Zusammenfassung - BuK

Turing Maschinen

$\Sigma^* = \bigcup_{k\in\mathbb{N}} \Sigma^k$ ist der Kleensche Abschluss von Σ und enthält alle Wörter über Σ: $\varepsilon, 0, 1, 00, 01, 10, \dots$

Ein Paar (x, y) liegt in der Relation $R \subseteq \Sigma^* \times \Sigma'^*$, wenn y eine zulässige Ausgabe zur Eingabe x ist

Bsp.: Relation zur Primfaktorbestimmung:

$$R = \{(x,y) \in \{0,1\}^* \times \{0,1\}^* | x = bin(q), y = bin(p), q,p \in \mathbb{N}, q \geq 2, p\ prim, p\ teilt\ q\}$$

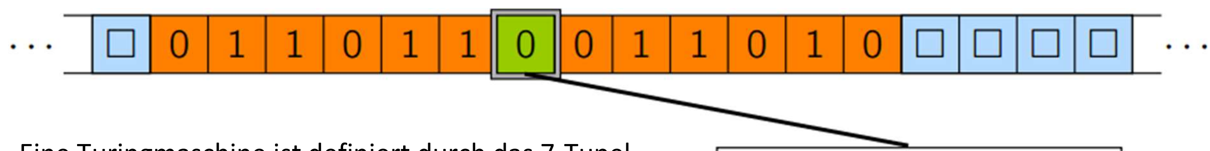
$$(110,11) \in R, aber\ (101,11) \notin R$$

Gibt es nur eine Lösung für ein Problem kann das Problem als Funktion beschrieben werden:

$f: \Sigma^* \to \Sigma'^*$. Die zur Eingabe $x \in \Sigma^*$ gesuchte Ausgabe ist $f(x) \in \Sigma'^*$.

Entscheidungsprobleme: $f: \Sigma^* \to \{0,1\}$, wobei 0 als „Nein" und 1 als „Ja" interpretiert wird.

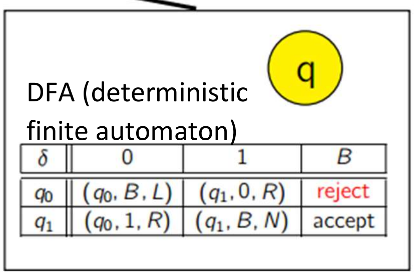
Die Sprache $L = f^{-1}(1) \subseteq \Sigma^*$ beschreibt die Menge der Eingaben, die mit „Ja" beantwortet werden



Eine Turingmaschine ist definiert durch das 7-Tupel $(Q, \Sigma, \Gamma, B, q_0, \bar{q}, \delta)$:

Q	die endliche Zustandsmenge
Σ	das endliche Eingabealphabet
$\Gamma \supset \Sigma$	das endliche Bandalphabet
$B \in \Gamma \setminus \Sigma$	das Leerzeichen (Blank, in Bildern□)
$q_0 \in Q$	der Anfangszustand
$\bar{q} \in Q$	der Endzustand

$\delta: (Q \setminus \{\bar{q}\}) \times \Gamma \to Q \times \Gamma \times \{R, L, N\}$ die Zustandsübergangsfunktion



δ	0	1	B
q_0	(q_0, B, L)	$(q_1, 0, R)$	reject
q_1	$(q_0, 1, R)$	(q_1, B, N)	accept

Der Kopf bewegt sich um eine Position nach rechts (falls R), nach links (falls L), nicht (falls N).

Die TM stoppt bei q_0. ("$reject$" $\triangleq (\bar{q}, 0, N)$, "accept" $\triangleq (\bar{q}, 1, N)$)

Die TM akzeptiert bei Entscheidungsproblemen, wenn sie terminiert und das Ausgabewort mit einer 1 beginnt und verwirft, wenn das Ausgabewort mit einer 0 beginnt.

Laufzeit = Anzahl an Zustandsübergängen bis zur Terminierung

Speicherbedarf = Anzahl von Bandzellen, die während der Berechnung besucht werden

Eine Funktion $f: \Sigma^* \to \Sigma^*$ heißt rekursiv (T-berechenbar), wenn es eine TM gibt, die aus der Eingabe x den Funktionswert $f(x)$ berechnet.

Eine Sprache $L \subseteq \Sigma^*$ heißt rekursiv (T-berechenbar), wenn es eine TM gibt, die für alle Eingaben terminiert und die Eingabe w genau dann akzeptiert, wenn $w \in L$ ist.

Eine Konfiguration einer TM ist ein String $\alpha q \beta$, mit $q \in Q$ und $a, \beta \in \Gamma^*$.

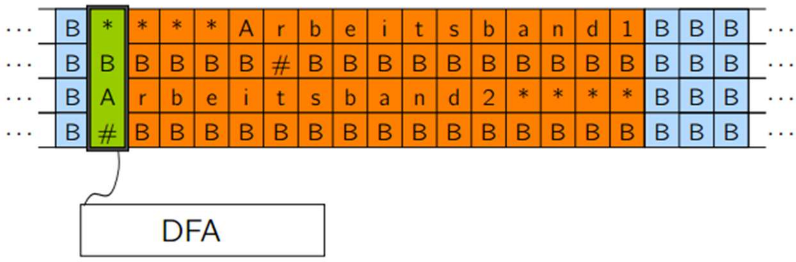
Bsp.: $q_0 0011 \ \vdash 0q_0 011 \vdash 00q_0 11 \vdash 001q_1 1 \vdash 0011q_1 B \vdash 001q_2 1$

Techniken zur Programmierung von TMs

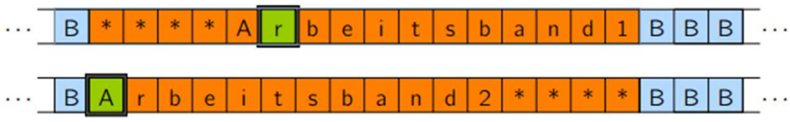
1. Speichern im Zustandsraum:

   k Zeichen ($k \in \mathbb{N}$) speichern, wenn Zustandsraum um Faktor $|\Gamma|^k$ vergrößert ($Q_{neu} \coloneqq Q \times \Gamma^k$)

2. k-spurige TM: Erweiterung des Bandalphabets um k-dimensionale Vektoren, z.B.: $\Gamma_{neu} \coloneqq \Gamma \cup \Gamma^k$



k-Band TM



TM mit k-Arbeitsbändern mit jeweils unabhängigem Kopf. # ≙ Kopfposition des jeweiligen Bandes.

Zustandsübergangsfunktion: $\delta: (Q \setminus \{\bar{q}\}) \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{R, L, N\}^k$

Band 1 fungiert als Ein- und Ausgabeband (wie TM), die Zellen der anderen Bänder sind anfangs leer.

Satz: Eine k-Band-TM M mit Zeitbedarf $t(n)$ und Platzbedarf $s(n)$ kann von einer 1-Band-TM M‘ mit Zeitbedarf $O\left(t^2(n)\right)$ und Platzbedarf $O\left(s(n)\right)$ simuliert werden.

Die universelle k-Band TM

Das „Programm" der universellen TM U ist die Kodierung (≙ $\langle M \rangle$ ≙ *Gödelnummer*) einer beliebigen TM M.

Eingabe: $\langle M \rangle w$ (TM-Kodierung + Wort)

Gödelnummer ist präfixfrei, d.h. keine Gödelnummer ist Präfix einer anderen Gödelnummer
-> jede Gödelnummer beginnt und endet mit 111 und ansonsten ist 111 nicht in der Kodierung.

Kodierung der Übergangsfunktion

Der Übergang $\delta(q_i, x_i) = (q_k, x_l, D_m)$ wird kodiert durch den Binärstring $0^i 1 0^j 1 0^k 1 0^l 1 0^m$, wobei $X_9 = 0, X_2 = 1, X_3 = B$ und $D_1 = L, D_2 = N, D_3 = R$ ist. Die Kodierung des t-ten Übergangs bezeichnen wir mit $code(t)$ -> Bsp.: $\langle M \rangle = 111\ code(1)\ 11\ code(2)\ 11 \ldots\ code(s)\ 111$

δ	0	1	B
q_1	(q_1, B, R)	(q_3, B, R)	(q_2, B, N)
q_3	$(q_1, 1, R)$	$(q_2, 0, R)$	(q_1, B, L)

111 0101010001000 11 0100100010001000 11 010001001000100 11 00010101001000 11 000100100101000 11 000100010100010 111

Implementierung der universellen TM

Eingabe: $\langle M \rangle w$ für ein beliebiges $w \in \{0,1\}^*$ als 3-Band-TM:

Band 1 von TM U simuliert das Band der TM M.
Band 2 von U enthält die Gödelnummer von M.
Auf Band 3 speichert U den jeweils aktuellen Zustand von M.

Initialisierung (Laufzeit = $O(1)$):

- Überprüfe Eingabe (korrekte Gödelnummer?)
- Kopiere $\langle M \rangle$ auf Band 2
- Schreibe Kodierung des Anfangszustands auf Band 3
- Schreibe w auf Band 1 und setze Kopf auf erstes Zeichen von w

Simulation eines Schrittes (Laufzeit = $O(1)$):

- U sucht zum Zeichen an der Kopfposition von Band 1 und dem Zustand auf Band 3 die Kodierung des entsprechenden Übergangs von M auf das Band 2.
- Wie in Übergangsfunktion beschrieben:
  - aktualisiert U die Inschrift auf Band 1
  - bewegt U den Kopf auf Band 1, und
  - verändert U den auf Band 3 abgespeicherten Zustand von M.

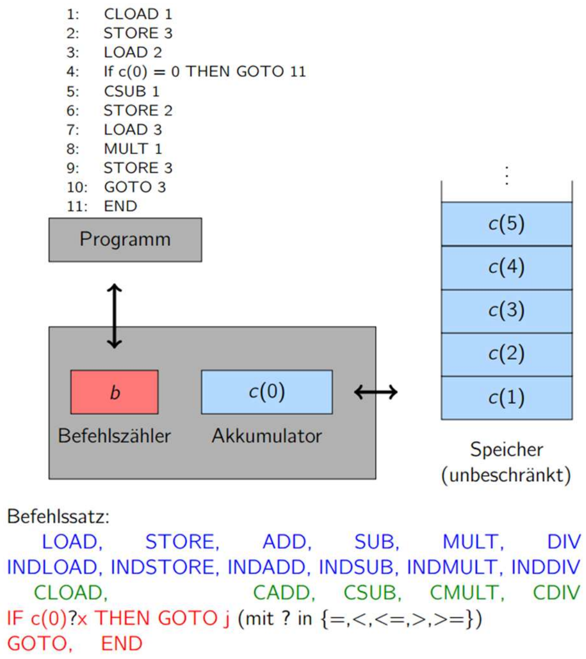
➔ konstanter Zeitverlust für Simulation.

Church-Turing These

Die Klasse der TM-berechenbaren Funktionen stimmt mit der Klasse der „intuitiv berechenbaren" Funktionen überein.

Berechenbare Funktion = TM-berechenbare Funktion = rekursive Funktion
Entscheidbare Sprache = TM-berechenbare Sprache = rekursive Sprache

Registermaschinen



C in CLOAD, CSTORE, CADD, … für "constant"
IND in INDLOAD, INDSTORE, INDADD, … für "indirect"

CLOAD 1	=> Lade Zahl 1 in c(0)
LOAD 1	=> Lade Zahl in c(1) in c(0)
INDLOAD 1	=> Lade Zahl in c(c(1)) in c(0)

LOAD i:	c(0) := c(i)
INDLOAD i:	c(0) := c(c(i))
CLOAD i:	c(0) := i
STORE i:	c(i) := c(0)
INDSTORE i:	c(c(i)) := c(0)
ADD i:	c(0) := c(0) + c(i)
SUB i:	c(0) := max{0, c(0) – c(i)}
MULT i:	c(0) := c(0) * c(i)
DIV I:	$c(0) \coloneqq \begin{cases} \lfloor c(0) \,/\, c(i) \rfloor & falls\ c(i) \neq 0 \\ 0 & sonst \end{cases}$

Auf einer RAM können wir alle Befehle höherer Programmiersprachen realisieren (z.B. Schleifen)

Uniformes Kostenmaß: Jeder Schritt zählt als eine Zeiteinheit
Logarithmisches Kostenmaß: Die Laufzeitkosten eines Schrittes sind proportional zur binären Länge der Zahlen in den angesprochenen Registern.

Satz: Für jede im logarithmischen Kostenmaß $t(n)$-zeitbeschränkte RAM R gibt es ein Polynom q und eine $q\big(n + t(n)\big)$-zeitbeschränkte TM M, die R simuliert.

Die Klasse der Polynome ist unter Hintereinanderausführung abgeschlossen Polyn.(Polyn.(x)) = Polyn.

Simulation einer RAM mit einer 2-Band TM

- Band 1 simuliert die Befehle
- Auf Band 2 wird Inhalt aller Register abgespeichert
- RAM-Programm P besteht aus p Programmzeilen
- Für jede Programmzeile schreiben wir ein TM-Unterprog. M_i für die Programmzeile $i, 1 \leq i \leq p$
- M_0 für die Initialisierung; M_{P+1} für die Aufbereitung der Ausgabe des Ergebnisses

Befehlszähler wird im Zustand gespeichert, da P konstant lang
Registerinhalte werden wie folgt auf Band 2 abgespeichert:
$\#\#0\#bin(c(0))\#\#bin(i_1)\#\ bin(c(i_1))\#\# \dots \#\#\ bin(i_m)\#\ bin(c(i_m))\#\#\#$,
wobei $0, i_1, \dots, i_m$ die Indizes der benutzten Register sind.
$O\big(n + t(n)\big)$ Platzbedarf, da für jedes neu erzeugte Bit mind. Eine Zeiteinheit benötigt wird.
Rechenschritt:
TM ruft das im Programmzähler b spezifizierte Unterprogramm M_b auf.
Unterprogramm M_b:

- Kopiert Inhalt der in b angesprochenen Register auf Band 1,
- Führt die notwendigen Operationen auf diesen Registerinhalten durch,
- Kopiert dann das Ergebnis in das in Zeile b angegebene Register auf Band 2 zurück, und
- Aktualisiert zuletzt den Programmzähler b.

Laufzeitanalyse:

- Initialisierung: $O(n)$
- Unterprogramme: $n + t(n)$ polyn. Laufzeit in der Länge des aktuellen Wortes auf Band 2

➔ Gesamtlaufzeit: $n + t(n)$

Simulation einer TM durch RAM

Jede $t(n)$-beschränkte TM kann durch eine RAM simuliert werden, die zeitbeschränkt ist durch
$O(t(n) + n)$ im uniformen Kostenmaß ($O(n)$ Initialisierung, TM-Schritte konstant)
$O((t(n) + n) * \log(t(n) + n))$ im logarithmischen Kostenmaß (Schritte * Simulation der Schritte
Simulation ≙ Kodierung der Bandpositionen, die durch $max\{n, t(n)\} \leq n + t(n)$ beschränkt sind)

- Die Zustände und Zeichen werden wie die Zellen des Bandes der TM durchnummeriert und mit ihren Nummern identifiziert, sodass sie in den Registern abgespeichert werden können.
- Register 1 speichert den Index der Kopfposition
- Register 2 speichert den aktuellen Zustand
- Die Register 3, 4, 5, … speichern die Inhalte der besuchten Bandpositionen 0, 1, 2, …

Simulation:
Auswahl des richtigen TM-Übergangs und Durchführung des TM-Übergangs:
Die RAM verwendet eine zweistufige if-Abfrage:

- Auf einer ersten Stufe von $|Q|$ vielen if-Abfragen wird der aktuelle Zustand selektiert
- Für jeden möglichen Zustand gibt es eine zweite Stufe von $|\Gamma|$ vielen if-Abfragen, die das gelesene Zeichen selektieren

Je nach Ausgang der if-Abfragen aktualisiert die RAM

- den TM-Zustand in Register 2,
- die TM-Bandinschrift in Register $c(1)$ und
- die TM-Bandposition in Register 1

Collatz-Problem

Offenes Problem: Hält die RAM (die folgendes Problem berechnet) auf allen Eingaben?

$$x \leftarrow \begin{cases} {}^{x}/_{2} & wenn\ x\ gerade \\ 3x + 1 & wenn\ x\ ungerade \end{cases}$$

Unentscheidbarkeit

Es gibt unentscheidbare Probleme, da es mehr Sprachen/Probleme als TM/Algorithmen gibt

Abzählbare Menge

- M heißt abzählbar, wenn:
  - M leer ist
  - es eine surjektive Funktion $c: \mathbb{N} \to M$ gibt
- Jede endliche Menge ist abzählbar
- Für abzählbar unendliche Mengen ist $c: \mathbb{N} \to M$ bijektiv
- Abzählbar unendliche Mengen sind gleichmächtig wie $\mathbb{N}$ ($\mathbb{N}, \mathbb{Z}, \mathbb{Q}$ sind abzählbar unendlich)

$\mathbb{N}/\mathbb{N}$	1	2	3	…
1	1/1	2/1	3/1	
2	1/2	2/2	3/2	
3	1/3	2/3	3/3	…
⋮		⋮		⋱

Bsp.:
Die Menge der ganzen Zahlen $\mathbb{Z} = \{0, -1, 1, -2, 2, \dots\}$ mit der Bijektion:

$$c(i) = \begin{cases} i/2 & falls\ i\ gerade \\ -(i+1)/2 & falls\ i\ ungerade \end{cases}$$

Die Menge der rationalen Zahlen $\mathbb{Q}$:

$$0, \frac{1}{1}, \frac{2}{1}, \frac{1}{2}, \frac{3}{1}, \dots, \frac{i}{1}, \frac{i-1}{2}, \frac{i-2}{3}, \dots, \frac{1}{i}, \dots$$

Auch Σ^* die Menge der Wörter über endlichem Alphabet Σ ist abzählbar unendlich -> die Menge der Gödelnummern -> die Menge der TMen

Notation: über $\Sigma = \{0,1\}$ bezeichnen wir das i-te Wort mit w_i und die i-te TM mit M_i

Überabzählbarkeit

Potenzmenge $P(\mathbb{N}) \triangleq$ die Menge aller Teilmengen von $\mathbb{N}$
Beweis: $P(\mathbb{N})$ ist überabzählbar durch Diagonalisierung:

- Zwecks Widerspruchs nehmen wir an, dass $P(\mathbb{N})$ abzählbar ist
- Es sei $S_0, S_1, S_2, S_3, \dots$ eine Aufzählung von $P(\mathbb{N})$
- Wir definieren eine 2-dimensionale unendliche Matrix $(A_{ij})_{ij \in \mathbb{N}}$ mit $A_{ij} = \begin{cases} 1 & falls\ j\ \in S_j \\ & sonst \end{cases}$

		0	1	2	3	4	5	6	
$\{1, 2, 4, 6, \dots\} =$	S_0	**0**	1	1	0	1	0	1	…
$\{0, 1, 2, 4, 6, \dots\} =$	S_1	1	**1**	1	0	1	0	1	…
	S_2	0	0	**1**	0	1	0	1	…
	S_3	0	1	1	**0**	0	0	1	…
	S_4	0	1	0	0	**1**	0	1	…
$S_{\text{diag}} = \{1, 2, 4, \dots\}$	S_5	0	1	1	0	1	**0**	0	…
$\bar{S}_{\text{diag}} = \{0, 3, 5, 6 \dots\}$	S_6	1	1	1	0	1	0	**0**	…
	⋮	⋮	⋮	⋮	⋮	⋮	⋮		

Diagonalmenge $S_{diag} = \{i \in \mathbb{N} | A_{i,j} = 1\}$ und das Komplement $\bar{S}_{diag} = \mathbb{N} \backslash S_{diag} = \{i \epsilon \mathbb{N} | A_{i,j} = 0\}$
$\bar{S}_{diag}$ auch Teilmenge von $\mathbb{N}$
$\bar{S}_{diag}$ kommt in der Aufzählung $S_1, S_2, S_3, \dots$ von $P(\mathbb{N})$ vor
=> es gibt eine Zeile $k \in \mathbb{N}$, sodass $\bar{S}_{diag} = S_k$

Daraus ergeben sich 2 Fälle, die zum Widerspruch führen:

Fall 1: $A_{k,k} = 1 \xRightarrow{Def.\ \bar{S}_{diag}} k \notin \bar{S}_d \Rightarrow k \in S_k \xRightarrow{Def.\ A} A_{k,k} = 0$

Fall 2: $A_{k,k} = 0 \xRightarrow{Def.\ \bar{S}_{diag}} k \notin \bar{S}_d \Rightarrow k \in S_k \xRightarrow{Def.\ A} A_{k,k} = 1$

Folglich gibt es keine derartige Aufzählung von $P(\mathbb{N})$

Unentscheidbare Probleme

Es sei $\mathcal{L}$ die Menge aller Entscheidungsprobleme über $\{0,1\}^*$
Ein Entscheidungsproblem $L \in \mathcal{L}$ ist eine Teilmenge von $\{0,1\}^*$
$\mathcal{L} \triangleq$ Menge aller Teilmengen von $\{0,1\}^*$ => $\mathcal{L} = P(\{0,1\}^*)$
$\{0,1\}^*$ hat dieselbe Mächtigkeit wie $\mathbb{N}$ => $\mathcal{L}$ hat die gleiche Mächtigkeit wie $P(\mathbb{N})$
=> $\mathcal{L}$ ist überabzählbar, aber die Menge der TM/Gödelnummern ist abzählbar
=> es existieren unentscheidbare Sprachen

Diagonalsprache

$D = \{w \in \{0,1\}^* | \, w = w_i \; und \; M_i akzeptiert \; w \; nicht\}$

Das i-te Wort w_i ist genau dann in der Diagonalsprache D, wenn die i-te TM M_i dieses Wort w_i nicht akzeptiert

	w_0	w_1	w_2	w_3	w_4	
M_0	**0**	1	1	0	1	...
M_1	1	**0**	1	0	1	...
M_2	0	0	**1**	0	1	...
M_3	0	1	1	**1**	0	...
M_4	0	1	0	0	**0**	...
⋮	⋮	⋮	⋮	⋮		

$A_{i,j} = \begin{cases} 1 & falls \; M_i \; das \; Wort \; w_j \; akzeptiert \\ 0 & sonst \end{cases}$

Die Diagonalsprache lässt sich von der Diagonale der Matrix ablesen.
Es gilt: $D = \{w_i | A_{i,i} = 0\}$

Fall 1: $w_j \in D \xRightarrow{M_j \; ents \;\; . \; D} M_j akzeptiert \; w_j \xRightarrow{Def. \; von \; D} w_j \notin D$
Fall 2: $w_j \notin D \xRightarrow{M_j \; ents \;\; . \; D} M_j akzeptiert \; w_j \; nicht \xRightarrow{Def. \; von \; D} w_j \in D$

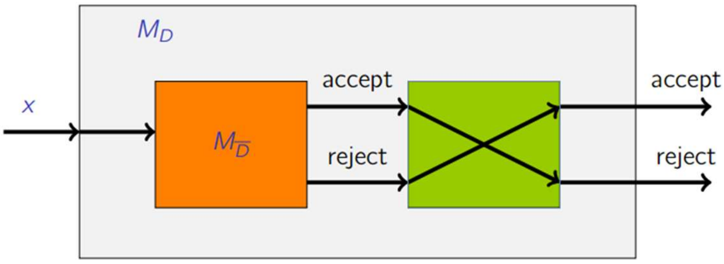
↯ Somit ist D unentscheidbar.

Unentscheidbarkeit des Diagonalsprachenkomplements

$\overline{D} = \{w \in \{0,1\}^* | \, w = w_i \; und \; M_i akzeptiert \; w\}$

Das Komplement $\overline{D}$ ist nicht entscheidbar.

Annahme: Es gibt eine TM $M_{\overline{D}}$ die die Sprache $\overline{D}$ entscheidet. Die TM hält dann auf jeder Eingabe w, und akzeptiert w g.d.w. $w \in D$.
Konstruktion: TM M, die $M_{\overline{D}}$ als Unterprogramm verwendet: M startet $M_{\overline{D}}$ auf der vorliegenden Eingabe und negiert anschließend die Ausgabe von $M_{\overline{D}}$.
Ergebnis: Diese TM M entscheidet dann offensichtlich D. => Widerspruch zur Unentsch. von D.



Die Existenz von $M_{\overline{D}}$ widerspricht der Unentscheidbarkeit von D.
Daher kann es das Programm $M_{\overline{D}}$ nicht geben.
Daher ist $\overline{D}$ unentscheidbar.

Unterprogrammtechnik

Es sei L' eine bereits analysierte, nicht-entscheidbare Sprache.
Es sei L eine neue Sprache, die wir untersuchen wollen.
Um nachzuweisen, dass L nicht entscheidbar ist, genügt es zu zeigen, dass man mit Hilfe von Unterprogrammaufrufen einer TM M_L (zum Entscheiden von L) auch die Sprache L' entscheiden kann.

Beobachtung: Wenn Sprache $L \subseteq \{0,1\}^*$ (un-)entscheidbar => Komplement $\overline{L}$ (un-)entscheidbar

Das Halteproblem

Besteht darin, zu entscheiden, ob ein gegebenes Programm mit einer Eingabe terminiert.

$H = \{\langle M \rangle w \mid M \text{ hält auf } w\}$

Beweis, dass das Halteproblem nicht entscheidbar ist durch Unterprogrammtechnik:

M_H ist eine TM, die H entscheidet:

- M_H hält auf jeder Eingabe
- Akzeptiert nur Eingaben der Form $\langle M \rangle w$, bei denen M auf w hält

Konstruktion neuer TM $M_{\bar{D}}$ mit M_H als Unterprogramm die $\bar{D}$ entscheidet:

1. Auf Eingabe w berechne zuerst Index i mit $w = w_i$
2. Berechne dann die Gödelnummer der i-ten TM M_i, also $\langle M_i \rangle$
3. Starte M_H als Unterprogramm mit Eingabe $\langle M_i \rangle w_i$
   3.1. Falls M_H akzeptiert, so simuliere das Verhalten von M_i auf w_i
   3.2. Falls M_H verwirft, so verwerfe die Eingabe

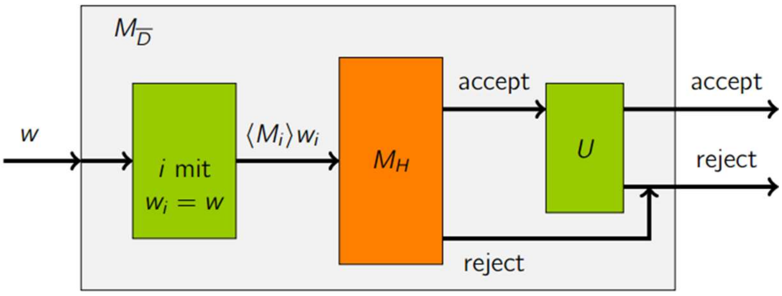
Korrektheit:

- $w \in \bar{D} \Rightarrow M_{\bar{D}}$ akzeptiert w
- $w \notin \bar{D} \Rightarrow M_{\bar{D}}$ verwirft w

Es sei $w = w_i$. Dann gilt:

$w \in \bar{D} \Rightarrow M_i$ akzeptiert w_i
$\Rightarrow M_H$ und U akzeptieren $\langle M_i \rangle w_i$
$\Rightarrow M_{\bar{D}}$ akzeptiert w

$w \notin \bar{D} \Rightarrow M_i$ akzeptiert w_i nicht
$\Rightarrow$ (M_i hält nicht auf w_i) oder (M_i verwirft w_i)
$\Rightarrow$ (M_H verwirft $\langle M_i \rangle w_i$) oder (M_H akzeptiert und U verwirft $\langle M_i \rangle w_i$)
$\Rightarrow M_{\bar{D}}$ verwirft w



Das Epsilon-Halteproblem

$H_\epsilon = \{\langle M \rangle \mid M \text{ hält auf der Eingabe } \epsilon\}$

Z.z.: H_ϵ ist nicht entscheidbar mit Unterprogrammtechnik

Konstruktion neuer TM M_H die das Halteproblem H entscheidet und M_ϵ als Unterprogramm ausführt

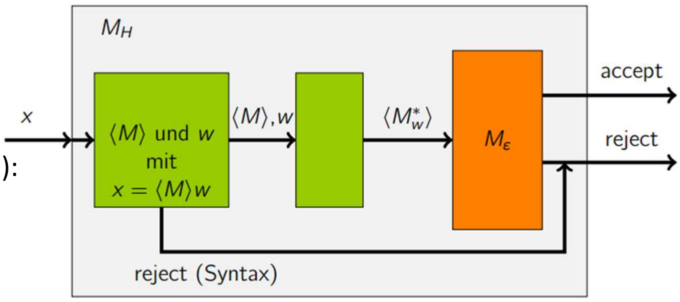
1. Falls die Eingabe nicht mit der korrekten Gödelnummer beginnt, verwirft M_H die Eingabe
2. Sonst, also auf Eingaben der Form $\langle M \rangle w$, berechnet M_H die Gödelnummer einer TM M_w^* mit folgenden Eigenschaften:
   2.1. Falls M_w^* die Eingabe ϵ erhält, so schreibt sie zunächst das Wort w aufs Band und simuliert dann die TM M mit der Eingabe w
   2.2. Bei Eingaben ungleich ϵ kann sich M_w^* beliebig verhalten
3. M_H startet nun M_ϵ mit der Eingabe $\langle M_w^* \rangle$ und akzeptiert (verwirft) genau dann, wenn M_ϵ akzeptiert (verwirft)

Korrektheit:

- $w \in \bar{D} \Rightarrow M_{\bar{D}}$ akzeptiert w
- $w \notin \bar{D} \Rightarrow M_{\bar{D}}$ verwirft w

Annahme Eingabe $x = \langle M \rangle w$ (andernfalls verwerfe):

$\langle M \rangle w \in H \Rightarrow M$ hält auf Eingabe w
$\Rightarrow M_w^*$ hält auf der Eingabe ϵ
$\Rightarrow M_\epsilon$ akzeptiert $\langle M_w^* \rangle$
$\Rightarrow M_H$ akzeptiert $\langle M \rangle w$



$\langle M\rangle w \notin H$ $\Rightarrow M$ hält nicht auf Eingabe w

$\Rightarrow M_w^*$ hält nicht auf der Eingabe ϵ

$\Rightarrow M_\epsilon$ verwirft $\langle M_w^*\rangle$

$\Rightarrow M_H$ verwirft $\langle M\rangle w$

=> H_ε ist unentscheidbar

Partielle Funktionen

- Die von einer TM M berechnete Funktion ist von der Form: $f_M: \{0,1\}^* \to \{0,1\}^* \cup \{\perp\}$.
  Das Zeichen $\perp$ steht dabei für undefiniert und bedeutet, dass die Maschine nicht hält
- Im Fall von Entscheidungsproblemen ist die Funktion von der Form $f_M: \{0,1\}^* \to \{0,1,\perp\}$.
  Dabei steht 0 für Verwerfen, 1 für Akzeptieren und $\perp$ für Nicht-Halten.

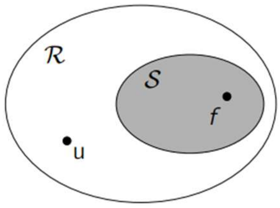
Satz von Rice

Es sei R die Menge der von TMen berechenbaren partiellen Funktionen.
Es sei S eine Teilmenge von R mit $\emptyset \not\subseteq S \not\subseteq R$.
Dann ist die Sprache $L(S) = \{\langle M\rangle | \; M$ berechnet eine Funktion aus $S\}$ nicht entscheidbar.
=> Alle nicht-trivialen Aussagen über die von einer TM berechnete Funktion sind unentscheidbar.

Beispiel 1: Es sei $S = \{f_M | \; f_M(\epsilon) \neq \perp\}$.
Dann ist: $L(S) = \{\langle M\rangle | \; M$ berechnet eine Funktion aus $S\}$ = $\{\langle M\rangle | \; M$ hält auf Eingabe $\epsilon\}$ = H_ϵ
Beispiel 2: Es sei $S = \{f_M | \; \forall w \in \{0,1\}^*: f_M(w) = 1\}$.
=> $L(S) = \{\langle M\rangle | \; M$ berechnet eine Funktion aus $S\}$ = $\{\langle M\rangle | \; M$ hält auf jeder Eingabe mit Ausgabe 1$\}$

Beweis des Satz von Rice durch Unterprogrammtechnik:

- Sei u die überall definierte Funktion $u(w) \equiv \perp$
- O.B.d.A. $u \notin S$
- Sei f eine Funktion aus S
- Sei N eine TM, die f berechnet



Konstruktion neuer TM M_{H_ϵ} mit Unterprogramm $M_{L(S)}$:

1. Falls die Eingabe nicht aus einer korrekten Gödelnummer besteht, so verwirft M_{H_ϵ} die Eingabe
2. Andernfalls berechnet M_{H_ϵ} aus der Eingabe $\langle M\rangle$ die Gödelnummer der TM M^*:
   2.1. Zuerst simuliert M^* das Verhalten von TM M bei Eingabe ε auf einer für diesen Zweck reservierten Spur.
   2.2. Danach simuliert M^* das Verhalten von TM N bei Eingabe x. M^* hält, sobald N hält, und übernimmt die Ausgabe.
3. Schlussendlich starten wir $M_{L(S)}$ mit der Eingabe $\langle M^*\rangle$.
   Wir akzeptieren (verwerfen) genau dann, wenn $M_{L(S)}$ akzeptiert (verwirft)

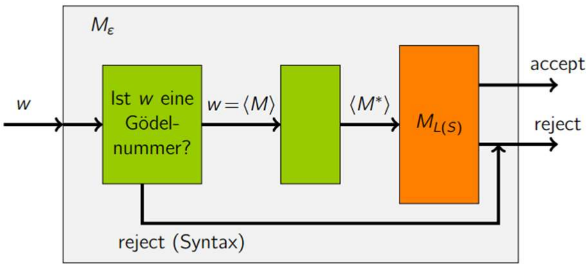
Korrektheit:
Eingabe von w, wobei w keine Gödelnummer ist, verwirft M_{H_ϵ}
Bei Eingabe von $\langle M\rangle w$ gilt:
$w \in H_\epsilon \Rightarrow M$ hält auf ϵ

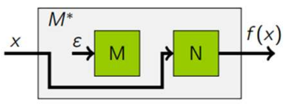
$\Rightarrow M^*$ berechnet f

$\overset{f \in S}{\Longrightarrow} \langle M^*\rangle \in L(S)$

$\Rightarrow M_{L(S)}$ akzeptiert $\langle M^*\rangle$

$\Rightarrow M_{H_\epsilon}$ akzeptiert w





$w \notin H_\epsilon \Rightarrow M$ hält nicht auf ϵ

$\Rightarrow M^*$ berechnet u

$\xRightarrow{u \notin S} \langle M^* \rangle \notin L(S)$

$\Rightarrow M_{L(S)}$ verwirft $\langle M^* \rangle$

$\Rightarrow M_{H_\epsilon}$ verwirft w

Satz von Rice für Java Programme:
Es gibt keine algorithmische Methode (von Hand oder automatisiert; heute oder morgen oder in ferner Zukunft) um festzustellen, ob ein gegebenes Java Programm einer nicht-trivialen Spezifikation entspricht.

Beispiel 3: Es sei $L_{17} = \{\langle M \rangle |\ M$ berechnet bei Eingabe der Zahl 17 die Zahl 42$\}$.
Es ist $L_{17} = L(S)$ für $S = \{f_M |\ f_M\big(bin(17)\big) = bin(42)\}$.
Da $\emptyset \nsubseteq S \nsubseteq R$ gilt, ist die Sprache L_{17} gemäß dem Satz von Rice nicht entscheidbar.
Beispiel 4: Es sei $H_{32} = \{\langle M \rangle |$ auf jeder Eingabe hält M nach höchstens 32 Schritten$\}$.
=> Über diese Sprache sagt der Satz von Rice nichts aus! Ist L_{32} entscheidbar?
Beispiel 5: Es sei $L_D = \{\langle M \rangle |\ M$ entscheidet die Diagonalsprache$\}$.
Dann ist $L_D = L(S)$ für $\mathrm{S} = \{\mathrm{f_D}\}$ wobei $f_D(w) = \begin{cases} 1 & wenn\ w \in D \\ 0 & sonst \end{cases}$
=> Über diese Sprache sagt der Satz von Rice nichts aus!
Aber: Die Sprache ist entscheidbar, denn $L_D = \{\,\}$.

Rekursive Aufzählbarkeit

Semi-Entscheidbarkeit

Eine Sprache L wird von einer TM M entschieden, wenn

- M auf jeder Eingabe hält, und
- M genau die Wörter aus L akzeptiert

Wenn eine TM ex., die die Sprache L entscheidet, so wird L als rekursiv oder entscheidbar bezeichnet
Eine Sprache L wird von einer TM M erkannt, wenn

- M jedes Wort aus L akzeptiert, und
- M kein Wort akzeptiert, das nicht in L enthalten ist

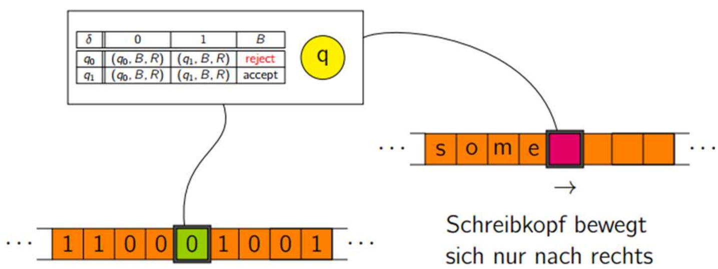
Wenn eine TM ex., die die Sprache L erkennt, so wird L als semi-entscheidbar bezeichnet

Das Halteproblem $H = \{\langle M \rangle w \mid M$ hält auf $w\}$ ist nicht entscheidbar, aber semi-entscheidbar
Beweis: Die folgende TM M_H erkennt die Sprache H
Erhält M_H eine syntaktisch inkorrekte Eingabe, so verwirft M_H die Eingabe.
Erhält M_H eine Eingabe der Form $\langle M \rangle w$, so simuliert M_H die TM M mit Eingabe w und akzeptiert, sobald/falls M auf w hält

Ein Aufzähler für eine Sprache $L \subseteq \Sigma^*$ ist eine Variante der TM mit einem angeschlossenen Drucker

Der Drucker ist ein zusätzliches Ausgabeband, auf dem sich der Kopf nur nach rechts bewegen kann und auf dem nur geschrieben wird

- Aufzähler wird mit leerem Arbeitsband gestartet, und gibt mit der Zeit alle Wörter in L auf dem Drucker aus
- Die ausgegebenen Wörter werden immer durch ein Trennzeichen separiert, das nicht in Σ enthalten ist
- Aufzähler druckt ausschließlich Wörter in L



Rekursive Aufzählbarkeit

Wenn es für eine Sprache L einen Aufzähler gibt, so wird L als rekursiv aufzählbar bezeichnet
Satz: Eine Sprache ist genau dann rekursiv aufzählbar, wenn L semi-entscheidbar ist

Rekursiv aufzählbar $\Leftrightarrow$ semi-entscheidbar (Beweis ausgelassen, da in Klausur als Gegebenheit richtig)

Abschlusseigenschaften

Durchschnitt:

a) Wenn die Sprachen L_1 und L_2 entscheidbar sind, so ist auch die Sprache $L_1 \cap L_2$ entscheidbar
b) Wenn die Sprachen L_1 und L_2 rekursiv aufzählbar sind, so ist auch die Sprache $L_1 \cap L_2$ rekursiv aufzählbar

Beweis: Seien M_1 und M_2 zwei TMen, die L_1 respektive L_2 entscheiden/erkennen
Eine TM M, die $L_1 \cap L_2$ entscheidet/erkennt:

- Eingabe w: M simuliert das Verhalten von M_1 auf w und dann das Verhalten von M_2 auf w
- Falls M_1 und M_2 beide das Wort w akzeptieren, so akzeptiert auch M; andernfalls verwirft M

Korrektheit: Falls $w \in L_1 \cap L_2$, so wird w akzeptiert. Andernfalls wird w verworfen

Vereinigung:
Gilt wie beim Durchschnitt auch, nur wird bei der Beweisführung ∩ mit ∪ vertauscht.

Komplement:
Wenn sowohl die Sprache $L \subseteq \Sigma^*$ als auch ihr Komplement $\overline{L} \coloneqq \Sigma^* \setminus L$ rekursiv aufzählbar sind, so ist L entscheidbar
Beweis:

- Es seien M und $\overline{M}$ zwei TMen, die L respektive $\overline{L}$ erkennen
- Für ein Eingabewort w simuliert die neue TM M' das Verhalten von M auf w und das Verhalten von $\overline{M}$ auf w parallel auf zwei Bändern
- Wenn M akzeptiert, so akzeptiert M'
- Wenn $\overline{M}$ akzeptiert, so verwirft M'
- Da entweder $w \in L$ oder $w \notin L$ gilt, tritt eines der beiden obigen Ereignisse nach endlicher Zeit ein. Damit ist die Terminierung von M' sichergestellt

Satz 1: Wenn die Sprache L entscheidbar ist, so ist auch ihr Komplement $\overline{L}$ entscheidbar
Satz 2: Wenn die Sprache L rekursiv aufzählbar ist, so ist ihr Komplement $\overline{L}$ nicht unbedingt rek. aufz.

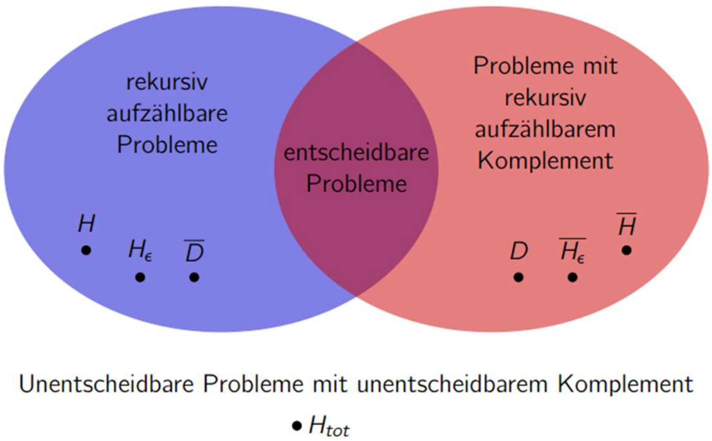
Beispiel: Das Halteproblem H ist rekursiv aufzählbar. Falls $\overline{H}$ ebenfalls rekursiv aufzählbar wäre, so wäre H entscheidbar. Daher ist $\overline{H}$ nicht rekursiv aufzählbar

Berechenbarkeitslandschaft

Jede Sprache L fällt genau in eine der folgenden vier Familien:

(1) L ist entscheidbar, und sowohl L als auch $\overline{L}$ sind rek. aufz.
(2) L ist rek. aufz., aber $\overline{L}$ ist nicht rek. aufz.
(3) $\overline{L}$ ist rek. aufz., aber L ist nicht rek. aufz.
(4) Weder L noch $\overline{L}$ sind rek. aufz.

Beispiele:
Familie 1: Graphenzusammenhang, Hamiltonkreis
Familie 2: H, H_ϵ, $\overline{D}$
Familie 3: $\overline{H}$, $\overline{H}_\epsilon$, D
Familie 4: $H_{tot} = \{\langle M \rangle \mid M \text{ hält auf jeder Eingabe}\}$



Reduktionen

Es seien L_1 und L_2 Sprachen über einem Alphabet Σ. Dann heißt L_1 auf L_2 reduzierbar ($L_1 \leq L_2$), wenn eine berechenbare Funktion $f: \Sigma^* \to \Sigma^*$ existiert, sodass für alle $x \epsilon \Sigma$ gilt: $x \in L_1 \Leftrightarrow f(x) \in L_2$

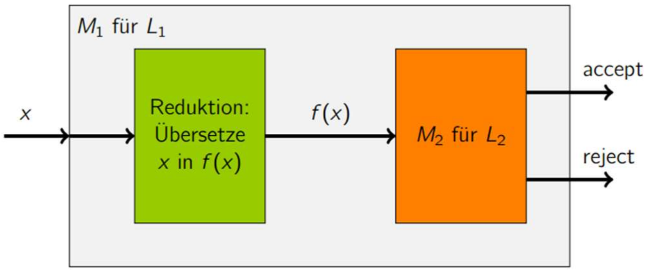
Eine Reduktion ist ein Algorithmus, der die Instanzen eines Startproblems als Spezialfälle eines Zielproblems formuliert

Satz:
Falls $L_1 \leq L_2$ und falls L_2 rekursiv aufzählbar ist, so ist auch L_1 rekursiv aufzählbar
Falls $L_1 \leq L_2$ und falls L_1 nicht rekursiv aufzählbar ist, so ist auch L_2 nicht rekursiv aufzählbar

Beweis:
Konstruktion einer TM M_1, die L_1 erkennt, indem sie als Unterprogramm eine TM M_2 verwendet, die L_2 erkennt:

- Für eine Eingabe x berechnet die TM M_1 zunächst $f(x)$
- Danach simuliert M_1 die TM M_2 mit der Eingabe $f(x)$
- M_1 akzeptiert die Eingabe x, falls M_2 die Eingabe $f(x)$ akzeptiert

Korrektheit:
M_1 akzeptiert $x \Leftrightarrow M_2$ akzeptiert $f(x)$
$\Leftrightarrow f(x) \in L_2$
$\Leftrightarrow x \in L_1$



Das totale Halteproblem

$H_{tot} = \{\langle M \rangle \mid M$ hält auf jeder Eingabe$\}$

Es gilt $\overline{H_\epsilon} \leq \overline{H}_{tot}$ und $\overline{H_\epsilon} \leq H_{tot}$ ➔ Weder $\overline{H}_{tot}$ noch H_{tot} sind rekursiv aufzählbar

Korrektheit von A: $\overline{H}_\epsilon \leq \overline{H}_{tot}$: $w \in \overline{H}_\epsilon \Rightarrow f(w) \in \overline{H}_{tot}$ (a) und $w \notin \overline{H}_\epsilon \Rightarrow f(w) \notin \overline{H}_{tot}$ (b)

Korrektheit von B: $\overline{H}_\epsilon \leq H_{tot}$: $w \in \overline{H}_\epsilon \Rightarrow f(w) \in H_{tot}$ (a) und $w \notin \overline{H}_\epsilon \Rightarrow f(w) \notin H_{tot}$ (b)

Für A: Wir beschreiben eine berechenbare Funktion f, die Ja-Instanzen von $\overline{H}_\epsilon$ auf Ja-Instanzen von $\overline{H}_{tot}$ und Nein-Instanzen von $\overline{H}_\epsilon$ auf Nein-Instanzen von $\overline{H}_{tot}$ abbildet.

Es sei w die Eingabe für $\overline{H}_\epsilon$

- Wenn w keine gültige Gödelnummer ist, so setzen wir $f(x) = w$
- Falls $w = \langle M \rangle$ für eine TM M, so sei $f(w) \coloneqq \langle M_\epsilon^* \rangle$ die Gödelnummer der TM M_ϵ^*:
  - M_ϵ^* ignoriert die Eingabe und simuliert M mit der Eingabe ϵ

Beweis A a): $w \in \overline{H}_\epsilon \Rightarrow f(w) \in \overline{H}_{tot}$ für die Korrektheit von $\overline{H}_\epsilon \leq \overline{H}_{tot}$:
Falls w keine Gödelnummer ist, gilt $w \in \overline{H}_\epsilon$ und $f(x) \in \overline{H}_{tot}$
Falls $w = \langle M \rangle$ für eine TM M, so betrachten wir $f(w) \coloneqq \langle M_\epsilon^* \rangle$. Dann gilt:
$w \in \overline{H}_\epsilon \Rightarrow M$ hält nicht auf der Eingabe ϵ
$\Rightarrow M_\epsilon^*$ hält auf gar keiner Eingabe
$\Rightarrow \langle M_\epsilon^* \rangle \notin H_{tot}$
$\Rightarrow f(w) = \langle M_\epsilon^* \rangle \in \overline{H}_{tot}$

Beweis A b): $w \notin \overline{H}_\epsilon \Rightarrow f(w) \notin \overline{H}_{tot}$ für die Korrektheit von $\overline{H}_\epsilon \leq \overline{H}_{tot}$:
Falls $w = \langle M \rangle$ für eine TM M, so betrachten wir $f(w) \coloneqq \langle M_\epsilon^* \rangle$. Dann gilt:

$w \notin \overline{H}_\epsilon \Rightarrow w \in H_\epsilon$
$\Rightarrow M$ hält auf der Eingabe ϵ
$\Rightarrow M_\epsilon^*$ hält auf jeder Eingabe
$\Rightarrow \langle M_\epsilon^* \rangle \in H_{tot}$
$\Rightarrow f(w) = \langle M_\epsilon^* \rangle \notin \overline{H}_{tot}$

Für B: Wir beschreiben eine berechenbare Funktion f, die Ja-Instanzen von $\overline{H}_\epsilon$ auf Ja-Instanzen von H_{tot} und Nein-Instanzen von $\overline{H}_\epsilon$ auf Nein-Instanzen von H_{tot} abbildet.

Es sei w die Eingabe für $\overline{H}_\epsilon$. Es sei w' irgendein Wort aus H_{tot}.

- Wenn w keine gültige Gödelnummer ist, so setzen wir $f(x) = w'$
- Falls $w = \langle M \rangle$ für eine TM M, so sei $f(w) \coloneqq \langle M' \rangle$ die Gödelnummer der TM M', die sich auf Eingaben der Länge l wie folgt verhält:
  - M' simuliert die ersten l Schritte von M auf der Eingabe ϵ. Wenn M innerhalb dieser l Schritte hält, dann geht M' in eine Endlosschleife; andernfalls hält M'.

Beweis B a): $w \in \overline{H}_\epsilon \Rightarrow f(w) \in H_{tot}$ für die Korrektheit von $\overline{H}_\epsilon \leq H_{tot}$:
Falls w keine Gödelnummer ist, gilt $w \in \overline{H}_\epsilon$ und $f(x) \in H_{tot}$
Falls $w = \langle M \rangle$ für eine TM M, so betrachten wir $f(w) \coloneqq \langle M' \rangle$. Dann gilt:
$w \in \overline{H}_\epsilon \Rightarrow M$ hält nicht auf der Eingabe ϵ
$\Rightarrow \neg\exists i$: M hält innerhalb von i Schritten auf ϵ
$\Rightarrow \neg\forall i$: M hält nicht innerhalb von i Schritten auf ϵ
$\Rightarrow \neg\forall i$: M' hält auf allen Eingabe der Länge i
$\Rightarrow M'$ hält auf jeder Eingabe
$\Rightarrow f(w) = \langle M' \rangle \in H_{tot}$

Beweis B b): $w \notin \overline{H}_\epsilon \Rightarrow f(w) \notin H_{tot}$ für die Korrektheit von $\overline{H}_\epsilon \leq H_{tot}$:
Falls $w = \langle M \rangle$ für eine TM M, so betrachten wir $f(w) \coloneqq \langle M' \rangle$. Dann gilt:
$w \notin \overline{H}_\epsilon \Rightarrow M$ hält auf der Eingabe ϵ
$\Rightarrow \exists i$: M hält innerhalb von i Schritten auf ϵ
$\Rightarrow \exists i$: M hält auf keiner Eingabe mit Länge $\geq i$
$\Rightarrow M'$ hält nicht auf jeder Eingabe
$\Rightarrow f(w) = \langle M' \rangle \notin H_{tot}$

Das Postsche Correspondenzproblem (PCP)

Puzzle aus Dominosteinen mit je einem Wort unten und einem Wort oben über einem Alphabet Σ. Gegeben ist eine Menge K von Dominos, wobei jeder Domino beliebig oft verwendet werden darf. Die Aufgabe besteht darin eine korrespondierende Folge von Dominos aus K zu finden, mit der sich oben und unten jeweils dasselbe Wort befindet. Die Folge muss mind. einen Domino enthalten.

Beispiel: Für die Dominomenge $K = \{\left[\frac{b}{ca}\right], \left[\frac{a}{ab}\right], \left[\frac{ca}{a}\right], \left[\frac{dbd}{cef}\right], \left[\frac{abc}{c}\right], \left[\frac{caeef}{abce}\right]\}$ gibt es die korrespondierende Folge $\langle 2,1,3,2,5 \rangle$ mit $\left[\frac{a}{ab}\right]\left[\frac{b}{ca}\right]\left[\frac{ca}{a}\right]\left[\frac{a}{ab}\right]\left[\frac{abc}{c}\right]$

Eine Instanz des PCP besteht aus einer endlichen Menge $K = \{\left[\frac{x_1}{y_1}\right], \dots, \left[\frac{x_k}{y_k}\right]\}$ wobei $x_1, \dots, x_k$ und $y_1, \dots, y_k$ nichtleere Wörter über einem endlichen Alphabet Σ sind.

Das Problem besteht darin, zu entscheiden, ob es eine (nicht-leere) korrespondierende Folge $\langle i_1, \dots, i_n \rangle$ von Indizes in $\{1, \dots, k\}$ gibt, sodass $x_{i_1} x_{i_2} \dots x_{i_n} = y_{i_1} y_{i_2} \dots y_{i_n}$.
Die Elemente von K nennen wir Dominosteine.

Das modifizierte PCP

Selbe Definition wie PCP, aber das Problem besteht darin, zu entscheiden, ob es eine korrespondierende Folge $\langle i_1, \dots, i_n \rangle$ von Indizes in $i_1 = 1$ gibt, sodass $x_{i_1} x_{i_2} \dots x_{i_n} = y_{i_1} y_{i_2} \dots y_{i_n}$.
D.h. die Modifikation besteht darin, dass es einen vorgegebenen Startdomino gibt.

$MPCP \leq PCP$

- Wir betrachten MPCP Instanz $K = \{\left[\frac{x_1}{y_1}\right], \dots, \left[\frac{x_k}{y_k}\right]\}$
- Es seien # und \$ zwei Symbole, die nicht im MPCP vorkommen
- Wir konstruieren x_i' aus x_i, indem wir hinter jedem Buchstaben ein # einfügen
- Wir konstruieren y_i' aus y_i, indem wir vor jedem Buchstaben ein # einfügen
- Ferner setzen wir $x_0' = \#x_1'$; $y_0' = y_1'$; $x_{k+1}' = \$$; und $y_{k+1}' = \#\$$
- Damit berechnen wir die folgende PCP Instanz: $f(K) = \{\left[\frac{x_0'}{y_0'}\right], \left[\frac{x_1'}{y_1'}\right], \dots, \left[\frac{x_k'}{y_k'}\right], \left[\frac{x_{k+1}'}{y_{k+1}'}\right]\}$

Beispiel: MPCP $K = \{\left[\frac{ab}{a}\right], \left[\frac{c}{abc}\right], \left[\frac{a}{b}\right]\}$ modelliert als PCP $f(K) = \{\left[\frac{\#a\#b\#}{\#a}\right], \left[\frac{a\#b\#}{\#a}\right], \left[\frac{c\#}{\#a\#b\#c}\right], \left[\frac{a\#}{\#b}\right], \left[\frac{\$}{\#\$}\right]\}$

$K \in MPCP \Rightarrow f(K) \in PCP$

- Sei $(i_1, i_2, \dots i_n)$ Lösung für MPCP K. Dann gilt $i_1 = 1$ und $x_{i_1} x_{i_2} \dots x_{i_n} = y_{i_1} y_{i_2} \dots y_{i_n} = a_1 a_2 \dots a_s$
- Dann ist $(0, i_2, \dots i_n, k+1)$ eine Lösung für PCP $f(K)$, denn $x_0' x_{i_2}' \dots x_{i_n}' \$ = \#a_1\#a_2\# \dots \#a_s\$ = y_0' y_{i_2}' \dots y_{i_n}'$
- Aus einer Lösung für MPCP K lässt sich also eine Lösung für PCP $f(K)$ konstruieren. Damit ist die Implikation gezeigt

$f(K) \in PCP \Rightarrow K \in MPCP$

- Es sei $(i_1, i_2, \dots i_n)$ eine Lösung minimaler Länge für $f(K)$
- Fakt A: $i_1 = 0$, da nur x_0' und y_0' mit demselben Zeichen beginnen
- Fakt B: $i_n = k+1$, da nur x_{k+1}' und y_{k+1}' mit demselben Zeichen enden
- Fakt C: $i_j \neq k+1$ für $1 \leq j \leq n$. Andernfalls kürzere Lösung
- Fakt D: $i_j \neq 0$ für $2 \leq j \leq n$. Andernfalls folgen im oberen Wort zwei #-Zeichen direkt aufeinander, was im unteren Wort unmöglich ist.
- Durch das Löschen aller # und \$ Symbole wird das PCP Lösungswort also zum MPCP Lösungswort

$H \leq MPCP$

Illustrierender Beweis ausgelassen

- Wir beschreiben eine berechenbare Funktion f, die eine syntaktisch korrekte Instanz $\langle M \rangle w$ fürs Halteproblem H in eine syntaktisch korrekte Instanz $K := f(\langle M \rangle w)$ fürs MPCP übersetzt
- Dabei gilt: M hält auf $w \Leftrightarrow K$ hat korrespondierende Folge
- Syntaktisch nicht korrekte Eingaben für H werden auf syntaktisch nicht korrekte Eingaben fürs MPCP abgebildet
- Für die MPCP Instanz verwenden wir das Alphabet $\Gamma \cup Q \cup \{\#\}$ mit $\# \notin \Gamma \cup Q$

Reduktion

- Startdomino:
  - $\left[\frac{\#}{\#\#q_0w\#}\right]$
- Kopierdominos:
  - $\left[\frac{a}{a}\right]$ für alle $a \in \Gamma \cup \{\#\}$

- Überführungsdominos:
    - $\left[\frac{qa}{q'c}\right]$ falls $\delta(q,a) = (q',c,N)$, für $q \in Q \setminus \{\bar{q}\}, a \in \Gamma$
    - $\left[\frac{qa}{cq'}\right]$ falls $\delta(q,a) = (q',c,R)$, für $q \in Q \setminus \{\bar{q}\}, a \in \Gamma$
    - $\left[\frac{bqa}{q'bc}\right]$ falls $\delta(q,a) = (q',c,L)$, für $q \in Q \setminus \{\bar{q}\}, a,b \in \Gamma$
- Überführungsdominos für implizite Blanks:
    - $\left[\frac{\#qa}{\#q'Bc}\right]$ falls $\delta(q,a) = (q',c,L)$, für $q \in Q \setminus \{\bar{q}\}, a \in \Gamma$
    - $\left[\frac{q\#}{q'c\#}\right]$ falls $\delta(q,B) = (q',c,N)$, für $q \in Q \setminus \{\bar{q}\}$
    - $\left[\frac{q\#}{cq'\#}\right]$ falls $\delta(q,B) = (q',c,R)$, für $q \in Q \setminus \{\bar{q}\}$
    - $\left[\frac{bq\#}{q'bc\#}\right]$ falls $\delta(q,B) = (q',c,L)$, für $q \in Q \setminus \{\bar{q}\}, b \in \Gamma$
    - $\left[\frac{\#q\#}{\#q'Bc\#}\right]$ falls $\delta(q,B) = (q',c,L)$, für $q \in Q \setminus \{\bar{q}\}$
- Lösungsdominos:
    - $\left[\frac{a\bar{q}}{\bar{q}}\right]$ und $\left[\frac{\bar{q}a}{\bar{q}}\right]$ für $a \in \Gamma$
- Abschlussdomino:
    - $\left[\frac{\#\bar{q}\#\#}{\#}\right]$

Korrektheitsargument

1. f ist berechenbar (ist bereits erledigt)
2. M hält auf $w \Rightarrow K \in MPCP$
3. $K \in MPCP \Rightarrow M$ hält auf w

Beweis von 2.: M hält auf $w \Rightarrow K \in MPCP$

- Berechnung von M auf w entspricht endlicher Konfigurationsfolge $k_0 \vdash k_1 \vdash \cdots \vdash k_{t-1} \vdash k_t$, wobei k_0 die Startkonfiguration im Zustand q_0 und k_t die Endkonfiguration im Zustand q_0 ist
- Wir konstruieren eine korrespondierende Folge, die mit dem Startdomino beginnt
  Der obere String ist ein Präfix des unteren Strings:
  $\#\#k_0\#\#k_1\#\# \dots \#\#k_{t-1}\#$
  Der untere String gibt die vollständige Konfigurationsfolge an:
  $\#\#k_0\#\#k_1\#\# \dots \#\#k_{t-1}\#\#k_t\#$
- Durch Hinzufügen von einer Folge von Löschdominos kann das Nachhinken des oberen Strings fast ausgeglichen werden. Danach sind beide Strings identisch bis auf einen Sufix der Form $\#\bar{q}\#$, der im oberen String fehlt
- Hinzufügen des Abschlussdominos macht beide Strings identisch

Beweis von 3.: $K \in MPCP \Rightarrow M$ hält auf w

- Der Satz von Dominosteinen im MPCP hat folgende Eigenschaften:
    - Beim Startdomino ist der obere String kürzer als der untere
    - Bei Kopier- und Übergangsdominos ist der obere String immer höchstens so lang wie der untere String
    - Nur auf Abschluss- und Löschdominos ist der obere String länger als der untere String
- Die korresp. Folge für K liefert uns eine entsprechende Konfigurationsfolge von M auf w
    - Diese Konfigurationsfolge beginnt mit dem Startdomino
    - Diese Konfigurationsfolge muss zumindest einen Lösch- oder Abschlussdomino enthalten (andernfalls wäre der untere String länger als der obere String)
    - Deshalb erscheint der Zustand $\bar{q}$ in dieser Konfigurationsfolge, und die Rechnung von M auf w terminiert

Die Transivität der Reduzierbarkeit impliziert $H \leq PCP$. Das PCP ist unentscheidbar.

Varianten des PCP

- Wenn alle Wörter auf den Dominos Länge 1 haben, so ist das PCP entscheidbar
- Wenn alle Wörter auf den Dominos Länge 1 oder 2 haben, so ist das PCP unentscheidbar
- Für 1 Domino trivial; Für 2 Domino entscheidbar; Für 3,4 ungeklärt; Für 5, 7 unentscheidbar

Turing-Mächtigkeit

Entscheidungsprobleme für CFGs

Eine „context free grammar" G ist ein Quadrupel (N, Σ, P, S), wobei

- $N =$ Menge der non-Terminal Symbole
- $\Sigma =$ Terminal Alphabet
- $P =$ Menge von Regeln der Form $A \to w$ wobei $A \in N$ und $(\Sigma \cup N)^*$
- $S =$ Element von N (Startsymbol)

Beispiel: $N = \{S\}, \Sigma = \{a, b, c\}, P = S \to aSa|bSb|cSc; S \to a|b|c; S \to aa|bb|cc$
Herleitung: $S \to aSa \to acSca \to acaSaca \to acaaSaaca \to acaabbaaca$

Definition: $L(G)$ ist die Menge aller Wörter über dem Terminal Alphabet Σ, die durch wiederholte Anwendung von Regeln in P aus dem Startsymbol S hergeleitet werden können.

Entscheidbare Probleme:

- Gegeben CFG $\langle G \rangle$ und $w \in \Sigma^*$, gilt $w \in L(G)$?
- Gegeben CFG $\langle G \rangle$, ist $L(G)$ leer?
- Gegeben CFG $\langle G \rangle$, ist $L(G)$ endlich?

Unentscheidbare Probleme:

- Gegeben CFG $\langle G \rangle$, ist G eindeutig?
- Gegeben CFG $\langle G \rangle$, gilt $L(G) \in \Sigma^*$?
- Gegeben CFG $\langle G \rangle$, ist $L(G)$ regulär?
- Gegeben CFG $\langle G_1 \rangle$ und $\langle G_2 \rangle$, gilt $L(G_1) \subseteq L(G_2)$?
- Gegeben CFG $\langle G_1 \rangle$ und $\langle G_2 \rangle$, gilt $L(G_1) \cap L(G_2)$ leer?

Beweis, das „Gegeben CFG $\langle G_1 \rangle$ und $\langle G_2 \rangle$, gilt $L(G_1) \cap L(G_2)$ leer?" unentscheidbar ist:

- Betrachte beliebige PCP Instanz $\{\left[\frac{x_1}{y_1}\right], \dots, \left[\frac{x_k}{y_k}\right]\}$
- Es seien a, b, c drei Buchstaben, die nicht in x_i und y_i vorkommen
- Konstruiere CFGs G_1 und G_2 mit folgenden Regeln:
  - $G_1: S \to x_1Sa^1b|x_2sa^2b|x_3Sa^3b| \dots |x_kSa^kb|c$
  - $G_2: S \to y_1Sa^1b|y_2sa^2b|y_3Sa^3b| \dots |y_kSa^kb|c$
- PCP lösbar g.d.w. $L(G_1) \cap L(G_2)$ nicht leer
  - $G_1: S \overset{*}{\to} x_1x_4x_2x_5x_1x_4\ c\ a^4\ b\ a^1\ b\ a^5\ b\ a^2\ b\ a^4\ b\ a^1b$
  - $G_1: S \overset{*}{\to} y_1y_4y_2y_5y_1y_4\ c\ a^4\ b\ a^1\ b\ a^5\ b\ a^2\ b\ a^4\ b\ a^1b$

Integration in geschlossener Form

Satz: Wenn eine Funktion $f(x)$ der Quotient von zwei Polynomen $P(x)$ und $Q(x)$ ist, so kann $f(x)$ als Summe von Termen der Form $\frac{a}{(x-b)^n}$ und $\frac{ax+}{((x-c)^2+d^2)^n}$ geschrieben werden.

Da jeder derartige Term geschlossen integrierbar ist, ist jede rationale Funktion $f(x)$ geschlossen integrierbar.

Eine Funktion heißt elementar, wenn sie durch Kombination von Addition, Subtraktion, Multiplikation, Division, Potenzieren, Wurzel ziehen, Logarithmieren, Betragsfunktion und den trigonometrischen Funktionen konstruiert werden kann.

Satz von Richardson: Es ist unentscheidbar, ob eine gegebene elementare Funktion eine elementare Stammfunktion besitzt.

Hilberts zehntes Problem

Konstruiere eine Turing Maschine, die die folgende Sprache entscheidet:
$Dioph = \{\langle P \rangle | p$ ist ein Polynom mit ganzzahligen Koeffizienten und mit (mindestens) einer ganzzahligen Nullstelle$\}$

Diophantische Gleichungen:
Setzt ein Polynom = 0. Die Lösungen der Gleichung entsprechen also den Nullstellen des Polynoms.
Beispiel: $6x^3yz^2 + 3xy^2 - x^3 = 10$ mit der Nullstelle $(x, y, z) = (5,3,0)$
z.B. offenes Problem: Besitzt Gleichung $x^3 + y^3 + z^3 = 33$ eine ganzzahlige Nullstelle?

Rekursive Aufzählbarkeit von $Dioph$

Für ein Polynom $p \in \mathbb{Z}[x_1, \dots x_l]$ in l Variablen entspricht der Wertebereich der abzählbar unendlichen Menge $\mathbb{Z}^l$.

Algorithmus der Dioph erkennt:

- Zähle die l-Tupel aus $\mathbb{Z}^l$ in kanonischer Reihenfolge auf und werte p für jedes Tupel aus
- Akzeptiere, sobald eine dieser Auswertungen den Wert 0 ergibt
  => Die Sprache $Dioph$ ist rekursiv aufzählbar.

Satz von Matiyasevich: Das Problem, ob ein ganzzahliges Polynom eine ganzzahlige Nullstelle besitzt, ist unentscheidbar.

Satz: Für jede Teilmenge $X \in \mathbb{Z}$ der ganzen Zahlen sind die beiden folgenden Aussagen äquivalent:

- X ist rekursiv aufzählbar
- Es existiert ein $(k+1)$-variates Polynom $p(x, y_1, \dots y_k)$ mit ganzzahligen Koeffizienten, sodass $X = \{x \in \mathbb{Z} | \exists y_1, \dots y_k \in \mathbb{Z}$ mit $p(x, y_1, \dots y_k) = 0\}$

Ganzzahlige Polynome sind also genauso „berechnungsstark" und „mächtig" wie Turing Maschinen.

Turing-Mächtigkeit

Ein Rechnermodell wird als Turing-mächtig bezeichnet, wenn jede Funktion, die durch eine TM berechnet werden kann, auch durch dieses Rechnermodell berechnet werden kann.

Beispiele:

- Nicht Turing-mächtig:
  - Reines HTML
  - Tabellenkalkulationen
- Turing-mächtig:
  - Lambda Calculus von Alonzo Church
  - μ-rekursive Funktionen von Stephen Kleene
  - Alle gängigen höheren Programmiersprachen (C++, Python, Java, C, …)
  - PostScript, Text, Latex
  - PowerPoint (wegen Animated Features)
  - Conway's Game of Life

LOOP und WHILE Programme

Die Programmiersprache LOOP

Syntax:

- Variablen $x_1\ x_2\ x_3$ …
- Konstanten: 0 und 1
- Symbole: $\coloneqq\ +\ ;$
- Schlüsselwörter: LOOP DO ENDLOOP

Zuweisung:
Für alle Variablen x_i und y_i und für jede Konstante $c\ \{0,1\}$ ist die Zuweisung $x_i \coloneqq x_j + c$ ein LOOP-Programm

Hintereinanderausführung:
Falls P_1 und P_2 LOOP Programme sind, so ist auch $P_1; P_2$ ein LOOP-Programm

LOOP-Konstrukt:
Falls P ein LOOP-Programm ist, so ist auch LOOP x_j DO P ENDLOOP ein LOOP-Programm
(P wird x_j mal ausgeführt)

Die Eingabe ist in den Variablen $x_1, \dots x_m$ enthalten, alle anderen Variablen werden mit 0 initialisiert.
Resultat eines LOOP-Programms: Zahl, die sich am Ende der Abarbeitung in der Variablen x_0 ergibt

Semantik: trivial, im Folgenden ein paar Formalitäten:

- Ein Programm P mit k Variablen berechnet eine k-stellige Funktion der Form $[P]: \mathbb{N}^k \to \mathbb{N}^k$
- Ist P die Zuweisung $x_j \coloneqq x_j + c$, so ist $[P](r_1, \dots, r_k) = (r_1, \dots, r_{i-1}, r_j + c, r_{i+1}, \dots, r_k)$
- Ist $P = P_1; P_2$ eine Hintereinanderausführung, so ist $[P](r_1, \dots, r_k) = [P_2]([P_1](r_1, \dots, r_k))$
- Ist $P =$ LOOP x_j DO Q ENDLOOP ein LOOP-Konstrukt, so gilt $[P](r_1, \dots, r_k) = [Q]^{r_i}(r_1, \dots, r_k)$

Beispiel: Addition $x_0 \coloneqq x_1 + x_2$:
$x_0 \coloneqq x_1;$ LOOP x_2 DO $x_0 \coloneqq x_0 + 1$ ENDLOOP

Beispiel: Multiplikation $x_0 \coloneqq x_1 * x_2$:
$x_0 \coloneqq 0;$ LOOP x_2 DO $x_0 \coloneqq x_0 + x_1$ ENDLOOP

Beispiel: IF $x_1 = 0$ THEN P_1 ELSE P_2 ENDIF:
$x_2 \coloneqq 1;\ x_3 \coloneqq 0;$
LOOP x_1 DO $x_2 \coloneqq 0;\ x_3 = 1$ ENDLOOP
LOOP x_2 DO P_1 ENDLOOP
LOOP x_3 DO P_2 ENDLOOP

$x_i \coloneqq x_j$ DIV x_k

```
y := 0;
x_i := 0;
d := x_j + 1;
d := d ∸ y;
z := x_j + 1;
LOOP z DO
    IF d = 0 THEN ELSE
        y := y + x_k;
        x_i := x_i + 1;
        d := x_j + 1;
        d := d ∸ y
    ENDIF
ENDLOOP;
x_i := x_i ∸ 1
```

$x_i \coloneqq x_j$ MOD x_k

```
y := x_j DIV x_k;
y := y · x_k;
x_i := x_j ∸ y
```

Die Programmiersprache WHILE

Syntax:

- Variablen $x_1\ x_2\ x_3$ …
- Konstanten: 0 und 1
- Symbole: $\coloneqq\ +\ ;\ \neq$
- Schlüsselwörter: WHILE DO ENDWHILE

Zuweisungen und Hintereinanderausführungen wie bei LOOP

WHILE-Konstrukt:
Falls P ein WHILE-Programm ist und x_i eine Variable, so ist auch WHILE $x_i \neq 0$ DO P ENDWHILE ein WHILE-Programm (P wird solange ausgeführt, bis x_i den Wert 0 erreicht hat)

Die Eingabe ist in den Variablen $x_1, \dots x_m$ enthalten, alle anderen Variablen werden mit 0 initialisiert.
Resultat eines WHILE-Programms: Zahl, die sich am Ende der Abarbeitung in der Variablen x_0 ergibt.

Semantik: trivial, im Folgenden ein paar Formalitäten:

- Ein Programm P mit k Variablen berechnet eine k-stellige Funktion der Form $[P]: \mathbb{N}^k \to \mathbb{N}^k$
- Ist $P =$ WHILE $x_i \neq 0$ DO P ENDWHILE ein WHILE-Konstrukt,
  so ist $[P](r_1, \dots, r_k) = [Q]^l(r_1, \dots, r_k)$ für die kleinste Zahl l, für die i-te Komponente von $[Q]^l(r_1, \dots, r_k)$ gleich 0. Falls solch ein l nicht existiert, so ist $[P](r_1, \dots, r_k)$ undefiniert

WHILE vs. LOOP

Die LOOP-Schleife "LOOP x_i DO P ENDLOOP" kann durch die folgende WHILE-Schleife simuliert werden: „$y \coloneqq x_i$; WHILE $y \neq 0$ DO $y \coloneqq y - 1; P$ ENDWHILE"
=> Jede LOOP-berechenbare Funktion $f: \mathbb{N}^k \to \mathbb{N}^k$ ist auch WHILE-berechenbar

Satz: Jedes LOOP-Programm hält auf jeder möglichen Eingabe nach endlich vielen Schritten an.
Es gibt WHILE-Programme, die nicht terminieren:
$x_i \coloneqq 1;$
WHILE $x_1 \neq 0$ DO $x_1 \coloneqq x_1 + 1$ ENDWHILE

Mächtigkeit von WHILE

Satz: Die Programmiersprache WHILE ist Turing-mächtig

Wir betrachten eine TM M:

- Zustandsmenge $Q = \{q_0, \dots, q_t\}$; Anfangszustand: q_1; Endzustand: q_0
- TM im Zustand $q_i \Leftrightarrow$ WHILE Variable Zustand $= i$
- Bandalphabet $\Gamma = \{1,2,B\}$
- WHILE kodiert Bandalphabet zu Dezimalziffern: $1 = 1; 2 = 2; B = 0$

Konfiguration einer TM: $1122q_3 21211$
Vier entsprechende Variablen im WHILE-Programm

Band-vor-Kopf		1122		Variable BandLinks
Kopf	Zustand	2	3	Variable UntermKopf; Zustand
pfoK-ba-danB		11121		Variable BandRechts

Simulation:

(A) Update von Zustand:
Zustand $\coloneqq i$;

(B) Update von Symbol unterm Kopf:
UntermKopf $\coloneqq \sigma$;

(C) Bewegung des Kopfes L, R, N:

- Nach links (L):
  BandRechts $\coloneqq 10 *$ BandRechts $+$ UntermKopf;
  UntermKopf $\coloneqq$ BandLinks MOD 10;
  BandLinks $\coloneqq$ BandLinks DIV 10;
- Nach rechts (R):
  BandLinks $\coloneqq 10 *$ BandLinks $+$ UntermKopf;
  UntermKopf $\coloneqq$ BandRechts MOD 10;
  BandRechts $\coloneqq$ BandRechts DIV 10;
- Keine Bewegung (N):
  Alle Variablen unverändert lassen

Initialisierung:

- Zustand $:= 1$;
- BandLinks $:= 0$;
- UntermKopf $:=$ Erstes Symbol im Eingabewort (als Dezimalziffer);
- BandRechts $:=$ Restliches gespiegeltes Eingabewort (dezimal);

WHILE Zustand $\neq 0$ DO

IF Zustand $= 1$ AND UntermKopf $= 0$ THEN *Schritt* ENDIF;
IF Zustand $= 1$ AND UntermKopf $= 1$ THEN *Schritt* ENDIF;
IF Zustand $= 1$ AND UntermKopf $= 2$ THEN *Schritt* ENDIF;

IF Zustand $= 2$ AND UntermKopf $= 0$ THEN *Schritt* ENDIF;
...

IF Zustand $= t$ AND UntermKopf $= 0$ THEN *Schritt* ENDIF;
IF Zustand $= t$ AND UntermKopf $= 1$ THEN *Schritt* ENDIF;
IF Zustand $= t$ AND UntermKopf $= 2$ THEN *Schritt* ENDIF;

ENDWHILE

Ackermann Funktion

Die Ackermann Funktion $A: \mathbb{N}^2 \to \mathbb{N}$ ist folgendermaßen definiert:

$$
\begin{array}{lll}
A(0,n) & = n+1 & \text{für } n \geq 0 \\
A(m+1,0) & = A(m,1) & \text{für } m \geq 0 \\
A(m+1,n+1) & = A(m,A(m+1,n)) & \text{für } m,n \geq 0
\end{array}
$$

Beispiele:

$$A(1,n) = n+2$$
$$A(2,n) = 2n+3$$
$$A(3,n) = 2^{n+3} - 3$$
$$A(4,n) = \underbrace{2^{2^{\cdots^{2}}}}_{n+3 \text{ viele } 2\text{en}} - 3$$

$$A(1,0) = A(0,1) = 2$$
$$A(1,1) = A\big(0, A(1,0)\big) = A(1,0) + 1 = 3$$
$$A(1,2) = A\big(0, A(1,1)\big) = A(1,1) + 1 = 4$$
$$A(1,3) = A\big(0, A(1,2)\big) = A(1,2) + 1 = 5$$

Bereits der Wert $A(4,2) = 2^{65536} - 3$ ist größer als die Anzahl aller Atome im Weltraum.

UpArrow Notation

- Addition ist iterierte Nachfolgerbildung: $\underbrace{S(S(\dots(S(a)\dots)}_{b\ mal} = a+b$
- Multiplikation ist iterierte Addition: $\underbrace{a + \cdots + a}_{b\ mal} = a * b$
- Potenzierung ist iterierte Multiplikation: $\underbrace{a * \dots * a}_{b\ mal} = a^b =: a \uparrow b$
- Der Potenzturm ist iterierte Potenzierung: $\underbrace{a^{a^{\cdots^{a}}}}_{b\ mal} =: a \uparrow\uparrow b$
- Wiederholte Potenzturmbildung gibt: $a \uparrow\uparrow a \uparrow\uparrow \ \dots\ \uparrow\uparrow a =: a \uparrow\uparrow\uparrow b$

Definition:

$$
a \uparrow^m b := \begin{cases} 1 & \text{wenn } b = 0 \\ a \cdot b & \text{wenn } m = 0 \\ a^b & \text{wenn } m = 1 \\ a \uparrow^{m-1} (a \uparrow^m (b-1)) & \text{sonst} \end{cases}
$$

Es gilt:

$$A(1,n) = 2 + (n+3) - 3$$
$$A(2,n) = 2 * (n+3) - 3$$
$$A(3,n) = 2 \uparrow (n+3) - 3$$
$$A(4,n) = 2 \uparrow\uparrow (n+3) - 3$$
$$A(5,n) = 2 \uparrow\uparrow\uparrow (n+3) - 3$$
...
$$A(m,n) = 2 \uparrow^{m-2} (n+3) - 3 \text{ für } m \geq 2$$

Berechenbarkeit der Ackermannfunktion

Die Ackermann Funktion ist Turing-berechenbar.

Beweis:

- Wir zeigen mit Induktion über $m \geq 0$, dass jede Funktion $f_m: \mathbb{N} \to \mathbb{N}$ mit $f_m(x) = A(m, x)$ berechenbar ist.
- Für $m = 0$ ist f_0 die Nachfolgerfunktion $f_0(x) = x + 1$
- Für $m \geq 1$ berechnen wir $f_m(x)$, indem wir der Reihe nach induktiv alle Werte $f_m(0), f_m(1), \dots f_m(x-1)$ bestimmen
- Zum Schluss berechnen wir: $f_m(x) = A(m, x) = A\big(m-1, A(m, x-1)\big) = f_{m-1}(f_m(x-1))$

Monotonie-Verhalten der Ackermannfunktion

1. $A(m, n+1) > A(m, n)$
2. $A(m+1, n) > A(m, n)$
3. $A(m+1, n-1) \geq A(m, n)$

Folgerung aus 1. und 2.: Für $m \geq m'$ und $n \geq n'$ gilt immer: $A(m, n) \geq A(m', n')$

Beweis von 1.: $A(m, n+1) > A(m, n)$

$$
\begin{aligned}
A(m, n+1) \quad &= A\big(m-1, A(m, n)\big) \\
A(m, n) \quad &= A\big(m-1, A(m, n-1)\big)
\end{aligned}
$$

Induktion liefert zuerst $x := A(m, n) > A(m, n-1) =: y$ und danach $A(m-1, x) > A(m-1, y)$

Beweis von 2.: $A(m+1, n) > A(m, n)$ (nur als Übung durchgenommen)

Beweis von 3.: $A(m+1, n-1) \geq A(m, n)$

Induktionsanfang: Es gilt $A(1, n-1) = n + 1 = A(0, n)$ für alle $n \geq 1$.
Es gilt $A(m+1, 0) = A(m, 1)$ für alle $m \geq 0$.

Induktionsschritt: Wir nehmen an, dass für (m', n') mit $m' < m)$ oder $(m' = m) \wedge (n' < n)$ immer $A(m'+1, n'-1) \geq A(m', n')$ gilt.

$$
\begin{aligned}
A(m+1, n-1) \quad &= A(m, A(m+1, n-2)) && \text{(Def)} \\
&\geq A(m, A(m, n-1)) && \text{(Ind+Mono)} \\
&\geq A(m-1, A(m, n-1) + 1) && \text{(Ind)} \\
&\geq A(m-1, A(m, n-1)) && \text{(Mono)} \\
&= A(m, n) && \text{(Def)}
\end{aligned}
$$

Das Wachstumslemma

Wir betrachten ein fixes LOOP-Programm P mit den k Variablen $x_1, x_2, \dots, x_k$ unter der Annahme, dass in LOOP-Konstrukten LOOP x_i DO P ENDLOOP die Zählervariable x_i nicht im inneren Programmteil P vorkommt. Andernfalls führen wir für die Schleife eine frische Zählervariable x_i' ein.

Für die Anfangswerte $\vec{a} = (a_1, \dots, a_k) \in \mathbb{N}^k$ der Variablen definieren wir $f_P(\vec{a}) := b_1 + b_2 + \cdots + b_k$ als die Summe aller Ergebniswerte in $[P](\vec{a}) = (b_1, \dots, b_k)$.
Die Wachstumsfunktion $F_P(n) = \max\{f_P(\vec{a}) | \, \vec{a} \in \mathbb{N}^k \text{ mit } \sum_{i=1}^{k} a_i \leq n\}$
(Funktion f_P beschreibt das maximale Wachstum der Variablensumme durch das LOOP-Programm P)

Wachstumslemma: Für jedes LOOP-Programm P existiert eine natürliche Zahl m_P, sodass für alle $n \in \mathbb{N}$ gilt: $F_P(n) < A(m_P, n)$

Beweisführung:

Induktionsanfang (Zuweisungen):

- Es sei P von der Form: $x_i \coloneqq x_j + c$ mit $c \in \{0,1\}$
  => $F_P(n) < A(2,n)$
- Argument:
  - P verändert nur den Wert der Variablen x_i
  - Am Anfang war $x_i \geq 0$, und am Ende ist $x_i = x_j + c \leq n + 1$
  - Die Summe aller Variablenwerte erhöht sich um höchstens $n + 1$, und springt damit von höchstens n (am Anfang) auf höchstens $2n + 1$ (am Ende)

Induktionsschritt (Hintereinanderausführung):

- Es sei P von der Form: $P_1; P_2$
- Induktionsannahme: $\exists q \in \mathbb{N}: F_{P_1}(l) < A(q,l)$ und $F_{P_2}(l) < A(q,l)$
  => $F_P(n) < A(q+1,n)$
- Argument:
  - Verwendung der Abschätzung 3 des Monotonieverhaltens: $A(m+1,n-1) \geq A(m,n)$
  - Dann folgt mit den Monotonieeigenschaften, dass

$F_P(n)$	$\leq F_{P_2}(F_{P_1}(n))$	(Def)
	$\leq A(q, A(q,n))$	(Ind)
	$\leq A(q, A(q+1, n-1))$	(Mono + 3. Abschätzung)
	$\leq A(q+1,n)$	(Def)

Induktionsschritt (LOOP-Konstrukt):

- Es sei P von der Form: „LOOP x_i DO Q ENDLOOP"
- Induktionsannahme: $\exists q \in \mathbb{N}: F_Q(l) < A(q,l)$
  => $F_P(n) < A(q+1,n)$
- Argument:
  - Wir betrachten jenen Wert $\alpha = \alpha(n)$ für die Variable x_i, mit dem der größtmögliche Funktionswert $F_P(n)$ angenommen wird
  - Dann gilt $F_P(n) \leq F_Q\left(F_Q\left(\dots F_Q\left(F_Q(n-\alpha)\right)\dots\right)\right) + \alpha$, wobei die Funktion $F_Q(\cdot)$ hier α-fachineinander eingesetzt ist
  - Die Induktionsannahme gibt uns $F_Q(l) < A(q,l)$
  - Dies wenden wir auf die äußerste Funktion F_Q an und erhalten
    $F_P(n) < A\left(q, F_Q\left(\dots F_Q\left(F_Q(n-\alpha)\right)\dots\right)\right) + \alpha$
  - Wiederholte Anwendung liefert die α-zeilige Ungleichungskette
    $F_P(n) < A(q, A(q, A\left(q, \dots, A(q, n-\alpha)\right)\dots)) + \alpha$
    Daraus ergibt sich $F_P(n) \leq A(q, A(q, A\left(q, \dots, A(q, n-\alpha)\right)\dots))$
  - Im innersten Teil verwenden wir $A(q, n-\alpha) \leq A(q+1, n-\alpha-1)$
  - Ergo: $F_P(n) \leq A(q, A(q, A\left(q, \dots, A(q, A(q+1, n-\alpha-1))\right)\dots))$
  - Jetzt arbeiten wir uns $(\alpha - 1)$-mal von innen nach außen vor, und wenden in jedem Schritt die Ackermann Definition an.
  - Das ergibt: $F_P(n) \leq A(q+1, n-1)$ und $F_P(n) < A(q+1,n)$

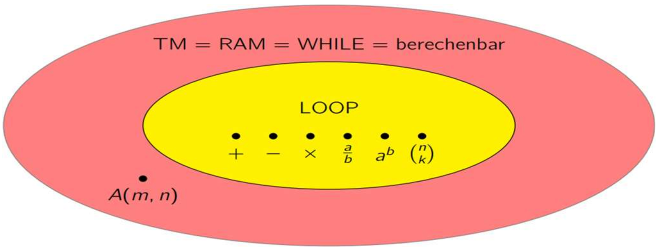
Mächtigkeit von LOOP

Satz: Die Ackermann Funktion ist nicht LOOP-berechenbar.

Beweis:

- Zwecks Widerspruchs nehmen wir an, dass die Ackermann Funktion LOOP-berechenbar ist. Dann gibt es auch ein LOOP-Programm P, das die Hilfsfunktion $B(n) = A(n, n)$ berechnet.
- Das Wachstumslemma liefert uns eine Zahl m_P, sodass für alle $n \in \mathbb{N}$ gilt: $F_P(n) < A(m_p, n)$
- Aufruf des LOOP-Programm P mit Eingabe m_P, dann berechnet P Funktionswert $B(m_P)$
- Da F_P das Wachstum des Programms P beschränkt, gilt per Definition: $B(m_P) \leq F_P(m_P)$
- Zusammengefasst: $B(m_P) \leq F_P(m_P) < A(m_P, m_P) = B(m_P)$ ↯Widerspruch

Da die Ackermann Funktion WHILE-berechenbar ist, kann gefolgert werden, dass:
Menge der LOOP-berechenbaren Funktionen bildet echte Teilmenge der berechenbaren Funktionen
Ergo: Die Programmiersprache LOOP ist nicht Turing-mächtig.



Primitiv rekursive Funktionen

Die primitiv rekursiven Funktionen bilden eine Unterfamilie der Funktionen $\mathbb{N}^k \to \mathbb{N}$ mit $k \geq 1$.
Sie sind induktiv definiert und werden durch zwei Operationen aus den Basisfunktionen zusammen-gebaut.

Basisfunktionen:

- Alle konstanten Funktionen sind primitiv rekursiv
- Alle identischen Abbildungen (Projektionen auf eine der Komponenten) sind primitiv rekursiv
- Die Nachfolgerfunktionen $succ(n) = n + 1$ ist primitiv rekursiv

Beispiel: $\pi_{6,4}(a, b, c, d, e, f) = d$

Operationen (ebenfalls primitiv rekursiv):

- Jede Komposition von primitiv rekursiven Funktionen ist primitiv rekursiv
- Jede Funktion, die durch primitive Rekursion aus prim. rek. Funktionen entsteht ist prim. rek. Wenn die beiden Funktionen $g: \mathbb{N}^k \to \mathbb{N}$ und $h: \mathbb{N}^{k+2} \to \mathbb{N}$ prim. rek. sind, so ist auch die wie folgt definierte Funktion $f: \mathbb{N}^{k+1} \to \mathbb{N}$ prim. rek.:
$$f(0, x_1, \dots, x_k) = g(x_1, \dots, x_k)$$
$$f(n + 1, x_1, \dots, x_k) = h(n, f(n, x_1, \dots, x_k), x_1, \dots, x_k)$$

Jede primitiv rekursive Funktion ist berechenbar und total.

- Klar für Basisfunktionen
- Komposition von berechenbaren und totalen Funktionen ist ebenfalls berechenbar und total:
  - Angenommen $g: \mathbb{N}^k \to \mathbb{N}$ und $h: \mathbb{N}^{k+2} \to \mathbb{N}$ berechenbar, dann berechne der Reihe nach:
  - $y_0 \coloneqq g(x_1, \dots, x_k)$
  - $y_1 \coloneqq h(0, y_0, x_1, \dots, x_k)$
  - $y_2 \coloneqq h(1, y_1, x_1, \dots, x_k)$
  - …
  - $y_n \coloneqq h(n - 1, y_{n-1}, x_1, \dots, x_k)$ => Damit auch $f(n, x_1, \dots, x_k) = y_n$ berechnet.

Beispiel: Addition
Die Additionsfunktion $add: \mathbb{N}^2 \to \mathbb{N}$ mit $add(x, y) = x + y$ ist primitiv rekursiv:

$$add(0, x) = x$$
$$add(n + 1, x) = succ(add(n, x))$$

Genauer: $add: \mathbb{N}^2 \to \mathbb{N}$ entsteht durch prim. Rekursion aus der identischen Abbildung $g(x) = x$ und aus der prim. rek. Funktion $h(a, b, c) = succ(b)$:

$$add(0, x) = g(x)$$
$$add(n + 1, x) = h(n, add(n, x), x)$$

Beispiel: Multiplikation
Die Multiplikationsfunktion $mult: \mathbb{N}^2 \to \mathbb{N}$ mit $mult(x, y) = x * y$ ist primitiv rekursiv:

$$mult(0, x) = 0$$
$$mult(n + 1, x) = add(mult(n, x), x)$$

Beispiel: Vorgänge
Die Vorgängerfunktion $pred: \mathbb{N} \to \mathbb{N}$ mit $pred(x) = \max\{x - 1,0\}$ ist primitiv rekursiv:

$$pred(0) = 0$$
$$pred(n + 1) = n$$

Beispiel: Subtraktion
Die (modifizierte) Subtraktionsfunktion $sub: \mathbb{N}^2 \to \mathbb{N}$ mit $sub(x, y) = x \dot{-} y = \max\{x - y, 0\}$ ist primitiv rekursiv:
Dazu definieren wir die Hilfsfunktion $aux(y, x) = x \dot{-} y$.

$$aux(0, x) = x$$
$$aux(n + 1, x) = pred(aux(n, x))$$

Dann definieren wir $sub(x, y) = aux(x, y)$

Weitere primitiv rekursive Funktionen:

- Signum: $f: \mathbb{N} \to \{0,1\}$ mit $f(x) = [x \geq 1]$
- Gleichheit: $f: \mathbb{N}^2 \to \{0,1\}$ mit $f(x, y) = [x = y]$
- Kleiner: $f: \mathbb{N}^2 \to \{0,1\}$ mit $f(x, y) = [x < y]$
- Maximum: $f: \mathbb{N}^2 \to \mathbb{N}$mit $f(x, y) = \max\{x, y\}$
- Minimum: $f: \mathbb{N}^2 \to \mathbb{N}$ mit $f(x, y) = \min\{x, y\}$
- Division: $f: \mathbb{N}^2 \to \mathbb{N}$ mit $f(x, y) = \lfloor x / y \rfloor$
- Parität: $f: \mathbb{N} \to \{0,1\}$ mit $f(x) = [x \; ungerade]$
- Teilbarkeit: $f: \mathbb{N}^2 \to \{0,1\}$ mit $f(x, y) = [x|y]$
- Primzahl: $f: \mathbb{N}^2 \to \{0,1\}$ mit $f(x) = [x \; prim]$

Primitiv rekursive Bijektionen

Beispiele:

- Eine Funktion $binom_2: \mathbb{N} \to \mathbb{N}$ mit $binom_2(x) = \binom{x}{2} = \frac{1}{2}x(x - 1)$ ist primitiv rekursiv
$$binom_2(0) = 0$$
$$binom_2(n + 1) = add(n, binom_2(n))$$
- Die Funktion $\beta: \mathbb{N}^2 \to \mathbb{N}$ mit $\beta(x, y) = \binom{x+y+1}{2} + x$ ist primitiv rekursiv und eine Bijektion zwischen $\mathbb{N}^2$und $\mathbb{N}$.
- Die Funktion $f: \mathbb{N}^3 \to \{0,1\}$ mit $f(x, y, n) = [\beta(x, y) = N]$ ist primitiv rekursiv
- Die Funktion $g: \mathbb{N}^3 \to \{0,1\}$ mit $g(x, y, n) = [\exists y \leq k: \beta(x, y) = n]$ ist primitiv rekursiv:
$$g(x, 0, n) \coloneqq f(x, 0, n)$$
$$g(x, k, n) \coloneqq f(x, k, n) + g(x, k - 1, n) * (1 - f(x, kn,))$$

- Die Funktion $h: \mathbb{N}^3 \to \mathbb{N}$ mit $h(x,k,n) = \max\{x \leq m: \exists y \leq k: \beta(x,y) = n\}$ ist prim. rek.
$$h(0,k,n) := 0$$
$$h(m,k,n) := \max\{h(m-1,k,n), m * g(m,k,n)\}$$
- Die Funktion $\gamma: \mathbb{N} \to \mathbb{N}$ mit $\gamma(n) = h(n,n,n)$ ist primitiv rekursiv
    - Also: $\gamma(n) = h(n,n,n)$ ist die größte Zahl $x \leq n$, für die eine Zahl $y \leq n$ existiert, sodass $\beta(x,y) = n$ gilt.
    - Für jedes $n \in \mathbb{N}$ gibt es ein einziges Zahlenpaar $(x,y) \in \mathbb{N}^2$ mit $\beta(x,y) = n$. Die Funktion $\gamma(n)$ gibt uns daher die (eindeutig bestimmte) Zahl x dieser Gleichung an.
    - Analog: Es gibt eine primitiv rekursive Funktion $\delta(n)$, die die (eindeutig bestimmte) Zahl y in der Gleichung $\beta(x,y) = n$ angibt

Zusammenfassend: Die Umkehrfunktionen γ und δ der Bijektion β sind primitiv rekursiv.
Für alle $n \in \mathbb{N}$ gilt $\beta\big(\gamma(n), \delta(n)\big) = n$.

Äquivalenz zu LOOP-Programmen

Satz: Die Menge der prim. rek. Funktionen fällt mit der Menge der LOOP-berechenbaren zusammen.

Beweis: LOOP → primitiv rekursiv

- Wir betrachten eine LOOP-berechenbare Funktion f, die von einem LOOP-Programm P berechnet wird. Die in P verwendeten Variablen seien $x_0, x_1, \dots, x_k$.
- Wir beweisen mit Induktion über den Aufbau von P, dass eine primitiv rekursive Funktion $g_P: \mathbb{N} \to \mathbb{N}$ existiert, die die Arbeitsweise von P simuliert:
- Wenn das Programm P die Anfangswerte $a_0, a_1, \dots, a_k$ der Variablen in die Endwerte $b_0, b_1, \dots, b_k$ übersetzt, dann gilt entsprechend $g_P(\langle a_0, a_1, \dots, a_k \rangle) = \langle b_0, b_1, \dots, b_k \rangle$
- Falls P aus der Zuweisung $x_i := x_j + c$ besteht, so ist $g_P(x) = \big\langle u_0(x), \dots, u_{i-1}(x), u_j(x) + c, u_{i+1}(x), \dots, u_k(x) \big\rangle$
- Hintereinanderausführung: Falls P die Form $Q; R$ hat, so gilt: $g_P(x) = g_P(g_Q(x))$
- LOOP-Konstrukt: Angenommen, P ist ein LOOP-Programm der Form: LOOP x_i DO Q ENDLOOP
    - Wir definieren zunächst die (prim. rek.) Hilfsfunktion $h: \mathbb{N}^2 \to \mathbb{N}$ mit
$$h(0,x) = x$$
$$h(n+1,x) = g_P(h(n,x))$$
    - Der Aufruf $h(n,x)$ wendet das Programm Q n-mal hintereinander auf Eingabe x an
    - Da der Wert x_i am Anfang der Schleife durch $u_i(x)$ gegeben ist, gilt $g_P(x) = h(u_i(x), x)$
- Anfang von LOOP: Die Eingabe ist in den Variablen $x_1, \dots, x_m$ enthalten. Alle anderen mit 0 init.
    - Wenn die m Eingabevariablen die Werte $x_1', \dots, x_m'$ haben, so ist der Eingabewert für die primitiv rekursive Simulation $x = \left\langle 0, x_1', x_2', \dots, x_m', \underbrace{0,0, \dots, 0}_{k-m} \right\rangle$
- Ende von LOOP: Das Resultat ist die Zahl, die sich am Ende in der Variablen x_0 ergibt.
    - Resultat von P: $u_0(g_P(\langle 0, x_1', x_2', \dots, x_m', 0,0, \dots, 0 \rangle))$.

Beweis: primitiv rekursiv → LOOP

- Wir betrachten eine prim. rek. Funktion $f: \mathbb{N}^k \to \mathbb{N}$ und beweisen mit Induktion über die Definition von f, dass ein LOOP Programm P_f existiert, das die Auswertung von f simuliert: Das Programm P_f nimmt die Variablenwerte $x_1, \dots, x_k$ und terminiert mit $x_0 = f(x_1, \dots, x_k)$.
- Basisfunktionen:
    - konstante Funktion $f(x_1, \dots, x_k) = c$ wird durch LOOP-Programm $x_0 := c$ simuliert
    - Projektion $f(x_1, \dots, x_k) = x_j$ wird durch LOOP-Programm $x_0 := x_j$ simuliert
    - Nachfolgerfunktion $succ(x_j) = x + 1$ wird durch LOOP-Programm $x_0 := x_j + 1$ simuliert

- Komposition: Falls die prim. rek. Funktion f durch Komposition aus anderen prim. rek. Funktionen $f_1, \dots, f_s$ entsteht, so wird f simuliert, indem man die LOOP-Programme für $f_1, \dots, f_s$ geeignet hintereinander ausführt
- Primitive Rekursion: Angenommen die Funktion $f: \mathbb{N}^{k+1} \to \mathbb{N}$ entsteht durch primitive Rekursion aus den beiden primitiv rekursiven Funktionen $g: \mathbb{N}^k \to \mathbb{N}$ und $h: \mathbb{N}^{k+2} \to \mathbb{N}$, wobei
$$f(0, x_1, \dots, x_k) = g(x_1, \dots, x_k)$$
$$f(n+1, x_1, \dots, x_k) = h(n, f(n, x_1, \dots, x_k), x_1, \dots, x_k)$$
- Das simulierte LOOP-Programm P_f sieht dann wie folgt aus:
  $x_0 := g(x_1, \dots, x_k);$
  $s := 0;$
  LOOP n DO
  $\quad x_0 := h(s, x_0, x_1, \dots, x_k);$
  $\quad s := s + 1;$
  ENDLOOP;

μ-rekursive Funktionen

In der folgenden Funktion gilt: $min\emptyset = \perp$.
Es sei $g: \mathbb{N}^{k+1} \to \mathbb{N} \cup \{\perp\}$ eine Funktion mit $k+1$ Argumenten. Durch Anwendung des μ-Operators auf g entsteht eine neue Funktion $f: \mathbb{N}^k \to \mathbb{N} \cup \{\perp\}$ mit k Argumenten, wobei
$f(x_1, \dots, x_k) = \min\{n | g(n, x_1, \dots, x_k) = 0$ und für alle $m < n$ gilt $g(m, x_1, \dots, x_k) \neq \perp\}$.
Die entstehende Funktion f wird mit $\mu g: \mathbb{N}^k \to \mathbb{N} \cup \{\perp\}$ bezeichnet.

Beispiele:
- Es sei $g: \mathbb{N}^2 \to \mathbb{N}$ mit $g(x, y) = 5x^2 - 4xy - y^2 - 19$.
  Dann ist μg die kleinste Zahl $x \in \mathbb{N}$ mit $5x^2 - 4xy - y^2 = 19$
- Es sei $g: \mathbb{N}^3 \to \mathbb{N}$ mit $g(x, y, z) \equiv 1$. Dann gilt $\mu g(y, z) \equiv \perp$

Die Klasse der μ-rekursiven Funktionen ist die kleinste Klasse von (partiellen und totalen) Funktionen,
- die die Basisfunktionen (konstant Funktionen; identischen Abb.; Nachfolgefunktionen) enthält und
- die abgeschlossen ist unter Komposition, primitiver Rekursion und Anwendung des μ-Operators

Satz: Die Menge der μ-rekursiven Funktionen fällt mit der Menge der WHILE-berechenbaren (Turing-berechenbaren; RAM-berechenbaren) Funktionen zusammen.

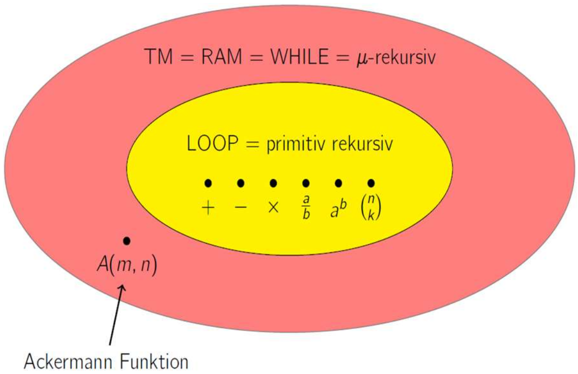
- Der Beweis dieses Satzes basiert auf dem Beweis von primitiv rekursiv $\Leftrightarrow$ LOOP-berechenbar und bildet eine einfache Erweiterung
- In der einen Richtung: WHILE Schleifen können mit Hilfe des μ-Operators simuliert werden
- In der anderen Richtung: μ-Operator kann durch eine WHILE Schleife simuliert werden

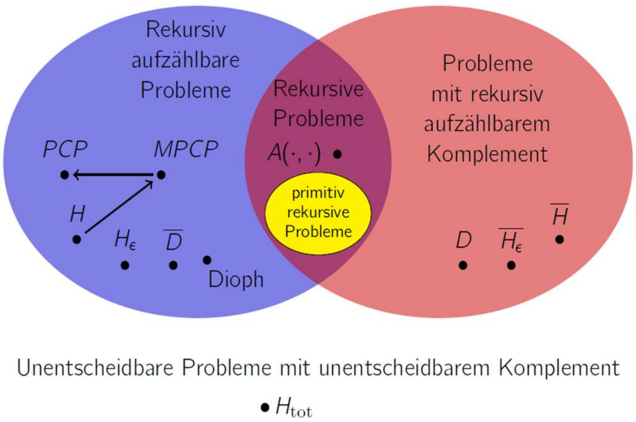
Beweis: WHILE $\to$ μ-rekursiv
- Simulation eines WHILE Programms P: WHILE $x_i \neq 0$ DO Q ENDWHILE
- Wir recyclen die Hilfsfunktion $h: \mathbb{N}^2 \to \mathbb{N}$, die durch den Aufruf $h(n, x)$ das Programm Q genau n-mal auf x anwendet:
$$h(0, x) = x$$
$$h(n+1, x) = g_Q(h(n, x))$$
- $u_i(h(n, x))$ gibt den Wert der Variablen x_i nach n-maliger Anwendung von Q an
- $\mu u_i(h(n, x))$ gibt dann die kleinste Zahl n an, für die die Variable x_i nach n-maliger Anwendung von Q den Wert 0 hat
- Daher wird das WHILE Programm P simuliert durch $g_P(x) = h(\mu u_i(h(n, x)), x)$

Beweis: μ-rekursiv → WHILE

- Simulation des μ-Operators
- Angenommen, die Funktion $f: \mathbb{N}^k \to \mathbb{N}$ entsteht durch den μ-Operator aus der μ-rekursiven Funktion $g: \mathbb{N}^{k+1} \to \mathbb{N}$, also $f = \mu g$
- Das simulierte WHILE Programm P_f sieht dann wie folgt aus:
  $x_0 \coloneqq 0;$
  $y \coloneqq g(0, x_1, \dots, x_k);$
  WHILE $y \neq 0$ DO
  $\quad x_0 \coloneqq x_0 + 1;$
  $\quad y \coloneqq g(x_0, x_1, \dots, x_k);$
  ENDWHILE;

Zusammenfassung Berechenbarkeit





Turing-mächtige Rechenmodelle und Programmiersprachen

- Turingmaschine (TM)
- k-Band TM
- Registermaschine (RAM)
- WHILE-Programme (und somit C, Java, PostScript, etc.)
- μ-rekursive Funktionen (und somit LISP, Haskell, OCaml, etc.)

Nicht-Turing-mächtige Rechenmodelle und Programmiersprachen

- LOOP-Programme
- Primitiv rekursive Funktionen

Church-Turing These

Die Klasse der TM-berechenbaren Funktionen stimmt mit der Klasse der „intuitiv berechenbaren" Funktionen überein. (In anderen Worten: Ein Problem kann genau dann „algorithmisch gelöst werden", wenn es eine TM für dieses Problem gibt)

Methoden zum Nachweis von Nicht-Berechenbarkeit

- Diagonalisierung
- Unterprogrammtechnik
- Reduktion
- *Satz von Rice*: Aussagen über Eigenschaften von Funktionen, die durch eine gegebene TM berechnet werden, sind nicht entscheidbar

Wichtige nicht berechenbare Probleme:

- Halteproblem (in den verschiedenen Varianten)
- Postsches Korrespondenzproblem
- verschiedene Entscheidungsprobleme für CFGs
- Erkennung von Funktionen mit elementaren Stammfunktionen
- Hilberts zehntes Problem
- *Schlussfolgerung aus Rice*: Die automatische Verifikation von Programmen in TM-mächtigen Programmiersprachen ist unmöglich

P vs. NP

Komplexitätstheorie

Die Komplexitätstheorie versucht (entscheidbare) algorithmische Probleme nach ihrem Bedarf an Berechnungsressourcen (Zeit und Speicher) zu klassifizieren und sie in Komplexitätsklassen einzuteilen

Polynomielle Algorithmen

Die Worst Case Laufzeit $t_A(n)$ eines Algorithmus A …

- … misst die maximalen Laufzeitkosten auf Eingaben der Länge n bezüglich des logarithmischen Kostenmaßes der RAM.
- … ist polynomiell beschränkt, falls gilt: $\exists \alpha \in \mathbb{N}: t_A(n) \in O(n^\alpha)$
    - Einen Algorithmus mit polynomiell beschränkter Worst Case Laufzeit bezeichnen wir als polynomiellen Algorithmus oder auch als Polynomialzeitalgorithmus.
    - Beispiele: $O(n); O(n \log n); O(n^3); O(n^{100})$

Problem: SORTIEREN
Eingabe: Zahlen $a_1, \dots, a_n \in \mathbb{N}$ in Binärdarstellung
Ausgabe: Die aufsteigend sortierte Folge der Eingabezahlen
Satz: SORTIEREN $\in P$
Beweis:

- Wir lösen das Problem mit $MergeSort$ oder $HeapSort$
- Laufzeit im uniformen Kostenmaß: $O(n \log n)$
- Laufzeit im logarithmischen Kostenmaß:
    - $O(\ln \log n)$, wobei $l = max_{1 \leq i \leq n} \log a_i$
    - Für die Gesamtlänge L der Eingabe gilt $L \geq l$ und $L \geq n$
    - Somit ist die Laufzeit durch $\ln \log n \leq L^3$ beschränkt

Folgende Probleme können in polynomieller Zeit gelöst werden:

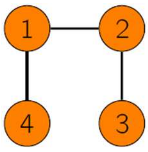
- Eulerkreis
- Kürzester Weg
- Minimaler Spannbaum
- Maximum Matching
- Größter gemeinsamer Teiler
- Konvexe Hülle in 2D
- Primzahltest

P ist die Klasse aller Entscheidungsprobleme, für die es einen polynomiellen Algorithmus gibt.
Anmerkung:

- **P** steht für Polynomiell
- **P** enthält ausschließlich Entscheidungsprobleme

- Statt der RAM könnte man in der Definition der polynomiellen Laufzeit und der polynomiellen Algorithmen genauso gut die TM verwenden: RAM (mit logarithmischen Kostenmaß) und TM simulieren einander ja mit polynomiellem Zeitverlust
- Polynomielle Algorithmen werden oft als effiziente Algorithmen bezeichnet, und **P** als die Klasse der effizient lösbaren Probleme

Problem: GRAPHZUSAMMENHANG
Eingabe: Ein ungerichteter Graph $G = (V, E)$
Ausgabe: Ist G zusammenhängend?
Anmerkung: Beim Graphproblem wird grundsätzlich von einer Adjazenzmatrix ausgegangen



$$\rightsquigarrow \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \rightsquigarrow 0101101001001000$$

Satz: GRAPHZUSAMMENHANG $\in P$
Beweis:

- Wir lösen das Problem mit Tiefensuche (DFS)
- Laufzeit im uniformen Kostenmaß: $O(|V| + |E|) = O(|V|^2)$
- Laufzeit im logarithmischen Kostenmaß: $O(|V|^2 * \log|V|)$
- Die Gesamtlänge der Eingabe ist $L = |V|^2$
- Die Gesamtlaufzeit liegt somit in $O(|V|^2 \log|V|) \subseteq O(L \log L) \subseteq O(L^2)$

Die non-deterministische Turingmaschine (NTM)

Eine non-deterministische Turingmaschine (NTM) verfügt über

- ein beidseitig unbeschränktes Arbeitsband,
- einen Lese/Schreibkopf, und
- einen Mechanismus, der die Zustandsüberführungen steuert.

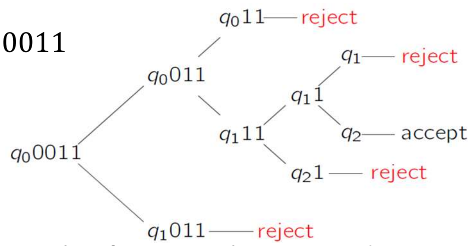
Der einzige Unterschied zur deterministischen Turingmaschine TM besteht darin, dass die Zustands-überführungen bei der NTM nicht durch eine Funktion sondern durch eine Relation gesteuert werden

$$\delta \subseteq ((Q \backslash \{\bar{q}\} \times \Gamma) \times (Q \times \Gamma \times \{L, R, N\})$$

Berechnungen auf einer NTM sind nicht deterministisch, da es zu einer Konfiguration mehrere Nachfolgekonfigurationen geben kann.

Die unten abgebildete NTM hat auf der Eingabe $w = 0011$ den rechts abgebildeten Berechnungsbaum

δ	0	1	B
q_0	$\{(q_0, B, R), (q_1, B, R)\}$	$\{$reject$\}$	$\{$reject$\}$
q_1	$\{$reject$\}$	$\{(q_1, B, R), (q_2, B, R)\}$	$\{$reject$\}$
q_2	$\{$reject$\}$	$\{$reject$\}$	$\{$accept$\}$



Rechenweg einer NTM = Konfigurationsfolge, die mit Startkonfiguration beginnt und mit Nachfolgekonfigurationen fortgesetzt wird, bis eine Endkonfiguration im Zustand $\bar{q}$ erreicht wird

Die möglichen Rechenwege einer NTM M auf einer Eingabe können in einem Berechnungsbaum zusammengefasst werden:

- Die Knoten des Baumes entsprechen Konfigurationen
- Die Wurzel des Baumes entspricht der Startkonfiguration
- Die Kinder einer Konfiguration entsprechen den möglichen Nachfolgekonfigurationen

Der maximale Verzweigungsgrad des Berechnungsbaumes ist $\Delta \coloneqq \max\{|\delta(q, a)| \mid q \in Q \backslash \{\bar{q}\}, a \in \Gamma$

Akzeptanzverhalten der NTM:
Eine NTM M akzeptiert die Eingabe $x \in \Sigma^*$, falls es mindestens einen Rechenweg von M gibt, der in eine Konfiguration mit akzeptierendem Zustand führt.
Die von der NTM M erkannte Sprache $L(M)$ besteht aus allen von M akzeptierten Wörtern

Die Laufzeit einer NTM M auf einer Eingabe x ist wie folgt definiert.

- Falls $x \in L(M)$, so ist die Laufzeit $T_M(x)$ die Länge des kürzesten akzeptierenden Rechenweges von M auf x
- Falls $x \notin L(M)$, so ist die Laufzeit $T_M(x) = 0$.

Die Worst Case Laufzeit $t_M(n)$ der NTM M auf Eingaben der Länge $n \in \mathbb{N}$ ist definiert als
$t_M(n) = \max\{T_M(x) | x \in \Sigma^n\}$

Die Komplexitätsklasse NP

NP ist die Klasse aller Entscheidungsprobleme, die durch eine NTM M erkannt werden, deren Worst Case Laufzeit $t_M(n)$ polynomiell beschränkt ist. (**NP** = **N**on-deterministisch **P**olynomiell)

Problem: CLIQUE
Eingabe: Ein ungerichteter Graph $G(V, E)$; eine Zahl k
Frage: Enthält G eine Clique mit $\geq k$ Knoten?
Satz: CLIQUE ∈ NP
Beweis: Wir beschreiben eine NTM M mit $L(M) =$ CLIQUE

- Syntaktisch inkorrekte Eingaben werden verworfen
- M rät non-deterministisch einen 0-1-String y der Länge $|V|$
- M akzeptiert, falls der String y genau k Einsen enthält und falls die Knotenmenge $C = \{i \in V \mid y_i = 1\}$ eine Clique bildet

Korrektheit: ∃ akzeptierender Rechenweg ⇔ G enthält k-Clique
Laufzeit: Jede Phase kostet polynomielle Zeit

Alternative Charakterisierung von NP

Eine Sprache $L \subseteq \Sigma^*$ liegt genau dann in NP, wenn es einen polynomiellen (deterministischen) Algorithmus V und ein Polynom p mit der folgenden Eigenschaft gibt:

$$x \in L \Leftrightarrow \exists y \in \{0,1\}^*, |y| \leq p(|x|): V \text{ akzeptiert } y\#x$$

Anmerkungen:

- Der polynomielle Algorithmus V heißt auch Verifizierer
- Das Wort $y \in \{0,1\}^*$ heißt auch Zertifikat

Beweis: NTM ⇒ Zertifikat

- Es sei M eine NTM, die L in polynomieller Zeit erkennt
- Die Laufzeit von M sei beschränkt durch ein Polynom q
- Der maximale Verzweigungsgrad eines Berechnungsbaumes sei Δ
- Konstruktion von Zertifikat und Verifizierer:
    - Für die Eingabe $x \in L$ betrachten wir den kürzesten akzeptierenden Rechenweg im Berechnungsbaum
    - Das Zertifikat y kodiert den akzeptierenden Rechenweg Schritt für Schritt, mit dem $d := \log_2 \Delta$ Bits pro Verzweigung
    - Das Zertifikat hat polynomielle Länge $|y| \leq d * q(|x|)$
    - Der Verifizierer V erhält $y\#x$ und simuliert den Rechenweg der NTM M für die Eingabe x

Beweis: NTM ⇐ Zertifikat

- Es sei V ein Verifizierer mit polynomieller Laufzeitschranke
- Das Polynom p beschränkt die Länge des Zertifikats
- Konstruktion von NTM:
  - Für die Eingabe x rät M zunächst non-deterministisch das Zertifikat $y \in \{0,1\}^*$ mit $|y| \leq p(|x|)$
  - Dann simuliert M den Verifizierer V auf dem Wort $y\#x$, und akzeptiert, wenn der Verifizierer akzeptiert
  - Das Zertifikat wird in polynomieller Zeit $p(|x|)$ geraten. Die Zeit für die Simulation ist polynomiell beschränkt in der polynomiellen Laufzeit des Verifizierers.

Katalog von Problemen in NP

Satisfiability (SAT)

Eingabe: Eine Boole'sche Formel φ in CNF über einer Boole'schen Variablenmenge $X = \{x_1, \dots, x_n\}$
Frage: Existiert eine Wahrheitsbelegung von X, die φ erfüllt?

- Literal: positive oder negierte Variable
- Klausel: ODER-Verknüpfung von einigen Literalen

Beispiele:

- $\varphi_1 = (x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$
- $\varphi_1 = (x \vee y \vee z) \wedge (\bar{x} \vee \bar{y} \vee \bar{z})$
- $\varphi_2 = (x + y)(\neg x + y)(x + \neg y)(\neg x + \neg y)$
- $\varphi_2 = (x + y)(\bar{x} + y)(x + \bar{y})(\bar{x} + \bar{y})$

Frage: Wie sieht ein NP-Zertifikat für SAT aus?

Clique / Independent Set / VC

Problem: Clique
Eingabe: Ein ungerichteter Graph $G = (V, E)$; eine Zahl k
Frage: Enthält G eine Clique mit $\geq k$ Knoten?
Problem: Independent Set (INDEP-SET)
Frage: Enthält G eine unabhängige Menge mit $\geq k$ Knoten?
Problem: Vertex Cover (VC)
Frage: Enthält G ein Vertex Cover mit $\geq k$ Knoten?

- Unabhängige Menge (independent set) $S \subseteq V$: spannt keine Kanten
- Vertex Cover $S \subseteq V$: berührt alle Kanten

Frage: Wie sieht ein NP-Zertifikat für CLIQUE / INDEP-SET / VC aus?

Hamiltonkreis / TSP

Problem: Hamiltonkreis (Ham-Cycle)
Eingabe: Ein ungerichteter Graph $G = (V, E)$
Frage: Besitzt G einen Hamiltonkreis?
Problem: Travelling Salesman Problem (TSP)
Eingabe: Städte $1, \dots, n$; Distanzen $d(i, j)$; eine Zahl γ
Frage: Gibt es eine Rundreise (TSP-Tour) mit Länge höchstens γ?
Frage: Wie sieht ein NP-Zertifikat für Ham-Cycle / TSP aus?

Exact Cover (EX-COVER)

Eingabe: Eine endliche Menge X; Teilmengen $S_1, \dots, S_m$ von X
Frage: Ex. eine Indexmenge $I \subseteq \{1, \dots, m\}$, sodass die Menge S_i mit $i \in I$ eine Partition von X bilden?
Frage: Wie sieht ein NP-Zertifikat für Exact Cover aus?

SUBSET-SUM / PARTITION

Problem: SUBSET-SUM
Eingabe: Positive ganze Zahlen $a_1, \dots, a_n$; eine ganze Zahl b
Frage: Existiert eine Indexmenge $I \subseteq \{1, \dots, n\}$ mit $\Sigma_{i \in I}\, a_i = b$?
Problem: PARTITION
Eingabe: Positive ganze Zahlen $a_1, \dots, a_n$; mit $\Sigma_{i \in I}^{n}\, a_i = 2A$
Frage: Existiert eine Indexmenge $I \subseteq \{1, \dots, n\}$ mit $\Sigma_{i \in I}\, a_i = A$?
Frage: Wie sieht ein NP-Zertifikat für SUBSET-SUM / PARTITION aus?

P vs. NP

Falls P = NP ist:

- Viele schwierige Probleme aus Wirtschaft und Industrie können schnell gelöst werden
- Perfekte Fahrpläne, Produktionspläne, Transportpläne, etc.
- Die Mathematik erreicht eine neue Stufe: Wenn es für ein Theorem einen kurzen Beweis gibt, so können wir diesen Beweis auch finden
- Die moderne Kryptographie bricht zusammen

Falls P ≠ NP ist:

- Schwierige Probleme aus Wirtschaft und Industrie können nur durch viel Rechenzeit und Expertenwissen attackiert werden
- Perfekte Lösungen für schwierige Probleme mit großen Datenmengen sind nicht zu erwarten
- Mathematik und Kryptographie arbeiten genauso weiter wie bisher

Polynomielle Reduktionen

Lösbarkeit finden vs. Lösbarkeit entscheiden

Ein beliebiges Entscheidungsproblem in NP
Eingabe: Ein diskretes Objekt X.
Frage: Existiert für dieses Objekt X eine Lösung Y ?
Dilemma:

- Entscheidungsproblem beschäftigt sich nur mit der Frage, ob eine Lösung Y existiert
- eigentlich will man das Lösungsobjekt Y auch genau bestimmen, und dann damit arbeiten

Ausweg:

- Ein schneller Algorithmus für das Entscheidungsproblem liefert (durch wiederholte Anwendung) oft auch einen schnellen Algorithmus zum Berechnen eines expliziten Lösungsobjekts

Problem: SAT
Eingabe: Boole'sche Formel φ in CNF über $X = \{x_1, \dots, x_n\}$
Frage: Existiert eine Wahrheitsbelegung von X, die φ erfüllt?

- Wenn wir in φ eine Variable $x := 1$ setzen,
- so werden alle Klauseln mit Literal x dadurch erfüllt und
- in allen Klauseln mit Literal $\bar{x}$ fällt dieses Literal einfach weg.
- Wir erhalten wir eine kürzere CNF-Formel $\varphi[x = 1]$.
- Analog erhalten wir mit $x := 0$ die CNF-Formel $\varphi[x = 0]$.

Beispiel:
Für $\varphi = (x \lor y \lor z) \land (\neg x \lor \neg y \lor\ \neg z) \land (\neg y \lor z) \land (u \lor z)$
gilt $\varphi[y = 1] = (\neg x \lor \neg z) \land (z) \land (u \lor z)$
und $\varphi[z = 0] = (x \lor z) \land (\neg y) \land (u)$

Wir betrachten SAT mit n Variablen und k Klauseln
Satz: Angenommen, Algorithmus A entscheidet SAT Instanzen in $T(n, m)$ Zeit. Dann gibt es einen Algorithmus B, der für erfüllbare SAT Instanzen in $n * T(n, m)$ Zeit eine erfüllende Wahrheitsbelegung konstruiert.
Beweis:

- Wir fixieren der Reihe nach Wahrheitswerte von $x_1, \dots, x_n$
- FOR $i = 1,2, \dots n$ DO
- Wenn $\varphi[x_i = 1]$ erfüllbar, so setze $x_i := 1$ und $\varphi := \varphi[x_1 = 1]$. Andernfalls setze $x_i := 0$ und $\varphi := \varphi[x_i = 0]$
- Am Ende ergeben die fixierten Wahrheitswerte von $x_1, \dots, x_n$ eine erfüllende Wahrheitsbelegung für φ

Problem: CLIQUE
Eingabe: Ein ungerichteter Graph $G = (V, E)$; eine Zahl k
Frage: Enthält G eine Clique mit $\geq k$ Knoten?

- Wenn wir aus G einen Knoten v und alle zu v inzidenten Kanten wegstreichen, so erhalten wir den kleineren Graphen $G - v$
- Falls $G - v$ eine k-Clique enthält, so ist v irrelevant
- Falls $G - v$ aber keine k-Clique enthält, so muss die Nachbarschaft $N[v]$ des Knoten v in $G - v$ eine $(k - 1)$-Clique enthalten

Satz: Angenommen, Algorithmus A entscheidet CLIQUE in $T(n)$ Zeit. Dann gibt es einen Algorithmus B, der für Ja-Instanzen in $n * T(n)$ Zeit eine k-Clique konstruiert.

Problem: Hamiltonkreis
Eingabe: Ein ungerichteter Graph $G = (V, E)$
Frage: Besitzt G einen Hamiltonkreis?

- Wenn wir aus G eine Kante e wegstreichen, so erhalten wir den kleineren Graphen $G - v$
- Falls $G - e$ einen Hamiltonkreis enthält, so ist e irrelevant
- Falls $G - e$ aber keinen Hamiltonkreis enthält, so ist e nicht irrelevant

Satz: Angenommen, Algorithmus A entscheidet Ham-Cycle in $T(n)$ Zeit. Dann gibt es einen Algorithmus B, der für Ja-Instanzen in $|E| * T(n)$ Zeit einen Hamiltonkreis konstruiert.

Optimieren vs. Lösbarkeit entscheiden

Die Eingabe eines Optimierungsproblems spezifiziert (implizit oder explizit) eine Menge $\mathcal{L}$ von zulässigen Lösungen zusammen mit einer Zielfunktion $f: \mathcal{L} \rightarrow \mathbb{N}$, die Kosten, Gewicht, oder Profit misst.
Das Ziel ist es, eine optimale Lösung in $\mathcal{L}$ zu berechnen. In Minimierungsproblemen sollen die Kosten minimiert werden, und in Maximierungsproblemen soll der Profit maximiert werden.

- Dilemma:
  - Die Klassen P und NP enthalten keine Optimierungsprobleme, sondern nur Entscheidungsprobleme
- Ausweg:
  - Optimierungsprobleme in sehr ähnliche "Entscheidungsprobleme“ umformulieren

Beispiel: Rucksackproblem

- Beim Rucksackproblem (Knapsack Problem, KP) sind n Objekte mit Gewichten $w_1, \dots, w_n$ und Profiten $p_1, \dots, p_n$ gegeben
- Außerdem ist eine Gewichtsschranke b gegeben
- Wir suchen eine Teilmenge K der Objekte, die in einen Rucksack mit Gewichtsschranke b passt und die den Gesamtprofit maximiert

Problem: Rucksack / Knapsack (KP)
Eingabe: Natürliche Zahlen $w_1, \dots, w_n, p_1, \dots, p_n$ und b
Zulässige Lösung: Menge $K \subseteq \{1, \dots, n\}$ mit $w(K) \coloneqq \Sigma_{i \in K} w_i \leq b$
Ziel: Maximiere $p(K) \coloneqq \Sigma_{i \in K} p_i$?
Entscheidungsproblem:

- Die Eingabe enthält zusätzlich eine Schranke γ für den Profit
- Frage: Existiert eine zulässige Lösung K mit $p(K) \geq \gamma$?

Beispiel: Bin Packing

- Beim Bin Packing sollen n Objekte mit Gewichten $w_1, \dots, w_n$ auf eine möglichst kleine Anzahl an Kisten mit Gewichtslimit b verteilt werden

Problem: Bin Packing (BPP)
Eingabe: Natürliche Zahlen b und $w_1, \dots, w_n \in \{1, \dots, b\}$
Zulässige Lösung: Zahl $K \in \mathbb{N}$ und Funktion $f: \{1, \dots, n\} \to \{1, \dots, k\}$
sodas $\forall_i \in \{1, \dots, k\}$: $\sum_{j \in f^{-1}(i)} w_j \leq b$
Ziel: Minimiere k (= Anzahl der Kisten)?
Entscheidungsproblem:

- Die Eingabe enthält zusätzlich eine Schranke γ
- Frage: Existiert eine zulässige Lösung mit $\leq \gamma$?

Beispiel: Travelling Salesman

- Beim Travelling Salesman Problem sind Städte $1, \dots, n$ gegeben, zusammen mit Distanzen $d(i, j)$ für $1 \leq i \neq j \leq n$ verteilt werden
- Gesucht ist eine möglichst kurze Rundreise (Hamiltonkreis; Tour) durch alle Städte

Problem: Travelling Salesman (TSP)
Eingabe: Natürliche Zahlen $d(i, j)$ für $1 \leq i \neq j \leq n$
Zulässige Lösung: Permutation π von $1, \dots, n$
Ziel: Minimiere $d(\pi) \coloneqq \sum_{i=1}^{n-1} d\big(\pi(i), \pi(i+1)\big) + d(\pi(n), \pi(1))$?
Entscheidungsproblem:

- Die Eingabe enthält zusätzlich eine Schranke γ
- Frage: Existiert eine zulässige Lösung mit Länge $d(\pi) \leq \gamma$?

Für ein Optimierungsproblem mit einer Menge $\mathcal{L}$ von zulässigen Lösungen und einer Gewichtsfunktion $f: \mathcal{L} \to \mathbb{N}$ definieren wir das entsprechende Entscheidungsproblem:
Eingabe: Wie im Optimierungsproblem; plus Schranke $\gamma \in \mathbb{N}$
Frage: Existiert eine zulässige Lösung $x \in \mathcal{L}$ mit $f(x) \geq \gamma$ (für Maximierungsprobleme) respektive mit $f(x) \leq \gamma$ (für Minimierungsprobleme)?

- Mit Hilfe eines Algorithmus, der das Optimierungsproblem löst, kann man immer das entsprechende Entscheidungsproblem lösen. (Wie?)
- Mit Hilfe eines Algorithmus, der das Entscheidungsproblem löst, kann man den optimalen Zielfunktionswert bestimmen (und oft auch die dazugehörende optimale Lösung finden).

Beispiel: Rucksackproblem
Eingabe: Natürliche Zahlen $w_1, \dots, w_n, p_1, \dots, p_n; b; \gamma$
Zulässig: Menge $K \subseteq \{1, \dots, n\}$ mit $w(K) \leq b$
Optimierung: Berechne K mit maximalem $p(K)$
Entscheidung: Existiert K mit $p(K) \geq \gamma$?

Satz: Wenn das Entscheidungsproblem für KP in polynomieller Zeit lösbar ist, so ist auch das Optimierungsproblem für KP in polynomieller Zeit lösbar.
Beweis: Aus polynomiellem Algorithmus A fürs Entscheidungsproblem

- konstruieren wir zuerst einen polynomiellen Algorithmus B, der den optimalen Zielfunktionswert bestimmt
- und dann einen polynomiellen Algorithmus C, der die optimale zulässige Lösung bestimmt

Algorithmus B für (Phase 1):
Wir führen eine Binäre Suche mit den folgenden Parametern durch:

- Der minimale Profit ist 0.
- Der maximale Profit ist $P \coloneqq \sum_{i=1}^{n} p_i$
- Wir finden den optimalen Zielfunktionswert durch BS über dem Wertebereich $\{0, \dots, P\}$
- In jeder Iteration verwenden wir den polynomiellen Algorithmus A (für das Entscheidungsproblem), der uns sagt in welcher Richtung wir weitersuchen müssen

Die Anzahl der Iterationen der Binärsuche ist $\lceil log(P+1) \rceil$.

Untersuchung der Eingabelänge:

- Die Kodierungslänge einer Zahl $a \in \mathbb{N}$ ist $\kappa(a) \coloneqq \lceil log(a+1) \rceil$
- Die Logarithmusfunktion ist sub-additiv: Für alle $a, b \in \mathbb{N}$ gilt $\kappa(a+b) \leq \kappa(a) + \kappa(b)$
- Die Eingabelänge L des Rucksackproblems beträgt mindestens $\sum_{i=1}^{n} \kappa(p_i) \geq \kappa(\sum_{1=1}^{n} p_i) = \kappa(P) = \lceil log(P+1) \rceil$

- Algorithmus B besteht aus $\lceil log(P+1) \rceil \leq L$ Aufrufen des polyn. Algorithmus A
- Also ist die Gesamtlaufzeit von Algorithmus A und B polyn. In der Eingabelänge des Rucksackproblems beschränkt
- Aus Algorithmus B konstruieren wir nun noch den Algorithmus C, der die optimale zulässige Lösung bestimmt:

```
K := {1, ..., n};
opt := B(K);
FOR i := 1 TO n do
      IF B(K \ {i}) = opt THEN K := K \ {i}; ENDIF;
ENDFOR;
OUTPUT  K
```

- Algorithmus C besteht im Wesentlichen aus $n+1$ Aufrufen des polynomiellen Algorithmus B
- Also ist die Gesamtlaufzeit von Algorithmus C polynomiell in der Eingabelänge des Rucksackproblems beschränkt

Die Komplexitätsklasse EXPTIME

EXPTIME ist die Klasse aller Entscheidungsprobleme, die durch eine DTM M entschieden werden, deren Worst Case Laufzeit durch $2^{q(n)}$ mit einem Polynom q beschränkt ist
Laufzeit-Beispiele: $2^{\sqrt{n}}, 2^n, 3^n, n!, n^n$. Aber nicht 2^{2^n}
Satz: NP ⊆ EXPTIME

- Es sei $L \in$ NP
- Dann gibt es ein Polynom p und einen polyn. Algorithmus V
  $x \in L \Leftrightarrow \exists y \in \{0,1\}^*, |y| \leq p(|x|): V$ akzeptiert $y\#x$
- Wir enumerieren alle Kandidaten $y \in \{0,1\}^*$ mit $|y| \leq p(|x|)$. Wir testen jeden Kandidaten mit dem Verifizierer V. Wir akzeptieren, falls V einen der Kandidaten akzeptiert
- Anzahl der Kandidaten $\approx 2^{p(|x|)}$
- Zeit pro Kandidaten ≈ polyn. In $|x|$ plus $|y|$
- Gesamtzeit $\approx poly(|x|) * 2^{p(|x|)}$

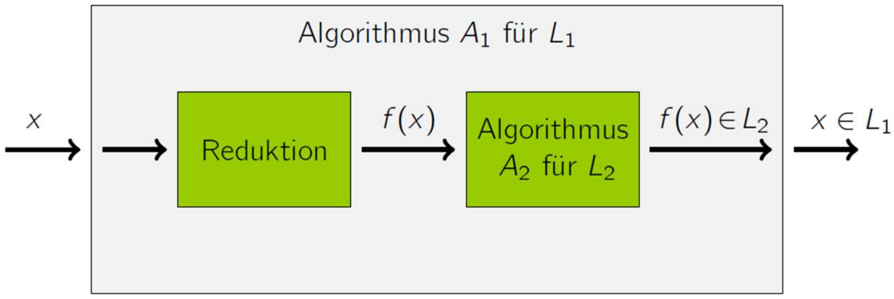
Polynomielle Reduktionen

Es seien L_1 und L_2 Sprachen über einem Alphabet Σ. Dann heißt L_1 polynomiell reduzierbar auf L_2 ($L_1 \leq L_2$), wenn eine polynomiell berechenbare Funktion $f: \Sigma^* \to \Sigma^*$ existiert, sodass für alle $x \epsilon \Sigma$ gilt: $x \in L_1 \Leftrightarrow f(x) \in L_2$

Satz: Falls $L_1 \leq_p L_2$ und falls $L_2 \in P$, so gilt $L_1 \in P$

Beweis:

- Die Reduktion f hat die polyn. Laufzeitschranke $p(\cdot)$
- Der Algorithmus A_2 entscheidet L_2 mit einer polyn. Laufzeitschranke $q(\cdot)$
- Wir konstruieren einen Algorithmus A_1, der L_1 entscheidet:
  - Schritt 1: Berechne $f(x)$
  - Schritt 2: Simuliere Algorithmus A_2 auf $f(x)$
  - Schritt 3: Akzeptiere x, g.d.w. A_2 akzeptiert
- Schritt 1 hat Laufzeit $p(|x|)$
- Schritt 2 hat Laufzeit $q(|f(x)|) \leq q(p(|x|) + |x|)$



Beispiel zu Reduktionen: COLORING $\leq_p$ SAT

Problem: Knotenfärbung / COLORING

Eingabe: Ein ungerichteter Graph $G = (V, E)$; eine Zahl $k \in \mathbb{N}$

Frage: Gibt es eine Färbung $c: V \to \{1, \dots, k\}$ der Knoten mit k Farben, sodass benachbarte Knoten verschiedene Farben erhalten? $\forall e = \{u, v\} \in E: c(u) \neq c(v)$?

Problem: SAT

Eingabe: Boole'sche Formel φ in CNF über $X = \{x_1, \dots, x_n\}$

Frage: Existiert eine Wahrheitsbelegung von X, die φ erfüllt?

Satz: COLORING $\leq_p$ SAT

Reduktion:

- Die Boole'sche Variablen
  - Für jeden Knoten $v \in V$ und für jede Farbe $i \in \{1, \dots, k\}$ führen wir eine Boole'sche Variable x_v^i ein
- Die Klauseln
  - Für jeden Knoten $v \in V$ verwenden wir eine Klausel $(x_v^1 + x_v^2 + \cdots + x_v^k)$
  - Für jede Kante $\{u, v\} \in E$ und jede Farbe $i \in \{1, \dots, k\}$ verwenden wir die Klausel $(\bar{x}_u^i + \bar{x}_v^i)$
- Anzahl der Variablen $= k|V|$
- Anzahl der Klauseln $= |V| + k|E|$
- Gesamtlänge der Formel $= k|V| + 2k|E| \in O(k|V|^2)$

Korrektheit:

Graph G hat k-Färbung $\Rightarrow$ Formel φ ist erfüllbar

- Es sei c eine k-Färbung für G
- Für jeden Knoten v mit $c(v) = i$ setzen wir $x_v^i = 1$ Alle anderen Variablen setzen wir auf 0

- Für jeden Knoten $v \in V$ ist $(x_v^1 + x_v^2 + \cdots + x_v^k)$ erfüllt
- Für $\{u, v\} \in E$ und $i \in \{1, \dots, k\}$ ist $(\bar{x}_u^i + \bar{x}_v^i)$ erfüllt (Andernfalls u und v selbe Farbe i
- Ergo: Diese Wahrheitsbelegung erfüllt die Formel φ

Formel φ ist erfüllbar ⇒ Graph G hat k-Färbung
- Wir betrachten eine beliebige erfüllende Belegung für φ
- Wegen Klausel $(x_v^1 + x_v^2 + \cdots + x_v^k)$ gibt es für jeden Knoten v mind. eine Farbe i mit $x_v^i = 1$
- Für jeden Knoten wählen wir eine beliebige derartige Farbe aus
- Wir behaupten: $c(u) \neq c(v)$ gilt für jede Kante $\{u, v\} \in E$
- Beweis: Falls $c(u) = c(v) = i$, dann gilt $x_u^i = x_v^i = 1$. Dann wäre aber Klausel $(\bar{x}_u^i + \bar{x}_v^i)$ verletzt

Konsequenzen:
- Wenn SAT einen polyn. Algorithmus hat, so hat auch COLORING einen polyn. Algorithmus
- Wenn COLORING keinen polyn. Algorithmus hat, so hat auch SAT keinen polyn Algorithmus

Nicht-Beispiel zu Reduktionen: Vertex Cover $\leq_p$ SAT

Problem: Vertex Cover (VC)
Eingabe: Ein ungerichteter Graph $G = (V, E)$; eine Zahl $k \in \mathbb{N}$
Frage: Enthält G ein Vertex Cover mit $\geq k$ Knoten?
Vertex Cover $S \subseteq V$ enthält mindestens einen Endpunkt von jeder Kante
Problem: SAT
Eingabe: Boole'sche Formel φ in CNF über $X = \{x_1, \dots, x_n\}$
Frage: Existiert eine Wahrheitsbelegung von X, die φ erfüllt?

Satz???: Vertex Cover $\leq_p$ SAT
Reduktion:
- Die Boole'sche Variablen
  - Für jeden Knoten $v \in V$ führen wir eine Boole'sche Variable x_v ein
- Die Klauseln
  - Für jede Kante $\{u, v\} \in E$ verwenden wir die Klausel $(x_u + x_v)$
  - Für jede $(k+1)$-elementige Teilmenge $S \subseteq V$ verwenden wir die Klausel $\bigvee_{v \in S} \bar{x}_v$
- Anzahl der Variablen = $|V|$
- Anzahl der Klauseln $\approx |V|^k$

Gesamtlänge der Formel $\approx k|V|^k$ ←!!!#!!&!!!!!!!

Der Satz von Cook & Levin

NP-Vollständigkeit

Ein Problem L heißt NP-schwer, falls gilt: $\forall L' \in$ NP: $L' \leq_p L$
Satz: Wenn L NP-schwer ist, dann gilt: $L \in$ P ⇒ P = NP
Beweis: Ein polynomieller Algorithmus für L liefert zusammen mit der Reduktion $L' \leq_p L$ einen polynomiellen Algorithmus für alle $L' \in$ NP.
Fazit: Für NP-schwere Probleme gibt es keine polynomiellen Algorithmen, es sei denn P=NP.

Ein Problem L heißt NP-vollständig, falls gilt:
- $L \in$ NP
- L ist NP-schwer

Die Klasse der NP-vollständigen Probleme wird mit NPC bezeichnet.
Wir werden zeigen: SAT, CLIQUE, Ham-Cycle, PARTITION, Rucksack, … sind NP-vollständig
Unter Annahme P ≠ NP besitzt keines dieser Probleme einen polynomiellen Algorithmus.

Satz von Cook & Levin

Der Ausgangspunkt für alle unsere NP-Vollständigkeitsbeweise ist das Erfüllbarkeitsproblem SAT.
Problem: SAT
Eingabe: Boole'sche Formel φ in CNF über $X = \{x_1, \dots, x_n\}$
Frage: Existiert eine Wahrheitsbelegung von X, die φ erfüllt?

Satz von Cook & Levin: SAT ist NP-vollständig.
Fazit: Wenn P $\neq$ NP, besitzt SAT keinen polynomiellen Algorithmus

Beweis des Satzes von Cook & Levin

- Es sei $L \subseteq \Sigma^*$ ein beliebiges Problem in NP. Es sei M eine NTM, die L in polyn. Zeit erkennt.
- Wir müssen/werden zeigen, dass $L \leq_p$ SAT gilt.
- Dazu konstruieren wir eine polyn. Berechenbare Funktion f, die jedes $x \in \Sigma^*$ auf eine Formel $\varphi \coloneqq f(x)$ abbildet, sodass gilt: $x \in L \Leftrightarrow M$ akzeptiert $x \Leftrightarrow \varphi \in$ SAT

Annahme folgender Eigenschaften der NTM M an:

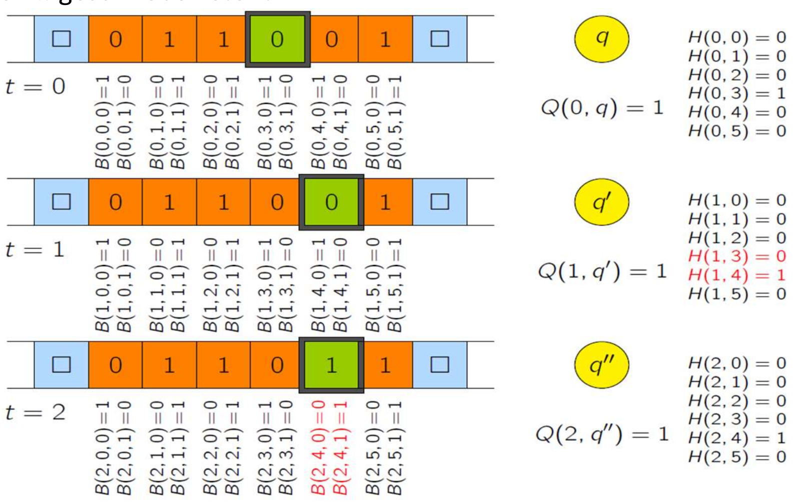
- M besucht keine Bandzelle links von der Startzelle.
- Eine akzeptierende Rechnung von M geht in den Zustand q_{accept} über, und bleibt dann dort in einer Endlosschleife.
- Es gibt ein Polynom $p(\cdot)$, sodass M eine Eingabe x genau dann
- akzeptiert, wenn es einen Rechenweg gibt, der nach $p(|x|)$ Schritten
- im Zustand q_{accept} gelandet ist.

Beobachtung

- Es sei $K_0 = q_0 x$ Startkonfiguration von M. Die NTM M akzeptiert ein Wort x mit $|x| = n$ g.d.w. es eine Konfigurationsfolge $K_0 \vdash K_1 \vdash \cdots \vdash K_{p(n)}$ gibt, bei der $K_{p(n)}$ im Zustand q_{accept} ist.
- Unsere Formel φ wird derart konstruiert, dass φ genau dann erfüllbar ist, wenn solch eine akzeptierende Konfigurationsfolge existiert.

Die Variablen der Formel φ

- $Q(t, q)$ für $t \in \{0, \dots, p(n)\}$ und $q \in Q$
- $H(t, j)$ für $t, j \in \{0, \dots, p(n)\}$
- $B(t, j, a)$ für $t, j \in \{0, \dots, p(n)\}$ und $a \in \Gamma$
- Interpretation der Variablen:
  - Die Belegung $Q(t, q) = 1$ besagt, dass sich die Berechnung zum Zeitpunkt t im Zustand q befindet
  - Die Belegung $H(t, j) = 1$ steht dafür, dass sich der Kopf zum Zeitpunkt t an Bandposition j befindet
  - Die Belegung $B(t, j, a) = 1$ bedeutet, dass zum Zeitpunkt t an Bandposition j das Zeichen a geschrieben steht



Wir werden die akzeptierende Konfigurationsfolge in drei Phasen in die Formel φ übersetzen:

- Arbeitsphase A: Für jeden Zeitpunkt t beschreiben die Variablen $Q(t,q), H(t,j), B(t,j,a)$ eine legale Konfiguration
- Arbeitsphase B: Die Konfiguration zum Zeitpunkt $t+1$ entsteht legal aus der Konfiguration zum Zeitpunkt t
- Arbeitsphase C: Startkonfiguration und Endkonfiguration sind legal

Arbeitsphase A:

Für jeden Zeitpunkt t konstruieren wir eine Teilformel φ_t von Formel φ, die nur dann erfüllt ist, wenn die Variablen $Q(t,q), H(t,j), B(t,j,a)$ eine legale Konfiguration K_t beschreiben.

- A1. Es gibt genau einen Zustand $q \in Q$ mit $Q(t,q) = 1$
- A2. Es gibt genau eine Bandposition $j \in \{0, \dots, p(n)\}$ mit $H(t,j) = 1$
- A3. Es gibt für jedes $j \in \{0, \dots, p(n)\}$ jeweils genau ein Zeichen $a \in \Gamma$ mit $B(t,j,a) = 1$

Boole'sches Werkzeug:

- Für eine beliebige Variablenmenge $\{y_1, \dots, y_k\}$ besagt die folgende Formel in CNF, dass genau eine der Variablen y_i den Wert 1 annimmt: $(y_1 \vee y_2 \vee \dots \vee y_k) \wedge \bigwedge_{i \neq j} (\overline{y_i} \vee \overline{y_j})$
- Anzahl der Literale in dieser Formel ist $O(k^2)$ und quadratisch in der Anzahl der Variablen.

Die drei erwünschten Eigenschaften A1/A2/A3 zum Zeitpunkt t (für legale Konfigurationen) können daher jeweils durch eine Formel φ_t mit polynomiell beschränkter Länge kodiert werden.
Phase A ist damit abgeschlossen.

Arbeitsphase B:

Für jeden Zeitpunkt t konstruieren wir eine Teilformel φ_t von Formel φ, die erzwingt, dass die Konfiguration K_t eine direkte Nachfolgekonfiguration von Konfiguration K_{t-1} ist.

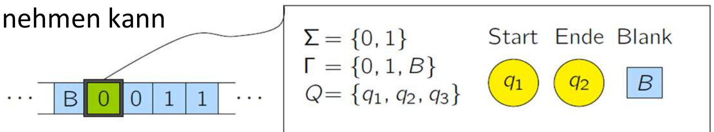
- B1. Der Bandinhalt der Konfiguration K_t stimmt an allen Positionen mit dem Bandinhalt der Konfiguration K_{t-1} überein, mit möglicher Ausnahme jener Position, an der der Kopf zum Zeitpunkt $t-1$ ist.
- B2. Zustand, Kopfposition und Bandinhalt an Kopfposition verändern sich im Einklang mit der Übergangsrelation δ.

Eigenschaft B1 (Bandinhalt von K_t stimmt mit Bandinhalt von K_{t-1} überein, ausgenommen Kopfposition) wird wie folgt kodiert: $\bigwedge_{i=0}^{p(n)} \bigwedge_{a \in \Gamma} B(t-1,i,a) \wedge \neg H(t-1,i) \Rightarrow B(t,i,a)$
Boole'sches Werkzeug:

- $x_1 \Rightarrow x_2$ äquivalent zu $\neg x_1 \vee x_2$; $\neg(x_1 \vee x_2)$ äquivalent zu $\neg x_1 \vee \neg x_2$ (De Morgan)
- $y_1 \wedge \neg y_2 \Rightarrow y_3$ äquivalent zu $\neg(y_1 \wedge \neg y_2) \vee y_3$ äquivalent zu $\neg y_1 \vee y_2 \vee y_3$

$\bigwedge_{i=0}^{p(n)} \bigwedge_{a \in \Gamma} (\neg B(t-1,i,a) \vee H(t-1,i) \vee B(t,i,a))$

Eigenschaft B2 (Zustand, Kopfposition und Bandinhalt an Kopfposition verändern sich gemäß Übergangsrelation δ) wird wie folgt kodiert:
Für alle $q \in Q$, für alle $j \in \{0, \dots, p(n)-1\}$ und für alle $a \in \Gamma$ verwenden wir die Teilformel
$Q(t-1,q) \wedge H(t-1,j) \wedge B(t-1,j,a) \Rightarrow \bigvee_{(q,a,q',a',\kappa) \in \delta} (Q(t,q') \wedge H(t,j+\kappa) \wedge B(t,j,a'))$
wobei κ die Werte $L=-1, N=0$ und $R=1$ annehmen kann



δ	0	1	B
q_1	$\{(q_1,B,R),(q_2,1,L)\}$	$\{(q_3,B,R)\}$	$\{(q_2,B,N)\}$
q_3	$\{(q_1,1,R)\}$	$\{(q_2,0,R)\}$	$\{(q_1,B,L)\}$

- Die Formel in Rot ist nicht CNF
- Die Formel in Rot besteht aus höchstens $3\Delta + 3$ Literalen ($\Delta =$ max. Verzweigungsgrad)
- Die Formel in Rot kann in äquivalente Formel in CNF mit höchstens $\leq 3^{3\Delta+3}(3\Delta+3)$ Literalen umgeformt werden

Phase B ist damit abgeschlossen.

$$
\begin{aligned}
Q(t-1,q_1) \wedge H(t-1,j) \wedge B(t-1,j,0) \quad &\Rightarrow \\
&(Q(t,q_1) \wedge H(t,j+1) \wedge B(t,j,B)) \\
\vee \quad &(Q(t,q_2) \wedge H(t,j-1) \wedge B(t,j,1))
\end{aligned}
$$

Arbeitsphase C:
Zum Schluss sorgen wir noch dafür, dass Startkonfiguration und Endkonfiguration korrekt beschrieben werden.

- C1. Startkonfiguration: $Q(0, q_0) \wedge H(0,0) \wedge \bigwedge_{i=0}^{n-1} B(0, i, x_i) \wedge \bigwedge_{i=n}^{p(n)} B(0, i, B)$
- C2. Endkonfiguration: $Q(p(n), q_{accept})$

Phase C ist damit abgeschlossen.

Zusammenfassung:

- Die Gesamtformel φ setzt sich aus allen Teilformeln zusammen, die wir für A1/A2/A3 und B1/B2 und C1/C2 konstruiert haben.
- Insgesamt sind das polyn. viele Klauseln, die jeweils aus polyn. vielen Literalen bestehen.
- Länge von φ daher polyn. beschränkt in n, und φ kann aus x in polyn. Zeit berechnet werden
- Die Formel φ genau dann erfüllbar, wenn es eine akzeptierende Konfigurationsfolge der Länge $p(n)$ für M auf x gibt.

Kochrezept für NP-Vollständigkeitsbeweise

Satz: Wenn L^* NP-schwer ist, dann gilt: $L^* \leq_p L \Rightarrow L$ ist NP-schwer
Beweis: Für alle $L' \in$ NP gilt $L' \leq_p L^*$ und $L^* \leq_p L$.
Die Transivität von $\leq_p$ impliziert $L' \leq_p L$ für alle $L' \in$ NP

Kochrezept:

1. Man zeige $L \in$ NP
2. Man wähle eine NP-vollständige Sprache L^*
3. (Reduktionsabbildung): Man konstruiere eine Funktion f, die Instanzen von L^* auf Instanzen von L abbildet
4. (Polynomielle Zeit): Man zeige, dass f in polynomieller Zeit berechnet werden kann
5. (Korrektheit): Man beweise, dass f tatsächlich eine Reduktion ist.
   Für $x \in \{0,1\}^*$ gilt $x \in L^*$ genau dann, wenn $f(x) \in L$.

NP-Vollständigkeit von 3-SAT

- Eine k-Klausel ist eine Klausel, die aus exakt k Literalen besteht
- Eine CNF-Formel φ ist eine k-CNF, wenn sie aus k-Klauseln besteht
- Beispiel einer Formel in 3-CNF: $\varphi = (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3)$ (Je 3 Literale)

Problem: 3-SAT
Eingabe: Boole'sche Formel φ in 3-CNF
Frage: Besitzt φ eine erfüllende Belegung?
3-SAT ist ein Spezialfall von SAT und liegt deshalb wie SAT in NP.
Satz: SAT $\leq_p$ 3-SAT
Beweis:

- Gegeben sei eine beliebige Formel φ in CNF (Instanz von SAT)
- Wir werden eine zur Formel φ äquivalente Formel φ' in 3-CNF konstruieren:
  φ ist erfüllbar $\Leftrightarrow$ φ' ist erfüllbar
- Aus einer 1-Klausel oder 2-Klausel machen wir eine äquivalente 3-Klausel, indem wir ein oder zwei Literale duplizieren
- 3-Klauseln bleiben 3-Klauseln
- Auf k-Klauseln mit $k \geq 4$ wenden wir wiederholt die folgende Klauseltransformation an:
  Die Klausel $c = (l_1 + l_2 + \cdots + l_k)$ wird ersetzt durch die beiden neuen Klauseln
  $(l_1 + \cdots + l_{k-2} + h)$ und $(\bar{h} + l_{k-1} + l_k)$. h bezeichnet eine neu eingeführte Hilfsvariable.

Beispiel: Klauseltransformation für eine 5-Klausel

- Wir beginnen mit der 5-Klausel $(x_1 + \bar{x}_2 + x_3 + x_4 + \bar{x}_5)$
- Im ersten Transformationsschritt wird daraus eine 4-Klausel und eine 3-Klausel gemacht: $(x_1 + \bar{x}_2 + x_3 + h_1)(\bar{h}_1 + x_4 + \bar{x}_5)$
- Auf die 4-Klausel wird die Transformation dann erneut angewendet. Dadurch entsteht nun $(x_1 + \bar{x}_2 + h_2)(\bar{h}_2 + x_3 + h_1)(\bar{h}_1 + x_4 + \bar{x}_5)$, und es sind nur noch 3-Klauseln vorhanden.

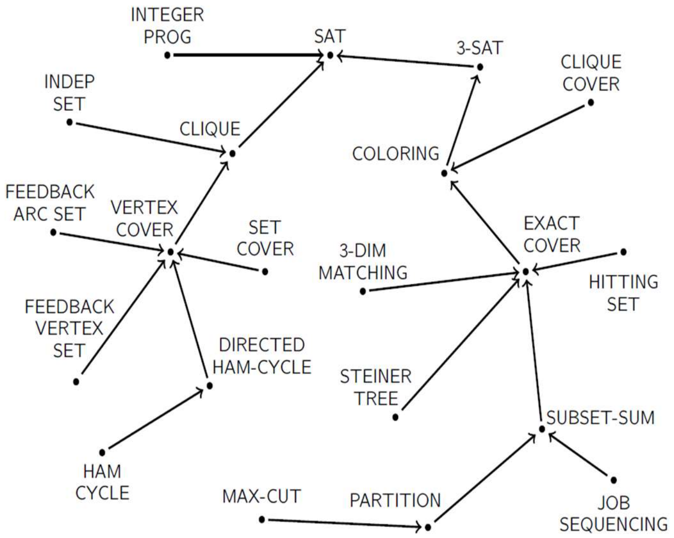
Korrektheit:

- Alte Klausel: $c = (l_1 + l_2 + \cdots + l_k)$
- Neue Klausel $c' = (l_1 + l_2 + \cdots + l_{k-2} + h)$ und $c'' = (\bar{h} + l_{k-1} + l_k)$
- (1) Wenn eine Wahrheitsbelegung c' und c'' erfüllt, so erfüllt sie automatisch auch c:
  - Wenn $h = 0$, dann ist $l_1 + \cdots + l_{k-2}$ wahr
  - Wenn $h = 1$, dann ist $l_{k-1} + l_k$ wahr
- (2) Wenn eine Wahrheitsbelegung c erfüllt, so kann sie auf h erweitert werden, sodass die beiden Klauseln c' und c'' erfüllt sind:
  - Die Wahrheitsbelegung macht mindesten ein Literal aus c wahr
  - Falls $l_1 + \cdots + l_{k-2}$ wahr ist, setzen wir $h = 0$
  - Falls $l_{k-1} + l_k$ wahr ist, so setzen wir $h = 1$

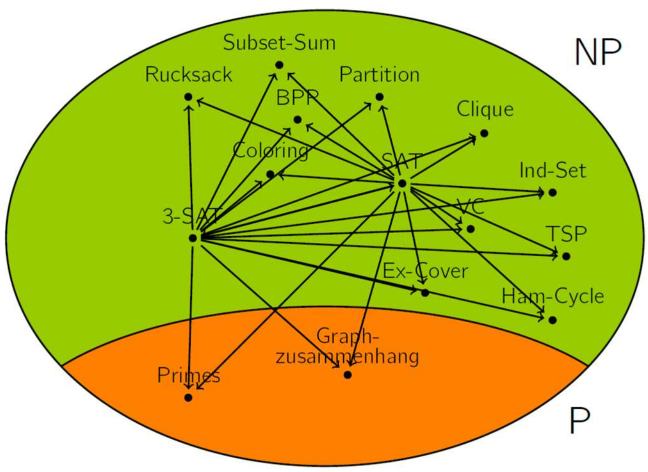
- Durch Anwendung der Klauseltransformation entstehen aus einer k-Klausel eine $(k-1)$-Klausel und eine 3-Klausel.
- Nach $k - 3$ Iterationen sind dann aus einer einzelnen alten k-Klausel genau $k - 2$ neue 3-Klauseln entstanden.
- Aus $k \geq 4$ alten Literalen entstehen also $3k - 6$ neue Literale.
- Diese Transformation wird solange auf die Formel φ angewandt, bis die Formel nur noch 3-Klauseln enthält.
- Wenn φ aus p Literalen besteht, so besteht φ' aus höchstens $3p$ Literalen.
- Die Laufzeit der Reduktion ist daher polynomiell beschränkt.

Satz: 3-SAT ist NP-vollständig.

Karp's Liste mit 21 Problemen



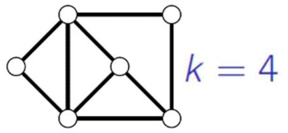
SAT	3-SAT
INTEGER PROGRAMMING	COLORING
CLIQUE	CLIQUE COVER
INDEP-SET	EXACT COVER
VERTEX COVER	3-DIM MATCHING
SET COVER	STEINER TREE
FEEDBACK ARC SET	HITTING SET
FEEDBACK VERTEX SET	SUBSET-SUM
DIR HAM-CYCLE	JOB SEQUENCING
UND HAM-CYCLE	PARTITION
	MAX-CUT



Einige NP-vollständige Graphenprobleme

NP-Vollständigkeit von CLIQUE

Problem: CLIQUE
Eingabe: Ein ungerichteter Graph $G = (V.E)$; eine Zahl k
Frage: Enthält G eine Clique mit $\geq k$ Knoten?
Satz: CLIQUE ist NP-vollständig

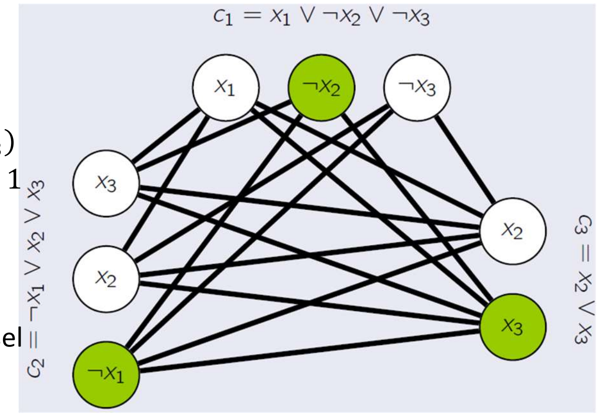


Nach Kochrezept:

1. Wir wissen bereits, dass CLIQUE in NP liegt.

In die richtige Richtung reduzieren!!!

2. Wir wählen die NP-vollständige Sprache $L^* =$ SAT und wir werden zeigen: SAT $\leq_p$ CLIQUE
3. Reduktionsabbildung:
   Wir konstruieren eine CNF-Formel φ in einen Graphen $G = (V, E)$ und eine Zahl $k \in \mathbb{N}$ transformiert, sodass gilt: φ ist erfüllbar $\Leftrightarrow$ G besitzt k-Clique
   - Es seien $c_1, \ldots, c_m$ die Klauseln der Formel φ. Es sei k_i die Anzahl an Literalen in Klausel c_i. Es seien $l_{i,1}, \ldots, l_{i,k}$ die Literale in Klausel c_i.
   - Für jedes Literal in jeder Klausel erzeugen wir einen entsprechenden Knoten: $V = \{l_{i,j} | 1 \leq i \leq m, 1 \leq j \leq k_i\}$
   - Zwei Knoten werden mit einer Kante verbunden, wenn sie aus verschiedenen Klauseln stammen und wenn ihre Literale nicht Negationen voneinander sind.
   - Wir setzten $k = m$
4. Polynomielle Zeit:
   Die Funktion f ist in Polynomialzeit berechenbar.
   Beispiel: $\varphi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee x_3)$
   Erfüllende Belegung des Beispiel rechts: $x_1 = 0, x_2 = 0, x_3 = 1$



   Korrektheit:
   Lemma A: Formel φ erfüllbar $\Rightarrow$ G hat m-Clique
   - Betrachte beliebige erfüllende Belegung von φ
   - Bilde Menge U mit einem erfüllten Literal von jeder Klausel
   - Behauptung: U bildet m-Clique
   - Begründung:
     - Laut Definition ist $|U| = m$
     - Es seien l und l' zwei verschieden Literale aus U
     - Nach Konstruktion kommen l und l' aus verschiedenen Klauseln
     - Da l und l' erfüllt sind, sind sie nicht Negationen voneinander
     - Also gibt es eine Kante zwischen l und l'

   Lemma B: G hat m-Clique $\Rightarrow$ Formel φ erfüllbar
   - Betrachte m-Clique U in G
   - Dann gehören die Literale in U zu lauter verschiedenen Klauseln
   - U enthält somit genau ein Literal pro Klausel
   - Kein Literal tritt sowohl positiv als auch negativ auf
   - Ergo: Alle diese Literale können gleichzeitig erfüllt werden
   - Also ist φ erfüllbar
5. Korrektheit:
   f ist Reduktion: $x \in L^* \Leftrightarrow f(x) \in L$

$\varphi \in$ SAT $\Leftrightarrow f(\varphi) = \langle G; m \rangle \in$ CLIQUE

NP-Vollständigkeit von INDEP-SET und Vertex Cover

Problem: Independent Set (INDEP-SET)
Eingabe: Ein ungerichteter Graph $G' = (V', E')$; eine Zahl k'
Frage: Enthält G' eine unabhängige Menge mit $\geq k'$ Knoten?
Satz: INDEP-SET ist NP-vollständig

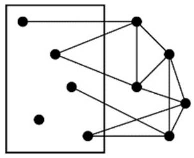
- Wir zeigen CLIQUE $\leq_p$ INDEP-SET
- Setze $V' = V$ und $E' = V \times V - E$ und $k' = k$

Problem: Vertex Cover (VC)
Eingabe: Ein ungerichteter Graph $G'' = (V'', E'')$; eine Zahl k''
Frage: Enthält G'' ein Vertex Cover mit $\geq k''$ Knoten?
Satz: Vertex Cover ist NP-vollständig

- Wir zeigen INDEP-SET $\leq_p$ Vertex Cover
- Setze $V'' = V'$ und $E'' = E'$ und $k'' = |V'| - k'$

Beobachtung: In einem ungerichteten Graphen $G = (V, E)$ gilt für alle $S \subseteq V$:

- S ist unabhängige Menge $\Leftrightarrow$ $V - S$ ist Vertex Cover
- S ist Vertex Cover $\Leftrightarrow$ $V - S$ ist unabhängige Menge

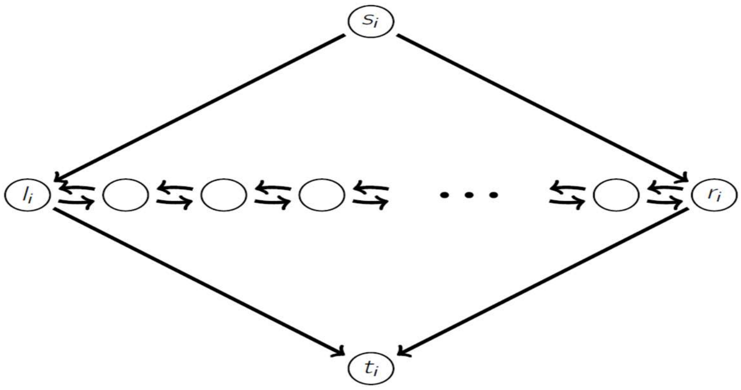


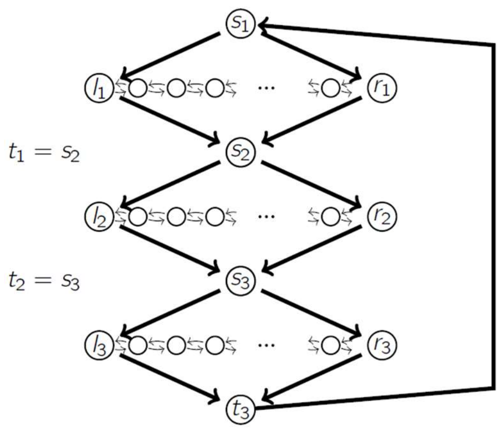
NP-Vollständigkeit von Ham-Cycle (gerichtet)

Problem: Gerichteter Hamiltonkreis (D-Ham-Cycle)
Eingabe: Ein gerichteter Graph $G = (V, A)$
Frage: Besitzt G einen gerichteten Hamiltonkreis?
Satz: D-Ham-Cycle ist NP-vollständig

Nach Kochrezept:

1. D-Ham-Cycle liegt in NP.
2. Wir wählen die NP-vollständige Sprache $L^* =$ SAT und wir werden zeigen: SAT $\leq_p$ D-Ham-Cycle
3. Reduktionsabbildung:
   Wir konstruieren eine Funktion f, die eine CNF-Formel φ in einen gerichteten Graphen $G = (V, A)$ transformiert, sodass gilt: φ ist erfüllbar $\Leftrightarrow$ G hat gerichteten Hamiltonkreis.
   Die CNF-Formel φ besteht aus Klausel $c_1, \dots, c_m$ mit Boole'schen Variablen $x_1, \dots, x_n$.
   Für jede Variable enthält der Graph G das Diamantengadget G_i:



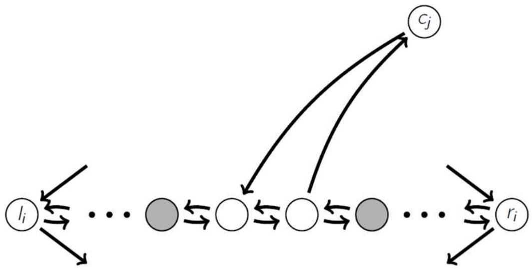


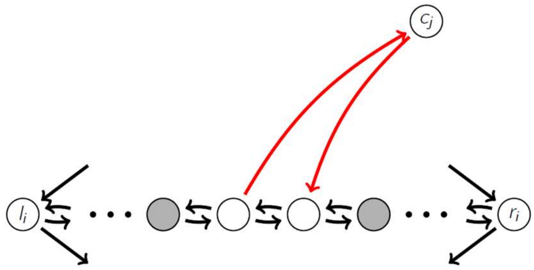
   Diese n Diamantengadgets werden miteinander verbunden, indem wir die Knoten t_i und s_{i+1} (für $1 \leq i \leq n-1$) sowie t_n und s_1 miteinander identifizieren (s. oben rechts)
   In dem resultierenden Graphen besucht jede Rundreise, die im Knoten s_1 startet, die Diamantengadgets in der Reihenfolge $G_1, \dots, G_n$. Die Rundreise hat dabei für jedes Gadget G_i die Freiheit, das Gadget
   - entweder von links nach rechts (also: von l_i bis r_i)
   - oder von rechts nach links (also: von r_i nach l_i) zu durchlaufen.

Die LR Variante interpretieren wir als Variablenbelegung $x_i = 0$, die RL Variante als Variablenbelegung $x_i = 1$.
Jetzt fügen wir für jede Klausel c_i einen weiteren Knoten ein.

(a) Falls das Literal x_i in Klausel c_j enthalten ist, so verbinden wir Gadget G_i wie unten links mit dem Klauselknoten c_j

(b) Falls das Literal $\bar{x}_i$ in Klausel c_j enthalten ist, so verbinden wir Gadget G_i wie unten rechts mit dem Klauselknoten c_j

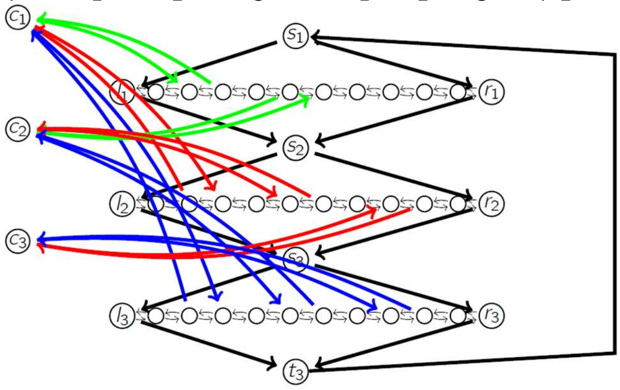




Frage: Ist es nach Hinzufügen der Klauselknoten möglich, dass eine Rundreise zwischen den Diamantengadgets hin- und herspringt, anstatt sie in der vorgesehenen Reihenfolge zu besuchen?

Antwort: Nein

$\varphi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee x_3)$



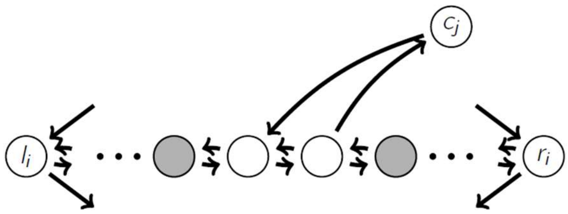
Korrektheit:

Lemma A: G hat einen gerichteten Hamiltonkreis $\Rightarrow$ φ ist erfüllbar

- Wenn ein Klauselknoten c_j aus einem Gadget G_i heraus von links nach rechts durchlaufen wird, so muss nach unserer Konstruktion die Klausel c_j das Literal $\bar{x}_i$ enthalten.
- Also wird diese Klausel durch die mit der Laufrichtung von links nach rechts assoziierter Belegung $x_i = 0$ erfüllt.
- Wenn ein Klauselknoten c_j aus einem Gadget G_i heraus von rechts nach links durchlaufen wird, so muss nach unserer Konstruktion die Klausel c_j das Literal x_i enthalten.
- Also wird diese Klausel c_j durch die mit der Laufrichtung von rechts nach links assoziierter Belegung $x_i = 1$ erfüllt.
- Also erfüllt die mit der Rundreise assoziierte Wahrheitsbelegung der Variablen die Formel φ.

Lemma B: φ ist erfüllbar $\Rightarrow$ G hat einen gerichteten Hamiltonkreis

- Eine erfüllende Wahrheitsbelegung der Variablen legt für jedes Diamantengadget $G_1, \dots, G_n$ fest, ob es von rechts nach links oder von links nach rechts durchlaufen wird.
- Klauselknoten c_j können wir in die Rundreise einbauen, indem wir eine Variable x_i auswählen, die c_j erfüllt, und c_j durch einen kleinen Abstecher vom Diamantengadget G_i aus besuchen.



- Wenn c_j für $x_i = 1$ erfüllt ist, so ist x_i positiv in c_j enthalten. Ein Besuch von c_j beim Durchlaufen des Diamantengadgets G_i von rechts nach links ist möglich.
- Wenn c_j für $x_i = 0$ erfüllt ist, so ist x_i in negierter Form in c_j enthalten. Ein Besuch von c_j beim Durchlaufen des Diamantengadgets G_i von links nach rechts ist möglich.
- Also können alle Klauselknoten in die Rundreise eingebunden werden.

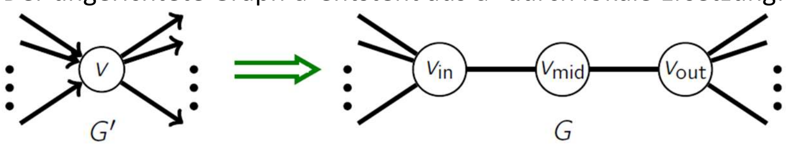
4. Polynomielle Zeit:
   Die Funktion f ist polynomiell berechenbar.
   - Die Konstruktion verwendet n Diamantengadgets mit je $O(m)$ Knoten
   - Die Konstruktion verwendet m Klauselknoten
6. Korrektheit:
   f ist Reduktion: $x \in L^* \Leftrightarrow f(x) \in L$

$\varphi \in$ SAT $\Leftrightarrow f(\varphi) = \langle G \rangle \in$ D-Ham-Cycle

NP-Vollständigkeit von Ham-Cycle (ungerichtet)

Problem: Hamiltonkreis
Eingabe: Ein ungerichteter Graph $G = (V, E)$
Frage: Besitzt G einen Hamiltonkreis?
Satz: Ham-Cycle ist NP-vollständig
Beweis:

- Wir zeigen D-Ham-Cycle $\leq_p$ Ham-Cycle
- Es sei $G' = (V', A')$ eine Instanz von D-Ham-Cycle
- Wir konstruieren in polyn. Zeit einen ungerichteten Graphen $G = (V, E)$, sodass gilt:
  $G' \in$ D-Ham-Cycle $\Leftrightarrow G \in$ Ham-Cycle
- Es sei $G' = (V', A')$ eine Instanz von D-Ham-Cycle
- Der ungerichtete Graph G entsteht aus G' durch lokale Ersetzung:

  Interpretation
  - v_{in} ist der Eingangsknoten für v_{mid}
  - v_{out} ist der Ausgangsknoten für v_{mid}
- Korrektheit:
  (A) Jeder Hamiltonkreis in G' kann in einen Hamiltonkreis in G transformiert werden
  (B) Wie sieht es mit der Umkehrrichtung aus?
  - Jeder Hamiltonkreis in G besucht den Knoten v_{mid}zwischen den beiden Knoten v_{in} und v_{out}
  - Entweder: $v_{in} - v_{mid} - v_{out}$ ODER: $v_{out} - v_{mid} - v_{in}$
  - Von v_{out} aus kann man nur Knoten vom Typ u_{in} erreichen (und dazu muss der gerichtete Graph die entsprechende gerichtete Kante von v nach u enthalten)
  - Daher kann jeder Hamiltonkreis in G in einen gerichteten Hamiltonkreis G' übersetzt werden.

NP-Vollständigkeit des TSP

Problem: Travelling Salesman (TSP)
Eingabe: Städte $1, \dots, n$; Distanzen $d(i, j)$; eine Zahl γ
Frage: Gibt es eine Rundreise (TSP-Tour) mit Länge höchstens γ?

Zwei Spezialfälle:

Problem: Δ-TSP
Eingabe: Städte $1, \dots, n$; symmetrische Distanzen $d(i,j)$ mit Dreiecksgleichung $d(i,j) \le d(i,k) + d(k,j)$; eine Zahl γ
Frage: Gibt es eine Rundreise (TSP-Tour) mit Länge höchstens γ?

Problem: $\{1,2\}$-TSP
Eingabe: Städte $1, \dots, n$; symmetrische Distanzen $d(i,j) \in \{1,2\}$; eine Zahl γ
Frage: Gibt es eine Rundreise (TSP-Tour) mit Länge höchstens γ?

Satz: TSP und Δ-TSP und $\{1,2\}$-TSP sind NP-schwer.

- Es genügt zu zeigen, dass $\{1,2\}$-TSP NP-schwer ist.
- Wir zeigen: Ham-Cycle $\le_p$ $\{1,2\}$-TSP
- Aus einem ungerichteten Graphen $G = (V, E)$ für Ham-Cycle konstruieren wir eine TSP Instanz.
- Jeder Knoten $v \in V$ wird zu einer Stadt
- Der Abstand zwischen Stadt u und Stadt v beträgt $d(u,v) = \begin{cases} 1\ falls\ \{u,v\} \in E \\ 2\ falls \{u,v\} \notin E \end{cases}$
- Wir setzen $\gamma := |V|$
- Der Graph G hat genau dann einen Hamiltonkreis, wenn die konstruierte TSP Instanz eine Tour mit Länge $\le \gamma$ hat.

NP-vollständige Zahlprobleme

NP-Vollständigkeit von SUBSET-SUM

Problem: SUBSET-SUM
Eingabe: Positive ganze Zahlen $a_1, \dots, a_n$; eine ganze Zahl b
Frage: Existiert eine Indexmenge $I \subseteq \{1, \dots, n\}$ mit $\Sigma_{i \in I}\, a_i = b$?
Beispiel: Zahlen 1,2,4,8,16,32,64,128,256,512,1024 und $b = 999$
Satz: SUBSET-SUM ist NP-vollständig.
Beweis:

- SUBSET-SUM liegt in NP
- Wir zeigen 3-SAT $\le_p$ SUBSET-SUM
  - Die Boole'sche Formel φ in 3-CNF sei eine Instanz von 3-SAT
  - Die Formel hat Klauseln $c_1, \dots, c_m$ mit den Variablen $x_1, \dots, x_n$
- Reduktion:
  - (In der Reduktion arbeiten wir mit Dezimalzahlen mit jeweils $n + m$ Ziffern. Die k-te Ziffer einer Zahl z bezeichnen wir dabei mit $z(k)$
  - Wir definieren:
    - $S^+(i) = \{j \in \{1, \dots, m\} | \text{Klausel } c_j \text{ enthält Literal } x_i\}$
    - $S^-(i) = \{j \in \{1, \dots, m\} | \text{Klausel } c_j \text{ enthält Literal } \bar{x}_i\}$
  - Für jede Boolesche Variable x_i mit $1 \le i \le n$ erzeugen wir zwei entsprechende Var-Zahlen a_i^+ und a_i^- mit den folgenden Ziffern:
    - $a_i^+(i) = 1$ und für alle $j \in S^+(i)$: $a_i^+(n + j) = 1$
    - $a_i^-(i) = 1$ und für alle $j \in S^-(i)$: $a_i^-(n + j) = 1$
    - Alle anderen Ziffern in diesen Dezimaldarstellungen sind 0

  Beispiel: $(x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \bar{x}_3 \vee \bar{x}_4)$
  - Die folgenden Var-Zahlen werden erzeugt:
  - $a_1^+ = 100010; a_1^- = 100000; a_2^+ = 010011; a_2^- = 010000;$
    $a_3^+ = 001010; a_3^- = 001001; a_4^+ = 000100; a_4^- = 000101;$

- Wir definieren für jede Klausel $c_j$zwei entsprechende Dummy-Zahlen d_j und d_j'
- Dummy-Zahlen haben nur an der Zifferposition $n + j$ eine Ziffer 1; alle anderen Ziffern 0
- Wir betrachten wieder die Formel $(x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \bar{x}_3 \vee \bar{x}_4)$
- Die Dummy-Zahlen für die beiden Klauseln lauten dann:
- $d_1 = 000010; d_1' = 000010; d_2 = 000001; d_2' = 000001$
- Die Zielsumme b definieren wir folgendermaßen:
- $b(k) = 1$ für $1 \leq k \leq n; b(k) = 3$ für $n + 1 \leq k \leq n + m$
- Die Zielsumme lautet dann $b = 111133$

Beweis:

- Für jede Dezimalstelle $i \in \{1, \dots, n\}$ gilt: nur zwei der Var-Zahlen und Dummy-Zahlen haben an dieser Stelle die Ziffer 1; alle anderen Zahlen die Ziffer 0
- Für jede Dezimalstelle $i \in \{n + 1, \dots, n + m\}$ gilt: Nur fünf der Var-Zahlen und Dummy-Zahlen haben an dieser Stelle die Ziffer 1; alle anderen Zahlen die Ziffer 0
- Beobachtung: Keine Carry-Overs:
  - Wird eine beliebige Menge von Var-Zahlen und Dummy-Zahlen addiert, so tritt von keiner Dezimalstelle zur nächsten ein Additionsübertrag auf.
- Laufzeit der Reduktion:
  - Die SAT Instanz φ besteht aus n Variablen und m Klauseln. Eingabelänge ist $\geq m + n$
  - Die konstruierte SUBSET-SUM Instanz besteht aus $2n + 2m + 2$ Dezimalzahlen mit je $m + n$ Dezimalstellen
  - Die Reduktion wird in polynomieller Zeit $O((m + n)^2)$ druchgeführt
- Korrektheit der Reduktion:
- Lemma A: φ ist erfüllbar $\Rightarrow$ SUBSET-SUM Instanz ist lösbar
  - Es gibt eine erfüllende Belegung x^* für die Formel φ
  - Falls $x_i^* = 1$, so wählen wir a_i^+ aus; andernfalls wählen wir a_i^-
  - Die Summe der ausgewählten Var-Zahlen bezeichnen wir mit A
  - Da für jedes $j \in \{1, \dots, n\}$ entweder a_i^+ oder a_i^- ausgewählt wurde, gilt $A(i) = 1$
  - Außerdem gilt $A(n + j) \in \{1,2,3\}$ für $1 \leq j \leq m$, weil in jeder Klausel ein oder zwei oder drei Literale erfüllt sind
  - Falls $A(n + j) \in \{1,2\}$, so wählen wir zusätzlich d_j und/oder d_j' aus, um die Ziffer 3 an Ziffernposition $n + j$ der Summe zu erhalten.
  - Also gibt es eine Teilmenge mit der gewünschten Zielsumme b.
- Lemma B: SUBSET-SUM Instanz ist lösbar $\Rightarrow$ φ ist erfüllbar
  - Es gibt eine Teilmenge K_A der Var-Zahlen (mit Summe A) und eine Teilmenge K_D der Dummy-Zahlen (mit Summe H), die sich zur Zielsumme b aufaddieren; also: $A + H = b$
  - Die Menge K_A enthält für jedes $i \in \{1, \dots, n\}$ genau eine der beiden Var-Zahlen a_i^+ und a_i^-; andernfalls wäre $A(i) \neq 1$
  - Wir setzen $x_i = 1$ falls $a_i^+ \in K_A$, und andernfalls $x_i = 0$
  - Es gilt $A(n + j) \geq 1$ für $1 \leq j \leq m$.
    Ansonsten wäre $A(n + j) + H(n + j) \leq A(n + j) + 2 < 3$.
  - Dadurch ist sichergestellt, dass in jeder Klausel mindestens eines der Literale den Wert 1 hat.
  - Die Formel φ ist also erfüllbar.

NP-Vollständigkeit von PARTITION

Problem: PARTITION

Eingabe: Positive ganze Zahlen $a_1', \dots, a_n'$; mit $\Sigma_{i \in I}^n a_i' = 2A'$

Frage: Existiert eine Indexmenge $I \subseteq \{1, \dots, n\}$ mit $\Sigma_{i \in I}\, a_i' = A'$?

Partition ist ein Spezialfall von SUBSET-SUM mit $b := (\sum a_i) \,/\, 2$

Satz: PARTITION ist NP-Vollständig.

Beweis:

- PARTITION liegt in NP
- Wir zeigen SUBSET-SUM $\leq_p$ PARTITION
- Reduktion:
  - Es sei $a_1, \dots, a_n \in \mathbb{N}$ und $b \in \mathbb{N}$ eine beliebige Instanz von SUBSET-SUM
  - Es sei $S := \Sigma_{i=1}^n a_i$, und o.B.d.A. gilt $S \geq b$
  - Wir bilden diese SUBSET-SUM Instanz auf eine PARTITION Instanz ab, die aus den folgenden $n + 2$ Zahlen $a_1', \dots, a_{n+2}'$ besteht:
    - $a_i' = a_i$ für $1 \leq i \leq n$
    - $a_{n+1}' = 2S - b$ und $a_{n+2}' = S + b$
  - Die Summe dieser $n + 2$ Zahlen beträgt $\Sigma_{i=1}^{n+2} a_i' = 4S$
  - Daher gilt $A' := 2S$ für die PARTITION Instanz
  - Die Reduktion wird in polynomieller Zeit durchgeführt.
  - Korrektheit:
  - Lemma A: SUBSET-SUM Instanz lösbar $\Rightarrow$ PARTITION Instanz lösbar
    - Wenn es in der SUBSET-SUM Instanz eine Teilmenge der Zahlen $a_1, \dots, a_n$ mit der Summe b gibt, so haben die entsprechenden Zahlen $a_1', \dots, a_n'$ in der PARTITION Instanz ebenfalls die Summe b.
    - Wir fügen die Zahl $a_{n+1}' = 2S - b$ zu dieser Teilmenge dazu und erhalten eine Teilmenge mit der gewünschten Zielsumme $A' = 2S$
  - Lemma B: PARTITION Instanz lösbar $\Rightarrow$ SUBSET-SUM Instanz lösbar
    - In der Lösung der PARTITION Instanz sind die beiden Zahlen $a_{n+1}' = 2S - b$ und $a_{n+2}' = S + b$ nicht in derselben Teilmenge, da $a_{n+1}' + a_{n+2}' = 3S > 2S = A'$ gilt
    - Eine der Teilmengen besteht aus $a_{n+1}' = 2S - b$ und eine der Teilmengen der Zahlen $a_1', \dots, a_n'$ mit der Gesamtsumme $A' = 2S$
    - Die entsprechenden Zahlen in der SUBSET-SUM Instanz haben dann die Summe b

NP-Vollständigkeit von Bin Packing und Rucksack

Problem: Bin Packing (BPP)

Eingabe: Zahlen B und $w_1, \dots, w_n \in \{1, \dots, b\}$; eine Schranke γ

Frage: Können Objekte mit den gegebenen Größen $w_1, \dots, w_n$ in γ Kisten der Größe B gepackt werden?

Satz: Bin Packing ist NP-Vollständig

Beweis:

- Wir zeigen PARTITION $\leq_p$ Bin Packing
- Wir setzen $\gamma = 2$, und $w_i = a_i'$ für $1 \leq i \leq n$, und $B = A'$

Problem: Rucksack / Knapsack (KP)

Eingabe: Natürliche Zahlen $w_1, \dots, w_n, p_1, \dots, p_n, B, \gamma$

Frage: Existiert eine Teilmenge der Objekte mit Gesamtgewicht höchstens B und Gesamtprofit mindestens γ?

Satz: Rucksack ist NP-Vollständig

Beweis:

- Wir zeigen SUBSET-SUM $\leq_p$ Rucksack
- Wir setzen $w_i = a_i$ und $p_i = a_i$ für $1 \leq i \leq n$, und $B = \gamma = b$

Pseudo-polynomielle Zeit und Starke NP-schwere Probleme

- Es sei X ein algorithmisches Problem
- Die Laufzeit eines Algorithmus A für Problem X messen wir in der Kodierungslänge der Instanzen I von X
- Die Kodierungslänge $|I|$ ist die Anzahl der Symbole in einer „vernünftigen“ Beschreibung der Instanz I
- Kleine (polynomiell große) Änderungen in derartigen Beschreibungen sind für unsere Definitionen / Sätze / Beweise / Resultate irrelevant
- Beispiel: ungerichtete Graphen
  - Vernünftige Beschreibungen von ungerichteten Graphen $G = (V, E)$ sind
    - Adjazenzlisten mit Länge $l_1(G) = O(|E| \log |V|)$
    - Adjazenzmatrizen mit Länge $l_2(G) = O(|V|^2)$
  - Es gilt:
    - $l_1(G)$ ist polynomiell beschränkt in $l_2(G)$
    - $l_2(G)$ ist polynomiell beschränkt in $l_1(G)$
- Beispiel: natürliche Zahlen
- Vernünftige Beschreibungen von natürlichen Zahlen n sind
  - Dezimaldarstellung mit Länge $\approx log_{10} n$
  - Binärdarstellung mit Länge $\approx log_2 n$
  - Oktaldarstellung mit Länge $\approx log_8 n$
  - Hexadezimaldarstellung mit Länge $\approx log_{16} n$
- Für alle reellen Zahlen $a, b > 1$ gilt: $\log_a n = \log_a b * \log_b n$
- Die verschiedenen Kodierungslängen unterscheiden sich daher nur um einen konstanten Faktor.
- Anmerkung: Die Zahl n stellt den Wert n mit Kodierungslänge $O(\log n)$ dar.
  Der Wert hängt also exponentiell von der Kodierungslänge ab.

Definition: $Number$

Für eine Instanz I eines Entscheidungsproblems bezeichnen wir mit $Number(I)$ den Wert der größten I vorkommenden Zahl.

Beispiel:

- Für eine TSP Instanz I ist $Number(I)$ der Wert der größten Städteinstanz $max_{i,j} d(i,j)$ oder der Wert γ
- Für eine SUBSET-SUM Instanz I ist $Number(I)$ das Maximum der Zahlen $a_1, \dots, a_n$ und b
- Für eine SAT Instanz I ist $Number(I)$ das Maximum der Zahlen n und m. (Ergo: $Number(I) \leq |I|$.)
- Der Parameter $Number(I)$ ist nur für Probleme relevant, in denen Distanzen, Kosten, Gewichte, Längen, Profite, Zeitintervalle, Abstände, etc. eine Rolle spielen.

Definition: Pseudo-polynomielle Zeit

Ein Algorithmus A löst ein Problem X in pseudo-polynomieller Zeit, falls die Laufzeit von A auf Instanzen I von X polynomiell in $|I|$ und $Number(I)$ beschränkt ist.

Satz: Die Probleme SUBSET-SUM, PARTITION und Rucksack sind pseudo-polynomiell lösbar.

Problem: SUBSET-SUM
Eingabe: Positive ganze Zahlen $a_1, \dots, a_n$; eine ganze Zahl b
Frage: Existiert eine Indexmenge $I \subseteq \{1, \dots, n\}$ mit $\Sigma_{i \in I}\, a_i = b$?
Satz: SUBSET-SUM ist pseudo-polynomieller Zeit $O(n * b)$ lösbar.
Beweis:

- Dynamische Programmierung: Für $k = 0, \dots, n$ und $c = 0, \dots, b$ setzen wir $F[k, c] = TRUE$ genau dann, wenn es eine Indexmenge $I \subseteq \{1, \dots, k\}$ mit $\Sigma_{i \in I} a_i = c$ gibt.
- $F[0, c] == (c == 0)$ für $c = 0, \dots, b$
  $F[k, c] = F[k-1, c-a_k] \vee F[k-1, c]$
- Schlussendlich findet man die Antwort in $F[n, b]$

Definition: Stark NP-schwer
Ein Entscheidungsproblem X ist stark NP-schwer, wenn es ein Polynom $q: \mathbb{N} \to \mathbb{N}$ gibt, sodass die Restriktion X_q von X auf Instanzen I mit $Number(I) \leq q(|I|)$ NP-schwer ist.
Also: Das Problem X ist sogar dann NP-schwer, wenn alle Zahlenwerte in der Instanz I nur polynomiell groß (gemessen in $|I|$) sind.
Satz: Es sei X ein stark NP-schweres Entscheidungsproblem. Falls X pseudo-polynomiell lösbar ist, so gilt P=NP.
Also: Pseudo-polynomiell und stark NP-schwer schließen einander aus.
Beweis:

- X ist stark NP-schwer
- Ergo gibt es ein Polynom $q: \mathbb{N} \to \mathbb{N}$, für das Die Restriktion X_q von X auf Instanzen I mit $Number(I) \leq q(|I|)$ NP-schwer ist
- Ein pseudo-polynomieller Algorithmus A auf X hat Laufzeit polynomiell beschränkt in $|I|$ und $Number(I)$
- Wendet man Algorithmus A auf X_q an, so ist die Laufzeit polynomiell beschränkt in $|I|$ und $q(|I|)$, und daher polynomiell beschränkt in $|I|$
- (X_q NP-schwer) und (X_q polynomiell lösbar) $\Rightarrow$ P=NP

Problem: THREE-PARTITION
Eingabe: Positive ganze Zahlen $a_1, \dots, a_n, b_1, \dots, b_n$ und $c_1, \dots, c_n$ mit $\Sigma_{i=1}^{n}(a_i + b_i + c_i) = nS$
Frage: Gibt es zwei Permutationen α, β von $1, \dots, n$, sodass $a_{\alpha(i)} + b_{\beta(i)} + c_i = S$ für $1 \leq i \leq n$ gilt?
Satz (ohne Beweis): THREE-PARTITION ist stark NP-schwer.

Jenseits von P und NP

Die Komplexitätsklasse coNP

Ein Entscheidungsproblem $X \subseteq \Sigma^*$ liegt in **coNP**, wenn für jedes Wort $x \notin X$ ein polynomiell langes Zertifikat y existiert, das (zusammen mit x) in polynomieller Zeit verifiziert werden kann.
Intuition:

- Wenn X in NP, dann gibt es für JA-Instanzen $x \in X$ kurze und einfach überprüfbare Beweise
- Wenn X in coNP, dann gibt es für NEIN-Instanzen $x \notin X$ kurze und einfach überprüfbare Beweise

Beispiele:

- Beispiel 1:
  - **Problem**: Non-Hamiltonkreis (Non-Ham-Cycle)
  - Eingabe: Ein ungerichteter Graph $G = (V, E)$
  - Frage: Besitzt G keinen Hamiltonkreis?

    Frage: Wie sieht das coNP-Zertifikat für Non-Ham-Cycle aus?

- Beispiel 2:
  - **Problem**: Unsatisfiability (UNSAT)
  - Eingabe: Boole'sche Formel φ in CNF über der Boole'schen Variablenmenge $X = \{x_1, \dots, x_n\}$
  - Frage: Existiert keine Wahrheitsbelegung von X, die φ erfüllt?
    Problem: TAUTOLOGY
  - Eingabe: Boole'sche Formel φ in DNF über der Boole'schen Variablenmenge $X = \{x_1, \dots, x_n\}$
  - Frage: Wird φ in allen Wahrheitsbelegung von X erfüllt?
  - **Frage**: Wie sehen coNP-Zertifikate für UNSAT und TAUTOLOGY aus?
- Beispiel 3: Lineare Programmierung
  - Ein primales Lineares Programm (P):
    $$\begin{aligned} &max\ \Sigma_{j=1}^{n} c_j x_j \\ &\text{s.t. } \Sigma_{j=1}^{n} a_{ij} x_j \le b_i \ \text{ für } i = 1, \dots, m \\ &\qquad\qquad x_j \ge 0 \end{aligned}$$
  - Das entsprechende duale Lineare Programm (D):
    $$\begin{aligned} &min\ \Sigma_{i=1}^{m} c_i y_i \\ &\text{s.t. } \Sigma_{i=1}^{m} a_{ij} y_i \le b_i \ \text{ für } j = 1, \dots, m \\ &\qquad\qquad y_i \ge 0 \end{aligned}$$
  - Satz (Starker Dualitätssatz): Wenn beide LPs zulässige Lösungen haben, so haben beide denselben optimalen Zielfunktionswert.
  - Primal $=$ "$max\ cx$ s.t. $Ax \le b$" Dual $=$ "$min\ by$ s.t. $yA \ge c$"
  - **Problem**: Lineare Programmierung (LP)
  - Eingabe: reelle $m \times n$ Matrix A; Vektoren $b \in \mathbb{R}^n$ und $c \in \mathbb{R}^n$; eine Schranke $\gamma \in \mathbb{R}^n$
  - Frage: Existiert ein Vektor $x \in \mathbb{R}^n$, der $Ax \le b$ und $x \ge 0$ erfüllt, und dessen Zielfunktionswert $cx \ge \gamma$ ist?
  - Beobachtung: LP liegt in NP.
  - NP-Zertifikat $=$ Vektor x fürs primale LP mit $cx \ge \gamma$
  - Beobachtung: LP liegt in coNP.
  - coNP-Zertifikat $=$ Vektor y fürs primale LP mit $bx \ge \gamma$
  - Zusammenfassung: LP liegt in NP ∩ coNP
  - Satz: LP liegt in P

coNP-Vollständigkeit

Ein Entscheidungsproblem X ist coNP-vollständig, wenn X ∈ coNP und alle Y ∈ coNP polynomiell auf X reduzierbar sind

Intuition:

- X ist NP-vollständig, wenn es zu den schwierigsten Problemen in NP gehört
- X ist coNP-vollständig, wenn es zu den schwierigsten Problemen in coNP gehört

Satz: Wenn das Entscheidungsproblem X NP-vollständig ist, so ist das komplementäre Problem $\bar{X}$ coNP-vollständig.

Komplementäres Problem: Ja-Instanzen von X werden zu Nein-Instanzen von $\bar{X}$ und Nein-Instanzen von X werden zu Ja-Instanzen von $\bar{X}$.

Satz: P ⊆ NP ∩ coNP

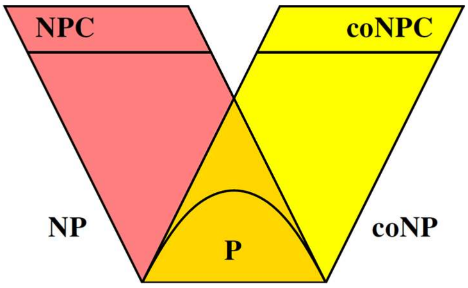
Beweis: P = coP

Satz: Wenn coNP ein NP-vollständiges Problem X enthält, dann NP $=$ coNP.
Beweis:

- $X \in$ NPC impliziert: $L \leq_p X$ für alle $L \in$ NP
- $X \in$ NPC impliziert: $\overline{L} \leq_p \overline{X}$ für alle $L \in$ NP
- $X \in$ NPC impliziert: $K \leq_p \overline{X}$ für alle $K \in$ coNP
- Ergo: Alle $K \in$ coNP sind auf $\overline{X} \in$ NP reduzierbar
- Ergo: Alle $K \in$ coNP liegen in NP.

Daraus erhalten wir das folgende Werkzeug:
„X NP-vollständig“ ist Evidenz für „$X \notin$ coNP“
„X coNP-vollständig“ ist Evidenz für „$X \notin$ NP“
Beispiele:

- Ham-Cycle ist NP-vollständig.
- Ham-Cycle hat gute Zertifikate für Ja-Instanzen.
- Ergo: Ham-Cycle hat (höchstwahrscheinlich) keine guten Zertifikate für Nein-Instanzen.
- SAT ist NP-vollständig.
- SAT hat gute Zertifikate für Ja-Instanzen.
- Ergo: SAT hat (höchstwahrscheinlich) keine guten Zertifikate für Nein-Instanzen.

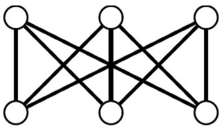


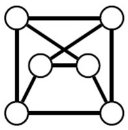
Viele Mathematiker denken, dass NP ∩ coNP = P gilt.

Zwischen P und NPC: NP-intermediate

Das Graphisomorphieproblem

Zwei Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ sind isomorph, wenn es eine Bijektion $f: V_1 \to V_2$ gibt, die Adjazenz und Nicht-Adjazenz erhält. Eine solche Bijektion heißt Isomorphismus.





Problem: GRAPH-ISOMORPHUS
Eingabe: Zwei ungerichtete Graphen G_1 und G_2
Frage: Gibt es einen Isomorphismus von G_1 nach G_2?
Satz: GRAPH-ISOMORPHUS liegt in NP.
Beweis: Verwende Isomorphismus als Zertifikat.

Folgende Fragen sind derzeit noch ungelöst:

- Liegt GRAPH-ISOMORPHUS in P?
- Ist GRAPH-ISOMORPHUS NP-vollständig?
- Liegt GRAPH-ISOMORPHUS in coNP?

Satz: GRAPH-ISOMORPHUS auf Graphen mit n Knoten kann in $2^{p(\log n)}$ Zeit gelöst werden (wobei p ein Polynom ist).

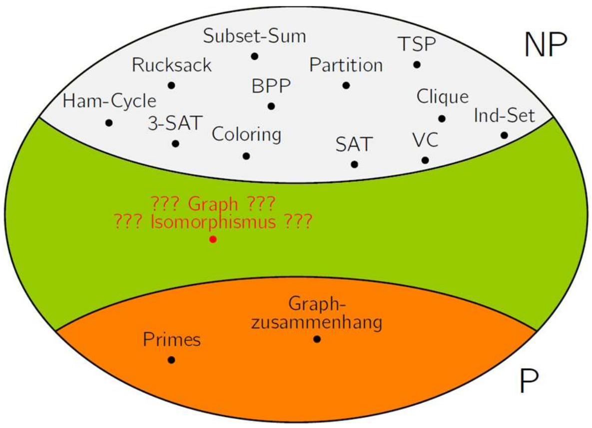
Exponential Time Hypothesis (ETH): Es existiert eine reelle Zahl $\delta > 0$, sodass kein Algorithmus 3-SAT in Zeit $O(2^{\delta n})$ löst.

- ETH ist unbewiesen, impliziert aber P ≠ NP. Falls GRAPH-ISOMORPHUS in NP => ETH falsch!

NP-intermediate

Ein Entscheidungsproblem $X \subseteq \Sigma^*$ heißt NP-intermediate, wenn $L \in$ NP und wenn sowohl $L \notin$ P als auch $L \notin$ NPC gilt.

Satz von Ladner: Wenn P $\neq$ NP gilt, dann existieren Probleme, die NP-intermediate sind.



Die Komplexitätsklassen PSPACE und EXPTIME

PSPACE ist die Klasse aller Entscheidungsprobleme, die durch eine DTM M entschieden werden, deren Worst Case Speicherplatzbedarf durch $q(n)$ mit einem Polynom q beschränkt ist.

NPSPACE ist die Klasse aller Entscheidungsprobleme, die durch eine NTM M entschieden werden, deren Worst Case Speicherplatzbedarf durch $q(n)$ mit einem Polynom q beschränkt ist.

Satz von Savitch: PSPACE = NSPACE

Da sich der Kopf einer Turingmaschine in einem Schritt nur um eine Position bewegen kann gilt:

NP $\subseteq$ NPSPACE = PSPACE

Problem: QUANTIFIED-SAT (Q-SAT)

Eingabe: Eine Boole'sche Formel φ in CNF über $\{x_1, \dots, x_n, y_1, \dots, y_n\}$

Frage: $\exists x_1 \forall y_1 \exists x_2 \forall y_2 \dots \exists x_n \forall y_n\ \varphi$?

Satz: Q-SAT liegt in PSPACE.

Anmerkung: Q-SAT ist PSPACE-vollständig.

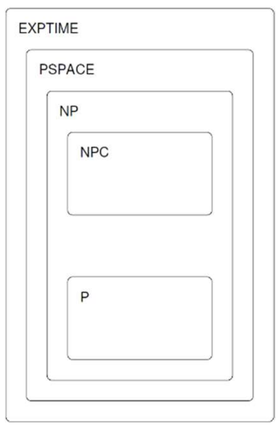
EXPTIME ist die Klasse aller Entscheidungsprobleme, die durch eine DTM M entschieden werden, deren Worst Case Laufzeit durch $2^{q(n)}$ mit einem Polynom q beschränkt ist.

Laufzeitbeispiele: $2^{\sqrt{n}}, 2^n, 3^n, n!, n^n$ Aber nicht: 2^{2^n}

- Bei einer Speicherplatzbeschränkung $s(n)$ gibt es nur $2^{O(s(n))}$ viele verschiedenen Konfigurationen für eine Turingmaschine. Daher ist die Rechenzeit durch $2^{O(s(n))}$ beschränkt.
- Die Probleme in PSPACE können deshalb in Zeit $2^{p(n)}$ gelöst werden: PSPACE $\subseteq$ EXPTIME

Problem: k-Schritt-HALTEPROBLEM

- Eingabe: Eine deterministische Turingmaschine M; eine ganze Zahl k
- Frage: Wenn M mit leerem Band gestartet wird, hält M dann nach höchstens k Schritten an? Die Zahl k ist binär (oder dezimal) kodiert.
- Satz: Das k-Schritt-HALTEPROBLEM liegt in EXPTIME.
- Anmerkung: Das k-Schritt-HALTEPROBLEM ist EXPTIME-vollständig.



- Wir haben gezeigt: P $\subseteq$ NP $\subseteq$ PSPACE $\subseteq$ EXPTIME
- Es ist nicht bekannt, welche dieser Inklusionen strikt sind.
- Möglicherweise gilt P = PSPACE oder NP = EXPTIME.
- Wir wissen allerdings, dass P $\neq$ EXPTIME gilt.