Problems. Gegeben sind Zahlen $a_1, a_2, \ldots, a_n, b \in \mathbb{N}$. Stelle fest, ob es $I \subseteq \{1,\ldots,n\}$ gibt so dass $\sum_{i \in I} a_i = b$.

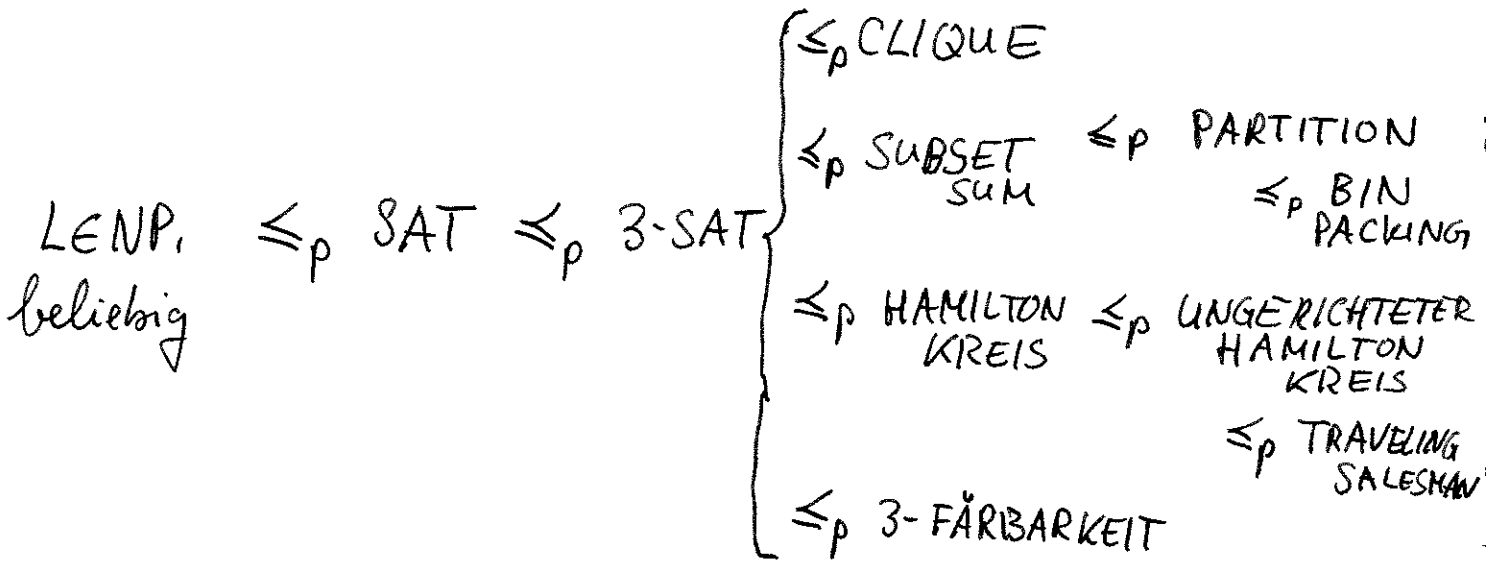
- Ein sehr ähnliches zahlentheoretisches Problem ist $\text{PARTITION} = \{ (a_1, \ldots, a_n) \in \mathbb{N}^n \mid$ es gibt Teilmenge $I \subseteq \{1, \ldots, n\}$ so dass $\sum_{i \in I} a_i = \sum_{j \notin I} a_j \}$

- BIN PACKING: Gegeben sind „Objekte" $a_1, \ldots, a_n$ die in k Behälter zu packen sind. Jeder Behälter hat Größe b.

Formal:

$\text{BIN PACKING} = \{ (a_1, \ldots, a_n, k, b) \in \mathbb{N}^{n+2} \mid$ es gibt $f: \{1, \ldots, n\} \to \{1, \ldots, k\}$ so dass für $j = 1, \ldots, k$ gilt: $\sum_{\substack{\text{alle } i \\ \text{mit } f(i)=j}} a_i \leq b \}$

Die verschiedenen NP-Vollständigkeitsnachweise sollen wie folgt nachgewiesen werden:



Später werden noch einige Lösungsmöglichkeiten für NP-vollständige Probleme diskutiert wie Approximationsalgorithmen (zwar polynomial, erreichen aber nicht notwendiger Weise die optimale Lösung) und gemäßigt exponentiale Algorithmen (statt Laufzeit 2^n bei naiver Vorgehensweise eine bessere Laufzeit c^n, $c>1$, z.B. $c=1.4$).

Die NP-vollständigen Probleme sind per Definition Entscheidungsprobleme (ja/nein-Probleme). Dahinter steckt oft die natürlichere Formulierung als

Optimierungsproblem.

Beispiel: TRAVELING SALESMAN als Optimierungsproblem

geg.: Graph $G=(V,E)$ mit Kantenbewertungen

$$f: E \to \mathbb{N}$$

gesucht: Eine „Rundreise", also eine Permutation

$\pi \in S_n$, $n=|V|$, so dass

$$(\pi(1), \pi(2)) \in E, \ldots, (\pi(n-1), \pi(n)) \in E, (\pi(n), \pi(1)) \in E$$

---

und $\underbrace{f(\pi(1), \pi(2)) + \ldots + f(\pi(n-1), \pi(n)) + f(\pi(n), \pi(1))}_{\text{abgekürzt: } f(\pi)} \to \min!$

Gewissermaßen künstlich wird daraus ein ja/nein-Problem gemacht, indem man definiert:

$$\text{TRAVELING SALESMAN} = \{ (G, f, k) \mid G \text{ ist Graph, } f \text{ ist Kantenbewertung, } k \in \mathbb{N}, \text{ und es gibt eine Rundreise } \pi \text{ in } G \text{ mit Gesamtlänge } f(\pi) \leq k \}$$

In ähnlicher Weise sind die Probleme

CLIQUE: Finde im Graph G die größte Clique

und

FÄRBBARKEIT: Finde für den Graph G eine Färbung mit minimaler Farbenzahl

eigentlich Optimierungsprobleme. Als Entscheidungsprobleme formuliert:

$$\text{CLIQUE} = \{ (G,k) \mid \text{es gibt in } G \text{ eine Clique der Größe } k \}$$

$$\text{FÄRBBARKEIT} = \{ (G,k) \mid \text{es gibt eine Färbung von } G \text{ mit } k \text{ Farben} \}$$

Interessant ist hier allerdings ein gewisser Unterschied:

Während für jedes k das Problem

$$k\text{-CLIQUE} = \{ G \mid \text{es gibt in } G \text{ eine Clique der Größe } k \}$$

in P liegt, da es nur $\binom{n}{k} = O(n^k)$ viele Möglichkeiten für eine solche Clique gibt, die man alle systematisch durchprobieren kann.

Für das Problem

$$k\text{-Färbbarkeit} = \{ G \mid \text{es gibt Färbung von } G \text{ mit } k \text{ Farben} \}$$

gilt, dass dieses NP-vollständig ist, sofern $k \geq 3$.

Es gibt noch einige Optimierungsprobleme, die sehr nahe mit den bisherigen Entscheidungsproblemen verwandt sind, z.B.

MAX (k) SAT: gegeben Formel in (k-) KNF, finde eine Belegung, die eine <u>maximale</u> Anzahl der vorhandenen Klauseln erfüllt.

Zum „Warmwerden" zeigen wir einige der polynomialen Reduktionen, bevor wir zum sog. Cook'schen Satz: $L \in NP$, beliebig $\leq_p$ SAT
(auch: Satz von Cook und Levin
1971 1973)
kommen.

Es gilt: SAT $\leq_p$ 3-SAT

Gegeben eine (beliebig verschaltete) Boolesche Formel F. Man soll eine Abbildung $F \mapsto F'$ angeben so dass gilt:

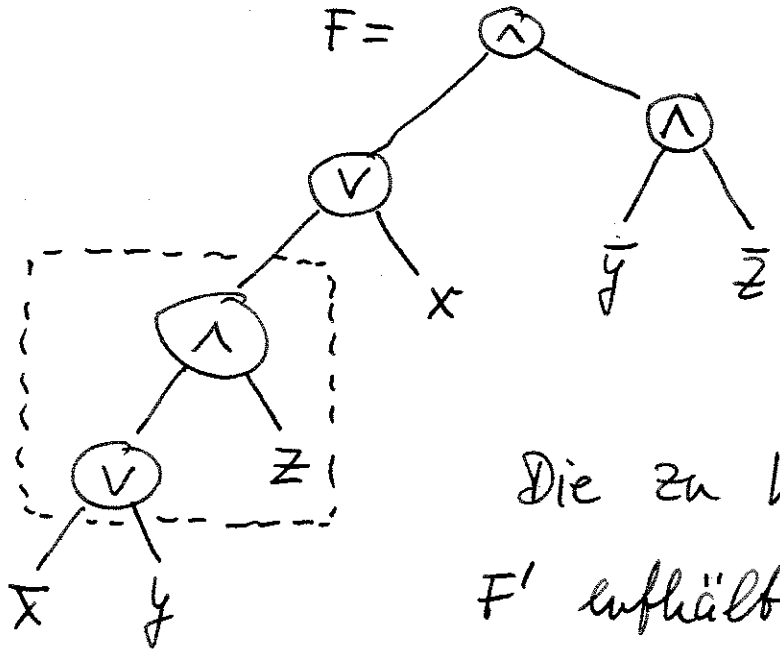
F ist erfüllbar $\iff$ F' ist eine Formel in 3-KNF und F' ist erfüllbar.

Zunächst kann man annehmen, dass die Negationen in F alle bis direkt vor die Variablen geschoben wurden (mittels de Morgan-Regel).

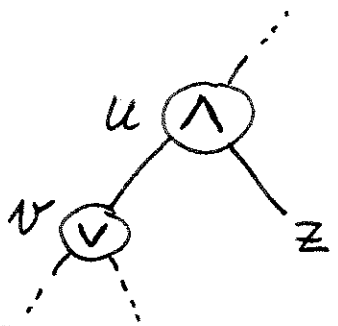
Beispiel:

$$F = (((\bar{x} \vee y) \wedge z) \vee x) \wedge (\bar{y} \wedge \bar{z})$$

F, dargestellt als Baum:



Die zu konstruierende Formel F' enthält außer den bisher vorkommenden Variablen (hier: x, y, z) weitere Variablen; eine für jeden inneren Knoten in der Baumdarstellung. Hier exemplarisch ein Ausschnitt des Baumes:



Jeder solchen Verzweigung wird eine Teilformel zugeordnet; in diesem Fall:

$$u \leftrightarrow (v \wedge z)$$

Diese Teilformeln können äquivalent in 3-KNF umgeformt werden; in diesem Fall: $(u \vee \bar{v} \vee \bar{z}) \wedge (\bar{u} \vee v) \wedge (\bar{u} \vee z)$

Sei w die Variable, die der Baumwurzel zugeordnet ist. Die Formel F' hat dann die Form:

$$F' = (w) \wedge \bigwedge_{\substack{\text{innere} \\ \text{Knoten}}} (\text{3-KNF-Darstellung der Teilformeln})$$

Man überzeugt sich davon, dass F und F' erfüllbarkeitsäquivalent sind. Außerdem ist die Umformung $F \mapsto F'$ in Polynomialzeit möglich. Es folgt: $SAT \leq_p 3\text{-}SAT$.

Es gilt: $3\text{-}SAT \leq_p CLIQUE$

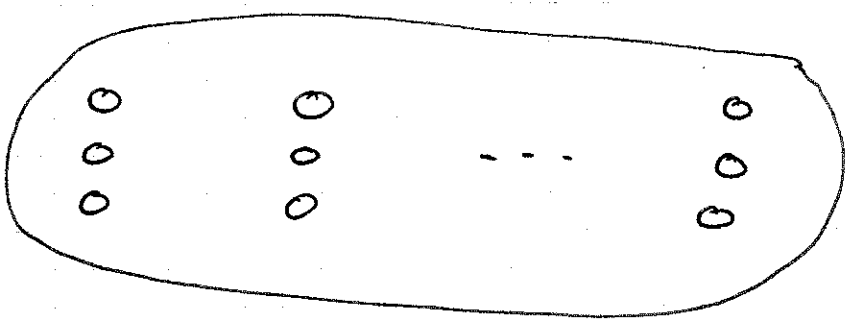
Gegeben eine Formel F in 3-KNF muss ein Graph G konstruiert werden und eine Zahl k so dass gilt:

F ist erfüllbar $\Leftrightarrow$ G enthält eine Clique der Größe k

Wenn F eine Formel ist mit n Variablen und m Klauseln; jede Klausel bestehend aus genau 3 Literalen (Variable oder negierte Variable), so wird der Graph G insgesamt 3m Knoten besitzen:

Skizze: $F = K_1 \wedge K_2 \wedge \dots \wedge K_m$

$\downarrow I$

$G =$ 

Die Kanten in G werden folgendermaßen definiert:

- innerhalb derselben Klausel K_i gibt es keine Kantenverbindung zwischen den 3 Knoten
- zwischen Knoten, die verschiedenen Klauseln entstammen, sind alle Kantenverbindungen vorhanden
  - bis auf den Sonderfall, wenn die betreffenden Literale (die den Knoten entsprechen) komplementär zueinander sind.

Die theoretisch denkbare, maximale Cliquen-größe in G ist m, und zwar je ein Knoten aus jedem Klausel-Teilgraphen. Dies klappt aber nur, wenn dabei keine komplementären Literale beteiligt sind. Man sieht leicht ein, dass:

F ist erfüllbar $\iff$ G besitzt eine Clique der Größe m

Also ist $F \mapsto (G, m)$ die gesuchte Reduktion (die polynomial berechenbar ist), welche beweist, dass $3\text{-SAT} \leq_p \text{CLIQUE}$.

Bem: Die Cliquengröße m (= Anzahl der Klauseln) konnte natürlich (aufgrund der obigen Diskussion) keine Konstante sein.

Zur Abwechslung hier ein gemäßigt exponentieller Algorithmus für 3-SAT (Monien-Speckenmeier 1985):

```
procedure test (F: 3-KNF)
// liefert 1, falls F erfüllbar ist
If (F enthält keine Variablen mehr, also
    F ∈ {0,1}) then return F;
Sei K = (x v y v z) eine Klausel in F.
if test(F|_{x=1}) then return 1;
if test(F|_{x=0,y=1}) then return 1;
if test(F|_{x=0,y=0,z=1}) then return 1;
return 0
```

Die Prozedur enthält (bis zu) 3 rekursive Aufrufe. Startet man mit einer Formel mit n Variablen, so enthalten die rekursiven Aufrufe $n-1$, $n-2$ bzw. $n-3$ Variablen. Zur Analyse der Laufzeit muss man die Rekursion $T(n) = T(n-1) + T(n-2) + T(n-3)$ untersuchen. Es ergibt sich: $T(n) = O(1{,}84^n)$

Ein weiterer, diesmal probabilistischer Algorithmus für 3-SAT (Schö, 1999):

Eingabe: Formel $F \in$ 3-KNF mit n Variablen

Rate per Zufall eine Belegung $a_1 a_2 \ldots a_n \in \{0,1\}^n$

Solange keine erfüllende Belegung gefunden wird, tue für n Schritte Folgendes:

Wähle eine Klausel $K = (x \vee y \vee z)$ aus, die unter der aktuellen Belegung nicht erfüllt wird (also $x=y=z=0$).

Wähle jeweils mit Wahrscheinlichkeit $\frac{1}{3}$ x oder y oder z aus und ändere die aktuelle Belegung, indem x bzw. y bzw. z auf 1 gesetzt wird.

Man kann die W'keit, dass dieser Algorithmus eine erfüllende Belegung findet (vorausgesetzt die Formel ist tatsächlich erfüllbar) wie folgt abschätzen:

Zufallsvariable: X = Hamming-Abstand der Anfangs-Belegung $a_1 \dots a_n$ zu einer (festen) erfüllenden Belegung. Es ist $X \in \{0, 1, \dots, n\}$ wobei

$$P(X = k) = \binom{n}{k} \left(\frac{1}{2}\right)^k .$$ D.h. X ist

binomialverteilt: $X \sim Bin(n, \frac{1}{2})$.

Nun zur Analyse der n nachfolgenden Schritte: Mit Wahrscheinlichkeit $\frac{1}{3}$ kann pro Schritt der Hamming-Abstand um 1 verkleinert werden. Mit W'keit $\frac{2}{3}$ (im worst case) kann sich der Hamming-Abstand aber auch vergrößern.

Zufallsvariable: Y = Anzahl der Schritte, bei denen sich der Hamming-Abstand zur erfüllenden Belegung vergrößert.

Y ist binomialverteilt: $Y \sim Bin(n, \frac{2}{3})$, d.h.

$$P(Y = k) = \binom{n}{k} \cdot \left(\frac{2}{3}\right)^k \cdot \left(\frac{1}{3}\right)^{n-k}$$

Nun gilt:

$P($Algorithmus findet erfüllende Belegung$) \geq$
(sofern F erfüllbar)

$$\geq P(X=\tfrac{n}{3}) \cdot P(Y=\tfrac{n}{3})$$

$$= \binom{n}{n/3}\left(\tfrac{1}{2}\right)^{n/3} \cdot \binom{n}{n/3}\left(\tfrac{2}{3}\right)^{n/3}\left(\tfrac{1}{3}\right)^{2n/3}$$

Bis auf einen kleinen vernachlässigbaren Term kann man Binomialkoeffizienten der Form $\binom{n}{\alpha n}$ wie folgt abschätzen: $\binom{n}{\alpha n} \approx \left(\left(\tfrac{1}{\alpha}\right)^{\alpha} \cdot \left(\tfrac{1}{1-\alpha}\right)^{1-\alpha}\right)^{n}$

Dies eingesetzt erhalten wir:

$$\approx \left(3^{1/3} \cdot \left(\tfrac{3}{2}\right)^{2/3}\right)^{n} \cdot \left(\tfrac{1}{2}\right)^{n} \cdot \left(3^{1/3} \cdot \left(\tfrac{3}{2}\right)^{2/3}\right)^{n} \cdot \left(\tfrac{2}{3}\right)^{n/3} \cdot \left(\tfrac{1}{3}\right)^{2n/3}$$

$$= \left(\tfrac{3}{4}\right)^{n}.$$

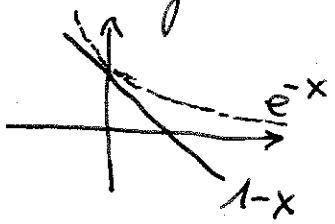
Das heißt, ein einziger Lauf des Algorithmus hat noch keine so große Chance, eine erfüllende Belegung zu finden. Aber angenommen, man wiederholt diesen „Basisalgorithmus" mit unabhängigen Zufallszahlen $c \cdot \left(\tfrac{4}{3}\right)^{n}$ -mal, so beträgt die W'heit trotz vorhandener erfüllender

Belegung diese nicht zu finden, höchstens

$$\left(1-\left(\frac{3}{4}\right)^n\right)^{c\cdot\left(\frac{4}{3}\right)^n} \leq e^{-\left(\frac{3}{4}\right)^n \cdot c \cdot \left(\frac{4}{3}\right)^n} = e^{-c}$$

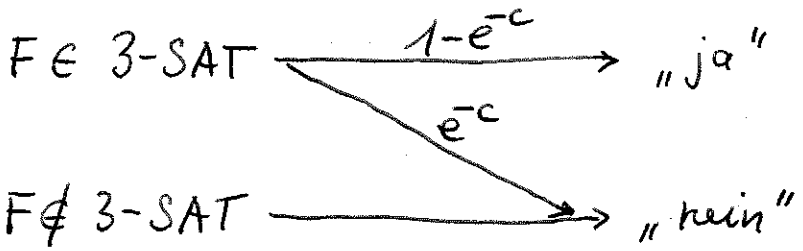
Hier wurde die bekannte Ungleichung

$$1-x \leq e^{-x}$$



verwendet.

Das bedeutet, wir haben einen probabilistischen Algorithmus für 3-SAT mit Laufzeit $O(1{,}334^n)$, dessen Fehlerwahrscheinlichkeit beliebig klein gehalten werden kann.



Aufgabe: Man gebe eine polynomiale Reduktion an

von

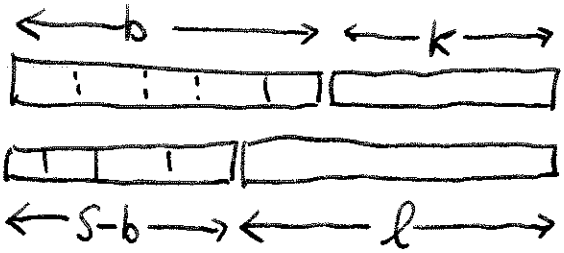
$$\text{SUBSET SUM} = \{(a_1,\dots,a_n,b) \in \mathbb{N}^{n+1} \mid \exists I \subseteq \{1,\dots,n\}: \sum_{i\in I} a_i = b\}$$

nach

$$\text{PARTITION} = \{(a_1,\dots,a_n) \in \mathbb{N}^{n} \mid \exists I \subseteq \{1,\dots,n\}: \sum_{i\in I} a_i = \sum_{j\notin I} a_j\}$$

Antwort: Sei $S = \sum_{i=1}^{n} a_i$ für das gegebene SUBSET SUM - Problem. Sollte $S = 2b$ gelten, so ist $(a_1,\dots,a_n,b) \longmapsto (a_1,\dots,a_n)$ eine Reduktion von SUBSETSUM nach PARTITION.

Nehmen wir nun an, $S \neq 2b$.
Dann könnte das Szenario so aussehen:



Man benötigt 2 „Schlupfzahlen" k und l, wobei:

$$b + k \stackrel{!}{=} S - b + l$$

Nun muss man k, l groß genug wählen, etwa

$k = S$, dann ergibt sich $\ell = 2b$.

„Groß genug" deshalb, dass es nicht möglich ist k und ℓ in dieselbe Menge I zu setzen.

Die Gesamtsumme von $a_1, \dots, a_n, k, \ell$ ist $S + S + 2b = 2 \cdot (S+b)$. Das bedeutet, dass es nicht möglich ist, eine PARTITION-Lösung zu erreichen mit k und ℓ in derselben Gruppe.

Daher folgt:

Wenn $(a_1, \dots, a_n, b)$ eine SUBSET SUM-Lösung I besitzt, also $\sum_{i \in I} a_i = b$, dann ist dies eine Lösung von Partition: $\underbrace{\underbrace{\sum_{i \in I} a_i + \underbrace{k}_{=S}}_{=b+S} = \underbrace{\sum_{i \notin I} a_i + \underbrace{\ell}_{=2b}}_{= S - b + 2b = b + S}}$

Ebenso gilt die Umkehrung: Wenn $J \subseteq \{1, \dots, n\}$ eine PARTITION-Lösung von $(a_1, \dots, a_n, k, \ell)$ ist, dann ist entweder J oder $\overline{J}$ eine SUBSET SUM-Lösung für $(a_1, \dots, a_n, b)$, □

und zwar dasjenige, das k enthält.

Ein gemäßigt exponentieller Algorithmus für PARTITION:

Zerlege die Zahlenmenge $(a_1, \ldots, a_n)$ in 2 gleichgroße Hälften: $A = (a_1, \ldots, a_{n/2})$, $B = (a_{n/2+1}, \ldots, a_n)$

Sei $S = \sum_{i=1}^{n} a_i$. Ermittle für alle möglichen Teilmengen von A deren Summen:

$$\left\{ \sum_{i \in I} a_i \;\middle/\; I \subseteq \{1, \ldots, \tfrac{n}{2}\} \right\} = M$$

Dies sind $2^{n/2}$ Zahlen. Speichere diese in einer effizient abfragbaren Datenstruktur (Suchbaum oder Hashtabelle).

Führe folgende Schleife durch:

for all $I \subseteq \{\frac{n}{2}+1, \ldots, n\}$ do

if $\frac{S}{2} - \sum_{i \in I} a_i \in M$ then „Lösung gefunden"

Die Laufzeit dieser Prozedur ist im wesentlichen

$$2^{n/2} \approx 1{,}414^n .$$

(1971) (1973)

Satz von Cook und Levin: SAT ist NP-vollständig.

Beweis: Sei L beliebige Sprache in NP. Das heißt, es gibt eine polynomiale nichtdeterministische Turingmaschine M, die L akzeptiert. Sei $p(n)$ ein Polynom, das die Länge von Rechnungen, bei Eingaben der Länge n, beschränkt. Also:

$x \in L \Leftrightarrow$ es gibt eine Rechnung von M der Länge $p(|x|)$, die mit akzeptierendem Zustand endet.

Wir drücken „$x \in L$" in Form der Erfüllbarkeit einer gewissen Booleschen Formel F aus. Dann ist $x \mapsto F$ die gesuchte Reduktion von L nach SAT. Die Formel F enthält eine Reihe von Booleschen Variablen, die einen solchen Rechenablauf von M zerlegen in bit-weise binäre Entscheidungen.

Variable	intendierte Bedeutung
$z_{t,i}$	im Zeitpunkt t befindet sich M im Zustand i
$p_{t,j}$	im Zeitpunkt t befindet sich der Schreib-Lese-Kopf auf Bandposition j.
$b_{t,j,k}$	im Zeitpunkt t befindet sich auf dem Band an Position j das k-te Zeichen des Arbeitsalphabets.

Die gesuchte Formel F setzt sich aus verschiedenen Teilformeln zusammen:

$$F = A \wedge E \wedge \ddot{U}_1 \wedge \ddot{U}_2 \wedge R$$

A beschreibt, dass zum Zeitpunkt $t=0$ die korrekte Anfangs~~positi~~konfiguration vorliegt:

$$A = z_{0,1} \wedge p_{0,1} \wedge b_{0,-p(n),k_{-p(n)}} \wedge \dots \wedge b_{0,p(n),k_{p(n)}}$$

($z_{0,1}$: „1“ ↑ Nummer des Startzustands)

wobei die k_μ diejenigen Buchstaben des Arbeitsalphabets adressieren, die an der jeweiligen Position

des Bandes stehen müssen (von Position 1 bis n die Buchstaben aus denen die Eingabe x besteht, und Blanks rechts und links davon).

Die Teilformel E besagt, dass sich M zum Zeitpunkt p(n) im akzeptierenden Endzustand befinden muss (sagen wir, dieser habe die Nummer 42):

$$E = Z_{p(n),42}$$

Die Teilformel R regelt, dass zu jedem Zeitpunkt immer nur ein Zustand aktiv sein kann, sowie der Schreiblesekopf nur auf genau einer Position j stehen kann, sowie dass an jeder Position j nur genau ein Zeichen auf dem Band sein kann. Dazu wird eine Hilfsformel $G(x_1,\dots,x_n)$ benötigt, die genau dann den Wert 1 erhält, wenn genau eine ihrer Variablen den Wert 1 hat: Wir splitten dies auf in 2 Teilformeln,

die „mindestens eine" und „höchstens eine" besagen:

$$G(x_1, \dots, x_n) = \underbrace{(x_1 \vee \dots \vee x_n)}_{\text{„mindestens eine"}} \wedge \underbrace{\bigwedge_{1 \le i < j \le n} (\overline{x_i} \vee \overline{x_j})}_{\text{„höchstens eine"}}$$

Mit Hilfe von G lässt sich R definieren:

$$R = \bigwedge_t G(z_{t,1}, z_{t,2}, \dots, z_{t,|Z|})$$

$$\wedge \bigwedge_t G(p_{t,-p(n)}, \dots, p_{t,p(n)})$$

$$\wedge \bigwedge_t \bigwedge_j G(b_{t,j,1}, \dots, b_{t,j,|\Gamma|})$$

Die Formeln $\ddot{U}_1$ und $\ddot{U}_2$ regeln das Übergangsverhalten vom Zeitpunkt t zum Zeitpunkt $t+1$. $\ddot{U}_1$ besagt, dass sich bei Positionen, auf denen der Schreib-Lese-Kopf sich zum Zeitpunkt t <u>nicht</u> befand, sich zum Zeitpunkt $t+1$ nichts geändert haben kann.

$$\ddot{U}_1 = \bigwedge_t \bigwedge_j \bigwedge_k \left(\neg p_{t,j} \wedge b_{t,j,k} \rightarrow b_{t+1,j,k} \right)$$

$\ddot{U}_2$ regelt die Situation, wenn sich der Schreib-Lesekopf an Position j befindet. Dann muss die Übergangsfunktion δ angewandt werden. Da M eine nichtdeterministische Maschine ist, kann es jeweils mehrere mögliche Aktionen gemäß δ geben (Beispiel: $\delta(z,a) = \{(z',b,R),(z'',c,L)\}$)

$$\ddot{U}_2 = \bigwedge_t \bigwedge_j \bigwedge_i \bigwedge_k \Big(z_{t,i} \wedge p_{t,j} \wedge b_{t,j,k} \rightarrow \bigvee_{\substack{(z_{i'}, a_{k'}, x) \\ \in \delta(z_i, a_k)}} \left(z_{t+1,i'} \wedge p_{t+1,j+x} \wedge b_{t+1,j,k'} \right) \Big)$$

wobei $x \in \{-1, 0, +1\}$
$\phantom{wobei x \in \{} L \quad N \quad R$

Die Länge der Formel F ist polynomial in n.

Daher kann sich auch in polynomialer Zeit erzeugt werden.
Somit: $L \leq_p SAT$ für alle $L \in NP$.

Es muss noch gezeigt werden, dass $SAT \in NP$.
Ein nichtdeterministischer Algorithmus kann bei Eingabe einer Boole'schen Formel F wie folgt arbeiten: Er verwendet den Nichtdeterminismus, um für jede vorkommende Variable x_i einen Wahrheitswert $\in \{0,1\}$ auszuwählen. Bei n Variablen ergibt dies 2^n mögliche Rechenverläufe.

Auf dem jeweiligen Rechenverlauf wird ausgerechnet, ob die nichtdeterministisch „geratene" Belegung die Formel erfüllt. Wenn ja, stoppt der Algorithmus in einem akzeptierenden Zustand.

Solcherart lässt sich für alle NP-Vollständigen Probleme auf der Liste der Nachweis führen, dass sie in NP liegen. (Man nennt dies „guess-and-check-Methode".)

Reduktion von 3SAT nach SUBSET SUM:

Zu gegebener Formel $F = K_1 \wedge \dots \wedge K_m \in 3KNF$ muss eine Menge von Zahlen $(a_1, \dots, a_\ell, b)$ angegeben werden, so dass F erfüllbar ist genau dann, wenn man eine Menge von a_i's auswählen kann, die in der Summe genau b ergibt.

Die Summen-Zahl b hat (im Dezimalsystem) die Form

$$b = \underbrace{4\,4 \dots 4}_{m\text{-mal}}\ \underbrace{1\,1 \dots 1}_{n\text{-mal}}$$

m = Anzahl Klauseln; n = Anzahl Variablen

Wir demonstrieren die Reduktion an einem konkreten Beispiel. Sei

$$F = (x_1 \vee \overline{x}_2 \vee \overline{x}_4) \wedge (x_2 \vee x_3 \vee \overline{x}_4) \wedge (\overline{x}_1 \vee \overline{x}_3 \vee x_4)$$

Wir haben $n = 4$ und $m = 3$.

Somit ist $b = 4441111$.

Für jede positiv vorkommende Variable haben wir eine Zahl, die im hinteren Teil dem Variablen-Index entspricht, im vorderen Teil den Vorkommen in den verschiedenen Klauseln:

Vorkommen von x_1 : 100 1000

〃 x_2 : 010 0100

〃 x_3 : 010 0010

〃 x_4 : 001 0001

Analog für die Vorkommen der negierten Variablen:

Vorkommen von $\overline{x_1}$: 001 1000

〃 $\overline{x_2}$: 100 0100

〃 $\overline{x_3}$: 001 0010

〃 $\overline{x_4}$: 110 0001

Darüber hinaus brauchen wir ein paar „Schlupfzahlen", die dazu dienen, die vorderen m Ziffern der Summe exakt so aufzufüllen, dass sich der Wert 4 ergibt.

100 0000
010 0000
001 0000

200 0000
020 0000
002 0000

Beispielsweise ist $X_1=1, X_2=0, X_3=0, X_4=0$ eine erfüllende Belegung, die in der ersten Klausel 3 Literale, in der 2. Klausel 1 Literal und in der 3. Klausel 1 Literal wahr macht.

Es gilt:

	100 1000	für $X_1=1$
	100 0100	für $X_2=0$
	001 0010	für $X_3=0$
+	110 0001	für $X_4=0$

Zwischensumme: 311 1111

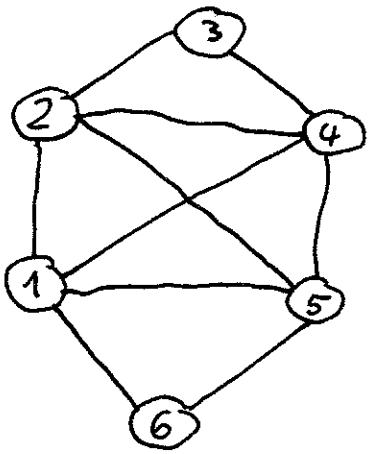
	100 0000	
	020 0000	} entsprechende Schlupfzahlen
	010 0000	
	002 0000	
+	001 0000	

444 1111 = b

Eulerkreise und Hamiltonkreise

Ein Kreis in einem Graph ist ein Pfad, der zum Anfangsknoten zurückkehrt.
Ein solcher Kreis ist ein Eulerkreis, wenn jede Kante des Graphen dabei verwendet wird (Knoten können ggf. mehrfach vorkommen).

Beispiel:



Ein möglicher Eulerkreis ist:

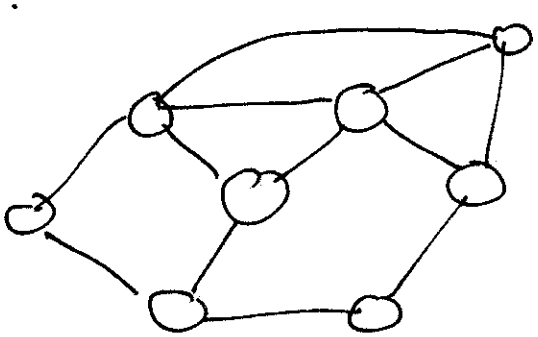
1-2-3-4-5-6-1-5-
-2-4-1

Satz (Euler 1736): Ein Graph (der zusammenhängend ist) besitzt keinen Eulerkreis

gdw.

es gibt einen Knoten im Graphen, der ungeraden Grad hat.

Bsp:



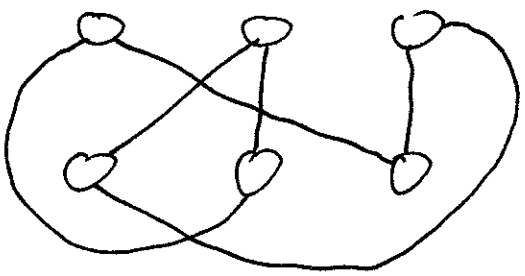
kein Eulerkreis

Dieses Kriterium ist leicht überprüfbar. Also:

$$EULER = \{ G \mid G \text{ besitzt Eulerkreis} \} \in P$$

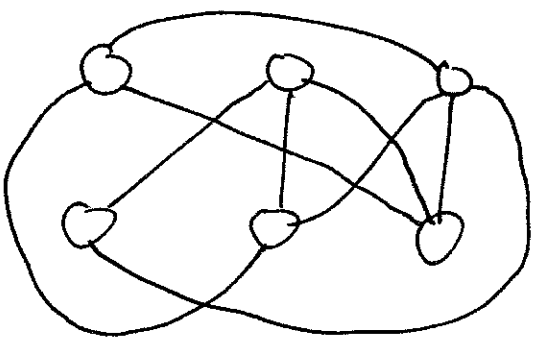
Ein <u>Hamiltonkreis</u> ist ein Kreis, auf dem jeder <u>Knoten</u> genau einmal vorkommt. (Die Kanten müssen nicht alle vorkommen.)

<u>Beispiel</u>:



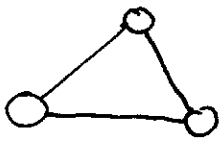
Das soll der vorgesehene Hamiltonkreis sein.

Der Graph bleibt hamiltonisch, auch wenn man weitere Kanten hinzufügt:

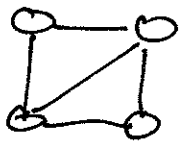


(Nur könnte der Ham. Kreis nun schwieriger zu finden sein.)

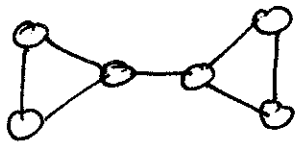
Bsp:



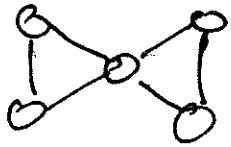
Ham.
Euler



Ham.
¬Euler



¬Ham.
¬Euler



¬Ham.
Euler

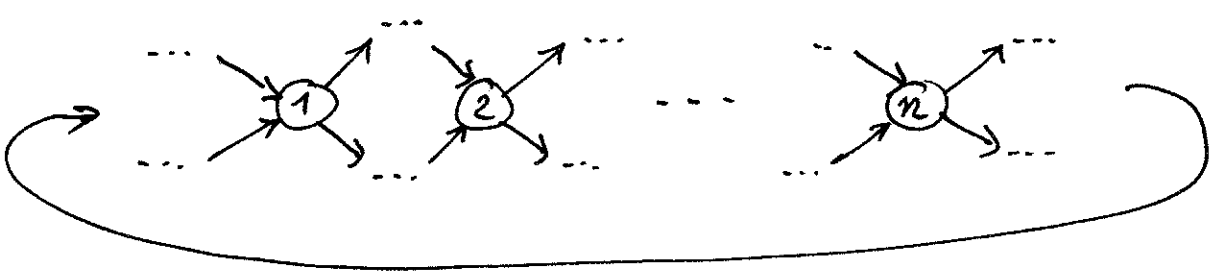
Satz: Das Problem $HAM = \{ G \mid G$ besitzt Hamiltonkreis $\}$ ist NP-vollständig.

Beweis: Wir zeigen dies zunächst für gerichtete Graphen, dann für ungerichtete.

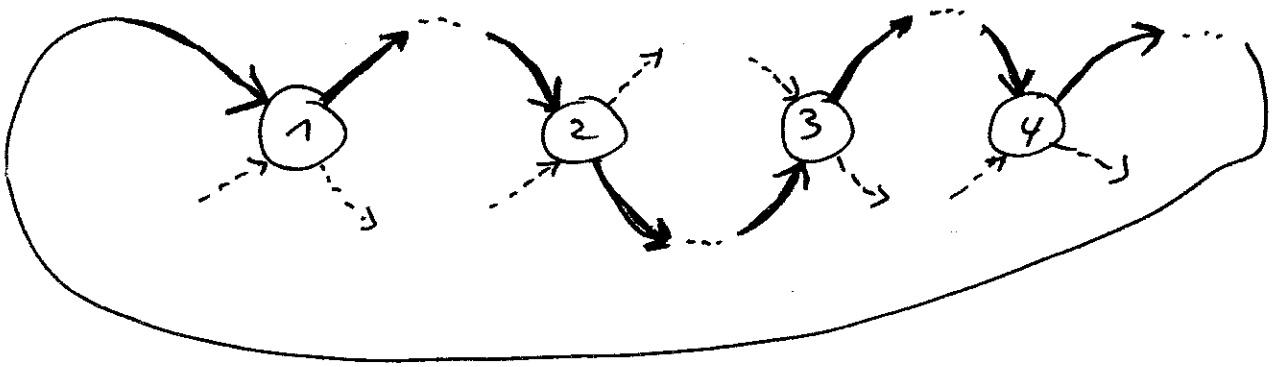
Ziel: $3\text{-SAT} \leq_p$ (gerichtet) HAM nachweisen!

Sei $F = k_1 \wedge k_2 \wedge \dots \wedge k_m$ eine Formel in KNF, wobei jede Klausel k_i aus 3 Literalen besteht.

Der zu konstruierende Graph $G = G(F)$ hat folgende Bauart:

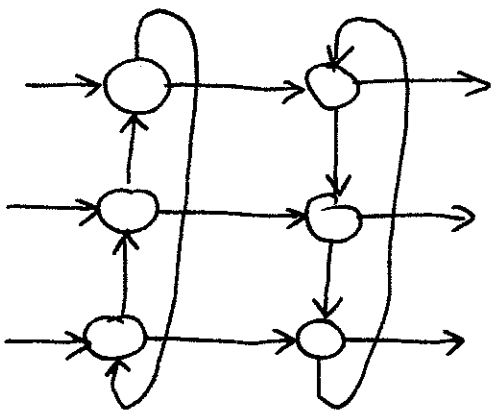


Idee: Wenn $1011 \in \{0,1\}^4$ eine erfüllende Belegung sein sollte (bei 4 Variablen), dann sollte folgender Kreis zu einem Hamiltonkreis erweiterbar sein:

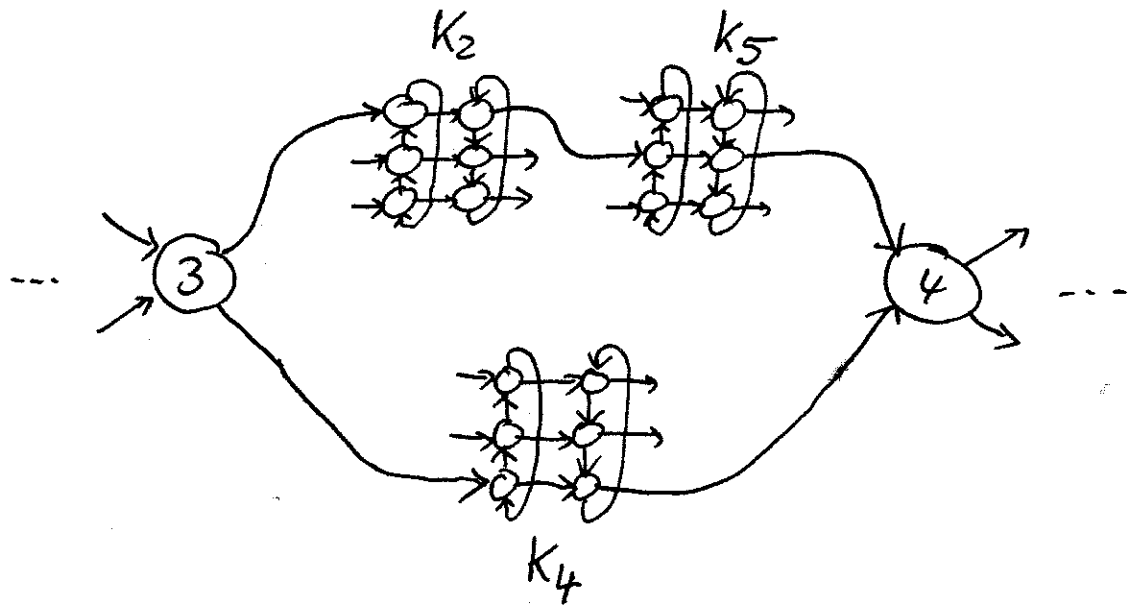


Dieser Kreis verläuft durch weitere Teilgraphen, die den Klauseln $K_1, \ldots, K_m$ zugeordnet sind:

$K_i =$



Bsp: Nehmen wir an, die Variable x_3 kommt positiv in K_2 und K_5 vor sowie negativ (also $\overline{x_3}$) in K_4. Dann wird der Knoten (3) zum Knoten (4) wie folgt verbunden:



Hierbei ist: x_3 das erste Literal in K_2,
das zweite Literal in K_5.
und $\overline{x_3}$ ist das dritte Literal in K_4.

Solcherart werden alle Pfade von (i) nach (i+1) (bzw. von (n) nach (1)) durch die Klauselgraphen K_i geführt. Damit sind alle (hinein- und hinaus- führenden) Kanten der Klauselgraphen K_i angeschlossen.

Diese Klauselgraphen haben eine wichtige Eigenschaft. Wenn eine erfüllende Belegung der Formel F vorliegt, so gibt es in jeder

Klausel mindestens 1 erfülltes Literal.

Insgesamt gibt es 7 verschiedene „Muster" wie die 3 Literale in einer Klausel k_i erfüllt sein können:

001, 010, 100, 011, 101, 110, 111

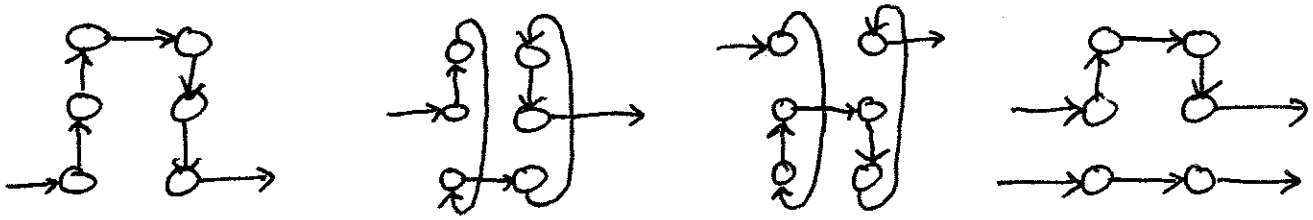
001, 010, 100: ein Literal wird wahr

011, 101, 110: 2 Literale werden wahr

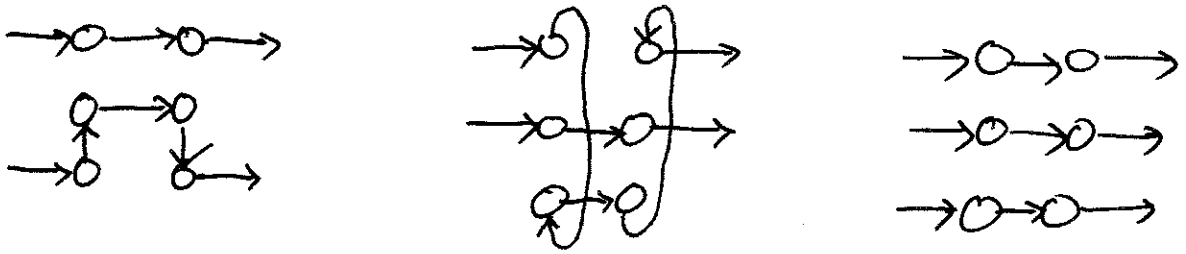
111: alle 3 Literale wahr

Für jedes dieser Muster gibt es Pfade durch die k_i – Graphen, die für die wahren Literale links beginnen und rechts gegenüber den k_i verlassen, dabei werden <u>alle</u> Knoten von k_i durchlaufen:

001 010 100 011



101 110 111

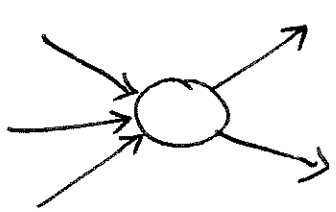


Nun kann man zeigen: Die Formel F ist erfüllbar gdw. es gibt einen Kreis durch diesen so konstruieren Graph G(F).
Also ist $F \mapsto G(F)$ eine polynomiale Reduktion von 3-SAT nach (gerichtet) HAM.

Desweiteren: (gerichtet) HAM $\leq_p$ (ungerichtet) HAM

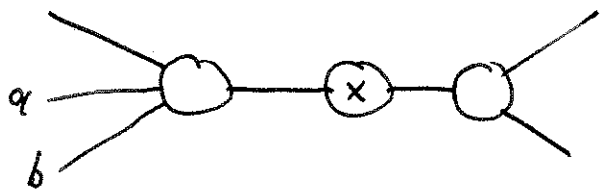
Jeder Knoten des gerichteten Graphen besitzt hereinkommende und hinausgehende Kanten

Bsp.:



Der potenzielle Hamiltonkreis könnte z.B. durch

die 2. Kante in den Knoten hereinkommen und diesen durch die obere Kante verlassen.
Um bei einem ungerichteten Graphen diese Reihenfolge zu erzwingen bzw. andere Kantenauswahlen (entgegen der Pfeilrichtung) zu verhindern, ersetzt man jeden Knoten des gerichteten Graphen durch 3 Knoten:



Sollte z.B. der Pfad von a kommend den ersten Knoten über b verlassen, so wird der Knoten x isoliert und deshalb ist in diesem Fall kein Hamiltonkreis mehr möglich.

Beim Travelling Salesman-Problem besitzen die Kanten des Graphen Entfernungsangaben.

Gesucht ist eine Rundreise mit minimaler Länge.

Man kann $HAM \leq_p TSP$ reduzieren, indem man den Graphen G mit n Knoten abbildet auf K_n (den vollst. Graphen mit n Knoten). Für jede in G vorhandene Kante $\{x,y\}$ ordnet man die Entfernung $c(x,y)=1$ und für nicht-vorhandene Kanten $\{u,v\}$ in G den Wert $c(u,v)=2$ zu.

Nun gilt:

G besitzt einen Hamiltonkreis $\iff$ es gibt in K_n eine Rundreise der Länge n.

Sehr oft sind NP-vollständige Probleme Optimierungsprobleme. Vielfach lassen sich polynomiale Algorithmen für solche Probleme angeben,

die zwar nicht unbedingt die optimale Lösung finden, die aber eine Lösung finden, die nicht allzu viel schlechter ist als die optimale. Hier gibt es verschiedene Kategorien der Approximierbarkeit bei NP-vollständigen Problemen:

$$PTAS \subseteq APX \subseteq \{\text{Probleme, für die keine polyn. Approximation möglich ist}\}$$

PTAS (polynomial-time approximation scheme):

$\forall \varepsilon > 0$ $\exists$ polyn. Algorithmus, der das fragliche Problem löst. Die berechnete Lösung ist höchstens ε schlechter als die optimale.

APX (approximable):

$\exists \varepsilon > 0$ $\exists$ polyn. Alg. wie oben.

Falls $P \neq NP$, so sind die Inklusionen echt!

$$PTAS \subsetneq APX \subsetneq \{\text{nicht approximierbar}\}$$

Das TSP ist ein Beispiel für Nicht-Approximierbarkeit – in folgendem Sinne:

Falls $TSP \in APX$, so $P = NP$.

Beweis:
Angenommen, es gibt einen polynomialen Algorithmus, der immer eine Rundreise-Lösung produziert, die höchstens 2x so lang ist wie die optimale.
Dann ließe sich das HAM.-Problem in polynomialer Zeit lösen. Man betrachte folgende Reduktion:

$$\underset{(V,E)}{G} \longmapsto K_n,$$

Kantenbewertungen: $c(x,y) = 1$ falls $\{x,y\} \in E$,
$c(x,y) =$ sehr großer Wert, falls $\{x,y\} \notin E$.

Durchgang durch die Themen der Vorlesung Berechenbarkeit und Komplexität („Was könnte in der Klausur drankommen?")

- Der „richtige" Berechenbarkeitsbegriff (Church'sche These), Äquivalenz von Turing-, Mehrband Turing-, GOTO-, WHILE-Berechenbarkeit sowie μ-Rekursivität. Konzept des Nicht-Stoppens bei partiellen Funktionen; total berechenbar Entscheidbarkeit. Semi-Entscheidbarkeit (= rekursive Aufzählbarkeit, Co-Semi-Entscheidbarkeit. Methode der Diagonalisierung, Halteproblem Paar-Codierungsfunktion $c : \mathbb{N}^2 \to \mathbb{N}$ und deren Umkehrfunktionen f und g.

- Eingeschränkte Berechenbarkeit: LOOP-Berechenbarkeit, gleichwertig: primitiv rekursiv.
  Die Ackermann-Funktion ist total-berechenbar, aber nicht LOOP-berechenbar.
  Kleene-Normalform; eine While-Schleife reicht aus.

- Halbeproblem. Reduzierbarkeit, Satz von Rice,
- Postsches Korrespondenzproblem, Unentscheidbarkeit in der Theorie Formaler Sprachen.
  Unentscheidbarkeit der Prädikatenlogik.
  Gödelscher Unvollständigkeitssatz.

- Komplexitätstheorie: polynomial versus nicht-polynomial, Nichtdeterminismus, P versus NP, Polynomiale Reduzierbarkeit, NP-Vollständigkeit, NP-hart, Cookscher Satz (SAT ist NP-vollst.), NP-Vollständigkeit weiterer Probleme (Clique, Subset Sum, Partition, Hamiltonkreis, TSP)

- Gemäßigt exponentielle Algorithmen für manche NP-vollst. Probleme

- Konzept verschiedener Arten der polynomialen Approximierbarkeit NP-vollständiger Optimierungs-Probleme (PTAS und APX).