Berechenbarkeit und Komplexität

Zusammenfassung 2000 / 2001 von Klaus Ridder und Matthias Hensler

1. Algorithmenbegriff und Turingmaschine

1.1. Wörter, Zahlen und Funktionen

Wörter, Wortfunktionen: Datentransformation Wort → Wort

z.B. Programmiersprache → Maschinencode.

Alphabet S = Menge aus Symbolen.
Wort **u, v, w**, Länge |w|, e = leeres Wort.
S* = alle möglichen Wörter des Alphabets Σ
S^+ = S* ohne e
$\mathbf{u}^R$ = Wort rückwärts
Verkettung von 2 Wörtern u · v: direkt aneinanderhängen ohne Leerzeichen.

(Bem.: Verkettung ist Halbgruppe mit neutralem Element (keine Gruppe, da kein inverses), (nicht Kommutativ)

1.1.1. Wörter und natürliche Zahlen

Kanonische Reihenfolge der Wörter:
(**m** = Anzahl der Buchstaben im Alphabet, **L** = Länge des Wortes)

1,2,...,m,11,12,...,1m,21,...,2m,...,m1,...,mm,111,...

also zuerst die 1-Buchstaben-Wörter, dann die 2-Buchstaben-Wörter, dann die 3-Buchstaben-Wörter, ect.

Anzahl der Wörter der Länge l: $\mathbf{m^L}$
Anzahl der Wörter der Länge maximal l: $\mathbf{m^L + m^{L-1} + m^{L-2} + ... m^1} = (m^{L+1}-1) / (m-1)$

d_4**(224)** = **44** = Position des Wortes 224 in der Kanonischen Reihenfolge beim Alphabet mit **4** Buchstaben.

$\mathbf{4}\cdot 4^0 + \mathbf{2}\cdot 4^1 + \mathbf{2}\cdot 4^2 = \mathbf{4}\cdot 1 + \mathbf{2}\cdot 4 + \mathbf{2}\cdot 16 = 44$

allgemein: $k_0\cdot m^0 + ... + k_0\cdot m^0$ für das Wort $k_r...k_0$ (beachte: Rückwärts!!)

$g_2(4)$ = 12 = das **4.** Wort in der kanonischen Reihenfolge beim Alphabet mit **2** Buchstaben.
(also 1,2,11,**12**).

Allgemein: $g_m(n)$ = Wort $k_r...k_2k_1k_0$ mit $n= k_r \times m^r + ... + k_1 \times m^1 + k_0 \times m^0$

Beispiel: $g_3(14)$ = Wort 112 mit $14= 2 \times 1 + 1 \times 3 + 1 \times 9$

Konkret: $\gamma_4(44)$ = Wir suchen die größte Zahl, die mit 4 multipliziert gerade noch kleiner als 44 ist: Das ist 10. (Der Rest – hier 4 – ist nun die letzte Zahl unseres gesuchten Wortes.). Nun suchen wir die größte Zahl, die mit 4 multipliziert gerade noch kleiner als 10 ist, der Rest bieldet nun die vorletzte Zahl, und wir machen mit der gefunden Zahl – hier 2 – weiter, etc. So lange, bis di gefundene Zahl = 0 ist → fertig.

$44 = 10 \cdot 4 + 4$
$10 = 2 \cdot 4 + 2$
$2 = 0 \cdot 4 + 2$ → 224

Wortfunktionen:

f,g,h: Funktionen, Def(f), Bild(f)

partielle Funktion: A - -> B, wenn nur ein **Teil des Wertebereiches** verwendet wird.
Totale Funktion: A → B, wenn der **gesamte Wertebereich** erfasst wird.

^ = f(a) **undefinert**, d.h. a nicht in Def(f).

1.2. Algorithmische Probleme, Algorithmen

1.2.1. Algorithmus = Wortfunktion.

Beispiele:
1. Produkt
2. Primfaktorzerlegung
3. Wurzel
4. Teilerfremdheit
5. Polynom: hat es Nullstelle ? Polynome wie in LATEX schreiben (in 1 Zeile)
6. M3-Programm: terminiert für Eingabe 0? (nicht algorithmisch lösbar!)
7. Graph: Weg von s nach t? → Codierung: AnzahlKnoten,Kantenliste(1-2),s,t.

f: (**{0,...,9,/}***)^2 → **{0,...,9,/}***

* = 1,12,1444,... erlaubt (mehrere Zeichen, d.h. alle Wörter über Alphabet)
^{2} = man kann 2 Eingaben machen.

1.2.2. Entscheidungsprobleme:

Entscheidungsprobelm (Ausgabe nur 0 oder 1): gut durch Relation darstellbar:

Relation R = alle **Wörter** aus unserem Wertebereich, die als **Ausgabe 1** haben.
f ist dann die „**characteristische Funktion**“ dieser Relation.

Algorithmus =
- verarbeitet **schrittweise** Wörter,
- eindeutiger, endlicher **Text**
- **Ausgabe** oder **nicht terminiert**.

Funktion berechenbar = Algorithmus existiert: ***F total*** = *immer eine Ausgabe !!!*
Eingabe Î **DEF** → Algorithmus macht **Ausgabe**,
Eingabe **nicht** Î **DEF** → Algorithmus **terminiert nicht**.

Relation bzw. Wortmenge entscheidbar = Algorithmus existiert:
Algorithmus **terminiert immer**, und zwar mit **0** oder **1**.
(d.h.: Ausgabe 0, wenn Wort nicht ∈ der Relation ist.)
in dieser Relation liegen dann alle Wörter drin, die 1 liefern.

Satz 1.2.1:

f berechenbar « R_f entscheidbar:

→ Wir geben der Relation ein Paar **u,v**: **u** schicken wir durch die Funktion, und vergleichen die Ausgabe mit **v**. Ist sie gleich (also **u,v** ein **Eingabe-Ausgabe-Paar** der Funktion), dann geben wir **1** aus, sonst **0**.
... also: unsere Relation ist praktisch ein „Tester“, ob ein Wertepaar ein x-y-Paar der Funktion ist.

← Also. Zu jeder Eingabe, z.B. 1,2,3,4, haben wir in einer „Tabelle“ unserer Relation gespeichert, was die Ausgabe ist.

vertikal: Eingabe, horizontal: Ausgabe.

	1	2	3	4
1			x	
2	x			
3				x
4		x		

Nun gehen wir z.B. bei 4 alle Werte 1,2,3,4 durch, bis wir ein x finden; das ist hier bei 2.
Diese 2 geben wir aus. → fertig.

1.2.3. Semi-entscheidbare Relation:

... wie entscheidbare Relation, nur statt **0** gibt es ein „**nicht terminiert**“.

1.2.4. Satz 1.2.2: W entscheidbar « w semi-entscheidbar

Ausgabe des Alphabets:

RRRRRRRRRRRNNNNNNNNNNNNNNNNN R=Relation1, N=Relation2
11111111111111100000000000000000000000

Relation=Wortmenge, die 1 liefert.
Wir haben einen Algorithmus, der bei einer Eingabe 1 oder 0 liefert: Liefert er 1, so liegt der Wert in unserer Relation R1, liefert er 0, so liegt der Wert in unserer Relation R2.
R1 und R2 zusammen bilden das gesamte Alphabet.

→ für R1: Ausgabe 1 → Ausgabe 1, Ausgabe 0 → keine Ausgabe.
für R2: Ausgabe 1 → keine Ausgabe, Ausgabe 0 → Ausgabe 1.

← Wie haben von R1 und R2 je einen Algorithmus, der jeweils 1 ergibt, wenn das Wort in der entsprechenden Relation drin ist, sonst nicht terminiert.
Wir lassen nun beide Algorithmen wechselweise 1 Schritt laufen: Wenn der 1. terminiert, geben wir 1 aus, wenn der 2. terminiert, geben wir 0 aus. → fertig.

1.2.5. Satz 1.2.3:

diesmal betrachten wir eine nicht totale Funktion (d.h. der Def.-Bereich ist nicht die komplette Wortmenge des Alphabetes).

nicht totale Funktion berechenbar « Graph R_f semi-entscheidbar.
(Graph, weil jedeim Eingabe-Tupel u,v 1 (Punkt) pder 0 (kein Punkt) zugeordnet wird.

→ u,v in Funktion: f(u)=v → 1, sonst nicht terminieren.

← Wir geben ein u vor, wissen aber nicht, welches v (entsprechend der Funktion f) zu diesem u gehört (falls es überhaupt eins gibt).
PROBLEM: wir können nicht einfach der Reihe nach alle v’s durchprobieren, da der Algorithmus gar nicht terminiert, wenn das v nicht zu dem u passt, und wir so nie fertig werden würden.
LÖSUNG: Wir laufen mit Der 1. Möglichkeit für v (Wort1) den 1. Schritt:
Wort 1 – 1. Schritt
Wort 1 – 2. Schritt, Wort 2 – 1. Schritt.
Wort 1 – 3. Schritt, Wort 2 – 2. Schritt, Wort 3 – 1. Schritt.
Wort 1 – 4. Schritt, Wort 2 – 3. Schritt, Wort 3 – 2. Schritt, Wort 4 – 1. Schritt.

WENN der Algorithmus nun für irgendeinen dieser v’s terminiert, so finden wir dieses v in endlicher Zeit. → fertig.

Aufzählungsalgorithmus:

... einfach ein Algorithmus, der die Elemente irgendeiner Relation aufzählt. Ohne Eingabe.
(z.B.: die Relation „alle Primzahlen“ ist aufzählbar, endet aber nie.)

beachte: Algorithmus darf sich wiederholen, und in beliebiger Reihenfolge ausgeben.

1.3. Turingmaschine

abstaktes Modell für ALLE Algorithmen (wenn auch u.U. ziemlich komplex)

Eingabe-Tupel: $M = \{Q, \Sigma, \Gamma, q0, qs, \delta)$

Q = Zustandsmenge: $q_1, q_2, q_3, ..., q_n$ - endlich !
Σ = Eingabealphabet: a,b,c,|,*,Blank,/,\, ...
Γ = Arbeitsalphabet: Eingabealphabet + weitere Zeichen:
diese dürfen nicht eingegeben, aber von der T.-Maschine geschrieben werden.
$q_ß$ = Anfangszustand
q_s = Stopzustand
δ = Transitions-Funktionen (-Tupel):

Ein Tupel: d**(q,a)=(q’, a’, LRN)**
q= bisheriger Zustand
a=bisheriges Zeichen an der aktuellen Position

q’=neuer Zustand
a’=neues Zeichen zu schreiben
LRN=Bewegung danach: rechts, links, no

f **Turing-berechenbar**: es gibt eine Turing-Maschine, die f berechnet.
R **Turing-entscheidbar**: es gibt eine Turing-Maschine, die R entscheidet.

beachte: Ausgabe = von Position bei Stopzustand bis zum nächsten Blank.

(Γ-Konfiguration: Zustand, Symbole links von Pos., Symbole rechts von Pos. inkl. aktuell.)

Algorithmus (nicht) berechenbar « Algorithmus (nicht) Turing-berechenbar.

1.4. Mehrband-Turingmaschinen

Mehrere Bänder
Stop-Kondifuration: wie vorher, auf Band 1.

Diese lässt sich auf 1 Band zurückführen: einfach alle Bänder aneinanderhängen, und aktuelle Position jeweils durch Unterstreichung klar definieren.

2. WHILE- und GOTO – Programme

2.1. Funktionen über N, induktive Definitionen

Definition normaler Rechenregeln; immer definiert.

Einsetzung und Iteration von Funktionen:

Man kann eine Funktion in mehrere Funktionen einsetzen.
Eine Funktion kann dabei natürlich auch mehrere Eingaben erwarten, die dann wiederum Funktionsergebnisse sind.
Ist irgendeiner dieser Schritte nicht definiert, so ist es die ganze Funktion nicht.

Induktive Definition einer Menge aus M:

Eine Menge durch Induktion definieren (z.B. 0, n→n+1)

Induktive Herleitung Funktionen:

Ergebnis einer Funktion aus den Ergebnissen vorheriger Funktionen herleiten, z.B.:
Wortlänge: Länge ($\in$) = 0, Länge (wa) = Länge (w) +1, für a=Buchstabe.

Induktive Beweise:

normale Induktion.

2.2. WHILE-Programme:

Syntax, Semantik

SYNTAX:

- Wertzuweisungen: Xi=0;
- BEGIN # # END;
- IF # THEN # ELSE # END;
- WHILE # DO #;
- LOOP Xi #;

SEMANTIK:
"Zustand" = Vektor alle Variablen
„Projektion" = 1 Variable i daraus.
Wertzuweisung: ändert jeweils 1 Wert in diesem Vektor: x1:=0 ist (x1,x2,x3) → (0,x2,x3)
... etc.

Beispiel für Semantik 2:

wir schreiben einen Zustand [P](k1, k2, ...). Das bedeutet:
Wir führen das gesamte Programm P aus, und zwar mit dem Variablenvektor (k1, k2, ...).
Dann ersetzten wir [P] (..) durch [P2]([P1](..)), etc.
$[P3]^{k2}$ = P3 k2-mal ausführen.

While-Berechenbarkeit: Es gibt ein WHILE-Programm, das f berechnet.
While0-Berechenbarkeit: wie oben, aber keine direkten Wertzuweisungen benutzen.
klar: While-Berechenbar → While0-Berechenbar.

2.3. Vergleich von LOOP und WHILE

LOOP ist durch WHILE simulierbar:
WHILE ist aber nicht durch LOOP simulierbar!

2.4. GOTO – Programme

Syntax: Zeilen durchnumerieren, IF → GOTO.
Man kann: ++, --, vergleichen, JUMP ON ZERO.
→ entspricht Assembler mit DEC, INC, BEQ (BRACH-EQUAL-ZERO)

2.5. Der Äquivalenzsatz

zu zeigen: **While → Goto → TM → While**

.1.) **WHILE → GOTO**:

- BEGIN # # END;

klar, einfach 2 GOTO-Blöcke aneinanderhängen.

- IF X>0 THEN # ELSE # END;

BOZ übernext line; GOTO END;

- WHILE X>0 DO #;

1 BOZ übernext line; GOTO END;
2 #####
3 GOTO 1;

- LOOP Xi #;

1 BOZ END;
2 ####…..
3 X:=X-1;
4 GOTO 1;

q.e.d.

GOTO → TM (TM = Touringmaschine)

X:=X+1: 5 = #|||||# ,

5→ 6:

- alles nach # eins nach rechts verschieben
- Raute durch | ersetzten, Raufe 1 nach rechts,
- fertig.

X:=X-1 analog,

BOZ:

- Wenn # folgt → Zustand für Sprung
- Wenn | folgt → Zustand nicht für Sprung.

→ q.e.d.

TM → While

In WHILE dürfen wir:

- Wertzuweisungen: Xi=0;
- BEGIN # # END;
- IF # THEN # ELSE # END;
- WHILE # DO #;
- LOOP Xi #;

Wir speichern das gesamte Band links und rechts von der aktuellen Position nach folgendem Schema:
Bei z.B. 10 verschiedenen Zeichen (0...9): schreibe einfach die komplette Zahlenkette links auf, und die komplette Zahlenkette rechts, aber rückwärs.
BEISPIEL:
1234-5-6789 (aktuelle Pos. = 5)
LinkesBand := 1234; RechtesBand := 9876.

Bei Bewegung nach Links: NachLinks():
1.) Berechne 1234 MOD 10 = **123**, Rest 4. → Neue Pos.Wert = 4 (unser Rest),
→ linkesBand = 123
→ rechtesBand = 9876*10 + 5 = **98765**
Bewegung nach rechts analog dazu.

Wir müssen also speichern: (INT ist hier Unlimited-INT)

- **LinkesBand**, **RechtesBand** : INT;
- **AktuellerPositionsWert**: INT; (Zeichen Codiert in INT)
- Zustandsübergänge:
  - IF Z1 and AktPosWert=7 THEN AktPosWert=8; NachLinks();

2.5.1. Äquivalenzsatz – Zusammenfassung:

→ GOTO und WHILE reicht für ALLE Algorithmen aus!
→ nur feste Anzahl Schleifen benötigt, nur 1 WHILE-Schleife:
(einfach nach TM wandeln und zurück ;))

Satz 2.5.5.: Normalformensatz:

While-Berechenbar → Programm möglich mit:
- 1 While-Schleife
- Loop-Schleifen, deren Anzahl nur von n (Anzahl der Eingabevariablen) abhängt.

rekursive Funktionen = WHILE- / GOTO- / TM- Berechenbar

Funktionen aus Grundfunktionen bildbar durch
- einsetzen: 2 Funktionen ineinander setzen: aus g und h mache f.
- Rekursion: Funktion rekursiv definiert.
- µ - Operator: g→f: wir wählen den kleinstmöglichen Wert für den letzten Eingabewert (y), so dass die Funktion noch 0 ergibt. Die neue Funktion f hat nun alle Eingabeparameter außer y, und y als Wert. Sie terminiert nicht, wenn kein solches y existiert.

Grundfunktionen:
I- INC(X);
II- Wert von xi (aus dem Vektor x)
III- Konstante k.

3. Kapitel: Unentscheidbarkeit

Nicht alle Algorithmen (=Turingmaschinen) bzw. kombinatorische Probleme sind lösbar.

3.1. Wortproblem für Turingmaschine, Diagonalisierung

Def.:
M: w→stop: Turingmaschine **M** stoppt bei Eingabe von **w**.
M: w→v: Turingmaschine **M** stoppt bei Eingabe von **w** mit Ausgabe **v**.
M: w→stop: Turingmaschine **M** stoppt bei Eingabe von **w** nicht.

„**Wortproblem**“ für TM: Stoppt die Turingmaschine bei einem bestimmten Eingabewort?
Dieses Problem ist **unentscheidbar**!

WARUM? → Ganz einfach (na ja, fast):

1.) wir kodieren die Turingmaschine als einen langen Wortstang:
(soVieleNullenWieVorAktuellerPosition+1+ soVieleNullenWieNachAktuellerPosition+1+....
eine 1 als Trenner zwischen den einzelnen Zuständen, und noch mal eine 1 am Ende. Auf jeden Fall ist das eindeutig.)

Nun machen wir folgendes:
Wir nehmen an, wir haben eine Turingmaschine M_0, die entscheidet, ob M mit Eingabe z.B. M terminiert. Ist dem nicht so, so gibt sie 0 aus.
Desweiteren haben wir eine Turingmaschine M1, die zu M_0 identisch ist, mit der Ausnahme, dass sie statt 0 STOP ausgibt und statt 1 nicht terminiert.
M_0 bekommt also als Eingabe <M>w, in diesem Fall also <M><M>. Dies soll jetzt mal nicht terminieren (M:(M) → ∞) Dann gibt M_0 also 0 aus. M1 gibt dann nach Voraussetzung Stopp aus.

D.h.: für alle Wörter M gilt: wenn M angewandt auf M nicht terminiert, dann gibt M_0 0 aus, und M_1 Stop (nach Voraussetzung, siehe auch S. 42 unten).
Wenn wir nun als Eingabewort M M_1 nehmen, haben wir einen Widerspruch:
M1:M1 → Stop ↔ M1:M1 → nicht terminiert - WIDERSPRUCH !

Cantors Beweis: Nehmen wir eine Liste aller Bitfolgen, kommt die invertierte Diagonale NIE vor. (Glaubt es uns, Cubbi und ich haben es zum Exzess ausprobiert – es geht einfach nicht.)

Satz 3.1.3: Man kann nicht alle totalen Funktionen aufzählen (s.o.).
Denn: man nehme z.B. die totale Funktion, die z.B. für F(20) die 20. Funktion nimmt, den Wert damit berechnet und 1 dazuzählt. Diese Funktion ist dann nicht in der Aufzählung vorhanden (analog zu Cantor’s Beweis).

Daraus folgt: ∃ eine totale, berechenbare Funktion, die durch kein P_i berechnet wird.
(Pi = Menge von Programmen, die totale Funktionen berechnen).

3.2. Reduktion, weitere unentscheidbare Probleme

Idee, um weitere unentscheidbare Probleme zu finden: Noch schwerere finden als bereits bekannte unentscheidbare Probleme.

<Vorlesungs-Mitschrift>
Format eines Entscheidungsproblems P:
Gegeben: Objekt X $\in$ Inputbereich I_P (I_P = Menge der Probleminstanzen)
Frage: gehört X zu vorgegebener Menge P ? (P aus I_P ist die Menge der Eingaben, die die Antwort „JA“ liefern.)
Formal: ist P = (I_P, P) mit P $\subseteq$ I_P.

DEF: Das Problem P ist reduzierbar auf das Problem Q, (P $\leq$ Q), falls es eine berechenbare Funktion gibt, die jedem Input, der P terminieren lässt, auch Q terminieren lässt, und falls P nicht terminiert, auch Q nicht terminieren lässt.

Reduktionslemma: Seit P $\leq$ Q. Wenn P unentscheidbar ist, dann ist es auch Q.

Beispiel: Wortproblem für Turingmaschine über {0,1}:
Wortproblem WP = (I_{WP}, WP) mit I_{WP} = Menge der Paare (Turingmaschine M über {0,1}, Wort w aus der Menge aller 0-1-Folgen.

3.2.1. Halteproblem unentscheidbar:

klar: wenn für eine TM mit keinem der Eingabewörter entschieden werden kann, ob sie stoppt, geht es auch nicht mit dem leeren Eingabewort (Beweis: Eingabewort einfach hart kodiert in TM packen)

3.2.2. Äquivalenzproblem unentscheidbar

Gegeben: Eingabewörter M1, M2 (beide 0-1-Folgen).

Frage: berechnen beide Maschinen dasselbe?
Wir müssen vom Halteproblem auf das Äqu. - Problem schließen: Wir machen also aus einer zwei Turingmaschinen.
M1: wie M, nur Eingabewort vorher gelöscht. Und am Ende noch mal BLANK schreiben.
M1: liefert also BLANK, wenn M stoppt).
M2: liefert einfach immer BLANK.

f: M1 $\rightarrow$ M2 ist also berechenbar.: diese beiden Funktionen berechnen genau dann das gleiche Wort, wenn die Funktion M stoppt.

Def. 3.2.3:
semantisch: es kommt nur auf das Ein-Ausgabeverhandeln der TM’s an, nicht auf deren Aufbau. (Entscheidungsproblem P = (alle Eingaben, zu überprüfende Eingaben).)

Eine Entscheidungsproblem ist **nicht-trivial**, wenn es TM’s innerhalb und außerhalb der Betrachtung (P) gibt.

3.2.3. Satz 3.2.3: (von Rice):

jedes semantische (kommt nur auf E/A-Verhalten an), nicht-triviale Entscheidlungsproblem über TM's ist unentscheidbar.

3.3. Dominoproblem und Post'sches Korrespondenzproblem.

Domino-Parkettierung von links oben nach rechts unten. (4. Quadrant).
ZEIGE NUN: Das ist nicht entscheidbar.
Beweis: Rückführung auf TM: jede TM als Domino-Parkett darstellbar:

Eine TM-Kondiguration war das folgende: (unser Band):

b = irgendwelche Zeichen
q = Zustand
a = aktuelles Zeichen, wird mit a' überschrieben,

z.B. wenn wir nach rechts gehen:

b	b	b	b	qa	b	b	b	b	b
				qa	qa				
b	b	b	b	a′	q′b	b	b	b	b

der Zustand steht immer vor dem aktuellen Zeichen!

bei dem „Übergangsstein" steht an der Seite noch „qa", sonst immer ein anderes eindeutiges Zeichen (z.B. „*" oder so.)

Damit entspricht das Dominospiel der TM .

3.3.1. Das post'sche Korrespondenzproblem

Wir haben 2 Lisen von Wörtern, z.B.:
{1 10; 011}sowie {101, 00, 11}.
Die Frage ist nun, ob wir eine Indexfolge finden, so dass bei BEIDEN Listen, wenn man die Wörter einfach hinztereinanderschreibt, das gleiche rauskommt. Dies ist hier möglich:
1-3-2-3: 1 011 10 011 = 101 11 00 11

3.4. Ausblick in die Rekursionstheorie

Univ = enthält alle TM (0/1) und Wörter (w: 0/1) die zusammen stoppen.

3.4.1. Satz 3.4.1: Univ ist Semi-Entscheidbar:

Beweis: man nehme Eingabe v als TM+w. Bei Stop von TM+w gebe 1 aus.

Erinnerungen:
Satz 1.2.2: Alphabet S entscheidbar $\leftrightarrow$ W aus S semi-entscheidbar und $\neg$W aus S semi-ents.
Satz 1.2.4: Sprache semi-entscheidbar $\leftrightarrow$ Sprache aufzählbar.

$\rightarrow$ zeige: es gibt Sprachen, die nicht semi-entscheidbar sind.
Dafür nehmen wir alle 0-1-Wörter, die nicht in Univ enthalten sind. Dann können wir sagen: die Wörter, die nicht in Univ enthalten sind, sind nicht semi-entscheidbar ist, da nicht beide Mengen gleichzeitig semi-entscheidbar sein (sonst wäre die ganze Menge zusammen entscheidbar).

3.4.2. Satz 3.4.2:

(semi-entscheidbar: du kannst feststellen, wenn das Wort in der Menge drin ist).

Eine Wortmenge ist **aufzählbar,** genau dann wenn :
Wenn wir ein Wort **u** vorgeben, können wir sicher sagen (entscheiden) ob es ein **v** gibt, so dass **(u,v)** in der Relation liegt.

$\rightarrow$: Gegeben ist also ein Algorithmus A, der alle Wörter der Sprache L aufzählt. Gesucht ist jetzt so eine Relation, die Entscheidbar ist. Also definierst du dir da einfach die Relation über alle (u,v) - Tupel, wobei diese Relation zu jedem u die Position in der Auszählung des Algorithmus zuordnet. q.e.d.
Unsere (u,v) - Tupel sind dann (Wort, Position des Wortes).

$\leftarrow$: ganz einfach: alle möglichen (u,v) durchgehen. (z.B. Zick-Zack-Algorithmus, oder nach der Summe der Länge von u und v). Nun alle u ausgeben, wenn (u,v) in der Relation ist. q.e.d.

Def. 3.4.1:
Sigma-1-Sprache:
Wir haben 2-Tupel von Wörtern.
Genau dann, wenn das erste Wort in der Sprache L enthalten ist, ist das Tupel Teil der Relation. Das zweite Wort ist egal.

Sigma-n-Sprache: („arithmetische Hierarchie“):
Wir haben jetzt (n+1)-Tupel von Wörtern. Diese bilden wieder eine Relation:
Die Wörter der Sprache L bilden das erste Element (u). Danach folgen abewechselnd Elemente, für die $\exists$ und $\forall$ gilt: BEISPIEL:
L={2; 3}. Elemente der Relation sind dann z.B.:
R={ (2,7,n), (3,9,m) (n und m beliebige natürliche Zahlen) }. oder:

R={ 2,7,n,4,m,2,o,7,…), … }

3.4.3. Satz 3.4.3:

Wdh.: Sprache Univ = enthält alle TM (0/1) und Wörter (w: 0/1) die zusammen stoppen.

Diese Sprache Univ ist **aufzählbar**, und und jede andere Sprache **L** ist **kleiner als Univ**.

3.4.4. Satz 3.4.4.: Rekursionstheorem (???)

Für jeden Algorithmus, der ein Programm sabotieren soll, gibt es immer 1 Programm, bei dem dieser Algorithmus keinerlei Wirkung hat.

4. Zeitkomplexität

4.1. Einführung: Zeitbeschränkte Turingmaschinen

Problem lösbar, aber in unendlicher Zeit??

Def. 4.1.1: Worst-Case-Zeitkomplexität:
f(n)-Zeit-Turingmaschine: Die maximale Anzahl der Schritte, die eine TM bei einem Eingabewort der Länge n laufen kann.
O(g(n)) = wie in Datenstrukturem, also z.B. $f(n)=20n^2 \rightarrow O(g(n)) = n^2$.

Def. 4.1.2: TIME(t(n)) = Sprache, welche von einer O(t(n)) – zeitbeschränkten Turingmaschine entschieden wird.

Beispiel-Sprache: k Nullen, dann k Einsen.
TM prüfe, ob Wort Teil dieser Sprache:
Möglichkeit 1: $O(n^2)$:

1.) Schaue, ob auch wirklich nur Null und dann Eins
2.) Lösche eine 0 und eine 1
3.) Noch beide vorhanden? → 2.)

Möglichkeit 2: O(n*log n): Lösche immer die Hälfte aller Nullen und Einsen (wenn Anzahl gerade).

2-Band-Turingmaschinen: Verbesserung möglich! (hier: O(n): Nullen auf Band 2 kopieren, dann auf Band 1 (mit Einsen) und 2 (mit Nullen) simultan laufen.

PROBLEM: Wir wollen eine maschinenunabhängige Betrachtung!

Satz 4.1.1:
K-Band-Turingmaschine mir t(n) → 1-Band-TM mit $O(t^2(n))$

Warum? → 1 m-Schritt → ganzes Band max. 4-mal durchlaufen.
Die Anzahl der Bänder ist dabei egal.

4.2. Die Klasse P

Probleme, die man in polynomialer Zeit lösen kann (z.B. p^6).
So eine Sprache nennt sich „**effizient entscheidbar**“.

Weitere Beispielprobleme:

1.) Graph-Erreichbarkeitsproblem: Gibt es Weg? → codiert in TM.
2.) Knoten-Erzeugbarkeitsproblem:
wenn alle umgebenden Knoten „S“ sind, ist es auch der aktuelle Knoten.
Man geht nun n-mal alle Knoten durch. (Zielknoten*Quellknoten), also n^3.

4.3. Die Klasse NP

Probleme: (Beispiel: 3-Färbung eines Graphen.)
Wenn wir eine Färbung haben, können wir in polynomialer Zeit festetellen, ob diese „korrekt" ist. Wir haben aber exponentielle viele mögliche Färbungen (3^{10}): bei 3 Farben und 10 Partitionen:

Def. 4.3.3: **Klasse NP**:
EINE Lösung für das Problems zu überprüfen ist in polynomialer Zeit möglich.
Es gibt aber sehr viele Lösungen – exponentiell wachsend zu der Eingabemenge.

ein Wort w ist Element der Sprache L genau dann, wenn es eine Partitionierung v gibt, ...
?????

Def. 4.3.4: **NTM**: Nichtdeterministische Turingmaschine: wir bisherige, aber wir können von einem Zustand in n weitere Zustände kommen. Wir müssen praktisch alle Möglichkeiten durchsuchen, um einen Weg zu finden, der als Ausgabe dann 1 liefert.

Def. 4.3.5: **NTM** ist **t(n)-zeitbeschränkt**, wenn sie nach weniger als t(|w|) Schritten stoppt.

Def. 4.3.6: **NTIME**(t(n))) = Sprachen, die durch t(n)-zeitbeschränkte TM entschieden wird.

Satz 4.3.1: Jede NTM lässt sich durch eine DTM simulieren.
Satz 4.3.2: Aufwand dafür = $2^{O(t(n))}$.

Satz 4.3.3: NP-Sprachen werden durch eine t(n) – NTM entschieden.
Der Zeitaufwand ist polynomial in w.

4.4. NP-vollständige Probleme:

Def. 4.4.1: Polynomzeit-Reduzierbarkeit:
Es gibt eine Funktion, in polynomialer Zeit eine Sprache in eine andere überführen kann.
Die neue Sprache ist in der gleichen Klasse wie die alte (P oder NP).

Def. 4.4.2: NP-vollständig:
Eine NP-vollständige Sprache ist mindestens so schwierog wie JEDE andere Sprache in NP.

Man kann NP-vollständige Probleme nie in P lösen.

Satz 4.4.1 (Cook, 1971): Das Problem, ob aussagenlogische Formeln φ erfüllbar sind, ist NP-vollständig.

Def. 4.4.7: SAT = Menge der erfüllbaren Formeln in KNF. Dies ist NP-vollständig:
→ BEWEIS ??????

Satz 4.4.3: 3SAT ist NP-vollständig (Formel φ in KNF, wobei in jeder Klausel maximal 3 Literale vorkommen dürfen).

Satz 4.4.4: COLOR3 ist NP-vollständig.
→ BEWEIS ????

5. Ausblick und Rückblick

5.1. Platzkomplexität

Trennung: Eingabegröße, Arbeitsgröße.

Def. 5.1.1: Offline-Turingmaschine: Eingabeband (read-only) + Arbeitsband (rw)
Zustand, Eingabe, alt/ne-Arbeitsband, <-> Eingabeband, <-> Ariebtsband.

DSPACE(f(n)) = alle deteministischen TM, die fn-Platzbeschränkt sind.
(statt D in DSPACE ein N = nitdeteministische TM)

Beispiel: NLOG (Graph).

Def. 5.1.2: quantifizierte Formel = Formel mit Quantoren /ex, /all.
Voll quantifiziert = alle Variablen quantifiziert.

Satz 5.1.2:
Wenn man voll quantifizierte boolsche Formeln hat d.h.($\exists/\forall x_1,...,\exists/Ax_n$ ($x_1...x_n$)), dann kann man gucken, ob die erfüllt ist oder nicht. Einfach rekursiv alles durchgehen.
Das liegt in PSPACE, weil wir immer n Variablen brauchen. (? Cubbi hat's verstanden)

Def. 5.1.3: Sprache $\mathbf{L_0}$ ist **PSPACE-vollständig**, wenn:
- L_0 in PSPACE liegt und alle anderen Sprachen aus PSPACE weniger Platz verbrauchen.

BEISPIEL: Arbeitsband mit n Positionen.

Anzahl der Konfigurationen
= Anzahl der Bandpositionen
* Anzahl der Zustände
* Größe_Alphabet$^{\text{Anzahl_Positionen}}$ (an jeder Position kann ja was anderes stehen).

Du hast also
- alle möglichen Beschriftungen des Bandes,
- alle möglichen aktuellen Positionen
- alle möglichen aktuellen Zustände.

Dies ist logischweise durch die Längenbeschränkung des Bandes beschränkt.