Berechenbarkeit und Komplexitätstheorie

Norbert Müller

Januar 2019

8. Das Postsche Korrespondenzproblem
9. Unentscheidbare Grammatikprobleme
10. Der Gödelsche Satz
11. Der λ-Kalkül
12. Komplexitätsklassen und das ***P*-*NP***-Problem
13. **NP**-Vollständigkeit
14. Weitere **NP**-vollständige Probleme
15. ‘Harte’ Probleme

Literatur


Uwe Schöning: Theoretische Informatik - kurzgefasst, 4. Auflage. Spektrum, Heidelberg 2001


John E. Hopcroft, Jeffrey D. Ullman: Introduction to Automata Theory, Languages and Computation, Addison-Wesley, Reading, Massachusetts, 1979 (Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie, Oldenbourg-Verlag, München, 2000).


C. Meinel, M. Mundhenk: Mathematische Grundlagen der Informatik. Mathematisches Denken und Beweisen, eine Einführung. Teubner, 2002.


K. A. Ross, C. R. B. Wright: Discrete Mathematics. Prentice Hall, 1988.


E. Kinber / C. Smith: Theory of Computing. A Gentle Introduction. Prentice Hall.

Es gibt zahlreiche gute Skripten im Internet, z.B.: Skripten zur Vorlesung Informatik-B2 an der Universität Duisburg-Essen (Prof. Luther, Prof. Hertling)

Es gibt ungezählte Einführungen in die Thematik; finden Sie am besten eine für Sie am besten geeignete heraus !

$\rightsquigarrow$ nicht nachprüfbare Hausaufgabe: gehen Sie in die Bibliothek oder in eine Buchhandlung und beginnen Sie, in verschiedenen Büchern zu lesen: wenigstens zwei davon sollten Sie über die Vorlesung begleiten...

Definition 1.1 (Intuitiver Berechenbarkeitsbegriff)

- *Die von einem Algorithmus **A** (z.B. einem JAVA-Programm) berechnete Funktion $f_A : \mathbb{N}^k \dashrightarrow \mathbb{N}$ ist definiert wie folgt:*

$$f_A(n_1, ..., n_k) = \begin{cases} n, & \text{falls } A \text{ bei Eingabe von } (n_1, ..., n_k) \text{ das Resultat } n \text{ berechnet und dann anhält,} \\ \text{undefiniert}, & \text{falls } A \text{ bei Eingabe von } (n_1, ..., n_k) \text{ endlos rechnet.} \end{cases}$$

- *Eine Funktion f heißt berechenbar, wenn es einen Algorithmus **A** gibt, der sie berechnet (also $f = f_A$).*
- *Insbesondere hält **A** damit genau dann bei der Eingabe $(n_1, ..., n_k)$ an, wenn $f(n_1, ..., n_k)$ definiert ist.*

Beispiel 1.2

Berechnung der Summe zweier natürlicher Zahlen $\boldsymbol{m}$ *und* $\boldsymbol{n}$ *als Funktion* $\boldsymbol{f} : \mathbb{N}^2 \to \mathbb{N}$ *mit* $\boldsymbol{f}(\boldsymbol{m}, \boldsymbol{n}) = \boldsymbol{n} + \boldsymbol{m}$*:*

```
1  import java.util.Vector;
2  import java.math.BigInteger;
3
4  public class Addition {
5    public static void main( String args[] ){
6
7      BigInteger m = new BigInteger( args[0] );
8      BigInteger n = new BigInteger( args[1] );
9
10     while ( n.compareTo(BigInteger.ZERO) > 0 ) {
11       n = n.subtract( BigInteger.ONE );
12       m = m.add( BigInteger.ONE );
13     }
14     System.out.println( m.toString() );
15   }
16 }
```

Beispiel 1.3

Berechnung einer partiellen Funktion $\mathbf{f} : \mathbb{N} \dashrightarrow \mathbb{N}$ *durch folgenden Algorithmus:*

```
1  import java.math.BigInteger;
2
3  public class Forever {
4    public static void main( String args[] ) {
5      BigInteger n = BigInteger.ONE;
6      while ( n.compareTo(BigInteger.ZERO) > 0 ) {
7        n = n.add(BigInteger.ONE);
8      }
9    }
10 }
```

Abbruchbedingung niemals erfüllt!

$\Rightarrow$ *Algorithmus berechnet partielle Funktion* $\mathbf{f}$ *von* $\mathbb{N}$ *nach* $\mathbb{N}$*, die nirgends definiert ist.*

Beispiel 1.4

Berechnung einer totalen Funktion $\boldsymbol{f} : \mathbb{N} \to \mathbb{N}$ *mit*

- $\boldsymbol{f}(\boldsymbol{n}) = \mathbf{1}$, *falls die Dezimaldarstellung von* $\boldsymbol{n}$ *ein Anfangsabschnitt der Dezimalbruchentwicklung von* $\pi = \mathbf{3,1415...}$ *ist,*
- $\boldsymbol{f}(\boldsymbol{n}) = \mathbf{0}$ *sonst.*

z.B.:

$$\boldsymbol{f}(\mathbf{3}) = \mathbf{1} \quad \boldsymbol{f}(\mathbf{314}) = \mathbf{1} \quad \boldsymbol{f}(\mathbf{31415}) = \mathbf{1} \quad \boldsymbol{f}(\mathbf{31416}) = \mathbf{0}$$

$$\boldsymbol{f}(\mathbf{3141592653589793238462643383279502884197169399}) = \mathbf{1}$$

$$\boldsymbol{f}(\mathbf{3141592653589793238462643383279502884197269399}) = \mathbf{0}$$

Dezimalbruchentwicklung von π *kann prinzipiell mit beliebiger Stellenzahl berechnet werden:*

$$\pi = \mathbf{4} \cdot \lim_{t \to \infty} (\mathbf{1} - \frac{\mathbf{1}}{\mathbf{3}} + \frac{\mathbf{1}}{\mathbf{5}} - \frac{\mathbf{1}}{\mathbf{7}} + \ldots \pm \frac{\mathbf{1}}{\mathbf{2}\boldsymbol{t} - \mathbf{1}})$$

$\Rightarrow$ *Für jedes* $\boldsymbol{n}$ *kann man* $\boldsymbol{f}(\boldsymbol{n})$ *in endlich vielen Schritten ausrechnen.*
$\Rightarrow$ $\boldsymbol{f}$ *ist berechenbar.*

Beispiel 1.5

Berechnung einer totalen Funktion $\boldsymbol{g} : \mathbb{N} \to \mathbb{N}$ *mit*

- $\boldsymbol{g}(\boldsymbol{n}) = \mathbf{1}$, *falls die Dezimaldarstellung von* $\boldsymbol{n}$ *als Teilwort in der Dezimalbruchentwicklung von* $\pi = \mathbf{3,1415...}$ *vorkommt,*
- $\boldsymbol{g}(\boldsymbol{n}) = \mathbf{0}$, *sonst.*

Ob $\boldsymbol{g}$ *berechenbar ist oder nicht, ist nicht bekannt!*

... zu wenig über die Zahl π *bekannt...*
... evtl. kommt jede Folge von Dezimalziffern bei π *vor...*

Wenn ja: $\boldsymbol{g}$ *ist die Funktion, die überall den Wert* $\mathbf{1}$ *annimmt.*

Beispiel 1.6

Berechnung einer totalen Funktion $h : \mathbb{N} \to \mathbb{N}$ *mit*

- $h(n) = 1$, *falls die* **7** *mindestens* n *mal hintereinander in der Dezimalbruchentwicklung von* π *vorkommt,*
- $h(n) = 0$, *sonst.*

h *ist berechenbar:*

- *Falls beliebig lange Folgen von* **7** *bei* π *vorkommen:*
  $\Rightarrow$ h *ist die konstante Funktion, die überall den Wert* **1** *hat.*
- *Falls* **7** *maximal* n_0*-fach hintereinanderbei* π *vorkommt:*
  $\Rightarrow$ *In diesem Fall gilt* $h(n) = 1$ *für* $n \leq n_0$, $h(n) = 0$ *für* $n > n_0$.

In jedem Fall gibt es einen Algorithmus zur Berechnung von h.

Allerdings: Noch unbekannt ist, welches der richtige Algorithmus ist...

Anmerkung: analog zu Beispiel 1.4 setze für $r \in \mathbb{R}$

- $f_r(n) = 1$, falls n Anfangsabschnitt der Dezimalbruchentwicklung von r ist,
- $f_r(n) = 0$, sonst.

Aber:

- Es gibt überabzählbar viele reelle Zahlen...
- Es gibt nur abzählbar viele (JAVA-)Algorithmen...

$\Rightarrow$ Die Dezimalbruchentwicklungen fast aller Zahlen kann man nicht berechnen!

f_r ist i.A. nicht berechenbar!

Beliebte formale Definitionen von Berechenbarkeit über

- Turingmaschinen
- WHILE-Programme
- Kalkül der μ-rekursiven Funktionen
- λ-Kalkül

Jedoch: Alle diese Berechenbarkeitsbegriffe sind äquivalent.

Zwei schwächere, äquivalente Definitionen:

- LOOP-Programme
- primitiv-rekursive Funktionen

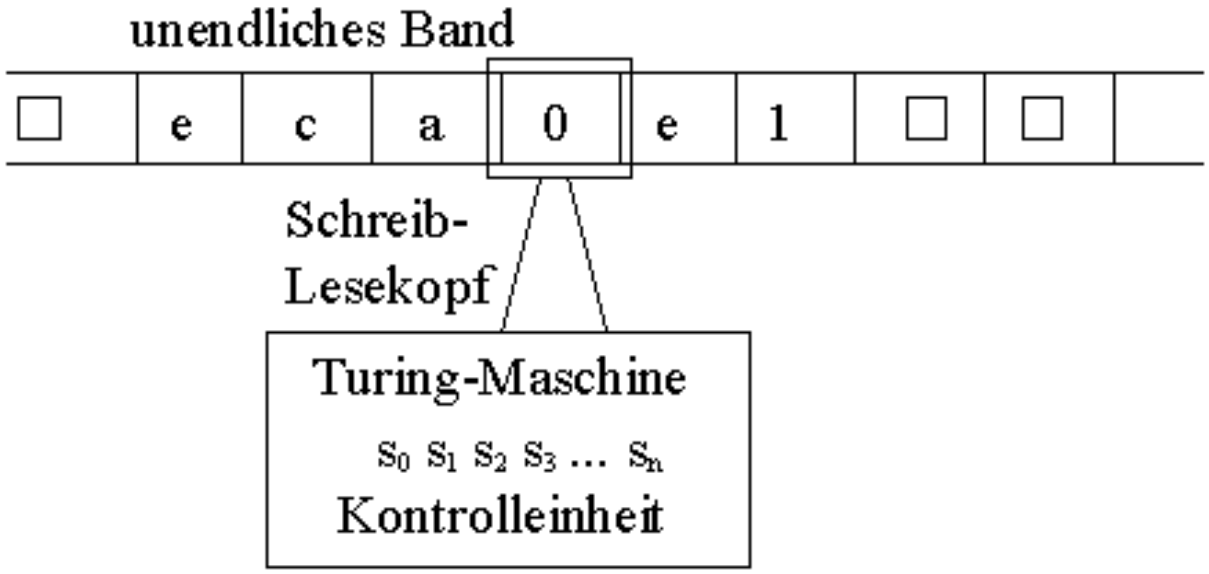
Church'sche These

Die durch Turingmaschinen berechenbaren Funktionen
(gleichbedeutend: WHILE-berechenbar, μ-rekursiv, ...)
sind genau die im intuitiven Sinn berechenbaren Funktionen auf natürlichen Zahlen.

Wichtige verwendete Schreibweisen
(i.W. aus “Automaten und Formale Sprachen”):

- $\mathbb{N} = \{\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}, \ldots\}$: natürliche Zahlen (inkl. $\mathbf{0}$)
- totale Funktion $\boldsymbol{f} : \boldsymbol{A} \to \boldsymbol{B}$
- partielle Funktion $\boldsymbol{f} : \boldsymbol{A} \dashrightarrow \boldsymbol{B}$
- Potenzmenge $\mathcal{P}(\boldsymbol{X}) := \{\boldsymbol{U} \mid \boldsymbol{U} \subseteq \boldsymbol{X}\}$ einer Menge $\boldsymbol{X}$
- Alphabet Σ: endliche Menge von Zeichen, z.B. $\Sigma = \{\boldsymbol{a}, \boldsymbol{b}\}$
- Worte/Zeichenketten über Σ: endliche Folgen von Zeichen aus Σ (vgl. Java-`String`)
- $|\boldsymbol{w}|$ Länge des Wortes $\boldsymbol{w}$ (vgl. Java-`length()`)
- $\#_{\boldsymbol{a}}\boldsymbol{w}$ gibt an, wie oft das Zeichen $\boldsymbol{a}$ in $\boldsymbol{w}$ vorkommt
- $\Sigma^{\boldsymbol{k}}$: Worte der Länge $\boldsymbol{k} \in \mathbb{N}$,
- ε: leeres Wort, String mit Länge $\mathbf{0}$
- Σ^*: Worte beliebiger Länge, $\Sigma^* = \bigcup_{\boldsymbol{k} \in \mathbb{N}} \Sigma^{\boldsymbol{k}}$

Bekannt(?): Turingmaschinen als Akzeptoren für Typ-0-Sprachen



Jetzt: Turingmaschinen als allgemeines Maschinenmodell zur Berechnung von Funktionen (Alan Turing, 1930):

$$\boldsymbol{f} : \boldsymbol{E}^* \dashrightarrow \boldsymbol{E}^* \text{ bzw. } \boldsymbol{f} : \mathbb{N}^{\boldsymbol{k}} \dashrightarrow \mathbb{N}$$

Definition 2.1 (Turingmaschine)

Eine (nichtdeterministische) Turingmaschine (TM) ist durch ein 7-Tupel beschrieben:

$$\boldsymbol{TM} = (\boldsymbol{S}, \boldsymbol{E}, \boldsymbol{A}, \delta, \boldsymbol{s_0}, \square, \boldsymbol{F})$$

Dabei bedeuten

- $\boldsymbol{S} = \{\boldsymbol{s_0}, \boldsymbol{s_1}, \ldots, \boldsymbol{s_n}\}$ *die Menge der Zustände,*
- $\boldsymbol{E} = \{\boldsymbol{e_1}, \boldsymbol{e_2}, \ldots, \boldsymbol{e_r}\}$ *das endliche Eingabealphabet,*
- $\boldsymbol{A} = \{\boldsymbol{a_0}, \boldsymbol{a_1}, \ldots, \boldsymbol{a_m}\}$ *das endliche Arbeitsalphabet (auch Bandalphabet genannt), es sei dabei* $\boldsymbol{E} \subset \boldsymbol{A}$,
- $\boldsymbol{s_0}$ *der Startzustand,*
- $\boldsymbol{a_0} = \square$ *das Blank, das zwar dem Arbeitsalphabet, aber nicht dem Eingabealphabet angehört,*
- $\boldsymbol{F} \subseteq \boldsymbol{S}$ *die Menge der Endzustände*

Fortsetzung von 2.1:

- δ sei die (totale) *Überführungsfunktion*
  mit (im deterministischen Fall)

$$\delta : (\boldsymbol{S} \setminus \boldsymbol{F}) \times \boldsymbol{A} \rightarrow \boldsymbol{S} \times \boldsymbol{A} \times \{\boldsymbol{L}, \boldsymbol{R}, \boldsymbol{N}\}$$

  bzw. (im nichtdeterministischen Fall)

$$\delta : (\boldsymbol{S} \setminus \boldsymbol{F}) \times \boldsymbol{A} \rightarrow \mathcal{P}(\boldsymbol{S} \times \boldsymbol{A} \times \{\boldsymbol{L}, \boldsymbol{R}, \boldsymbol{N}\})$$

  Hier bedeutet: ***L***= links, ***R*** = rechts, ***N***= neutral (nicht bewegen).

Die Überführungsfunktion wird folgendermaßen interpretiert:

Die Definition

$$\delta(\boldsymbol{s}, \boldsymbol{a}) = (\boldsymbol{s'}, \boldsymbol{b}, \boldsymbol{x})$$

bzw. im nichtdet. Fall

$$(\boldsymbol{s'}, \boldsymbol{b}, \boldsymbol{x}) \in \delta(\boldsymbol{s}, \boldsymbol{a})$$

mit $\boldsymbol{x} \in \{\boldsymbol{L}, \boldsymbol{R}, \boldsymbol{N}\}$, beschreibt folgendes Verhalten(bzw. folgendes mögliche Verhalten im nichtdet. Fall):

> *Wenn sich der Automat im Zustand **s** befindet und unter dem Kopf das Zeichen **a** steht, so schreibt der Automat **b** und geht ein Feld nach rechts (**R**), bzw. links (**L**), bzw. bewegt sich nicht (**N**) und geht in den Zustand **s'** über.*

Darstellung von δ: als Tabelle, evtl. mit Mengeneinträgen...

Definition 2.2 (Konfiguration)

Eine Konfiguration einer Turingmaschine $\boldsymbol{TM} = (\boldsymbol{S}, \boldsymbol{E}, \boldsymbol{A}, \delta, \boldsymbol{s_0}, \square, \boldsymbol{F})$ *ist ein Tripel* $(\boldsymbol{u}, \boldsymbol{s}, \boldsymbol{v})$ *aus* $\boldsymbol{A}^* \times \boldsymbol{S} \times \boldsymbol{A}^+$*:*

- $\boldsymbol{uv}$ *ist aktuelle Bandinschrift,*
- $\boldsymbol{s}$ *ist der aktuelle Zustand,*
- *Schreib-Lesekopf über erstem Zeichen von* $\boldsymbol{v}$*, daher* $\boldsymbol{v} \neq \varepsilon$*,*
- *Start der Maschine:*
  Startkonfiguration $(\varepsilon, \boldsymbol{s_0}, \boldsymbol{w}\square)$ *bei Eingabe von* $\boldsymbol{w}$*.*

O.B.d.A. $\boldsymbol{S} \cap \boldsymbol{A} = \emptyset$*, daher Schreibweise* $\boldsymbol{usv}$ *statt* $(\boldsymbol{u}, \boldsymbol{s}, \boldsymbol{v})$

Definition 2.3 (Übergangsrelation $\vdash$)

Zu einer (nicht)deterministischen Turingmaschine

$$\boldsymbol{TM} = (\boldsymbol{S}, \boldsymbol{E}, \boldsymbol{A}, \delta, \boldsymbol{s_0}, \square, \boldsymbol{F})$$

wird die Übergangsrelation $\vdash_{\boldsymbol{TM}}$ *(oder kurz* $\vdash$*) aus* $(\boldsymbol{A}^* \times \boldsymbol{S} \times \boldsymbol{A}^+)^2$ *folgendermaßen definiert:*

- $$\boldsymbol{a_1 \ldots a_m s b_1 \ldots b_n} \vdash \boldsymbol{a_1 \ldots a_m s' c b_2 \ldots b_n}$$

  falls: $(\boldsymbol{s'}, \boldsymbol{c}, \boldsymbol{N}) \in \delta(\boldsymbol{s}, \boldsymbol{b_1}), \boldsymbol{m} \geq \boldsymbol{0}, \boldsymbol{n} \geq \boldsymbol{1}$

- $$\boldsymbol{a_1 \ldots a_m s b_1 \ldots b_n} \vdash \boldsymbol{a_1 \ldots a_m c s' b_2 \ldots b_n}$$

  falls: $(\boldsymbol{s'}, \boldsymbol{c}, \boldsymbol{R}) \in \delta(\boldsymbol{s}, \boldsymbol{b_1}), \boldsymbol{m} \geq \boldsymbol{0}, \boldsymbol{n} \geq \boldsymbol{2}$

- $$\boldsymbol{a_1 \ldots a_m s b_1 \ldots b_n} \vdash \boldsymbol{a_1 \ldots a_{m-1} s' a_m c b_2 \ldots b_n}$$

  falls: $(\boldsymbol{s'}, \boldsymbol{c}, \boldsymbol{L}) \in \delta(\boldsymbol{s}, \boldsymbol{b_1}), \boldsymbol{m} \geq \boldsymbol{1}, \boldsymbol{n} \geq \boldsymbol{1}$

Fortsetzung von 2.3:

- $$a_1 \ldots a_m s b_1 \vdash a_1 \ldots a_m c s' \square$$

  falls $(s', c, R) \in \delta(s, b_1)$, $m \geq 0, n = 1$

- $$s b_1 \ldots b_n \vdash s' \square c b_2 \ldots b_n$$

  falls $(s', c, L) \in \delta(s, b_1)$, $m = 0, n \geq 1$

Bei Überschreiten der Grenzen der Bandinschrift werden also Blanks an die Teilwörter v bzw. u angefügt.

Dabei bedeutet das Symbol $\vdash$, dass die Konfiguration links in die jeweilige Konfiguration rechts übergehen kann (bzw. muss, wenn es genau eine Möglichkeit gibt).

Weiterhin sei $\vdash^*$ wieder die reflexiv-transitive Hülle der Relation $\vdash$.

Beispiel 2.4 (Inkrementieren einer Binärzahl)

Turingmaschine

$$TM = (\{s_0, s_1, s_2, s_f\}, \{0, 1\}, \{0, 1, \square\}, \delta, s_0, \square, \{s_f\})$$

mit

$\delta(s, a)$	0	1	$\square$
s_0	$(s_0, 0, R)$	$(s_0, 1, R)$	$(s_1, \square, L)$
s_1	$(s_2, 1, L)$	$(s_1, 0, L)$	$(s_f, 1, N)$
s_2	$(s_2, 0, L)$	$(s_2, 1, L)$	$(s_f, \square, R)$
s_f	–	–	–

Beispiel:

$$\begin{aligned} s_0111\square &\vdash 1s_011\square \vdash 11s_01\square \vdash 111s_0\square \\ &\vdash 11s_11\square \vdash 1s_110\square \vdash s_1100\square \vdash s_1\square000\square \\ &\vdash s_f1000\square \end{aligned}$$

Definition 2.5 (Initialkonfiguration, Finalkonfiguration, akzeptierte Sprache)

- *Initialkonfiguration beim Start der Turingmaschine mit Eingabe* $w \in E^*$ *ist* $s_0 w \square$.
- *Finalkonfigurationen sind alle Konfigurationen* $u s_f v$ *mit* $s_f \in F$. *Hier kann die Berechnung nicht mehr fortgesetzt werden.*
- *Weiter ist*

$$L(TM) := \{w \in E^* \mid s_0 w \square \vdash^* u s_f v, s_f \in F, u, v \in A^*\}$$

die von der Turingmaschine akzeptierte Sprache L.

Simulation endlicher Automat durch Turingmaschine einfach:

- Schreib-Lesekopf nutzt ausschließlich lesende Funktion
- Lesekopf zeichenweise lesend nach rechts
- Zustandsübergänge des endlichen Automaten werden übernommen
- Akzeptieren bei Erreichen von $\square$ (d.h. Ende der Eingabe) im 'Automaten-Endzustand'

Simulation Kellerautomat durch Turingmaschine z.B. wie folgt:

- Turing-Band zweigeteilt: Rechts Eingabewort, links Keller
- Nach Start sofort Markierung für 'Boden' des Kellers
- Übergangsfunktion des NKA durch mehrere Schritte der TM
- Gelesene Zeichen der Eingabe mit Spezialzeichen markieren
- Bestimmung des Überganges im NKA durch Inspektion der noch nicht markierten Eingabe und des simulierten Kellers
- Spitze des Kellers leicht zu finden, da links davon ein Blank steht.

$\Longrightarrow$ Turingmaschinen mindestens so mächtig wie NKA

Beispiel 2.6 (Turingmaschine zu $L = \{a^n b^n c^n \mid n > 0\}$)

(vgl. Sander, Stucky und Herschel S. 194f.)
Dazu definieren wir $TM := (S, E, A, \delta, s_0, \square, F)$ *wie folgt*

$$\begin{aligned} S &= \{s_0, s_1, s_2, s_3, s_4, s_f\} \\ A &= \{a, b, c, 0, 1, 2, \square\} \\ F &= \{s_f\} \end{aligned}$$

Idee der Arbeitsweise:

- *Ersetze jeweils erstes* a *durch* 0, b *durch* 1 *und* c *durch* 2
- *Achte dabei auf korrekte Reihenfolge*
- *Teste am Ende, ob alle* a, b, c *ersetzt wurden*

d.h.

$$s_0 a^n b^n c^n \square \vdash^* 0^k s_0 a^m 1^k b^m 2^k c^m \square \vdash^* 0^n s_0 1^n 2^n \square \vdash^* 0^n 1^n 2^n s_f \square$$

Fortsetzung von 2.6:

	a	b	c	0	1	2	$\square$
s_0	$(s_1, 0, R)$				$(s_4, 1, R)$		
s_1	(s_1, a, R)	$(s_2, 1, R)$			$(s_1, 1, R)$		
s_2		(s_2, b, R)	$(s_3, 2, L)$			$(s_2, 2, R)$	
s_3	(s_3, a, L)	(s_3, b, L)		$(s_0, 0, R)$	$(s_3, 1, L)$	$(s_3, 2, L)$	
s_4					$(s_4, 1, R)$	$(s_4, 2, R)$	$(s_f, \square, N)$
s_f							

Ist die Übergangsfunktion nicht definiert, bleibt der Automat stehen.

Jetzt: Turingmaschinen als allgemeines Maschinenmodell zur Berechnung von Funktionen:

$$\boldsymbol{f} : \boldsymbol{E}^* \dashrightarrow \boldsymbol{E}^* \text{ bzw. } \boldsymbol{f} : \mathbb{N}^{\boldsymbol{k}} \dashrightarrow \mathbb{N}$$

Für Zahlenfunktionen notwendig: Notation der Zahlen!

- Dezimalnotation $\boldsymbol{dez}(\boldsymbol{n})$, z.B. $\boldsymbol{dex}(\mathbf{12^2}) = \texttt{144}$ (mit $\{\texttt{0,1,2,3,4,5,6,7,8,9,}\} \subseteq \boldsymbol{E}$),
- Binärnotation $\boldsymbol{bin}(\boldsymbol{n})$, z.B. $\boldsymbol{bin}(\mathbf{13}) = \texttt{1101}$ und $\boldsymbol{bin}(\mathbf{0}) = \texttt{0}$ (mit $\{\mathbf{0}, \mathbf{1}\} \subseteq \boldsymbol{E}$) oder
- Unärnotation: $\boldsymbol{n}$ durch $\boldsymbol{n}$ Zeichen, z.B. $\mathbf{13}$ durch $\texttt{||||||||||||}$
- Eingabe von Vektoren: mit Trennsymbol (z.B. Blank oder #) zwischen den Zahlen
- zu Beginn: Lese-Kopf steht über dem ersten Eingabe-Zeichen (von links)
- am Ende: Ausgabe steht auf dem Band, Kopf steht links über dem ersten Zeichen

Definition 2.7

Eine Funktion $f : E^* \dashrightarrow E^*$ *heißt Turing-berechenbar, falls es eine deterministische Turingmaschine* **TM** *gibt derart, dass für* $x, y \in E^*$ *genau dann* $f(x) = y$ *gilt, wenn es einen Endzustand* $s' \in F$ *und Worte* u, v *gibt mit*

$$s_0 x \quad \vdash_{TM}^* \quad u s' v$$

und y *das längste Präfix von* v *über* E *ist.*

Eine Funktion $f : \mathbb{N}^k \dashrightarrow \mathbb{N}$ *heißt Turing-berechenbar, falls es eine deterministische Turingmaschine* **TM** *gibt derart, dass für* $n_1, \ldots, n_k, m$ *aus* $\mathbb{N}$ *genau dann* $f(n_1, \ldots, n_k) = m$ *gilt, wenn es einen Endzustand* $s' \in F$ *und Worte* u, v *gibt mit*

$$s_0 bin(n_1) \square bin(n_2) \square \ldots \square bin(n_k) \square \quad \vdash_{TM}^* \quad u s' v$$

und $bin(m)$ *das längste Präfix von* v *über* $\{0, 1\}$ *ist. Dabei sei* $bin(n)$ *die Binärdarstellung von* n.

Beispiel 2.8

a) Wir haben bereits in Beispiel 2.4 gesehen, dass die folgende Funktion $\boldsymbol{f}$ *(bezüglich der Binärdarstellung) berechenbar ist:*

$$\boldsymbol{f}(\boldsymbol{n}) = \boldsymbol{S}(\boldsymbol{n}) = \boldsymbol{n} + \mathbf{1}$$

b) Die nirgends definierte Funktion $\boldsymbol{f} : \mathbb{N}^{\boldsymbol{k}} \dashrightarrow \mathbb{N}$ *ist ebenfalls berechenbar.*
Passende Turingmaschine:
— $\boldsymbol{s_0}$*,*
— $\boldsymbol{s_1} \neq \boldsymbol{s_0}$*,*
— $\delta(\boldsymbol{s_0}, \boldsymbol{a}) = (\boldsymbol{s_0}, \boldsymbol{a}, \boldsymbol{N})$ *für alle* $\boldsymbol{a} \in \boldsymbol{A}$

Beispiel 2.9

- *Typ-0-Sprachen = Sprachen, die von n.det. TM erkannt werden.*
- *n.det. TM durch det. TM simulierbar*
- *d.h. L ist vom Typ 0, wenn es eine det. TM gibt mit $L = L(TM)$*

Dabei: $x \in L(TM) \Leftrightarrow$ von $s_0 x \square$ ist $u s_f v$ mit $s_f \in F$ erreichbar.
Verhalten der TM nach Erreichen von s_f für Akzeptieren unwichtig!
Daher: Turingmaschinen so definiert, dass sie nach Erreichen eines Endzustandes nicht weiterrechnen (können).
Jede TM ist so modifizierbar, dass sie nur mit $\ldots \square s_f \square \ldots$ akzeptiert...
Damit:

> *Eine Sprache L ist genau dann vom Typ 0,*
> *wenn die folgende Funktion $g : E^* \to E^*$ berechenbar ist:*

$$g(x) = \begin{cases} \varepsilon & \text{für } x \in L \\ \text{undefiniert} & \text{für } x \notin L \end{cases}$$

Bei $\boldsymbol{x} \notin \boldsymbol{L}(\boldsymbol{TM})$ wird *nie* ein Endzustand $\boldsymbol{s_f}$ erreicht, mögliche Gründe:

1. Die Maschine rechnet endlos oder
2. die Maschine ‘bleibt stecken’
   (d.h. zu $\boldsymbol{s}$ und $\boldsymbol{a}$ kein Übergang $\delta(\boldsymbol{s}, \boldsymbol{a})$ definiert)

Fall (2) vermeidbar mit Modifikation δ' von δ:

- neuer Zustand $\boldsymbol{s_l}$ mit $\delta'(\boldsymbol{s_l}, \boldsymbol{a}) = (\boldsymbol{s_l}, \boldsymbol{a}, \boldsymbol{N})$ für alle $\boldsymbol{a} \in \boldsymbol{A}$
- $\delta'(\boldsymbol{s}, \boldsymbol{a}) = (\boldsymbol{s_l}, \boldsymbol{a}, \boldsymbol{N})$, falls $\delta(\boldsymbol{s}, \boldsymbol{a})$ nicht definiert (für $\boldsymbol{s} \in \boldsymbol{S} \setminus \boldsymbol{F}$)

Damit:

Eine Sprache **L** *ist genau dann vom Typ 0,*
wenn es eine deterministische Turingmaschine gibt,
die genau bei den Wörtern aus **L** *(als Eingabe)* anhält.

Erweiterung der Turingmaschinen-Definition:

Mehrband-Turingmaschine ***MTM***

- mit $\boldsymbol{k} \geq \mathbf{1}$ Bändern und Schreibköpfen
- alle Schreibköpfe können unabhängig voneinander operieren
- Übergangsfunktion $\delta : \boldsymbol{S} \times \boldsymbol{A}^{\boldsymbol{k}} \rightarrow \boldsymbol{S} \times \boldsymbol{A}^{\boldsymbol{k}} \times \{\boldsymbol{L}, \boldsymbol{R}, \boldsymbol{N}\}^{\boldsymbol{k}}$
- Definition von Konfiguration und Übergangsrelation passend dazu...

Satz 2.10

*Zu jeder Mehrbandmaschine **MTM** gibt es eine Einbandmaschine **TM**, die dieselbe Sprache akzeptiert, bzw. dieselbe Funktion berechnet.*

Beweisidee: Gegeben Mehrbandmaschine ***MTM***

- mit k Bändern (**1** bis k)
- Arbeitsalphabet A

Konstruiere Turingmaschine ***TM***, die ***MTM*** simuliert:

- Schreib-Leseband der ***TM*** in $2k$ Spuren (**1** bis $2k$) unterteilt
- Spur $2i-1$ enthält Inhalt von Band i der ***MTM***
- Spur $2i$ hat an genau einer Stelle das Zeichen $*$, ansonsten nur Blanks.
- Das Zeichen $*$ zeigt an, dass der Lese-Schreibkopf des i-ten Bandes der ***MTM*** gerade an dieser Stelle steht.
- Kopf der ***TM*** testet alle Kopfpositionen der ***MTM*** nacheinander
- Neues Arbeitsalphabet $A' = A \cup (A \cup \{*\})^{2k}$.

Arbeitsweise der ***TM***:

- Gestartet wird mit der Eingabe $a_1 \ldots a_n \in E^*$.
- Zunächst erzeugt ***TM*** die Startkonfiguration von ***MTM***:
  — Eingabewort $a_1 \ldots a_n$ auf Spur **1**
  — alle anderen ungeraden Spuren sind leer
  — alle geraden Spuren haben $*$ unter dem Lese-Schreibkopf
- In Spuren unterteilter Bereich des Bandes ist in Blanks eingeschlossen
  $\Rightarrow$ ***TM*** kann stets durch Lauf über alle beschriebenen Zellen Information über den nächsten Schritt den ***MTM*** sammeln und ihn simulieren
- jeder Einzelschritt der ***MTM*** wird durch mehrere (=viele) Schritte der ***TM*** simuliert.

Vorteile von Mehrbandmaschinen:

- Jedes Band stellt ein unendlich langes Register dar.
- Besitzt eine Maschine nur eines dieser Bänder, ist sie meist damit beschäftigt, auf diesem Band hin- und herzufahren...
- Mehrbandmaschine: Daten auf Bänder verteilt, mit leichterem Zugriff

Z.B.: Simulation einer Einbandmaschine mit k-Band-TM auf Band i ohne Änderung der anderen Bänder:

Übergangsfunktion dazu z.B. statt $\delta(s, c) = (s', d, R)$ jetzt

$$\delta'(s, a_1, ..., c, ..., a_k) = (s', a_1, ..., d, ..., a_k, N, ..., R, ..., N)$$

I.d.R: k groß genug gewählt und dann in der Schreibweise weggelassen... (Beispiele: später)

Im Folgenden: 'Programmieren' mit Turingmaschinen...

Vgl. Beispiel 2.4, Inkrementieren von Zahlen mit Einbandmaschine

- Analog ***MTM***, die Inhalt von Band ***i*** inkrementiert
  Schreibweise:
  Band ***i*** *:= Band* ***i*** $+$ **1**

- Analog: ***MTM***, die Zahlen dekrementiert,
  d.h. aus $\boldsymbol{n}>\mathbf{0}$ wird $\boldsymbol{n}-\mathbf{1}$, Dekrement von **0** sei wieder **0**
  Schreibweise:
  Band ***i*** *:= Band* ***i*** $-$ **1**

- Weitere naheliegende Abkürzungen:
  Band ***i*** $:=$ **0**

  und
  Band ***i*** $:=$ *Band* ***j***

Hier Verzicht auf Angabe der Turingtabellen ...

- Hintereinanderschaltung von Turingmaschinen:
  Gegeben

$$M_1 = (S_1, E, A_1, \delta_1, s_{01}, \square, F_1)$$

$$M_2 = (S_2, E, A_2, \delta_2, s_{02}, \square, F_2)$$

  O.B.d.A. $S_1 \cap S_2 = \emptyset$
  Schreibweise

$$\text{start} \longrightarrow M_1 \longrightarrow M_2 \longrightarrow \text{stop}$$

  oder $M_1; M_2$ für

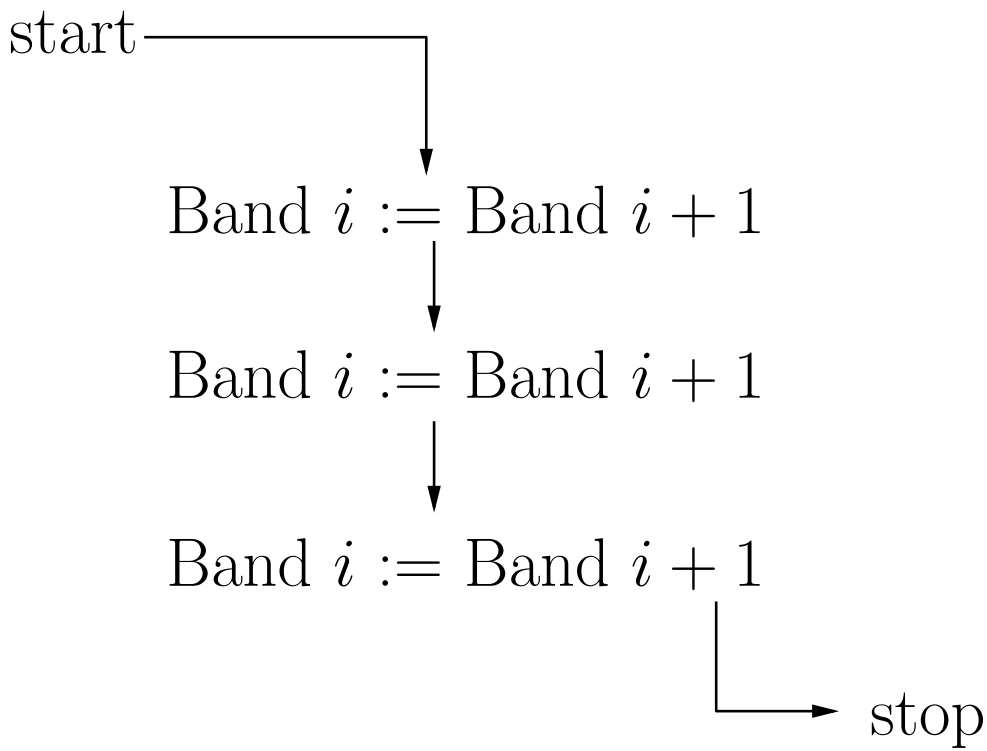
$$M = (S_1 \cup S_2, E, A_1 \cup A_2, \delta, s_{01}, \square, F_2)$$

  mit

$$\delta = \delta_1 \cup \delta_2 \cup \{((s_f, a), (s_{02}, a, N)) \mid s_f \in F_1, a \in A_1\}$$

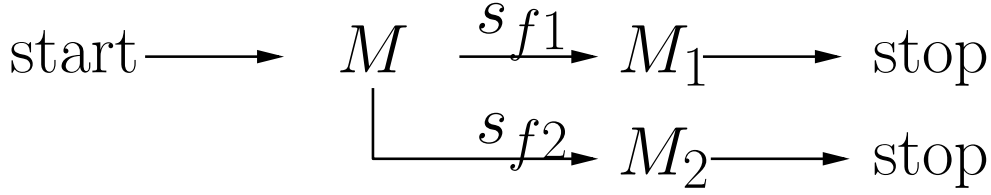
(Hier: Übergangsfunktionen als Relationen über kartesischen Produkten)

Beispiel: Die durch:



beschriebene Maschine realisiert die Operation ‘Band ***i*** := Band ***i*** + **3**’.

Weiteres Beispiel: Die Maschine



realisiert eine Verzweigung über die Endzustände s_{f1} und s_{f2} der Maschine M und ein Fortfahren mit entweder M_1 oder M_2.

Beispiel: Test, ob **0** (als Zahl) auf dem Band steht, und bedingte Verzweigung

Schreibweise:

Band = **0***?*

Zustandsmenge $\boldsymbol{S} := \{\boldsymbol{s_0}, \boldsymbol{s_1}, \boldsymbol{ja}, \boldsymbol{nein}\}$

Endzustände ***ja*** und ***nein***

Arbeitsalphabet $\boldsymbol{A} = \{\mathbf{0}, \mathbf{1}, \square\}$.

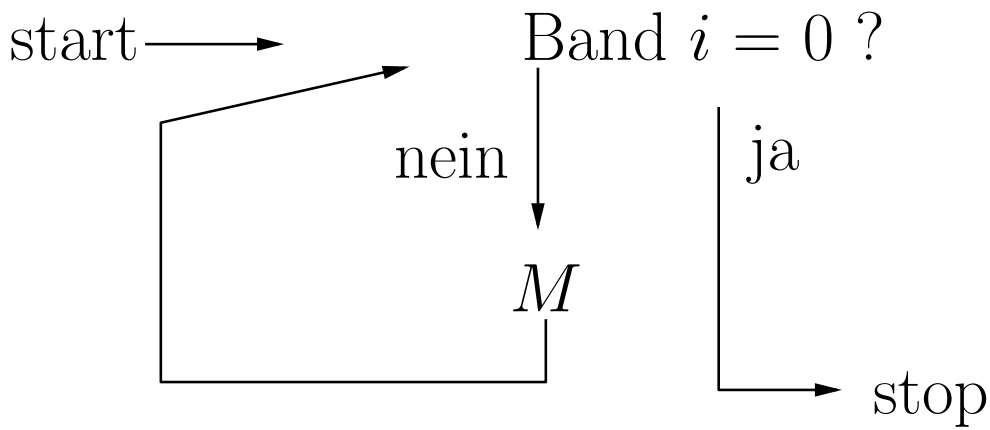
Übergangsfunktion:

δ	**1**	**0**	$\square$
$\boldsymbol{s_0}$	$(\boldsymbol{nein}, \mathbf{1}, \boldsymbol{N})$	$(\boldsymbol{s_1}, \mathbf{0}, \boldsymbol{R})$	$(\boldsymbol{ja}, \square, \boldsymbol{N})$
$\boldsymbol{s_1}$	$(\boldsymbol{nein}, \mathbf{1}, \boldsymbol{L})$	$(\boldsymbol{s_1}, \mathbf{0}, \boldsymbol{R})$	$(\boldsymbol{ja}, \square, \boldsymbol{L})$

Analog:

Band $\boldsymbol{i}$ = **0***?*

Simulation einer WHILE-Schleife:



mit der Schreibweise

WHILE Band ***i*** $\neq$ **0** *DO* ***M***

Syntaktische Grundbausteine von LOOP-Programmen:

- Variablen: x_0 x_1 x_2 **. . .**
- Konstanten: **0 1 2 . . .**
  (also für *jede* natürliche Zahl eine Konstante)
- Trennsymbole und Operationszeichen: $;$ $:=$ $+$ $-$
- Schlüsselwörter: **`LOOP DO END`**

LOOP-Programme sind dann:

- Jede Wertzuweisung $x_i := x_j + c$ und $x_i := x_j - c$ ist ein LOOP-Programm (für Variablen x_i, x_j und Konstanten c).
- Sind P_1 und P_2 LOOP-Programme,
  dann ist auch $P_1 ; P_2$ ein LOOP-Programm.
- Ist x Variable und P LOOP-Programm, dann ist auch
  `LOOP` x **`DO`** P **`END`** ein LOOP-Programm.
- Weitere Konstruktionen sind nicht zugelassen.

Die Semantik (Bedeutung) von LOOP-Programmen ergibt sich aus:

- Die Werte der Variablen x_i sind (beliebig) aus $\mathbb{N}$.
- Bei $x_i := x_j + c$ erhält x_i den Wert $x_j + c$
  analog zu üblichen Programmiersprachen...
- Bei $x_i := x_j - c$ erhält x_i den Wert $x_j - c$, falls $x_j \geq c$,
  ansonsten den Wert 0. (Damit bleiben alle Werte in $\mathbb{N}$.)
- Bei $P_1; P_2$ wird erst P_1 ausgeführt, dann P_2.
- Bei `LOOP` x `DO` P `END` wird das Programm P so oft ausgeführt,
  wie der Wert von x *zu Beginn* der LOOP-Anweisung angibt.

Definition 3.1 (LOOP-Berechenbarkeit)

Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ *heißt LOOP-berechenbar, falls*

- *es ein LOOP-Programm* P *gibt, so dass* P*,*
- *gestartet mit* $n_1, \ldots, n_k$ *aus* $\mathbb{N}$ *in den Variablen* $x_1, \ldots, x_k$
- *und* 0 *in allen anderen Variablen*
- *mit dem Wert* $f(n_1, \ldots, n_k)$ *in der Variablen* x_0 *stoppt.*

Anmerkungen:

1. alle LOOP-berechenbaren Funktionen sind total:
   Jedes LOOP-Programm hält auf *jeder* Eingabe!
   (Beweis: Induktion über den Aufbau der LOOP-Programme...)
2. Es gibt also Turing-berechenbare Funktionen,
   die nicht LOOP-berechenbar sind...
3. Sogar: Nicht einmal jede *totale* Turing berechenbare Funktion
   ist LOOP-berechenbar...

Im Folgenden: ‘Programmierübungen’ mit LOOP-Programmen...
(später als abgekürzte Schreibweisen für LOOP-Programme benutzt...)

- Zuweisung ’$x_i := x_k$’ durch ’$x_i := x_k + 0$’
  mit der Konstanten $c = 0$
- Zuweisung ’$x_i := c$’ durch ’$x_i := x_k + c$’
  für eine Variable x_k, die immer Wert 0 hat.
- Bedingte Ausführung ’`IF` $x_k = 0$ `THEN` Q `END`’ durch

  $y := 1$;
  `LOOP` x_k `DO` $y := 0$ `END`;
  `LOOP` y `DO` Q `END`

  mit einer ansonsten nicht benutzten Variable y
- Bedingte Ausführung ’`IF` $x_k = 0$ `THEN` Q `ELSE` R `END`’
  als Übungsaufgabe...

- Additionen '$x_i := x_j + x_k$' mit $i \neq j$:

$$x_i := x_k \,;$$
$$\texttt{LOOP}\ \ x_j\ \ \texttt{DO}\ \ x_i := x_i + 1\ \ \texttt{END}$$

- Spezialfall '$x_0 := x_1 + x_2$':
  berechnet Addition $+ : \mathbb{N}^2 \to \mathbb{N}$ in Sinne von Definition 3.1
- Der Fall $i = j$, d.h. '$x_i := x_i + x_k$', ist sogar noch einfacher:

$$\texttt{LOOP}\ \ x_k\ \ \texttt{DO}\ \ x_i := x_i + 1\ \ \texttt{END}$$

- Subtraktion '$x_i := x_j - x_k$': ähnlich...
  (wieder mit Wert 0 bei $x_j < x_k$)

- Multiplikationen '$x_i := x_j \cdot x_k$'

$$x_i := 0;$$
$$\texttt{LOOP } x_j \texttt{ DO } x_i := x_i + x_k \texttt{ END}$$

was wiederum nur eine Abkürzung ist für

$$x_i := 0;$$
$$\texttt{LOOP } x_j \texttt{ DO}$$
$$\quad \texttt{LOOP } x_k \texttt{ DO } x_i := x_i + 1 \texttt{ END}$$
$$\texttt{END}$$

- Die ‘Signum’-Funktion $\boldsymbol{sg} : \mathbb{N} \to \mathbb{N}$ mit

$$\boldsymbol{sg}(\boldsymbol{x}) := \begin{cases} \mathbf{1} \text{ falls } \boldsymbol{x} > \mathbf{0} \\ \mathbf{0} \text{ falls } \boldsymbol{x} = \mathbf{0} \end{cases}$$

  ist realisierbar über

$$\boldsymbol{x_0} := \mathbf{1}\,; \\ \texttt{IF } \boldsymbol{x_1} = \mathbf{0} \texttt{ THEN } \boldsymbol{x_0} := \mathbf{0} \texttt{ END}$$

- Die inverse Signum-Funktion $\overline{\boldsymbol{sg}}(\boldsymbol{x}) := \mathbf{1} - \boldsymbol{sg}(\boldsymbol{x})$ kann ähnlich berechnet werden.
- Die ‘Gleichheits’-Funktion $\boldsymbol{se}(\boldsymbol{x}, \boldsymbol{y})$ mit

$$\boldsymbol{se}(\boldsymbol{x}, \boldsymbol{y}) := \begin{cases} \mathbf{1} \text{ falls } \boldsymbol{x} = \boldsymbol{y} \\ \mathbf{0} \text{ falls } \boldsymbol{x} \neq \boldsymbol{y} \end{cases}$$

  kann über die Funktionen $\boldsymbol{sg}$ und $\overline{\boldsymbol{sg}}$ realisiert werden.

- Ganzzahldivision $x_i := x_j \text{ DIV } x_k$ mit $x_k > 0$:
  Dazu zunächst '`IF` $x_k \leq x_j$ `THEN` Q `END`' über

$$
\begin{aligned}
&x_0 := x_k - x_j\,;\\
&x_1 := \overline{sg}(x_0)\,;\\
&\texttt{LOOP } x_1 \texttt{ do } Q \texttt{ END}
\end{aligned}
$$

  Damit erhalten wir '$x_i := x_j$ DIV x_k' durch:

$$
\begin{aligned}
&x_i := 0\,;\\
&\texttt{LOOP } x_j \texttt{ DO}\\
&\quad \texttt{IF } x_k \leq x_j \texttt{ THEN } x_i := x_i + 1 \texttt{ END};\\
&\quad x_j := x_j - x_k\,;\\
&\texttt{END}
\end{aligned}
$$

- Zur Übung realisieren Sie z.B. die Modulo-Funktion MOD als LOOP-Programm (wird später noch benötigt...).

- Weitere Übung: Cantor'sche Bijektion $\langle \cdot, \cdot \rangle : \mathbb{N}^2 \to \mathbb{N}$

$$\langle x, y \rangle \quad := \quad y + \sum_{i \leq x+y} i \quad = \quad y + \frac{(x+y)(x+y+1)}{2}$$

$\langle x, y \rangle$	0	1	2	3	4	5	...
0	0	2	5	9	14	20	...
1	1	4	8	13	19	26	...
2	3	7	12	18	25	33	...
3	6	11	17	24	32	41	...
4	10	16	23	31	40	50	...
5	15	22	30	39	49	60	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	

Sowohl die Bijektion von $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ als auch die einzelnen Komponenten $\boldsymbol{p_1}, \boldsymbol{p_2} : \mathbb{N} \to \mathbb{N}$ der Umkehrfunktion (d.h. $\langle \boldsymbol{p_1}(\boldsymbol{z}), \boldsymbol{p_2}(\boldsymbol{z}) \rangle = \boldsymbol{z}$) sind LOOP-berechenbar:

- Berechnung von $\boldsymbol{y} + \sum_{\boldsymbol{i} \leq \boldsymbol{x}+\boldsymbol{y}} \boldsymbol{i}$ benötigt i.W. LOOP-Schleife mit Additionen
- Berechnung der Komponenten $\boldsymbol{p_1}(\boldsymbol{z}), \boldsymbol{p_2}(\boldsymbol{z})$:
  - Suche (von $\boldsymbol{k} = \boldsymbol{0}, \ldots, \boldsymbol{z}$) nach größtem $\boldsymbol{k}$ mit $\sum_{\boldsymbol{i} \leq \boldsymbol{k}} \boldsymbol{i} \leq \boldsymbol{z}$.
  - Dann $\boldsymbol{p_2}(\boldsymbol{z}) = \boldsymbol{z} - \sum_{\boldsymbol{i} \leq \boldsymbol{k}}$ und $\boldsymbol{p_1}(\boldsymbol{z}) = \boldsymbol{k} - \boldsymbol{p_2}(\boldsymbol{z})$.

Erweiterung der LOOP-Programme zu WHILE-Programmen:

- unverändert wie bei Loop:
  Variablen x_0 x_1 x_2 ...,
  Konstanten **0 1 2 ...**,
  Trennsymbole und Operationszeichen ; := + −
- Schlüsselwörter: **LOOP DO END**
- neues, zusätzliches Schlüsselwort: **WHILE**

WHILE-Programme sind dann wie folgt definiert:

- Jede Wertzuweisung $x_i := x_j + c$ und $x_i := x_j - c$ ist ein WHILE-Programm (für Variablen x_i, x_j und Konstanten c).
- Sind P_1 und P_2 WHILE-Programme,
  dann ist auch P_1; P_2 ein WHILE-Programm.
- Ist x Variable und P WHILE-Programm, dann sind auch
  LOOP x **DO** P **END** und **WHILE** x **DO** P **END**
  WHILE-Programme
- Weitere Konstruktionen sind nicht zugelassen.

Jedes LOOP-Programm ist damit auch ein WHILE-Programm.

Die Semantik von WHILE-Programmen wird analog zu LOOP definiert.

Wichtig ist dabei die Semantik der neuen WHILE-Schleife:

- Bei **`WHILE`** ***x*** **`DO`** ***P*** **`END`** wird wiederholt ***P*** ausgeführt,
  bis der (sich evtl. ändernde!) Inhalt von ***x***
  zu Beginn des Schleifenrumpfes ***P*** den Wert. **0** hat.
- Bei **`LOOP`** ***x*** **`DO`** ***P*** **`END`** wurde ***P*** so oft ausgeführt,
  wie der Wert von ***x*** *zu Beginn* der LOOP-Anweisung angibt.

Also:

- LOOP-Programme müssen im Voraus wissen, wie oft eine Schleife ausgeführt wird.
- WHILE-Programme können hingegen unbeschränkt ‘suchen’...

Definition 3.2 (WHILE-Berechenbarkeit)

Eine Funktion $\boldsymbol{f} : \mathbb{N}^{\boldsymbol{k}} \dashrightarrow \mathbb{N}$ *heißt WHILE-berechenbar, falls*

- *es ein WHILE-Programm* $\boldsymbol{P}$ *gibt, so dass* $\boldsymbol{P}$
- *gestartet mit* $\boldsymbol{n_1}, \ldots, \boldsymbol{n_k}$ *aus* $\mathbb{N}$ *in den Variablen* $\boldsymbol{x_1}, \ldots, \boldsymbol{x_k}$
- *und* $\mathbf{0}$ *in allen anderen Variablen*
- *mit* $\boldsymbol{f}(\boldsymbol{n_1}, \ldots, \boldsymbol{n_k})$ *in* $\boldsymbol{x_0}$ *stoppt, falls* $\boldsymbol{f}(\boldsymbol{n_1}, \ldots, \boldsymbol{n_k})$ *definiert ist,*
- *und nie stoppt, falls* $\boldsymbol{f}(\boldsymbol{n_1}, \ldots, \boldsymbol{n_k})$ *nicht definiert ist.*

Bei WHILE-Programmen kann man auf die LOOP-Anweisung sogar komplett verzichten, '**LOOP** x_j **DO** P **END**' wird simuliert durch

$$x_k := x_j;$$
$$\texttt{WHILE } x_k \texttt{ DO } x_k := x_k - 1; \; P \texttt{ END}$$

Dabei sei x_k eine Variable, die im Programm P nicht verwendet wird.

Alle LOOP-berechenbaren Funktionen sind WHILE-berechenbar.

Die Umkehrung gilt nicht, z.B. für die partielle Funktion f mit

$$f(x) := \begin{cases} 42 & \text{falls } x = 0 \\ \textit{undefiniert} & \text{falls } x > 0 \end{cases}$$

f ist sicher nicht LOOP-, aber WHILE-berechenbar:

$$\texttt{WHILE } x_1 \texttt{ DO } x_1 := x_1 + 1 \texttt{ END}; \; x_0 := 42$$

Es gibt sogar *totale* Funktionen, die nicht LOOP-, aber WHILE-berechenbar sind (z.B. Ackermann-Funktion)!

Vergleich WHILE-Programm mit Mehrband-Turingmaschinen:

- Per Induktion über Aufbau von WHILE-Programmen:
  Für jedes WHILE-Programm gibt es eine MTM, die die gleiche Funktion berechnet)
- Verwende pro benutzter Variable x_i ein eigenes Band i der MTM, nutze die bereits betrachteten MTM-Konstruktionen

Damit sofort:

Satz 3.3

Jede WHILE-berechenbare Zahlenfunktion ist Turing-berechenbar.

Satz 3.4

Turing-berechenbare Zahlenfunktionen sind WHILE-berechenbar.

Gegeben Turingmaschine

$$\boldsymbol{TM} = (\boldsymbol{S}, \boldsymbol{E}, \boldsymbol{A}, \delta, \boldsymbol{s_0}, \square, \boldsymbol{F})$$

zur Berechnung einer Zahlenfunktion $\boldsymbol{f}$.

TM wird durch WHILE-Programm aus drei Teilen simuliert:

1. Eingabe $\boldsymbol{n_1}, \ldots, \boldsymbol{n_l}$ für $\boldsymbol{f}$ wird in drei Zahlen $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ umgerechnet, die die Startkonfiguration der ***TM*** darstellen,
   $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ werden dabei in Variablen gespeichert!
2. Die Berechnung durch ***TM*** wird Schritt für Schritt durch entsprechende Änderung der drei Zahlen $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ nachvollzogen
3. Aus den Zahlen $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$, die die Endkonfiguration beschreiben, wird der Ausgabewert extrahiert und in $\boldsymbol{x_0}$ gespeichert

Sei

$$S = \{s_0, \ldots, s_k\} \quad A = \{a_1, \ldots, a_m\}$$

Wähle $b \in \mathbb{N}$ mit $b > m$

Eine Konfiguration

$$a_{i_1} \ldots a_{i_p}\ s_l\ a_{j_1} \ldots a_{j_q}$$

wird durch drei Zahlen x, y, z beschrieben:

$$x = (i_1 \ldots i_p)_b \quad y = (j_q \ldots j_1)_b \quad z = l$$

Dabei sei

$$(i_1 \ldots i_p)_b = \sum_{l=1}^{p} i_l \cdot b^{p-l}$$

d.h. Worte werden als Zahlen zur Basis b notiert.

Wichtig: Bei y ist die Reihenfolge der Ziffern vertauscht!

- x: Bandinschrift links des Lese-Schreibkopfes
- y: Bandinschrift rechts des Lese-Schreibkopfes, inklusive dem Zeichen unter dem Kopf
- z: aktueller Zustand der *TM*

Simulation der Schritte der ***TM*** durch Wiederholung einer großen Verzweigung:

```
WHILE Zustand s ist kein Endzustand DO
    Bestimme Zeichen a unter dem Kopf;
    IF (s = s_0) und (a = a_1):
       simuliere δ(s_0, a_1) in x, y, z;
    ELSE IF...
       ...
    ELSE IF (s = s_k) und (a = a_m):
       simuliere δ(s_k, a_m) in x, y, z;
END WHILE;
```

- Bestimmung des *Zeichens* a: Berechnung der *Zahl* 'y MOD b'
- Vergleich '$a = a_j$' entspricht Test 'y MOD $b = j$'
- Vergleich '$s = s_i$' entspricht Test '$z = i$'.

Insgesamt $(k+1) * m$ verschiedene Fälle:

- für $k+1$ Zustände s und
- für m Symbole a im Arbeitsalphabet A (unter dem Kopf)

Simulation der Übergangsfunktion $\delta(\boldsymbol{s_i}, \boldsymbol{a_j}) = (\boldsymbol{s_{i'}}, \boldsymbol{a_{j'}}, \boldsymbol{B})$:
Erstellung der Folgekonfiguration in den Variablen $\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{y}$

Hier nur $\boldsymbol{B} = \boldsymbol{L}$ bei $\boldsymbol{x} \neq \boldsymbol{0}$ als Beispiel:

$\boldsymbol{z} := \boldsymbol{i'}$;
$\boldsymbol{y} := \boldsymbol{y}$ DIV $\boldsymbol{b}$;
$\boldsymbol{y} := \boldsymbol{b} * \boldsymbol{y} + \boldsymbol{j'}$;
$\boldsymbol{y} := \boldsymbol{b} * \boldsymbol{y} + (\boldsymbol{x}$ MOD $\boldsymbol{b})$;
$\boldsymbol{x} := \boldsymbol{x}$ DIV $\boldsymbol{b}$;

D.h.:

- Variable für $\boldsymbol{z}$ direkt passend für Zustand $\boldsymbol{s_{i'}}$ ändern,
- $\boldsymbol{a_{j'}}$ als neues letztes Zeichen in $\boldsymbol{y}$ vermerken (statt $\boldsymbol{a_j}$) und
- letztes Zeichen aus $\boldsymbol{x}$ in $\boldsymbol{y}$ übertragen (Kopf nach links!)
- bei $\boldsymbol{x} = \boldsymbol{0}$ statt $\boldsymbol{y} := \boldsymbol{b} * \boldsymbol{y} + (\boldsymbol{x}$ MOD $\boldsymbol{b})$ nun $\boldsymbol{y} := \boldsymbol{b} * \boldsymbol{y} + \boldsymbol{j}_{\square}$ (bei $\boldsymbol{a}_{\boldsymbol{j}_{\square}} = \square$)

Erster Teil (Kodierung der Eingabe) und dritter Teil (Extraktion der Ausgabe): i.W. analog...

Als Folgerung aus der Konstruktion ergibt sich:

Satz 3.5

Jedes Turing-Programm kann durch ein WHILE-Programm mit nur einer einzigen WHILE-Schleife berechnet werden.

...nur eine WHILE-Schleife, aber evtl. viele LOOP-Schleifen $\Rightarrow$

Satz 3.6 (Kleenesche Normalform für WHILE-Programme)

Zu jedem WHILE-Programm gibt es ein äquivalentes WHILE-Programm mit nur einer einzigen WHILE-Schleife.

Dazu:

- Transformiere WHILE-Programm P in Turing-Programm P'
- Transformiere Turing-Programm P' in WHILE-Programm P'' mit nur einer WHILE-Schleife (nach 3.5)

Im Folgenden:
Anderer Zugang zu Berechenbarkeit von $f : \mathbb{N}^k \to \mathbb{N}$ für $k \in \mathbb{N}$

Dabei auch $k = 0$ erlaubt! $\Rightarrow$ Nullstellige Funktion, Konstante

Beachte Unterschied z.B bei

$$f : \mathbb{N}^0 \to \mathbb{N} \quad \text{mit} \quad f() = 42$$
$$g : \mathbb{N}^2 \to \mathbb{N} \quad \text{mit} \quad (\forall x, y)\ g(x, y) = 42$$

In Programmiersprachen:
`int f()` und `int g(int,int)` haben verschiedenen Typ!

Nutze folgenden Grundstock an sehr einfachen Funktionen:

- $Z : \mathbb{N}^0 \to \mathbb{N}$, die nullstellige Funktion (=Konstante) mit $Z() = 0$
- $S : \mathbb{N}^1 \to \mathbb{N}$, die einstellige Nachfolgerfunktion mit $S(n) = n + 1$
- $pr_j^k : \mathbb{N}^k \to \mathbb{N}$, die k-stellige Projektion auf die j-te Komponente, also $pr_j^k(x_1, \ldots, x_k) = x_j$, definiert für $k \geq 1$ und $1 \leq j \leq k$.

Mit Erzeugungsschemata neue Funktionen aus gegebenen Zahlenfunktionen:

Kompositionsschema $\boldsymbol{Komp} : (\boldsymbol{g_1}, \ldots, \boldsymbol{g_m}, \boldsymbol{h}) \mapsto \boldsymbol{f}$

- gegeben: $\boldsymbol{m}$ Stück $\boldsymbol{k}$-stellige Funktionen $\boldsymbol{g_1}$,...,$\boldsymbol{g_m}$ und eine $\boldsymbol{m}$-stellige Funktion $\boldsymbol{h}$
- erzeugt: eine $\boldsymbol{k}$-stellige Funktion $\boldsymbol{f}$ mit

$$\boldsymbol{f}(\boldsymbol{x_1}, \ldots, \boldsymbol{x_k}) := \boldsymbol{h}(\boldsymbol{g_1}(\boldsymbol{x_1}, \ldots, \boldsymbol{x_k}), \ldots, \boldsymbol{g_m}(\boldsymbol{x_1}, \ldots, \boldsymbol{x_k}))$$

Dabei ist $\boldsymbol{m} \geq \boldsymbol{1}$ und $\boldsymbol{k} \geq \boldsymbol{0}$.

Rekursionsschema $\boldsymbol{PrRek} : (\boldsymbol{g}, \boldsymbol{h}) \mapsto \boldsymbol{f}$ (für 'primitive Rekursion')

- gegeben: eine $\boldsymbol{k}$-stellige Verankerungsfunktion $\boldsymbol{g}$ und eine $(\boldsymbol{k}+\boldsymbol{2})$-stellige (Rekursions-)Funktion $\boldsymbol{h}$
- erzeugt: eine (rekursiv definierte) $(\boldsymbol{k}+\boldsymbol{1})$-stellige Funktion $\boldsymbol{f}$ mit

$$\begin{aligned} \boldsymbol{f}(\boldsymbol{x_1}, \ldots, \boldsymbol{x_k}, \boldsymbol{0}) &:= \boldsymbol{g}(\boldsymbol{x_1}, \ldots, \boldsymbol{x_k}) \\ \boldsymbol{f}(\boldsymbol{x_1}, \ldots, \boldsymbol{x_k}, \boldsymbol{y}+\boldsymbol{1}) &:= \boldsymbol{h}(\boldsymbol{x_1}, \ldots, \boldsymbol{x_k}, \boldsymbol{y}, \boldsymbol{f}(\boldsymbol{x_1}, \ldots, \boldsymbol{x_k}, \boldsymbol{y})) \end{aligned}$$

Dabei ist $\boldsymbol{k} \geq \boldsymbol{0}$.

Definition 4.1

Die Menge der primitiv rekursiven Funktionen besteht genau aus den Funktionen, die sich aus der Menge der Grundfunktionen durch endlich oft wiederholte Anwendung des Kompositionsschemas und des Rekursionsschemas bilden lassen.

Bemerkungen:

- 'Endlich oft wiederholbar' umfasst auch nullfache Wiederholung. Die Grundfunktionen selbst sind also auch primitiv rekursiv.
- Man kann die neue Klasse auch als kleinste Funktionenklasse definieren, die die Menge der Grundfunktionen enthält und unter den Anwendungen der beiden Schemata abgeschlossen ist.

Formaler kann man die Klasse auch folgendermaßen definieren:

1. Alle Grundfunktionen sind primitiv rekursiv.
2. Sind m k-stellige Funktionen g_1,...,g_m primitiv rekursiv und eine m-stellige Funktion h ebenfalls primitiv rekursiv, so ist auch die im Kompositionsschema definierte k-stellige Funktion $f = Komp(g_1, ..., g_m, h)$ primitiv rekursiv.
3. Sind eine k-stellige Funktion g und eine $(k+2)$-stellige Funktion h primitiv rekursiv, so ist auch die im Rekursionsschema definierte $(k+1)$-stellige Funktion $f = PrRek(g, h)$ primitiv rekursiv.
4. Weitere primitiv rekursive Funktionen gibt es nicht.

- Rekursionsschema ***PrRek*** definiert wirklich eine Funktion!
- Alle primitiv rekursiven Funktionen sind total!

primitive Rekursivität für charakteristische Funktionen und Prädikate:

'charakteristische Funktion' $\boldsymbol{ch_T} : \mathbb{N} \to \mathbb{N}$ für Menge $\boldsymbol{T} \subseteq \mathbb{N}$:

$$\boldsymbol{ch_T}(\boldsymbol{x}) = \begin{cases} \mathbf{1} \text{ für } \boldsymbol{x} \in \boldsymbol{T} \\ \mathbf{0} \text{ sonst} \end{cases}$$

Analog für Teilmengen aus $\mathbb{N}^{\boldsymbol{k}}$...

Damit:
$\boldsymbol{T} \subseteq \mathbb{N}^{\boldsymbol{k}}$ heißt primitiv rekursiv, wenn $\boldsymbol{ch_T}$ primitiv rekursiv ist...

Alternativer Zugang zu Mengen: über 'Prädikate' $\boldsymbol{P} : \mathbb{N} \to \{\text{true}, \text{false}\}$

Ein Prädikat $\boldsymbol{P}$ heißt primitiv rekursiv, wenn die folgende Funktion $\boldsymbol{f_P}$ primitiv rekusiv ist:

$$\boldsymbol{f_P}(\boldsymbol{x}) = \begin{cases} \mathbf{1} \text{ falls } \boldsymbol{P}(\boldsymbol{x}) = \text{ true} \\ \mathbf{0} \text{ falls } \boldsymbol{P}(\boldsymbol{x}) = \text{ false} \end{cases}$$

Mit Hilfe der charakteristischen Funktionen:

Für primitiv rekursive $\boldsymbol{P_1}$, $\boldsymbol{P_2}$ sind auch $\boldsymbol{P_1} \vee \boldsymbol{P_2}$, $\boldsymbol{P_1} \wedge \boldsymbol{P_2}$ und $\neg \boldsymbol{P_1}$ primitiv rekursiv!

Beispiel 4.2

(a) *Betrachte* $\boldsymbol{c_1^{(0)} := Komp(Z, S)}$*:*
Stelligkeiten passen zusammen,
Resultat $\boldsymbol{c_1^{(0)}}$ *ist nullstellig, mit* $\boldsymbol{c_1^{(0)}() = 1}$
d.h. $\boldsymbol{c_1^{(0)}}$ *ist nullstellige Konstante mit Wert* **1**
Analog: nullstellige Konstante $\boldsymbol{c_m^{(0)}}$ *mit Wert* $\boldsymbol{m}$ *durch*

$$\boldsymbol{c_m^{(0)} = \underbrace{Komp(Komp(...Komp}_{m}(Z, S), ..., S), S)}$$

(b) *Betrachte* $\boldsymbol{c_m^{(1)} := PrRek(c_m^{(0)}, pr_2^2)}$
Stelligkeiten passen wieder zusammen,
Resultat $\boldsymbol{c_m^{(1)}(0) = c_m^{(0)}() = m}$, $\boldsymbol{c_m^{(1)}(n+1) = c_m^{(1)}(n) = ... = m}$
d.h. $\boldsymbol{c_m^{(1)}}$ *ist einstellige Konstante mit Wert* $\boldsymbol{m}$

Fortsetzung von 4.2:

(c) Betrachte $\boldsymbol{c_m^{(k)}} := \boldsymbol{Komp}(\boldsymbol{pr_1^k}, \boldsymbol{c_m^{(1)}})$:
$\boldsymbol{k}$-stellige konstante Funktion mit Wert $\boldsymbol{m}$

(d) Identität $\boldsymbol{id} : \mathbb{N} \to \mathbb{N}$ ist primitiv rekursiv:
$\boldsymbol{id} = \boldsymbol{pr_1^1}$, d.h. $\boldsymbol{id}$ ist Grundfunktion

(e) Rekursive Definition der Addition:

$$\begin{aligned} \boldsymbol{add}(\boldsymbol{x}, \boldsymbol{0}) &= \boldsymbol{x} \\ \boldsymbol{add}(\boldsymbol{x}, \boldsymbol{y}+\boldsymbol{1}) &= \boldsymbol{S}(\boldsymbol{add}(\boldsymbol{x}, \boldsymbol{y})) \end{aligned}$$

Damit

$$\boldsymbol{add} = \boldsymbol{PrRek}(\boldsymbol{id}, \boldsymbol{Komp}(\boldsymbol{pr_3^3}, \boldsymbol{S}))$$

Fortsetzung von 4.2:

(f) Rekursive Definition der Multiplikation:

$$\begin{aligned} \boldsymbol{mult}(\boldsymbol{x}, \boldsymbol{0}) &= \boldsymbol{0} \\ \boldsymbol{mult}(\boldsymbol{x}, \boldsymbol{y}+\boldsymbol{1}) &= \boldsymbol{add}(\boldsymbol{x}, \boldsymbol{mult}(\boldsymbol{x}, \boldsymbol{y})) \end{aligned}$$

Damit

$$\boldsymbol{mult} = \boldsymbol{PrRek}(\boldsymbol{c}_{\boldsymbol{0}}^{(\boldsymbol{1})}, \boldsymbol{Komp}(\boldsymbol{pr}_{\boldsymbol{1}}^{\boldsymbol{3}}, \boldsymbol{pr}_{\boldsymbol{3}}^{\boldsymbol{3}}, \boldsymbol{add}))$$

(g) Vorgängerfunktion $\boldsymbol{V} : \mathbb{N} \to \mathbb{N}$ mit

$$\begin{aligned} \boldsymbol{V}(\boldsymbol{0}) &= \boldsymbol{0} \\ \boldsymbol{V}(\boldsymbol{n}+\boldsymbol{1}) &= \boldsymbol{n} \end{aligned}$$

Damit

$$\boldsymbol{V} = \boldsymbol{PrRek}(\boldsymbol{Z}, \boldsymbol{pr}_{\boldsymbol{1}}^{\boldsymbol{2}})$$

Fortsetzung von 4.2:

(h) modifizierte Subtraktion $\boldsymbol{sub} : \mathbb{N} \to \mathbb{N}$ mit

$$\begin{aligned} \boldsymbol{sub}(\boldsymbol{x}, \boldsymbol{y}) &= \begin{cases} \boldsymbol{x} - \boldsymbol{y}, & \text{falls } \boldsymbol{x} \geq \boldsymbol{y} \\ \boldsymbol{0}, & \text{falls } \boldsymbol{x} < \boldsymbol{y} \end{cases} \\ \text{d.h.} \quad \boldsymbol{sub}(\boldsymbol{x}, \boldsymbol{0}) &= \boldsymbol{x} \\ \boldsymbol{sub}(\boldsymbol{x}, \boldsymbol{y} + \boldsymbol{1}) &= \boldsymbol{V}(\boldsymbol{sub}(\boldsymbol{x}, \boldsymbol{y})) \end{aligned}$$

Damit

$$\boldsymbol{sub} = \boldsymbol{PrRek}(\boldsymbol{id}, \boldsymbol{Komp}(\boldsymbol{pr}_{\boldsymbol{3}}^{\boldsymbol{3}}, \boldsymbol{V}))$$

(i) Vorzeichenfunktion $\boldsymbol{sg} : \mathbb{N} \to \mathbb{N}$ mit

$$\begin{aligned} \boldsymbol{sg}(\boldsymbol{0}) &= \boldsymbol{0} \\ \boldsymbol{sg}(\boldsymbol{n} + \boldsymbol{1}) &= \boldsymbol{1} \end{aligned}$$

Damit

$$\boldsymbol{sg} = \boldsymbol{PrRek}(\boldsymbol{Z}, \boldsymbol{c}_{\boldsymbol{1}}^{(\boldsymbol{2})})$$

Beispiel 4.3

Abgeleitetes Erzeugungsschema: beschränkte Nullstellensuche

- *Gegeben totale Funktion* $f : \mathbb{N}^{k+1} \to \mathbb{N}$
- *Definiere totale Funktion* $g : \mathbb{N}^{k+1} \to \mathbb{N}$ *durch*

$$\begin{aligned} g(x_1, ..., x_k, x_{k+1}) &= \min(\\ &\{x_{k+1}\} \cup \{n \in \mathbb{N} |\ n < x_{k+1} \textit{ und } f(n, x_1, ..., x_k) = 0\}) \end{aligned}$$

Dann gilt

$$\begin{aligned} g(x_1, ..., x_k, 0) &= 0 \\ g(x_1, ..., x_k, y+1) &= g(x_1, ..., x_k, y) \\ &\quad + s_g(f(g(x_1, ..., x_k, y), x_1, ..., x_k)) \end{aligned}$$

Daher gilt: wenn f *primitiv rekursiv ist, ist auch* g *primitiv rekursiv.*

Fortsetzung von 4.3:

Beispiel zur Anwendung der beschränkten Minimalisierung:

- Sei $f : \mathbb{N}^2 \to \mathbb{N}$ definiert durch

$$f(n, x) := sub(x, mult(n, n)) \quad = x - n^2$$

- Sei g die nach dem eben beschriebenen Verfahren zu f definierte primitiv rekursive Funktion, d.h.

$$g(x, y) = \min(\ \{y\} \cup \{n \in \mathbb{N} \mid n < y \text{ und } n^2 \geq x\}\ )$$

- Sei $h : \mathbb{N}^2 \to \mathbb{N}$ definiert durch $h(x) := g(x, x+1)$.
  Dann ist auch h primitiv rekursiv, und es gilt:

$$h(x) = \min\{n \in \mathbb{N} \mid n^2 \geq x\}$$

  Für eine Quadratzahl x ist $h(x) = \sqrt{x}$, generell:

$$h(x) = \left\lceil \sqrt{x} \right\rceil$$

Äquivalenzbeweis Turing-berechenbar $\Rightarrow$ WHILE-berechenbar:
Verschlüsselung des Bandes (= Vektor von Zeichen) durch eine Zahl

Jetzt: Verschlüsselung von *Zahlen*vektoren durch eine Zahl

Betrachte

$$\begin{aligned} & & \boldsymbol{c_2(n)} & := & \frac{\boldsymbol{n \cdot (n+1)}}{\boldsymbol{2}} = \sum_{\boldsymbol{i=0}}^{\boldsymbol{n}} \boldsymbol{i} \\ \text{also} & & \boldsymbol{c_2(0)} & = & \boldsymbol{0} \\ & & \boldsymbol{c_2(n+1)} & = & \boldsymbol{c_2(n) + n + 1 = S(add(n, c_2(n)))} \end{aligned}$$

Damit

$$\boldsymbol{c_2 = PrRek(Z, Komp(add, S))}$$

und

$$\boldsymbol{\langle x, y \rangle = c_2(x + y) + x}$$

Also: Die Cantorsche Bijektion ist primitiv rekursiv!

Umkehrung dieser Bijektion:

- Sei $\boldsymbol{z}$ gegeben
- Gesucht $\boldsymbol{x}$ und $\boldsymbol{y}$ mit $\boldsymbol{z} = \langle \boldsymbol{x}, \boldsymbol{y} \rangle$
- Bestimme $\boldsymbol{n}$ mit $\boldsymbol{c_2}(\boldsymbol{n}) \leq \boldsymbol{z} < \boldsymbol{c_2}(\boldsymbol{n}+\mathbf{1})$
- Dann $\boldsymbol{x} = \boldsymbol{z} - \boldsymbol{c_2}(\boldsymbol{n})$ und $\boldsymbol{y} = \boldsymbol{n} - \boldsymbol{x}$.

Beispiel: $\boldsymbol{z} = \langle \boldsymbol{x}, \boldsymbol{y} \rangle = \mathbf{18} \Rightarrow \boldsymbol{n} = \mathbf{5}, \boldsymbol{x} = \mathbf{3}, \boldsymbol{y} = \mathbf{2}$.

Betrachte Komponenten der Umkehrung, d.h.

$$p_1 := pr_1^2 \circ \langle \cdot, \cdot \rangle^{-1} \text{ und } p_2 := pr_2^2 \circ \langle \cdot, \cdot \rangle^{-1}$$

D.h. $p_1(\langle x, y \rangle) = x$, $p_2(\langle x, y \rangle) = y$ und $\langle p_1(z), p_2(z) \rangle = z$.

p_1 und p_2 sind primitiv rekursiv:

- Zu z bestimme kleinstes n mit $z < c_2(n+1)$:
  Dazu beschränkte Nullstellensuche mit

$$\begin{aligned} f(n, z) &:= sub(1, sub(c_2(n+1), z)) = 1 - (c_2(n+1) - z) \\ g(z, t) &:= \min(\{t\} \cup \{n \in \mathbb{N} \mid n < t \text{ und } f(n, z) = 0\}) \\ &= \min(\{t\} \cup \{n \in \mathbb{N} \mid n < t \text{ und } z < c_2(n+1)\}) \end{aligned}$$

  gesuchte Zahl n ist nicht größer als z, also $n := g(z, z)$.
- Dann

$$\begin{aligned} x &= p_1(z) = z - c_2(g(z, z)) \\ y &= p_2(z) = n - x = g(z, z) - (z - c_2(g(z, z))) \end{aligned}$$

Damit sowohl p_1 als auch p_2 primitiv rekursiv.

Definiere primitiv rekursive Bijektion $\langle \cdot \rangle : \mathbb{N}^k \to \mathbb{N}$ wie folgt:

$$\langle n_1, n_2, ..., n_k \rangle := \langle n_1, \langle n_2, ..., \langle n_{k-1}, n_k \rangle ... \rangle \rangle$$

Dabei sei $k \geq 1$.

Betrachte Komponenten $d_i^{(k)}$ der Umkehrfunktion, d.h.

$$d_i^{(k)}(\langle n_1, n_2, ..., n_k \rangle) = n_i$$

Alle $d_i^{(k)}$ sind ebenfalls primitiv rekursiv:

$$\begin{aligned}
d_1^{(k)}(n) &= p_1(n) \\
d_2^{(k)}(n) &= p_1(p_2(n)) \\
&... \\
d_{k-1}^{(k)}(n) &= p_1(p_2(p_2(...p_2(n)...) \\
d_k^{(k)}(n) &= p_2(p_2(...p_2(n)...))
\end{aligned}$$

Satz 4.4

Eine Funktion ist primitiv rekursiv genau dann, wenn sie LOOP-berechenbar ist.

Beweis '$\Rightarrow$'

durch Induktion über den Aufbau der primitiv rekursiven Funktionen:

(a) Grundfunktionen sind LOOP-berechenbar:

Funktion	LOOP-Programm
Z	$x_0 := x_0 + 0$
S	$x_0 := x_1 + 1$
pr_j^k	$x_0 := x_j + 0$

(b) Kompositionsschema $f = h(g_1, \ldots, g_m)$ mit k-stelligen g_i:

Setze Programme H, G_1,..., G_m für h, g_1,..., g_m
zu Programm F für f zusammen durch:

ν sei maximaler Index der Variablen in H, G_1,..., G_m

Dann arbeite F wie folgt:

- Sicherung der Startwerte $n_1, \ldots, n_k$ von $x_1, \ldots, x_k$ in $x_{\nu+1}, \ldots, x_{\nu+k}$
- Ausführung von G_1, d.h. Berechnung von $j_1 = g_1(n_1, \ldots, n_k)$
- Sicherung des Resultates j_1 durch $x_{\nu+k+1} := x_0$
- Alle Variablen mit Index $\leq \nu$ wieder auf 0 setzen
- Werte n_i für $1 \leq i \leq k$ wieder in x_i speichern
- Ausführung von G_2, d.h. Berechnung von $j_2 = g_2(n_1, \ldots, n_k)$
- usw...
- Am Ende: Berechnung von $h(j_1, \ldots, j_m)$

(c) Rekursionsschema $f = \mathit{PrRek}(g, h)$, d.h.

$$\begin{aligned} f(x_1, \ldots, x_k, 0) &:= g(x_1, \ldots, x_k) \\ f(x_1, \ldots, x_k, y{+}1) &:= h(x_1, \ldots, x_k, y, f(x_1, \ldots, x_k, y)) \end{aligned}$$

Gegeben seien also LOOP-berechenbare g und h

Arbeitsweise eines LOOP-Programms für f:

$x_0 := g(x_1, \ldots, x_k)$;
$t := 0$;
LOOP x_{k+1} DO $x_0 := h(x_1, \ldots, x_k, t, x_0)$; $t := t + 1$ END;

Umsetzung in LOOP-Programm (analog zur Komposition):

- Werte $x_1, \ldots, x_k$ werden gesichert (in geeigneten Variablen)
- Variablen, die bei g oder h genutzt werden, werden wieder gelöscht
- Damit: Keine 'Seiteneffekte' bei den Berechnungen von g und h
- t ist Variable, die ansonsten ungenutzt ist

LOOP-Programm berechnet offenbar f !

Beweis ’$\Leftarrow$’

durch Induktion über den Aufbau der LOOP-berechenbaren Funktionen:

- $\boldsymbol{P}$ sei LOOP-Programm für $\boldsymbol{r}$-stellige Funktion $\boldsymbol{f}$
- in $\boldsymbol{P}$ vorkommende Variablen seien $\boldsymbol{x_0}$,$\boldsymbol{x_1}$,..., $\boldsymbol{x_k}$ mit $\boldsymbol{k} \geq \boldsymbol{r}$.
- Kodiere $\boldsymbol{x_0}$,$\boldsymbol{x_1}$,..., $\boldsymbol{x_k}$ in einer einzelnen Zahl mit Bijektion $\langle \cdot \rangle : \mathbb{N}^{\boldsymbol{k}+1} \to \mathbb{N}$ und Umkehrungen $\boldsymbol{d_i^{(k+1)}}$
- Konstruiere (induktiv) primitiv rekursive Funktion $\boldsymbol{g_P} : \mathbb{N} \to \mathbb{N}$, die das Verhalten von $\boldsymbol{P}$ auf allen Variablen $\boldsymbol{x_0}$, $\boldsymbol{x_1}$,..., $\boldsymbol{x_k}$ simuliert
- d.h. zu gegebenen Anfangswerten $\boldsymbol{a_0}$, $\boldsymbol{a_1}$,..., $\boldsymbol{a_k}$ und Endwerten $\boldsymbol{b_0}$, $\boldsymbol{b_1}$,..., $\boldsymbol{b_k}$ (nach Ablauf von $\boldsymbol{P}$) der Variablen $\boldsymbol{x_0}$, $\boldsymbol{x_1}$,..., $\boldsymbol{x_k}$ gelte

$$\boldsymbol{g_P}(\langle \boldsymbol{a_0}, \boldsymbol{a_1}, \ldots, \boldsymbol{a_k} \rangle) = \langle \boldsymbol{b_0}, \boldsymbol{b_1}, \ldots, \boldsymbol{b_k} \rangle \quad (*)$$

(a) Falls P die Form $x_i := x_j \pm c$ hat, so setze

$$\begin{aligned} g_P(z) \quad := \langle \quad & d_1^{(k+1)}(z), \ldots, d_i^{(k+1)}(z), \\ & d_{j+1}^{(k+1)}(z) \pm c, \\ & d_{i+2}^{(k+1)}(z), \ldots, d_{k+1}^{(k+1)}(z) \quad \rangle \end{aligned}$$

(Achtung: Variable x_i enspricht $i+1$-ter Komponente!)

Damit ist g_P primitiv rekursiv und hat Eigenschaft $(*)$

(b) Falls P die Form $Q; R$ hat:

- Programme Q und R sind kürzer als P
- mit Induktionsannahme:
  es gibt primitiv rekursive Funktionen g_Q für Q und g_R für R
- Definiere g_P durch $g_P(z) = g_R(g_Q(z))$.

Damit ist g_P primitiv rekursiv und hat Eigenschaft $(*)$

(c) Falls P die Form `LOOP` x_i `DO` Q `END` hat:

- mit Induktionsannahme:
  es gibt primitiv rekursive Funktion g_Q für Q mit $(*)$
- Definiere mit primitiver Rekursion zweistellige Funktion h:

$$h(0, x) = x; \qquad h(n+1, x) = g_Q(h(n, x))$$

- $h(n, x)$ simuliert in x (für $\langle x_0, x_1, ..., x_k \rangle$) n Anwendungen von Q
- Setze $g_P(x) := h(d_{i+1}^{(k+1)}(x), x)$

Damit ist wieder g_P primitiv rekursiv mit Eigenschaft $(*)$

Also: Für alle LOOP-Programme P existiert primitiv rekursives g_P mit

$$g_P(\langle a_0, a_1, ..., a_k \rangle) = \langle b_0, b_1, ..., b_k \rangle$$

Wird $f : \mathbb{N}^r \to \mathbb{N}$ durch P berechnet, so ergibt sich

$$f(n_1, ..., n_r) = d_1^{(k+1)}(g_P(\langle 0, n_1, ..., n_r, \underbrace{0, ..., 0}_{k-r} \rangle))$$

d.h. f ist auch primitiv rekursiv.

Neues Erzeugungsschema: μ-Operator, $\mu : f \mapsto g$

- gegeben: $(k+1)$-stellige (evtl. partielle!) Funktion f
- erzeugt: k-stellige Funktion g mit

$$\begin{aligned} g(x_1, ..., x_k) \quad = \quad & \min\{n \in \mathbb{N} \mid f(n, x_1, ..., x_k) = 0 \text{ und} \\ & \text{für alle } m < n \text{ ist } f(m, x_1, ..., x_k) \text{ definiert und} \\ & \text{für alle } m < n \text{ ist } f(m, x_1, ..., x_k) > 0\} \end{aligned}$$

oder kürzer

$$g(x_1, ..., x_k) = (\mu n)[f(n, x_1, ..., x_k) = 0]$$

Anmerkungen:

- Schreibweise: $g := \mu f$
- Mimimum der leeren Menge ist undefiniert, damit:
- g ist evtl. nicht total:
  - wenn $f(n, x_1, ..., x_k)$ nie Null wird, ist $g(x_1, ..., x_k)$ undefiniert
  - wenn $f(n, x_1, ..., x_k) = 0$, aber $f(m, x_1, ..., x_k)$ undefiniert ist für ein $m < n$, so ist $g(x_1, ..., x_k)$ ebenfalls undefiniert
- Zur Auswertung von g: Werte $f(n, x_1, ..., x_k)$ für $n = 0, 1, 2, ...$ der Reihe nach ausrechnen...

Definition 4.5

Die Menge der μ-rekursiven (auch: partiell rekursiven) Funktionen besteht genau aus den Funktionen, die sich aus der Menge der Grundfunktionen durch endlich oft wiederholte Anwendung des Kompositionsschemas, des Rekursionsschemas und des μ-Operators bilden lassen.

Satz 4.6

Die Klasse der μ-rekursiven Funktionen stimmt mit der Klasse der WHILE- (TURING-) berechenbaren Funktionen überein.

Beweis:

- analog zum Beweis 'LOOP-berechenbar $\Leftrightarrow$ primitiv rekursiv'
- erweitert um μ-Operator und WHILE-Schleife

Alter Beweis also nur noch um Fälle (d) erweitert:

Fall (d) für '$\Rightarrow$': $g = \mu f$ für μ-rekursive Funktion f, also

$$g(x_1, ..., x_k) = (\mu n)[f(n, x_1, ..., x_k) = 0]$$

Induktiv: WHILE-Programm Q zur Berechnung von f existiert...

Passendes WHILE-Programm für g:

$x_0 := 0;$
$y := f(0, x_1, ..., x_k);$
`WHILE` y `DO`
  $x_0 := S(x_0);$
  $y := f(x_0, x_1, ..., x_k);$
`END;`

(Berechnung von f über Q wieder mit Sicherung der Variablen...)

Fall (d) für '$\Leftarrow$':
WHILE-Programm P habe die Form `WHILE` x_i `DO` Q `END`

In P vorkommende Variablen seien x_0, x_1,..., x_k mit $k \geq r$.

Wieder gesucht: μ-rekursive Funktion $g_P : \mathbb{N} \dashrightarrow \mathbb{N}$ mit

$$g_P(\langle a_0, a_1, ..., a_k \rangle) = \langle b_0, b_1, ..., b_k \rangle$$

für gegebene Anfangswerte a_0, a_1,..., a_k und Endwerte b_0, b_1,..., b_k (nach Ablauf von P) der Variablen x_0, x_1,..., x_k

Induktiv: Entsprechende (μ-rekursive!) Funktion g_Q für Q existiert...

Definiere (mit primitiver Rekursion) zweistellige Funktion h:

$$h(0, x) = x; \qquad h(n+1, x) = g_Q(h(n, x))$$

D.h. h beschreibt wieder n Anwendungen von Q

Dann: $(\mu n)[d_{i+1}^{(k+1)}(h(n, x)) = 0]$ ist minimale Wiederholungszahl von Q bis zum Erreichen einer Nullstelle, also setze

$$g_P(x) = h(\quad (\mu n)[d_{i+1}^{(k+1)}(h(n, x)) = 0]\ ,\ x\ )$$

(Rest des Beweises zu 4.4 wird unverändert übernommen!)

Satz 4.7 (Kleene)

Für jede k-stellige μ-rekursive Funktion f
gibt es zwei $(k+1)$-stellige primitiv rekursive Funktionen p und q,
so dass sich f darstellen lässt als

$$f(x_1, \ldots, x_k) := p(x_1, \ldots, x_k, \mu q(x_0, x_1, \ldots, x_k))$$

Hierbei ist μq durch die Anwendung des μ-Operators auf q
entstanden und steht abkürzend für

$$(\mu x_0)[q(x_0, x_1, \ldots, x_n) = 0]$$

Beweis:

f μ-rekursiv
$\Rightarrow$ WHILE-Programm P für f
$\Rightarrow$ WHILE-Programm P' für P mit nur einer WHILE-Schleife
$\Rightarrow$ μ-rekursive Funktion f' für P' mit nur einem μ-Operator

Zwei übliche Beweise:

- Betrachte $\boldsymbol{A} : \mathbb{N}^{\mathbf{2}} \rightarrow \mathbb{N}$ (Ackermann, 1926, Rozsa Peter, 1955) mit

$$\begin{aligned} \boldsymbol{A}(\mathbf{0}, \boldsymbol{y}) &= \boldsymbol{y} + \mathbf{1} \\ \boldsymbol{A}(\boldsymbol{x}+\mathbf{1}, \mathbf{0}) &= \boldsymbol{A}(\boldsymbol{x}, \mathbf{1}) \\ \boldsymbol{A}(\boldsymbol{x}+\mathbf{1}, \boldsymbol{y}+\mathbf{1}) &= \boldsymbol{A}(\boldsymbol{x}, \boldsymbol{A}(\boldsymbol{x}+\mathbf{1}, \boldsymbol{y})) \end{aligned}$$

  $\boldsymbol{A}$ wächst schnell: $\boldsymbol{A}(\mathbf{4}, \mathbf{2})$ hat bereits $\approx$20000 Dezimalstellen...
  Nachweis der WHILE-Berechenbarkeit: relativ einfach
  Nachweis der Nicht-LOOP-Berechenbarkeit: komplex
  (vgl Schöning, S.108-113)
- Alternativer Beweis:
  Verwende Diagonalisierung als Beweis-/Konstruktionsprinzip

Betrachte 10-elementiges Alphabet $\boldsymbol{A}$:

$$\boldsymbol{A} := \{\ + \ - \ := \ ;\ \texttt{LOOP DO END}\ \boldsymbol{x}\ \mathbf{0}\ \mathbf{1}\ \}$$

$\Rightarrow$ jedes LOOP-Programm ist als Wort über $\boldsymbol{A}$ schreibbar!

(dabei Variable $\boldsymbol{x_i}$ durch '$\boldsymbol{x}$ $\boldsymbol{bin}(\boldsymbol{i})$' und Konstante $\boldsymbol{c}$ durch $\boldsymbol{bin}(\boldsymbol{c})$)

Bilde Liste $\boldsymbol{P_0}, \boldsymbol{P_1}, \boldsymbol{P_2}, \boldsymbol{P_3}, \ldots$ *aller* LOOP-Programme durch

- Sortierung nach Länge des Programmes und
- bei gleicher Länge: Alphabetische Sortierung (mit beliebiger Ordnung auf $\boldsymbol{A}$)

Lemma 5.1

Die folgende Funktion $\boldsymbol{g} : \mathbb{N}^{\mathbf{2}} \to \mathbb{N}$ *ist WHILE-(Turing-) berechenbar:*

$\boldsymbol{g}(\boldsymbol{i}, \boldsymbol{n}) :=$ *der Wert der einstelligen Zahlenfunktion, die von* $\boldsymbol{P_i}$ *berechnet wird, bei Eingabe von* $\boldsymbol{n}$.

Beweis:

- aus i kann P_i berechnet werden (mit einer Turingmaschine):

```
m := 0
FOR k := 0, 1, 2, ... DO
  FOR w ∈ A* mit |w| = k DO
    IF w codiert LOOP-Programm THEN
      IF m = i THEN RETURN w END
      m := m + 1
    END
  END
END
```

- Wenn P_i und n gegeben sind: Die Berechnung von P_i auf $(0, n, 0, 0, \ldots)$ in den Variablen $x_0, x_1, \ldots$ kann simuliert werden (wieder mit einer Mehrband-Turingmaschine)
- Dabei z.B. P_i auf einen Band gespeichert und alle Variablen, die P_i nutzt, auf einem anderen Band
- Es ist nicht möglich, für jede Variable ein eigenes Band vorzusehen (da die Zahl der Variablen von P_i abhängt)

Also: aus i und n kann das Resultat von P_i auf n berechnet werden
$\Rightarrow$ g ist berechenbare (totale) Funktion!

Satz 5.2

Die Funktion g *aus Lemma 5.1 ist nicht LOOP-berechenbar. Außerdem gibt es eine totale Funktion* $f : \mathbb{N} \to \mathbb{N}$*, die WHILE-berechenbar, aber nicht LOOP-berechenbar ist.*

Beweis: Benutze folgende 'Diagonalkonstruktion' einer Funktion f:

$$f(n) := g(n, n) + 1$$

Damit:

- g total, also auch f total
- g WHILE-berechenbar, also auch f WHILE-berechenbar
- Wäre g LOOP-berechenbar, so wäre auch f LOOP-berechenbar.

Annahme: f sei LOOP-berechenbar.

- Dann gibt es ein LOOP-Programm P_j, das f berechnet.
- Damit gilt $f(n) = g(j, n)$ für alle $n \in \mathbb{N}$
- Insbesondere auch $f(j) = g(j, j)$
- Nach Definition von f gilt jedoch $f(j) = g(j, j) + 1$
- Widerspruch...

Damit: f nicht LOOP-berechenbar,
also auch g nicht LOOP-berechenbar

Idee der Diagonalisierung: Betrachte Diagramm aller Werte $g(i, n)$:

$i \backslash n$	0	1	2	3	$\cdots$
P_0	$g(0,0)$	$g(0,1)$	$g(0,2)$	$g(0,3)$	
P_1	$g(1,0)$	$g(1,1)$	$g(1,2)$	$g(1,3)$	
P_2	$g(2,0)$	$g(2,1)$	$g(2,2)$	$g(2,3)$	
P_3	$g(3,0)$	$g(3,1)$	$g(3,2)$	$g(3,3)$	
$\vdots$					

$f(n)$	$g(0,0)+1$	$g(1,1)+1$	$g(2,2)+1$	$g(3,3)+1$	$\cdots$

f wird so definiert, dass f sich in der Diagonale von jeder der Funktionen unterscheidet.

Beispiel 6.1

Sei $E = \{e_1, e_2, \ldots, e_n\}$ ein endliches Alphabet.
Definiere Bijektion $\nu : \mathbb{N} \to E^*$ durch Sortieren:

- *Wörter in E^* nach Länge sortiert*
- *Wörter gleicher Länge alphabetisch sortiert*

i	0	1	2	$\ldots$	n	$n+1$	$n+2$	$\ldots$	n^2+n	n^2+n+1	$\ldots$
$\nu(i)$	ε	e_1	e_2	$\ldots$	e_n	e_1e_1	e_1e_2	$\ldots$	e_ne_n	$e_1e_1e_1$	$\ldots$

- *Abbildung $\nu : \mathbb{N} \to E^*$ ist eine Bijektion*
- *Umkehrabbildung $\nu^{-1} : E^* \to \mathbb{N}$ ist definiert und ebenfalls bijektiv.*

- Ist $\boldsymbol{f} : \mathbb{N} \dashrightarrow \mathbb{N}$ einstellige Zahlenfunktion,
  so ist $\Gamma(\boldsymbol{f}) := \boldsymbol{\nu} \circ \boldsymbol{f} \circ \boldsymbol{\nu}^{-1}$ eine Wortfunktion:

$$\Gamma(\boldsymbol{f}) = \boldsymbol{\nu} \circ \boldsymbol{f} \circ \boldsymbol{\nu}^{-1} : \boldsymbol{E}^* \xrightarrow{\nu^{-1}} \mathbb{N} \overset{\boldsymbol{f}}{\dashrightarrow} \mathbb{N} \xrightarrow{\nu} \boldsymbol{E}^*$$

- Ist $\boldsymbol{g} : \boldsymbol{E}^* \dashrightarrow \boldsymbol{E}^*$ eine Wortfunktion,
  so ist $\Delta(\boldsymbol{f}) = \boldsymbol{\nu}^{-1} \circ \boldsymbol{g} \circ \boldsymbol{\nu}$ einstellige Zahlenfunktion
- Es ist $\Delta \circ \Gamma(\boldsymbol{f}) = \boldsymbol{\nu}^{-1} \circ \left(\boldsymbol{\nu} \circ \boldsymbol{f} \circ \boldsymbol{\nu}^{-1}\right) \circ \boldsymbol{\nu} = \boldsymbol{f}$
  und analog $\Gamma \circ \Delta(\boldsymbol{g}) = \boldsymbol{g}$

$\Rightarrow \Delta$ und Γ definieren Bijektion zwischen:

- Menge aller Zahlenfunktionen $\boldsymbol{f} : \mathbb{N} \dashrightarrow \mathbb{N}$ und
- Menge aller Wortfunktionen $\boldsymbol{g} : \boldsymbol{E}^* \dashrightarrow \boldsymbol{E}^*$

Satz 6.2

1. *Ist* $f : \mathbb{N} \dashrightarrow \mathbb{N}$ *eine (Turing-)berechenbare Zahlenfunktion, so ist* $\nu \circ f \circ \nu^{-1}$ *eine (Turing-)berechenbare Wortfunktion.*
2. *Ist* $g : E^* \dashrightarrow E^*$ *eine (Turing-)berechenbare Wortfunktion, so ist* $\nu^{-1} \circ g \circ \nu$ *eine (Turing-)berechenbare Zahlenfunktion.*

Beweis von (1):

- Sei $f : \mathbb{N} \dashrightarrow \mathbb{N}$ berechenbare Zahlenfunktion.
- Betrachte entsprechende Turingmaschine ***TM***:
  TM berechne aus $bin(n)$ den Wert $bin(f(n))$
  (für n aus dem Definitionsbereich von f)
- Aufgabe: konstruiere Turingmaschine ***TM′***,
  die aus w das Wort $\nu \circ f \circ \nu^{-1}(w)$ berechnet,
  (für w im Definitionsbereich von $\nu \circ f \circ \nu^{-1}$)

Vorgehensweise von ***TM′***:

- Erst aus Wort ***w*** das Wort ***bin***$(\nu^{-1}(\boldsymbol{w}))$ bestimmen
- Auf ***bin***$(\nu^{-1}(\boldsymbol{w}))$ die Turingmaschine ***TM*** anwenden.
- Auf (binär kodiertes) $\boldsymbol{f}(\nu^{-1}(\boldsymbol{w}))$ wiederum ν anwenden, mit Resultat $\nu(\boldsymbol{f}(\nu^{-1}(\boldsymbol{w})))$.
- Das Ganze geht mit Turingmaschinen, da die Funktion, die Wörter ***w*** auf ***bin***$(\nu^{-1}(\boldsymbol{w}))$ abbildet, sicher berechenbar ist, ebenso ihre Umkehrung.

Beweis von (2): analog...

Damit:
Berechenbarkeitsbegriffe für Zahlen / für Funktionen gleichwertig.

Im Folgenden analog Funktionen $\mathbb{N} \dashrightarrow \boldsymbol{E}^*$ und $\boldsymbol{E}^* \dashrightarrow \mathbb{N}$ verwendet...

Lemma 6.3

Die Menge aller von Turingmaschinen berechneten Funktionen über einem gegebenen Alphabet E *ist abzählbar, es gibt zu jedem* E *eine totale Funktion* h *mit Definitionsbereich* $\{0,1\}^*$*, deren Bild alle(!) berechenbaren Funktionen umfasst.*

Konstruiere dazu Liste aller Turing-berechenbaren Funktionen

Dazu: (beachte: viele Feinheiten der Konstuktion sind willkürlich!)

- Beschränkung auf deterministische Turingmaschinen $TM = (S, E, A, \delta, s_0, \square, F)$
- Festes Alphabet E mit $\{0, 1, \#\} \subseteq E$
- $\delta(s, a)$ undefiniert für alle $a \in A$ und Endzustände $s \in F$
- $\delta(s, a)$ definiert für alle $a \in A$ und Nicht-Endzustände $s \notin F$
- Falls bei Eingabe w mit Konfiguration $us_f v$ hält:

$$f_{TM}(w) := \text{ das längste Präfix } \in E^* \text{ von } v$$

O.B.d.A.:

- $\boldsymbol{S} = \{\boldsymbol{s_0}, \boldsymbol{s_1}, \ldots, \boldsymbol{s_n}\}$ mit Startzustand $\boldsymbol{s_0}$
- Endzustände am Listenende: $\boldsymbol{F} = \{\boldsymbol{s_{n-e+1}}, \ldots, \boldsymbol{s_n}\}$ mit $\boldsymbol{e} = |\boldsymbol{F}|$
- $\boldsymbol{A} = \{\boldsymbol{a_0}, \boldsymbol{a_1}, \ldots, \boldsymbol{a_k}\}$
- $\boldsymbol{E} \subset \boldsymbol{A}$ mit $\boldsymbol{E} = \{\boldsymbol{a_0}, \boldsymbol{a_1}, \ldots, \boldsymbol{a_{l-1}}\}$ für $\boldsymbol{l} = |\boldsymbol{E}|$
- $\square = \boldsymbol{a_d}$ für ein $\boldsymbol{d}$

Übergänge der Form '$\delta(\boldsymbol{s_i}, \boldsymbol{a_j})$ enthält $(\boldsymbol{s_t}, \boldsymbol{a_m}, \boldsymbol{B})$' beschreibbar durch

$$\Delta_{\boldsymbol{i},\boldsymbol{j}} := \#\#\boldsymbol{bin}(\boldsymbol{i})\#\boldsymbol{bin}(\boldsymbol{j})\#\boldsymbol{bin}(\boldsymbol{t})\#\boldsymbol{bin}(\boldsymbol{m})\#\boldsymbol{bin}(\boldsymbol{b})$$

mit $\boldsymbol{b} = \mathbf{0}$, wenn $\boldsymbol{B} = \boldsymbol{L}$, $\boldsymbol{b} = \mathbf{1}$, wenn $\boldsymbol{B} = \boldsymbol{N}$ und $\boldsymbol{b} = \mathbf{2}$, wenn $\boldsymbol{B} = \boldsymbol{R}$.

$\Rightarrow$ ***TM*** beschreibbar durch $\boldsymbol{n}$, $\boldsymbol{e}$, $\boldsymbol{k}$, $\boldsymbol{l}$, $\boldsymbol{d}$ und Angabe der $\delta(\boldsymbol{s_i}, \boldsymbol{a_j})$,
d.h. als Wort über $\{\mathbf{0}, \mathbf{1}, \#\}$ in folgender Form:

$$\boldsymbol{bin}(\boldsymbol{n})\#\boldsymbol{bin}(\boldsymbol{e})\#\boldsymbol{bin}(\boldsymbol{k})\#\boldsymbol{bin}(\boldsymbol{l})\#\boldsymbol{bin}(\boldsymbol{d})\#\Delta_{\mathbf{0},\mathbf{0}}\ldots\Delta_{\boldsymbol{n-e},\boldsymbol{k}}$$

Codiere z.B. $\mathbf{0} \mapsto \mathbf{00}$, $\mathbf{1} \mapsto \mathbf{01}$ und $\# \mapsto \mathbf{11}$
$\Rightarrow$ *jede* Turingmaschine durch ein $\boldsymbol{w} \in \{\mathbf{0}, \mathbf{1}\}^*$ beschreibbar.

Ziel: Aufzählung aller berechenbaren Wortfunktionen, dazu

- Zu jedem $w \in \{0,1\}^*$ definiere M_w durch

$$M_w := \begin{cases} M, & \text{falls } w \text{ eine Beschreibung der Maschine } M \text{ ist .} \\ M^{ud}, & \text{sonst} \end{cases}$$

- Dabei sei M^{ud} beliebig gewählt, z.B.

$$M^{ud} := (\{s_0, s_1\}, E, E \cup \{\Box\}, \delta, \Box, \{s_1\})$$

  mit $\delta(s_0, a) = (s_0, a, N)$ für $a \in A$ (d.h. M^{ud} hält nie an...)
- h_w sei die von M_w berechnete Wortfunktion.

$\Rightarrow$ w ist 'Programm' für die Funktion $h_w : E^* \dashrightarrow E^*$

$\Longrightarrow$ Abbildung $w \mapsto h_w$ ist 'Programmiersprache'

Definition 6.4

Sei E *ein Alphabet.*
Eine Notation für die berechenbaren Funktionen $g : E^* \dashrightarrow E^*$ *ist eine surjektive Funktion*

$$h' : \{0,1\}^* \to \{ \text{berechenbare Wortfunktionen über } E^* \}$$

- *Jedem* $w \in \{0,1\}^*$ *wird eine berechenbare Wortfunktion zuordnet.*
- *Zu jeder berechenbaren Wortfunktion* g *gibt es ein Wort* $w \in \{0,1\}^*$ *derart, dass* $h'(w)$ *gerade die Funktion* g *ist.*

Wir schreiben oft auch h'_w *für die Funktion* $h'(w)$.

Mögliche Eigenschaften von Notationen:

utm-Eigenschaft

Die Wortfunktion, die Wörter $w\#x$ für $w \in \{0,1\}^*$ und $x \in E^*$ auf $h'_w(x)$ abbildet, ist berechenbar.

smn-Eigenschaft

Ist $g : E^* \dashrightarrow E^*$ eine berechenbare Wortfunktion, so gibt es eine totale berechenbare Wortfunktion r derart, dass für alle $x \in \{0,1\}^*$ und alle $y \in E^*$ gilt:

$$g(x\#y) = h'_{r(x)}(y)$$

Satz 6.5

Die von uns oben definierte Funktion h*, die jedes Wort* w *aus* $\{0,1\}^*$ *auf eine berechenbare Funktion* $h_w : E^* \dashrightarrow E^*$ *abbildet, ist eine Notation der berechenbaren Wortfunktionen.*
h *hat zudem die utm-Eigenschaft und die smn-Eigenschaft.*

Beweis:

Nach Konstruktion ist h Notation der berechenbaren Wortfunktionen:

- h_w ist berechenbare Funktion für jedes w.
- Jede (normierte) Turingmaschine wird durch ein w kodiert.

Zur utm-Eigenschaft: Bilde Turingmaschine U wie folgt:

- Bei Eingabe $w\#x$ bestimmt U die von w codierte Maschine M (d.h. direkt $M = M_w$ oder aber $M = M^{ud}$)
- Danach simuliert U die Maschine M auf der Eingabe x

f_U ist dann die für die utm-Eigenschaft notwendige Funktion!

Zur smn-Eigenschaft:

- Sei $g : E^* \dashrightarrow E^*$ eine berechenbare Wortfunktion.
- Sei M_g Turingmaschine, die g berechnet (fest gegeben!)

Betrachte eine Turingmaschine R, die wie folgt arbeitet:

- R liest ihre Eingabe x und modifiziert dann (eine Codierung von) M_g zu (der Codierung eines) M_g^x mit folgender Arbeitsweise:
  Bei einer Eingabe y schreibt M_g^x zunächst das Wort $x\#$ links vor das Wort y und arbeitet dann weiter wie M_g
- x wird dabei in der Zustandsmenge von M_g^x codiert
- R erzeugt dann als Ausgabe eine Codierung von M_g^x
- M_g wird in der Zustandsmenge von R codiert

R erzeugt also aus einer Eingabe x (die Codierung von) M_g^x mit

$$h_{f_R(x)}(y) = f_{M_g^x}(y) = f_{M_g}(x\#y) = g(x\#y)$$

f_R ist also das in der smn-Eigenschaft geforderte (totale!) r.

Anwendung der smn-Eigenschaft in JAVA:

Betrachte Funktion zu Addition zweier Zahlen in JAVA aus Kapitel 1.2, wende smn-Eigenschaft für ersten Parameter ***m*** an:

```
class SMN {
public static void main(String args[]) {
System.out.println(¨import java.math.BigInteger;¨);
System.out.println(¨class Addition {¨);
System.out.println(¨public static void main(String args[]){¨);
System.out.println(¨BigInteger m=new BigInteger(\”¨
                +  args[0] + ¨\”);¨ );
System.out.println(¨BigInteger n=new BigInteger(args[0]);¨);
System.out.println(¨while(n.compareTo(BigInteger.ZERO)>0){¨);
System.out.println(¨n = n.subtract(BigInteger.ONE);¨);
System.out.println(¨m = m.add(BigInteger.ONE);¨);
System.out.println(¨}¨);
System.out.println(¨System.out.println(m.toString());¨);
System.out.println(¨}¨);
System.out.println(¨}¨);
} }
```

’utm’ steht für Universelle Turing-Maschine

- Die Maschine $\boldsymbol{U}$ kann jedes ’Turingprogramm’ $\boldsymbol{w}$ auf jeder Eingabe $\boldsymbol{x}$ simulieren
- Analogie zum realen Computer und realen Programmiersprachen: Beliebige Programme können auf beliebigen Eingaben gestartet werden...
- Jede ’gute’ Programmiersprache sollte erlauben, Programme auf Eingaben anzuwenden...

Zur smn-Eigenschaft:

- $\boldsymbol{r}$ erlaubt es, das Turingprogramme für $\boldsymbol{g}$ zu spezialisieren
- aus einer ’Eingabe’ $\boldsymbol{x}$ wird eine ’Konstante’ $\boldsymbol{x}$ im Programm $\boldsymbol{M_g^x}$
- Kürzel ’smn’: nur historische Gründe...

Vergleichbarkeit von Notationen:

- eine Notation h'' ist in eine Notation h' übersetzbar, wenn es eine totale(!) berechenbare Wortfunktion

  $$u : \{0,1\}^* \to \{0,1\}^*$$

  gibt mit

  $$h''_w(x) = h'_{u(w)}(x)$$

  für alle $w \in \{0,1\}^*$ und $x \in E^*$, d.h. $h''_w = h'_{u(w)}$
- Jedes 'Programm' w der 'Programmiersprache' h'' kann also dann automatisch in ein 'Programm' der 'Sprache' h' übersetzt werden.
- u heißt Übersetzungsfunktion.

Notationen h' und h'' heißen äquivalent, wenn man jede in die andere übersetzen kann.

Satz 6.6 (Äquivalenzsatz von Rogers)

Eine Notation für die berechenbaren Wortfunktionen ist genau dann zu der von uns definierten Notation $\boldsymbol{h}$ *äquivalent, wenn sie die utm-Eigenschaft und die smn-Eigenschaft hat.*

Beweis:

- Sei $\boldsymbol{h}$ wie oben
- Sei $\boldsymbol{h'}$ weitere Notation für berechenbaren Wortfunktionen.

'$\Rightarrow$': $\boldsymbol{h}$ und $\boldsymbol{h'}$ seien äquivalent, mit $\boldsymbol{h_w} = \boldsymbol{h'_{u(w)}}$ und $\boldsymbol{h'_w} = \boldsymbol{h_{u'(w)}}$

(a) utm-Eigenschaft für $\boldsymbol{h}$: $\boldsymbol{f_h}$ mit $\boldsymbol{f_h}(\boldsymbol{w}\#\boldsymbol{x}) := \boldsymbol{h_w}(\boldsymbol{x})$ ist berechenbar

Definiere $\boldsymbol{f'}$ durch $\boldsymbol{f'}(\boldsymbol{w}\#\boldsymbol{x}) := \boldsymbol{f_h}(\boldsymbol{u'}(\boldsymbol{w})\#\boldsymbol{x})$

Dann $\boldsymbol{f'}$ berechenbar und

$$\boldsymbol{h'_w}(\boldsymbol{x}) = \boldsymbol{h_{u'(w)}}(\boldsymbol{x}) = \boldsymbol{f_h}(\boldsymbol{u'}(\boldsymbol{w})\#\boldsymbol{x}) = \boldsymbol{f'}(\boldsymbol{w}\#\boldsymbol{x})$$

d.h. $\boldsymbol{f'}$ zeigt utm-Eigenschaft für $\boldsymbol{h'}$

(b) Sei $g : E^* \dashrightarrow E^*$ irgendeine berechenbare Wortfunktion.

smn-Eigenschaft für h: r mit $h_{r(x)}(y) := g(x\#y)$ ist berechenbar

Dann ist $u \circ r$ berechenbar (und total) mit

$$g(x\#y) = h_{r(x)}(y) = h'_{u\circ r(x)}(y)$$

D.h. h' hat auch die smn-Eigenschaft.

'$\Leftarrow$': h' besitze ebenfalls die utm- und die smn-Eigenschaft:

utm-Eigenschaft für h: f_h mit $f_h(w\#x) = h_w(x)$ berechenbar

Wende smn-Eigenschaft für h' auf f_h an, mit entsprechendem r':

$$h_w(x) = f_h(w\#x) = h'_{r'(w)}(x)$$

Damit Übersetzung von h nach h' mit r'

Analog: utm-E. für h' und smn-E. für $h \Rightarrow h'$ in h übersetzbar.

Damit: h und h' äquivalent!

Satz 6.7

Eine Notation h' *für die berechenbaren Wortfunktionen ist genau dann zu der von uns definierten Notation* h *äquivalent, wenn sie die utm-Eigenschaft und die folgende Eigenschaft hat:*

(effektive Komposition) Es gibt eine totale berechenbare Wortfunktion $r : \{0,1\}^*\#\{0,1\}^* \to \{0,1\}^*$ *derart, dass für alle* v, w *aus* $\{0,1\}^*$ *und alle* x *aus* E^* *gilt:*

$$h'_v(h'_w(x)) = h'_{r(v\#w)}(x)$$

r *bestimmt also aus zwei 'Programmen'* v, w *ein 'Programm'* $r(v\#w)$ *für die Komposition der Funktionen* h'_v *und* h'_w.

(Ohne Beweis.)

- Alle 'vernünftigen' Notationen der berechenbaren Wortfunktionen sind äquivalent (mit 'vernünftig' = 'utm+smn' = 'utm+effKomp')
- Jede zu h äquivalente Notation heißt Standardnotation

Betrachte Teilmenge $A \subseteq E^*$:

- Die charakteristische Funktion

$$ch_A : E^* \to \{0, 1\}$$

ist definiert durch

$$ch_A(w) := \begin{cases} 1, & w \in A \\ 0, & w \notin A \end{cases}$$

- Die eingeschränkte charakteristische Funktion

$$ch'_A : E^* \dashrightarrow \{0, 1\}$$

ist definiert durch

$$ch'_A(w) := \begin{cases} 1, & w \in A \\ \text{undefiniert}, & w \notin A \end{cases}$$

Wichtig: ch_A ist total; ch'_A ist partiell mit Definitionsbereich A

Definition 7.1

1. *Eine Teilmenge* $A \subseteq E^*$ *heißt* ***entscheidbar****, wenn ihre charakteristische Funktion* $ch_A : E^* \to \{0,1\}$ ***berechenbar*** *ist.*
2. *Eine Teilmenge* $A \subseteq E^*$ *heißt* ***semi-entscheidbar****, wenn ihre eingeschränkte charakteristische Funktion* $ch'_A : E^* \dashrightarrow \{0,1\}$ ***berechenbar*** *ist.*

Für Mengen $A \subseteq \mathbb{N}^k$ *werden die Begriffe analog definiert. Der Begriff '****A*** *ist unentscheidbar' bedeutet im folgenden das Gleiche wie '****A*** *ist nicht entscheidbar'.*

A ist entscheidbar, wenn ein Algorithmus (d.h. deterministische Turingmaschine) existiert, der bei Eingabe eines beliebigen w anhält und dann sagt, ob $w \in A$ oder $w \notin A$.

A ist semi-entscheidbar, wenn ein Algorithmus mit folgender Eigenschaft existiert:

- falls $w \in A$, muss er nach endlich vielen Schritten anhalten und anzeigen, dass $w \in A$ gilt,
- falls $w \notin A$, so hält der Algorithmus nie an.

Satz 7.2

*Eine Sprache **A** ist genau dann entscheidbar, wenn sowohl **A** als auch ihr Komplement $E^* \setminus A$ semi-entscheidbar sind.*

Beweis:

'$\Rightarrow$': Sei A entscheidbar,
d.h. ch_A ist berechenbar (mit einer Maschine M)

Konstruiere M' wie folgt:

- Bei Eingabe w berechnet M' zunächst $ch_A(w)$ (mittels M)
- Ist das Resultat **1**, so hält M' und akzeptiert w
- Ist das Resultat **0**, so startet M' eine Endloschleife

$\Rightarrow$ M' berechnet ch'_A, d.h. A ist semi-entscheidbar

Vertausche Rollen von **1** und **0** bei M':
$\Rightarrow$ M' berechnet jetzt $ch'_{E^* \setminus A}$, d.h. $E^* \setminus A$ ist semi-entscheidbar

'$\Leftarrow$': Gegeben Maschinen M', M'':
M' berechne ch'_A, M'' berechne $ch'_{E^* \setminus A}$

Konstruiere neue Turingmaschine M wie folgt:

- w sei Eingabe für M
- M simuliert abwechselnd einen Rechenschritt von M' auf w und einen Schritt von M'' auf w
- Hält M' an, so hält auch M und akzeptiert w
- Hält M'' an, so hält auch M und verwirft w

Bei Eingabe von w hält genau eine der Maschine M' und M'', d.h.

- M hält in jedem Fall an und
- M berechnet ch_A

Definition 7.3

Eine Sprache $A \subseteq E^*$ heißt rekursiv aufzählbar, falls A leer ist oder Bildbereich einer totalen berechenbaren Funktion $f : \mathbb{N} \to E^*$ ist:

$$A := \{f(0), f(1), ...\}$$

Für Teilmengen $A \subseteq \mathbb{N}^k$ wird der Begriff analog definiert.

Satz 7.4

Eine Sprache A aus E^* ist genau dann semi-entscheidbar, wenn sie rekursiv aufzählbar ist.

Beweis:

Sei $\boldsymbol{A}$ rekursiv aufzählbar.

Fall (1), $\boldsymbol{A} = \emptyset$:

Dann $\boldsymbol{ch_A(x) = 0}$ *für alle* $\boldsymbol{x}$*, also* $\boldsymbol{A}$ *entscheidbar*

Fall (2), $\boldsymbol{A} = \{\boldsymbol{f(0), f(1), ...}\}$ mit totalem berechenbaren $\boldsymbol{f} : \mathbb{N} \to \boldsymbol{E}^*$:

Betrachte (berechenbares!) $\boldsymbol{g}$ *definiert über*

```
INPUT(w);
FOR n = 0,1,2,3,... DO
    IF f(n) = w THEN OUTPUT(1) END;
END.
```

$\boldsymbol{w} \in \boldsymbol{A} \Rightarrow \boldsymbol{g(w) = 1}$

$\boldsymbol{w} \notin \boldsymbol{A} \Rightarrow \boldsymbol{g(w)}$ *undefiniert*

$\Longrightarrow$ $\boldsymbol{A}$ *semi-entscheidbar*

Sei nun A semi-entscheidbar, $A \neq \emptyset$

- M sei eine Turingmaschine, die ch'_A berechnet.
- a sei beliebig gewähltes Element von A.

Betrachte (berechenbares!) g definiert über

```
INPUT(n);
k := p1(n); l := p2(n); w := ν(k);
IF  Angesetzt auf w stoppt M
                nach höchstens l Schritten mit Ausgabe von 1
THEN OUTPUT(w) ELSE OUTPUT(a)
END.
```

- p_1, p_2: LOOP-berechenbare Umkehrfunktionen der Cantorschen Bijektion c zwischen $\mathbb{N}^2$ und $\mathbb{N}$
- n codiert (bijektiv!) zwei Zahlen k und l
- k codiert (bijektiv!) Worte w

$\Longrightarrow$ Algorithmus testet

- für alle möglichen Wörter $w \in E^*$ und
- für alle möglichen Laufzeiten l,
- ob M bei Eingabe von w nach l Schritten anhält.

1. Die berechnete Funktion g ist total
2. Stets gilt $g(n) \in A$
3. Für jedes $w \in A$ hält M nach l_w Schritten für ein $l_w \in \mathbb{N}$, d.h. $g(n_w) = w$ für n_w mit $p_2(n_w) = l_w$ und $\nu p_1(n_w) = w$

Damit $A = \{g(n) \mid n \in \mathbb{N}\}$

Bezeichnung der Verfahrenweise: dove-tailing.

Statt $A \subseteq E^*$ ist auch $A \subseteq \mathbb{N}^k$ möglich, damit:

Für $A \subseteq E^*$ sind folgende Aussagen äquivalent:

- A ist Sprache vom Typ 0
- A wird von einer nichtdeterministischen Turingmaschine erkannt
- A wird von einer deterministischen Turingmaschine erkannt
- Es gibt eine deterministische Turingmaschine, die bei Eingabe eines Wortes $w \in E^*$ genau dann nach endlich vielen Schritten anhält, wenn $w \in A$ ist
- A ist semi-entscheidbar, d.h. die eingeschränkte charakteristische Funktion ch'_A ist berechenbar
- A ist Definitionsbereich einer berechenbaren Funktion
- A ist rekursiv aufzählbar
- A ist leer oder Wertebereich einer totalen berechenbaren Funktion

Äquivalent dazu ist auch

- A ist Wertebereich einer (partiellen) berechenbaren Funktion

(Beweis der noch nicht bewiesenen Äquivalenzen als Übung...)

- Sei h die Notation der berechenbaren Wortfunktionen aus 6.3
- M_w sei die durch $w \in \{0,1\}^*$ bezeichnete Turingmaschine
- h_w sei die durch $w \in \{0,1\}^*$ berechnete Wortfunktion

Definition 7.5

Das spezielle Halteproblem oder Selbstanwendbarkeitsproblem ist die Menge

$$\begin{aligned} K &:= \{w \in \{0,1\}^* \mid M_w \textit{ angesetzt auf } w \\ &\qquad \textit{hält nach endlich vielen Schritten an}\} \\ &= \{w \in \{0,1\}^* \mid h_w(w) \textit{ ist definiert }\} \end{aligned}$$

Satz 7.6

Das spezielle Halteproblem K ist rekursiv aufzählbar, aber nicht entscheidbar.

K ist rekursiv aufzählbar:

- $g(w\#v) := h_w(v)$ ist berechenbar (utm-Eigenschaft)
- also auch f mit $f(w) := g(w\#w) = h_w(w)$
- $w \in K \Leftrightarrow f(w)$ definiert

Annahme: K sei entscheidbar.

- Dann insbesondere $E^* \setminus K$ semi-entscheidbar
- Sei TM eine Turingmaschine, die genau dann auf w hält, wenn $w \in E^* \setminus K$
- Sei x ein Codewort für TM, d.h.

$$h_x(w) \text{ ist definiert } \Leftrightarrow w \in E^* \setminus K$$

Betrachte $h_x(x)$:

$$\begin{aligned} x \in K &\Leftrightarrow TM \text{ hält nicht auf } x \\ &\Leftrightarrow h_x(x) \text{ nicht definiert} \\ &\Leftrightarrow x \notin K \text{ (Widerspruch...)} \end{aligned}$$

Also war die Annahme falsch, und K ist *nicht* entscheidbar.

zugrundeliegende Beweisidee: Alle “Programme” listen und an der Diagonale Widerspruch einbauen:

	0	1	2	3	⋯
M_{w_0}	$h_{w_0}(w_0)$	$h_{w_0}(w_1)$	$h_{w_0}(w_2)$	$h_{w_0}(w_3)$	⋯
M_{w_1}	$h_{w_1}(w_0)$	$h_{w_1}(w_1)$	$h_{w_1}(w_2)$	$h_{w_1}(w_3)$	⋯
M_{w_2}	$h_{w_2}(w_0)$	$h_{w_2}(w_1)$	$h_{w_2}(w_2)$	$h_{w_2}(w_3)$	⋯
M_{w_3}	$h_{w_3}(w_0)$	$h_{w_3}(w_1)$	$h_{w_3}(w_2)$	$h_{w_3}(w_3)$	⋯
⋮	⋮	⋮	⋮	⋮	

Betrachte nur, ob das Resultat definiert ist (wg. rek.-aufzählbar):

	0	1	2	3	⋯
M_{w_0}	*def*	?	?	?	⋯
M_{w_1}	?	*udef*	?	?	⋯
M_{w_2}	?	?	*udef*	?	⋯
M_{w_3}	?	?	?	*def*	⋯
⋮	⋮	⋮	⋮	⋮	
K	$\in$	$\notin$	$\notin$	$\in$	
$E^* \setminus K$	$\notin$	$\in$	$\in$	$\notin$	

Damit passt $E^* \setminus K$ zu keiner der Zeilen darüber.

Bemerkung 7.7

nur utm-Eigenschaft der Programmiersprache h *notwendig:*

Ist h' *eine Notation für die berechenbaren Wortfunktionen mit der utm-Eigenschaft, so ist die Menge*

$$K = \{w \in \{0,1\}^* \mid h'_w(w) \text{ ist definiert}\}$$

rekursiv aufzählbar, aber nicht entscheidbar.

Definition 7.8

Seien $A, B \subseteq E^*$ *Sprachen.*
Dann heißt A *auf* B *reduzierbar (‘*$A \leq B$*’), wenn es eine totale berechenbare Funktion* $f : E^* \to E^*$ *gibt, so dass für alle* $x \in E^*$ *gilt*

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

- $A \leq B$: Lösung von Problem A durch Lösung von Problem B (daher: A auf B ‘reduziert’)
- B ist in der Regel ‘schwerer’ lösbar als A, aber evtl. ‘Lösung’ für B schon bekannt....

Satz 7.9

Sei die Sprache A *auf* B *mittels der Funktion* f *reduzierbar. Dann gilt:*

- *Ist* B *entscheidbar, so ist auch* A *entscheidbar.*
- *Ist* A *nicht entscheidbar, so ist auch* B *nicht entscheidbar.*
- *Ist* B *rekursiv-aufzählbar, so ist auch* A *rekursiv-aufzählbar.*
- *Ist* A *nicht rek.-aufzählbar, so ist auch* B *nicht rek.-aufzählbar.*

Beweis: f sei berechenbar und reduziere A auf B

(a) B sei entscheidbar.

Dann ch_B berechenbar, Komposition $ch_B \circ f$ berechenbar und

$$ch_A(x) = 1 \iff x \in A \iff f(x) \in B \iff ch_B(f(x)) = 1$$
$$ch_A(x) = 0 \iff x \notin A \iff f(x) \notin B \iff ch_B(f(x)) = 0$$

Also $ch_A = ch_B \circ f$ berechenbar, und A entscheidbar

(b) B sei rekursiv-aufzählbar

Dann: ch'_B berechenbar, Komposition $ch'_B \circ f$ berechenbar und

$$ch'_A(x) = 1 \iff x \in A \iff f(x) \in B \iff ch'_B(f(x)) = 1$$
$$ch'_A(x) = \text{undef.} \iff x \notin A \iff f(x) \notin B \iff ch'_B(f(x)) = \text{undef.}$$

Also $ch'_A = ch'_B \circ f$ berechenbar, und A rekursiv-aufzählbar

Definition 7.10

Das allgemeine Halteproblem ist die Sprache

$$\boldsymbol{H} = \{\ \boldsymbol{w}\$\boldsymbol{x} \in \{\mathbf{0},\mathbf{1}\}^*\{\$\}\boldsymbol{E}^* \quad | \ \boldsymbol{M_w}\ \textit{angesetzt auf}\ \boldsymbol{x}\ \textit{hält an}\}$$

Dabei sei $ *irgendein Symbol, das in* $\boldsymbol{E}$ *nicht vorkommt.*

Satz 7.11

Das allgemeine Halteproblem ist rekursiv aufzählbar, aber nicht entscheidbar.

Beweis:

$\boldsymbol{H}$ semi-entscheidbar / rekursiv aufzählbar: sofort mit utm-Eigenschaft

$\boldsymbol{H}$ unentscheidbar: $\boldsymbol{K}$ ist auf $\boldsymbol{H}$ reduzierbar mit folgendem $\boldsymbol{f}$

$$\boldsymbol{f}(\boldsymbol{w}) := \boldsymbol{w}\$\boldsymbol{w}$$

$\boldsymbol{f}$ ist sicherlich total und berechenbar.

Satz 7.12 (Rice)

Sei $\boldsymbol{R}$ *die Klasse aller Turing-berechenbaren Wortfunktionen über dem Alphabet* $\boldsymbol{E}$*.*
Sei $\boldsymbol{S}$ *eine echte, nichttriviale Teilmenge von* $\boldsymbol{R}$*, d.h.* $\emptyset \neq \boldsymbol{S} \neq \boldsymbol{R}$
Dann ist die folgende Sprache unentscheidbar:

$$\boldsymbol{C}(\boldsymbol{S}) = \{\boldsymbol{w} \in \{\boldsymbol{0},\boldsymbol{1}\}^* \mid \textit{die von } \boldsymbol{M_w} \textit{ berechnete Funktion } \boldsymbol{h_w} \textit{ liegt in } \boldsymbol{S}\}$$

Beweis: Sei $\boldsymbol{S}$ mit $\emptyset \neq \boldsymbol{S} \neq \boldsymbol{R}$ gegeben, sei $\boldsymbol{ud}$ die überall undefinierte (berechenbare!) Funktion.

Also $\boldsymbol{ud} \in \boldsymbol{S}$ oder $\boldsymbol{ud} \notin \boldsymbol{S}$. Wir behandeln diese beiden Fälle getrennt.

(a) Sei $ud \in S$.

Wegen $S \neq R$ gibt es $q \in R \setminus S$

Sei Q eine Turingmaschine für q.

Betrachte folgende Turingmaschine TM:

> *Bei Eingabe von $w\#x$ simuliert TM erst die Maschine M_w angesetzt auf w. Kommt diese Rechnung zu einem Ende, so soll TM anschließend Q angesetzt auf x simulieren.*

g sei die von TM berechnete Funktion.

Mit der smn-Eigenschaft von h gibt es ein totales berechenbares $r : \{0,1\}^* \to \{0,1\}^*$ mit $g(w\#x) = h_{r(w)}(x)$

Damit gilt:

- Hält M_w auf w, so ist $h_{r(w)} = q$.
- Hält M_w nicht auf w, so ist $h_{r(w)} = ud$

Dann gilt:

$$
\begin{aligned}
w \in K \quad &\Rightarrow \quad \text{Angesetzt auf } w \text{ stoppt } M_w \\
&\Rightarrow \quad h_{r(w)} \text{ ist die Funktion } q \\
&\Rightarrow \quad h_{r(w)} \text{ liegt nicht in } S \\
&\Rightarrow \quad r(w) \notin C(S).
\end{aligned}
$$

sowie

$$
\begin{aligned}
w \notin K \quad &\Rightarrow \quad \text{Angesetzt auf } w \text{ stoppt } M_w \text{ nicht} \\
&\Rightarrow \quad h_{r(w)} \text{ ist die Funktion } ud \\
&\Rightarrow \quad h_{r(w)} \text{ liegt in } S \\
&\Rightarrow \quad r(w) \in C(S).
\end{aligned}
$$

Also: r reduziert $E^* \setminus K$ auf $C(S)$, d.h $C(S)$ nicht entscheidbar

(b) Analog: Reduktion von K auf $C(S)$, d.h. $C(S)$ nicht entscheidbar

Bemerkung 7.13

(1) Der Beweis des Satzes von Rice nutzt nur die utm-Eigenschaft und die smn-Eigenschaft von $\boldsymbol{h}$*. Eine andere Formulierung ist daher:*

Ist $\boldsymbol{h'}$ *Notation der berechenbaren Wortfunktionen mit utm-Eigenschaft und smn-Eigenschaft, dann gilt:*
Für jede Teilmenge $\boldsymbol{S}$ *von* $\boldsymbol{R}$ *mit* $\emptyset \neq \boldsymbol{S} \neq \boldsymbol{R}$ *ist die Menge*

$$\boldsymbol{C}(\boldsymbol{S}) = \{\boldsymbol{w} \in \{\boldsymbol{0},\boldsymbol{1}\}^* | \boldsymbol{h'_w} \; liegt \; in \; \boldsymbol{S}\}$$

nicht entscheidbar.

Fortsetzung von 7.13:

(2) Anwendungsbeispiele:

- $S_1 = \{f \text{ berechenbar und total }\}$
  Man kann bei (realen) Programmen i.d.R. nicht entscheiden, ob sie immer halten.
- $S_2 = \{g\}$ für ein gegebenes g
  Man kann bei (realen) Programmen i.d.R. nicht entscheiden, ob sie eine exakt vorgegebene Funktion berechnen
- $S_3 = \{g \mid g(\varepsilon) = \varepsilon\}$
  Man kann bei (realen) Programmen i.d.R. nicht entscheiden, ob sie eine vorgegebene Spezifikation erfüllen (hier: $g(\varepsilon) = \varepsilon$)

Die Beispiele zeigen, dass zum Beispiel Verifikation von Software schwierig ist, sobald die Programmiersprache 'vernünftig' ist (d.h. alle berechenbaren Funktionen umfasst und utm/smn-Eigenschaften hat).

Es gibt Probleme, die ‘noch unlösbarer’ als das spezielle oder allgemeine Halteproblem sind.

Dazu gehört das Äquivalenzproblem für Turingmaschinen:

$$\boldsymbol{A} = \{\boldsymbol{v}\$\boldsymbol{w} \mid \text{die Turingmaschinen } \boldsymbol{M_v} \text{ und } \boldsymbol{M_w} \text{ berechnen dieselbe Funktion}\}$$

Man kann zwar $\boldsymbol{K}$ auf $\boldsymbol{A}$ reduzieren, aber $\boldsymbol{A}$ nicht auf $\boldsymbol{K}$.
Darauf aufbauend kann man ganze Hierarchien im Grad der Unlösbarkeit schwieriger werdender Probleme aufbauen.

Betrachte ***PCP*** = Post's Correspondence Problem:

- Eingabe: eine endliche Folge von Wortpaaren

$$(x_1, y_1),\ \ (x_2, y_2), \ldots,\ \ (x_k, y_k)$$

  wobei $k \geq 1$ und $x_i, y_i \in E^+$ seien, (also $x_i, y_i \neq \varepsilon$).
- Frage: gibt es eine Folge $\mathcal{I}$ von Indizes i_1,..., i_n aus $\{1, 2, \ldots, k\}$ mit $n \geq 1$ und mit

$$x_{i_1} \ldots x_{i_n} = y_{i_1} \ldots y_{i_n}$$

Eine derartige Folge $\mathcal{I}$ nennt man
Lösung des Korrespondenzproblems (x_1, y_1), (x_2, y_2),..., (x_k, y_k).

Beispiel 8.1

i	x_i	y_i
1	*bab*	*a*
2	*ab*	*abb*
3	*a*	*ba*

Bei der Indexfolge $\mathcal{I} = (2, 1, 3)$ *ergibt sich* ***ab bab a = abb a ba***.

Fortsetzung von 8.1:

Weiteres Beispiel, Schöning [1]:

i	x_i	y_i
1	001	0
2	01	011
3	01	101
4	10	001

Kürzeste Lösung:

01|1 0|01|10|1 0|01|001|01|1 0|01|1 0| ...001|001|01
01 1|0 01|10 1|0 01|001|01 1|0|01 1|0 01|101|001|...0|0|101

mit 66 Indizes:

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

Bemerkung 8.2

PCP *ist semi-entscheidbar (durch einfache Suche nach einer Lösung):*

- *Eingabe sei* $(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)$
- *Für alle Längen* $n = 1, 2, 3, \ldots$
  und alle Indexfolgen $i_1, i_2, \ldots, i_n$ *der Länge* n*:*

  teste, ob $x_{i_1} \ldots x_{i_n} \stackrel{?}{=} y_{i_1} \ldots y_{i_n}$
  Wenn ja, akzeptiere die Eingabe!

Jetzt: ***PCP*** ist unentscheidbar, genauer: Die Menge

$$\begin{aligned} \textbf{PCP} \quad = \quad \{((x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)) \mid \\ k \in \mathbb{N}, x_i, y_i \in E^+ \text{ für } i = 1, \ldots, k, \text{ und} \\ ((x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)) \text{ hat eine Lösung}\} \end{aligned}$$

ist nicht entscheidbar, mit Beweisweg:

(1) Selbstanwendbarkeitsproblem ***K*** ist reduzierbar auf ***MPCP*** (‘modifiziertes ***PCP***’)

(2) ***MPCP*** ist reduzierbar auf ***PCP***

Das Problem ***MPCP*** ist definiert wie folgt:

- Eingabe: wie beim Problem ***PCP***.
- Frage: gibt es eine Lösung $i_1, i_2, ..., i_n$ des ***PCP*** mit $i_1 = 1$?

Lemma 8.3

*Das Problem **MPCP** ist auf **PCP** reduzierbar.*

Beweis:

Verwende neue Symbole \$ und #, die im Alphabet E des ***MPCP***s nicht vorkommen. Für ein Wort $w = a_1 a_2 ... a_m$ aus E^+ sei

$$\begin{aligned} w^a &:= \#a_1\#a_2\#...\#a_m\# \\ w^b &:= a_1\#a_2\#...\#a_m\# \\ w^c &:= \#a_1\#a_2\#...\#a_m \end{aligned}$$

Definiere Reduktion f von ***MPCP*** auf ***PCP*** wie folgt:

$$\begin{array}{lrcl} \text{zu} & P & = & ((x_1, y_1), (x_2, y_2), ..., (x_k, y_k)) \\ \text{setze} & f(P) & := & ((x_1^a, y_1^c), (x_1^b, y_1^c), (x_2^b, y_2^c), ..., (x_k^b, y_k^c), (\$, \#\$)) \end{array}$$

- f ist offensichtlich berechenbar
- Zeige:

$$\begin{aligned} & P \text{ besitzt eine Lösung mit } i_1 = 1, \text{ d.h. } P \in MPCP \\ \Longleftrightarrow \quad & f(P) \text{ besitzt (irgend)eine Lösung, d.h. } f(P) \in PCP \end{aligned}$$

Beweis von '$\Longrightarrow$':
P besitze eine Lösung $i_1, i_2, ..., i_n$ mit $i_1 = 1$ ($MPCP$!).

Dann ist $(1, i_2+1, ..., i_n+1, k+2)$ eine Lösung für $f(P)$.

Beweis von '$\Longleftarrow$':
$f(P)$ besitze eine Lösung $i_1, i_2, ..., i_n$ aus $\{1, ..., k+2\}$.

- Aufbau der Wortpaare in $f(P)$: nur $i_1 = 1$ und $i_n = k+2$ möglich
- Falls $i_m = k+2$ für ein $m < n$: $i_1, i_2, ..., i_m$ ist ebenfalls Lösung...
- Daher o.B.d.A.: $i_m \neq k+2$ für $m < n$,
  dann auch $i_m \neq 1$ für $m < n$
- Dann ist $(1, i_2-1, ..., i_{n-1}-1)$ eine Lösung für P (d.h. bei $MPCP$)

Lemma 8.4

K *ist reduzierbar auf* ***MPCP***.

Gesucht also: totale berechenbare Funktion f mit $f(w) = P$, wobei

- w eine Turingmaschine M_w und ein Eingabewort w für M_w darstellt, und
- $P = ((x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k))$ eine Probleminstanz passend zum ***MPCP*** ist,

so dass gilt:

$(*)$ M_w angesetzt auf w hält nach endlich vielen Schritten an,
$\Longleftrightarrow$
$P = ((x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k))$ besitzt Lösung (mit $i_1 = 1$).

Sei w gegeben und $M_w = (S, E, A, \delta, s_0, \square, F)$.

- Nutze $A \cup S \cup \{\#\}$ als Alphabet für das ***MPCP***
- Erstes Wortpaar sei $(\#, \#\square s_0 w \square\#)$

$\Rightarrow$ jede Lösung muss mit diesem Paar beginnen!

Die weiteren Paare werden wie folgt konstruiert:

1. *Kopierregeln*:

$(\boldsymbol{a}, \boldsymbol{a})$ für alle $\boldsymbol{a} \in \boldsymbol{A} \cup \{\#\}$

2. *Überführungsregeln*:

Für alle Übergänge der Turingmaschine $\boldsymbol{M_w}$:

$(\boldsymbol{sa}, \boldsymbol{s'c})$	falls $\delta(\boldsymbol{s}, \boldsymbol{a}) = (\boldsymbol{s'}, \boldsymbol{c}, \boldsymbol{N})$
$(\boldsymbol{sa}, \boldsymbol{cs'})$	falls $\delta(\boldsymbol{s}, \boldsymbol{a}) = (\boldsymbol{s'}, \boldsymbol{c}, \boldsymbol{R})$
$(\boldsymbol{bsa}, \boldsymbol{s'bc})$	falls $\delta(\boldsymbol{s}, \boldsymbol{a}) = (\boldsymbol{s'}, \boldsymbol{c}, \boldsymbol{L})$, für alle $\boldsymbol{b}$ aus $\boldsymbol{A}$
$(\#\boldsymbol{sa}, \#\boldsymbol{s'}\square\boldsymbol{c})$	falls $\delta(\boldsymbol{s}, \boldsymbol{a}) = (\boldsymbol{s'}, \boldsymbol{c}, \boldsymbol{L})$
$(\boldsymbol{s}\#, \boldsymbol{s'c}\#)$	falls $\delta(\boldsymbol{s}, \square) = (\boldsymbol{s'}, \boldsymbol{c}, \boldsymbol{N})$
$(\boldsymbol{s}\#, \boldsymbol{cs'}\#)$	falls $\delta(\boldsymbol{s}, \square) = (\boldsymbol{s'}, \boldsymbol{c}, \boldsymbol{R})$
$(\boldsymbol{bs}\#, \boldsymbol{s'bc}\#)$	falls $\delta(\boldsymbol{s}, \square) = (\boldsymbol{s'}, \boldsymbol{c}, \boldsymbol{L})$, für alle $\boldsymbol{b}$ aus $\boldsymbol{A}$

3. *Löschregeln*:

$(\boldsymbol{as_f}, \boldsymbol{s_f})$ und $(\boldsymbol{s_f a}, \boldsymbol{s_f})$ für alle $\boldsymbol{a} \in \boldsymbol{A}$ und $\boldsymbol{s_f} \in \boldsymbol{F}$.

4. *Abschlussregeln*:

$(\boldsymbol{s_f}\#\#, \#)$ für alle $\boldsymbol{s_f} \in \boldsymbol{F}$.

- $f : w \mapsto P$ mit o.a. Paaren P ist berechenbar
- Zeige noch: Bedingung $(*)$ ist erfüllt.

'$\Longrightarrow$', M_w stoppt bei Eingabe w:
Dann gibt es eine Folge von Konfigurationen $(k_0, k_1, \ldots, k_t)$ mit

- $k_0 = \square s_0 w \square$ (Zusätzliche $\square$ stören nicht...)
- k_t ist Endkonfiguration (also $k_t = u s_f v$ mit $u, v \in A^*, s_f \in F$)
- $k_i \vdash k_{i+1}$ für $i = 0, \ldots, t-1$.

P besitzt dann eine Lösung für das *MPCP* der Form:

$$\begin{array}{lccccccccccccc} x: & \# & & k_0\# & k_1\# & k_2\# & \ldots & k_t\# & k_t'\# & k_t''\# & \ldots & s_f\#\# \\ y: & \#k_0\# & k_1\# & k_2\# & \ldots & & k_t\# & k_t'\# & k_t''\# & \ldots & s_f\# & \# \end{array}$$

- Folge $x_{i_1} \ldots x_{i_j}$ liegt eine Konfiguration hinter den $y_{i_1} \ldots y_{i_j}$ zurück.
- $k_t', k_t'', \ldots$ entstehen aus $k_t = u s_f v$ durch Löschen um s_f herum

'$\Longleftarrow$', $\boldsymbol{P}$ hat eine Lösung (mit $\boldsymbol{i_1 = 1}$):

- Lösungswort ist durch $\#$ in Konfigurationen $\boldsymbol{k_0, k_1, \ldots k_t}$ unterteilt
- Wegen Start mit $\boldsymbol{i_1 = 1}$: $\boldsymbol{k_0 = \square s_0 w \square}$
- Nach Konstruktion: $\boldsymbol{k_i \vdash k_{i+1}}$ oder $\boldsymbol{k_i}$ ist Endkonfiguration

Also hält $\boldsymbol{M_w}$ auf Eingabe $\boldsymbol{w}$.

Insgesamt:

- $\boldsymbol{K}$ ist auf ***MPCP*** reduzierbar
- $\boldsymbol{K}$ ist nicht entscheidbar, also auch ***MPCP*** nicht entscheidbar
- ***MPCP*** ist auf ***PCP*** reduzierbar, also auch ***PCP*** nicht entscheidbar

Satz 8.5

*Das Postsche Korrespondenzproblem **PCP** ist unentscheidbar.*

Satz 8.6

*Das **PCP** ist bereits dann nicht entscheidbar, wenn man sich auf das Alphabet $\{\mathbf{0}, \mathbf{1}\}$ beschränkt.*
Diese Problemvariante heißt $\mathbf{01}-\boldsymbol{PCP}$.

Zum Beweis zeige, dass ***PCP*** auf $\mathbf{01}-\boldsymbol{PCP}$ reduzierbar ist:

Sei $\boldsymbol{E} = \{\boldsymbol{a_1}, \ldots, \boldsymbol{a_m}\}$ das Alphabet des gegebenen ***PCP***s.
Jedem Symbol $\boldsymbol{a_j} \in \boldsymbol{E}$ ordne das Wort $\boldsymbol{a'_j} = \mathbf{01}^{\boldsymbol{j}} \in \{\mathbf{0}, \mathbf{1}\}^*$ zu;
verallgemeinere dies auf beliebige Wörter

$$\boldsymbol{w} = \boldsymbol{a_1} \ldots \boldsymbol{a_n} \in \boldsymbol{E}^+ \quad \mapsto \quad \boldsymbol{w'} = \boldsymbol{a'_1} \ldots \boldsymbol{a'_n} \in \{\mathbf{0}, \mathbf{1}\}^*$$

Dann gilt offensichtlich:

$$\begin{aligned} & (\boldsymbol{x_1}, \boldsymbol{y_1}), \ldots, (\boldsymbol{x_k}, \boldsymbol{y_k}) \text{ hat eine Lösung} \\ \iff \quad & (\boldsymbol{x'_1}, \boldsymbol{y'_1}), \ldots, (\boldsymbol{x'_k}, \boldsymbol{y'_k}) \text{ hat eine Lösung} \end{aligned}$$

Satz 9.1

Die folgenden Fragestellungen sind unentscheidbar:
Gegeben seien zwei kontextfreie Grammatiken G_1 und G_2.

1. *Ist* $L(G_1) \cap L(G_2) = \emptyset$ *?*
2. *Ist* $|L(G_1) \cap L(G_2)| = \infty$ *?*
3. *Ist* $L(G_1) \cap L(G_2)$ *kontextfrei ?*
4. *Ist* $L(G_1) \subseteq L(G_2)$ *?*
5. *Ist* $L(G_1) = L(G_2)$ *?*

Beweis:

(1) Gegeben Postsches Korrespondenzproblem P über $\{0, 1\}$ mit

$$P = \{(x_1, y_1), ..., (x_k, y_k)\}$$

Konstruiere zwei kontextfreie Grammatiken $G_1 = (N_1, T, P_1, S)$ und $G_2 = (N_2, T, P_2, S)$ wie folgt:

$$N_1 = \{S, A, B\}, \quad N_2 = \{S, R\}, \quad T = \{0, 1, \$, a_1, ..., a_k\}$$

Die Grammatik G_1 besitzt die Ableitungsregeln P_1

$$\begin{aligned} S &\rightarrow A\$B \\ A &\rightarrow a_1 A x_1 \mid \ldots \mid a_k A x_k \mid a_1 x_1 \mid \ldots \mid a_k x_k \\ B &\rightarrow y_1^r B a_1 \mid \ldots \mid y_k^r B a_k \mid y_1^r a_1 \mid \ldots \mid y_k^r a_k \end{aligned}$$

wobei mit w^r das gespiegelte Wort zu w bezeichnet wird.

Diese Grammatik G_1 erzeugt die Sprache

$$L_1 = \{a_{i_1} \ldots a_{i_n} x_{i_n} \ldots x_{i_1} \$ y_{j_1}^r \ldots y_{j_m}^r a_{j_m} \ldots a_{j_1} \mid n, m \geq 1, i_\nu, j_\mu \in \{1, .., k\}\}$$

Betrachte eine zweite Grammatik G_2 mit folgenden Regeln P_2:

$$\begin{aligned} S &\rightarrow a_1 S a_1 \mid \ldots \mid a_k S a_k \mid R \\ R &\rightarrow 0R0 \mid 1R1 \mid \$ \end{aligned}$$

Sie erzeugt die Sprache

$$L_2 = \{uv\$v^r u^r \mid u \in \{a_1, \ldots, a_k\}^*, v \in \{0, 1\}^*\}$$

Beide Sprachen sind übrigens sogar deterministisch kontextfrei.

Idee der Konstruktion:

- Jedes Wort w aus L_1 bzw. L_2 hat vier Komponenten:
  $w = ab\$c^r d^r$ mit $a, d \in \{a_1, \ldots, a_k\}^*$ und $b, c \in \{0, 1\}^*$
- L_1: Indizes der x_i bei b in a bzw. der y_i bei c in d beim *PCP* P,
  aber kein Zusammenhang zwischen a und d oder b und c.
- L_2: Übereinstimmung $a = d$ bzw. $b = c$,
  aber keinerlei Verbindung mit P
- Schnittbildung: b und c enstehen aus P (wg. L_1), sind gleich
  ($b = c$) und nutzen die gleichen Indizes ($a = d$)

Damit: P besitzt Lösung $i_1, \ldots, i_n$ genau dann, wenn $w \in L_1 \cap L_2$ für

$$w = a_{i_n} \ldots a_{i_1} x_{i_1} \ldots x_{i_n} \$ y^r_{i_n} \ldots y^r_{i_1} a_{i_1} \ldots a_{i_n}$$

Also: Abbildung $f : P \mapsto (G_1, G_2)$ reduziert *PCP* auf das Komplement des Schnittproblems.

Insgesamt: *PCP* unentscheidbar
$\Rightarrow$ Komplement des Schnittproblems unentscheidbar
$\Rightarrow$ Schnittproblem unentscheidbar.

(2) Verwende gleiche Konstruktion $P \mapsto (G_1, G_2)$ wie bei (1):

$$
\begin{aligned}
P \in PCP \quad &\Leftrightarrow \quad \text{es gibt eine Lösung } \mathcal{I} = (i_1, \ldots, i_n) \text{ für } P \\
&\Leftrightarrow \quad \text{es gibt } \infty\text{-viele Lösungen } \mathcal{I}, \mathcal{II}, \mathcal{III}, \ldots \text{ für } P \\
&\Rightarrow \quad |L_1 \cap L_2| = \infty \\
&\Rightarrow \quad L_1 \cap L_2 \neq \emptyset \quad \Leftrightarrow \quad P \in PCP
\end{aligned}
$$

(3) Wieder gleiche Konstruktion $P \mapsto (G_1, G_2)$,
setze $L := L_1 \cap L_2$

Jedes Wort in L hat Form $ab\$c^r d^r$, mit $a = d$, $b = c$ und
'Index-Übereinstimmung' zwischen a und b bzw. c und d

Also: a ober d bekannt $\Rightarrow$ $ab\$c^r d^r$ eindeutig festgelegt!

Behauptung: Falls $|L| = \infty$, dann L ist nicht kontextfrei

Annahme: L sei kontextfrei (und unendlich), d.h. Pumping-Lemma für kontextfreie Sprachen sei anwendbar, n sei die Pumpingzahl.

Wähle $ab\$c^r d^r \in L$ mit $n < |a| = |d| \leq |b| = |c|$

Betrachte beliebige Zerlegung $ab\$c^r d^r = uvwxy$ mit
$uwy = uv^0wx^0y \in L$, $|vx| \neq 0$ und $|vwx| \leq n$

Fall (1): $|u| \geq |a|$: damit a Präfix von uwy,
aber b, c, d durch a festgelegt
$\Rightarrow$ Widerspruch zu $|vx| > 0$ und $uwy \in L$.

Fall (2): $|u| < |a|$: damit $|uvwx| < |ab|$ und d Suffix von uwy,
aber a, b, c durch d festgelegt
$\Rightarrow$ wieder Widerspruch zu $|vx| > 0$ und $uwy \in L$.

Damit also: $|L| = \infty \Rightarrow L$ nicht kontextfrei

Zusammen:

$$\begin{array}{lcll} P \in PCP & \Rightarrow & |L| = \infty & \Rightarrow \quad L \text{ nicht kontextfrei} \\ P \notin PCP & \Rightarrow & L = \emptyset & \Rightarrow \quad L \text{ kontextfrei} \end{array}$$

(4) und (5): Nutze, dass L_1, L_2 deterministisch kontextfrei sind

Deterministisch kontextfreie Sprachen sind effektiv unter Komplementbildung abgeschlossen:

> *Zu G_1 und G_2 kann man effektiv Grammatiken G_1' und G_2' konstruieren mit $L(G_1') = \mathit{Komplement}(L_1)$ und $L(G_2') = \mathit{Komplement}(L_2)$.*

Dann folgt

$$L_1 \cap L_2 = \emptyset \quad \Leftrightarrow \quad L_1 \subseteq L(G_2')$$

D.h. $P \mapsto (G_1, G_2')$ reduziert ***PCP*** auf das Inklusionsproblem kontextfreier Sprachen.

Aus G_1 und G_2' konstruiere G_3 mit $L(G_3) = L_1 \cup L(G_2')$, dann

$$\begin{aligned} L_1 \cap L_2 = \emptyset \quad &\Leftrightarrow \quad L_1 \cup L(G_2') = L(G_2') \\ &\Leftrightarrow \quad L(G_3) = L(G_2') \end{aligned}$$

D.h. $P \mapsto (G_3, G_2')$ reduziert ***PCP*** auf das Äquivalenzproblem kontextfreier Sprachen.

- Inklusions- und Äquivalenzproblem kontextfreier Sprachen damit nicht entscheidbar
- $L_1 \cup L(G_2')$ ist kontextfrei, aber i.d.R. nicht deterministisch
- Das Äquivalenzproblem für deterministisch kontextfreie Sprachen ist entscheidbar (Senizergues, 1997)
- Nichtentscheidbarkeit des Äquivalenzproblems für kontextfreie Grammatiken

  $\Longrightarrow$

  Nichtentscheidbarkeit des Äquivalenzproblems für
  - nichtdeterministische Kellerautomaten,
  - kontextsensitive Grammatiken,
  - LBA's, Turingmaschinen,
  - LOOP-, WHILE-Programme
  - etc.

L_1 und L_2 waren deterministisch kontextfrei, damit sogar:

Satz 9.2

Deterministisch kontextfreie Sprachen L_1 und L_2 seien durch ihre deterministische Kellerautomaten gegeben.
Dann sind die folgenden Fragestellungen unentscheidbar:

1. *Ist $L_1 \cap L_2 = \emptyset$?*
2. *Ist $|L_1 \cap L_2| = \infty$?*
3. *Ist $L_1 \cap L_2$ kontextfrei ?*
4. *Ist $L_1 \subseteq L_2$?*

Weitere Unentscheidbarkeitsresultate (ohne Beweis):

Satz 9.3

Betrachte kontextfreie Grammatiken G.
Dann sind die folgenden Fragen unentscheidbar:

1. *Ist G mehrdeutig?*
2. *Ist das Komplement von $L(G)$ kontextfrei?*
3. *Ist $L(G)$ regulär?*
4. *Ist $L(G)$ deterministisch kontextfrei?*

Satz 9.4

Betrachte kontextfreie Sprachen L_1 und reguläre Sprachen L_2.
Es ist nicht entscheidbar, ob $L_1 = L_2$ gilt.

Satz 9.5

Das Leerheits- und das Endlichkeitsproblem für Chomsky Typ-1 Sprachen sind nicht entscheidbar.

Informal: Zusammenhänge zwischen

- (Nicht)-Berechenbarkeit
- Beweisbarkeit in der Zahlentheorie

Dazu: arithmetische Formeln als (kontextfreie) Sprache $\mathcal{F}$ über

$$\{\mathbf{0}, \mathbf{1}, \boldsymbol{x}, (,), +, *, =, \neg, \wedge, \vee, \exists, \forall\}$$

Zunächst Sprache $\mathcal{T}$ der Terme:

- Jede natürliche Zahl $\boldsymbol{i} \in \mathbb{N}$ ist ein Term
- Jede Variable $\boldsymbol{x_i}$ ist ein Term (mit $\boldsymbol{i} \in \mathbb{N}$), setze $\boldsymbol{V} := \{\boldsymbol{x_i} \mid \boldsymbol{i} \in \mathbb{N}\}$
- Sind $\boldsymbol{t_1}$, $\boldsymbol{t_2}$ Terme, dann auch $(\boldsymbol{t_1} + \boldsymbol{t_2})$ und $(\boldsymbol{t_1} * \boldsymbol{t_2})$

Sprache $\mathcal{F}$ der Formeln:

- Sind $\boldsymbol{t_1}$, $\boldsymbol{t_2}$ Terme, so ist $(\boldsymbol{t_1} = \boldsymbol{t_2})$ eine Formel
- Sind $\boldsymbol{F}$, $\boldsymbol{G}$ Formeln, so auch $\neg\boldsymbol{F}$, $(\boldsymbol{F} \wedge \boldsymbol{G})$ und $(\boldsymbol{F} \vee \boldsymbol{G})$
- Ist $\boldsymbol{x}$ Variable und $\boldsymbol{F}$ Formel, so sind $\exists\boldsymbol{x}\ \boldsymbol{F}$ und $\forall\boldsymbol{x}\ \boldsymbol{F}$ Formeln.

Dabei werden Zahlen $\boldsymbol{i}$ und die Indizes bei $\boldsymbol{x_i}$ binär notiert.

Terme können ausgewertet werden, wenn den Variablen Werte zugewiesen werden:

- Eine Belegung ϕ ist eine Abbildung $\phi : \boldsymbol{V} \to \mathbb{N}$
- ϕ kann auf Terme erweitert werden durch

$$\begin{aligned} \phi(\boldsymbol{n}) &= \boldsymbol{n} \\ \phi(\ (\boldsymbol{t_1} + \boldsymbol{t_2})\ ) &= \phi(\boldsymbol{t_1}) + \phi(\boldsymbol{t_2}) \\ \phi(\ (\boldsymbol{t_1} * \boldsymbol{t_2})\ ) &= \phi(\boldsymbol{t_1}) \cdot \phi(\boldsymbol{t_2}) \end{aligned}$$

Beispiel: Mit $\phi(\boldsymbol{x_1}) = \boldsymbol{5}$ und $\phi(\boldsymbol{x_2}) = \boldsymbol{6}$ ist $\phi(\ (\boldsymbol{x_1} * (\boldsymbol{2} + \boldsymbol{x_2}))\ ) = \boldsymbol{40}$

ϕ kann auf Formeln erweitert werden;
$\phi(\boldsymbol{F})$ ist dabei einer der Wahrheitswerte '***wahr***' oder '***falsch***'

- $\phi(\ (\boldsymbol{t_1} = \boldsymbol{t_2})\ ) =$ ***wahr*** genau dann, wenn $\phi(\boldsymbol{t_1}) = \phi(\boldsymbol{t_2})$
- $\phi(\neg \boldsymbol{F})$ ist ***wahr***, falls $\phi(\boldsymbol{F})$ nicht ***wahr*** ist,
- $\phi(\ (\boldsymbol{F} \wedge \boldsymbol{G})\ )$ ist ***wahr***, falls $\phi(\boldsymbol{F})$ und $\phi(\boldsymbol{G})$ ***wahr*** sind,
- $\phi(\ (\boldsymbol{F} \vee \boldsymbol{G})\ )$ ist ***wahr***, falls $\phi(\boldsymbol{F})$ oder $\phi(\boldsymbol{G})$ ***wahr*** ist,
- $\phi(\ \exists \boldsymbol{y}\ \boldsymbol{F})$ ist ***wahr***, falls $\phi'(\boldsymbol{F}) =$ ***wahr***
  bei einer beliebigen Belegung ϕ' mit $\phi'(\boldsymbol{x}) = \phi(\boldsymbol{x})$ für $\boldsymbol{x} \neq \boldsymbol{y}$,
- $\phi(\ \forall \boldsymbol{y}\ \boldsymbol{F}\ )$ ist ***wahr***, falls $\phi'(\boldsymbol{F}) =$ ***wahr***
  bei allen Belegungen ϕ' mit $\phi'(\boldsymbol{x}) = \phi(\boldsymbol{x})$ für $\boldsymbol{x} \neq \boldsymbol{y}$.

Beispiele:

- $\phi(\ (\boldsymbol{x_1} + \mathbf{1}) = \boldsymbol{x_2}\ ) =$ ***wahr*** für jedes ϕ mit $\phi(\boldsymbol{x_1}) + \mathbf{1} = \phi(\boldsymbol{x_2})$
- $\phi(\ \exists \boldsymbol{x_1}\ ((\boldsymbol{x_1} + \mathbf{1}) = \boldsymbol{x_2})\ ) =$ ***wahr*** für jedes ϕ mit $\phi(\boldsymbol{x_2}) > \mathbf{0}$
- $\phi(\ \exists \boldsymbol{x_1}\ ((\boldsymbol{x_1} + \mathbf{1}) = \boldsymbol{x_2})\ ) =$ ***falsch*** für jedes ϕ mit $\phi(\boldsymbol{x_2}) = \mathbf{0}$
- $\phi(\ \exists \boldsymbol{x_2}\ ((\boldsymbol{x_1} + \mathbf{1}) = \boldsymbol{x_2})\ ) =$ ***wahr*** für jedes beliebige ϕ

Eine Funktion $f : \mathbb{N}^k \dashrightarrow \mathbb{N}$ heißt arithmetisch repräsentierbar, wenn es eine arithmetische Formel F gibt, so dass für alle $n_0, n_1, \ldots, n_k \in \mathbb{N}$ gilt:

$$f(n_1, \ldots, n_k) = n_0 \text{ genau dann,}$$
$$\text{wenn } \phi(F) = \textbf{wahr} \text{ für alle } \phi \text{ mit } \phi(x_i) = n_i, 0 \leq i \leq k$$

Beispiele:

- Die Addition wird repräsentiert durch $x_0 = x_1 + x_2$
- Die Subtraktion wird repräsentiert durch $(x_2 + x_0 = x_1) \vee (x_0 = 0 \wedge \exists x_3(x_1 + x_3 + 1 = x_2))$

Es gilt nun der folgende Satz:

Lemma 10.1

Jede WHILE-berechenbare Funktion ist arithmetisch repräsentierbar.

Beweis: durch Induktion über den Aufbau von WHILE-Programmen. Der Beweis ist allerdings technisch etwas aufwändig...

Eine arithmetische Formel $\boldsymbol{F}$ heißt wahr (oder gültig), wenn für alle ϕ stets $\phi(\boldsymbol{F}) = \boldsymbol{wahr}$ ist, z.B

$$\boldsymbol{x_1} + \boldsymbol{x_2} = \boldsymbol{x_2} + \boldsymbol{x_1}$$
$$\neg\exists \boldsymbol{x_4}\ ((\boldsymbol{x_1} + \boldsymbol{1})^{\boldsymbol{x_4}+\boldsymbol{3}} + (\boldsymbol{x_2} + \boldsymbol{1})^{\boldsymbol{x_4}+\boldsymbol{3}} = \boldsymbol{x_3}^{\boldsymbol{x_4}+\boldsymbol{3}})$$

Mit Reduktion eines unserer Halteprobleme auf die Menge der wahren arithmetischen Formeln zeigt man:

Lemma 10.2

Die Menge der wahren arithmetischen Formeln ist nicht rekursiv aufzählbar.

Ein Beweissystem $(\boldsymbol{B}, \Psi)$ für eine Formelmenge $\boldsymbol{A}$ besteht aus

- einer Menge $\boldsymbol{B}$ von Beweisen und
- einer Interpretationsfunktion $\Psi : \boldsymbol{B} \to \boldsymbol{A}$.

Ein Beweis $\boldsymbol{b} \in \boldsymbol{B}$ ist eine Zeichenkette über einem Alphabet $\boldsymbol{E}$,

- die nach gewissen syntaktischen Regeln und Schlussschemata aufgebaut sein muss;
  diese inhaltlichen Zusammenhänge brauchen wir hier nicht!!!
- statt dessen als Minimalforderung:
  die Menge der Beweise $\boldsymbol{B}$ sei *entscheidbar* ; man sollte einer Zeichenkette ansehen können, ob sie ein Beweis ist oder nicht.

Außerdem:

- aus einem Beweis $\boldsymbol{b}$ sollte man die bewiesene Aussage $\boldsymbol{y} \in \boldsymbol{A}$ herauslesen können
- Dazu: Interpretationsfunktion Ψ mit Definitionsbereich $\boldsymbol{B}$,
- $\Psi(\boldsymbol{b}) = \boldsymbol{y}$ bedeutet, dass $\boldsymbol{y}$ durch $\boldsymbol{b} \in \boldsymbol{B}$ bewiesen wurde.
- Ψ sollte sinnvollerweise eine berechenbare Funktion sein.

Die Menge aller durch $(\boldsymbol{B}, \Psi)$ beweisbaren Formeln ist

$$\boldsymbol{Bew}(\boldsymbol{B}, \Psi) = \{\boldsymbol{y} \in \boldsymbol{E}^* \mid \text{es gibt } \boldsymbol{b} \in \boldsymbol{B} \text{ mit } \Psi(\boldsymbol{b}) = \boldsymbol{y}\}$$

Satz 10.3 (Gödelscher Unvollständigkeitssatz)

Für jedes Beweissystem $(\boldsymbol{B}, \Psi)$ für wahre arithmetische Formeln gilt: Die Menge der wahren arithmetischen Formeln ist eine echte Obermenge der Menge $\boldsymbol{Bew}(\boldsymbol{B}, \Psi)$.

Beweis:

- Menge $\boldsymbol{B}$ der Beweise entscheidbar
- Interpretationsfunktion Ψ berechenbar
- damit $\boldsymbol{Bew}(\boldsymbol{B}, \Psi)$ rekursiv aufzählbar
- jedoch: Menge der wahren arithmetischen Formeln ist nicht rekursiv aufzählbar

Da das Beweissystem nur wahre arithmetische Formeln beweisen soll ('Korrektheit'), gibt es wahre Formeln, die nicht beweisbar sind!

— nicht behandelt —

Neues Ziel: Aufwand der Lösung von Problemen untersuchen

wichtigste Ressourcen: Rechenzeit und Speicherplatz der Lösungsalgorithmen

obere Schranken für ein Problem:

- Untersuche *einen* Lösungsalgorithmus und
- schätze Bedarf an Rechenzeit und Speicherplatz ab

untere Schranken für ein Problem:

- Gültig für *alle* Lösungsalgorithmen,
- i.d.R. sehr schwer

Hier nur: Zeitkomplexität von Berechnungsproblemen

Oft trade-off zwischen Zeitbedarf und Speicherplatzbedarf bei Problemen:

- schnelle Algorithmen mit hohem Speicherplatzbedarf oder
- langsame Algorithmen auf wenig Speicherplatz.

Definition 12.1

Für eine deterministische Mehrband-Turingmaschine $\boldsymbol{M}$ *sei*

$$\boldsymbol{time_M(w)} = \textit{ Schrittzahl von } \boldsymbol{M} \textit{ bei Eingabe von } \boldsymbol{w}$$

(Schrittzahl: Zahl Übergänge von Startkonf. bis Berechnungsende)
Sei $\boldsymbol{f} : \mathbb{N} \to \mathbb{N}$ *eine totale Zahlenfunktion.*

- *Die Klasse* $\boldsymbol{FTIME(f)}$ *besteht aus allen totalen Wortfunktionen* $\boldsymbol{g} : \boldsymbol{E}^* \to \boldsymbol{E}^*$ *(über irgendeinem Alphabet* $\boldsymbol{E}$*), für die es eine deterministische Mehrband-Turingmaschine* $\boldsymbol{M}$ *mit Eingabealphabet* $\boldsymbol{E}$ *gibt, die die Funktion* $\boldsymbol{g}$ *berechnet und bei Eingabe eines Wortes* $\boldsymbol{w}$ *nach höchstens* $\boldsymbol{f}(|\boldsymbol{w}|)$ *Schritten anhält, d.h.* $\boldsymbol{time_M(w)} \leq \boldsymbol{f}(|\boldsymbol{w}|)$ *für alle Eingaben* $\boldsymbol{w}$ *erfüllt.*
- *Die Klasse* $\boldsymbol{TIME(f)}$ *besteht aus allen Sprachen* $\boldsymbol{L}$ *(über irgendeinem Alphabet* $\boldsymbol{E}$*), für die es eine deterministische Mehrband-Turingmaschine* $\boldsymbol{M}$ *mit Eingabealphabet* $\boldsymbol{E}$ *gibt, die* $\boldsymbol{L(M)} = \boldsymbol{L}$ *und* $\boldsymbol{time_M(w)} \leq \boldsymbol{f}(|\boldsymbol{w}|)$ *für alle Eingaben* $\boldsymbol{w}$ *erfüllt.*

Bemerkung 12.2

(1) Zeitbedarf wird in Abhängigkeit von $|w|$ gemessen

(2) Mehrbandmaschinen liefern realistischere Zeitkomplexitäten als Einbandmaschinen.

Jedoch: Arbeitet Mehrbandmaschine in Zeit $O(f(n))$, so gibt es äquivalente Einbandmaschine in Zeit $O(f^2(n))$

(3) M muß in allen Fällen anhalten.

Bei $w \notin L(M)$ also keine Endlosschleifen möglich,
d.h. M muß dann in einem Nichtendzustand steckenbleiben.

Fortsetzung von 12.2:

(4) Komplexität von Zahlenfunktionen $h : \mathbb{N} \to \mathbb{N}$
über korrespondierende Wortfunktion $g : E^* \to E^*$,
die für jede Zahl n das Wort $bin(n)$ auf $bin(h(n))$ abbildet

(5) Bezeichnung: Bitkomplexität: jeder Übergang ändert Konfiguration der TM 'um konstant viele Bits'

(6) Alternativ: uniforme Komplexität

hier: Zählung der 'elementaren' Rechenoperationen,
die nötig sind, um $f(n)$ aus n zu berechnen.

Beispiel: Berechne Zahl 2^{2^n} wie folgt:

Starte mit $x := 2$ und führe n-fach $x := x^2$ aus

- Uniforme Komplexität: n
- Bit-Komplexität mindestens 2^n, da 2^{2^n} bereits Länge 2^n hat!
- Uniforme Komplexität nicht immer angemessen.
- Turingmaschinenmodell / Bitkomplexität i.d.R. passender!

Definition 12.3

- ***FP*** *sei die Menge aller in Polynomzeit berechenbaren Wortfunktionen:*

$$\boldsymbol{FP} = \bigcup_{\boldsymbol{p} Polynom} \boldsymbol{FTIME}(\boldsymbol{p})$$

- ***P*** *sei die Menge aller in Polynomzeit lösbaren Probleme:*

$$\boldsymbol{P} = \bigcup_{\boldsymbol{p} Polynom} \boldsymbol{TIME}(\boldsymbol{p})$$

Analog:

Klasse ***EXP*** der in exponentieller Zeit lösbaren Probleme,
mit Zeitschranken $\boldsymbol{2^{c \cdot n}}$

Definition 12.4

Für eine nichtdeterministische Turingmaschine $\boldsymbol{M}$ *sei*

$$\boldsymbol{ntime_M(w)} = \max\{ \quad \textit{Schrittzahl irgendeiner Rechnung von } \boldsymbol{M} \textit{ bei Eingabe von } \boldsymbol{w}\}$$

- *Sei* $\boldsymbol{f} : \mathbb{N} \to \mathbb{N}$ *eine totale Zahlenfunktion.*
  Die Klasse $\boldsymbol{NTIME(f)}$ *besteht aus allen Sprachen* $\boldsymbol{L}$ *über irgendeinem Alphabet* $\boldsymbol{E}$ *derart, dass es eine nichtdeterministische Mehrband-Turingmaschine* $\boldsymbol{M}$ *gibt mit* $\boldsymbol{L(M)} = \boldsymbol{L}$ *und* $\boldsymbol{ntime_M(w)} \leq \boldsymbol{f}(|\boldsymbol{w}|)$ *für alle Eingaben* $\boldsymbol{w}$.
- $\boldsymbol{NP} = \bigcup_{\boldsymbol{p} Polynom} \boldsymbol{NTIME(p)}$

M nichtdeterministisch:

- i.d.R. zu einer Eingabe ***w*** viele mögliche Berechnungen, oft unendlich lang
- $\boldsymbol{w} \in \boldsymbol{L}(\boldsymbol{M}) \Longleftrightarrow$ es gibt (mindestens) eine (endliche) akzeptierende Berechnung auf ***w***
- ***NTIME***(***f***): *jede* Berechnung hält nach maximal $\boldsymbol{f}(|\boldsymbol{w}|)$ Schritten
  Keine unendlichen Berechnungen erlaubt, ***M*** hält *stets*!
- Halt in Endzustand: Eingabewort ***w*** akzeptiert.
- Halt (Steckenbleiben) in Nicht-Endzustand: ***w*** nicht akzeptiert.

Nach Definition der Turingmaschinen sofort $\boldsymbol{P} \subseteq \boldsymbol{NP}$, offen jedoch:

$\boldsymbol{P}$-$\boldsymbol{NP}$-Problem

$$\boldsymbol{P} \stackrel{?}{=} \boldsymbol{NP}$$

- berühmtestes Problem der theoretischen Informatik.
- Lösung wertvoll, Preisgeld 1 Million US-$
  vgl. `http://www.claymath.org/millennium`
- Grund: viele *praktisch relevante Probleme* liegen in $\boldsymbol{NP}$,
  für die keine brauchbaren Algorithmen bekannt sind
  (d.h. unbekannt ist, ob sie in $\boldsymbol{P}$ liegen)

Spezielle große Problemklasse: $\boldsymbol{NP}$-vollständige Probleme

Liegt auch nur ein $\boldsymbol{NP}$-vollständiges Problem auch in $\boldsymbol{P}$, so ist $\boldsymbol{P} = \boldsymbol{NP}$
Liegt auch nur ein $\boldsymbol{NP}$-vollständiges Problem *nicht* in $\boldsymbol{P}$, so ist $\boldsymbol{P} \neq \boldsymbol{NP}$

Arbeitshypothese (z.B. für Kryptographie): Eher $\boldsymbol{P} \neq \boldsymbol{NP}$ als $\boldsymbol{P} = \boldsymbol{NP}$...

Satz 12.5

Die Klasse $\boldsymbol{NP}$ *ist enthalten in* $\bigcup_{\boldsymbol{p}Polynom} \boldsymbol{TIME}(\boldsymbol{2^{p(n)}})$.

Beweisskizze: Betrachte $\boldsymbol{L} \in \boldsymbol{NP}$

- $\boldsymbol{M}$ sei nichtdeterministische Turingmaschine mit $\boldsymbol{L}(\boldsymbol{M}) = \boldsymbol{L}$
- $\boldsymbol{M}$ arbeite auf Eingabe $\boldsymbol{w}$ in Zeit $\leq \boldsymbol{q}(|\boldsymbol{w}|)$ für Polynom $\boldsymbol{q}$
- $\boldsymbol{c}$ sei Maximalzahl möglicher Nachfolgekonfigurationen einer Konfiguration.

Betrachte Berechungsbaum $\boldsymbol{B_w}$ aller Berechnungen von $\boldsymbol{M}$ auf $\boldsymbol{w}$

- Knoten in $\boldsymbol{B_w}$ sind markiert mit Konfigurationen von $\boldsymbol{M}$
- Kante $(\boldsymbol{K}, \boldsymbol{K'})$ in $\boldsymbol{B_w}$ entspricht Übergang $\boldsymbol{K} \vdash \boldsymbol{K'}$ bei $\boldsymbol{M}$
- jeder Knoten in $\boldsymbol{B_w}$ hat Grad $\leq \boldsymbol{c}$
- $\boldsymbol{B_w}$ hat Tiefe $\leq \boldsymbol{q}(|\boldsymbol{w}|)$, also hat $\boldsymbol{B_w}$ höchstens $\boldsymbol{c}^{\boldsymbol{q}(|\boldsymbol{w}|)}$ Knoten

Deterministischer Algorithmus für L:

- Durchsuche B_w, ob akzeptierende Konfiguration enthalten
- Wenn ja, akzeptiere w.
- Wenn nein, verwerfe w.

Da $\leq c^{q(|w|)}$ Knoten: Zeitaufwand $2^{p(|w|)}$ für Polynom p

Damit

$$L \in TIME(2^{p(n)})$$

Eine Sprache L (aus Worten!) heißt LOOP-berechenbar,
wenn die charakteristische Funktion der folgenden (Zahlen-!)Menge
LOOP-berechenbar ist:

$$\nu^{-1}(L) = \{n \in \mathbb{N} \mid \nu(n) \in L\}$$

Damit:

Satz 12.6

Ist $f : \mathbb{N} \to \mathbb{N}$ eine LOOP-berechenbare Funktion,
so ist jede Sprache L aus $TIME(f)$ LOOP-berechenbar.

Beweisskizze: Sei L aus $TIME(f)$

- Betrachte Turingmaschine M mit $L = L(M)$
- M arbeite in Zeit $f(|w|)$ bei Eingaben w

Konstruiere aus M neue Turingmaschine M' mit:

- M' berechnet aus einer Eingabe $n \in \mathbb{N}$ (genauer: aus $u \in \{0,1\}^*$ mit $n = bin(u)$) zunächst w mit $w = \nu(n) = \nu(bin(u))$
- Danach wendet M' die Maschine M auf w an.
- Akzeptiert M, so gibt M' eine **1** aus, sonst eine **0**

Also:

- M' berechnet die charakteristische Funktion von $\nu^{-1}(L)$
- Achtung: Rechenzeit von M' wird gemessen in $|u|$, d.h. in $\log_2 n$.
- M' arbeitet in Zeit

$$f'(|u|) := f(|w|) + (\text{ Zeit zur Umwandlung } n \mapsto w)$$

- Da f LOOP-berechenbar ist, gibt es LOOP-berechenbares g mit $f'(|u|) \leq g(n)$

Aus Satz 3.5:

- Zu M' gibt es ein WHILE-Programm WP, das M' simuliert
- WP hat dabei nur eine WHILE-Schleife
  (mit Test auf den 'simulierten' Zustand x_z von M')
- Ein Schleifendurchlauf bei WP entspricht einem Schritt von M'

Ersetze nun die WHILE-Schleife in WP

$$\texttt{WHILE } x_z \neq 0 \texttt{ DO } \ldots$$

durch

$$y := g(n);\ \texttt{LOOP } y \texttt{ DO IF } x_z \neq 0 \texttt{ THEN } \ldots \texttt{ END}$$

mit Resultat WP'

WP' berechnet ebenfalls $ch_{\nu^{-1}(L)}$

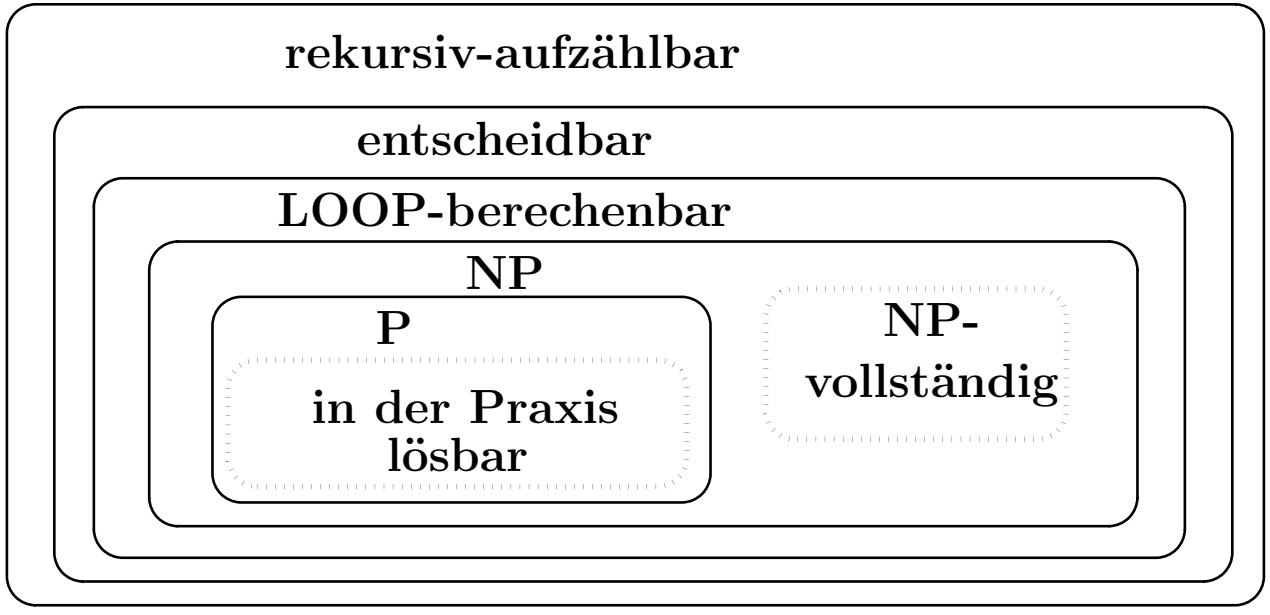
g ist LOOP-berechenbar, also ist auch WP' LOOP-berechenbar!

Beispiele:

- alle Polynome p sind LOOP-berechenbar
- alle Funktionen der Form $2^{p(n)}$ (mit Polynom p) sind LOOP-berechenbar.

Damit sind alle Mengen in P oder NP auch LOOP-berechenbar!

Aktueller Stand der Kenntnisse und Vermutungen:



Definition 13.1

Seien A *und* B *Sprachen über einem Alphabet* E*.*
Dann heißt A *auf* B *polynomial reduzierbar, wenn es eine in Polynomzeit berechenbare Funktion* $f : E^* \to E^*$ *gibt, so dass für alle* $w \in E^*$ *gilt:*

$$w \in A \iff f(w) \in B$$

Schreibweise: $A \leq_p B$

- $f \in FP$, also f total
- Falls A, B über verschiedenen Alphabeten E_A, E_B:
  Wähle z.B: $E := E_A \cup E_B$

Lemma 13.2

- *Falls* $A \leq_p B$ *und* $B \in P$, *so ist auch* $A \in P$.
- *Falls* $A \leq_p B$ *und* $B \in NP$, *so ist auch* $A \in NP$.

Beweis:
Sei $A \leq_p B \in P$ mit Reduktionsfunktion f:

- M_B sei Turingmaschine M_B mit $L(M_B) = B$
- M_B arbeite in Zeit q für Polynom q
- M_f berechne f
- M_f arbeite in Zeit p für Polynom p

Betrachte folgende Maschine M:

- Aus Eingabe w berechnet M zuerst $v := f(w)$
  durch Simulation von M_f
- M testet, ob $v \in B$ durch Simulation von M_B
- M akzeptiert w genau dann, wenn v dabei von M_B akzeptiert wird

Damit sofort $L(M) = A$

Zeitbedarf von M:

$$time_M(w) \leq p(|w|) + q(|f(w)|)$$

O.B.d.A.: q hat keine negativen Koeffizienten, damit q monoton

Mit $|f(w)| \leq p(|w|) + |w|$ also
(dabei $\ldots + |w|$ nur, weil w evtl. nicht ganz gelesen wird!)

$$time_M(w) \leq p(|w|) + q(|f(w)|) \leq p(|w|) + q(p(|w|) + |w|)$$

d.h. polynomial in $|w|$

Fall $A \leq_p B \in NP$: analog

Lemma 13.3

Die Relation $\leq_p$ ist transitiv, d.h. aus $A \leq_p B \leq_p C$ folgt $A \leq_p C$.

Beweis: Komposition der Reduktionen, Zeitschranke analog zu 13.2

Anmerkung: $\leq_p$ ist reflexiv,
aber weder symmetrisch (d.h. $A \leq_p B \Rightarrow B \leq_p A$ gilt nicht)
noch antisymmetrisch (d.h. $(A \leq_p B \vee B \leq_p A)$ gilt nicht)

Definition 13.4

- *Eine Sprache **L** heißt **NP**-hart,*
  *wenn alle Sprachen aus **NP** auf sie polynomial reduzierbar sind.*
- *Eine Sprache **L** heißt **NP**-vollständig,*
  *wenn sie aus **NP** ist und **NP**-hart ist.*

Damit: ***NP***-vollständige Probleme sind schwierigste Probleme in ***NP***!

Satz 13.5

***L** sei **NP**-vollständig. Dann folgt:*

$$\boldsymbol{L} \in \boldsymbol{P} \iff \boldsymbol{P} = \boldsymbol{NP}$$

Beweis von '$\Leftarrow$': trivial

Beweis von '$\Rightarrow$': Sei $\boldsymbol{L} \in \boldsymbol{P}$ und ***NP***-vollständig.

Betrachte beliebiges $\boldsymbol{L'} \in \boldsymbol{NP}$, damit $\boldsymbol{L'} \leq_{\boldsymbol{p}} \boldsymbol{L}$.

Mit Lemma 13.2 also $\boldsymbol{L'} \in \boldsymbol{P}$

Damit gilt $\boldsymbol{NP} \subseteq \boldsymbol{P}$; Inklusion $\boldsymbol{P} \subseteq \boldsymbol{NP}$ gilt nach Definition

(kontextfreie) Sprache $\mathcal{F}$ der aussagenlogischen Formeln:

- Jede Variable x_i ist eine Formel (mit $i \in \mathbb{N}$),
  setze $V := \{x_i \mid i \in \mathbb{N}\}$
- Sind F, G Formeln, so auch $\neg F$, $(F \wedge G)$ und $(F \vee G)$

Formeln können ausgewertet werden, wenn den Variablen Werte zugewiesen werden:

- Eine Belegung ϕ ist eine Abbildung $\phi : V \to \{0, 1\}$
- ϕ kann auf Formeln erweitert werden durch

$$\begin{aligned} \phi(\neg F) &= 1 - \phi(F) \\ \phi(\,(F \wedge G)\,) &= \min\{\phi(F), \phi(G)\} \\ \phi(\,(F \vee G)\,) &= \max\{\phi(F), \phi(G)\} \end{aligned}$$

Kodiere Variablen x_i binär durch $x\ bin(i)$, dann gilt:

- Für jede aussagenlogische Formel F ist $code(F)$ Wort über

$$E = \{0, 1, x, (,), \neg, \wedge, \vee\}$$

- Hat F Länge m und n Variablen, dann gilt $|code(F)| \leq m \log n$.

Weitere Schreibweisen:

- aussagekräftigere Namen für Variablen sind erlaubt, etwa

$$F = \neg(\text{wichtige Variable} \vee (\text{andere Variable} \wedge \text{wichtige Variable}))$$

  mit $code(F) = \neg(x1 \vee (x10 \wedge x1))$
- Klammern dürfen fehlen (wo keine Mißverständnisse entstehen)
- Abkürzungen sind erlaubt, etwa $\bigvee_{1 \leq k \leq 4} x_k$ für $x_1 \vee x_2 \vee x_3 \vee x_4$ bzw. $x \rightarrow y$ als Umschreibung für $\neg x \vee y$
- aber stets: $code(F)$ als Wort über $E = \{0, 1, x, (,), \neg, \wedge, \vee\}$

Lemma 13.6

Für jedes m *gibt es eine Formel* G *der Länge* $O(m^2)$
mit den Variablen $x_1, \ldots, x_m$,
so dass G *genau dann den Wahrheitswert* 1 *erhält,*
wenn genau eine der Variablen mit 1 *belegt wird.*

Beweis: Man wähle

$$G(x_1, \ldots, x_m) = (x_1 \vee \ldots \vee x_m) \wedge \Big(\bigwedge_{1 \le j \le m-1} \; \bigwedge_{j+1 \le l \le m} \neg(x_j \wedge x_l) \Big)$$

Definition 13.7 (Erfüllbarkeitsproblem der Aussagenlogik)

SAT $:= \{$***code***$(\boldsymbol{F})$ *aus* $\boldsymbol{E}^*$ $\mid$ $\boldsymbol{F}$ *erfüllbare Formel der Aussagenlogik*$\}$

übliche, weniger formale Formulierung:

- gegeben: eine aussagenlogische Formel $\boldsymbol{F}$
- gefragt: Ist $\boldsymbol{F}$ erfüllbar,
  d.h. gibt es eine Belegung der Variablen in $\boldsymbol{F}$ mit **0** und **1**,
  so dass die Formel $\boldsymbol{F}$ den Wert **1** erhält?

Satz 13.8 (Cook)

Das Erfüllbarkeitsproblem ***SAT*** *der Aussagenlogik ist* ***NP****-vollständig.*

Beweisskizze zu $\boldsymbol{SAT} \in \boldsymbol{NP}$:

Eine aussagenlogische Formel $\boldsymbol{F}$ sei gegeben durch $\boldsymbol{code}(\boldsymbol{F})$

Betrachte nichtdeterministische Turingmaschine $\boldsymbol{M}$ wie folgt:

- Phase 1: $\boldsymbol{M}$ rät einen Wert **0** oder **1** für jede Variable $\boldsymbol{x_i}$ von $\boldsymbol{F}$.

  Aufwand: polynomial (d.h. $\leq \boldsymbol{p}(|\boldsymbol{code}(\boldsymbol{F})|)$ für ein Polynom $\boldsymbol{p}$)
- Phase 2: $\boldsymbol{M}$ bestimmt Wert $\boldsymbol{z}$ von $\boldsymbol{F}$ bei diesen Werten für die $\boldsymbol{x_i}$
  Falls $\boldsymbol{z} = \mathbf{1}$, akzeptiert $\boldsymbol{M}$ die Eingabe $\boldsymbol{code}(\boldsymbol{F})$
  Falls $\boldsymbol{z} = \mathbf{0}$, verwirft $\boldsymbol{M}$ die Eingabe $\boldsymbol{code}(\boldsymbol{F})$

  Aufwand: wieder polynomial in $|\boldsymbol{code}(\boldsymbol{F})|$

Offensichtlich: $\boldsymbol{M}$ arbeitet stets in Polynomzeit und

$$\boldsymbol{code}(\boldsymbol{F}) \in \boldsymbol{SAT} \iff \boldsymbol{M} \text{ kann akzeptieren}$$

Übliche Nachweismethode für $\boldsymbol{L} \in \boldsymbol{NP}$: *guess and check*

— rate nichtdeterministisch Lösungversuch (Phase 1, guess)

— überprüfe, ob Lösungsversuch korrekt (Phase 2, check)

Beweis, dass ***SAT NP***-hart ist:

Sei dazu $L \in NP$ beliebig.

- M sei eine nichtdeterministische Turingmaschine mit $L = L(M)$ mit Arbeitsalphabet $A = \{a_1, ..., a_l\}$, Zustandsmenge $S = \{s_0, ..., s_k\}$, Endzustandsmenge S_f
- M arbeite in Polynomzeit
- O.B.d.A.: M ist Einbandmaschine
- Polynom p sei obere Schranke für den Zeitverbrauch von M
- Nutze u.a., dass M im Endzustand nicht mehr weiterrechnet

Zeige: $L \leq_p SAT$, d.h.
zu jeder Eingabe $y = y_1, ..., y_n$ (für L gedacht, und mit $|y| =: n$) konstruiere (in Polynomzeit!) Boolesche Formel F_y mit

$$F_y \text{ erfüllbar} \Leftrightarrow M \text{ kann } y \text{ akzeptieren} \Leftrightarrow y \in L$$

Also: Abbildung $y \mapsto F_y$ ist die gesuchte Reduktionsfunktion!

Zunächst Auflistung der Variablen von F_y:

Variable	Indizes	Bedeutung
$zust_{t,s}$	$t = 0, ..., p(n)$ $s \in S$	$zust_{t,s} = 1 \Leftrightarrow$ nach t Schritten befindet sich M im Zustand s
$pos_{t,i}$	$t = 0, ..., p(n)$ $i = -p(n), ..., p(n)$	$pos_{t,i} = 1 \Leftrightarrow$ Schreib-/Lesekopf von M befindet sich nach t Schritten auf Position i
$band_{t,i,a}$	$t = 0, ..., p(n)$ $i = -p(n), ..., p(n)$ $a \in A$	$band_{t,i,a} = 1 \Leftrightarrow$ nach t Schritten befindet sich auf Position i das Zeichen a

M rechnet nur $\leq p(n)$ Schritte
$\Rightarrow$ M bewegt den Kopf nur um $\leq p(n)$ Positionen

Also: i mit $-p(n) \leq i \leq p(n)$ reicht für alle vorkommenden Positionen des Lese-Schreibkopfes!

Struktur von F_y: Fünf Teilformeln, durch $\wedge$ verbunden:

(R: Konsistenzbedingung, B_a Anfangsbedingung,
U_1, U_2 Übergangsbedingungen, B_e Endbedingung)

$$F_y = R \wedge B_a \wedge U_1 \wedge U_2 \wedge B_e$$

In R wird ausgedrückt (mit Formel G aus Lemma 13.6), dass:

- Zu jedem Zeitpunkt t gilt $zust_{t,s} = 1$ für genau ein s.
- Zu jedem Zeitpunkt t gibt es genau eine Bandposition i, über der der Kopf steht: $pos_{t,i} = 1$.
- Zu jedem Zeitpunkt t und jeder Position i gibt es genau ein a mit $band_{t,i,a} = 1$.

Also:

$$\begin{aligned} R := \bigwedge_t \quad [\ & G(zust_{t,s_0}, \ldots, zust_{t,s_k}) \\ & \wedge \quad G(pos_{t,-p(n)}, \ldots, pos_{t,p(n)}) \\ & \wedge \quad \bigwedge_i G(band_{t,i,a_1}, \ldots, band_{t,i,a_l}) \ ] \end{aligned}$$

B_a beschreibt den Status der Variablen zum Startzeitpunkt $t = 0$:

$$B_a = zust_{0,s_0} \wedge pos_{0,1} \wedge \bigwedge_{1 \le j \le n} band_{0,j,y_j} \wedge \bigwedge_{-p(n) \le j \le 0} band_{0,j,\square} \wedge \bigwedge_{n+1 \le j \le p(n)} band_{0,j,\square}$$

B_e prüft nach, ob im Zeitpunkt $p(n)$ ein Endzustand erreicht wird:

$$B_e = \bigvee_{s \text{ Endzustand}} zust_{p(n),s}$$

U_2 betrifft die Felder des Bandes, über denen der Kopf nicht steht (hier gibt es keine Änderungen!)

$$U_2 = \bigwedge_{t,i,a} ((\neg pos_{t,i} \wedge band_{t,i,a}) \rightarrow band_{t+1,i,a})$$

Zu $\boldsymbol{U_1}$:

- betrifft die Bandpositionen, an denen sich der Kopf befindet
- beschreibt den (nichtdeterministischen) Übergang vom Zeitpunkt $\boldsymbol{t}$ zum Zeitpunkt $\boldsymbol{t}+\mathbf{1}$.
- Dabei steht $\boldsymbol{b}$ für -1 (L), 0 (N) oder 1 (R)

$$\boldsymbol{U_1} = \bigwedge_{\boldsymbol{t},\boldsymbol{s},\boldsymbol{i},\boldsymbol{a}} [(\boldsymbol{zust_{t,s}} \wedge \boldsymbol{pos_{t,i}} \wedge \boldsymbol{band_{t,i,a}}) \rightarrow \bigvee_{\boldsymbol{s'},\boldsymbol{a'},\boldsymbol{b} \text{ mit } (\boldsymbol{s'},\boldsymbol{a'},\boldsymbol{b}) \in \delta(\boldsymbol{s},\boldsymbol{a})} (\boldsymbol{zust_{t+1,s'}} \wedge \boldsymbol{pos_{t+1,i+b}} \wedge \boldsymbol{band_{t+1,i,a'}})]$$

Ist dabei $\delta(\boldsymbol{s}, \boldsymbol{a})$ leer:

- d.h. für Endzustände $\boldsymbol{s} \in \boldsymbol{S_f}$ oder wenn $\boldsymbol{M}$ in $\boldsymbol{s}$ steckenbleibt
- dann setze $\delta(\boldsymbol{s}, \boldsymbol{a}) := (\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{N})$
- d.h. Konfiguration bleibt dann bei $\boldsymbol{t} + \mathbf{1}$ exakt wie bei $\boldsymbol{t}$
- damit Akzeptanz/Nichtakzeptanz am Zeitpunkt $\boldsymbol{t} = \boldsymbol{p}(\boldsymbol{n})$ ablesbar! (vgl. $\boldsymbol{B_e}$)

Zeige $y \in L \Leftrightarrow F_y$ ist erfüllbar:

Beweis von '$\Rightarrow$': Sei $y \in L$

- Also exisitiert Rechnung von M auf y mit Länge $\leq p(n)$, die zu Endzustand führt.
- Definiere Belegung der Variablen in F_y nach ihrer Bedeutung
- Damit: alle Teilformeln erhalten Wert $\mathbf{1}$, d.h. F_y ist erfüllbar

Beweis von '$\Leftarrow$': Sei F_y durch Belegung erfüllt.

- alle Teilformeln erhalten Wert $\mathbf{1}$
- R erfüllt: Variablenwerte von $zust_{t,s}$, $pos_{t,i}$ und $band_{t,i,a}$ definieren Konfiguration K_t von M für jedes $0 \leq t \leq p(n)$
- B_a erfüllt: K_0 entspricht Startkonfiguration $I_M(y)$ (mit Zusatz-Leerzeichen)
- U_1, U_2 erfüllt: $K_t \vdash K_{t+1}$ oder Berechnung endet mit $K_t = K_{t+1}$
- B_e erfüllt: Berechnung endet in Endzustand, d.h. $y \in L(M) = L$

Komplexität der Bestimmung von F_y:

Insgesamt gibt es $O(p(n)^2)$ Variablen, Länge der Teilformeln:

- R hat Länge $O(p(n)^3)$ ($p(n)$-fach Formel G mit $O(p(n))$ Variablen)
- U_1, U_2 haben Länge $O(p(n)^2)$
- B_a hat Länge $O(p(n))$
- B_e hat konstante Länge $O(1)$

Also: Länge von $code(F_y)$ ist in $O(p(n)^3 \log p(n))$

Aufwand zur Erstellung von F_y aus y:

- Linear in $|code(F_y)|$
- d.h. polynomial in $n = |y|$

Nachweis der $\boldsymbol{NP}$-Vollständigkeit eines Problems $\boldsymbol{L}$ i.d.R. durch:

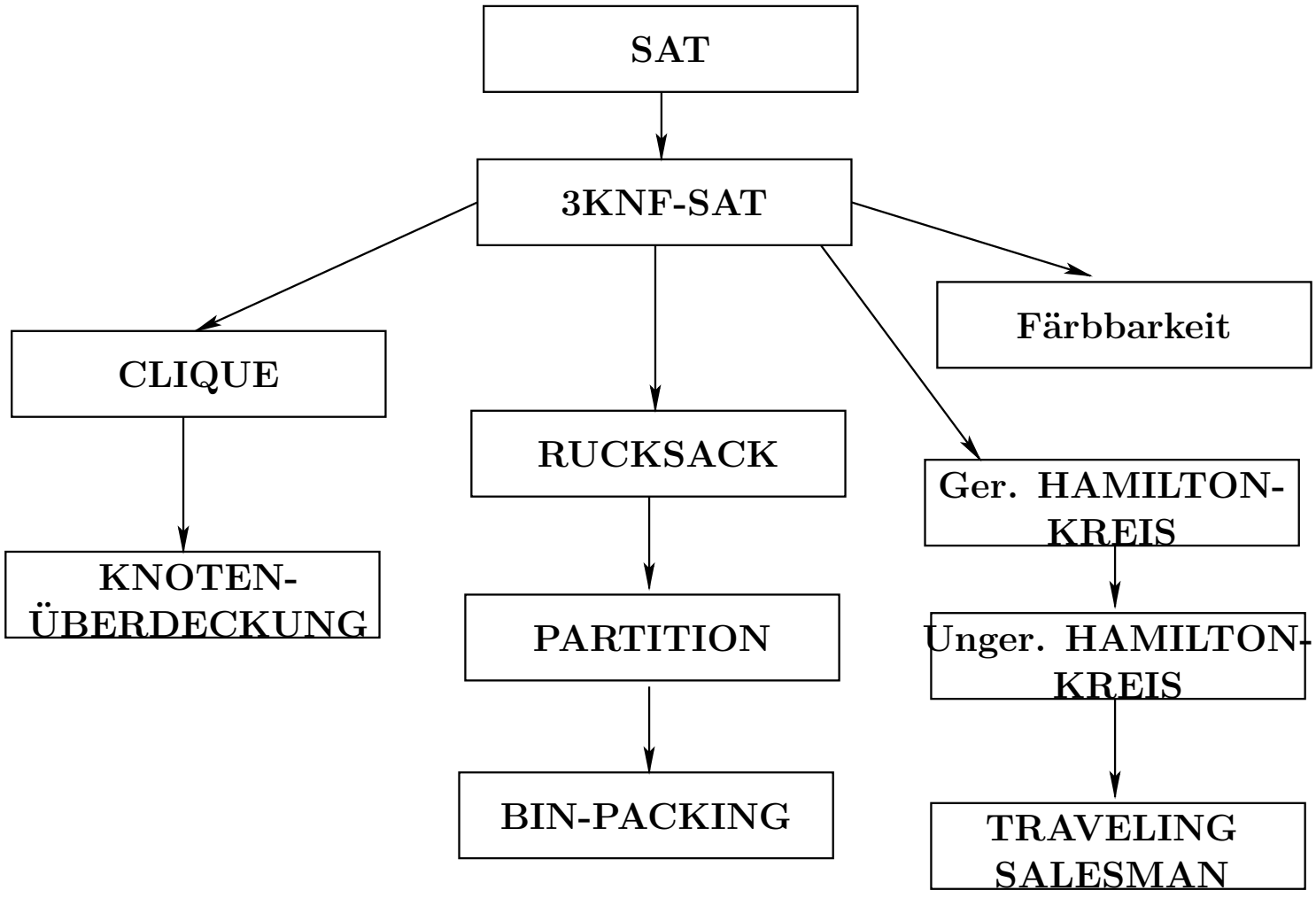
1. Zeige $\boldsymbol{L} \in \boldsymbol{NP}$ durch Konstruktion einer nichtdeterministischen Turingmaschine, die $\boldsymbol{L}$ erkennt und in Polynomzeit arbeitet.

   Meist: 'Guess and Check'-Methode
   - in nichtdeterministischer Phase 1: Rate Lösungsvorschlag
   - in deterministischen Phase 2: Überprüfe Lösungsvorschlag

2. Zeige, dass $\boldsymbol{L}$ $\boldsymbol{NP}$-hart ist
   durch Beweis von $\boldsymbol{L'} \leq_{\boldsymbol{p}} \boldsymbol{L}$ für bekanntes $\boldsymbol{NP}$-hartes Problem $\boldsymbol{L'}$

   Für beliebiges $\boldsymbol{L''} \in \boldsymbol{NP}$ gilt dann $\boldsymbol{L''} \leq_{\boldsymbol{p}} \boldsymbol{L'} \leq_{\boldsymbol{p}} \boldsymbol{L}$
   und damit $\boldsymbol{L''} \leq_{\boldsymbol{p}} \boldsymbol{L}$.

Beispiele ***NP***-vollständiger Probleme mit möglichen Reduktionen:



Satz 14.1 (NP-vollständige Probleme)

***NP**-vollständig sind:*

3KNF-Sat

- *gegeben: Eine Boolesche Formel **F** in konjunktiver Normalform (Klauselform) mit höchstens **3** Literalen pro Klausel.*
- *gefragt: Ist **F** erfüllbar?*

CLIQUE

- *gegeben: Ein ungerichteter Graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ und eine Zahl $\boldsymbol{k} \in \mathbb{N}$*
- *gefragt: Besitzt **G** eine 'Clique' der Größe mindestens **k**?*

*(Eine 'Clique' der Größe **k** ist eine Menge $\boldsymbol{V'} \subseteq \boldsymbol{V}$ mit $|\boldsymbol{V'}| \geq \boldsymbol{k}$ und $\{\boldsymbol{u}, \boldsymbol{v}\} \in \boldsymbol{E}$ für alle $\boldsymbol{u}, \boldsymbol{v} \in \boldsymbol{V'}$ mit $\boldsymbol{u} \neq \boldsymbol{v}$.)*

Fortsetzung von 14.1:

KNOTENÜBERDECKUNG

- *gegeben:* Ein ungerichteter Graph $G = (V, E)$ und eine Zahl $k \in \mathbb{N}$
- *gefragt:* Besitzt G eine 'überdeckende Knotenmenge' der Größe höchstens k?

(Eine 'Knotenüberdeckung' der Größe k ist eine Menge $V' \subseteq V$ mit $|V'| \leq k$, so dass für alle $\{u, v\} \in E$ gilt: $u \in V'$ oder $v \in V'$.)

RUCKSACK (oder **SUBSET SUM**)

- *gegeben:* Natürliche Zahlen $a_1, a_2, \ldots, a_k, b \in \mathbb{N}$
- *gefragt:* Gibt es eine Teilmenge $J \subseteq \{1, 2, \ldots, k\}$ mit

$$\sum_{i \in J} a_i = b$$

Fortsetzung von 14.1:

PARTITION

- *gegeben:* Natürliche Zahlen $a_1, a_2, \ldots, a_k \in \mathbb{N}$
- *gefragt:* Gibt es eine Teilmenge $J \subseteq \{1, 2, \ldots, k\}$ mit

$$\sum_{i \in J} a_i = \sum_{i \notin J} a_i$$

BIN PACKING

- *gegeben:* Eine 'Behältergröße' $b \in \mathbb{N}$, die Anzahl $k \in \mathbb{N}$ der Behälter, Objekte $a_1, a_2, \ldots, a_n \in \mathbb{N}$.
- *gefragt:* Können die Objekte so auf die k Behälter verteilt werden, dass kein Behälter überläuft?

(Gibt es eine Abbildung $f : \{1, \ldots, n\} \to \{1, \ldots, k\}$, so dass für alle $j \in \{1, \ldots, k\}$ gilt $\sum_{f(i)=j} a_i \leq b$?)

Fortsetzung von 14.1:

GERICHTETER HAMILTON-KREIS

- *gegeben:* Ein gerichteter Graph $G = (V, E)$.
- *gefragt:* Besitzt G einen Hamilton-Kreis?

(D.h. gibt es Permutation π der Knotenindizes $(v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(n)})$, so dass stets $(v_{\pi(i)}, v_{\pi(i+1)}) \in E$ und $(v_{\pi(n)}, v_{\pi(1)}) \in E$)

UNGERICHTETER HAMILTON-KREIS

- *gegeben:* Ein ungerichteter Graph $G = (V, E)$.
- *gefragt:* Besitzt G einen Hamilton-Kreis?

(D.h. gibt es Permutation π der Knotenindizes $(v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(n)})$, so dass stets $\{v_{\pi(i)}, v_{\pi(i+1)}\} \in E$ und $\{v_{\pi(n)}, v_{\pi(1)}\} \in E$)

Fortsetzung von 14.1:

TRAVELING SALESMAN

- *gegeben:* Eine $n \times n$-Matrix $(M_{i,j})$ von 'Entfernungen' zwischen n 'Städten' und eine Zahl k.
- *gefragt:* Gibt es Permutation π (eine 'Rundreise'), so dass

$$\sum_{i=1}^{n-1} M_{v_{\pi(i)}, v_{\pi(i+1)}} + M_{v_{\pi(n)}, v_{\pi(1)}} \leq k$$

FÄRBBARKEIT

- *gegeben:* Ein ungerichteter Graph $G = (V, E)$ und eine Zahl $k \in \mathbb{N}$
- *gefragt:* Gibt es eine Färbung der Knoten in V mit k verschiedenen Farben, so dass keine zwei benachbarten Knoten in G dieselbe Farbe haben.

Mit Guess and Check: Alle Probleme liegen in ***NP***

Jetzt: Teile der Reduktionen...

Satz 14.2

***3KNF-SAT** ist **NP**-vollständig.*

- *Literal*: Variable oder negierte Variable
- *Klausel*: Disjunktion von Literalen
- *konjunktive Normalform* (**KNF**): Konjunktion von Klauseln
- **3KNF**: Konjunktion von Klauseln mit je höchstens drei Literalen

Beweis durch Nachweis von $\mathbf{SAT} \leq_p \mathbf{3KNF\text{-}SAT}$

D.h.: Gegeben Formel F, konstruiere 3KNF F' mit

$$F \text{ erfüllbar} \Leftrightarrow F' \text{ erfüllbar}$$

Achtung:

- F' hat mehr Variablen als F
- Daher nicht $F \Leftrightarrow F'$
- Verlangt ist nur Erfüllbarkeitsäquivalenz.

$\boldsymbol{F}$ wird in mehreren Schritten umgeformt:

1. Schritt: Wende die de Morgan-Regeln an und bringen alle Negationszeichen zu den Literalen:

$$\neg(\boldsymbol{F} \wedge \boldsymbol{G}) \quad \text{ersetzt durch} \quad (\neg\boldsymbol{F} \vee \neg\boldsymbol{G})$$
$$\neg(\boldsymbol{F} \vee \boldsymbol{G}) \quad \text{ersetzt durch} \quad (\neg\boldsymbol{F} \wedge \neg\boldsymbol{G})$$

2. Schritt: Ersetze iterativ Teilformeln $(\boldsymbol{x} \circ \boldsymbol{x'})$ mit *Literalen* $\boldsymbol{x}$, $\boldsymbol{x'}$ durch neue Variablen $\boldsymbol{y}$ und ergänze jeweils durch Formel $[\boldsymbol{y} \Leftrightarrow (\boldsymbol{x} \circ \boldsymbol{x'})]$ (Tiefe der Formeln wird reduziert!),z.B.

$$(\boldsymbol{F} \circ (\boldsymbol{x} \vee \boldsymbol{x'}) \circ \boldsymbol{G}) \quad \text{durch} \quad (\boldsymbol{F} \circ \boldsymbol{y} \circ \boldsymbol{G}) \wedge [\boldsymbol{y} \Leftrightarrow (\boldsymbol{x} \vee \boldsymbol{x'})]$$
$$(\boldsymbol{F} \circ (\boldsymbol{x} \wedge \boldsymbol{x'}) \circ \boldsymbol{G}) \quad \text{durch} \quad (\boldsymbol{F} \circ \boldsymbol{y} \circ \boldsymbol{G}) \wedge [\boldsymbol{y} \Leftrightarrow (\boldsymbol{x} \wedge \boldsymbol{x'})]$$

3. Schritt: Ersetze alle Teilformeln $\boldsymbol{y} \Leftrightarrow (\boldsymbol{x} \circ \boldsymbol{x'})$ durch Klauseln:

$$\boldsymbol{y} \Leftrightarrow (\boldsymbol{x} \vee \boldsymbol{x'}) \quad \text{durch} \quad (\boldsymbol{y} \vee \neg\boldsymbol{x}) \wedge (\boldsymbol{y} \vee \neg\boldsymbol{x'}) \wedge (\neg\boldsymbol{y} \vee \boldsymbol{x} \vee \boldsymbol{x'})$$
$$\boldsymbol{y} \Leftrightarrow (\boldsymbol{x} \wedge \boldsymbol{x'}) \quad \text{durch} \quad (\neg\boldsymbol{y} \vee \boldsymbol{x}) \wedge (\neg\boldsymbol{y} \vee \boldsymbol{x'}) \wedge (\boldsymbol{y} \vee \neg\boldsymbol{x} \vee \neg\boldsymbol{x'})$$

Beispiel:

$$
\begin{aligned}
F = F_0 &= \neg(\neg(x_1 \vee \neg x_3) \vee x_2) \\
\text{Schritt 1: } F_1 &= ((x_1 \vee \neg x_3) \wedge \neg x_2 \\
\text{Schritt 2: } F_2 &= (y_1 \wedge \neg x_2) \\
&\quad \wedge (y_1 \Leftrightarrow (x_1 \vee \neg x_3)) \\
F_3 &= y_2 \\
&\quad \wedge [y_2 \Leftrightarrow (y_1 \wedge \neg x_2)] \\
&\quad \wedge [y_1 \Leftrightarrow (x_1 \vee \neg x_3)] \\
\text{Schritt 3: } F_4 &= y_2 \\
&\quad \wedge (\neg y_2 \vee y_1) \wedge (\neg y_2 \vee \neg x_2) \wedge (y_2 \vee \neg y_1 \vee x_2) \\
&\quad \wedge (y_1 \vee \neg x_1) \wedge (y_1 \vee x_3) \wedge (\neg y_1 \vee x_1 \vee \neg x_3)
\end{aligned}
$$

Stets: F_i erfüllbar $\Leftrightarrow$ F_{i+1} erfüllbar, damit: F erfüllbar $\Leftrightarrow$ F_4 erfüllbar

Aufwand der Umformungen: Polynomial in der Länge von F

Anmerkung: Das entsprechende Problem **2KNF-SAT** mit je maximal zwei Literalen liegt bereits in P...

Satz 14.3

*Das **CLIQUE**-Problem ist **NP**-vollständig.*

Beweis: Zeige **3KNF-SAT** $\leq_p$ **CLIQUE**

Sei F eine Formel in **3KNF**.

O.B.d.A.: Jede Klausel hat exakt 3 Literale, ansonsten:
Ersetze (z) durch $(z \vee z \vee z)$ und $(y \vee z)$ durch $(y \vee z \vee z)$

Also o.B.d.A.:

$$F = (z_{11} \vee z_{12} \vee z_{13}) \wedge \ldots \wedge (z_{m1} \vee z_{m2} \vee z_{m3})$$

mit z_{jk} aus $\{x_1, \ldots, x_n\} \cup \{\neg x_1, \ldots, \neg x_n\}$.

Betrachte folgenden Graphen $G(V, E)$ zur Formel F:

$$\begin{aligned} V &= \{(1,1),(1,2),(1,3),\ldots,(m,1),(m,2),(m,3)\} \\ E &= \{\, \{(i,p),(j,q)\} \mid i \neq j \wedge z_{ip} \neq \neg z_{jq}\} \end{aligned}$$

Anmerkung: Die Vervielfachung der Literale ist nur zur leichteren Formulierung notwendig...)

Als Beispiele Graphen zu:

- $$(\boldsymbol{x} \vee \boldsymbol{y}) \wedge (\neg \boldsymbol{x} \vee \boldsymbol{y}) \wedge (\boldsymbol{x} \vee \neg \boldsymbol{y})$$
- $$\boldsymbol{x} \wedge \boldsymbol{y} \wedge (\neg \boldsymbol{x} \vee \neg \boldsymbol{y})$$

Außerdem ordnen wir der Formel F die Zahl $k := m$ zu.

Dann ist F durch eine Belegung erfüllbar

- genau dann, wenn es in jeder Klausel ein Literal gibt, das den Wert **1** erhält, z.B. $z_{1p_1}, \ldots, z_{mp_m}$,
- genau dann, wenn es Literale $z_{1p_1}, \ldots, z_{mp_m}$ gibt mit $z_{ip_i} \neq \neg z_{jp_j}$ für $i \neq j$
- genau dann, wenn es Knoten $(\mathbf{1}, p_1), (\mathbf{2}, p_2), \ldots, (m, p_m)$ in G gibt, die paarweise verbunden sind,
- genau dann, wenn es in G eine Clique der Größe $k = m$ gibt.

Satz 14.4

KNOTENÜBERDECKUNG *ist* ***NP****-vollständig.*

Beweis: Zeige **CLIQUE** $\leq_p$ **KNOTENÜBERDECKUNG**

Verwende Abbildung

$$(G, k) \mapsto (G', k')$$

mit

$$G' = (V, V^2 \setminus E), \qquad k' := |V| - k$$

bei $G = (V, E)$

Satz 14.5

***RUCKSACK** ist **NP**-vollständig.*

Beweis: Zeige **3KNF-SAT** $\leq_p$ **RUCKSACK**.

Sei F Formel in **3KNF**.

O.B.d.A: jede Klausel in F enthält genau drei Literale:

$$F = (z_{11} \vee z_{12} \vee z_{13}) \wedge \ldots \wedge (z_{m1} \vee z_{m2} \vee z_{m3})$$

mit z_{jk} aus $\{x_1, \ldots, x_n\} \cup \{\neg x_1, \ldots, \neg x_n\}$.

Wähle $b = \underbrace{4\ldots4}_{m}\underbrace{1\ldots1}_{n}$ (im Dezimalsystem!)

Gesucht: Zahlen $a_1, \ldots, a_k$...

Menge $\{a_1, \ldots, a_k\}$ besteht aus vier Klassen von Zahlen:

- Zahlen $v_1, \ldots, v_n$ mit

$$v_j = n_1 \ldots n_m \underbrace{0\ldots0}_{j-1} 1 \underbrace{0\ldots0}_{n-j-1}$$

  wobei n_i= Anzahl des Vorkommens der Variablen x_i in Klausel i
- Zahlen $v'_1, \ldots, v'_n$ mit

$$v'_j = n'_1 \ldots n'_m \underbrace{0\ldots0}_{j-1} 1 \underbrace{0\ldots0}_{n-j-1}$$

  wobei n'_i= Anzahl des Vorkommens des Literals $\neg x_i$ in Klausel i
- Zahlen $c_1, \ldots, c_m$ mit

$$c_j = \underbrace{0\ldots0}_{j-1} 1 \underbrace{0\ldots0}_{m-j-1} \underbrace{0\ldots0}_{n}$$

- Zahlen $d_1, \ldots, d_m$ mit

$$d_j = \underbrace{0\ldots0}_{j-1} 2 \underbrace{0\ldots0}_{m-j-1} \underbrace{0\ldots0}_{n}$$

Beispiel:

$$F = (x_1 \vee \neg x_3 \vee x_5) \wedge (\neg x_1 \vee x_4 \vee x_5) \wedge (\neg x_2 \vee \neg x_2 \vee \neg x_5)$$

mit $m = 3$, $n = 5$ und

$v_1 = 100\ 10000$	$v'_1 = 010\ 10000$
$v_2 = 000\ 01000$	$v'_2 = 002\ 01000$
$v_3 = 000\ 00100$	$v'_3 = 100\ 00100$
$v_4 = 010\ 00010$	$v'_4 = 000\ 00010$
$v_5 = 110\ 00001$	$v'_5 = 001\ 00001$
$c_1 = 100\ 00000$	$d_1 = 200\ 00000$
$c_2 = 010\ 00000$	$d_2 = 020\ 00000$
$c_3 = 001\ 00000$	$d_3 = 002\ 00000$

Hat F eine erfüllende Belegung, so betrachte folgende Liste A von Zahlen aus $\{v_1, \ldots, v_n, v'_1, \ldots, v'_n, c_1, \ldots, c_m, d_1, \ldots, d_m\}$

- Falls $x_k = 1$, so füge v_k zu A
- Falls $x_k = 0$, so füge v'_k zu A
- Summe der Zahlen bisher: $z_1 \ldots z_m \underbrace{1 \ldots 1}_{n}$ mit $1 \leq z_i \leq 3$
- Falls $z_i = 1$ für $1 \leq i \leq m$, so füge c_i und d_i zu A
- Falls $z_i = 2$ für $1 \leq i \leq m$, so füge d_i zu A
- Falls $z_i = 3$ für $1 \leq i \leq m$, so füge c_i zu A

Damit

$$\sum_{a \in A} a = b$$

Sei andererseits

$$\sum_{a \in A} a = b = \underbrace{4 \ldots 4}_{m} \underbrace{1 \ldots 1}_{n}$$

Mit $c_1 + \ldots + c_m + d_1 + \ldots + d_m = \underbrace{3 \ldots 3}_{m} \underbrace{0 \ldots 0}_{n}$ gilt

- entweder v_k in A oder v'_k in A für $1 \leq k \leq n$
- v_k, v'_k nicht beide gleichzeitig in A!
- Summe der v_k, v'_k in A: $z_1 \ldots z_m \underbrace{1 \ldots 1}_{n}$ mit $z_i \geq 0$

Also: F erfüllt durch Belegung mit

$$x_k = 1 \Leftrightarrow v_k \in A$$

Satz 14.6

***PARTITION** ist **NP**-vollständig.*

Beweis: Zeige **RUCKSACK** $\leq_p$ **PARTITION**

Zu Rucksackproblem $(a_1, \ldots, a_k, b)$ und $M = a_1 + \ldots + a_k$ wähle

$$(a_1, \ldots, a_k, b) \mapsto (a_1, \ldots, a_k, M - b + 1, b + 1)$$

Ist Indexmenge $I \subseteq \{1, \ldots, k\}$ Lösung des Problems **RUCKSACK**, dann ist $I \cup \{k + 1\}$ Lösung von **PARTITION**:

$$\sum_I a_i + M - b + 1 = b + M - b + 1 = M - b + b + 1 = \sum_{C(I)} a_i + b + 1$$

Dabei bezeichnet $C(I)$ das Komplement von I.

Andererseits: Sei Lösung von **PARTITION** gegeben

- $M-b+1$ und $b+1$ nicht in gleicher Partition:
- ansonsten wäre Summe dieser Partition $\geq M + 2$, aber Summe der restlichen Zahlen $\leq M$

Also: Partition, die $M-b+1$ enthält, liefert Lösung I für **RUCKSACK**

Satz 14.7

***BIN-PACKING** ist **NP**-vollständig.*

Beweis: Zeige **PARTITION** $\leq_p$ **BIN-PACKING**

Partitionierungsproblem $(a_1, \ldots, a_k)$ sei gegeben.

Falls $\sum_{1 \leq i \leq k} a_i$ ungerade ist:

- wähle beliebige unlösbare Eingabe w für **BIN-PACKING**
- weder $(a_1, \ldots, a_k) \in$ **PARTITION** noch $w \in$ **BIN-PACKING**

Falls $\sum_{1 \leq i \leq k} a_i$ gerade ist:

- Anzahl der Behälter $k := 2$
- Behältergröße $b := \sum_{1 \leq i \leq k} a_i / 2$
- Wähle Zahlen unverändert als $(a_1, \ldots, a_k)$

Damit
$(a_1, \ldots, a_k) \in$**PARTITION** $\Leftrightarrow ((a_1, \ldots, a_k), 2, b) \in$ **BIN-PACKING**

Satz 14.8

*Das **GERICHTETE HAMILTON-KREIS** Problem **GHK** ist **NP**-vollständig.*

Satz 14.9

*Das **UNGERICHTETE HAMILTON-KREIS** Problem **UHK** ist **NP**-vollständig.*

Satz 14.10

*Das **TRAVELING SALESMAN**-Problem **TSP** ist **NP**-vollständig.*

Satz 14.11

*Das **Färbbarkeits**problem ist **NP**-vollständig.*

Beweise: vgl. Buch von Schöning

Zusammnhänge zwischen den Komplexitätsklassen und kontextsensitiven (d.h. monotonen) Grammatiken:

Satz 14.12

*Das Wortproblem für monotone Grammatiken ist **NP**-hart.*

Achtung:

- Wortproblem für monotone Grammatiken ist sogar **PSPACE**-vollständig,
- es ist also polynomialem Speicherplatz lösbar,
- alle anderen mit polynomialem Speicherplatz lösbaren Probleme sind auf das Wortproblem reduzierbar
- Bekannt ist: $\mathbf{NP} \subseteq \mathbf{PSPACE}$
- Vermutlich jedoch: $\mathbf{NP} \neq \mathbf{PSPACE}$

Satz 14.13

*Das Problem **Regulär-Inäquivalenz**, für zwei reguläre Ausdrücke festzustellen, ob sie inäquivalent sind, d.h. ob die zugehörigen Sprachen nicht identisch sind, ist **NP**-hart.*

Beweis: Zeige **3KNF-SAT** $\leq_p$ **Regulär-Inäquivalenz**

Sei $F = K_1 \wedge \ldots \wedge K_m$ eine KNF-Formel mit Variablen $x_1, x_2, \ldots, x_n$

Konstruiere reguläre Ausdrücke α und β über dem Alphabet $\{0, 1\}$:

$$\alpha := (\alpha_1|\alpha_2|\ldots|\alpha_m)$$

wobei $\alpha_i = \gamma_{i,1}\ldots\gamma_{i,n}$ ist mit

$$\gamma_{i,j} = \begin{cases} 0 & \text{falls } x_j \text{ in } K_i \text{ vorkommt} \\ 1 & \text{falls } \neg x_j \text{ in } K_i \text{ vorkommt} \\ (0|1) & \text{sonst} \end{cases}$$

Sei $a_1, \ldots, a_n$ eine Belegung der Variablen $x_1, \ldots, x_n$:

- $a_1, \ldots, a_n$ erfüllt die Klausel K_i genau dann *nicht*, wenn das Wort $a_1 \ldots a_n$ aus $L(\alpha_i)$ ist.
- $a_1, \ldots, a_n$ erfüllt die Formel F genau dann *nicht*, wenn sie eine der Klauseln nicht erfüllt, also genau dann, wenn das Wort $a_1 \ldots a_n$ aus $L(\alpha)$ ist.

Setze daher $\beta = \underbrace{(\mathbf{0}|\mathbf{1})(\mathbf{0}|\mathbf{1})\ldots(\mathbf{0}|\mathbf{1})}_{n}$, also $L(\beta) = \{\mathbf{0}, \mathbf{1}\}^n$.

Damit: F ist erfüllbar genau dann, wenn $L(\alpha) \neq L(\beta)$.

$F \mapsto (\alpha, \beta)$ reduziert also **3KNF-SAT** auf **Regulär-Inäquivalenz**

Anmerkung:

- Umwandlung 'regulärer Ausdruck $\rightarrow$ NEA' in polynomialer Zeit
- Äquivalenztest für DEA ebenfalls in polynomialer Zeit
- Aber: Umwandlung 'NEA $\rightarrow$ DEA' ist zu zeitaufwendig...

$G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ seien ungerichtete Graphen.

Eine Bijektion p von V_1 nach V_2 heißt Isomorphismus zwischen G_1 und G_2, falls gilt: $(v, w) \in E_1 \Leftrightarrow (p(v), p(w)) \in E_2$.

Satz 14.14 (Isomorphie von Graphen)

Betrachte folgende graphentheoretische Probleme:

GRAPH-ISOMORPHIE

- *gegeben: Ungerichtete Graphen $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$*
- *gefragt: Sind G_1 und G_2 isomorph?*

SUBGRAPH-ISOMORPHIE

- *gegeben: Ungerichtete Graphen $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$*
- *gefragt: Ist G_2 zu einem Subgraphen von G_1 isomorph, d.h. gibt es $V_1' \subseteq V_1$, so dass G_2 und $(V_1', E_1 \cap V_1' \times V_1')$ isomorph sind?*

Dabei gilt:

- ***SUBGRAPH-ISOMORPHIE** ist NP-vollständig.*
- ***GRAPH-ISOMORPHIE** $\in$ NP, aber Vollständigkeit unbekannt!*

- Sowohl **SUBGRAPH-ISOMORPHIE** als auch **GRAPH-ISOMORPHIE** liegen offensichtlich in NP.
- **CLIQUE**$\leq_p$ **SUBGRAPH-ISOMORPHIE** über die Reduktion

$$(G, k) \mapsto (G, C_k)$$

  wobei C_k der vollständige Graph mit k Knoten ist (d.h. die Clique der Größe k).
- Vermutung: ***GRAPH-ISOMORPHIE*** *weder NP-vollständig noch in P.*

Daher immerhin:

- Ist **GRAPH-ISOMORPHIE** reduzierbar auf ein Problem A, so ist A wahrscheinlich nicht in P...

Wichtige Teilklasse von NP: Constraint Satisfaction Probleme (CSP).

Aufbau der Probleme:

- n Variablen $x_1, \ldots, x_n$ für Werte aus endlichem Grundbereich D; Lösungsraum ist die Menge D^n.
- m Constraints $C_1, \ldots, C_m$, d.h. 0-1-wertige Funktionen auf D^n. C_j ist "erfüllt" für ein $(a_1, \ldots, a_n) \in D^n$, wenn $C_j(a_1, \ldots, a_n) = 1$
- Wenn jedes C_j nur von maximal k Variablen abhängt, hat das Problem die Ordnung k.

Angabe der Constraints geeignet codiert (Formeln, Graphen, o.ä..)

dabei: Überprüfung eines einzelnen Constraints sehr schnell möglich

Aufgabe: Gibt es eine Wertebelegung $(a_1, \ldots, a_n) \in D^n$ für die Variablen, so dass alle Constraints erfüllt sind?

Beispiele:

- **3-KNF-SAT** ist ein CSP:
  - mit $D = \{0, 1\}$
  - mit Klauseln als Constraints
  - der Ordnung **3**
- allgemein: k-KNF-SAT bei maximal k Variablen pro Klausel als CSP mit $|D| = 2$ und Ordnung k
- k**-Färbbarkeit** ist CSP:
  - $D = \{1, \ldots, k\}$ Menge der Farben
  - Constraints = Kanten,
    $C_{(u,v)}$ ist erfüllt, wenn u und v verschieden gefärbt sind
    (also Ordnung $= 2$)

Generell: CSPs sind NP-vollständig, wenn

- $|D| \geq 2$ und Ordnung ≥ 3 oder
- $|D| \geq 3$ und Ordnung ≥ 2

Sonderfall:$|D| = 2$ und Ordnung $= 2$, z.B. **2-KNF-SAT**

Satz 14.15 (2-KNF-SAT)

Das folgende Problem liegt in P:

2KNF-Sat

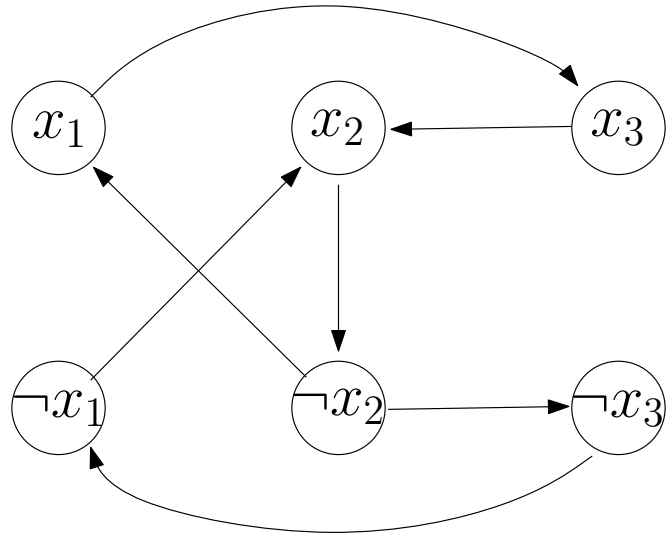
- *gegeben: Eine Boolesche Formel **F** in konjunktiver Normalform mit höchstens **2** Literalen pro Klausel.*
- *gefragt: Ist **F** erfüllbar?*

Beweisidee:

- Gegeben 2-KNF-Formel F mit Variablen $x_1, x_2, \ldots, x_n$.
- O.B.d.A: zwei Literale pro Klausel (statt (a) verwende $(a \vee a)$)
- Klausel $(a \vee b)$ entspricht Implikation $(\neg a \rightarrow b)$ und $(\neg b \rightarrow a)$
- Betrachte gerichteten Graphen $G_F = (V, E)$ mit Knotenmenge $V = \{x_1, \ldots, x_n, \neg x_1, \ldots, \neg x_n\}$
- Zu E: Für Klausel $(a \vee b)$ verwende Kanten $(\neg a, b)$ und $(\neg b, a)$
- Damit: Klausel mit einem Literal (a) ergibt Einzelkante $(\neg a, a)$

Beispiel: Graph G_F zur folgenden Formel F:

$$F = \underbrace{(x_1 \vee x_2)}_{1} \wedge \underbrace{(\neg x_1 \vee x_3)}_{2} \wedge \underbrace{(x_2 \vee \neg x_3)}_{3} \wedge \underbrace{(\neg x_2)}_{4}$$



F ist nicht erfüllbar:

in Formel F: x_2 falsch $\overset{1}{\rightsquigarrow}$ x_1 wahr $\overset{2}{\rightsquigarrow}$ x_3 wahr $\overset{3}{\rightsquigarrow}$ x_2 wahr $\overset{4}{\rightsquigarrow}$ x_2 falsch

im Graphen: $\neg x_2 \longrightarrow x_1 \longrightarrow x_3 \longrightarrow x_2 \longrightarrow \neg x_2$

Also: Nutze Erreichbarkeit im Graphen G_F!

Betrachte transitive Hülle $G_F^* = (V, E^*)$ des Graphen G_F,
d.h. $(a, b) \in E^* \Leftrightarrow$ es gibt in E Pfad von a nach b.

E^* kann in polynomialer Zeit berechnet werden
(z.B. mit Warshall-Algorithmus, kubische Komplexität in n)

Lemma 14.16

Eine 2-KNF-formel F ist genau dann erfüllbar, wenn für kein i in G_F ein Kreis der Form $x_i \to \cdots \to \neg x_i \to \cdots \to x_i$ existiert, d.h. wenn in G_F^* nie beide Kanten $(x_i, \neg x_i)$ und $(\neg x_i, x_i)$ existieren.

Beweis von '$\Longrightarrow$': $(a \vee b)$ entspricht Kanten $(\neg a, b), (\neg b, a) \in E$

Bei einer erfüllenden Belegung von F gilt damit

1. Ist Literal a wahr, sind alle Literale b wahr mit $(a, b) \in E^*$.
2. Ist Literal b falsch, sind alle Literale a falsch mit $(a, b) \in E^*$.
3. Ist $(\neg a, a) \in E^*$ für Literal a, so muss a wahr sein.

Damit direkt "F erfüllbar $\Rightarrow \neg\exists$ Kreis".

‘$\Longleftarrow$’: Für die Rückrichtung definiere ‘Belegung’ (mit $\{\mathbf{0}, \mathbf{1}\}$) wie folgt:

(1) Für alle Literale mit $(\neg \boldsymbol{a}, \boldsymbol{a}) \in \boldsymbol{E}^*$ setze $\boldsymbol{a} \mapsto \mathbf{1}$ (damit $\neg \boldsymbol{a} \mapsto \mathbf{0}$).

(2) dann ergänze für diese $\boldsymbol{a}$:

- Setze alle $\boldsymbol{b}$ mit $(\boldsymbol{a}, \boldsymbol{b}) \in \boldsymbol{E}^*$ auf $\mathbf{1}$.
- Setze alle $\boldsymbol{b}$ mit $(\boldsymbol{b}, \neg \boldsymbol{a}) \in \boldsymbol{E}^*$ auf $\mathbf{0}$.

(3) Solange noch nicht alle Literale einen Wahrheitswert haben:

- Wähle beliebiges(!) noch nicht gesetztes Literal $\boldsymbol{a}$, setze es auf $\mathbf{1}$.
- Setze wieder alle $\boldsymbol{b}$ mit $(\boldsymbol{a}, \boldsymbol{b}) \in \boldsymbol{E}^*$ auf $\mathbf{1}$.
- Setze wieder alle $\boldsymbol{b}$ mit $(\boldsymbol{b}, \neg \boldsymbol{a}) \in \boldsymbol{E}^*$ auf $\mathbf{0}$.

Gibt es keine Kanten $(\boldsymbol{x_i}, \neg \boldsymbol{x_i})$ und $(\neg \boldsymbol{x_i}, \boldsymbol{x_i})$ in $\boldsymbol{E}^*$,
so ist die Belegung wohldefiniert und erfüllt $\boldsymbol{F}$:
Nie folgt auf $\mathbf{1}$ im Graphen $\boldsymbol{G}$ eine $\mathbf{0}$, d.h. alle Klauseln sind erfüllt.

Anmerkungen zu *NP*-vollständigen Problemen:

- Viele ***NP***-vollständige Probleme mit großer praktischer Bedeutung
- Effiziente Algorithmen sind jedoch nicht bekannt...
- daher: Versuche, Problematik zu umgehen...

z.B. betrachte mittlere Laufzeit statt Worst-Case-Komplexität:

- z.B. für Hamiltonkreise: deterministische Algorithmen mit polynomialer mittlerer Laufzeit
- Vorsicht: Mittelwert gilt für gewisse Verteilung der Eingaben
- falls bei Anwendung andere Verteilung: exponentielle Laufzeit....

z.B. statt exakter Lösung nur Approximation der Lösung gewünscht:

- z.B. Traveling Salesman: 'günstige' Route reicht oft
- Route z.B. bis zu 50% teurer als eine optimale Reiseroute (+ weitere Voraussetzungen): deterministischer polynomialer Algorithmus existiert!

Allerdings:
Manchmal auch Lösungs-Approximationen noch ***NP***-vollständig...

15 ‘Harte’ Probleme

- Typische Problemklassen
- Exakte Verfahren
- Approximative Verfahren
- Randomisierte Verfahren

Grundform einer algorithmischen Aufgabenstellung:

- i.d.R. nicht als Entscheidungsproblem
- sondern sondern als funktionales Problem (oder Suchproblem)
  Gegeben ***x***, bestimme eine „Lösung" ***y***, falls eine solche existiert.

y ist dabei i.d.R. Resultat der Guess-Phase bei Guess-and-Check und kann schnell überprüft werden
(Bsp: Variaben-Belegung, Isomorphismus, Färbung,...)

Wichtig auch: Bei nichtdeterministischen Algorithmen ist es nicht gleichgültig, ob man

- die Existenz einer Lösung zeigen will oder
- aber zeigen will, dass gar keine Lösung existiert

Definition 15.1

Zu einem Problem K aus einer Grundmenge M sei co-$K := M \setminus K$. Die Problemklasse co-NP besteht aus allen Problemen K, für die co-K in NP liegt.

Es ist noch unbekannt, ob co-NP=NP gilt (vermutlich nicht...)

- Unbekannt ist z.B. ob co-SAT in NP liegt
  (Wie soll man durch Raten feststellen, dass z.B. keine erfüllende Belegung existiert?)
- Bei deterministischen Entscheidungsalgorithmen kann man das Resultat einfach negieren, d.h. co-P=P.

weitere Form: Optimierungsprobleme

- Zu Eingabe bestimme Lösung, die eine gegebene Kostenfunktion maximiert (oder minimiert)

Bei NP-vollständigen Problemen:

- funktionales Problem oder Optimierungsproblem auf Entscheidungsproblem polynomial “reduzierbar”
- gibt es polynomialen Algorithmus für das Entscheidungsproblem, so auch für das funktionale oder Optimierungsproblem

Entscheidungsprobleme für Theorie ausreichend
(aber nicht für Praxis: hier genauere Betrachtung notwendig...)

Optimierungsprobleme werden spezifiziert durch:

- die Menge der zulässigen Eingaben
- die zulässigen Lösungen $\boldsymbol{S}(\boldsymbol{x})$ für zulässige Eingaben $\boldsymbol{x}$
- eine Bewertungsfunktion $\boldsymbol{v}$, die Lösungen $\boldsymbol{y}$ mit $\boldsymbol{v}(\boldsymbol{y})$ bewertet.

Jede der drei Komponenten sollte polynomiale Komplexität haben:

- Zulässigkeit sollte mit polynomialer Komplexität überprüfbar sein!
- $\boldsymbol{y} \in \boldsymbol{S}(\boldsymbol{x})$ sollte in polynomialer Komplexität entscheidbar sein.
- $\boldsymbol{v}$ sei in polynomialer Komplexität berechenbar.

Optimierungsprobleme sind gegliedert in

- Minimierungsprobleme:
  suche Lösung $\boldsymbol{y} \in \boldsymbol{S}(\boldsymbol{x})$ mit $\boldsymbol{v}(\boldsymbol{y}) = \boldsymbol{min}\{\boldsymbol{v}(\boldsymbol{s}) \mid \in \boldsymbol{S}(\boldsymbol{x})\}$.
- Maximierungsprobleme:
  suche analog Lösung mit maximalem $\boldsymbol{v}$-Wert

Beispiel 15.2

Traveling Salesman Problem *(TSP) als Minimierungsproblem:*

- *zulässige Eingaben:* $\boldsymbol{n \times n}$ *Entfernungsmatrix* $\boldsymbol{M = (m_{i,j})}$
  (Abstände/Kosten zuje zwei Städten $\boldsymbol{i}$ *und* $\boldsymbol{j}$*,* $(\boldsymbol{i, j \in \{1, \ldots, n\}})$*)*
- *Erlaubt: Einträge mit Wert* ∞
  (keine direkte Straßenverbindung zwischen Ort $\boldsymbol{i}$ *und Ort* $\boldsymbol{j}$*)*
- *zulässige Lösungen sind Permutationen* π *auf* $\{1, 2, \ldots, n\}$
  (repräsentieren Rundreise durch alle Städte)
  zulässige Rundreise müssen ohne ∞*-Einträge auskommen*
- *Bewertung* $\boldsymbol{v}$ *einer zulässigen Permutation* π *für* $\boldsymbol{M}$ *ist definiert als*

$$\boldsymbol{v}(\pi) = \sum_{k=1}^{n-1} \boldsymbol{m}_{\pi(k),\pi(k+1)} + \boldsymbol{m}_{\pi(n),\pi(1)}$$

- *O.B.d.A. betrachte nur Permutationen* π *mit* $\pi(1) = 1$*.*

Ziel: ***Suche*** π ***mit minimalen Gesamtkosten*** $\boldsymbol{v}(\pi)$

Beispiel 15.3

MaxSAT *als Maximierungsproblem:*

- *Gegeben: eine* Menge $\boldsymbol{F}$ *von Klauseln,*
  gesucht: eine Belegung, die möglichst viele Klauseln erfüllt.
- *Zulässige Eingaben: syntaktisch korrekte Klauselmengen.*
- *Jede Belegung* Φ *der Variablen ist zulässige Lösung!*
- *Bewertung* $\boldsymbol{v}(\Phi) =$ *Anzahl der durch* Φ *erfüllten Klauseln*

Beispiel:

$$\boldsymbol{F} = \{(\boldsymbol{x_1} \vee \boldsymbol{x_2} \vee \boldsymbol{x_3}), (\neg \boldsymbol{x_2} \vee \boldsymbol{x_3}), \neg \boldsymbol{x_1}, \neg \boldsymbol{x_2}, \neg \boldsymbol{x_3}\}$$

mit (nicht-eindeutiger aber) optimaler Lösung

$$\Phi : \boldsymbol{x_1} \mapsto \boldsymbol{1}, \boldsymbol{x_2} \mapsto \boldsymbol{0}, \boldsymbol{x_3} \mapsto \boldsymbol{0}$$

und $\boldsymbol{v}(\Phi) = \boldsymbol{4}$.

Umwandlung vom Optimierungs- zum Entscheidungsproblem:

- natürliche Zahl $\boldsymbol{k}$ als weiterer Bestandteil der Eingabe, d.h. Eingaben sind

$$\{(\boldsymbol{x}, \boldsymbol{k}) \mid \boldsymbol{x} \text{ ist zulässig}, \boldsymbol{k} \in \mathbb{N}\}$$

- gesucht: Gibt es Lösung der Güte $\boldsymbol{k}$ oder besser?

Bei Minimierungsproblemen lautet das Entscheidungsproblem also:

- Ist zu $(\boldsymbol{x}, \boldsymbol{k})$ die Menge $\{\boldsymbol{y} \in \boldsymbol{S}(\boldsymbol{x}) \mid \boldsymbol{v}(\boldsymbol{y}) \leq \boldsymbol{k}\}$ nichtleer?

weitere Form: Zählprobleme

- Zu Eingabe bestimme genaue Anzahl der Lösungen

Für NP-vollständige Probleme sind die Zählprobleme wohl schwieriger als die Entscheidungsprobleme!

Andererseits beim Graphenisomorphieproblem:

- Aus fiktiven polynomialen Entscheidungsalgorithmus würde polynomialer Algorithmus für das Zählproblem (ohne Beweis...)
- daher Vermutung: Graphenisomorphieproblem ist evtl. nicht NP-vollständig...

Im Folgenden: Algorithmen für NP-vollständige Probleme

- insbesondere für Traveling Salesman Problem und Erfüllbarkeitsproblem als Beispiele

Algorithmen für exakte Lösungen (für funktionales oder Optimierungsproblem) haben (derzeit) exponentielle Laufzeit

Daher folgende Möglichkeiten:

- Suche nach exakten Algorithmen, die zumindest besser sind als naive Suche
- Suche nach polynomialen Algorithmen, die akzeptable Näherungslösungen berechnen
- Benutze dabei insbesondere Zufallszahlen als Hilfsmittel

15 ‘Harte’ Probleme

- Typische Problemklassen
- Exakte Verfahren
  - Dynamisches Programmieren
  - Backtracking
  - Branch and Bound
- Approximative Verfahren
- Randomisierte Verfahren

Bekannt: **Divide and Conquer**

- *Top-Down*-Verfahren
- Zerlege Problem in Teilprobleme, die getrennt gelöst werden
- Einfaches Zusammensetzen der Teillösungen
- Wichtig für Komplexität: Unabhängigkeit der Teilprobleme
- Parade-Beispiel: Quicksort
- Negativ-Beispiel: Fibonacci-Zahlen

Jetzt: **Dynamisches Programmieren**

- *Bottom-Up*-Verfahren
- Löse erst triviale Probleme
- Setze Teillösungen zu größeren Lösungen zusammen
- Wichtig für Algorithmus: Speicherung der Teillösungen!
- Bereits bekannt: CYK-Algorithmus für kontextfreie Grammatiken

Beispiel: **TSP**

Naiver Algorithmus:

- Untersuche alle Permutationen (d.h. $\Omega(\boldsymbol{n}!)$)...
- Triviale Verbesserung: Starte immer in Stadt **1** (d.h. $\Omega((\boldsymbol{n}-\mathbf{1})!)$)

Laufzeit (geschätzt, bei 1.000.000 Permutationen pro Sekunde):

n	Zeit
10	**0.3 *s***
11	**3.6 *s***
12	**40 *s***
13	**8 *m***
14	**1 *h* 43 *m***
15	**1 *d***
16	**15 *d***
17	**240 *d***
18	**11 *j***
19	**202 *j***
20	**3836 *j***

Uminterpretation von **TSP**:

- Betrachte **TSP** als (kantenbewerteten) Graphen
- Knotenmenge $V = \{1, \ldots, n\}$
- Kantenmenge $E = V \times V$, Bewertung: $m_{i,j}$ für Kante (i, j)

Teil-Lösungen für **TSP**:

- zu Knoten i und $S \subset V$ setze

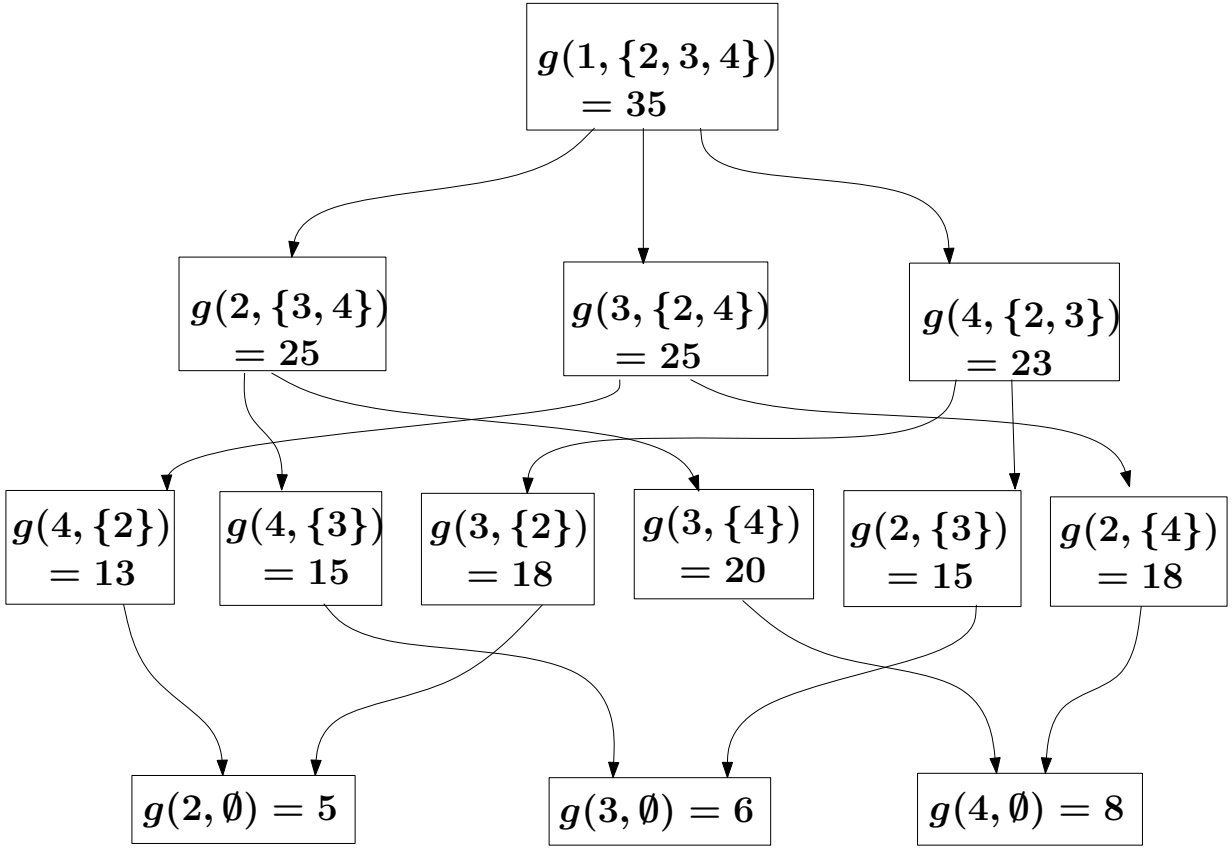
$$g(i, S) = \text{Gewicht des besten Weges von } i \text{ nach } 1, \text{ der nur über Knoten aus } S \text{ führt und zudem jeden Knoten } j \in S \text{ genau einmal berührt}$$

- Dann ist $g(1, V \setminus \{1\})$ gesuchte Lösung des TSP
- Rekursive Beschreibung von g:

$$g(i, S) = \begin{cases} m_{i,1} & S = \emptyset \\ \min_{j \in S} \left(m_{i,j} + g(j, S \setminus \{j\}) \right) & S \neq \emptyset \end{cases}$$

Beispiel:

$$\boldsymbol{M} = \begin{bmatrix} \mathbf{0} & \mathbf{10} & \mathbf{15} & \mathbf{20} \\ \mathbf{5} & \mathbf{0} & \mathbf{9} & \mathbf{10} \\ \mathbf{6} & \mathbf{13} & \mathbf{0} & \mathbf{12} \\ \mathbf{8} & \mathbf{8} & \mathbf{9} & \mathbf{0} \end{bmatrix}$$



Rekursive Implementierung $\rightsquigarrow$ ‘überflüssige’ Auswertungen von $\boldsymbol{g}$!

‘**Dynamisches Programm**’ für **TSP**:

- Bottom-Up Auswertung
- Speicherung der Funktionswerte in Tabelle

Iterativ programmiert:

```
FOR i:=2 TO n DO g[i,∅]=m[i,1];
FOR k:=1 TO n−2 DO
    FOR S, |S|=k, 1∉S DO
        FOR i ∈{2,...,n}\S DO
           berechne g[i,S] nach Formel
berechne g[1,{2,...,n}] nach Formel
```

Aufwand:

- Speicherplatz: Tabelle mit $< n \cdot 2^n$ Einträgen
- Zeit: (Größe der Tabelle) $\cdot$ (Aufwand pro Eintrag), d.h. $n^2 \cdot 2^n$

(Hilfsmittel zum Programmieren mit (kleinen) Mengen in `C++` oder `Java` z.B. `bitset`)

Vergleich der Laufzeiten:
(Intel Core i5, 2.53GHz)

	rekursiver Algorithmus	dyn.Prog, Zeit	dyn.Prog, Speicher
n	$(\approx (n-1)!\ )$	$(\approx n^2 \cdot 2^n)$	$(\approx n \cdot 2^n)$
10	**0.01 *s***	$<$ **0.01 *s***	?
11	**0.3 *s***	$<$ **0.01 *s***	?
12	**4 *s***	$<$ **0.01 *s***	?
13	**48 *s***	$<$ **0.01 *s***	?
14	**650 *s***	**0.015 *s***	?
16	?	**0.02 *s***	**5 *MB***
18	?	**0.1 *s***	**12 *MB***
20	?	**0.6 *s***	**43 *MB***
22	?	**2.7 *s***	**180 *MB***
24	?	**13 *s***	**770 *MB***
25	?	**32 *s***	**1.6 *GB***

leichte Reduktion des Speichers auf Kosten der Zeit möglich:

- für $g(i, S)$ nur $g(j, S')$ mit $|S'| = |S| + 1$ benötigt
- ca. *halber* Speicherbedarf bei $n = 24$ ...

weiteres Beispiel: $\mathbf{0}/\mathbf{1}$-Rucksack-Problem

- Gegeben: $\boldsymbol{n} \in \mathbb{N}$, $\boldsymbol{G} \in \mathbb{N}$, Vektoren $(\boldsymbol{v_1}, ..., \boldsymbol{v_n}), (\boldsymbol{g_1}, ..., \boldsymbol{g_n}) \in \mathbb{N}^{\boldsymbol{n}}$
- Interpretation: $\boldsymbol{n}$ Objekte mit
  - Werten/Kosten/Gewinne ($\boldsymbol{v}$-Vektor)
  - Gewichten ($\boldsymbol{g}$-Vektor)
  - (Gewichts-)Kapazität $\boldsymbol{G}$
- Gesucht: Wie groß kann der 'Wert' $\sum_{i \in I} \boldsymbol{v_i}$ einer Teilmenge $\boldsymbol{I}$ von Objekten werden,
  deren Gewicht $\sum_{i \in I} \boldsymbol{g_i}$ nicht größer als die Schranke $\boldsymbol{G}$?
- formal: bestimme

$$\max\{\sum_{i \in I} \boldsymbol{v_i} \mid \boldsymbol{I} \subseteq \{\mathbf{1}, ..., \boldsymbol{n}\} \wedge \sum_{i \in I} \boldsymbol{g_i} \leq \boldsymbol{G}\}$$

Bekanntes (NP-vollständiges) Rucksack-Problem ist zugehöriges Entscheidungsproblem mit:

- $(\boldsymbol{v_1}, ..., \boldsymbol{v_n}) = (\boldsymbol{g_1}, ..., \boldsymbol{g_n})$
- Fragestellung

$$\boldsymbol{G} = \max\{\sum_{i \in I} \boldsymbol{v_i} \mid \boldsymbol{I} \subseteq \{\mathbf{1}, ..., \boldsymbol{n}\} \wedge \sum_{i \in I} \boldsymbol{g_i} \leq \boldsymbol{G}\}$$

Formulierung von Teil-Lösungen:

- bei $n = 1$ (Wert v_1, Gewicht g_1, Schranke G):
  Lösung trivial:
  - v_1, falls $g_1 \leq G$
  - 0, falls $g_1 > G$
- bei $n > 1$ Objekten teste zwei Möglichkeiten:
  - Objekt n gehört zu optimaler Menge I:
    I nutzt Teil-Lösung $I' \subseteq \{1, \ldots, n-1\}$ mit Schranke $G - g_n$
  - Objekt n gehört nicht zu optimaler Menge I:
    I ist identisch zu Teil-Lösung $I' \subseteq \{1, \ldots, n-1\}$ mit Schranke G,

Daher setze

$$w(i, h) := \left\{ \begin{array}{l} \text{maximaler Lösungswert} \\ \text{für erste } i \text{ Objekte bei Schranke } h \end{array} \right.$$

mit rekursiver Formulierung

$$w(i, h) = \begin{cases} 0 & i = 0 \\ w(i-1, h) & i > 0 \wedge h < g_i \\ \max \left\{ \begin{array}{l} w(i-1, h), \\ w(i-1, h-g_i) + v_i \end{array} \right\} & \text{sonst} \end{cases}$$

Dynamisches Programm dazu:

- verwende Tabelle für $w(i, h)$ mit $0 \leq i < n$ und $0 \leq h \leq G$

```
FOR i:=0 TO n-1 DO
    FOR h:=0 TO G DO
            berechne w[i,h] nach Formel
berechne w[n,G] nach Formel
```

- Speicherplatz: $n \cdot (G + 1)$ (optimierbar zu $2(G + 1)$)
- Zeitaufwand: $n \cdot (G + 1) \cdot$ (Aufwand für Formelauswertung)

Uniformes Kostenmaß:

- Aufwand $O(n \cdot G)$, d.h. linear in n
- Bei 'beschränktem' G ist $0/1$-Rucksack also **polynomial**!

Wichtig also: betrachte auch G!

- falls G k Bits hat: Aufwand $O(n \cdot 2^k)$, d.h. exponentiell in k
- vgl. Reduktion 3-KNF $\mapsto$ RUCKSACK (Satz 14.5)
  - erzeugt Rucksackproblem mit $n \approx k$,
  - dynamisches Programm hat hier exponentielle Komplexität!

Alternative Vorgehensweise:

- Betrachte Teil-Lösungen für Kosten (und nicht für Gewichte)!
- definiere $\boldsymbol{g}(\boldsymbol{i}, \boldsymbol{v}) :=$ minimales Gewicht für Mengen von Objekten aus $\{\mathbf{1}, \ldots, \boldsymbol{i}\}$, die zu den Kosten $\boldsymbol{v}$ führen
- dann wieder rekursiv $\boldsymbol{g}(\boldsymbol{i}, \boldsymbol{v})$ über $\boldsymbol{g}(\boldsymbol{j}, \boldsymbol{v'})$ mit $\boldsymbol{j} < \boldsymbol{i}$ definierbar
- setze dabei insbesondere $\boldsymbol{g}(\mathbf{0}, \mathbf{0}) = \mathbf{0}$ und $\boldsymbol{g}(\mathbf{0}, \boldsymbol{v}) = \infty$ für $\boldsymbol{v} > \mathbf{0}$
- Einzelheiten: Übung...

15 ‘Harte’ Probleme

- Typische Problemklassen
- Exakte Verfahren
  - Dynamisches Programmieren
  - Backtracking
  - Branch and Bound
- Approximative Verfahren
- Randomisierte Verfahren

Backtracking:

- Durchlauf durch (großen) Suchbaum
- Baum i.d.R. nur implizit gegeben
- wichtiges Teilziel: Ausschluss von Teilbäumen
- i.d.R. rekursive Implementierung

Grundstruktur der Rekursion:

```
PROCEDURE backtrack(Lösungsansatz);
  IF (Lösungsansatz == vollständige Lösung)
    THEN gib Lösung aus
    ELSE FOR (wichtige Erweiterungen des Lösungsansatzes) DO
             backtrack(Lösungsansatz mit Erweiterung)
         ENDFOR
  ENDIF
  RETURN
ENDPROCEDURE
```

Aufruf mit `backtrack(leere Lösung)`

- Rekursion führt implizit zu Baumstruktur!
- Anzahl der Erweiterungen der Lösungsansätze klein halten!
- zur ‘Wichtigkeit’ der Erweiterungen:
  - Jede Erweiterung, die noch zu Lösung werden könnte, ist wichtig!
  - Leicht erkennbar, dass keine Lösung mehr entstehen kann: unwichtige Erweiterung
  - ansonsten: vorsichtshalber als wichtige Erweiterung einstufen...
- falls keine Erweiterung wichtig $\rightsquigarrow$ Teilbaum wird ausgelassen!

Insbesondere bei CSP mit $\boldsymbol{n}$ Variablen, $|\boldsymbol{D}| = \boldsymbol{d}$ und Ordnung $\boldsymbol{k}$:

- Erweiterung eines Lösungsansatzes = Belegung einer weiteren Variablen!
- Teste pro Variable alle $\boldsymbol{d}$ möglichen Werte
- Also: $\boldsymbol{d}$-ärer Suchbaum mit Tiefe $\boldsymbol{n}$, Größe $\boldsymbol{d^n}$
- Sobald für ein Constraint $\boldsymbol{C_j}$ alle Variablen belegt sind:
  Falls $\boldsymbol{C_j}$ dabei falsch wird: Werte weiterer Variablen unwichtig!

Grobe Abschätzung des Effektes:

- Constraint der Ordnung k führt zur d^k Teilbäumen
- Für jedes falsche Resultat: Ein Teilbaum weniger...
- Da jedes Constraint falsch werden kann (sonst trivial...):
  Statt d^k Teilbäumen i.d.R. nur $d^k - 1$
- Komplexität statt d^n nur $(d^k - 1)^{n/k}$

heuristische Verbesserungen:

- Sortiere Variablen nach Ordnung der Constraints:
- Erst Variablen in Constraints mit kleiner Ordnung belegen
- Bei Constraints mit gleicher minimaler Ordnung: Häufigst benutzte Variable zuerst belegen (reduziert Ordnung anderer Constraints!)

Beispiel KNF-SAT:

- Gegeben: Formel F in KNF mit n Variablen
- Gesucht: Erfüllende Belegung Φ

naiver Algorithmus: Durchsuche alle möglichen Belegungen, z.B.

```
Gegeben F mit n Variablen

FOR b := 0 TO power(2, n) − 1 DO
    Interpretiere die Einzelbits von b als Belegung Φ.
    Teste, ob F durch φ erfüllt wird.
    Wenn ja: RETURN TRUE

RETURN FALSE
```

Aufwand

- Formel nicht erfüllbar: $2^n \cdot$(Bestimmung des Wertes von F bei ϕ)
- Formel erfüllbar: schrumpft mit wachsender Zahl an Lösungen...

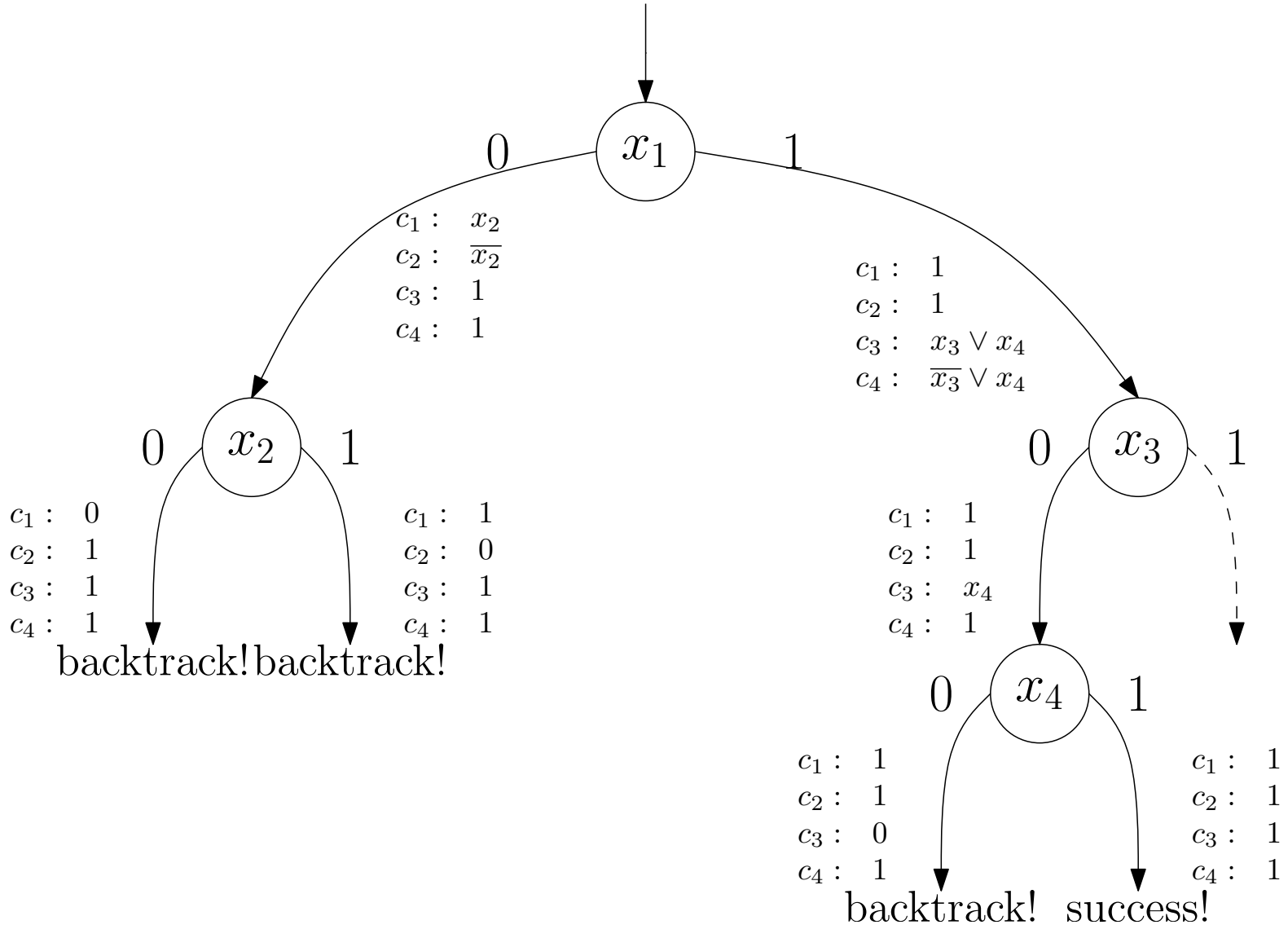
Backtracking-Algorithmus (einfachste Form):

Gegeben: Formel F *mit Variablen* $\{x_1, \ldots, x_n\}$
Belegung Φ *ist String der Länge* n *über* $\{0, 1, u\}$
(mit 0*: falsch,* 1*: wahr,* u*: unbelegt)*

```
BOOLEAN PROCEDURE backtrack(belegung Φ;
  IF (Φ enthält kein u)
    THEN RETURN F(Φ)
  ENDIF
  IF (Φ setzt eine Klausel in F auf 0 )
    THEN RETURN 0
  ENDIF
  wähle unbelegte Variable, Index sei i
  IF  backtrack( Φ[i : 0] )
    THEN RETURN 1
  ENDIF
  RETURN backtrack( Φ[i : 1] )
ENDPROCEDURE

main:
  backtrack(  u...u )
```

Beispiel: $F = \underbrace{(x_1 \vee x_2)}_{c_1} \wedge \underbrace{(x_1 \vee \overline{x_2})}_{c_2} \wedge \underbrace{(\overline{x_1} \vee x_3 \vee x_4)}_{c_3} \wedge \underbrace{(\overline{x_1} \vee \overline{x_3} \vee x_4)}_{c_4}$



Backtracking bei **2-KNF-SAT**:

- Pro Klausel von 4 Wahrheitswerten nur 3 wichtig
- Formal damit statt $\mathbf{2^n}$ nur $(\sqrt{\mathbf{3}})^{\boldsymbol{n}}$
- zudem: jede Variablen-Belegung legt Werte vieler direkt/indirekt verbundener Variablen fest!
- Damit nur noch 2 Versuche pro ‘Zusammenhangskomponente’!
- Topologische Sortierung der Komponenten $\rightsquigarrow$ Polynomialer Algorithmus (bei Modifikation des Backtracking)

Backtracking bei **3-KNF-SAT**:

- Statt $\mathbf{2^n}$ nur $\approx \mathbf{7^{n/3} = 1{,}913^n}$,
- formal: bei $\boldsymbol{n} = \mathbf{30}$ nur 1/4 der Komplexität
- Hauptvorteil aber: Jede Variablenbelegung macht aus manchen 3er-Klauseln kleinere 2er-Klauseln, d.h. dieser Teil dann wie 2-KNF-SAT behandelbar!

15 ‘Harte’ Probleme

- Typische Problemklassen
- Exakte Verfahren
  - Dynamisches Programmieren
  - Backtracking
  - Branch and Bound
- Approximative Verfahren
- Randomisierte Verfahren

Branch and Bound: Kombination zweier Strategien bei Optimierungsproblemen:

- Aufgabe: suche beste von allen(!) Lösungen
- wie bei Backtracking: Ausschluß von Lösungsansätzen
- zudem: bevorzugte Behandlung aussichtsreicher Lösungsansätze

Dazu benötigt:

- Optimierungsproblem mit baumartigem Lösungsalgorithmus
- einfache Bewertungsmöglichkeit g der Güte eines Lösungsansatzes
- evtl. (möglichst gute) Schranke G für die beste Gesamtlösung

Vergleich:

- Backtracking: Suche mit ‘nächster’ Teil-Lösung fortgesetzt
- Branch and Bound: Suche mit ‘bestem’ Lösungsansatz fortgesetzt
- statt Rekursionsstack wird Priority Queue verwendet

Branch-and-Bound-Algorithmus (einfachste Form):

- Gegeben: Mimimierungsproblem
  - mit Bewertungsfunktion v für Lösungen,
  - mit Gütefunktion g ('Bound') für Lösungsansätze
- Bedingung an v und g:
  Ist Lösung L Erweiterung des Ansatzes A, dann $v(L) \geq g(A)$

```
Ansatz          A := leerer Lösungsansatz
Schranke        G := ∞
Lösung          L
Priority Queue pq; pq.insert( A, g(A) )

WHILE pq.not_empty DO
A := pq.extract_min;
  IF (A ist vollständige Lösung)  THEN
     IF ( v(A) < G ) THEN L := A; G := v(A) ENDIF
  ELSE
     FORALL ( E Erweiterung von A ) DO
        IF ( g(E) < G ) THEN  pq.insert( E, g(E) ) ENDIF
     ENDFOR
  ENDIF
ENDDO
RETURN L
```

Beispiel: TSP

- $m_{i,j}$ Kosten/Gewicht der Kante (ij)
- g Gewicht eines Pfades $i_1 i_2 \ldots i_n$

$$g(i_1 i_2 \ldots i_n) = m_{i_1,i_2} + \ldots + m_{i_{n-1},i_n}$$

- v Gewicht einer kompletten Rundreise $i_1 i_2 \ldots i_n$ mit $i_1 = i_n$

$$v(i_1 i_2 \ldots i_n) = g(i_1 \ldots i_n)$$

- bei nichtnegativen(!) Kosten $m_{i,j}$ gilt für $k \leq n$

$$v(i_1 i_2 \ldots i_n) \geq g(i_1 i_2 \ldots i_k)$$

Betrachte TSP mit $M = \begin{bmatrix} - & 10 & 15 & 20 \\ 5 & - & 9 & 10 \\ 6 & 13 & - & 12 \\ 8 & 8 & 9 & - \end{bmatrix}$, Start mit $G = \infty$

Inhalt (Pfad,Gewicht) der Priority Queue:
(1, 0)
(12, 10) **(13, 15)** **(14, 20)**
(13, 15) **(123, 19)** **(124, 20)** **(14, 20)**
(123, 19) **(124, 20)** **(14, 20)** **(134, 27)** **(132, 28)**
(124, 20) **(14, 20)** **(134, 27)** **(132, 28)** **(1234, 31)**
(14, 20) **(134, 27)** **(132, 28)** **(1243, 29)** **(1234, 31)**
(134, 27) **(132, 28)** **(142, 28)** **(143, 29)** **(1243, 29)** **(1234, 31)**
(132, 28) **(142, 28)** **(143, 29)** **(1243, 29)** **(1234, 31)** **(1342, 35))**
(142, 28) **(143, 29)** **(1243, 29)** **(1234, 31)** **(1342, 35)** **(1324, 38)**
(143, 29) **(1243, 29)** **(1234, 31)** **(1342, 35)** **(1423, 37)** **(1324, 38)**
(1243, 29) **(1234, 31)** **(1342, 35)** **(1423, 37)** **(1324, 38)** **(1432, 42)**
erste Lösung gefunden: $L = (12431)$, neue Schranke $G = 35$ **(1234, 31)** **(1342, 35)** **(1423, 37)** **(1324, 38)** **(1432, 42)**
ab hier: ***pq*** wird nur noch geleert...

Optimale Lösung ist $L = (12431)$ mit Gewicht **35**

Verbesserungen des Algorithmus:

(1) Suche besseren Startwert der Schranke $\boldsymbol{G}$
   - z.B. durch Vorab-Bestimmung einer (nicht zu schlechten) Lösung

(2) Bessere Datenstrukturen:
Für TSP statt Pfad besser Paare (Knotenmenge, letzter Knoten)
   - z.B. statt $(\mathbf{1243}, \mathbf{29})$ und $(\mathbf{1423}, \mathbf{37})$ nur $(\ (\{\mathbf{2}, \mathbf{4}\}, \mathbf{3})\ , \mathbf{29})$ in $\boldsymbol{pq}$
   - benötigt `decreaseKey`-Operation für Priority Queues

(3) Versuche die Unterschiede zwischen guten und schlechten Pfaden zu vergrößern...
   - Reduziere Gewichte der Pfade um Anteile, die allen Pfaden gemeinsam sind
   - von Reduktion profitieren insbesondere gute Lösungen

Anwendung von (3) im Beispiel TSP: Reduktion von $\boldsymbol{M}$ wie folgt

- subtrahiere von jeder Zeile in $\boldsymbol{M}$ das Minimum $\boldsymbol{z_i}$ der Zeile $\boldsymbol{i}$, damit Gewicht jeder(!) Lösung um $\sum \boldsymbol{k_i}$ erniedrigt

$$\boldsymbol{M} = \begin{bmatrix} - & 10 & 15 & 20 \\ 5 & - & 9 & 10 \\ 6 & 13 & - & 12 \\ 8 & 8 & 9 & - \end{bmatrix} \quad \rightsquigarrow \quad \boldsymbol{M'} = \begin{bmatrix} - & 0 & 5 & 10 \\ 0 & - & 4 & 5 \\ 0 & 7 & - & 6 \\ 0 & 0 & 1 & - \end{bmatrix}$$

  mit Minima $\mathbf{10, 5, 6, 8}$, d.h. mit Erniedrigung um $\mathbf{10 + 5 + 6 + 8 = 29}$

- subtrahiere von jeder Spalte in $\boldsymbol{M'}$ das Minimum $\boldsymbol{s_i}$ der Spalte $\boldsymbol{i}$, damit Gewicht jeder(!) Lösung weiter um $\sum \boldsymbol{s_i}$ erniedrigt

$$\boldsymbol{M'} = \begin{bmatrix} - & 0 & 5 & 10 \\ 0 & - & 4 & 5 \\ 0 & 7 & - & 6 \\ 0 & 0 & 1 & - \end{bmatrix} \quad \rightsquigarrow \quad \boldsymbol{M''} = \begin{bmatrix} - & 0 & 4 & 5 \\ 0 & - & 3 & 0 \\ 0 & 7 & - & 1 \\ 0 & 0 & 0 & - \end{bmatrix}$$

  mit Erniedrigung um $\mathbf{0 + 0 + 1 + 5 = 6}$

- Optimale Lösungen bei $\boldsymbol{M}$ und $\boldsymbol{M''}$ gleich, aber im Gewicht verschieden, sogar um $\mathbf{29 + 6 = 35}$

Betrachte TSP mit $M'' = \begin{bmatrix} - & 0 & 4 & 5 \\ 0 & - & 3 & 0 \\ 0 & 7 & - & 1 \\ 0 & 0 & 0 & - \end{bmatrix}$, Start mit $G = \infty$

Inhalt (Pfad,Gewicht) der Priority Queue:
$(1, 0)$
$(12, 0)$ $(13, 4)$ $(14, 5)$
$(124, 0)$ $(123, 3)$ $(13, 4)$ $(14, 5)$
$(1243, 0)$ $(123, 3)$ $(13, 4)$ $(14, 5)$
erste Lösung gefunden: $L = (12431)$, neue Schranke $G = 0$ $(123, 3)$ $(13, 4)$ $(14, 5)$
ab hier: ***pq*** wird nur noch geleert...

Optimale Lösung für M'' ist $L = (12431)$ mit Gewicht 0, damit wieder:

Optimale Lösung für M ist $L = (12431)$ mit Gewicht $35 + 0 = 35$

Bestimmung guter Schrankenfunktionen:

- Gegeben: Mimimierungsproblem P
  - mit Bewertungsfunktion v für Lösungen,
  - benötigt: Gütefunktion g (‘Bound’) für Lösungsansätze
- bestimme g mit:
  Ist Lösung L Erweiterung des Ansatzes A, dann $v(L) \geq g(A)$

oft wird dazu **Relaxation** benutzt:

- Aufweichung der Aufgabenstellung von P zu P',
  z.B. Weglassen eigentlich notwendiger Bedingungen
- $\leadsto$ Lösungsraum von P' umfasst Lösungsraum von P
- evtl. effizienter Algorithmus für aufgeweichtes Problem P' bekannt
- Gütefunktion g' für P' ist auch Gütefunktion g für P...

Auch TSP-Beispiel als Relaxation interpretierbar:

- zyklenfreie Pfade als Lösung betrachten (statt Permutationen)
- dann: beste Lösung zu Ansatz $i_1 \ldots i_k$ ist gerade $i_1 \ldots i_k$ selbst...

15 ‘Harte’ Probleme

- Typische Problemklassen
- Exakte Verfahren
- Approximative Verfahren
  - Grundlegende Definitionen
  - Greedy-Algorithmen
  - lokale Suche
- Randomisierte Verfahren

Betrachte Optimierungsprobleme

Da vermutlich $\boldsymbol{P} \neq \boldsymbol{NP}$:

- Dynamisches Programmieren, Backtracking, Branch-and-Bound sind i.d.R. nicht polynomial!
- als Alternative: statt 'optimaler' nur 'gute' Lösungen verlangen!
- suche Lösung $\boldsymbol{y} \in \boldsymbol{S}(\boldsymbol{x})$, so dass $\boldsymbol{v}(\boldsymbol{y})$ nicht 'allzu weit' von optimalen $\boldsymbol{v}^*(\boldsymbol{x}) = \boldsymbol{opt}\{\boldsymbol{v}(\boldsymbol{z}) \mid \boldsymbol{z} \in \boldsymbol{S}(\boldsymbol{x})\}$ weg ist

Anmerkung: das Finden einer zulässigen Lösung $\boldsymbol{y} \in \boldsymbol{S}(\boldsymbol{x})$ sollte nicht bereits ***NP***-vollständig sein, Beispiele:

- Finden zulässiger Lösungen bei MaxSAT ist trivial.
- Finden zulässiger Lösungen bei TSP ist trivial, wenn ∞ in Lösungen zulässig ist
- Finden zulässiger Lösungen bei TSP ist NP-vollständig, wenn ∞ in Lösungen nicht zulässig ist (da damit die Existenz von Hamiltonkreisen getestet werden kann!)

Definition 15.4

Für ein Optimierungsproblem $\boldsymbol{P}$ *mit zulässiger Eingabe* $\boldsymbol{x}$ *und Lösungsmenge* $\boldsymbol{S}(\boldsymbol{x})$ *ist die Performanz* $\boldsymbol{r}(\boldsymbol{x}, \boldsymbol{y})$ *einer Lösung* $\boldsymbol{y} \in \boldsymbol{S}(\boldsymbol{x})$ *definiert durch*

$$\boldsymbol{r}(\boldsymbol{x}, \boldsymbol{y}) := \min \left\{ \frac{\boldsymbol{v}(\boldsymbol{y})}{\boldsymbol{v}^*(\boldsymbol{x})}, \frac{\boldsymbol{v}^*(\boldsymbol{x})}{\boldsymbol{v}(\boldsymbol{y})} \right\}$$

wobei $\boldsymbol{v}^*(\boldsymbol{x}) :=$ *Optimum aus* $\{\boldsymbol{v}(\boldsymbol{z}) \mid \boldsymbol{z} \in \boldsymbol{S}(\boldsymbol{x})\}$

Anmerkungen:

- Bei Minimierungsproblemen: $\boldsymbol{v}^*(\boldsymbol{x}) := \min\{\boldsymbol{v}(\boldsymbol{z}) \mid \boldsymbol{z} \in \boldsymbol{S}(\boldsymbol{x})\}$
- Bei Maximierungsproblemen: $\boldsymbol{v}^*(\boldsymbol{x}) := \max\{\boldsymbol{v}(\boldsymbol{z}) \mid \boldsymbol{z} \in \boldsymbol{S}(\boldsymbol{x})\}$
- Stets $\boldsymbol{r}(\boldsymbol{x}, \boldsymbol{y}) \leq \mathbf{1}$, gute Approximationen haben $\boldsymbol{r}(\boldsymbol{x}, \boldsymbol{y}) \approx \mathbf{1}$
- in Literatur auch Definition über Kehrwert $\mathbf{1}/\boldsymbol{r}$, dann Performanz stets $\geq \mathbf{1}$ ...

Definition 15.5 (APX)

*Ein Optimierungsproblem liegt in **APX**, wenn es **eine Zahl** $\delta \in (\mathbf{0}, \mathbf{1})$ und einen polynomialen Algorithmus gibt, der bei jeder zulässigen Eingabe $\boldsymbol{x}$ eine Lösung $\boldsymbol{y} \in \boldsymbol{S}(\boldsymbol{x})$ liefert mit einer zugehörigen Performanz $\boldsymbol{r}(\boldsymbol{x}, \boldsymbol{y}) \geq \mathbf{1} - \delta$.*
*Die Abkürzung **APX** deutet an, dass das Optimierungsproblem - bis zu einem gewissen Grad - approximierbar ist.*

δ darf beliebig nahe an **1** sein, d.h. schlechte Performanz erlaubt...

Stärkere Einschränkung / bessere Approximationen:

Definition 15.6 (PTAS)

*Ein Optimierungsproblem liegt in **PTAS**, wenn es **für jede Zahl** $\delta \in (\mathbf{0}, \mathbf{1})$ einen polynomialen Algorithmus gibt, der bei jeder zulässigen Eingabe $\boldsymbol{x}$ eine Lösung $\boldsymbol{y} \in \boldsymbol{S}(\boldsymbol{x})$ liefert mit einer zugehörigen Performanz $\boldsymbol{r}(\boldsymbol{x}, \boldsymbol{y}) \geq \mathbf{1} - \delta$.*
*Die Abkürzung **PTAS** steht für polynomial time approximation scheme.*

Hier ist jede beliebig gute Performanzschranke δ erreichbar.

Definition 15.7 (FPTAS)

*Ein Optimierungsproblem liegt in **FPTAS**, wenn es einen Algorithmus gibt, der bei jedem zulässigen x und $k \in \mathbb{N}$ als Eingabe eine Lösung $y \in S(x)$ liefert mit einer zugehörigen Performanz $r(x, y) \geq 1 - 1/k$. Die Komplexität des Algorithmus muss polynomial in x und k sein. Die Abkürzung **FPTAS** steht für fully polynomial-time approximation scheme.*

Bei FPTAS ist selbst die Abhängigkeit von der Performanzschranke noch polynomiell!

Damit ergibt sich folgende Inklusionskette

Menge exakt lösbarer Optimierungsprobleme aus *P*
$\subseteq$ FPTAS $\subseteq$ PTAS $\subseteq$ APX $\subseteq$
Menge aller *NP*-Optimierungsprobleme

Falls $\boldsymbol{P \neq NP}$, sind alle Inklusionen echt:

$$\textbf{Menge exakt lösbarer Optimierungsprobleme aus } \boldsymbol{P}$$
$$\subsetneq \quad \textbf{FPTAS} \quad \subsetneq \quad \textbf{PTAS} \quad \subsetneq \quad \textbf{APX} \quad \subsetneq$$
$$\textbf{Menge aller } \boldsymbol{NP}\textbf{-Optimierungsprobleme}$$

Typische Beispiele einiger Klassen sind:

- **Menge exakt lösbarer Optimierungsprobleme aus *P***:
  Maximale Flüsse in Graphen
  ($\rightsquigarrow$ Vorlesung Netzwerkalgorithmen)
- **FPTAS**: **0**/**1**-Rucksackproblem
  ($\rightsquigarrow$ über dynamisches Programmieren, s.u.)
- **APX**: MaxSAT
  - probabilistisches Approximationsverfahren (analog APX)
    ($\rightsquigarrow$ Vorl. Approximative Algorithmen)
  - $\boldsymbol{P \neq NP} \Rightarrow$ MaxSAT $\notin$ PTAS: über 'PCP-Theorem'
    (PCP: probabilistically checkable proofs)
    ($\rightsquigarrow$ Vorlesung Approximative Algorithmen)
- **Menge aller *NP*-Optimierungsprobleme**: TSP
  (vermutlich nicht in **APX**, s.u.)

Beispiel 15.8 (zur Approximierbarkeit von TSP)

Annahme: Es gibt polynomialen Algorithmus für TSP und $\delta \in (0,1)$ *mit Performanz von mindestens* $1-\delta$

- *Sei* G *beliebiger Graph mit* n *Knoten, definiere dazu TSP* x *durch*

$$m_{i,j} = \begin{cases} 1, & \text{falls } (i,j) \text{ Kante in } G \\ 1 + \lceil \frac{n}{1-\delta} \rceil, & \text{sonst} \end{cases}$$

- *TSP-Approximation liefert zu* x *Lösung* y *mit Performanz* $r(x,y) = \frac{v^*(x)}{v(y)} \geq 1-\delta$
- *Existiert Hamilton-Kreis in* G*, dann* $v^*(x) = n$*, also* $v(y) < \frac{n}{1-\delta}$*, d.h.* y *nutzt nur Kanten aus* G *(d.h. findet Hamilton-Kreis in* G*)*
- *Über* $v(y)$ *kann also entschieden werden, ob in* G *ein Hamilton-Kreis existiert ...*

also

$$TSP \in APX \Longrightarrow P = NP$$

d.h. TSP *liegt vermutlich nicht in* APX*!*

Beispiel 15.9 (**0**/**1**-Rucksack liegt in **FPTAS**)

Löse **0**/**1**-*Rucksack über 'minimales Gewicht für gesuchten Gewinn':*

- *Gegeben:* $n \in \mathbb{N}$, $G \in \mathbb{N}$, *Vektoren* $(v_1, ..., v_n), (g_1, ..., g_n) \in \mathbb{N}^n$
- *definiere* $g(i, w) :=$ *minimales Gewicht für Teilmengen von Objekten aus* $\{1, ..., i\}$, *die zum Gewinn* $\geq w$ *führen*
- *formal: berechne*

$$g(i, w) = \min\{\sum_{j \in I} g_j \mid I \subseteq \{1, ..., i\} \wedge \sum_{j \in I} v_j \geq w\}$$

- *Gesucht: größter Gewinn* w *mit* $g(n, w) \leq G$
- *rekursive Formulierung von* $g(i, w)$*:*

$$g(i, w) = \begin{cases} 0 & w \leq 0 \\ \infty & w > 0 \wedge i = 0 \\ \min \left\{ \begin{array}{l} g(i-1, w), \\ g(i-1, w-v_i) + g_i \end{array} \right\} & \text{sonst} \end{cases}$$

Fortsetzung von 15.9: 0/1-Rucksack in FPTAS

Dynamisches Programm dazu (für exakte Lösung):

- statt Funktion $g(i, w)$ verwende Tabelle $g[i, w]$ mit $0 \leq i < n$ aber nur für w mit $g[i, w] \leq G$

```
FOR i:=0 TO n DO
    w:=0;
    REPEAT
      w = w + 1;
      berechne g[i,w] nach Formel
    UNTIL g[i,w]> G
ENDFOR
Ausgabe: w-1
```

- Ausgabe also: größtes W mit $g[n, W] \leq G$
- I sei zu W gehörige optimale Teilmenge von $\{1, ..., n\}$
- zur Tabellengröße: Zugriffe nur auf $g[i, w]$ mit $w \leq W + 1$
- Aufwand: $\mathcal{O}(n \cdot W)$ (Funktionsauswerungen ignorierbar!)

Fortsetzung von 15.9: 0/1-Rucksack in FPTAS

jetzt approximative Lösung:

- wähle m geeignet (s.u.), setze $v'_i := \lfloor v_i/m \rfloor$
- löse $(0,1)$-Rucksack für $(v'_1, ..., v'_n)$
  (aber mit unveränderten $(g_1, ..., g_n)$ und G)
- J sei die dabei gefundene optimale Menge, mit $V' := \sum_{i \in J} v'_i$
- setze $V := \sum_{i \in J} v_i$ (d.h. neue optimale Menge, aber alte Kosten!)

Vergleiche nun J mit I sowie die drei Werte W, V' und V:

- Zwischen J und I lässt sich kein direkter Zusammenhang finden, Objekte der Lösungen I und J können komplett verschieden sein!
- J erfüllt $\sum_{i \in J} g_i \leq G$, d.h. J ist auch (i.d.R. nicht-optimale) Lösung für das Ursprungsproblem, d.h $V \leq W$, zudem

$$V' = \sum_{i \in J} v'_i \leq \sum_{i \in J} v_i/m = V/m \leq W/m$$

- damit sofort: Aufwand für die Lösung des modifizierten Problems mit $(v'_1, ..., v'_n)$ ist $\mathcal{O}(n \cdot V') = \mathcal{O}(n \cdot W/m)$

Fortsetzung von 15.9: 0/1-Rucksack in FPTAS

Betrachte folgendes Beispielproblem:

$$n = 3, (v_1, v_2, v_3) = (3, 5, 6), (g_1, g_2, g_3) = (1, 2, 3), G = 5$$

Resultate beim dynamischen Programm:

W :	1	2	3	4	5	6	7	8	9	10	11	12	...
$i = 1$	1	1	1	∞									
$i = 2$	1	1	1	2	2	3	3	3	∞				
$i = 3$	1	1	1	2	2	3	3	3	4	5	5	6	

Lösung: $W = 11$ bei der Menge $I = \{2, 3\}$
Teste Approximation mit $m = 3$, also $(v'_1, v'_2, v'_3) = (1, 1, 2)$

V' :	1	2	3	4	...
$i = 1$	1	∞			
$i = 2$	1	3	∞		
$i = 3$	1	3	4	6	

Lösung: $V' = 3$ bei $J = \{1, 3\}$, d.h. Approximation an W ist $V = 9$.

Fortsetzung von 15.9: 0/1-Rucksack in FPTAS

zur Performanz V/W der Lösung J (für das Originalproblem):

- Da J optimal für $(v'_1, \ldots, v'_n)$ gilt (1) $\sum_{i \in I} v'_i \leq V'$.
- Setze $v_{max} := \max\{v_1, \ldots, v_n\}$, damit sofort (2) $W \leq n \cdot v_{max}$
- o.B.d.A. stets $g_i \leq G$ (sonst Objekt i uninteressant...), damit andererseits (3) $v_{max} \leq W$
- Mit Setzung $v'_i := \lfloor v_i/m \rfloor$ ist $v'_i \geq v_i/m - 1$, also

$$\begin{aligned} V &= \sum_{i \in J} v_i \geq \sum_{i \in J} v'_i \cdot m = V' \cdot m \overset{(1)}{\geq} \sum_{i \in I} v'_i \cdot m \\ &\geq \sum_{i \in I} (v_i/m - 1) \cdot m \geq \sum_{i \in I} (v_i - m) \geq W - n \cdot m \end{aligned}$$

- für die Performanz ergibt sich:

$$\frac{V}{W} \geq \frac{W - nm}{W} \geq 1 - \frac{n \cdot m}{W} \overset{(3)}{\geq} 1 - \frac{n \cdot m}{v_{max}}$$

Fortsetzung von 15.9: 0/1-Rucksack in FPTAS

- Für Ziel-Performanz $\mathbf{1} - \mathbf{1}/\boldsymbol{k}$ wähle $\boldsymbol{m}$ so, dass $\frac{\boldsymbol{n} \cdot \boldsymbol{m}}{\boldsymbol{v_{max}}} \leq \mathbf{1}/\boldsymbol{k}$, z.B.

$$\boldsymbol{m} = \frac{\boldsymbol{v_{max}}}{\boldsymbol{k} \cdot \boldsymbol{n}}$$

  mit Aufwand

$$\mathcal{O}(\boldsymbol{n} \cdot \boldsymbol{W}/\boldsymbol{m}) = \mathcal{O}(\boldsymbol{k} \cdot \boldsymbol{n}^2 \cdot \boldsymbol{W}/\boldsymbol{v_{max}}) \overset{(2)}{=} \mathcal{O}(\boldsymbol{k} \cdot \boldsymbol{n}^3)$$

- Die Komplexität ist also polynomial in $\boldsymbol{n}$ und $\boldsymbol{k}$.

Anmerkungen:

- Ein Problem heißt *stark NP-vollständig*, wenn es auch noch NP-vollständig ist, wenn alle involvierten Zahlen in unärer Schreibweise (statt binär) eingegeben werden.
- Bei unärer Notation wird die Eingabe wesentlich länger, ein polynomialer Algorithmus darf länger laufen...
- Gibt man bei $\mathbf{0}/\mathbf{1}$-Rucksack die Schranke $\boldsymbol{G}$ unär an, so gibt es eine polynomiale Lösung.
  (Bsp: bekanntes dynamisches Programm mit Laufzeit $\mathcal{O}(\boldsymbol{n} \cdot \boldsymbol{G})$)
  $\mathbf{0}/\mathbf{1}$-Rucksack ist damit nur 'schwach NP-vollständig'!
- Stark NP-vollständig sind z.B. TSP, 3-KNF-SAT, Färbbarkeit,...
- Es gilt sogar:
  - **Falls P$\neq$ NP, so liegt kein stark NP-vollständiges Problem in FPTAS!**

Beispiel 15.10 (TSP-Varianten)

Wichtige Modifikationen des TSP sind:

- *Metrisches TSP: Kosten $m_{i,j}$ der Kanten sind symmetrisch und erfüllen die Dreiecksungleichung $m_{i,j} \leq m_{i,k} + m_{k,j}$*
- *Euklidisches TSP: Punkte liegen auf der Ebene, Entfernungen der Punkte entsprechen dem Euklidischen Abstand*

Hier gilt:

- *Metrisches TSP liegt in **APX**, wobei die Performanz **2**/**3** beträgt (d.h. die gefundene Route ist 50% maximal länger als die optimale Route)(Christofides, 1976)*
- *Metrisches TSP liegt nicht in **PTAS**(wenn P≠NP) (Arora, 1992)*
- *Euklidisches TSP ist NP-vollständig*
- *Euklidisches TSP liegt in **PTAS**(Arora, 1996)*

Im folgenden: Techniken, die bei Approximationsversuchen angewendet werden (i.d.R. ohne Garantie für Performanz oder polynomiale Zeitkomplexität).

15 ‘Harte’ Probleme

- Typische Problemklassen
- Exakte Verfahren
- Approximative Verfahren
  - Grundlegende Definitionen
  - Greedy-Algorithmen
  - lokale Suche
- Randomisierte Verfahren

Greedy-Verfahren:

- Ziel: exakte/approximative Lösung von Optimierungsproblemen
- wieder: Lösung durch Erweiterung von Lösungsansätzen...
- dynamisches Programm: jeweils beste Erweiterung ($\rightsquigarrow$ Tabelle...)
- Greedy: gute, leicht findbare Erweiterung (ohne Tabelle...)
- Grundidee: *Nimm immer das beste Stück!*

Beispiel: $\mathbf{0}/\mathbf{1}$-Rucksackproblem,

$$\boldsymbol{n} = \mathbf{3}, (\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3) = (\mathbf{3}, \mathbf{5}, \mathbf{6}), (\boldsymbol{g}_1, \boldsymbol{g}_2, \boldsymbol{g}_3) = (\mathbf{1}, \mathbf{2}, \mathbf{3}), \boldsymbol{G} = \mathbf{5}$$

Greedy-Heuristik hier:
Nimm das Stück mit den höchsten Wert pro Gewichtsanteil!

Reihenfolge für Beispiel:

$$\frac{\mathbf{3}}{\mathbf{1}} \geq \frac{\mathbf{5}}{\mathbf{2}} \geq \frac{\mathbf{6}}{\mathbf{3}}$$

also: erst Objekt 1 nehmen, dann Objekt 2, dann ... $\boldsymbol{G}$ überschritten.

Resultat: Lösung $\{\mathbf{1}, \mathbf{2}\}$ mit Wert $\mathbf{3} + \mathbf{5}$ (nicht optimal...)

Beispiel 15.11 (Bruchteil-Rucksackproblem)

- *Gegeben:* $n \in \mathbb{N}$, $G \in \mathbb{N}$, *Vektoren* $(v_1, ..., v_n), (g_1, ..., g_n) \in \mathbb{N}^n$
- *Gesucht: Wie groß kann der 'Wert' von Objekten werden, deren Gewicht* $\sum_{i \in I} g_i$ *nicht größer als die Schranke* G*?*
- *Aber: Jetzt dürfen Objekte zerteilt werden!*
- *als Lösung daher jetzt ein Vektor* $(a_1, ..., a_n)$ *mit* $a_i \in \mathbb{Q}$ *und* $0 \leq a_i \leq 1$ *(statt einer Menge* I*)*
- *formal: bestimme*

$$\max\{\sum_{i=1}^{n} a_i v_i \mid a_i \in [0,1] \text{ mit } \sum_{i=1}^{n} a_i g_i \leq G\}$$

Greedy-Ansatz:

- *Sortiere alle Objekte nach Wert* v_i / g_i*, d.h.* $\frac{v_1}{g_1} \geq \frac{v_2}{g_2} \geq ... \geq \frac{v_n}{g_n}$
- *dann nimm soviel von den hochwertigen Objekten, wie möglich...*

Fortsetzung von 15.11: Bruchteil-Rucksackproblem

Algorithmische Lösung:

$k := 0$

WHILE $\sum_{i=1}^{k+1} g_i \leq G$ DO $k := k + 1$ ENDWHILE

$b := (G - \sum_{i=1}^{k} g_i)/g_{k+1}$

Optimale Lösung ist $(a_1, a_2, \ldots, a_n) = (\underbrace{1, 1, \ldots 1}_{k}, b, \underbrace{0, 0, \ldots, 0}_{n-k-1})$

Grundidee zu Nachweis der Optimalität der Lösung:

- Sei $(a'_1, a'_2, \ldots, a'_n)$ eine andere Lösung.
- Betrachte $j := \min\{i \mid a'_i \neq a_i\}$
- Dann $\sum_{i=1}^{j} g_i < G$, also gibt es $m > j$ mit $a'_m > 0$.
- Transferiere dann Gewicht von Objekt m zu Objekt j, (d.h. verkleinere a'_m und vergrößere a'_j soweit wie möglich)
- die entstehende Lösung wäre nicht schlechter..., aber 'näher' an $(a_1, a_2, \ldots, a_n)$

Allgemeine Form der Probleme, bei denen Greedy anwendbar ist:

Definition 15.12

E sei endliche Menge, $\mathcal{U}$ sei eine Menge von Teilmengen von E. Die Struktur $(E, \mathcal{U})$ heißt Teilmengensystem, wenn

- $\emptyset \in \mathcal{U}$
- $A \subseteq B \wedge B \in \mathcal{U} \Rightarrow A \in \mathcal{U}$

$w : E \to \mathbb{Q}$ sei eine Kostenfunktion,
gesucht wird eine in $\mathcal{U}$ maximale Menge T (bzgl. $\subseteq$) mit maximalem Gesamtkosten

$$w(T) = \sum_{e \in T} w(e)$$

- Die Rolle der Gewichtsschranke G übernimmt das Teilmengensystem $\mathcal{U}$
- Dabei ist $T \in \mathcal{U}$ maximal in $\mathcal{U}$, wenn kein $T' \in \mathcal{U}$ mit $T \subset T'$ existiert.

kanonischer Greedy-Algorithmus für Teilmengensysteme:

```
Ordne die Elemente in E = {e_1, ..., e_n},
     so dass w(e_1) ≥ w(e_2) ≥ ... ≥ w(e_n)
Setze T := ∅
FOR k := 1 TO n DO
    IF T ∪ {e_k} ∈ U THEN T := T ∪ {e_k}
ENDFOR
Ausgabe von T als Lösung
```

Charakterisierung der Probleme, die Greedy-Algorithmus optimal löst:

Definition 15.13

Ein Teilmengensystem $(E, \mathcal{U})$ *heißt Matroid, wenn zusätzlich die Austauscheigenschaft gilt:*

- $A, B \in \mathcal{U} \wedge |A| < |B| \quad \Rightarrow \quad (\exists x \in B \setminus A)\ A \cup \{x\} \in \mathcal{U}$

- 'Matroid' verallgemeinert Begriff '(lineare) Unabhängigkeit'
- In Matroiden haben maximale Mengen gleiche Mächtigkeit!

Beispiel für Matroide:

- seien $n, k \in \mathbb{N}$ gegeben, $k \leq n$
- betrachte $E = \{1, 2, ..., n\}$
- setze $\mathcal{U} = \{A \subseteq E \quad : \quad |A| \leq k\}$

Maximale Menge sind hier die k-elementigen Mengen,
die Austauscheigenschaft ist offensichtlich erfüllt.

Weitere Beispiele für Matroide $(E, \mathcal{U})$:

- E: endliche Menge von Vektoren,
  $\mathcal{U}$: linear unabhängige Teilmengen von E
  (z.B.: E besteht aus Spalten einer Matrix, daher auch 'Matroid')
- E: Kantenmenge eines endlichen Graphen G,
  $\mathcal{U}$: kreisfreie Teilmengen von E
  (auch 'graphisches Matroid' genannt)
  Maximale Elemente von $\mathcal{U}$ sind aufspannende Wälder von G.

Bedeutung der Austauscheigenschaft:

- $\boldsymbol{A}$ und $\boldsymbol{B}$ seien maximale Elemente im Matroid $(\boldsymbol{E}, \mathcal{U})$
- Damit $|\boldsymbol{A}| = |\boldsymbol{B}|$
- Falls $\boldsymbol{A} \neq \boldsymbol{B}$:
  - Wähle $\boldsymbol{a} \in \boldsymbol{A} \setminus \boldsymbol{B}$.
  - Zu $\boldsymbol{A} \setminus \{\boldsymbol{a}\}$ gibt es $\boldsymbol{b} \in \boldsymbol{B} \setminus \boldsymbol{A}$ mit $\boldsymbol{A'} := \boldsymbol{A} \setminus \{\boldsymbol{a}\} \cup \{\boldsymbol{b}\} \in \mathcal{U}$
  - $\boldsymbol{A'} \setminus \boldsymbol{B}$ ist kleiner als $\boldsymbol{A} \setminus \boldsymbol{B}$, $\boldsymbol{A'} \cap \boldsymbol{B}$ ist größer als $\boldsymbol{A} \cap \boldsymbol{B}$
- Also: Man kann $\boldsymbol{A}$ durch Austausch einzelner Elemente schrittweise in $\boldsymbol{B}$ umwandeln, ohne dabei $\mathcal{U}$ zu verlassen!

Satz 15.14

Sei $(\boldsymbol{E}, \mathcal{U})$ ein Teilmengensystem.
*Der kanonische Greedy-Algorithmus liefert beim Optimierungsproblem genau dann **für jede beliebige** Kostenfunktionen $\boldsymbol{w} : \boldsymbol{E} \to \mathbb{Q}$ die optimale Lösung, wenn $(\boldsymbol{E}, \mathcal{U})$ ein Matroid ist.*

Beweis von "$\Leftarrow$":

- gegeben: $(\boldsymbol{E}, \mathcal{U})$ Matroid, $\boldsymbol{w} : \boldsymbol{E} \to \mathbb{Q}$ Gewichtsfunktion
- O.B.d.A.: bereits Ordnung $\boldsymbol{w}(\boldsymbol{e_1}) \geq \boldsymbol{w}(\boldsymbol{e_2}) \ldots \geq \boldsymbol{w}(\boldsymbol{e_n})$ vorhanden
- $\boldsymbol{T} = \{\boldsymbol{e_{i_1}}, \ldots, \boldsymbol{e_{i_k}}\}$ sei Lösung durch Greedy-Algorithmus.
- Annahme: Greedy-Algorithmus liefert nicht die optimale Lösung.
- $\boldsymbol{T'} = \{\boldsymbol{e_{j_1}}, \ldots, \boldsymbol{e_{j_k}}\}$ sei bessere Lösung, d.h. $\boldsymbol{w}(\boldsymbol{T'}) > \boldsymbol{w}(\boldsymbol{T})$
- O.B.d.A.: $\boldsymbol{i_1} < \boldsymbol{i_2} < \ldots < \boldsymbol{i_k}$ und $\boldsymbol{j_1} < \boldsymbol{j_2} < \ldots < \boldsymbol{j_k}$
- Also existiert minimales μ mit $\boldsymbol{w}(\boldsymbol{e_{j_\mu}}) > \boldsymbol{w}(\boldsymbol{e_{i_\mu}})$, insbesondere dabei $\boldsymbol{j_\mu} < \boldsymbol{i_\mu}$
- Wende Austauscheigenschaft auf $\boldsymbol{A} = \{\boldsymbol{e_{i_1}}, \ldots, \boldsymbol{e_{i_{\mu-1}}}\}$ und $\boldsymbol{B} = \{\boldsymbol{e_{j_1}}, \ldots, \boldsymbol{e_{j_\mu}}\}$ an:
  Es gibt $\boldsymbol{e_{j_\sigma}} \in \boldsymbol{B} \setminus \boldsymbol{A}$ mit $\boldsymbol{A} \cup \{\boldsymbol{e_{j_\sigma}}\} \in \mathcal{U}$
- Mit $\sigma \leq \mu$ jedoch $\boldsymbol{w}(\boldsymbol{e_{j_\sigma}}) \geq \boldsymbol{w}(\boldsymbol{e_{j_\mu}}) > \boldsymbol{w}(\boldsymbol{e_{i_\mu}})$, d.h.

  Greedy-Algorithmus hätte $\boldsymbol{e_{j_\sigma}}$ vor $\boldsymbol{e_{i_\mu}}$ in $\boldsymbol{T}$ aufnehmen müssen. ↯

Indirekter Beweis von "$\Rightarrow$":

- Annahme: Austauscheigenschaft gilt nicht,
  aber Greedy liefert für jedes $\boldsymbol{w}$ optimale Lösung.
- Also gibt es $\boldsymbol{A}, \boldsymbol{B} \in \mathcal{U}$ mit $(\forall \boldsymbol{b} \in \boldsymbol{B} \backslash \boldsymbol{A}) \boldsymbol{A} \cup \{\boldsymbol{b}\} \notin \mathcal{U}$.
- Setze $\boldsymbol{r} := |\boldsymbol{B}|$ und betrachte folgende Kostenfunktion $\boldsymbol{w}$:

$$\boldsymbol{w}(\boldsymbol{e}) := \begin{cases} \boldsymbol{r}+\mathbf{1}, & \boldsymbol{e} \in \boldsymbol{A} \\ \boldsymbol{r}, & \boldsymbol{e} \in \boldsymbol{B} \backslash \boldsymbol{A} \\ \mathbf{0}, & \text{sonst} \end{cases}$$

- Greedy-Algorithmus: Lösung $\boldsymbol{T}$ mit $\boldsymbol{A} \subseteq \boldsymbol{T}$ und $\boldsymbol{T} \cap (\boldsymbol{B} \setminus \boldsymbol{A}) = \emptyset$.
- Wegen $\boldsymbol{B} \in \mathcal{U}$ gibt es auch eine Lösung $\boldsymbol{T'}$ mit $\boldsymbol{B} \subseteq \boldsymbol{T'}$
- Dann jedoch

$$\begin{aligned} \boldsymbol{w}(\boldsymbol{T}) &= (\boldsymbol{r}+\mathbf{1}) \cdot |\boldsymbol{A}| \leq (\boldsymbol{r}+\mathbf{1}) \cdot (\boldsymbol{r}-\mathbf{1}) = \boldsymbol{r}^{\mathbf{2}} - \mathbf{1} \\ \boldsymbol{w}(\boldsymbol{T'}) &\geq \boldsymbol{r} \cdot |\boldsymbol{B}| = \boldsymbol{r}^{\mathbf{2}} \end{aligned}$$

Also $\boldsymbol{w}(\boldsymbol{T}) < \boldsymbol{w}(\boldsymbol{T'})$, d.h. Greedy versagt bei diesem $\boldsymbol{w}$. $\lightning$

Also:

- Teilmengensysteme erlauben die Anwendung von Greedy-Algorithmen
- Matroide führen zu optimaler Lösung
- Oft jedoch: Austauscheigenschaft nicht vollständig erfüllt (z.B.: $\exists$ mehrere maximale Mengen mit untschiedlicher Mächtigkeit)
  also Greedy-Lösung nicht optimal, sondern nur approximativ
- mögliches anderes Problem:
  nicht-additive Kostenfunktion, $\boldsymbol{w}(\boldsymbol{T}) \neq \sum_{\boldsymbol{e} \in \boldsymbol{T}} \boldsymbol{w}(\boldsymbol{e})$
- aber: Greedy liefert oft gute Heuristik!

Im Folgenden: Beispiele für ‘gute’ Greedy-Heuristiken bei

- Färbbarkeit
- TSP

Beispiel 15.15 (Färbbarkeit)

- *Gegeben Graph* $G = (V, E)$
- *Ziel: Färbe Knoten mit möglichst wenigen Farben, d.h. suche 'kleines'* $k \in \mathbb{N}$ *und Funktion* $f : V \to \{0, \ldots, k-1\}$, *so dass für Kanten* $(u, v) \in E$ *stets* $f(u) \neq f(v)$.
- *Greedy-Heuristik:*
  - *Sortiere* V *nach Grad* $\delta(v)$ *der Knoten* $v \in V$, *also o.B.d.A.:* $V = \{v_1, \ldots, v_n\}$ *mit* $\delta(v_i) \geq \delta(v_{i+1})$.
  - *Wiederhole für* $\mu = 1, 2, \ldots, n$*: Setze*

$$f(\mu) := \min(\mathbb{N} \setminus \{f(i) \mid i < \mu, (v_i, v_\mu) \in E\})$$

  - *Jeder Knoten erhält also, in der Reihenfolge absteigender Grade, die jeweils kleinstmögliche Farbe.*
- *Jeder Graph mit maximalem Grad* Δ *wird dabei mit höchstens* $\Delta + 1$ *Farben gefärbt!*

Aus Übung: 2-färbbare (d.h. bipartite) Graphen sind in Polynomzeit färbbar! Damit:

Lemma 15.16

Jeder 3-färbbare Graph mit n Knoten und Grad Δ kann in Polynomzeit mit $\mathcal{O}(\min(\Delta, \sqrt{n}))$ Farben gefärbt werden.

Beweis dazu:

- Ist G 3-färbbar, dann ist bei jedem Knoten v die Menge $N(v)$ seiner Nachbarn 2-färbbar!
- Für jeden Knoten $v \in V$ mit $\delta(v) \geq \sqrt{n}$ ist $\overline{N}(v) := N(v) \cup \{v\}$ also schnell mit 3 (neuen) Farben färbbar.
  Danach entferne $\overline{N}(v)$ aus dem Graphen.
- Da $|N(v)| \geq \sqrt{n}$: Nach maximal $\sqrt{n}$ Iterationen kein Knoten mehr in G mit $\delta(v) \geq \sqrt{n}$.
- Rest des Graphen dann mit $\leq \sqrt{n}$ weiteren Farben färbbar!

Bestes bekanntes Resultat (Baumann, 2004): Färbung 3-färbbarer Graphen in Polynomzeit mit $\mathcal{O}(n^{3/14})$ Farben...

Beispiel 15.17 (Greedy-Algorithmen für TSP)

- *Grundidee: Konstruiere sukzessive immer längere Strecken, bis schließlich eine Rundreise entsteht...*
- *Heuristik 'Nearest Neighbor':*
  - *Gehe vom aktuellen Tour-Ende immer zur nächstliegenden neuen Stadt*
- *Heuristik 'Tourerweiterungen':*
  - *Starte mit kurzer Tour aus zwei Städten.*
  - *Erweitere Tour iterativ um je einen Knoten; Heuristiken dabei:*
    — 'Random Insertion': Wähle neuen Knoten zufällig
    — 'Farthest Insertion': Wähle Knoten, dessen Minimalabstand zur aktuellen Route möglichst groß ist
  - *Gewählter Knoten wird an optimaler Position in die Tour eingefügt.*
- *'Nearest Neighbor' ist nur mäßig erfolgreich:*
  Anfangs viele kurze Strecken, aber am Ende kostspieliges Einsammeln übersehener, weit auseinanderliegender Städte (aber mit Nachverbesserungen evtl. brauchbar, s. lokale Suche)
- *'Farthest Insertion' scheint die bessere Heuristik zu sein:*
  Grobstruktur der Route wird relativ schnell festgelegt!

15 ‘Harte’ Probleme

- Typische Problemklassen
- Exakte Verfahren
- Approximative Verfahren
  - Grundlegende Definitionen
  - Greedy-Algorithmen
  - lokale Suche
- Randomisierte Verfahren

‘Lokale Suche’ zur Verbesserung von gefundenen Lösungen:

- Greedy-Algorithmus liefert Startwert für Maximumsuche...
- Ziel: analog zur Austauscheigenschaft (tausche beliebig $\boldsymbol{a} \in \boldsymbol{A}$ gegen $\boldsymbol{b} \in \boldsymbol{B} \setminus \boldsymbol{A}$) teste lokale Änderungen
- z.B.: nach Konstruktion einer Lösung entferne Teile der Lösung und suche eine ‘Zeitlang’ nach Verbesserungen

Grundstruktur:

```
Erzeuge eine Anfangslösung L.
REPEAT
     Modifiziere L zufällig zu L'
     IF L' ist besser als L THEN L = L' ENDIF
UNTIL längere Zeit keine Verbesserung gefunden
AUSGABE von L
```

Prinzipieller Nachteil:

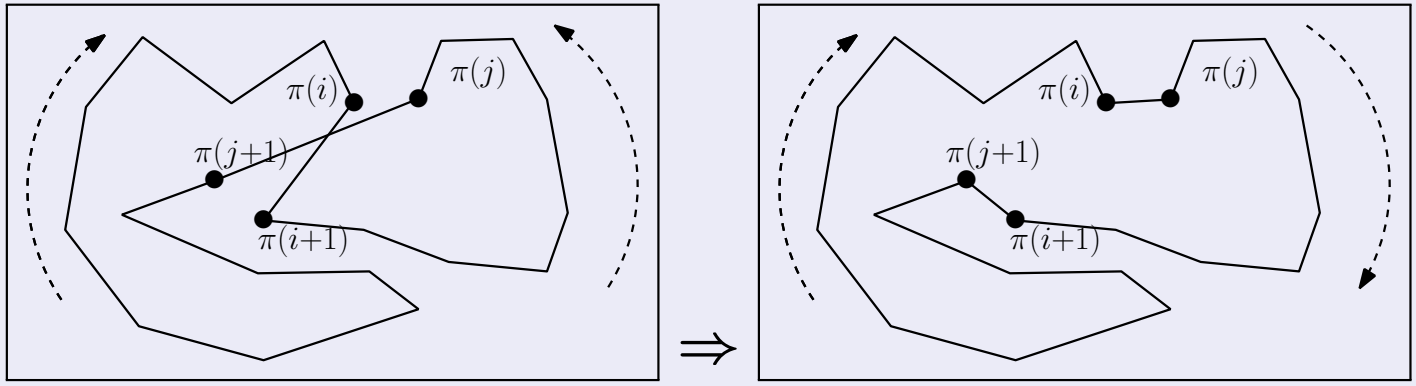
- die gefundene Lösung ist ein ‘lokales Optimum’...
- das globale Optimum kann beliebig weit entfernt sein!

Beispiel 15.18 (2-Opt-Heuristik bei TSP)

- *Betrachte TSP mit symmetrischen Entfernungen* $m_{i,j} = m_{j,i}$
- *Lösungen mit unendlichen Kosten seien erlaubt!*
- *Beginne mit beliebiger Permutation* π *der Knoten (zulässig!)*
- *Wähle zufällig zwei Knoten* $\pi(i)$, $\pi(j)$ *(mit* $i < j$*). Falls*

$$m_{\pi(i),\pi(i+1)} + m_{\pi(j),\pi(j+1)} > m_{\pi(i),\pi(j)} + m_{\pi(i+1),\pi(j+1)}$$

gilt, dann gibt es eine kürzere Route / bessere Permutation:



$$\rightsquigarrow \pi(1)\pi(2)\ldots\pi(i)\pi(j)\pi(j-1)\ldots\pi(i+2)\pi(i+1)\pi(j+1)\ldots\pi(n)$$

- *Animation: www-e.uni-magdeburg.de/mertens/TSP/TSP.html*
- *Laufzeit und Güte der Heuristik sind unbekannt!*

Weitere Heuristiken:

- Grundidee des Simulated Annealing (simulierte Abkühlung):
  - erlaube bei der lokalen Suche auch Zwischenlösungen, die um $\boldsymbol{c}(\boldsymbol{t})$ schlechter sind als die aktuelle Lösung
  - aber: $\boldsymbol{c}(\boldsymbol{t})$ geht mit wachsender Rechenzeit gegen $\mathbf{0}$, etwa

$$\boldsymbol{c}(\boldsymbol{t}) = \mathcal{O}(\mathbf{1}/\log \boldsymbol{t})$$

- Grundidee der genetischen Algorithmen:
  - Speichere mehrere verschiedenartige Lösungen
  - Kombiniere jeweils meherere Lösungen und verwende die besseren Resultate

15 ‘Harte’ Probleme

- Typische Problemklassen
- Exakte Verfahren
- Approximative Verfahren
- Randomisierte Verfahren
  - Probabilistische Turingmaschinen
  - Probabilistische Komplexitätsklassen

Bisher:
Nichtdeterminismus als Möglichkeit, eine Eingabe zu akzeptieren.

Naheliegend auch:

- Wahrscheinlichkeit der Akzeptanz einer Eingabe bzw.
- Wahrscheinlichkeit der Erzeugung einer Ausgabe

Annahme hier:

- jeder Befehl zufällig ausgewählt, mit gleicher Wahrscheinlichkeit
- die Einzel-Auswahlen seien unabhängig voneinander.

Definition 15.19

- *Eine probabilistische Turingmaschine* M *ist zunächst genauso definiert wie eine nichtdeterministische Turingmaschine.*
- *Die Wahrscheinlichkeit*

$$\mathbf{Prob}(K \xrightarrow[M]{} K')$$

mit der bei M *eine Konfiguration* K *in eine Konfiguration* K' *übergeht, wird definiert als:*

$$\mathbf{Prob}(K \xrightarrow[M]{} K') := \begin{cases} j/i, & \text{falls } K \xrightarrow[M]{} K' \text{ mit} \\ & i := \text{Anzahl der bei } K \text{ ausführbaren Befehle} \\ & j := \text{Anzahl dieser Befehle, bei denen aus } K\ K' \text{ wird} \\ 0, & \text{sonst} \end{cases}$$

Fortsetzung von 15.19:

- Die Wahrscheinlichkeit

$$\mathbf{Prob}(K \xrightarrow[M]{t} K')$$

mit der bei M nach t Schritten aus der Konfiguration K die Konfiguration K' wird, ergibt sich induktiv als:

$$\mathbf{Prob}(K \xrightarrow[M]{0} K') := \begin{cases} 1, & K = K' \\ 0, & K \neq K' \end{cases}$$

$$\mathbf{Prob}(K \xrightarrow[M]{t+1} K') := \sum_{K'' \text{ Konfiguration}} \mathbf{Prob}(K \xrightarrow[M]{t} K'') \cdot \mathbf{Prob}(K'' \xrightarrow[M]{} K')$$

Fortsetzung von 15.19:

- Per Definition: Nach Erreichen einer Endkonfiguration bleibt M stehen und rechnet nicht mehr weiter.
- Also ist für Endkonfigurationen K' sinnvoll definiert:

$$Prob(K \xrightarrow[M]{*} K')$$

mit der bei M nach beliebiger Zahl von Schritten aus K diese Endkonfiguration K' wird:

$$Prob(K \xrightarrow[M]{*} K') := \sum_{t=0}^{\infty} Prob(K \xrightarrow[M]{t} K')$$

Fortsetzung von 15.19:

- Setze $\boldsymbol{Acc_M(y)} :=$ Menge aller Endkonf., die zur Ausgabe $\boldsymbol{y}$ führen
- $\boldsymbol{I_M(x)} :=$ Startkonfiguration von $\boldsymbol{M}$ bei Eingabe von $\boldsymbol{x}$
- Wahrscheinlichkeit, mit der aus Eingabe $\boldsymbol{x}$ die Ausgabe $\boldsymbol{y}$ wird:

$$\boldsymbol{Prob}(\boldsymbol{x} \overset{}{\underset{M}{\mapsto}} \boldsymbol{y}) := \sum_{\boldsymbol{K' \in Acc_M(y)}} \boldsymbol{Prob}(\boldsymbol{I_M(x)} \overset{*}{\underset{M}{\rightarrow}} \boldsymbol{K'})$$

- Wahrscheinlichkeit, mit der bei $\boldsymbol{x}$ kein Ergebnis geliefert wird:

$$\boldsymbol{div_M(x)} := 1 - \sum_{\boldsymbol{y}} \boldsymbol{Prob}(\boldsymbol{x} \overset{}{\underset{M}{\mapsto}} \boldsymbol{y})$$

Da M mit Erreichen einer Endkonfiguration endet:

$$\sum \{ \mathbf{Prob}(I_M(x) \xrightarrow[M]{*} K') \mid K' \text{ Endkonfiguration} \} \leq 1$$

Damit für jedes x

$$\sum_y \mathbf{Prob}(x \overset{M}{\mapsto} y) \leq 1$$

Beide Werte können echt kleiner als 1 sein, da möglicherweise nicht bei jeder Berechnung eine Endkonfiguration erreicht wird!

Definition 15.20

- *Die von einer probabilistischen Turingmaschine* M *berechnete Funktion* $f_M^{(p)}$ *wird wie folgt definiert:*

$$f_M^{(p)}(x) := \begin{cases} y, & \text{falls } \mathbf{Prob}(x \overset{}{\underset{M}{\mapsto}} y) > 1/2 \\ \text{undefiniert}, & \text{falls } \mathbf{Prob}(x \underset{M}{\mapsto} y) \leq 1/2 \text{ für alle } y \end{cases}$$

- *Die von* M *akzeptierte Sprache* $L_M^{(p)}$ *wird definiert durch*

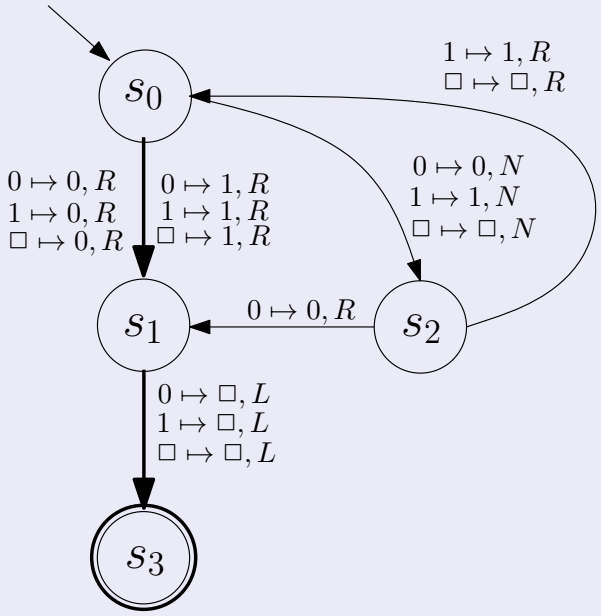
$$L_M^{(p)} := \{x \mid f_M^{(p)}(x) = 0\}$$

- *Die Fehlerwahrscheinlichkeit, mit der* $f_M^{(p)}$ *berechnet wird, ist*

$$\mathbf{err}_M(x) := \begin{cases} 1 - \mathbf{div}_M(x) \\ \quad -\mathbf{Prob}(x \underset{M}{\mapsto} f_M^{(p)}(x)), & \text{falls } f_M^{(p)}(x) \text{ definiert ist} \\ \text{undefiniert}, & \text{sonst} \end{cases}$$

Beispiel 15.21 (Probabilistische Turingmaschine)

Nichtdeterministische (=probabilistische) Maschine M*:*



Beispiele für Konfigurationsfolgen bei M*:*

(a) $s_0 0 \to 0 s_1 \square \to s_3 0 \square$

(b) $s_0 0 \to 1 s_1 \square \to s_3 1 \square$

(c) $s_0 0 \to s_2 0 \to 0 s_1 \square \to s_3 0 \square$

(d) $s_0 1 \to s_2 1 \to 1 s_0 \square \to 10 s_1 \square \to 1 s_3 0 \square$

(e) $s_0 1 \to s_2 1 \to 1 s_0 \square \to 11 s_1 \square \to 1 s_3 1 \square$

(f) $s_0 1 \to s_2 1 \to 1 s_0 \square \to 1 s_2 \square$
$\to 1 \square s_0 \square \to 1 \square s_2 \square \to 1 \square \square s_0 \square \ldots$

- *Hier gilt* $\mathbf{Prob}(x \overset{}{\underset{M}{\mapsto}} y) = 0$*, wenn* $y \notin \{0, 1\}$
- $\mathbf{Prob}(1 \underset{M}{\mapsto} 0) = \mathbf{Prob}(1 \underset{M}{\mapsto} 1) = \frac{1}{3} + \frac{1}{3^2} + \frac{1}{3^3} + \ldots = \sum_{i=1}^{\infty} (\frac{1}{3})^i = \frac{1}{2}$

  also $\mathbf{div}_M(1) = 0$*, zudem ist* $f_M^{(p)}(1)$ *nicht definiert.*
  (Achtung: M *könnte trotzdem unendlich oft zykeln!)*

Fortsetzung von 15.21: Probabilistische Turingmaschine

- analog für $v = \underbrace{1\ldots1}_{i}$ (bei $i \geq 0$):

$$Prob(v \overset{}{\underset{M}{\mapsto}} 0) = Prob(v \overset{}{\underset{M}{\mapsto}} 1) = \frac{1}{2}$$

  also $div_M(v) = 0$ und $f_M^{(p)}(v)$ ist nicht definiert.

- für $w := \underbrace{1\ldots1}_{i}\,0$ ergibt sich jedoch:

$$Prob(w \overset{}{\underset{M}{\mapsto}} 0) = \sum_{j=1}^{i+1} \frac{1}{3^j} + \frac{1}{3^{i+1}} = \frac{1}{2} \cdot (1 + \frac{1}{3^{i+1}})$$

$$Prob((w \overset{}{\underset{M}{\mapsto}} 1) = \sum_{j=1}^{i+1} \frac{1}{3^j} = \frac{1}{2} \cdot (1 - \frac{1}{3^{i+1}})$$

damit: $f_M^{(p)}(w) = 0$, $div_M(w) = 0$ und $err_M(w) = \frac{1}{2} \cdot (1 - \frac{1}{3^{i+1}})$

Fortsetzung von 15.21: Probabilistische Turingmaschine

Betrachte z.B. Spezialfall $w = 10$

- Zunächst:

$$\begin{aligned} \mathbf{Prob}((w \overset{M}{\mapsto} 1) &= \frac{1}{2} \cdot (1 - \frac{1}{3^2}) \approx 0.444\ldots =: \varepsilon \\ \mathbf{Prob}(w \overset{M}{\mapsto} 0) &= \frac{1}{2} \cdot (1 + \frac{1}{3^2}) \approx 0.555\ldots = 1 - \varepsilon \end{aligned}$$

- Jetzt: Lasse M n-fach laufen, zähle Anzahl positiver/negativer Resultate
  Dann gibt es $n+1$ Möglichkeiten:

Ausgabe **1**	Ausgabe **0**	Wahrscheinlichkeit
n-fach	**0**-fach	ε^n
$n-1$-fach	**1**-fach	$n \cdot \varepsilon^{n-1} \cdot (1-\varepsilon)$
...		...
1-fach	$n-1$-fach	$n \cdot \varepsilon^1 \cdot (1-\varepsilon)^{n-1}$
0-fach	n-fach	$(1-\varepsilon)^n$

Fortsetzung von 15.21: Probabilistische Turingmaschine

- Die Wahrscheinlichkeit $p(n)$, dass mehr als die Hälfte der Ausgaben **1** sind, fällt exponentiell mit n:

$$\begin{aligned} p(n) &= \sum_{i=0}^{\lfloor n/2 \rfloor} \binom{n}{i} (1-\varepsilon)^i \varepsilon^{n-i} \\ &\leq \sum_{i=0}^{\lfloor n/2 \rfloor} \binom{n}{i} (1-\varepsilon)^i \varepsilon^{n-i} \left(\frac{1-\varepsilon}{\varepsilon}\right)^{n/2-i} \text{ da } \varepsilon < 1/2,\ i \leq n/2 \\ &= (\varepsilon(1-\varepsilon))^{n/2} \cdot \sum_{i=0}^{\lfloor n/2 \rfloor} \binom{n}{i} \leq (\varepsilon(1-\varepsilon))^{n/2} \cdot 2^n = \delta^n \end{aligned}$$

  mit der Setzung von $\delta = 2\sqrt{\varepsilon(1-\varepsilon)}$ (wobei insbesondere $\delta < 1$).
- Um Fehler $p(n) \leq 10^{-k}$ zu erhalten, reicht also $n \geq k \cdot \frac{1}{-\log_{10} \delta}$
- Im Beispiel: $\varepsilon = 0.444\ldots$, $\delta = 0.9938\ldots$, $\frac{1}{-\log_{10} \delta} \approx 370$, d.h. nach $n = 3700$ Wiederholungen ist $p(n) \leq 10^{-10}$

Definition 15.22

Sei M probabilistische Turingmaschine, bei der $f_M^{(p)}(x)$ stets existiert.

1. *M heißt Random-Turingmaschine, wenn für $x \notin L_M^{(p)}$ gilt:*

$$Prob(x \overset{}{\underset{M}{\mapsto}} 0) = 0$$

   (M erkennt $L_M^{(p)}$ mit einseitiger Fehlerwahrscheinlichkeit 0.)

2. *M heißt Monte-Carlo-Turingmaschine, wenn ein $\varepsilon < 1/2$ existiert, so daß für alle x stets gilt:*

$$err_M(x) + div_M(x) < \varepsilon$$

   (M berechnet $f_M^{(p)}$ mit beschränkter Fehlerwahrscheinlichkeit.)

3. *M heißt Las-Vegas-Turingmaschine, wenn für alle x stets gilt:*

$$err_M(x) = 0$$

   (M berechnet $f_M^{(p)}$ mit Fehlerwahrscheinlichkeit 0.)

Interpretation der Definitionen:

1. Ergibt bei einer Random-Turingmaschine $\boldsymbol{M}$ eine Berechnung die Ausgabe 0, so gilt sicher $\boldsymbol{x} \in \boldsymbol{L}_{\boldsymbol{M}}^{(\boldsymbol{p})}$.
2. Bei einer Monte-Carlo-Turingmaschine $\boldsymbol{M}$ wird das richtige Ergebnis $\boldsymbol{f}_{\boldsymbol{M}}^{(\boldsymbol{p})}(\boldsymbol{x})$ mit signifikant höherer Wahrscheinlichkeit $(> \mathbf{1/2})$ ausgegeben als irgendein falsches $(< \varepsilon < \mathbf{1/2})$. Dabei ist ε unabhängig(!) von $\boldsymbol{x}$.
3. Eine Las-Vegas-Turingmaschine $\boldsymbol{M}$ 'lügt' nie, liefert aber mit Wahrscheinlichkeit $\boldsymbol{div}_{\boldsymbol{M}}(\boldsymbol{x}) < \mathbf{1/2}$ überhaupt kein Ergebnis. Jede Las-Vegas-Maschine ist per Definition auch eine Random-Maschine.

Zeitkomplexität über zwei Vorgehensweisen definierbar:

- über den Erwartungswert der Rechenzeit
- über die Zeit, die benötigt wird, um die Ausgabe mit einer gewissen Wahrscheinlichkeit festzulegen.

Definition 15.23

Sei M *eine probabilistische Turingmaschine.*
Die durchschnittliche (oder erwartete) Rechenzeit $T_M^{\emptyset}(x)$ *von* M *bei einer Eingabe* x *ist definiert als*

$$T_M^{\emptyset}(x) := \begin{cases} \sum_{t=0}^{\infty} t \cdot \sum_{K \text{ Endkonf.}} \mathbf{Prob}(I_M(x) \xrightarrow[M]{t} K), & \text{falls } \mathbf{div}_M(x) = 0 \\ \infty, & \text{sonst} \end{cases}$$

Für $\varepsilon \leq 1/2$ *sei* $T_M^{\varepsilon}(x)$ *die Zeit, nach der* $f_M^{(p)}(x)$ *mit Wahrscheinlichkeit* $1-\varepsilon$ *festliegt:*

$$T_M^{\varepsilon}(x) := \min\{ t \in \mathbb{N} \mid 1-\varepsilon < \sum_{K \in Acc_M(f_M^{(p)}(x)),\ t' \leq t} \mathbf{Prob}(I_M(x) \xrightarrow[M]{t'} K) \}$$

Beispiel 15.24 (probabilistische Zeitkomplexität)

Sei M wie im Beispiel 15.21

- *Betrachte zunächst $v = 1\ldots1$:*

$$\begin{aligned} T_M^{\emptyset}(v) &= \frac{2}{3} \cdot 2 + \frac{2}{3^2} \cdot 4 + \frac{2}{3^3} \cdot 6 + \ldots \\ &= 4 \cdot \sum_{t=1}^{\infty} \frac{t}{3^t} \stackrel{*}{=} 4 \cdot \left(\frac{1/3}{(1-1/3)^2}\right) = 3 \end{aligned}$$

Zu $(*)$: Für $p < 1$ gilt $\sum_{i=0}^{\infty} \frac{i}{p^i} = \frac{p}{(1-p)^2}$

(Beweis über $\sum_{i=0}^{\infty} \frac{1}{p^i} = \frac{1}{1-p}$ *und* $\frac{d}{dp}\frac{1}{1-p} = \frac{1}{(1-p)^2}$ *)*

- *Für beliebige w immer $T_M^{\emptyset}(w) \leq T_M^{\emptyset}(1\ldots1)$, wenn $|w| = |1\ldots1|$.*
- *Also: Bei M ist für alle w die erwartete Rechenzeit $T_M^{\emptyset}(w) \leq 3$.*

Fortsetzung von 15.24: probabilistische Zeitkomplexität

Andererseits:

- $T_M^{1/2}(1 \ldots 1) = \text{undef}$, da $f_M^{(p)}$ undefiniert ist
- $T_M^{1/2}(\underbrace{1 \ldots 1}_{i} 0) = 2i + 2$
- $T_M^{1/3}(\underbrace{1 \ldots 1}_{i} 0) = \text{undef}$, da $Prob_M(1 \ldots 10 \mapsto 0) \leq 2/3$

daher:

- erwartete Zeit ist im Allgemeinen ungeeignetes Maß...
- für viele Probleme gibt es probabilistische Maschinen mit konstanter erwarteter Rechenzeit
- Ausnahme: Las-Vegas-Maschinen, hier wird die erwartete Zeit verwendet

Beispiel 15.25 (Quicksort als Las-Vegas-Algorithmus)

Gegeben Feld $\boldsymbol{a}[\mathbf{1}], \ldots, \boldsymbol{a}[\boldsymbol{n}]$*, Ziel: rekursive Sortierfunktion* $\boldsymbol{qs}$*:*

- $\boldsymbol{qs}(\boldsymbol{i}, \boldsymbol{j})$ *sortiert Teilfeld* $\boldsymbol{a}[\boldsymbol{i}] \ldots \boldsymbol{a}[\boldsymbol{j}]$*, d.h. Aufruf mit* $\boldsymbol{qs}(\mathbf{1}, \boldsymbol{n})$
- $\boldsymbol{qs}(\boldsymbol{i}, \boldsymbol{j})$ *arbeitet wie folgt:*
  (1) *Wähle* $\boldsymbol{k}$ *mit* $\boldsymbol{i} \leq \boldsymbol{k} \leq \boldsymbol{j}$
  (2) *Modifiziere* $\boldsymbol{a}[\boldsymbol{i}] \ldots \boldsymbol{a}[\boldsymbol{j}]$ *so, dass* $\boldsymbol{a}[\boldsymbol{k}]$ *an eine Stelle* $\boldsymbol{m}$ *kommt mit*

$$(\forall \nu, \boldsymbol{i} \leq \nu \leq \boldsymbol{m}) \quad \boldsymbol{a}[\nu] \leq \boldsymbol{a}[\boldsymbol{m}]$$
$$(\forall \nu, \boldsymbol{m} \leq \nu \leq \boldsymbol{j}) \quad \boldsymbol{a}[\boldsymbol{m}] \leq \boldsymbol{a}[\nu]$$

  (3) *Falls* $\boldsymbol{i} < \boldsymbol{m}-\mathbf{1}$*: Rekursiver Aufruf von* $\boldsymbol{qs}(\boldsymbol{i}, \boldsymbol{m}-\mathbf{1})$
  (4) *Falls* $\boldsymbol{m}+\mathbf{1} < \boldsymbol{j}$*: Rekursiver Aufruf von* $\boldsymbol{qs}(\boldsymbol{m}+\mathbf{1}, \boldsymbol{j})$
- *Übliche, deterministische Setzungen bei (1):*
  (1a) *Wähle* $\boldsymbol{k} = \boldsymbol{i}$
  (1b) *Wähle* $\boldsymbol{k} = \lfloor \frac{\boldsymbol{i}+\boldsymbol{j}}{\mathbf{2}} \rfloor$
- *Resultat: Mittlere Laufzeit ist* $\mathcal{O}(\boldsymbol{n} \log \boldsymbol{n})$ *(gemittelt über alle möglichen Felder und im Einheitskostenmaß)*
- *Es gibt Felder mit Worst-Case-Verhalten von* $\mathcal{O}(\boldsymbol{n}^{\mathbf{2}})$*.*

Fortsetzung von 15.25: Quicksort als Las-Vegas-Algorithmus

- Problem: Kommen bei der Anwendung nicht alle Felder mit gleicher Wahrscheinlichkeit vor, greift die Mittelwertanalyse nicht!
- Ausweg: Probabilistischer Algorithmus mit

  (1c) Wähle k zufällig aus $\{i, \ldots, j\}$
- Dann gilt offensichtlich:
  1. Der Algorithmus hält immer.
  2. Er liefert stets das gleiche Resultat (das sortierte Feld)

  d.h.: es liegt ein Las-Vegas-Algorithmus vor.
- Durchschnittliche Laufzeit mit (1c) sofort:

$$T^{\emptyset}_{qs}(n) = \underbrace{c \cdot n}_{(1c),(2)} + \frac{1}{n} \sum_{m=1}^{n} \Big(T^{\emptyset}_{qs}(m-1) + T^{\emptyset}_{qs}(n-m-1) \Big)$$

  wieder mit Lösung $T^{\emptyset}_{qs}(n) = \mathcal{O}(n \log n)$
- Mittelwert ergibt sich sofort mit (1c), ‘Mittelwertanalyse’ unnötig!

Fortsetzung von 15.25: Quicksort als Las-Vegas-Algorithmus

- Sortierzeit bei gegebenem Feld nicht mehr konstant, sondern zufällig...
- Mittlere Laufzeit gilt jetzt bei jedem Feld!
- Achtung: Worst-Case könnte jetzt auch bei jedem Feld eintreten...
- Weitere Verbesserungen z.B.:

  (1d) Wähle drei Werte k_1, k_2, k_3 zufällig aus $\{i, \ldots, j\}$,
  setze k:=Index des mittleren Wertes aus $\{a[k_1], a[k_2], a[k_3]\}$

  Dann tritt der Worst-Case seltener ein, ist aber immer noch nicht ausgeschlossen!

Analoge Vorgehensweise ist bei vielen Algorithmen möglich, die Mengen elementweise verarbeiten:

- Statt determinierter Auswahl eines Element wähle zufällig aus...

Vorteile der Randomisierung:

- (Komplexitäts-)Analyse des randomisierten Algorithmus ist oft signifikant leichter
- keine umfangreichen Wahrscheinlichkeitsuntersuchungen und -annahmen über die Menge aller Eingaben
- zufällige Auswahl ermöglicht Mittelwertuntersuchungen
- unbekannte Verteilung der Eingabe bei det. Algorithmen erlaubt i.A. nur Worst-Case-Analyse
- Nach ‘Randomisierung’ eines Algorithmus evtl. erstmals Average-Case-Analyse möglich
- Bei randomisierten Algorithmen sind alle Eingaben gleich gut (bzw. gleich schlecht).
- Der Anwender muss sich nicht mehr darum kümmern, dass seine Eingaben den Mittelwertbedingungen genügen.

15 ‘Harte’ Probleme

- Typische Problemklassen
- Exakte Verfahren
- Approximative Verfahren
- Randomisierte Verfahren
  - Probabilistische Turingmaschinen
  - Probabilistische Komplexitätsklassen

Definition 15.26

Zu $t\colon \mathbb{N} \to \mathbb{N}$ *seien folgende Komplexitätsklassen definiert:*

$$\begin{aligned}
\texttt{PrTIME}(t) &= \{L_M^{(p)} \mid M \text{ prob. TM } \wedge f_M^{(p)}(x) \text{ exist. für alle } x \\
&\qquad \wedge (\exists c)(\forall x) T_M^{1/2}(x) \leq c \cdot t(lg(x)) + c\} \\
\texttt{MCTIME}(t) &= \{L_M^{(p)} \mid M \text{ prob. TM } \wedge f_M^{(p)}(x) \text{ exist. für alle } x \\
&\qquad \wedge (\exists \varepsilon < 1/2)(\exists c)(\forall x) T_M^{\varepsilon}(x) \leq c \cdot t(lg(x)) + c\} \\
\texttt{RTIME}(t) &= \{L_M^{(p)} \mid M \text{ prob. TM } \wedge f_M^{(p)}(x) \text{ exist. für alle } x \\
&\qquad \wedge (\forall x \notin L_M^{(p)}) err_M(x) = 0 \\
&\qquad \wedge (\exists c)(\forall x) T_M^{1/2}(x) \leq c \cdot t(lg(x)) + c\} \\
\texttt{LVTIME}(t) &= \{L_M^{(p)} \mid M \text{ prob. TM } \wedge f_M^{(p)}(x) \text{ exist. für alle } x \\
&\qquad \wedge (\forall x) err_M(x) = 0 \\
&\qquad \wedge (\exists c)(\forall x) T_M^{\emptyset}(x) \leq c \cdot t(lg(x)) + c\}
\end{aligned}$$

Fortsetzung von 15.26:

Ferner werden die folgenden Klassen definiert:

$\mathtt{PP}$	$= \bigcup_{k \in \mathbb{N}} \mathtt{PrTIME}(n^k)$	Probabilistic Polynomial time
$\mathtt{BPP}$	$= \bigcup_{k \in \mathbb{N}} \mathtt{MCTIME}(n^k)$	Bounded error Probabilistic Polynomial time
$\mathtt{RP}$	$= \bigcup_{k \in \mathbb{N}} \mathtt{RTIME}(n^k)$	Random Polynomial time
$\mathtt{ZPP}$	$= \bigcup_{k \in \mathbb{N}} \mathtt{LVTIME}(n^k)$	Zero-error Probabilistic Polynomial time

Satz 15.27

Für Polynome $t : \mathbb{N} \to \mathbb{N}$ *gilt:*

$$\begin{aligned}
\mathtt{DTIME}(t) &\overset{*}{\subseteq} \mathtt{LVTIME}(t) \\
&\subseteq \mathtt{RTIME}(t) \left\{ \begin{array}{l} \subseteq \mathtt{NTIME}(t) \overset{*}{\subseteq} \\ \overset{*}{\subseteq} \mathtt{MCTIME}(t) \subseteq \end{array} \right\} \mathtt{PrTIME}(t) \\
&\overset{*}{\subseteq} \mathtt{DSPACE}(t)
\end{aligned}$$

(Nur bei $\overset{*}{\subseteq}$ *wird benutzt, dass* t *Polynom ist*
(genauer: es wird die 'Zeitkonstruierbarkeit' von t *verwendet),*
die anderen Inklusionen gelten sogar für alle t*.)*
Damit gilt:

$$P \subseteq \mathtt{ZPP} \subseteq \mathtt{RP} \left\{ \begin{array}{c} \subseteq \mathtt{NP} \subseteq \\ \subseteq \mathtt{BPP} \subseteq \end{array} \right\} \mathtt{PP} \subseteq \mathit{PSPACE}$$

Einschub:

Inklusionsbeziehungen der prob. Komplexitätsklasse

- Inklusionsbeziehungen zwischen **BPP** und **NP** sind unbekannt!
- **PP**, **BPP**, **RP** und **ZPP** sind unter $\leq_p$ nach unten abgeschlossen (Beweis analog wie bei ***NP***)
- Vollständige Probleme nur für **PP** bekannt:

$$\textbf{MAJ} := \{\text{Formel } \phi \mid \begin{array}{l}\phi \text{ ist unter mehr als der Hälfte aller} \\ \text{Belegungen der Variablen aus } \boldsymbol{w} \text{ wahr}\}\end{array}$$

MAJ ist **PP**-vollständig (bzgl. $\leq_p$).

Unterschied zwischen $\mathtt{PrTIME}(t)$ und $\mathtt{MCTIME}(t)$ ist scheinbar gering:

$$(\exists c)(\forall x) T^{1/2}{}_M(x) \leq ct(lg(x)) + c$$

und

$$(\exists \varepsilon < \frac{1}{2})\ (\exists c)(\forall x) T^{\varepsilon}{}_M(x) \leq ct(lg(x)) + c$$

Jedoch:

- Bei Monte-Carlo-Maschinen ist Wahrscheinlichkeit einer nicht korrekten Ausgabe beliebig reduzierbar!
- Sie ist leicht unter die Wahrscheinlichkeit für unentdeckte Hardwarefehler zu drücken!

Satz 15.28

Sei $t : \mathbb{N} \to \mathbb{N}$ *Polynom (bzw. zeitkonstruierbar). Für jedes feste* ε, $0 < \varepsilon < \frac{1}{2}$, *gilt:*

$$\begin{aligned}
\mathtt{MCTIME}(t) &= \{L_M^{(p)} \mid M \text{ prob. TM } \wedge f_M^{(p)}(x) \text{ exist. für alle } x \\
&\qquad \wedge (\exists c)(\forall x) T_M^{\varepsilon}(x) \leq ct(lg(x)) + c\} \\
&= \{L_M^{(p)} \mid M \text{ prob. TM } \wedge f_M^{(p)}(x) \text{ exist. für alle } x \\
&\qquad \wedge (\forall x) err_M(x) < \varepsilon \\
&\qquad \wedge (\exists c)(\forall x) T_M^{\emptyset}(x) \leq ct(lg(x)) + c\}
\end{aligned}$$

- Für Praxis ohne Unterschied, ob Algorithmus deterministisch / Fehlerwahrscheinlichkeit unter 2^{-100} !
- 'deterministische' Ergebnisse könnten Hardware-Fehler haben...
- Kernkraftwerke gelten bei wesentlich höheren Katastrophenwahrscheinlichkeiten noch als sicher!
- **Probleme aus BPP sind also (bei moderaten Exponenten in der jeweiligen polynomialen Zeitschranke) mit gleichen Recht als 'praktisch berechenbar' anzusehen wie Probleme aus *P*.**

Einschub:

Fehlerreduktion für Monte-Carlo-Maschinen

- praktisch verwendete Monte-Carlo-Algorithmen:
  Primzahltests von Rabin (1976) und Solovay/Strassen (1977)
  $\Rightarrow$ Menge aller Primzahlen liegt in $\boldsymbol{co} - \mathtt{RP}$
- Adleman/Huang (1992): Menge aller Primzahlen liegt in $\mathtt{RP}$
  $\Rightarrow$ damit: Primzahlen in $\mathtt{ZPP}$
- Agarwal/Kayal/Saxena (2002): Primzahlen in $\boldsymbol{P}$
- Algorithmus noch (deutlich) langsamer als probabilistische Algorithmen
- keine Auswirkungen auf Kryptographie zu erwarten!

Anwendung großer Primzahlen z.B. bei den ‘public key cryptosystems’ nach dem RSA-Verfahren (Rivest/Shamir/Adleman, 1977).