Rheinische Friedrich-Wilhelms-Universität Bonn
Mathematisch-Naturwissenschaftliche Fakultät

Algorithmen und Berechnungskomplexität II

Skript SS 2016

Nach Aufzeichnungen von Rolf Klein und Elmar Langetepe

Bonn, Juni 2016

Inhaltsverzeichnis

Kapitel 1

Einführung

Im vergangenen Semester haben wir verschiedene klassische algorithmische Fragestellungen (Sortieren, Flussprobleme, Graphenprobleme, Bin-Packing, Knapsack, etc.) untersucht, Lösungspläne dafür entworfen und die Effizienz dieser Pläne untersucht. Die Lösungspläne beruhten dabei auf sehr unterschiedlichen Vorgehensweisen (Divide-and-Conquer, Inkrementelle Konstruktion, Greedy, Dynamische Programmierung, Sweep, etc.). Diese Paradigmen haben wir für verschiedene Fragestellungen zur Anwendung gebracht. Für die Analyse der Lösungen wurden dann die wesentlichen Berechnungsschritte asymptotisch abgeschätzt. Neben der formalen Beschreibung dieser Abschätzungen (O- und Ω-Notation) haben wir auch verschiedene Methoden der Analyse (Rekursionsgleichungen, Induktionsbeweise, strukturelle Eigenschaften, Rekursionstiefe, etc.) kennengelernt und angewendet. Außerdem haben wir gesehen, dass zur rechnergestützten Umsetzung der Lösungspläne eine geeignete, effiziente Repräsentation der Daten (Datenstrukturen) im Rechner notwendig ist. Ein wesentlicher Bestandteil der Vorlesung war die formale Analyse der vorgestellten Techniken und Strukturen. Dadurch konnten wir Gütegarantien angeben. Die Formulierung der Algorithmen wurde im Wesentlichen durch Pseudocode dargestellt, der sich im Prinzip in jeder höheren Programmiersprache umsetzen lässt.

In diesem Semester beschäftigen wir uns mit der theoretischen und praktischen *Berechenbarkeit* von (mathematischen) Problemen. Dazu gehen wir in Kapitel 2 der Frage nach, welche Probleme bzw. Funktionen wir mit Algorithmen und modernen Computern überhaupt lösen können. Eine mathematische Funktion nennen wir *intuitiv berechenbar*, falls es einen Algorithmus gibt, der zu jeder Eingabe aus dem Definitionsbereich eine Ausgabe produziert. Terminiert der Algorithmus nicht, so war die Eingabe nicht in der Definitionsmenge enthalten. Dabei verstehen wir unter einem Algorithmus ein Rechenverfahren mit den folgenden Eigenschaften:

1. Die Rechenvorschrift besteht aus einem endlichen Text; der Ablauf ist

eine Folge von (potenziell unendlich vielen) Rechenschritten.

2. Zu jedem Zeitpunkt ist eindeutig der nächste Rechenschritt bestimmt.

3. Jeder Rechenschritt hängt nur von der Eingabe und den bisher berechneten Zwischenergebnissen ab.

Intuitiv haben wir bereits eine Vorstellung, wann eine mathematische Funktionen berechenbar ist. In Kapitel 2 formalisieren wir diesen intuitiven Berechenbarkeitsbegriff. Wie wir sehen werden, gibt es tatsächlich auch Problemstellungen, die nicht berechenbar sind. Das heißt wir können sie mit modernen Computern nicht lösen. Im zweiten Teil dieser Vorlesung konzentrieren wir uns auf die *praktische Berechenbarkeit*. Obwohl ein Problem theoretisch berechenbar ist, kann es sein, dass die Berechnung des Ergebnisses viele Jahre dauern würde. Aus diesem Grund führen wir in Kapitel 3 eine Charakterisierung von Schwere-Klassen der Problemstellungen je nach Komplexität ein.

Kapitel 2

Theoretische Berechenbarkeit

2.1 Conways Spiel des Lebens

Zur Motivation betrachten wir das *Spiel des Lebens*, welches J. H. Conway 1970 entwarf. Das Spielfeld besteht aus einem Gitter quadratischer Zellen. Wir nehmen an, dass die Anzahl an Zellen in jede Richtung unbeschränkt ist. Jede der Zellen kann genau einen der Zustände *lebendig* bzw. *tot* annehmen. Das Spiel findet in Zeitschritten statt. Zu Beginn des Spiels, d.h. zum Zeitpunkt $t = 0$ wird eine erste *Generation* lebendiger Zellen auf dem Gitter platziert, die übrigen Zellen sind tot. Zu jedem Zeitpunkt $t \geq 0$ ist die Generation zum Zeitpunkt $t + 1$ durch die folgenden Spielregeln bestimmt:

1. Eine tote Zelle zum Zeitpunkt t lebt zum Zeitpunkt $t + 1$, falls genau drei (der acht direkten) Nachbarn zum Zeitpunkt t leben.
2. Eine lebende Zelle zum Zeitpunkt t überlebt, falls sie zum Zeitpunkt t zwei oder drei lebende Nachbarn hat.

Mit diesen Spielregeln können wir leicht einen Algorithmus angeben, der zu jedem Zeitpunkt die folgende Generation berechnet. Es kann jedoch keinen Algorithmus geben, der für zwei beliebige Generationen prüft, ob die zweite Generation zu irgend einem Zeitpunkt aus der ersten hervorgeht. Dies kann man auch formal beweisen. Offensichtlich haben wir hier eine Grenze der Berechenbarkeit erreicht. Solche Grenzen werden wir in Abschnitt 2.4 untersuchen.

Zunächst sehen wir uns an, was überhaupt *berechnet* werden kann. Intuitiv haben wir davon eine Vorstellung. Zur Formalisierung des Berechenbarkeitsbegriffs wurden einige Konzepte (bzw. Modelle) entwickelt. In dieser Vorle-

sung lernen wir zwei der Konzepte näher kennen. In Abschnitt 2.2 betrachten wir den Ansatz der μ-rekursiven Funktionen. Anschließend untersuchen wir die von Turing eingeführten Turingmaschinen. Beide Konzepte definieren den Begriff der Berechenbarkeit auf ganz verschiedene Art und Weise. Dennoch sind beide Konzepte gleich mächtig, was wir auch formal beweisen werden. Diese Tatsache nehmen wir als Indiz für die Gültigkeit der folgenden These, die die Fähigkeiten moderner Rechenmaschinen beschreibt.

Church-Turing-These *(Churchsche These) Die Klasse der durch Turingmaschinen (bzw. μ-rekursiven Funktionen) berechenbaren Funktionen umfasst alle intuitiv berechenbaren Funktionen.*

Da wir den Begriff *intuitiv berechenbarer* Funktionen nicht exakt formalisieren können, lässt sich diese These auch nicht beweisen. Es gibt jedoch eine Vielzahl weiterer Formalisierungen, die alle hinsichtlich ihrer Berechnungsstärke äquivalent sind.

2.2 μ-rekursive Funktionen

Wir konstruieren uns zunächst eine Klasse (bzw. Menge) von berechenbaren Funktionen. Die Idee besteht darin, einige wenige einfache **Grundfunktionen** zu definieren. Aus diesen lassen sich dann mittels einfacher **Operationen** (bzw. Schemata) neue berechenbare Funktionen gewinnen.

(A) Grundfunktionen (für $r, s \in \mathbb{N}_0$)

(i) **Konstante Funktionen**

$$\begin{aligned} c_s^r : \quad \mathbb{N}_0^r \quad &\to \mathbb{N}_0 \\ \mathfrak{x} = (x_1, x_2, \ldots, x_r) \quad &\mapsto s \end{aligned}$$

(ii) **Nachfolgefunktion**

$$\begin{aligned} N : \quad \mathbb{N}_0 \quad &\to \mathbb{N}_0 \\ x \quad &\mapsto x + 1 \end{aligned}$$

(iii) **Projektionen**

$$\begin{aligned} P_i^r : \quad \mathbb{N}_0^r \quad &\to \mathbb{N}_0 \\ \mathfrak{x} = (x_1, x_2, \ldots, x_r) \quad &\mapsto x_i \end{aligned}$$

(B) Operationen (für $r, s \in \mathbb{N}_0$)

(i) **Substitution** (simultanes Einsetzen) Für

$$\begin{aligned} g_1, \ldots, g_r : &\quad \mathbb{N}_0^m &\to \mathbb{N}_0 \\ g : &\quad \mathbb{N}_0^r &\to \mathbb{N}_0 \end{aligned}$$

definiere

$$\begin{aligned} h : &\quad \mathbb{N}_0^m &\to \mathbb{N}_0 \\ &\quad \mathfrak{x} &\mapsto g\left(g_1(\mathfrak{x}), \ldots, g_r(\mathfrak{x})\right) \end{aligned}$$

(ii) **Primitive Rekursion** Für

$$\begin{aligned} g : &\quad \mathbb{N}_0^r &\to \mathbb{N}_0 \\ f : &\quad \mathbb{N}_0^{r+2} &\to \mathbb{N}_0 \end{aligned}$$

definiere

$$\begin{aligned} h : &\quad \mathbb{N}_0^{r+1} &\to \mathbb{N}_0 \\ &\quad h(\mathfrak{x}, 0) &:= g(\mathfrak{x}) \\ &\quad h(\mathfrak{x}, y+1) &:= f(\mathfrak{x}, y, h(\mathfrak{x}, y)) \end{aligned}$$

Aus diesen Grundfunktionen und Operationen können wir nun die Klasse der primitiv rekursiven Funktionen wie folgt erzeugen.

Definition 1 (Primitiv rekursive Funktionen)**.** Die Klasse $\mathcal{P}$ der primitiv rekursiven Funktionen ist die kleinste Klasse von Funktionen, die

1. die Grundfunktionen enthält und
2. abgeschlossen ist unter den Operationen Substitution und primitive Rekursion.

Eine Funktion f heißt primitiv rekursiv genau dann wenn $f \in \mathcal{P}$ gilt.

Für eine primitiv rekursive Funktion können wir also eine Folge von Grundfunktionen und Operationen angeben, aus denen sie sich herleitet. Um zu beweisen, dass eine gegebene Funktion f tatsächlich primitiv ist ($f \in \mathcal{P}$), müssen wir für sie eine solche Herleitung finden. Wir illustrieren dies anhand folgender Beispiele.

Beispiel 1. 1. Addition $a\left(x, y\right) = x + y$

$$\begin{aligned} a\left(x, 0\right) &:= P_1^1\left(x\right) \\ x + y + 1 = a\left(x, y+1\right) &:= N\left(P_3^3\left(x, y, a\left(x, y\right)\right)\right) \end{aligned}$$

2. Multiplikation $m(x,y) = x \cdot y$

$$\begin{aligned} m(x,0) &:= c_0^1(x) \\ m(x,y+1) &:= a\left(P_3^3(x,y,m(x,y)), P_1^3(x,y,m(x,y))\right) \\ &= x \cdot y + x \end{aligned}$$

Folglich ist die Multiplikation primitiv rekursiv, da wir bereits gezeigt haben, dass die Addition primitiv rekursiv ist und $\mathcal{P}$ unter Substitution abgeschlossen ist.

3. Potenz $h(x,y) = x^y$

$$\begin{aligned} h(x,0) &:= c_1^1(x) \\ h(x,y+1) &:= m\left(P_3^3(x,y,h(x,y)), P_1^3(x,y,h(x,y))\right) \\ &= x^y \cdot x \end{aligned}$$

Hier verwenden wir, dass die Multiplikation primitiv rekursiv ist.

4. Vorgängerfunktion $V(y) = \begin{cases} 0 & , y = 0 \\ y-1 & , y > 0 \end{cases}$

$$\begin{aligned} V(0) &:= c_0^0 \\ V(y+1) &:= P_1^2(y, V(y)) \end{aligned}$$

5. Modifizierte Differenz $d(x,y) = \begin{cases} x-y & , x \geq y \\ 0 & , \text{sonst} \end{cases}$

$$\begin{aligned} d(x,0) &:= P_1^1(x) \\ d(x,y+1) &:= V\left(P_3^3(x,y,d(x,y))\right) \end{aligned}$$

Statt $d(x,y)$ werden wir im Folgenden auch $x \dot{-} y$ schreiben.

Um komplexere Funktionen beschreiben zu können, verwenden wir Prädikate. Ein $r-$stelliges *Prädikat* P über $\mathbb{N}_0$ ist eine Teilmenge von $\mathbb{N}_0^r$. Zu diesem Prädikat P gehört außerdem eine *charakteristische Funktion*

$$\begin{aligned} \chi_P : &\quad \mathbb{N}_0^r \to \{0,1\} \\ \chi_P(\mathfrak{x}) &= \begin{cases} 1 & , \mathfrak{x} \in P \\ 0 & , \mathfrak{x} \notin P. \end{cases} \end{aligned}$$

Statt $\mathfrak{x} \in P$ schreiben wir kurz $P(\mathfrak{x})$. Wir nennen das Prädikat P genau dann primitiv rekursiv, wenn die zugehörige charakteristische Funktion $\chi_P \in P$ primitiv rekursiv ist.

Lemma 1. *Mit P und Q sind auch die Prädikate*

$$\begin{aligned} P \wedge Q &:= P \cap Q \\ P \vee Q &:= P \cup Q \\ \neg P &:= \mathbb{N}_0^r \setminus P \end{aligned}$$

primitiv rekursiv.

Beweis. Es gilt

$$\begin{aligned} \chi_{P \wedge Q} &= \chi_P \cdot \chi_Q \\ \chi_{P \vee Q} &= \operatorname{sg}(\chi_P + \chi_Q) \\ \chi_{\neg P} &= 1 \dot{-} \chi_P = 1 - \chi_P \end{aligned}$$

wobei die Funktion $\operatorname{sg}(x) := \begin{cases} 1 & , x > 0 \\ 0 & , x = 0 \end{cases} \in \mathcal{P}$ ist. (Übungsaufgabe) □

Mit Hilfe von Prädikaten lässt sich bequem beweisen, dass die Definition von Funktionen mit einer endlichen Anzahl an Fallunterscheidungen nicht aus $\mathcal{P}$ herausführt. Jede so neu definierte Funktion ist also ebenfalls primitiv rekursiv, wie wir nun zeigen.

Theorem 1. *Seien $P_1, \dots P_k$ paarweise disjunkte r-stellige primitiv rekursive Prädikate, und seien $f_1, \dots, f_k : \mathbb{N}_0^r \to \mathbb{N}_0$ primitiv rekursive Funktionen. Dann ist auch folgende Funktion primitiv rekursiv:*

$$g(\mathfrak{x}) := \begin{cases} f_1(\mathfrak{x}) & , \textit{falls } P_1(\mathfrak{x}) \\ \vdots & \\ f_k(\mathfrak{x}) & , \textit{falls } P_k(\mathfrak{x}) \\ 0 & , \textit{sonst.} \end{cases}$$

Beweis. Wegen der Disjunktheit der Prädikate ist g wohldefiniert. Wir können g ausdrücken durch

$$\begin{aligned} g(\mathfrak{x}) &= \sum_{i=1}^{k} \chi_{P_i}(\mathfrak{x}) \cdot f_i(\mathfrak{x}) \\ &= \underbrace{a(\chi_{P_i}(\mathfrak{x}) \cdot f_1(\mathfrak{x}), a(\cdots))}_{k\text{–mal verschachtelt}} \end{aligned}$$

□

Nun stellt sich die Frage, ob wir mit der Klasse der primitiv rekursiven Funktionen bereits eine Beschreibung aller intuitiv berechenbaren Funktionen gefunden haben. Dass dem nicht so ist lässt sich mittels Diagonalisierung

beweisen. W. Ackermann gab sogar explizit eine mathematische Funktion an, die nicht primitiv rekursiv ist.

Da die Klasse der primitiv rekursiven Funktionen nicht alle intuitiv berechenbaren Funktionen abdeckt wollen wir jetzt eine entsprechende Erweiterung von $\mathcal{P}$ vornehmen. Wir führen einen Operator ein, der auf der Ganzheit der natürlichen Zahlen nach der kleinsten Nullstelle sucht.

Definition 2 ($\mu-$Operator)**.** Sei $f : \mathbb{N}_0^{r+1} \to \mathbb{N}_0$ eine Funktion. Dann wird $\mu f : \mathbb{N}_0^r \to \mathbb{N}_0$ definiert durch

$$\mu f\left(\mathfrak{x}\right) := \begin{cases} \text{das kleinste } y \text{ mit } f\left(\mathfrak{x}, y\right) = 0 \text{ und} & \\ f\left(\mathfrak{x}, 0\right), \ldots, f\left(\mathfrak{x}, y-1\right) \text{ ist definiert} & \text{, falls } y \text{ existiert} \\ \text{undefiniert} & \text{, sonst.} \end{cases}$$

Falls der zweite Fall nie eintritt, d.h. falls gilt $\forall \mathfrak{x} \exists y : f\left(\mathfrak{x}, y\right) = 0$, sagt man: μf entsteht aus f durch Anwendung des μ-Operators *im Normalfall*. Insbesondere ist $\mu f(\mathfrak{x})$ undefiniert, falls es kein y gibt mit $f(\mathfrak{x}, y) = 0$ oder $f(\mathfrak{x}, i)$ undefiniert ist für mindestens ein $0 \leq i \leq y - 1$. Wir erweitern nun die Klasse der primitiv rekursiven Funktionen, indem wir die Anwendung des μ Operators erlauben.

Definition 3 ($\mu-$rekursive Funktionen)**.** Die Klasse $\mathcal{F}_\mu^{tot}$ der $\mu-$rekursiven Funktionen ist die kleinste Klasse, die

1. die Grundfunktionen enthält,
2. abgeschlossen unter den Operationen Substitution und primitiver Rekursion ist und
3. abgeschlossen ist unter Verwendung des $\mu-$Operators im Normalfall.

Wir definieren $\mathcal{F}_\mu^{par}$ als die Klasse der partiellen $\mu-$rekursiven Funktionen analog, fordern jedoch die Abgeschlossenheit bzgl des μ-Operators generell.

Schauen wir uns zunächst ein paar Beispiele für die Leistungsfähigkeit des μ-Operators an:

Beispiel 2.
1. Sei $f(x, y) = x \dot{-} 13y$, dann ist $\mu f(x) = \lceil x/13 \rceil$ total.
2. Sei $g(x, y) = (x \dot{-} 13y) + (13y \dot{-} x)$. Dann ist $\mu g(x) = x/13$, falls x durch 13 teilbar ist, sonst undefiniert.
3. Es gilt folgende Folgerung: $f \in \mathcal{F}_\mu^{tot}$ bijektiv $\Rightarrow f^{-1} \in \mathcal{F}_\mu$.
   Zum Beweis setze $h\left(v, w\right) := \left(v \dot{-} f\left(w\right)\right) + \left(f\left(w\right) \dot{-} v\right)$. Dann ist h

μ-rekursiv und und $h(v,w) = 0$ genau dann, wenn $f(w) = v$. Setzen wir nun $g(v) := \mu h(v)$, dann folgt

$$\begin{aligned} g(v) &= \text{kleinstes } w \text{ mit } f(w) = v \\ &\underset{f \text{ bij.}}{=} \text{das } w \text{ mit } f(w) = v. \end{aligned}$$

Damit haben wir jedoch bereits f^{-1} gefunden, denn $g = f^{-1}$. Man könnte auch sagen, dass der μ-Operator nach $f^{-1}(v)$ *sucht*.

Nun müssen wir zeigen, dass wir mit Hilfe des μ-Operators tatsächlich die Klasse der primitiv rekursiven Funktionen erweitert haben.

Theorem 2.

$$\mathcal{P} \subsetneq \mathcal{F}_\mu^{tot} \subsetneq \mathcal{F}_\mu^{par}$$

Beweis. Die zweite Inklusion ist offensichtlich. Es ist jedoch zu beachten, dass sich nicht jedes $f \in \mathcal{F}_\mu^{par}$ zu einem $\hat{f} \in \mathcal{F}_\mu^{tot}$ fortsetzen lässt!

Die erste Inklusion kann mittels Diagonalisierung bewiesen werden. Der Beweis findet sich beispielsweise in [1]. Alternativ können wir auch eine intuitiv berechenbare Funktion konstruieren, die nicht primitiv rekursiv ist. Die Ackermannfunktion erfüllt genau diese Anforderungen, wie wir im folgenden zeigen. □

Wir konstruieren nun eine μ-rekursive Funktion A, die nicht in $\mathcal{P}$ liegt. Idee: extrapoliere $N, +, \cdot, x^y, \ldots$

$$\begin{aligned} f_1(x,y) &= x + y \\ f_2(x,y) &= x \cdot y \\ f_3(x,y) &= x^y \\ f_4(x,y) &= \underbrace{x^{x^{\cdot^{\cdot^{\cdot^{x}}}}}}_{y\text{–mal}} \\ &\vdots \\ f_{n+1}(x, y+1) &= f_n(x, f_{n+1}(x,y)) \end{aligned}$$

Wir wenden also bei jedem Folgenglied die Operation des vorigen Folgenglieds y-mal auf x an. Das schnelle Wachstum der Funktion hängt nicht so sehr von x ab (schon für $x = 2$ extrem). Deshalb eine kompaktere Definition:

$$\begin{aligned} f_0(y) &:= y + 1 \\ f_{n+1}(0) &:= f_n(1) \\ f_{n+1}(y+1) &:= f_n(f_{n+1}(y)) \end{aligned}$$

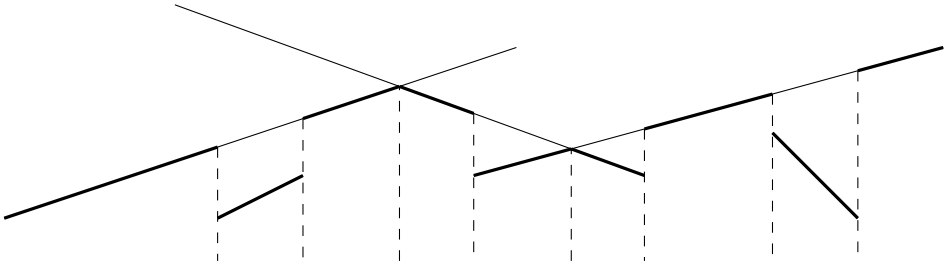

Abbildung 2.1: Zur Komplexität der unteren Kontur von Liniensegmenten.

Per Induktion lässt sich zeigen, dass jede Funktion f_n in $\mathcal{P}$ liegt. Es handelt sich um eine primitive Rekursion. Ab jetzt betrachten wir den Index n als Variable angesehen und erhalten die *Ackermannfunktion*

$$A(x,y) := f_x(y).$$

Also

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

Diese Funktion ist offensichtlich intuitiv berechenbar. Man rechnet leicht nach, dass

$$\begin{aligned} A(0,y) &= y+1 \\ A(1,y) &= y+2 \\ A(2,y) &= 2y+3 \\ A(3,y) &= 2^y \cdot 8 - 3 \\ A(4,y) &= \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{y+3\text{-mal}} - 3 \end{aligned}$$

Außerdem kommt diese *künstlich* erzeugte Funktion (bzw. ihr Inverses) tatsächlich *in der Natur* vor! Für

$$\alpha(n) := \mu x : A(x,x) \geq n$$

kann die untere Kontur (siehe Abbildung 2.1) von n Liniensegmenten aus $\Theta(n\alpha(n))$ vielen Stücken bestehen! Diese Tatsache können wir in der Algorithmischen Geometrie zur Laufzeitanalyse verwenden.

Wir werden nun im folgenden zeigen, dass die Ackermannfunktion nicht primitiv rekursiv, aber μ-rekursiv ist. Es gilt also tatsächlich $\mathcal{P} \subsetneq \mathcal{F}_\mu^{tot}$. Dazu zeigen wir:

1. A ist nicht primitiv rekursiv, d.h. $A \notin \mathcal{P}$. (Denn A wächst schneller als jede Funktion in $\mathcal{P}$.)
2. A ist μ-rekursiv, d.h. $A \in \mathcal{F}_\mu^{tot}$.

Der Beweis, dass A nicht primitiv rekursiv ist stützt sich auf folgendes Lemma, welches wir hier nicht beweisen werden.

Lemma 2. *Für jede primitiv rekursive Funktion $f \in \mathcal{P}$ gibt es ein k, sodass*

$$f(\mathfrak{x}) \leq A(k, \underbrace{\sum \mathfrak{x}}_{x_1+x_2+\cdots+x_r})$$

für alle $\mathfrak{x}$.

Mit Hilfe von Lemma 2 können wir folgenden Widerspruchsbeweis führen. Angenommen A wäre primitiv rekursiv, dann wäre auch $f(x) := A(x,x)+1$ primitiv rekursiv. Damit folgt aus dem Lemma 2, dass stets ein k existiert, sodass

$$f(x) = A(x,x) + 1 \leq A(k,x).$$

Mittels Diagonalisierung, also der Wahl $x := k$ folgt

$$A(k,k) + 1 \leq A(k,k),$$

was ein Widerspruch ist. Die Ackermannfunktion kann also nicht primitiv rekursiv sein. Intuitiv ist klar, dass die Ackermannfunktion μ-rekursiv ist. Auf den Beweis verzichten wir jedoch an dieser Stelle und verweisen auf Abschnitt 2.3.

Exkurs Schließlich betrachten wir noch folgende nützliche, schwächere Version des μ-Operators. Sie heißt *beschränkter μ-Operator* und wir werden sie später benötigen. Im Gegensatz zum μ-Operator führt sie nicht über $\mathcal{P}$ hinaus.

Theorem 3. *Mit $f : \mathbb{N}_0^{r+1} \to \mathbb{N}_0$ ist auch der beschränkte μ-Operator*

$$\begin{aligned} \mu_b f : \quad \mathbb{N}_0^{r+1} &\to \mathbb{N}_0 \\ (\mathfrak{x}, y) &\mapsto \begin{cases} \text{das kleinste } x \leq y & \\ \text{mit } f(\mathfrak{x}, x) = 0, & \text{falls ein solches existiert} \\ 0, & \text{sonst} \end{cases} \end{aligned}$$

primitiv rekursiv.

Beweis. Der Suchraum des Operators ist beschränkt und wir können primitive Rekursion anwenden:

$$\begin{aligned} \mu_b f(\mathfrak{x}, 0) &= 0 \\ \mu_b f(\mathfrak{x}, y+1) &= \begin{cases} y+1, & \text{falls } f(\mathfrak{x}, y+1) = 0 \text{ und} \\ & \mu_b f(\mathfrak{x}, y) = 0 \text{ und } f(\mathfrak{x}, 0) > 0 \\ \mu_b f(\mathfrak{x}, y), & \text{sonst.} \end{cases} \end{aligned}$$

Da wir eine endliche Fallunterscheidung verwendet haben und die zweistelligen Prädikate $\vee, =, >$ (Übungsaufgabe) und auch f primitiv rekursiv sind, folgt nach Theorem 1, dass auch $\mu_b f$ primitiv rekursiv ist. □

Rekapitulation

Zur Beschreibung intuitiv berechenbarer Funktionen haben wir zwei Klassen von Funktionen definiert:

$\mathcal{P}$	primitiv rekursive Funktionen	Konstante c_i^r, Nachfolger N, Projektionen P_i^r, abgeschlossen gegen Substitution und Primitive Rekursion
$\mathcal{F}_\mu$	$\mu-$rekursive Funktionen	$\mathcal{P}$ + $\mu-$Operator im Normalfall

Zum Nachweis, dass $\mathcal{P} \subsetneq \mathcal{F}_\mu$ haben wir gezeigt der $\mu-$Operator nicht entbehrlich ist. Dafür haben wir die Ackermannfunktion A konstruiert und gezeigt, dass diese nicht in $\mathcal{P}$ liegt. Dass A tatsächlich μ-rekursiv ist werden wir erst später exakt beweisen.

Die allgemeine Frage, der wir nachgehen ist, ob alle intuitiv berechenbaren Funktionen in $\mathcal{F}_\mu^{tot}$ liegen. Mit dem Ansatz der μ-rekursiven Funktionen haben wir eine *Stütze* für die Church-Turing-These gefunden. Diese lässt sich zwar nicht beweisen, jedoch durch verschiedene Ansätze erhärten. Im folgenden Kapitel betrachten wir einen zweiten Ansatz.

2.3 Turingmaschinen

In diesem Abschnitt führen wir als zweites Berechnungsmodell Turingmaschinen ein. Wir werden außerdem zeigen, dass beide Modell äquivalent sind. Das Modell wurde 1936, also noch vor der Entwicklung erster Computer, von Alan M. Turing beschrieben. Seine Abstraktion eines Computers wird ihm zu Ehren *Turingmaschine* genannt.

Ganz allgemein nehmen wir an, dass der Speicher aus k Halbbändern besteht. Wie wir später in Lemma 7 zeigen, reicht streng genommen ein einziges solches Halbband aus. In einigen Beweisen werden wir jedoch auch sehen, dass Turingmaschinen mit beliebig vielen Halbbändern deutlich einfacher zu beschreiben sind. Jedes Halbband besteht aus einer Folge von *Bandquadraten* und ist nach rechts unbeschränkt groß. In jedem Bandquadrat steht genau ein Zeichen aus einem endlichen Bandalphabet Σ. Das Bandalphabet enthält die *Sonderzeichen* $\$, \#$, und $\sqcup$. Das Zeichen $\$$ markiert den Beginn des Bandes, $\sqcup$ eine leere Zelle und $\#$ wird als Trennzeichen verwendet. Die Turingmaschine verwendet eine sogenannte *endliche Kontrolle*. Diese besteht aus je einem Lese-/Schreibkopf für jedes der Halbbänder. Dieser zeigt stets auf ein Bandquadrat des Bands und ermöglicht das Auslesen und Verändern des jeweiligen Zeichens. Der Kopf kann schrittweise über das jeweilige Band bewegt werden. Diese Bewegung kodieren wir durch die Zahlen -1, 1 und 0. Dabei bedeutet -1 eine Bewegung des Kopfs um ein Bandquadrat nach links, $+1$ nach rechts und 0 keine Bewegung. Die Kontrolle handelt deterministisch und die Maschine befindet sich zu jedem Zeitpunkt in einem Zustand q aus einer endlichen Zustandsmenge Q. Mittels einer (ggf. partiellen) Zustandsübergangsfunktion $\delta : Q \times \Sigma^k \to Q \times \Sigma^k \times \{-1, 0, +1\}^k$ regelt die endliche Kontrolle die Arbeit der Turingmaschine. Diese Funktion ordnet einem aktuellen Zustand q, unter Berücksichtigung der von allen Köpfen gelesenen Zeichen $s_1, \dots s_k$, einen neuen Zustand q' zu:

$$(q, s_1, \dots s_k) \mapsto (q', s'_1, \dots, s'_k, m_1, \dots, m_k).$$

Dabei legt sie auch fest, welche Zeichen des Bandalphabets in die ausgelesenen Bandquadrate geschrieben werden und welche Bewegung jeder Kopf nach dem Schreiben ausführt. Vorstellen können wir uns eine Turingmaschine wie in Abschnitt 2.3 dargestellt.

Turingmaschinen können wir auch als Erweiterung von endlichen Automaten in drei Kriterien interpretieren: Bewegung, Schreiben, Speicherplatz. Der Kopf darf sich sowohl nach rechts als auch nach links bewegen. Die Turingmaschine darf ein Eingabesymbol an der Position des Kopfes auch verändern. Außerdem ist der Arbeitsbereich (bzw. Speicher) nach rechts über die Eingabe hinaus unbeschränkt. Damit endet eine Berechnung nicht mehr nach dem Lesen des letzten Eingabezeichens sondern in speziellen *Endzuständen*.

Formal wird die Maschine wie folgt definiert.

Definition 4. Eine (deterministische) k-Band Turingmaschine (DTM) M ist ein 5-Tupel $M = (\Sigma, Q, \delta, q_0, F)$, wobei

1. Σ ein endliches Bandalphabet mit $\{0, 1, \#, \$, \sqcup\} \subseteq \Sigma$,
2. Q eine endliche Zustandsmenge mit $Q \cap \Sigma = \emptyset$,

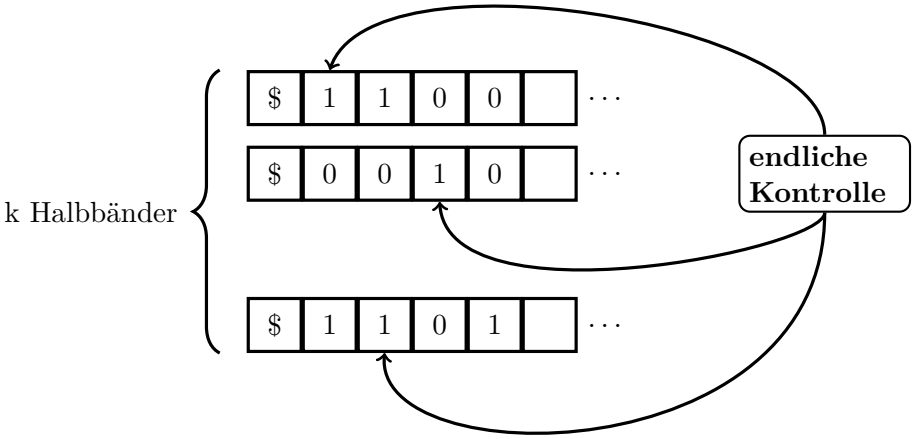

Abbildung 2.2: Darstellung einer Turingmaschine.

3. $q_0 \in Q$ der Startzustand,
4. $\delta : Q \times \Sigma^k \to Q \times \Sigma^k \times \{-1, 1, 0\}^k$ die Zustandsübergangsfunktion und
5. $F := \{q \in Q \mid \forall \mathfrak{x} \in \Sigma^k : \delta(q, \mathfrak{x}) \text{ ist undefiniert}\}$ die Menge der Endzustände

sind.

Die Maschine arbeitet schrittweise wie folgt. Das Zeichen $a \in \Sigma$ unter dem Lese-/Schreibkopf wird gelesen und wenn sich die Maschine im Zustand $q \in Q$ befindet, wird $\delta(q, a) = (q', b, d) \in Q \times \Sigma \times \{-1, 1, 0\}$ ausgewertet:

- Die Maschine schreibt $b \in \Sigma$ auf die aktuelle Zelle.
- Die Maschine bewegt den Lese-/Schreibkopf nach links ($d = -1$), nach rechts ($d = 1$) oder verbleibt auf der aktuellen Zelle ($d = 0$).
- Die Maschine wechselt in den Zustand q'.

Weiterhin legen wir zunächst folgende **Regeln** fest. Das Zeichen \$ darf nie überschrieben werden. Wenn das Zeichen \$ gelesen wird, dann darf die Maschine im nächsten Schritt den Lese-/Schreibkopf nicht nach links bewegen.

Der Kopf bewegt sich gemäß der oben angegebenen Aktion. Ein einzelner solcher Schritt wird als *Rechenschritt* bezeichnet. Die Anzahl der Rechenschritte, bis die Maschine in einem Endzustand landet, wird als *Laufzeit* bezeichnet. Die Anzahl der insgesamt aktiv verwendeten Zellen ergibt den

benötigten *Speicherplatz*. Die Maschine *terminiert*, wenn ein Endzustand erreicht wird. Zu Beginn steht jeder Lese-/Schreibkopf auf dem ersten Zeichen hinter dem Bandanfang \$. Die Eingabe der Länge n steht in den ersten n Bandquadraten des ersten Bandes. Alle übrigen Bandquadrate aller Bänder, bis auf die jeweils ersten, welche \$ enthalten, sind mit $\sqcup$ beschrieben.

Wir möchten nun spezifizieren, was es bedeutet, dass eine Turingmaschine M eine Funktion $f : \mathbb{N}_0^r \to \mathbb{N}_0$ berechnen kann. Dafür stellen wir die Eingaben mit der Abbildung

$$\begin{aligned} \text{bin} : \qquad \mathbb{N}_0 &\to \{0,1\}^+ = \bigcup_{n\geq 1} \{0,1\}^n \\ n = \sum_{i=1}^{s} w_i \cdot 2^{s-i} &\mapsto w_1 w_2 \dots w_s \end{aligned}$$

binär dar, wobei $w_i \in \{0, 1\}$ gilt.

Definition 5 (turingberechenbar)**.** Sei $M = (Q, \Sigma, \delta, q_0, F)$ eine k-Band Turingmaschine und $f : \mathbb{N}_0^r \to \mathbb{N}_0$ eine partielle Funktion. Wir sagen M *berechnet* f, falls für alle $\mathfrak{x} = (x_1, \dots, x_k) \in \mathbb{N}_0^k$ gilt:

M, angesetzt auf die Eingabe $\text{bin}(x_1)\#\,\text{bin}(x_2)\#\dots\#\,\text{bin}(x_k)$ stoppt genau dann nach endlich vielen Schritten, wenn $f(\mathfrak{x})$ definiert ist.

Da nun klar ist, was es bedeutet, dass eine (partielle) Funktion von einer deterministischen Turingmaschine berechnet wird, führen wir zwei Klassen zur Beschreibung turingberechenbarer Funktionen ein.

Definition 6 (turingberechenbare Funktionen)**.** Die Klasse der totalen turingberechenbaren Funktionen ist

$$\mathcal{F}_{\text{TM}}^{tot} := \{f : \mathbb{N}_0^r \to \mathbb{N}_0 \mid r \geq 0 \wedge \exists \text{ DTM M, die f berechnet}\}.$$

Die Klasse der partiellen turingberechenbaren Funktionen ist

$$\mathcal{F}_{\text{TM}}^{par} := \{f : \mathbb{D}_f \to \mathbb{N}_0 \mid r \geq 0 \wedge \exists \text{ DTM M, die f berechnet}\},$$

wobei $\mathbb{D}_f \subseteq \mathbb{N}_0^r$ den Definitionsbereich der Funktion f bezeichne.

Beispiel 3. Konstruktion einer 2-Band Turingmaschine M zur Berechnung der Nachfolgefunktion N. Dabei verwenden wir folgendes Programm:

- Kopiere Band 1 auf Band 2; (kurz Band 2 := Band 1)
- Lösche Band 1;
- Addiere binär 1 zum Inhalt von Band 2 von rechts nach links. Speichere Übertrag in Zuständen q_2 und q_3.
- Falls am Ende q_2, schreibe 1 auf Band 1 und bewege Kopf nach rechts;

- Hänge Inhalt von Band 2 an Band 1;
- Lösche Band 2;
- Köpfe nach vorne.

Zur konkreten Beschreibung der Übergangsfunktion δ verwenden wir eine etwas andere Tabellenschreibweise.

q	z_1	z_2	q'	z_1'	z_2'	m_1	m_2
q_0	$0/1$	$\sqcup$	q_0	$\sqcup$	$0/1$	1	1
q_0	$\sqcup$	$\sqcup$	q_1	$\sqcup$	$\sqcup$	-1	-1

Band 2 := Band 1;
Band 1 löschen;

q	z_1	z_2	q'	z_1'	z_2'	m_1	m_2
q_1	$\sqcup$	$0/1$	q_1	$\sqcup$	$0/1$	-1	0
q_1	\$	$0/1$	q_2	\$	$0/1$	1	0

Kopf 1 nach links;
Kopf 2 bleibt rechts;

q	z_1	z_2	q'	z_1'	z_2'	m_1	m_2
q_2	$\sqcup$	1	q_2	$\sqcup$	0	0	-1
q_2	$\sqcup$	0	q_3	$\sqcup$	1	0	-1
q_3	$\sqcup$	$0/1$	q_3	$\sqcup$	$0/1$	0	-1
q_3	$\sqcup$	\$	q_4	$\sqcup$	\$	0	1

Addiere 1 zu Band 2 von rechts nach links;
q_2 Übertrag,
q_3 kein Übertrag.

q	z_1	z_2	q'	z_1'	z_2'	m_1	m_2
q_2	$\sqcup$	\$	q_4	1	\$	1	1

Falls am Ende q_2, schreibe 1 auf Band 1;
Kopf 1 nach rechts;

q	z_1	z_2	q'	z_1'	z_2'	m_1	m_2
q_4	$\sqcup$	$0/1$	q_4	$0/1$	$\sqcup$	1	1
q_4	$\sqcup$	$\sqcup$	q_5	$\sqcup$	$\sqcup$	-1	-1

Hänge Band 2 an Band 1;
Lösche Band 2.

q	z_1	z_2	q'	z_1'	z_2'	m_1	m_2
q_5	$0/1$	$\sqcup$	q_5	$0/1$	$\sqcup$	-1	0
q_5	\$	$\sqcup$	q_6	\$	$\sqcup$	1	0
q_6	$0/1$	$\sqcup$	q_6	$0/1$	$\sqcup$	0	-1
q_6	$0/1$	\$	q_7	$0/1$	\$	0	1

Kopf 1 nach links;
Kopf 2 nach links.

$M = (\{\$, 0, 1, \sqcup\}, \{q_0, \ldots, q_7\}, \delta, q_0, \{q_7\})$ ist die gesuchte Turingmaschine.

2.3.1 Äquivalenz der Berechnungsmodelle

Auf den ersten Blick scheint der Ansatz der Turingmaschinen zur Beschreibung des Berechenbarkeitsbegriffs ganz anders als der Ansatz der μ-rekursiven Funktionen zu sein. Wie wir im folgenden Abschnitt zeigen, sind die Klasse der μ-rekursive Funktionen und die Klasse der turingberechenbaren Funktionen jedoch äquivalent. Dass beiden Modelle gleich mächtig sind, nehmen wir als weiteres Indiz für die Gültigkeit der Church-Turing-These.

Wir zeigen die Äquivalenz in zwei Schritten und beginnen mit dem Beweis, dass jede μ-rekursive Funktion auch turingberechenbar ist.

Theorem 4. *Jede μ-rekursive Funktion ist turingberechenbar, d.h. es gilt*

$$\mathcal{F}_\mu^{par} \subseteq \mathcal{F}_{\mathrm{TM}}^{par} \text{ und } \mathcal{F}_\mu^{tot} \subseteq \mathcal{F}_{\mathrm{TM}}^{tot}.$$

Beweis. Zunächst überlegen wir uns in Lemma 3, dass alle Grundfunktionen turingberechenbar sind. Anschließend zeigen wir mit je einem Lemma, dass wir auch die Operationen Substitution, primitive Rekursion und die Anwendung des μ-Operators durch Turingmaschinen realisieren können. □

Lemma 3. *Die konstanten Funktionen c_i^r, die Projektionen P_i^r und die Nachfolgefunktion N sind turingberechenbar.*

Dass die Grundfunktionen von $\mathcal{P}$ turingberechenbar sind, kann durch Angabe der jeweiligen Turingmaschine leicht gezeigt werden. Für die Nachfolgefunktion haben wir diese bereits in Beispiel 3 konstruiert. Die Konstruktion von Turingmaschinen zur Berechnung der anderen Funktionen ist eine Übungsaufgabe.

Etwas aufwändiger ist die Konstruktion von Turingmaschinen, welche die Operationen Substitution, primitive Rekursion und die Anwendung des μ-Operators simulieren. Hervorzuheben ist, dass jede dieser Turingmaschinen mit einer sehr kleinen (d.h. zwei, drei oder vier) Anzahl an zusätzlichen Halbbändern auskommt.

Lemma 4 (Substitution)**.** *Sei $h : \mathbb{N}_0^m \to \mathbb{N}_0, \mathfrak{x} \mapsto f(g_1(\mathfrak{x}), \ldots, g_r(\mathfrak{x}))$. Sind $f, g_1, \ldots g_r$ durch die k-Band Turing-Maschinen $F, G_1, \ldots G_r$ berechenbar, dann lässt sich h durch eine $(k+2)$-Band Turingmaschine H berechnen.*

Beweis. Für den Beweis beschreiben wird schematisch das Programm der Turingmaschine H. Zunächst beschreiben wir zu jeder Turing-Maschine G_i eine Maschine G_i':

G'_i: Band 3 := Band 1;
lasse G_i auf den Bändern $3, 4, \ldots k+2$ laufen;
für $i = 1$: Band 2 := Band 3;
für $2 \leq i \leq r$: Band 2 := Band 2# Band 3;
lösche Band 3;
(alle Köpfe nach vorne;)

Damit können wir nun die Turingmaschine H beschreiben:

H: $G'_1; G'_2; \ldots; G'_r$;
lösche Band 1; (auf Band 2 ist jetzt $g_1(\mathfrak{x})\#g_2(\mathfrak{x})\#\ldots\#g_r(\mathfrak{x})$)
lasse F auf den Bändern $2, 3, \ldots, k+1$ laufen;
Band 1:= Band 2;
lösche Band 2;
(alle Köpfe nach vorne;)

Dass zwei zusätzliche Bänder zur Verfügung stehen, haben wir also folgendermaßen genutzt. Auf einem Band wird zunächst die ursprüngliche Eingabe während der Simulation der Turingmaschinen G'_i unverändert gespeichert. Auf dem zweiten Band wird derweil die Ausgabe aller G_i's zusammengesetzt. Dies stellt dann wiederum die Eingabe für die Turingmaschine F dar. Neben diesen beiden Bändern sind natürlich k-Halbbänder nötig gewesen, um F und die G'_i's auszuführen. Mit zwei zusätzlichen Bändern kommen wir also bequem aus. □

Lemma 5 (primitive Rekursion)**.** *Sei* $h : \mathbb{N}_0^{r+1} \to \mathbb{N}_0$ *mit*

$$\begin{aligned} (0, \mathfrak{x}) &\mapsto g(\mathfrak{x}), \\ (n+1, \mathfrak{x}) &\mapsto f(n, h(n, \mathfrak{x}), \mathfrak{x}), \end{aligned}$$

und seinen G, F *zwei* k*-Band Turingmaschinen zu Berechnung von* g *und* f*. Dann gibt es eine* $(k+4)$*-Band Turingmaschine* H *zur Berechnung von* h*.*

Beweis. Wieder beweisen wir die Aussage durch Beschreibung des Programms der Turingmaschine H. Diesmal verwenden wir zwei der vier zusätzlichen Bänder (Band 1 und Band 4) als Zähler. Auf Band 1 zählen wir die Anzahl der noch durchzuführenden Berechnungen von f, während auf Band 4 die Anzahl der bereits durchgeführten Berechnungen von f steht. Band 2 dient zur Speicherung der Eingabe $\mathfrak{x}$. Auf Band 3 speichern wir schließlich das bisher berechnete (Teil-)Ergebnis, welches *bottom-up* berechnet wird:

H: kopiere den hinter dem ersten # stehenden Inhalt von Band 1 nach Band 2;
lösche diesen Inhalt auf Band 1;
Band 3 := Band 2;
lasse G auf den Bändern $3, 4 \ldots, k+2$ laufen;
Band 4 := 0;
WHILE Band 1 $\neq$ 0
DO
Band 5 := Band 4 # Band 3 # Band 2;
lasse F auf den Bändern $5, 6, \ldots, k+4$ laufen;
Band 3 := Band 5;
lösche Band 5;
Band 1 := Band 1 - 1;
Band 4 := Band 4 + 1;
OD
Band 1 := Band 3;
lösche Bänder 2,3,4;
(alle Köpfe nach vorne;)

Bemerkenswert ist die Tatsache, dass die **while**-Schleife im Programm von H auch durch eine Zählschleife ersetzen werden kann. Dies bedeutet, dass primitiv rekursive Funktionen generell mit Zählschleifen auskommen. (Das kann bspw. im Beweis, dass die Klasse der sogenannten $LOOP$-Programme und $\mathcal{P}$ äquivalent sind, verwendet werden.) □

Lemma 6 ($\mu-$Operator)**.** *Sei* $h = \mu f : \mathbb{N}_0^r \to \mathbb{N}_0$ *mit*

$$\mathfrak{x} \mapsto \begin{cases} \textit{das kleinste } n \textit{ mit } f(n, \mathfrak{x}) = 0 \textit{ und} & \\ f(0, \mathfrak{x}), \ldots, f(n-1, \mathfrak{x}) \textit{ ist definiert} & , \textit{falls } n \textit{ existiert} \\ \textit{undefiniert} & , \textit{sonst.} \end{cases}$$

Sei F *eine* k*-Band Turingmaschine zur Berechnung von* f*. Dann existiert eine* $(k+3)$*-Band Turingmaschine* H *zur Berechnung von* h*.*

Beweis. Zum Beweis geben wir wieder das Programm der Turingmaschine H an und verwenden die zusätzlichen Bänder wie folgt. Auf Band 2 wird die Eingabe $\mathfrak{x}$ dauerhaft gespeichert. Band 3 zählt beginnend bei 0 bis n, falls ein solches existiert. Auf Band 1 speichern wir stets das Ergebnis der Simulation von F und prüfen in einer while-Schleife, ob es bereits 0 wurde.

```
H:  Band 2 := Band 1;
    Band 1 := 1;
    Band 3 := 0;
    WHILE Band 1 ≠ 0
     DO
          Band 4 := Band 3 # Band 2;
          lasse F auf den Bändern 4, 5, ..., k + 3 laufen;
          Band 1 := Band 4; (Ergebnis)
          lösche Band 4;
          Band 3 := Band 3 + 1;
     OD
    Band 1 := Band 3 - 1;
    lösche Band 3;
    (alle Köpfe nach vorne;)
```

Im Beweis wird deutlich, dass die while-Schleife nicht einfach durch eine Zählschleife ersetzt werden kann. Tatsächlich entspricht die Anwendung des μ-Operators der (potentiell unendlich langen) Durchführung einer while-Schleife. (Tatsächlich lässt sich beweisen, dass die Klasse der sogenannten *WHILE*-Programme und $\mathcal{F}_{\mu}^{tot}$ äquivalent sind. Später werden wir mit Korollar 1 sogar zeigen, dass jede beliebige Turingmaschine mit einer einzigen while-Schleife auskommt.) □

Wir haben nun gezeigt, dass jede μ-rekursive Funktion auch turingberechenbar ist. Dass umgekehrt auch die Rechengänge einer Turingmaschine durch μ-rekursive Funktionen simuliert werden können ist schwieriger zu zeigen. Zur Vorbereitung und Vereinfachung zeigen wir zunächst, dass jede $k-$Band Turingmaschine streng genommen mit nur einem einzigen Band auskommt.

Lemma 7. *Sei $f : \mathbb{N}_0^r \to \mathbb{N}_0 \in \mathcal{F}_{\mathrm{TM}}^{par}$ auf einer k-Band Turingmaschine berechenbar. Dann lässt sich f auch auf einer 1-Band Turingmaschine $\tilde{F}$ berechnen.*

Beweis. Um die Arbeit von F mit nur einem Band zu simulieren teilen wir das Halbband von $\tilde{F}$ in $2k$ "Spuren" ein, wie Abbildung 2.3 verdeutlicht. Je zwei Spuren von $\tilde{F}$ dienen zur Simulation eines Bandes von F. Eine davon speichert die Position des Kopfes, die andere den Bandinhalt des entsprechenden Bands von F. $\tilde{F}$ simuliert nun die Rechengänge von F wie folgt:

- Suche nach den k Feldern, in denen $\downarrow$ vorkommt und merke die zugehörigen Bandinhalte im Zustand.
- Verwende Übergangsfunktion δ von F, um den neuen Zustand, die neuen Bandsymbole und die Bewegungen der k Köpfe zu berechnen.

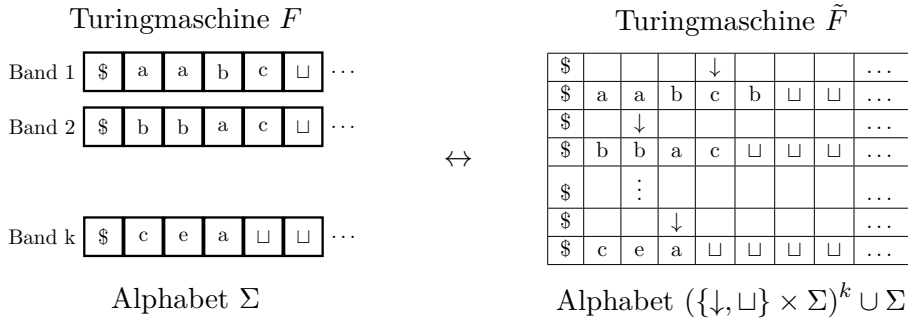

Abbildung 2.3: Transformation der k-Band DTM F zu 1-Band DTM $\tilde{F}$.

- Besuche nochmals alle Felder mit $\downarrow$ und ändere die Bandinschrift und die Kopfmarkierung entsprechend.

Sobald die Simulation der Berechnungen von F auf den k Bändern beendet ist, steht die Ausgabe in der zweiten Spur. Deshalb muss abschließend der Inhalt für jedes Feld rechts von \$ durch den Inhalt der zweiten Spur ersetzt werden. Dann kann $\tilde{F}$ den Kopf nach vorne bewegen und in einen Endzustand wechseln. □

Nun stellt sich die Frage, wie viel Mehraufwand die 1-Band Turingmaschine für diese Simulation betreiben muss. Erstaunlicherweise ist dieser Mehraufwand beschränkt, wie folgendes Lemma zeigt.

Lemma 8. *Die Anzahl der Schritte von $\tilde{F}$ ist quadratisch in der Anzahl der Schritte von F.*

Beweis. Um den i-ten Schritt von F zu simulieren, muss $\tilde{F}$ bis zur Position des am weitesten rechts stehenden Kopfes von F vorlaufen. Da F anfangs ganz links startet, können die Köpfe im i-ten Schritt höchstens i Felder nach rechts gewandert sein. □

Wir haben immer noch nicht die Äquivalenz der Klasse der μ-rekursiven Funktionen und der turingberechenbaren Funktionen gezeigt. Dazu fehlt uns noch folgende Beziehung.

Theorem 5. *Jede turingberechenbare Funktion ist μ-rekursiv, d.h. es gilt*

$$\mathcal{F}_{\mathrm{TM}}^{par} \subseteq \mathcal{F}_{\mu}^{par} \text{ bzw. } \mathcal{F}_{\mathrm{TM}}^{tot} \subseteq \mathcal{F}_{\mu}^{tot}.$$

Bei dem Beweis des Theorems wissen wir bereits wegen Lemma 7, dass jede turingberechenbare Funktion durch eine 1-Band Turingmaschine berechnet wird. Die Konfiguration einer 1-Band Turingmaschinen können wir auch einfach als Zeichenkette

$$K = \$a_1 \dots a_{k-1}\ q\ a_k \dots a_t$$

schreiben. Dabei entsprechen die a_i der Bandinschrift, q dem Zustand und a_k die Position des L/S-Kopfes. Wir nennen K genau dann eine *Endkonfiguration*, wenn q ein akzeptierender Zustand ist, d.h. wenn $q \in F$. Andernfalls gibt es für K eine eindeutige *Nachfolgekonfiguration*, die wir mit $\Delta(K)$ bezeichnen. Ist beispielsweise $\delta(q, a_k) = (q', a'_k, +1)$, so können wir die Nachfolgekonfiguration durch

$$\begin{array}{ccc} K & \overset{\Delta}{\longmapsto} & K' \\ \| & & \| \\ \$a_1 \dots a_{k-1}\ q\ a_k \dots a_t & & \$a_1 \dots a_{k-1} a'_k\ q'\ a_{k+1} \dots a_t \end{array}$$

leicht bestimmen. Der Beweis von Theorem 5 stellt uns jedoch vor eine Schwierigkeit, denn Turingmaschinen operieren auf Zeichenketten, während μ-rekursive Funktionen auf Zahlen operieren. Wir gehen daher in zwei Schritten vor:

(i) Konfigurationen (=Zeichenketten) durch Zahlen darstellen und

(ii) Übergangsfunktion Δ durch die Funktion $\tilde{\Delta}$ auf Zahlen simulieren.

Dies wird uns ermöglichen für jede turingberechenbare Funktion eine äquivalente μ-rekursive Funktion anzugeben, wie in Abbildung 2.4 skizziert.

Zu (i) Um die Konfiguration einer Turingmaschine als Zahl zu repräsentieren, wählen wir die folgende injektive Abbildung $\Psi : (Q \cup \Sigma)^* \to \mathbb{N}_0$. Für $Q \cup \Sigma = \{b_1, b_2, \dots, b_p\}$ ist (die $(p+1)$-adische Zahldarstellung)

$$\begin{array}{rcl} \epsilon & \longmapsto & 0 \\ b_i & \longmapsto & i \\ v_1 \dots v_s & \longmapsto & \sum_{j=1}^{s} \Psi(v_j)(p+1)^{s-j}, \end{array}$$

für $v_1, \dots, v_s \in Q \cup \Sigma$.

Zu (ii) Zentral ist die Konstruktion der Funktion $\tilde{\Delta}$, die für eine beliebige Kodierung die Nachfolgekodierung bestimmt. Um für eine beliebige Zahl feststellen zu können, ob sie der Kodierung einer Endkonfiguration entspricht, benötigen wir eine weitere Funktion. Das folgende Lemma liefert uns beide Funktionen.

Lemma 9. *Zu einer Turingmaschine M gibt es primitiv rekursive Funktionen $\tilde{\Delta}$ und* END *mit*

$$\tilde{\Delta}(x) = \begin{cases} \Psi(K'), & \text{falls } x = \Psi(K) \text{ und } K' \text{ die Folgekonfiguration von } K \text{ ist} \\ 0, & \text{sonst .} \end{cases}$$

$$\mathrm{END}(x) = \begin{cases} 0, & \text{falls } x = \Psi(K) \text{ und } K \text{ Endkonfiguration ist} \\ 1, & \text{sonst.} \end{cases}$$

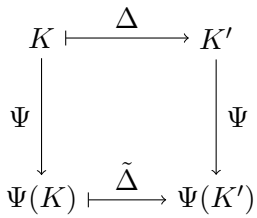

Abbildung 2.4: $\tilde{\Delta}$ macht das Diagramm kommutativ.

Das heißt, $\tilde{\Delta}$ macht das Diagramm in Abbildung 2.4 kommutativ.

Bevor wir Lemma 9 beweisen, fassen wir nochmal alle Schritte zusammen und führen den Beweis von Theorem 5 zu Ende.

Beweis von Theorem 5. Sei $f \in \mathcal{F}_{\mathrm{TM}}^{tot}$ eine r-stellige Funktion. Dann folgt aus Lemma 7, dass es eine 1-Band Turingmaschine $M = (Q, \Sigma, \delta, q_0, F)$ gibt, die f berechnet. Unter Verwendung der Übergangsfunktion $\tilde{\Delta}$ konstruieren wir mittels primitiver Rekursion folgende Funktion $D : \mathbb{N}_0^2 \to \mathbb{N}_0$, wobei

$$\begin{aligned} D(0, x) &:= x \\ D(n+1, x) &:= \tilde{\Delta}(D(n, x)) \end{aligned}$$

für $n \geq 0$. Folglich ist D primitiv rekursiv. Die Funktion liefert uns die Kodierung der Konfiguration K', die nach genau n Schritten aus der Konfiguration K entsteht:

$$D(n, \Psi(K)) = \begin{cases} \text{Kodierung } \Psi(K') \text{ derjenigen Konfiguration K', die entsteht, wenn M in K startet und n Rechenschritt macht,} & \text{falls das in K geht,} \\ 0, & \text{sonst.} \end{cases}$$

Mit Hilfe von D können wir in den Folgekonfigurationen von K nach der *ersten* auftretenden Endkonfiguration *suchen*. Dies können wir mit der μ-rekursiven Funktion $A : \mathbb{N}_0 \to \mathbb{N}_0$

$$A(\Psi(K)) = \begin{cases} \text{(minimale) Anzahl an Rechenschritten, die } M \text{ gestartet in Konfiguration } K \text{ macht, bis sie in Endkonfiguration gerät,} & \text{falls sie das tut, d.h. hält} \\ \text{undefiniert} & \text{sonst, d.h. sie hält nicht.} \end{cases}$$

A ist μ-rekursiv für $A(x) := \mu g(x)$ mit $g(n, x) := \mathrm{END}(D(n, x))$. Wir nehmen an, dass M angesetzt auf Konfiguration K nach $A(\Psi(K))$ Schritten

in Endkonfiguration $K' = \$q\,\mathrm{bin}(f(\mathfrak{x}))$ hält. Dann benötigen wir noch eine Funktion F, die uns aus der Endkonfiguration den Funktionswert $f(\mathfrak{x})$ extrahiert. Außerdem muss auch die Startkonfiguration $\$q_0\,\mathrm{bin}(x_1)\#bin(x_2)\ldots\#\,\mathrm{bin}(x_r)$ mit $\mathfrak{x} = (x_1, \ldots, x_r)$ durch eine Funktion E kodiert werden. Dass es diese primitiv rekursiven Funktionen gibt, zeigt das folgende (technische) Lemma, welches wir erst später beweisen.

Lemma 10. *Es gibt primitiv rekursive Funktionen E, F*

$$E(x_1, x_2, \ldots x_r) = \Psi\left(\$q_0\,\mathrm{bin}(x_1)\#\,\mathrm{bin}(x_2)\#\ldots\#\,\mathrm{bin}(x_r)\right) \quad \textit{und}$$

$$F\Big(\Psi\big(\$q\,\mathrm{bin}(x)\big)\Big) = x.$$

Für die Funktion f wissen wir also, dass bei einer Eingabe gilt

$$\begin{aligned} f(x_1, \ldots, x_r) \text{ ist definiert} \quad &\Leftrightarrow \quad M, \text{ angesetzt auf Konfiguration } \$q_0\,\mathrm{bin}(x_1)\#\ldots\#\,\mathrm{bin}(x_r), \text{ hält in Endzustand} \\ &\Leftrightarrow \quad A\Big(E(x_1, \ldots x_r)\Big) \text{ ist definiert.} \end{aligned}$$

Mit Lemma 10 gilt in diesem Fall:

$$\begin{aligned} f(x_1, \ldots x_r) &= F\Big(\Psi(\text{Endkonfiguration})\Big) \\ &= F\Bigg(D\Big(\underbrace{A(E(x_1, \ldots, x_r))}_{\text{Anzahl der Rechenschritte.}}, \underbrace{E(x_1, \ldots, x_r)}_{\Psi(\text{Startkonfiguration})}\Big)\Bigg) \end{aligned}$$

Damit ist f μ-rekursiv, womit Theorem 5 bewiesen wäre. □

Nun müssen noch die Beweise von Lemma 9 und Lemma 10 nachgeholt werden. Zur Konstruktion dieser primitiv rekursiven Funktionen benötigen wir ein paar Hilfsmittel. Wir simulieren einfache Operationen wie Längenbestimmung, Konkatenation, Präfixbildung, usw. auf $(Q \cup \Sigma)^*$ durch primitiv rekursive Funktionen.

Lemma 11. *Es gibt primitiv-rekursive Funktionen,*

$$\mathrm{L}, \textsc{Concat}, \textsc{Prefix}, \textsc{Suffix}, \textsc{First}, \textsc{Last}, \textsc{Select}$$

sodass für alle $b, c \in (Q \cup \Sigma)^*$ *und alle* i *mit* $0 \leq i \leq |B|$ *gilt:*

$$
\begin{aligned}
L\Big(\Psi(b)\Big) &= |b| \\
\textsc{Concat}\Big(\Psi(b), \Psi(c)\Big) &= \Psi(bc) \\
\textsc{Prefix}\Big(\Psi(b), i\Big) &= \begin{cases} \Psi(b_1 b_2 \dots b_i), & \textit{falls } i > 0 \\ \Psi(\epsilon), & \textit{falls } i = 0 \end{cases} \\
\textsc{Suffix}\Big(\Psi(b), i\Big) &= \begin{cases} \Psi(b_1 b_2 \dots b_i), & \textit{falls } i > 0 \\ \Psi(\epsilon), & \textit{falls } i = 0 \end{cases} \\
\textsc{First}\Big(\Psi(b)\Big) &= \Psi(b_1) \\
\textsc{Last}\Big(\Psi(b)\Big) &= \Psi(b_{|b|}) \\
\textsc{Select}\Big(\Psi(b), i\Big) &= \begin{cases} \Psi(b_i), & \textit{falls } i > 0 \\ \Psi(\epsilon), & \textit{falls } i = 0 \end{cases}
\end{aligned}
$$

Beweis. Zum Beweis geben wir explizit die Funktionen an:

$$
\begin{aligned}
\mathrm{L}(x) &:= \min\{m; m \leq x \text{ und } (p+1)^m > x\} \\
&= \text{niedrigste Potenz von } p+1 \text{ die in x nicht vorkommt}
\end{aligned}
$$

Denn sei $Q = \{(x, m); (p+1)^m \leq x\}$, dann ist $\mathrm{L}(x) = h(x, x) + 1$ für $h(z, x) = \mu_b \chi_Q(z, x) =$ kleinstes $m \leq x : \chi_Q(z, m) = 0$.

$$
\begin{aligned}
\textsc{Concat}(x, y) &= x(p+1)^{\mathrm{L}(y)} + y \\
\textsc{Prefix}(x, i) &= x \text{ div } (p+1)^{\mathrm{L}(x)-i}
\end{aligned}
$$

Denn $\Psi(b_1 \dots b_{|b|}) = \Psi(b_1)(p+1)^{|b|-1} + \Psi(b_2)(p+1)^{|b|-2} + \dots + \Psi(b_i)(p+1)^{|b|-i}$ + niedrigere Terme, wobei $|b| = L(\Psi(b))$ ist.

$$
\textsc{Suffix}(x, i) = \begin{cases} x \mod (p+1)^{\mathrm{L}(x)-i+1}, & \text{falls } i > 0 \\ 0, & \text{falls } i = 0 \end{cases}
$$

Denn $\Psi(b_1 \dots b_{|b|}) =$ höhere Terme $+\Psi(b_{i-1})(p+1)^{|b|-i+1} + \Psi(b_i)(p+1)^{|b|-i} + \Psi(b_{i+1})(p+1)^{|b|-i-1} + \dots + \Psi(b_{|b|})$

$$
\begin{aligned}
\textsc{First}(x) &= \textsc{Prefix}(x, 1) \\
\textsc{Last}(x) &= \textsc{Suffix}(x, \mathrm{L}(x)) \\
\textsc{Select}(x, i) &= \begin{cases} \textsc{First}(\textsc{Suffix}(x, i)), & \text{falls } i > 0 \\ 0 & \text{falls } i = 0 \end{cases}
\end{aligned}
$$

□

Im Folgenden werden wir einige Abkürzungen verwenden:

$$\begin{aligned} x_{(i)} &= \textsc{Select}(x, i) \\ [x, y] &= \textsc{Concat}(x, y) \\ [x, y, z] &= \textsc{Concat}((x, y), z) \end{aligned}$$

Nun holen wir die Beweise von Lemma 9 und Lemma 10 nach. Wir beginnen mit dem Beweis von Lemma 9:

Beweis zu Lemma 9. Zu zeigen ist, dass es Funktionen $\tilde{\Delta}, \mathrm{END} \in \mathcal{P}$ gibt, sodass

1. das Diagramm in Abbildung 2.4 kommutativ ist und
2. $\mathrm{END}\Big(\Psi(K)\Big) = \begin{cases} 0, & \text{falls K Endkonfiguration von M} \\ 1, & \text{sonst.} \end{cases}$

Da die Menge Q der Zustände der Turingmaschine M endlich ist, so ist auch $\Psi_Q := \{\Psi(q), q \in Q\}$ endlich und nach Theorem 1 folgt, dass die charakteristische Funktion χ_{Ψ_Q} primitiv rekursiv. Damit können wir in einer Konfiguration der Turingmaschine nach der Position des Kopfes suchen. Wir setzen dazu $q(x) := \min\{i; i \leq \mathrm{L}(x) \text{ und } x_{(i)} \in \Psi_Q\}$. Dann ist q nach Theorem 1 primitiv rekursiv und wenn x die Kodierung $\Psi(K)$ der Konfiguration $K = \$a_1 \dots a_{k-1} \underset{\uparrow}{q}\, a_k \dots a_t$ ist, dann ist

$$\begin{aligned} q\Big(\Psi(K)\Big) &= k + 1 \\ &= \text{die Stellung des L/S-Kopfs auf dem Band.} \end{aligned}$$

Mit den primitiv rekursiven Stringfunktionen aus Lemma 11 können wir uns nun primitiv rekursive Funktionen definieren, die eine Kodierung x in drei Teile $u(x)$ $w(x)$ $v(x)$ zerlegen:

$$\begin{aligned} u(x) &:= \mathrm{PREFIX}(x, q(x) \dot{-} 2) \\ v(x) &:= \mathrm{SUFFIX}(x, q(x) + 2) \\ w(x) &:= \Big[x_{(q(x) \dot{-} 1)}, x_{(q(x))}, q_{(q(x)+1)}\Big]. \end{aligned}$$

Diese Teile sind die interessanten Stellen, da wir auf dem Teil $w(x)$ (lokal) die Folgekonfiguration berechnen können. Dabei verwenden wir die Eigenschaften der Turingmaschine M. Sei beispielsweise $K = \$a_1 \dots a_{k-1} q a_k a_{k+1} \dots a_t$

eine Konfiguration der Turingmaschine M. Dann zerlegen wir K in

$$u\Big(\Psi(K)\Big) = \Psi(\$a_1 \dots a_{k-2})$$
$$w\Big(\Psi(K)\Big) = \Psi(a_{k-1}\ q\ a_k)$$
$$v\Big(\Psi(K)\Big) = \Psi(a_{k+1} \dots a_t)$$

und berechnen die Folgekonfiguration von $y = w(\Psi(K))$ mit

$$\tilde{\delta}(y) = \begin{cases} \Psi(a_{k-1}a'_k q'), & \text{falls } y = \Psi(a_{k-1}qa_k) \text{ und } \delta(q, a_k) = (q', a'_k, +1) \\ \Psi(a_{k-1}q'q'_k), & \text{falls } y = \Psi(a_{k-1}qa_k) \text{ und } \delta(q, a_k) = (q', a'_k, 0) \\ \Psi(q', a_{k-1}a'_k), & \text{falls } y = \Psi(a_{k-1}qa_k) \text{ und } \delta(q, a_k) = (q', a'_k, -1) \\ 0 & \text{sonst.} \end{cases}$$

Nach Theorem 1 ist $\tilde{\delta}$ primitiv rekursiv, da für a_{k-1}, a_k, q nur endlich viele Möglichkeiten existieren und $\tilde{\delta}$ durch endlich viele Fallunterscheidungen definiert ist. Damit erhalten wir die globale Übergangsfunktion auf ganzen Konfiguration

$$\tilde{\Delta} = \Big[u(x), \tilde{\delta}(w(x)), v(x)\Big].$$

Der Test auf Endkonfiguration ergibt sich durch

$$\mathrm{END}(x) = \begin{cases} 0, & \text{falls } \chi_{(q(x))} \in \Big\{\Psi(e); e \in F\Big\} \\ 1, & \text{sonst.} \end{cases}$$

Beide Funktionen sind somit primitiv rekursiv und leisten das Verlangte. □

Als letztes ist noch der Beweis von Lemma 10 zu zeigen.

Beweis von Lemma 10. Dazu konstruieren wir zwei primitiv rekursive Funktionen E, F, sodass

a) $E(x_1, x_2, \dots x_r) = \Psi\left(\$q_0\,\mathrm{bin}(x_1)\#\,\mathrm{bin}(x_2)\#\dots\#\,\mathrm{bin}(x_r)\right)$ zur Kodierung der Startkonfiguration und

b) $F\Big(\Psi\big(\$q\,\mathrm{bin}(y)\big)\Big) = y$ zum Auslesen des Funktionswerts in der Endkonfiguration

verwendet werden kann. Um Teil a) zu zeigen konstruieren wir zunächst eine primitiv rekursive Funktion $B : \mathbb{N}_0^2 \to \mathbb{N}_0$, die für $i, x \in \mathbb{N}_0$ das i-letzte Bit

in der Binärdarstellung von x berechnet.

$$\begin{aligned} B(i,x) &= (x \mod 2^i) \operatorname{div} 2^{i-1} \\ x &= w_1 2^{s-1} + w_2 2^{s-2} + \cdots + w_{s-i} 2^i + w_{s-i+1} 2^{i-1} \\ &\quad + w_{s-i+2} 2^{i-2} + \cdots + w_{s-1} s^1 + w_s \\ P(n,x) &= \sum_{i=1}^{n} \Psi\Big(\underbrace{B(i,x)}_{0 \text{ oder } 1}\Big)(p+1)^{i-1} \end{aligned}$$

Sowohl B als auch P sind primitiv rekursiv (Übungsaufgabe) und $P(L(x), x) = \Psi\Big(\operatorname{bin}(x)\Big)$. Damit erhalten wir E aus P durch endlich viele Anwendungen von Concat

$$E(x_1, \ldots x_r) = [\Psi(\$), \Psi(q_0), P(\mathrm{L}(x_1), x_1), \Psi(\#), \ldots, \Psi(\#), P(\mathrm{L}(x_r), x_r)]\,.$$

Da P, L und Concat primitiv rekursiv sind, gilt dies auch für E.

Nun zu Teil b). Hier müssen wir die primitiv rekursive Funktion F konstruieren, welche die $(p+1)$-adische Darstellung des Ergebnisses aus der Kodierung der Endkonfiguration liest und in die Zahl selbst transformiert. Dazu lesen wir zunächst die $(p+1)$-adische Darstellung aus der Kodierung der Endkonfiguration mit

$$G(x) = \textsc{Suffix}(x, q(x)+1).$$

G ist primitiv rekursiv und wie gewünscht ist $G(\Psi(\$q \operatorname{bin}(y))) = \Psi(\operatorname{bin}(y))$. Mit der primitiv rekursiven Funktion $H : \mathbb{N}_0 \to \mathbb{N}_0$

$$H(x) = \sum_{i=1}^{\mathrm{L}(x)} \Big(\Big(x \mod (p+1)^i\Big) \operatorname{div} (p+1)^{i-1} - 1\Big)\, 2^{i-1}$$

gilt schließlich $H(\Psi(\operatorname{bin}(y))) = y$, wenn wir o.E. annehmen, dass $\Psi(0) = 1$ und $\Psi(1) = 2$. Wir können nun $F(x) = H(G(x))$ explizit angeben

$$F\Big(\Psi(\$q \operatorname{bin}(y))\Big) = H\Big(\Psi(\operatorname{bin}(y))\Big) = y.$$

Nach Konstruktion ist F primitiv rekursiv und leistet das Verlangte. □

Damit ist schließlich der Beweis vollständig erbracht, dass die Klasse der μ-rekursiven Funktionen und die Klasse der turingberechenbaren Funktionen äquivalent sind. Dies interpretieren wir als Indiz für die Gültigkeit der Church-Turing-These. Wir sehen uns nun weitere Anwendungen an.

2.3.2 Folgerungen und Ausblick

Die Ackermannfunktion ist μ-rekursiv

In Abschnitt 2.2 haben wir noch nicht bewiesen, dass die Ackermannfunktion tatsächlich μ-rekursiv ist. Dies können wir nun sehr einfach nachholen. Die Ackermannfunktion war wie folgt definiert.

$$\begin{aligned} A(0,y) &= y+1 && (2.1)\\ A(x+1,0) &= A(x,1) && (2.2)\\ A(x+1,y+1) &= A(x,A(x+1,y)) && (2.3) \end{aligned}$$

Es reicht nun zu beweisen, dass die Ackermannfunktion turingberechenbar ist. Dazu schreiben wir $A(x,y)$ zunächst als Zeichenfolge:

$$\underbrace{1\ldots1}_{x\text{-mal}}\ A\ \underbrace{1\ldots1}_{y\text{-mal}}$$

Wir beschreiben nun eine Turing-Maschine M, die A berechnet:

Vorbereitung Wir übersetzen die Binärdarstellung von x und y, trennen x und y durch das Zeichen A und setzen den Lese-/Schreibkopf darauf.

$$\$\overset{\downarrow}{\text{bin}}(x)\,\#\text{bin}(y)\,\sqcup\ldots \Rightarrow \$\underbrace{1\ldots1}_{x}\overset{\downarrow}{A}\underbrace{1\ldots1}_{y}\sqcup\ldots$$

Solange ein A auf dem Band steht, führt M folgende Makros aus, abhängig von der Umgebung des am weitesten rechts stehenden A.

(2.1) Falls links vom rechtesten A ein \$ steht:

$$\$\overset{\downarrow}{A}\underbrace{1\ldots1}_{y}\sqcup\ldots \quad\Rightarrow\quad \$\overset{\downarrow}{\underbrace{1\ldots1}_{y+1}}\sqcup\ldots$$

und wir sind fertig. Falls links vom rechtesten A ein weiteres A steht:

$$\ldots A\overset{\downarrow}{A}\underbrace{1\ldots1}_{y}\sqcup\ldots \quad\Rightarrow\quad \ldots\overset{\downarrow}{A}\underbrace{11\ldots1}_{y+1}\sqcup\ldots$$

(2.2) Falls links vom rechtesten A eine 1 steht und rechts $\sqcup$:

$$\ldots1\overset{\downarrow}{A}\sqcup\ldots \quad\Rightarrow\quad \ldots\overset{\downarrow}{A}1\sqcup\ldots$$

(2.3) Falls links und rechts vom rechtesten A eine 1 steht:

$$\ldots \underbrace{L}_{\in\{A,\$\}} \underbrace{1\ldots11}_{x+1} \overset{\downarrow}{A} \underbrace{1\ldots11}_{y+1} \sqcup \ldots \quad \Rightarrow \quad \ldots A \underbrace{1\ldots1}_{x} A \underbrace{1\ldots11}_{x+1} \overset{\downarrow}{A} \underbrace{1\ldots1}_{y} \sqcup \ldots$$

Die $y+1$ Zeichen werden also weit genug nach rechts verschoben (kopiert) um Platz für eine Kopie von $x+1$ zu schaffen.

Schließlich wird das Ergebnis wieder in die binäre Darstellung umgewandelt:

$$\overset{\downarrow}{\$1}\ldots1\sqcup\ldots \Rightarrow \$\mathrm{bin}\left(A\left(x,y\right)\right)\sqcup\ldots$$

Beispiel $A\left(1,2\right) =?$

Band	Makro
$\overset{\downarrow}{\$1}A11\sqcup\ldots$	(initial)
$\$A\overset{\downarrow}{1}A1\sqcup\ldots$	(2.3)
$\$AA1\overset{\downarrow}{A}\sqcup\ldots$	(2.3)
$\$AA\overset{\downarrow}{A}1\sqcup\ldots$	(2.2)
$\$A\overset{\downarrow}{A}11\sqcup\ldots$	(2.1)
$\$\overset{\downarrow}{A}111\sqcup\ldots$	(2.1)
$\overset{\downarrow}{\$1}111\sqcup\ldots \Rightarrow A\left(1,2\right)=4$	(2.1)

Einmalige Anwendung des μ-Operators

Interessanterweise lässt sich zeigen, dass die einmalige Anwendung des μ-Operators ausreichend ist, um eine beliebige μ-rekursive Funktion zu definieren. Zunächst müssen wir definieren, wann zwei (partielle) Funktionen gleich heißen sollen.

Definition 7 (Gleichheit von Funktionen)**.** Zwei r-stellige, partielle, μ-rekursive Funktionen $f^{(r)}, g^{(r)} \in \mathcal{F}_{\mu}^{par}$ heißen *gleich*, wenn sie den selben Definitionsbereich $D = D\left(f\right) = D\left(g\right)$ besitzen und für alle $\mathfrak{x} \in D$ gilt, dass $f\left(\mathfrak{x}\right) = g\left(\mathfrak{x}\right)$. Wir schreiben dann $f \tilde{=} g$.

Aus dem Beweis von Theorem 5 folgern wir nun, dass jede beliebige μ-rekursive Funktion durch einmalige Anwendung des μ-Operators dargestellt werden kann. Eine solche Darstellung nennen wir Kleene'sche Normalform.

Korollar 1 (Kleene'sche Normalform)**.** *Zu jedem* $f^{(r)} \in \mathcal{F}_\mu^{(par)}$ *gibt es primitiv rekursive Funktionen* $p^{(r+1)}, q^{(r+1)}$*, sodass*

$$\forall \mathfrak{x} : f(\mathfrak{x}) \;\tilde{=}\; q(\mu p(\mathfrak{x}), \mathfrak{x}).$$

Beweis. Im Beweis von Theorem 5 haben wir gezeigt, dass wir zu jeder turingberechenbaren Funktion f eine äquivalente μ-rekursive Funktion konstruieren können. Dazu haben wir Funktionen F, D, A, E konstruiert, sodass

$$f(\mathfrak{x}) = F(D(A(E(\mathfrak{x})), E(\mathfrak{x}))).$$

Dabei waren F, D, E primitiv rekursive Funktionen. Lediglich zur Definition der Funktion $A(x) = \mu g(x)$ haben wir den μ-Operator einmalig auf die primitiv rekursive Funktion $g(n, x) = \mathrm{END}(D(n, x))$ angewendet. □

Dass die Anwendung des μ-Operators der Durchführung von while-Schleifen entspricht, haben wir bereits im Beweis von Lemma 6 gesehen. Folglich impliziert das obige Korollar, dass es für jede turingberechenbare Funktion eine Turingmaschine gibt, die mit einer einzigen while-Schleife auskommt. Auch beim Programmieren käme man deshalb rein theoretisch mit einer einzigen while-Schleife aus.

Normierte Registermaschinen

Abgesehen von Turingmaschinen, wurden noch viele andere Maschinenmodelle eingeführt und untersucht. So zum Beispiel *normierte Registermaschinen*. Eine solche besteht aus m Registern, in welchen jeweils eine natürliche Zahl gespeichert werden kann. Folgende Operationen stehen zur Verfügung:

a_i	:	addiere 1 im $i-$ten Register
s_i	:	subtrahiere 1 im $i-$ten Register
$(M)_i$	:	iteriere M so oft, bis im $i-$ten Register 0 steht
$M_1 M_2$	:	führe nacheinander M_1, M_2 aus

Beispiel 4. Das Programm $(s_1 a_2 a_2)_1 (s_2 a_1)_2$ führt auf der Eingabe $(x, 0, \ldots)$ (also x in Register 1 und 0 in Register 2) zu der Ausgabe $(2x, 0, \ldots)$.

Auch für normierte Registermaschinen kann man beweisen, dass die Klasse der zugehörigen berechenbaren Funktionen

$$\mathcal{F}_{\mathrm{NRM}} := \{f : f \text{ berechenbar durch norm. Registermaschine}\}$$

mit der Klasse der μ-rekursiven (bzw. turingberechenbaren) Funktionen übereinstimmt, d.h. $\mathcal{F}_{\mathrm{NRM}} = \mathcal{F}_\mu^{tot}$. Man erhält also eine weitere Stütze für die These von Church-Turing.

Universelle Turingmaschinen

Mit dem Normalformtheorem (Korollar 1) haben wir gezeigt, dass es zu jeder μ-rekursiven Funktion f zwei primitiv rekursive Funktionen p, q gibt, sodass f und $q(\mu p(\mathfrak{x}), \mathfrak{x})$ gleich sind, d.h.

$$\forall \mathfrak{x} : \ f(\mathfrak{x}) \ \tilde{=} \ q(\mu p(\mathfrak{x}), \mathfrak{x}).$$

Hierbei hingen die Funktionen p und q jedoch von der Funktion f (bzw. von der Turingmaschine M, die f berechnet) ab. Statt die Struktur von M in die Funktionen p und q „einzubauen“ könnte man auch eine spezielle Kodierung $k = \langle M \rangle$ (Gödelisierung) der Turingmaschine als Argument übergeben. Dieser Trick führt zu einer verstärkten Form des Normalformtheorems.

Theorem 6 (Starke Kleene'sche Normalform)**.** *Es gibt primitiv rekursive Funktionen U, T, φ, so dass zu jeder μ-rekursiven Funktion eine Kodierung k einer Turingmaschine existiert, so dass*

$$\forall \mathfrak{x} : f(\mathfrak{x}) \ \tilde{=} \ U(\mu T(k, \varphi(\mathfrak{x}))).$$

Das Theorem ermöglicht den Bau einer *universellen* Turingmaschine M^*, welche die Funktion

$$(k, \mathfrak{x}) \ \longmapsto \ U(\mu T(k, \varphi(\mathfrak{x})))$$

aus $\mathcal{F}_\mu^{par}$ berechnet. Bei Eingabe der *richtigen* Kodierung k kann die universelle Turingmaschine M^* also *jede* Funktion aus $\mathcal{F}_\mu^{tot}$ berechnen. Die Kodierungen (der Turingmaschinen) können als Programme interpretiert werden, welche nun nicht mehr selbst in der Turingmaschine gebunden sind sondern stattdessen Eingabedaten von M^* sind. Die universelle Turingmaschine führt diese Programme aus. Die Idee, Programme als Eingabe zu betrachten, liegt übrigens auch der Von-Neumann-Rechnerarchitektur zu Grunde.

2.4 Entscheidbarkeit

In Abschnitt 2.2 und Abschnitt 2.3 haben wir uns mit Modellen zur Beschreibung von Berechenbarkeit auseinander gesetzt. Nun möchten wir uns überlegen, was mit diesen Modellen prinzipiell ausgerechnet werden kann und wo wir die Grenzen dieser Modelle erreichen.

Für Conways Spiel des Lebens haben wir in Abschnitt 2.1 bereits eine Grenze kennengelernt. Es gibt keinen Algorithmus, der für zwei beliebige Generationen prüft, ob zu irgend einem Zeitpunkt des Spiels die eine aus der anderen hervorgeht. Es gibt jedoch viel einfachere und allgemeinere Probleme für die es keinen Algorithmus gibt. In Abschnitt 2.3 haben wir uns zum Schluss mit

universellen Turingmaschinen befasst. Als Eingabe bekam die universelle Turingmaschine M^* ein Tupel $(k, \mathfrak{x})$, wobei k die Kodierung einer beliebigen Turingmaschine M und $\mathfrak{x}$ die Eingabe für M war. Die universelle Turingmaschine simulierte die Berechnungen von M auf der Eingabe $\mathfrak{x}$. Hier stellt sich natürlich die Frage, ob es einen Algorithmus gibt, der für eine beliebige (Kodierung einer) Turingmaschine M und einer beliebigen Eingabe $\mathfrak{x}$ feststellt, ob M bei ihren Berechnungen irgendwann anhält. Diese informelle Problemstellung ist als *Halteproblem* für Turingmaschinen bekannt. Wir werden für dieses Problem formal beweisen, dass es keinen solchen Algorithmus gibt.

Da Turingmaschinen auf beliebigen Zeichenketten operieren, betrachten wir zunächst keine Funktionen sondern Sprachen. Natürlich gilt

$$\begin{aligned} & f : \Sigma^* \to \mathbb{N}_0 \text{ ist turingberechenbar} \\ \Leftrightarrow \quad & \tilde{f} : \mathbb{N}_0 \xrightarrow{\pi^{-1}} \Sigma^* \xrightarrow{f} \mathbb{N}_0 \text{ ist } \mu\text{-rekursiv.} \end{aligned}$$

Dabei bezeichne $\pi : \Sigma^* \to \mathbb{N}_0$ eine geeignete berechenbare Gödelisierung. Für eine Sprache führen wir nun den Begriff der Entscheidbarkeit ein.

Definition 8 (Entscheidbar)**.** Sei Σ ein endliches Alphabet. Eine Sprache $L \subseteq \Sigma^*$ heißt *entscheidbar*, wenn ihre charakteristische Funktion $\chi_L : \Sigma^* \to \{0, 1\}$ mit

$$\chi_L(w) = \begin{cases} 1, & \text{falls } w \in L \\ 0, & \text{sonst.} \end{cases}$$

turingberechenbar ist. Die Sprache L heißt *semi-entscheidbar*, wenn die partielle Funktion

$$\chi_L^*(w) = \begin{cases} 1, & \text{falls } w \in L \\ \text{undefiniert}, & \text{sonst.} \end{cases}$$

turingberechenbar ist.

Für eine entscheidbare Sprache L können wir also mit einem Verfahren (Algorithmus) in endlich vielen Schritten feststellen, ob ein Wort $w \in \Sigma^*$ in L liegt oder nicht. Wir können also für χ_L eine Turingmaschine M konstruieren und auf w ansetzen. Die Turingmaschine muss nach endlich vielen Schritten halten (da χ_L total) und den Wert 0 oder 1 liefern. Liefert M (bzw. χ_L) den Wert 1, so sagen wir M *akzeptiert* w. In diesem Fall liegt w in L. Andernfalls liefert M die Ausgabe 0, d.h. $w \notin L$, und wir sagen auch M *verwirft* w. Entscheidet eine Turingmaschine M die Sprache L, d.h. sie berechnet χ_L, so schreiben wir auch $L(M) = L$.

2.4.1 Unentscheidbarkeit des Halteproblems

Um das Halteproblem formal zu beschreiben überlegen wir uns zunächst, wie wir eine Turingmaschine M als Eingabe über den Alphabet $\{0, 1, \#\}$

ausdrücken können. Diese Darstellung bezeichnen wir als *Kodierung* $\langle M \rangle$ der Turingmaschine. Die Kodierung ist jedoch selbst eine Zeichenkette und daher von einer Gödelnummer zu unterscheiden. Sei $M = (\Sigma, Q, \delta, q_0, F)$ eine Turingmaschine. Wir nummerieren die Elemente von $Q \cup \Sigma$ wie folgt durch

$$\begin{aligned} Q &= \{q_0, q_1, \ldots, q_{p-1}\} && \text{Zustände,} \\ \Sigma &= \{a_p, a_{p+1}, \ldots, a_r\} && \text{Bandalphabet,} \\ F &= \left\{q_{p-|F|}, \ldots, q_{p-1}\right\} && \text{Endzustände } (F \subseteq Q)\,. \end{aligned}$$

Da so Zustände und Zeichen eindeutig einer natürlichen Zahl zugeordnet werden, können wir diese Zahl mit ihrer Binärdarstellung identifizieren und haben sie so mit dem Alphabet $\{0, 1, \#\}$ dargestellt. Nun müssen wir nur noch die Übergangsfunktion in geeigneter Weise darstellen. Dazu kodieren wir $\delta(q_i, a_j) = (q_{i'}, a_{j'}, m)$ durch den String

$$\#\# \operatorname{bin}(i) \# \operatorname{bin}(j) \# \operatorname{bin}\left(i'\right) \# \operatorname{bin}\left(j'\right) \# \operatorname{bin}(c)\,,$$

wobei wir die Kopfbewegung $m \in \{-1, 0, +1\}$ ausdrücken durch

$$c = \begin{cases} 0, & \text{falls } m = -1 \\ 1, & \text{falls } m = 0 \\ 2, & \text{falls } m = 1. \end{cases}$$

Insgesamt ergibt sich die Kodierung der Turingmaschine M somit durch

$$\begin{aligned} \langle M \rangle \;=\; \#\#\#\# \;& \overbrace{\operatorname{bin}(0) \# \cdots \# \operatorname{bin}(p-1)}^{Q} \\ \#\#\# \;& \underbrace{\operatorname{bin}(p) \# \cdots \# \operatorname{bin}(r)}_{\Sigma} \\ \#\#\# \;& \underbrace{\operatorname{bin}(p-|F|) \# \cdots \# \operatorname{bin}(p-1)}_{F} \\ \#\#\# \;& \underbrace{\delta_1 \# \cdots \# \delta_k}_{\delta} \\ \#\#\#\# & \end{aligned}$$

Durch die Kodierung des Alphabets ergibt sich automatisch eine Kodierung der Eingabe $w = a_{i_1} a_{i_2} \cdots a_{i_m}$ durch

$$\langle w \rangle = \operatorname{bin}(i_1) \# \operatorname{bin}(i_2) \# \cdots \# \operatorname{bin}(i_m) \#\#.$$

Das *allgemeine Halteproblem* für Turingmaschinen ist nun definiert durch die Sprache

$$H := \{x \in \{0, 1, \#\}^* \mid x = \langle M \rangle \langle w \rangle \text{ und } M \text{ hält bei Eingabe } w\}\,.$$

Nun möchten wir beweisen, dass diese Sprache unentscheidbar ist. Dabei ist leicht zu testen, ob $\langle M \rangle$ die Kodierung einer Turingmaschine ist. Ob diese Turingmaschine jedoch auf der Eingabe w hält, ist unentscheidbar. Um dies zu beweisen betrachten wir zunächst eine eingeschränkte Version des Halteproblems. Dazu betrachten wir diejenigen Turingmaschinen, die halten, wenn sie auf ihre eigene Kodierung als Eingabe angesetzt werden. Es genügt, wenn wir uns auf diesen Spezialfall beschränken.

Theorem 7. *Das eingeschränkte Halteproblem*

$$H_e := \{x \in \{0,1,\#\}^* \mid x = \langle M \rangle \text{ und } M \text{ hält bei Eigabe } x\}$$

ist unentscheidbar.

Beweis. Wir beweisen die Aussage indirekt, indem wir annehmen, dass H_e entscheidbar wäre. Somit existiert eine Turingmaschine M, welche χ_{H_e} berechnet. Nun können wir eine Turingmaschine M' konstruieren, sodass $\forall x \in \{0,1,\#\}^*$ gilt

- M' hält bei Eingabe x, falls $x \notin H_e$,
- M' hält nicht bei Eingabe x, falls $x \in H_e$.

Nun betrachten wir das Verhalten von M' bei Eingabe der eigenen Kodierung $x = \langle M' \rangle$. Dann ist

$$\begin{aligned} M' \text{ hält bei Eingabe } \langle M' \rangle &\Leftrightarrow \langle M' \rangle \notin H_e \\ &\Leftrightarrow M' \text{ hält nicht bei Eingabe } \langle M' \rangle \end{aligned}$$

ein Widerspruch und es folgt, dass H_e nicht entscheidbar ist. □

Aus dem Spezialfall folgt unmittelbar die Unentscheidbarkeit des allgemeinen Halteproblems.

Korollar 2. *Das allgemeine Halteproblem*

$$H := \{x \in \{0,1,\#\}^* \mid x = \langle M \rangle \langle w \rangle \text{ und } M \text{ hält bei Eingabe } w\}$$

ist unentscheidbar.

Beweis. Für die Eingabe $x = \langle M \rangle$ für eine Turingmaschine M gilt

$$x \in H_e \iff M \text{ hält bei Eingabe } \langle M \rangle \iff x' := \underbrace{\langle M \rangle \langle \langle M \rangle \rangle}_{x\langle x \rangle} \in H.$$

Da $x' = x\langle x \rangle$ leicht aus x berechnet werden kann, würde aus der Entscheidbarkeit von von H auch die Entscheidbarkeit von H_e folgen. □

Damit ist ein für die Informatik wichtiges Problem, die vollständige semantische Korrektheit von Programmen, als unentscheidbar erkannt. Es ist ebenfalls unentscheidbar, ob eine beliebige Turingmaschine eine vorgegebene Aufgabe löst. Dies ist Inhalt des folgenden Satzes von Rice, den wir hier ohne Beweis zitieren:

Theorem 8 (Satz von Rice)**.** *Sei* $\emptyset \subsetneq F \subsetneq \mathcal{F}_{\mu}^{par}$*, dann ist*

$$\{\langle M\rangle \ ; \ M \textit{ ist Turingmaschine, die eine Funktion aus } F \textit{ berechnet } \}$$

unentscheidbar.

2.4.2 Weitere Beispiele unentscheidbarer Probleme

Im Folgenden gehen wir noch kurz auf weitere unentscheidbare Probleme ein. Conways Spiel des Lebens haben wir bereits in Abschnitt 2.1 erwähnt. Auch haben wir bereits in der Vorlesung das Problem der *Wang-Parkettierung* kennengelernt. Gegeben sind Gruppen von Quadraten einheitlicher Größe, deren Kanten mit bestimmten Farben markiert sind.

Das Problem besteht darin, zu entscheiden, ob die Ebene mit den gegebenen Kachel-Typen lückenlos parkettiert werden kann. Dabei dürfen die Kacheln nicht rotiert werden und nur gleichfarbige Kanten aneinander gelegt werden.

Ein anderes unentscheidbares Problem ist *Hilberts zehntes Problem*. Gegeben ist ein Polynom $p(x_1, \ldots, x_n)$ mit ganzzahligen Koeffizienten, für welches entschieden werden soll, ob es $a_1, \ldots a_n \in \mathbb{Z}$ gibt mit $p(a_1, \ldots a_n) = 0$. Die Unentscheidbarkeit hängt hier von der Ganzzahligkeit der Lösungen ab. Ob es eine reelle Lösung gibt, ist (leicht) entscheidbar.

In vorigen Semestern haben wir uns bereits mit Grammatiken beschäftigt. Eine (unbeschränkte) Grammatik G war dabei gegeben durch ein 4-Tupel (T, N, S, R), wobei

T : die Menge der Terminalsymbole $(a, b, c, \ldots)$,
N : die Menge der Nichtterminalsymbole $(A, B, C, \ldots \notin T)$,
S : das ausgezeichnete Startsymbol aus N und
R : die Produktions- bzw. Ersetzungsregeln $\alpha \to \beta$ mit $\alpha, \beta \in \{N \cup T\}^*$

bezeichnen. Dabei sind T, N und R stets endliche Mengen. Die von G erzeugte Sprache $L(G)$ ist definiert als der transitive Abschluss der Relation

$\rightarrow$, d.h. $L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$. Für die Grammatik G besteht das allgemeine *Wortproblem* darin, zu entscheiden, ob ein beliebiges Wort $w \in T^*$ in der Sprache $L(G)$ liegt. Dieses Problem ist ebenfalls unentscheidbar, da man das Halteproblem darauf reduzieren kann (Übungsaufgabe).

Schließlich betrachten wir noch das *Post'sche Korrespondenzproblem*. Für ein Alphabet Σ sind zwei Listen $v_1, \ldots, v_k$ und $w_1, \ldots, w_k$ von Wörtern in Σ^* gegeben. Die Frage ist, ob es eine nicht-leere Indexfolge $i_1, i_2, \ldots, i_m$ gibt, sodass $v_{i_1} v_{i_2} \cdots v_{i_m} = w_{i_1} w_{i_2} \cdots w_{i_m}$ gilt. Auch hier kann mittels Reduktion gezeigt werden, dass dieses Problem unentscheidbar ist.

Es sei noch einmal erwähnt, dass Unentscheidbarkeit bedeutet, dass es keinen allgemeinen Algorithmus für *alle* Fälle gibt. Das Lösen von Spezialfällen kann dagegen durchaus möglich sein, erfordert aber im Allgemeinen einige Anstrengung.

2.4.3 Rekursive und rekursiv aufzählbare Prädikate

Analog zur Entscheidbarkeit (die wir über Turingberechenbarkeit definiert haben), können wir Eigenschaften von Prädikaten $P \subseteq \mathbb{N}_0$ untersuchen.

Definition 9. Sei P ein Prädikat, d.h. $P \subseteq \mathbb{N}_0$. P heißt genau dann *rekursiv*, wenn die charakteristische Funktion von P μ-rekursiv und total ist, d.h. $\chi_P \in \mathcal{F}_\mu^{tot}$. P heißt genau dann *rekursiv aufzählbar*, wenn gilt

- $P = \emptyset$ oder
- $P = \{f(x) \mid x \in \mathbb{N}_0\}$ für eine Funktion $f \in \mathcal{F}_\mu^{tot}$.

Mit dieser Definition sind die Begriffe *rekursiv* und *entscheidbar* gleichbedeutend. Wie das folgende Lemma zeigt, sind dann auch die Begriffe semientscheidbar und rekursiv aufzählbar äquivalent.

Lemma 12. *Ein Prädikat $P \subseteq \mathbb{N}_0$ ist genau dann rekursiv aufzählbar, wenn die Funktion c_P mit*

$$c_P(y) = \begin{cases} 1, & y \in P \\ \textit{undefiniert}, & y \notin P \end{cases}$$

in $\mathcal{F}_\mu^{par}$ liegt.

Beweis. Sei zunächst P rekursiv aufzählbar. Für $P = \emptyset$ ist c_P nirgends definiert und damit in $\mathcal{F}_\mu^{par}$. Sei also $P = \{f(x) \mid x \in \mathbb{N}_0\}$ für eine μ-rekursive Funktion f. Dann ist

$$c_P(y) = 1 \dot{-} |y - f(\underbrace{\mu x (f(x) = y)}_{\text{definiert} \Leftrightarrow y \in P})|$$

in $\mathcal{F}_{\mu}^{par}$, da die verwendeten Funktionen (modifizierte Differenz, Betragsdifferenz) primitiv rekursiv sind.

Ist nun umgekehrt $c_P \in \mathcal{F}_{\mu}^{par}$, so können wir c_P nach Korollar 1 durch zwei primitiv rekursive Funktionen und einmalige Anwendung des μ-Operators darstellen, also $c_P(y) \doteq q(\mu s(y), y)$ mit $q, s \in \mathcal{P}$. Es gilt also

$$y \in P \Leftrightarrow c_P(y) \text{ ist definiert } \Leftrightarrow \exists z : s(y, z) = 0. \tag{2.4}$$

Wir wollen zeigen, dass P rekursiv aufzählbar ist. Dazu verwenden wir den Trick der Paarkodierung, beispielsweise mit der Cantorschen Paarungsfunktion oder mit der Funktion $k : \mathbb{N}_0^2 \to \mathbb{N}_0$ mit $k(v, w) = 2^v 3^w$. So ist k primitiv rekursiv und es gibt zwei Umkehrfunktionen $d_1, d_2 \in \mathcal{P}$, sodass $\forall v, w$

$$\begin{aligned} d_1(k(v, w)) &= v \\ d_2(k(v, w)) &= w \end{aligned}$$

gilt. Ist $P = \emptyset$, so ist P nach Definition rekursiv aufzählbar. Sei also $P \neq \emptyset$, dann definieren wir für ein festes $p \in P$

$$f(x) := \begin{cases} d_1(x), & \text{falls } \exists v, w \in \mathbb{N}_0 : x = 2^v 3^w \text{ und } s(d_1(x), d_2(x)) = 0 \\ p, & \text{sonst.} \end{cases}$$

Die Funktion f ist selbst primitiv rekursiv und es gilt $P = \{f(x) \mid x \in \mathbb{N}_0\}$. Dabei gilt die Beziehung

- $\supseteq$: nach Definition von f und Gleichung 2.4;
- $\subseteq$: da für $y \in P$ nach Gleichung 2.4 gilt, dass $\exists z : s(y, z) = 0$; setze $x := k(y, z)$. Dann ist $s(d_1(x), d_2(x)) = s(y, z) = 0$, also $f(x) = d_1(x) = y$.

□

Wie der Beweis zeigt, lassen sich (nicht-leere) rekursiv aufzählbare Mengen sogar von einer primitiv rekursiven Funktion aufzählen. Wir halten noch folgende wichtige Beziehung fest.

Theorem 9. *Die Menge der rekursiven Prädikate ist echt enthalten in der Menge der rekursiv aufzählbaren Prädikate, d.h. es gilt*

$$\{P \subseteq \mathbb{N}_0 \mid P \textit{ rekursiv }\} \subsetneq \{P \subseteq \mathbb{N}_0 \mid P \textit{ rekursiv aufzählbar }\}.$$

Beweis. Um die echte Inklusion zu zeigen, betrachten wir erneut das Halteproblem

$$H = \{y \in \mathbb{N}_0 \mid y = \Psi(\langle M \rangle \langle w \rangle) \text{ und } M \text{ hält bei Eingabe } w\}$$

Nach Korollar 2 ist H nicht entscheidbar, also nicht rekursiv. Das Halteproblem ist jedoch rekursiv aufzählbar, denn

$$c_H(y) := 1 \dot{-} (1 \dot{-} (\#\text{Rechenschritte, nach denen } M \text{ bei Eingabe } w \text{ hält}))$$

ist in $\mathcal{F}_\mu^{par}$. (Das zeigt man wie im Beweis von Theorem 5.) □

Interessanterweise gilt für Sprachen und Grammatiken folgende Eigenschaft.

Theorem 10. *Eine Sprache $L \subseteq \Sigma^*$ ist genau dann rekursiv aufzählbar, wenn es eine Grammatik G gibt, die L erzeugt, d.h. es gilt $L = L(G)$.*

Allerdings gibt es auch Mengen, die nicht rekursiv aufzählbar sind. Zum Schluss zeigen wir noch einen nützlichen Zusammenhang zwischen rekursiven und rekursiv aufzählbaren Prädikaten.

Lemma 13. *Sei $P \subseteq \mathbb{N}_0$ ein Prädikat und $\mathbb{N}_0 \setminus P$ das Komplement. Sind P und $\mathbb{N}_0 \setminus P$ rekursiv aufzählbar, so ist P (bzw. das Komplement) rekursiv.*

Beweis. Da P und das Komplement rekursiv aufzählbar sind, gibt es $f, g \in \mathcal{F}_\mu^{tot}$, sodass

$$\begin{aligned} P &= \{f(x) \mid x \in \mathbb{N}_0\} \\ \mathbb{N}_0 \setminus P &= \{g(x) \mid x \in \mathbb{N}_0\}. \end{aligned}$$

Damit definieren wir die Funktion $h(y) := \mu x (f(x) = y \ \vee \ g(x) = y)$. Da $h \in \mathcal{F}_\mu^{tot}$ ist auch die charakteristische Funktion χ_P von P

$$\chi_P(y) = \begin{cases} 1, & \text{falls } f(h(y)) = y \\ 0, & \text{falls } g(h(y)) = y. \end{cases}$$

μ-rekursiv und total und P folglich rekursiv. □

Kapitel 3

Praktische Berechenbarkeit

Im vorangegangenen Kapitel haben wir Modelle zur Formalisierung des Berechenbarkeitsbegriffs eingeführt und Grenzen der Berechenbarkeit untersucht. Wir haben formell gezeigt, dass es Problemstellungen gibt, die durch keinen Algorithmus gelöst bzw. entschieden werden können. Nun wenden wir uns der Frage der Zeitkomplexität entscheidbarer Probleme zu. Entscheidbare Problemstellungen können so *schwer* zu lösen sein, dass aus praktischer Sicht keine effiziente (bezüglich Laufzeit und/oder Speicherplatz) Implementierung möglich ist.

Im Vorlesungsteil Algorithmen und Berechnungskomplexität I hatten wir als Zeit- und Platzkomplexitätsmaß die O-Notation eingeführt, Algorithmen in Pseudocode formuliert und einzelne Berechnungsschritte durch Konstanten abgeschätzt. Wir haben dabei festgestellt, dass viele klassische Problemstellungen in *Polynomialzeit* in Bezug auf die Eingabegröße gelöst werden können. Übertragen auf unsere Maschinenmodelle entspricht diese Vorgehensweise im Wesentlichen einer RAM im Einheitskostenmaß. Solange die verwendeten Objekte und Zahlen nicht zu groß werden, ist die Annahme, dass eine einzelne Rechenoperation in konstanter Zeit ausgeführt werden kann, insgesamt sinnvoll. Programme, die wir in Pseudocode formulieren können, lassen sich stets durch eine RAM oder eine Turingmaschine in annähernd (polynomieller Faktor) gleicher Laufzeit realisieren. Auf diesen Zusammenhang gehen wir in Abschnitt 3.1 ein. Als Konsequenz lassen sich Aussagen über Turingmaschinen auf die RAM und damit auch auf reale Computer übertragen. Im Vorlesungsteil Algorithmen und Berechnungskomplexität I waren obere und untere polynomielle Laufzeitschranken von Interesse. Das wird hier unerheblich sein, wir wollen eher wissen, ob es überhaupt polynomielle obere Schranken gibt oder nicht. Wir können daher großzügiger mit den Laufzeiten umgehen.

3.1 Die Random Access Machine

Im letzten Semester haben wir die sogenannte Random Access Machine (RAM) kennen gelernt und diese als theoretische Grundlage für Laufzeitanalysen verwendet. Die RAM besteht aus folgenden Elementen:

- Abzählbar unendlich viele Speicherzellen, die mit natürlichen Zahlen adressiert werden.
- Jede Speicherzelle kann eine beliebige reelle (approximiert, z.B. gemäß IEEE Standard) oder natürliche Zahl enthalten. Es ist sowohl eine direkter, als auch indirekter Speicherzugriff möglich
- Elementare Rechenoperationen: Addition, Subtraktion, Multiplikation, Division, Restbildung, Abrunden, Aufrunden
- Elementare Relationen: $\leq$, $\geq$, $=$, $\vee$, $\wedge$
- Kontrollierende Befehle: Verzweigung, Aufruf von Subroutinen, Rückgabe
- Datenbewegende Befehle: Laden, Speichern, Kopieren

Kostenmaße Als Kostenmaß der RAM haben wir für Laufzeitanalysen von Algorithmen im vergangenen Semester das *Einheitskostenmaß* verwendet. Für die Ausführung einer der Rechenoperationen wurde dazu eine Kosteneinheit veranschlagt. Die Art der Rechenoperation, sowie die Größe bzw. Länge der Zahlen war dabei unerheblich. Aus praktischer Sicht ist dies sinnvoll, wenn die Größe des Problems und die verwendeten Zahlen klein genug sind. Alle modernen Rechner verwenden beispielsweise eine fest eingebaute Floating-Point Arithmetik nach IEEE Standard, die in einem vorgegebenen relativ großen Zahlenbereich sehr effizient arbeitet.

Bei dieser Abstraktion ist jedoch Vorsicht geboten. Es ist nicht erlaubt, beliebig große Zahlen zu verwenden. Dadurch ließen sich vollständige (exponentiell große) Lösungen in einzelnen Zahlen *konstant* kodieren. Im Folgenden verwenden wir daher das (realistischere) *logarithmische Kostenmaß*. Im logarithmischen Kostenmaß gehen wir davon aus, dass alle Zahlen binär kodiert sind und der Zugriff auf ein Bit in $\Theta(1)$ durchgeführt werden kann. Entsprechend benötigt der Zugriff auf eine natürliche Zahl n beispielsweise $O(\log_2 n)$ Zeit.

Ein Programm für die RAM befindet sich in einem separaten Speicher, welcher nicht zur Laufzeit manipuliert werden kann. Das Programm besteht aus endlich vielen Zeilen. Zu Beginn dürfen nur endlich viele Speicherzellen mit der Eingabe belegt sein. Eine Programmzeile besteht stets aus einer

Operation op_j, einer Menge von Speicherzellen $i_1, i_2, \ldots, i_k$, auf welche diese Operation zugreifen muss und einer Speicherzelle i_p in welche das Ergebnis geschrieben wird.

Ausgehend von einer RAM R mit Laufzeit $t(n)$ für eine Eingabe aus insgesamt n Bits, möchten wir das Verhalten von R durch eine 2-Band Turingmaschine M simulieren. Dazu verwenden wir das erste Band als Speicher für die belegten Speicherzellen der RAM:

$$\text{Band 1: } \#\#\# \operatorname{bin}(i_1) \# \operatorname{bin}(c(i_1)) \#\# \operatorname{bin}(i_2) \# \operatorname{bin}(c(i_2)) \#\# \cdots \#\# \operatorname{bin}(i_m) \# \operatorname{bin}(c(i_m)) \#\#\#$$

Dabei sind $i_1, i_2, \ldots, i_m$ die aktuell belegten Speicherzellen und $c(i_1)$, $c(i_2)$, …, $c(i_m)$ deren Inhalt. Anfangs enthält Band 1 genau die Eingabe, also $O(n)$ Bits. Später enthält Band 1 höchstens $O(n + t(n))$ Bits, da jedes erzeugte Bit auch einen der maximal $t(n)$ vielen Rechenschritte kostete.

Die Turingmaschine M „merkt“ sich in ihrem Zustand, welche Programmzeile j von R als nächstes auszuführen ist. Sie schreibt die zugehörigen Registerinhalte $c(i_1)$, $c(i_2)$, …, $c(i_k)$ von Band 1 auf Band 2, führt darauf die Operation op_j aus und schreibt schließlich das Ergebnis in Zelle i_b auf Band 2 zurück. Existiert für i_b noch kein Eintrag auf Band 1, muss dieser erst erschaffen werden.

Betragen für ein Polynom g_1 die Kosten für das Ausführen einer Programmzeile $O(g_1(n + t(n)))$, so lassen sich die Gesamtkosten (bei $\leq t(n)$ auszuführenden Programmzeilen) polynomiell durch $g_2(n + t(n)) := t(n) \cdot g_1(n + t(n))$ abschätzen. Da wir eine 2-Band DTM durch eine 1-Band DTM nur durch quadratischen Mehraufwand simulieren können, erhalten wir insgesamt folgendes Ergebnis:

Theorem 11. *Eine RAM mit Laufzeit $t(n)$ im logarithmischen Kostenmaß lässt sich durch eine 1-Band-DTM mit Laufzeit $O(g(n + t(n)))$ simulieren. Dabei hängt das Polynom g von der RAM ab.*

Umgekehrt lässt sich aber auch eine DTM durch eine RAM simulieren.

Theorem 12. *Eine 1-Band-DTM mit Laufzeit $t(n)$ lässt sich auf einer RAM im logarithmischen Kostenmaß in Zeit $O((n + t(n)) \log(n + t(n)))$ simulieren.*

Beweis. (Skizze)

- Speichere Zustand, Kopfstellung und Inhalte aller jemals besuchten Felder der DTM in je einer Speicherzelle der RAM.
- Finde mit IF-Tests auf Zustand und Feldinhalt unterm Kopf heraus und welche Instruktion die DTM ausführen würde.

- Aktualisiere entsprechend die Inhalte der Zellen, Zustand, Kopfstellung und Bandinhalt.

Zu klären ist noch, woher der Faktor $\log(n + t(n))$ kommt. Die DTM kann $n + t(n)$ Felder auf ihrem Band besuchen. Die Indizes der entsprechenden Zellen der RAM haben also die Länge $\log(n + t(n))$. □

3.2 Entscheidungs- und Optimierungsprobleme

Berechenbarkeitsfragen haben wir formal durch die Entscheidbarkeit von Sprachen durch Maschinen beantwortet. Dabei wurde durch die Maschinen entschieden, ob ein bestimmtes Wort zu einer Sprache gehört oder nicht. Wir wollen hier motivieren, dass es auch bei der Einteilung von Problemen in Berechnungskomplexitätsklassen im Wesentlichen darum geht, die Berechnungskomplexität von Entscheidungsproblemen zu untersuchen. Andere Problemstellungen lassen sich daraus leicht ableiten.

In der Praxis ist man nicht nur an ja/nein Antworten interessiert, sondern auch einer optimalen Lösung eines Optimierungsproblems. Wir wollen hier am Beispiel des Travelling Salesperson Problem (TSP) zeigen, dass diese unterschiedlichen Problemstellungen aufeinander *reduziert* werden können. Effiziente Algorithmen für ein Entscheidungsproblem garantieren effiziente Algorithmen für das Optimierungsproblem, da die Reduktionen in der Laufzeit beschränkt sind.

Zu beachten ist, dass wir formal definierte Reduktionen bislang zwischen Sprachen betrachtet haben. Das werden wir auch so beibehalten und deshalb bei formalen Reduktionen stets nur über Entscheidungsprobleme argumentieren. Hier soll nur gezeigt werden, dass die Betrachtung von Entscheidungsproblemen keine Einschränkung darstellt, da sich dann auch Optimierungsprobleme lösen lassen. Die Reduktion ist daher hier eine direkte *Unterprogrammtechnik*.

Travelling Salesperson Problem: Gegeben sind n Orte $\{s_1, s_2, \ldots, s_n\}$ und Kosten $c_{ij} \in \mathbb{N}$ für die Reise von s_i nach s_j. Gesucht wird eine kostengünstigste Rundreise (jeder Ort wird genau einmal besucht). Die Eingabe kann ein Graph $G = (V, E)$ sein mit einer Kostenfunktion für die Kanten.

Natürlich lässt sich diese Eingabe binär kodieren und es stellt sich dann die Frage, ob das zugehörige Wort zu einer bestimmten Sprache gehört. Die wesentliche Aufgabe wird es sein, das folgende Entscheidungsproblem zu lösen. Dazu wird entschieden, ob ein Eingabewort der entsprechende Sprache angehört.

TSP-Entscheidungsproblem: Gegeben $G = (V, E)$ mit Kantengewichtsfunktion $c : E \to \mathbb{N}$ und $t \in \mathbb{N}$.

Frage: Gibt es eine Rundreise mit Gesamtkosten $\leq t$?

Falls es eine Turingmaschine M oder eine RAM gibt, die dieses Problem effizient entscheidet, dann lassen sich auch Maschinen beschreiben, die das folgende Optimierungsproblem lösen. Wir verwenden dazu ein kleines Programm im Pseudocode einer höheren Programmiersprache.

TSP-Optimierungsproblem: Gegeben $G = (V, E)$ mit Kantengewichtsfunktion $c : E \to \mathbb{N}$.
Problem: Bestimme die Kosten der günstigsten Rundreise.

Dieses Problem kann wie folgt auf das Entscheidungsproblem reduziert werden. Wir berechnen $k := \sum_{e \in E} c(e)$ und benutzen binäre Suche.

```
1: Min := 0; Max := k;
2: while Max − Min ≥ 1 do
3:    t := ⌊(Min+Max)/2⌋
4:    Löse Entscheidungsproblem mit  t;
5:    if Antwort JA then
6:       Max := t;
7:    else
8:       Min := t + 1;
9:    end if
10: end while
```

Zur Lösung des Optimierungsproblems müssen somit nur $\lceil \log k \rceil$ viele TSP-Entscheidungsprobleme gelöst werden. Da $\log k \leq \sum_{e \in E} \log c(e)$ gilt, ist $\log k$ durch die Eingabgröße beschränkt. Falls ein polynomieller Algorithmus für das TSP-Entscheidungsproblem existiert, dann gibt es auch einen polynomiellen Algorithmus für das TSP-Optimierungsproblem.

Natürlich ist man nicht nur an der Länge der optimalen Lösung, sondern auch am optimalen Rundweg selbst interessiert. Das führt zu folgender funktionalen Problembeschreibung:

Funktionales TSP-Optimierungsproblem: Gegeben $G = (V, E)$ mit Kantengewichtsfunktion $c : E \to \mathbb{N}$.
Problem: Bestimme eine Rundtour mit minimalen Kosten.

Auch dieses Problem können wir auf das TSP-Entscheidungsproblem reduzieren. Zunächst verwenden wir das TSP-Optimierungsproblem, um die minimalen Kosten t_0 zu bestimmen.

```
1: Bestimme optimale Kosten t_0;
2: for alle e ∈ E do
3:    c(e) := c(e) + 1;
4:    Bestimme optimale Kosten t'_0;
5:    if t'_o > t_0 then
6:       c(e) := c(e) − 1; {Kante e wird gebraucht}
7:    end if
```

8: **end for**

Nach der Terminierung sind genau die Kanten, die zu einer optimalen Rundtour gehören nicht erhöht worden. Maximal $(n^2 + 1)$ mal wurde das TSP-Optimierungsproblem aufgerufen. Falls ein polynomieller Algorithmus für das TSP-Entscheidungsproblem existiert, dann gibt es auch einen polynomiellen Algorithmus für das funktionale TSP-Optimierungsproblem.

Im Folgenden gehen wir davon aus, dass sich alle funktionalen Berechnungsprobleme durch entsprechende Entscheidungsprobleme lösen lassen und eine polynomielle Laufzeit nur vom jeweiligen Entscheidungsproblem abhängt. Die (optimale) Laufzeit hängt natürlich auch von der Reduktion selbst ab, uns geht es aber nicht um optimale Laufzeiten, sondern um die Größenordnung der Laufzeiten.

3.3 Zeitkomplexität von Turingmaschinen

Zunächst präzisieren wir nochmal den Laufzeitbegriff, verwenden dazu Turingmaschinen und gehen davon aus, dass eine Sprache L von einer Maschine M entschieden werden kann. Die Maschine hält also bei jeder Eingabe!

Für ein Wort $w \in \Sigma^*$ sei mit $T_M(w)$ die Anzahl der Rechenschritte von M bei Eingabe w definiert. Für $n \in \mathbb{N}$ sei dann

$$T_M(n) := \max\{T_M(w) | w \in \Sigma^*, |w| \leq n\}$$

und die Funktion $T_M : \mathbb{N} \to \mathbb{N}$ heißt die *Laufzeit oder Zeitkomplexität* von M.

Durch die obige Definition ist automatisch sichergestellt, dass T_M eine monoton wachsende Funktion ist. Wir sagen auch, dass die DTM M eine *Laufzeit oder Zeitkomplexität* $O(f(n))$ hat, falls $T_M(n) \in O(f(n))$ liegt.

Betrachten wir zum Beispiel eine Turingmaschine, die die Sprache $L = \{0^k 1^k | k \geq 1\}$ entscheidet. Wir verzichten im Folgenden auf die genaue Maschinenbeschreibung bzw. auf Implementierungsdetails und erläutern systematisch, was die Maschine tut.

1. Durchlaufe die Eingabe. Falls eine Null *nach* einer Eins auftaucht, lehne ab.
2. Wiederhole die folgenden Schritte, solange noch eine Eins *und* eine Null auf dem Band stehen.
3. Durchlaufe das Band von links nach rechts, lösche dabei die letzte Eins und die erste Null. Gehe nach rechts zum letzten Zeichen.

4. Falls alle Zahlen gelöscht wurden, akzeptiere die Eingabe. Falls nur noch Einsen und keine Null mehr auf dem Band steht oder falls nur noch Nullen und keine Eins mehr auf dem Band steht, lehne das Wort ab.

Für die Laufzeitanalyse ist zunächst klar, dass im 1. Schritt in $O(n)$ geprüft wird, ob w mit $|w| = n$ von der Form 0^i1^j ist. Danach wird in jedem Schritt 3. das aktuelle Wort vollständig durchlaufen und um zwei Zahlen gekürzt. Jeder dieser Schritte benötigt maximal $O(n)$ Schritte. Da aber in jedem Schritt 2 Zahlen *außen* gelöscht werden kann Schritt 3. maximal $n/2$ mal ausgeführt werden. Danach kann die Eingabe ggf. akzeptiert werden oder die Eingabe wurde bereits vorher verworfen. Die Laufzeit beträgt also maximal $O(n) + O(n^2) = O(n^2)$. Die Sprache L ist also in quadratischer Zeit entscheidbar.

Definition 10. Sei $t : \mathbb{N} \to \mathbb{N}$ eine monoton wachsende Funktion. Die Klasse DTIME($t(n)$) ist definiert als:

$$\text{DTIME}(t(n)) := \left\{ L \;\middle|\; \begin{array}{l} L \text{ ist eine Sprache, die von einer DTM } M \\ \text{in Zeit } O(t(n)) \text{ entschieden wird.} \end{array} \right\}.$$

Wir haben uns hier auf Sprachen beschränkt, über funktionale Optimierungsprobleme machen wir aber ohne Einschränkung ähnliche Aussagen.

Die obige Betrachtung zeigt nun $L \in \text{DTIME}(n^2)$. Die folgende informelle Beschreibung einer DTM besagt, dass L sogar mit geringerer Zeitkomplexität entschieden werden kann.

Beschreibung einer DTM M:

1. Durchlaufe die Eingabe. Falls eine Null *nach* einer Eins auftaucht, lehne ab. Sonst gehe zurück zum Anfang des Bandes.

2. Durchlaufe das Band von rechts nach links und stelle dabei fest, ob die Anzahl Einsen und Nullen beide gerade oder beide ungerade sind. Lehne ab, falls dies nicht der Fall ist. Sonst gehe zurück zum Beginn der Eingabe.

3. Wiederhole den folgenden Schritt, solange noch eine Eins *und* eine Null auf dem Band stehen.

4. Durchlaufe das Band und streiche jede zweite Null und jede zweite Eins, beginnend mit der ersten Null und der ersten Eins.

5. Falls nur noch Einsen und keine Null mehr auf dem Band steht oder falls nur noch Nullen und keine Eins mehr auf dem Band steht, lehne das Wort ab.

Bei dieser Programmabarbeitung ist es so, dass Schritt 3. und 4. wiederum $O(n)$ Aufwand bedeuten, aber insgesamt nur $O(\log n)$ mal aufgerufen werden. In jedem Schritt wird die Hälfte der Zahlen gestrichen. Es gilt also $L \in \text{DTIME}(n \log n)$.

3.4 Die Klasse *P*

Lemma 7 und die Theoreme 11 bzw. 12 zeigen, dass die folgende Definition effizient lösbarer Probleme unabhängig vom jeweiligen Maschinenmodell (1-Band DTM, Mehrband DTM, RAM) betrachtet werden kann.

Definition 11. Die Klasse P ist definiert als

$$P = \bigcup_{k \in \mathbb{N}} \text{DTIME}(n^k).$$

Die Klasse P ist also die Menge aller Sprachen, für die es ein beliebiges, aber festes k gibt, sodass eine DTM M existiert, die die Sprache L in Laufzeit $O(n^k)$ entscheidet. Der Parameter k darf dabei von einem Problem, jedoch nicht von der Größe einer Instanz abhängen. Die Klasseneinteilung ist sinnvoll, da es sich um Problemstellungen handelt,

- für die wir eine konkrete polynomielle Abschätzung vornehmen und
- die aus praktischer Sicht effizient lösbar sind.

Für die effiziente Lösbarkeit aus praktischer Sicht ist es notwendig, dass der zugehörige konstante Exponent k nicht zu groß wird. $k = 1000$ ist zum Beispiel aus praktischer Sicht für große n definitiv nicht mehr effizient. Es zeigt sich aber aus Erfahrung, dass für viele klassische Problemstellungen auch relativ kleine Konstanten ($k = 1, 2, 3, \ldots$) gefunden werden können. Beispiele für (funktionale) Problemstellungen aus P kennen wir bereits aus dem vergangenen Semester:

- Sortieren von Zahlen
- Berechnen eines minimalen Spannbaumes
- Maximaler Fluss in einem Netzwerk
- Zusammenhangskomponenten eines Graphen
- Tiefensuche und Breitensuche
- Kürzeste Wege im Graphen
- $\cdots$

3.5 Nichtdeterministische Turingmaschinen

Es gibt Problemstellungen, die vermutlich nicht in P liegen. Sei beispielsweise $G = (V, E)$ ein ungerichteter Graph mit Knotenmenge $V = \{1, 2, \ldots, n\}$, den wir durch eine Adjazenzmatrix binär kodieren können. Eine k-Clique im Graphen ist eine Teilmenge $V' \subseteq V$ mit $|V'| = k$, sodass $(i, j) \in E$ für alle $i, j \in V'$ mit $i \neq j$ gilt. Also ein Teilmenge von Knoten, sodass diese allesamt untereinander mit Kanten verbunden sind. Anders ausgedrückt suchen wir nach einer Permutation π der Knotenmenge, sodass in der Adjazenzmatrix eine $k \times k$-Submatrix entsteht, die nur mit Einsen gefüllt ist. Das Problem sei durch die folgende Sprache definiert:

$$\text{Clique} := \left\{ \langle V, E, k \rangle \;\middle|\; \begin{array}{l} G = (V, E) \text{ ist gerichteter Graph und } k \in \mathbb{N}, \\ \text{sodass } G \text{ eine } k - \text{Clique enthält} \end{array} \right\}.$$

Es ist kein polynomieller Algorithmus bekannt, der die Sprache Clique entscheidet. Falls wir aber einen Kandidaten von k Knoten haben, für den wir die Cliqueneigenschaft überprüfen wollen, dann geht das sehr wohl in polynomieller Zeit. Nehmen wir also an, wir hätten einen Mechanismus, der für uns den richtigen Beweiskandidaten *rät*. Dann können wir das Problem in polynomieller Zeit lösen. Das führt zur Definition der nichtdeterministischen Turingmaschinen NTM.

Definition 12. Eine nichtdeterministische k-Band Turingmaschine (NTM) ist ein 5-Tupel $N = (\Sigma, Q, \delta, q_0, F)$, wobei Σ, Q, q_0 und F genauso definiert sind wie bei der DTM. Die Zustandsübergangsfunktion sei von der Form

$$\delta : Q \times \Sigma^k \to P(Q \times \Sigma^k \times \{-1, 1, 0\}^k),$$

wobei $P(Q \times \Sigma^k \times \{-1, 1, 0\}^k)$ die Potenzmenge, also die Menge aller Teilmengen der Menge von $Q \times \Sigma^k \times \{-1, 1, 0\}^k$ bezeichnet.

Für eine 1-Band NTM ist die Interpretation eines Tupels der Form $\delta(q, a) = \{(q_1, c_1, b_1), (q_2, c_2, b_2), \ldots, (q_l, c_l, b_l)\}$ so, dass die Maschine nach dem Lesen des Buchstaben a im Zustand q in eine der Folgekonfigurationen übergehen darf. Der Übergang ist nicht deterministisch festgelegt, die Übergangsfunktion kann auch als Relation betrachtet werden.

Wir betrachten das folgende Beispiel einer NTM N. Es sei q_1 der Startzustand und q_3 der Endzustand.

δ	0	1	$\sqcup$
q_1	$(q_1, 0, 1), (q_3, 0, 0)$	$(q_2, 1, 1)$	$(q_3, 0, 0)$
q_2	$(q_1, 0, 1), (q_3, 0, 0)$	$(q_3, 1, -1)$	$(q_3, 0, 0)$

Konfigurationsänderungen werden nun nicht als einfache Ketten dargestellt, sondern durch Bäume, die den Nichtdeterminismus beschreiben. Wir sprechen dann also von einem *Berechnungsbaum*. Beispielsweise ergibt sich ein Berechungsbaum für die obige Maschine N bei einem Wort $w = 0011$ wie in Abbildung 3.1 gezeigt.

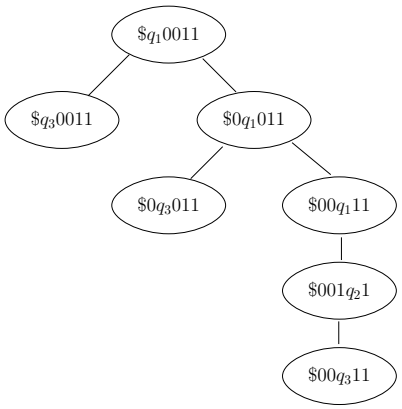

Abbildung 3.1: Der Berechnungsbaum der Maschine N.

Der Baum enthält ablehnende und akzeptierende Blätter. Bezüglich des Akzeptanzverhaltens reicht ein akzeptierender Pfad aus.

Definition 13. Eine NTM N akzeptiert die Eingabe $x \in \Sigma^*$, falls es mindestens eine Berechnung gibt, die in eine akzeptierende Endkonfiguration führt.

Die von N akzeptierte Sprache $L(N)$ ist definiert durch

$$L(N) := \{w \in \Sigma^* | N \text{ akzeptiert } w\}.$$

Die NTM N akzeptiert die Sprache L falls $L = L(N)$. Die NTM N entscheidet L, falls N immer hält und N die Sprache L akzeptiert. Wörter, die nicht in L liegen, werden verworfen.

Für Laufzeitabschätzungen der NTM berücksichtigen wir nur die Eingaben $w \in L(N)$. Wir gehen davon aus, dass sich die NTM immer den kürzesten akzeptierenden Pfad wählt, die Maschine *rät* somit den besten akzeptierenden Pfad.

Definition 14. Sei N eine NTM. Die Laufzeit $T_N(w)$ von N bei einer Eingabe $w \in \Sigma^*$ ist definiert durch:

$$T_N(w) := \begin{cases} \text{Länge des kürzesten akzept. Pfades,} & \text{falls } w \in L(N) \\ 0 & \text{sonst.} \end{cases}$$

Für $n \in \mathbb{N}$ ist dann

$$T_N(n) := \max\{T_N(w) | w \in \Sigma^*, |w| \leq n\}$$

und die Funktion $T_N : \mathbb{N} \to \mathbb{N}$ heißt Laufzeit oder Zeitkomplexität von N.

Durch die obige Definition ist automatisch sichergestellt, dass T_N eine monoton wachsende Funktion ist. Wir sagen auch, dass die NTM N eine *Laufzeit oder Zeitkomplexität* $O(f(n))$ hat, falls $T_N(n) \in O(f(n))$ liegt. Nun lässt sich analog zu den DTMs die Klasse NTIME($t(n)$) definieren.

Definition 15. Sei $t : \mathbb{N} \to \mathbb{N}$ eine monoton wachsende Funktion. Die Klasse NTIME($t(n)$ ist definiert als:

$$\text{NTIME}(t(n)) := \left\{ L \;\middle|\; \begin{array}{l} L \text{ ist eine Sprache, die von einer NTM } N \\ \text{in Zeit } O(t(n)) \text{ akzeptiert wird.} \end{array} \right\}.$$

Übungsaufgabe: Welche Sprache entscheidet die NTM N aus dem obigen Beispiel und welche Zeitkomplexität hat die Maschine?

3.6 Die Klasse *NP*

Erinnern wir uns an die Ausgangssituation der Überprüfung von Beweiskandidaten. Eine Maschine kann nichtdeterministisch einen beliebigen Beweiskandidaten erzeugen, den wir dann nur noch effizient überprüfen müssen. Problemstellungen in denen das insgesamt effizient möglich ist, fassen wir in der folgenden Klasse zusammen.

Definition 16. Die Klasse *NP* ist definiert als

$$NP = \bigcup_{k \in \mathbb{N}} \text{NTIME}(n^k).$$

Die Klasse *NP* ist also die Menge aller Sprachen, für die es ein beliebiges aber festes k gibt, sodass eine NTM N existiert, die die Sprache L in Laufzeit $O(n^k)$ akzeptiert. *NP* steht hierbei für *nichtdeterministisch polynomiell*. Als erstes wollen wir zeigen, dass Clique ein Problem der Klasse *NP* ist.

Lemma 14. *Clique* $\in$ *NP.*

Beweis. Wir beschreiben eine NTM N mit $L(N) =$ Clique. Falls $w = \langle V, E, k \rangle$ nicht dem Eingabeformat (kein Graph, keine entsprechende Zahl) entspricht, verwirft die Maschine. Ansonsten arbeitet die Maschine wie folgt.

1. Für $G = (V, E)$ sei n die Anzahl der Knoten, also $V = \{1, 2, \ldots, n\}$. Wir schreiben das Wort $\#^n$ hinter die Eingabe, der Kopf bewegt sich auf das erste # Zeichen.

2. Die Maschine läuft von links nach rechts und ersetzt den String $\#^n$ nichtdeterministisch durch einen String aus $\{0, 1\}^n$. Dadurch lassen sich alle Strings aus $\{0, 1\}^n$ in Zeit $O(n)$ erzeugen. Sei $x = (x_1, x_2, \ldots, x_n)$ der erzeugte String.

3. Betrachte $V_x = \{i \in V | x_i = 1\} \subseteq V$. Überprüfe, ob es sich um eine $k-$Clique handelt. Es wird getestet, ob es zwischen allen Knotenpaaren aus V_x Kanten in E gibt. Akzeptiere, falls es so ist, sonst verwerfe die Eingabe.

Offensichtlich ist $L(N) =$ Clique. Genau wenn G eine $k-$Clique enthält, dann existiert ein String $x = (x_1, x_2, \ldots, x_L)$ sodass V_x eine $k-$Clique darstellt. Die Laufzeit ist polynomiell in der Eingabe beschränkt. Schritt 1. und das Überprüfen der Eingabe können in $O(|E| + |V|)$ durchgeführt werden. Schritt 3. benötigt für jedes Knotenpaar V_x $O(|E| + |V|) = O(|E|)$ Zeit zur Überprüfung, insgesamt als $O(|V|^2|E|) = O(|E|^2)$ Zeit. Jeder nichtdeterministische Berechnungspfad aus Schritt 2. hat Länge $|V|$. Das Problem lässt sich nichtdeterministisch polynomiell lösen. □

Das Problem der Clique haben wir mit Hilfe des Nichtdeterminismus gelöst. Dazu wurde der richtige Beweiskandidat geraten. Ein generierter Beweiskandidat kann also in diesem Sinn auch als *Zertifikat* bezeichnet werden, welches wir in polynomieller Zeit mit der jeweiligen Maschine überprüfen können. Statt zu raten können auch alle $O(n^k)$ möglichen Beweiskandidaten nacheinander ausprobiert werden. Allerdings ist k variabel und die Laufzeit dieses Brute-Force Algorithmus somit nicht polynomiell.

Diese Sicht der Dinge werden wir im Folgenden präzisieren. Vorher formulieren wir noch einige Beispiele für Sprachen bzw. Problemstellungen, die in *NP* liegen.

3.6.1 Beispiele für Probleme aus *NP*

Im vergangenen Semester haben wir bereits das Rucksackproblem kennengelernt und verschiedene Algorithmen dazu betrachtet.

Rucksackproblem: Gegeben ist eine Menge $\{a_1, \ldots a_n\}$ von n Gegenständen. Jeder Gegenstand a_i besitzt ein Gewicht g_i und einen Wert w_i.
Gesucht ist eine bezüglich einer Gesamtgewichtskapazität G zulässige Lösung mit maximalen Wert, also eine Teilmenge A der Gegenstände mit $\sum_{a_i \in A} g_i \leq G$ mit maximalen Wert $\sum_{a_i \in A} w_i$ unter allen Teilmengen.

Für Gegenstände mit ganzzahligen Gewichten haben wir bereits einen Algorithmus kennengelernt, der eine optimale Lösung berechnet. Der Algorithmus verwendet dynamischer Programmierung, Laufzeit und Speicherplatz sind durch $O(n \cdot G)$ beschränkt. Die Gesamtgewichtskapazität G wird in der Eingabe durch $\log_2 G$ viele Bits dargestellt. Mit einer zusätzlichen Stelle in der Länge der Eingabe von G können doppelt so viele Zahlen dargestellt werden. Der Algorithmus ist also nicht polynomiell in der Eingabe G.

Das Rucksackproblem kann auch als Entscheidungsproblem definiert werden. Dabei wird entschieden, ob es für die Eingabe eine Lösung einer bestimmten Güte gibt oder nicht. Durch die Lösung des Entscheidungsproblems kann auch eine Lösung des Optimierungsproblems oder des obigen funktionalen Optimierungsproblems gefunden werden. Der *zusätzliche* Aufwand ist polynomiell beschränkt.

Hamiltonkreisproblem: Gegeben ist ein ungerichteter Graph $G = (V, E)$. Frage: Gibt es eine Rundtour in G, sodass jeder Knoten genau einmal besucht wird?

Interessanterweise liegt dieses Problem in *NP*, während das Eulerkreisproblem sehr leicht zu lösen ist. Ein Eulerkreis, d.h. eine Rundtour die jede Kante genau einmal besucht, existiert nämlich genau dann, wenn der Graph zusammenhängend ist und jeder Knoten einen geraden Kantengrad besitzt. Dieses Problem lässt sich leicht mit $O(n)$ Aufwand testen.

Ein sehr wichtiges Problem stammt aus dem Bereich der Aussagenlogik, die in der Vorlesung Logik und diskrete Strukturen behandelt wurde. Dazu sei $V = \{x_1, x_2, \ldots\}$ eine unendliche Menge aussagenlogischer *Variablen*, die den Wert 1 (wahr, true) oder 0 (nicht wahr, false) annehmen können.

Eine aussagenlogische Formel (oder auch aussagenlogischer Ausdruck) kann nun wie folgt als Konjunktion (Und-Verbindung) von Disjunktionen (Oder-Verbindung) definiert werden.

- Für $x_i \in V$ seien x_i und $\neg x_i$ *Literale*. Jedes Literal y_i ist eine aussagenlogische Formel.
- Seien $y_1, y_2, \ldots y_k$ Literale so ist der Ausdruck $(y_1 \vee y_2 \vee \cdots \vee y_k)$ eine *Klausel* vom Grad k und eine aussagenlogische Formel.
- Seien $k_1, k_2, \ldots, k_l$ Klauseln vom Grad $\leq k$, dann ist der Ausdruck $k_1 \wedge k_2 \wedge \cdots \wedge k_l$ eine aussagenlogische Formel in *konjunktiver Normalform* mit höchstens k Literalen pro Klausel.

Eine Belegung der in einer Formel in konjunktiver Normalform verwendeten Variablen ist eine Zuordnung dieser Variablen x_i in die Wahrheitswerte 0 oder 1.

Für eine gegebene Belegung gilt folgende Interpretation:

- Sei $x_i = 1 (x_i = 0)$ dann ist das Literal x_i true (false) und das Literal $\neg x_i$ false (true).
- Für Literale $y_1, y_2, \ldots y_k$ ist die Klausel $(y_1 \vee y_2 \vee \cdots \vee y_k)$ true, falls mindestens ein Literal für die Belegung true ist. Sonst ist die Klausel false.
- Der aussagenlogische Ausdruck $k_1 \wedge k_2 \wedge \cdots \wedge k_l$ für Klausel $k_1, k_2, \ldots, k_l$ ist true, falls alle Klauseln true sind. Sonst ist der Ausdruck false.

SAT (Satisfiability) Gegeben ein aussagenlogischer Ausdruck $k_1 \wedge k_2 \wedge \cdots \wedge k_l$ mit Klauseln vom Grad $\leq k$ in konjunktiver Normalform mit insgesamt m verwendeten Variablen $x_1, x_2, \ldots, x_m$.
Frage: Gibt es eine Belegung der Variablen, sodass der Ausdruck $k_1 \wedge k_2 \wedge \cdots \wedge k_l$ true ist?

Im obigen Problem kann für jedes Wort, das zu entscheiden ist, der Grad k verschieden sein. Wenn wir k vorab festlegen, ist das ein interessanter Spezialfall des Problems.

k-SAT Gegeben ein aussagenlogischer Ausdruck $k_1 \wedge k_2 \wedge \cdots \wedge k_l$ mit Klauseln vom Grad $\leq k$ mit fest vorgegebenen k in konjunktiver Normalform mit insgesamt m verwendeten Variablen $x_1, x_2, \ldots, x_m$.
Frage: Gibt es eine Belegung der Variablen, sodass der Ausdruck $k_1 \wedge k_2 \wedge \cdots \wedge k_l$ true ist?

Ein weiterer Spezialfall ist:

Max-k-SAT Gegeben ein aussagenlogischer Ausdruck $\alpha = k_1 \wedge k_2 \wedge \cdots \wedge k_\ell$ mit Klauseln vom Grad $\leq k$ in konjunktiver Normalform mit insgesamt m verwendeten Variablen $x_1, x_2, \ldots, x_m$, und eine Zahl t.
Frage: Gibt es eine Belegung der Variablen, sodass mindestens t Klauseln aus $k_1, k_2, \ldots, k_l$ true sind?

Beim funktionalen Optimierungsproblem von Max-k-SAT wird eine Belegung der Variablen gesucht, sodass die maximale Anzahl an Klauseln aus $k_1, k_2, \ldots, k_l$ true ist.

Übungsaufgabe: Zeigen Sie, dass 2-SAT in P liegt.

Set-Cover Gegeben ist eine Menge von Elementen $S = \{e_1, e_2, \ldots e_m\}$ und eine Menge $T = \{T_1, T_2, \ldots, T_k\}$ von Teilmengen T_i mit Elementen aus S.
Gesucht: Eine Teilmenge aus T, die exakt die Menge die Menge S abdeckt,

also eine Menge $T' = \{T_{i_1}, T_{i_2}, \dots T_{i_l}\} \subseteq T$ mit für jedes $e_j \in S$ existiert genau ein $T_{i_j} \in T'$ mit $e_j \in T_{i_j}$.

Auch für Set-Cover gibt es Spezialfälle:

3-Exakt-Cover Gegeben ist eine Menge $S = \{e_1, e_2, \dots e_m\}$ mit $m = 3n$ und eine Menge $T = \{T_1, T_2, \dots, T_k\}$ von 3-elementigen Teilmengen T_i.
Gesucht: Eine Teilmenge aus T, die exakt die Menge die Menge S abdeckt, also eine Menge $T' = \{T_{i_1}, T_{i_2}, \dots T_{i_l}\} \subseteq T$ mit folgender Eigenschaft: für jedes $e_j \in S$ existiert genau ein $T_{i_j} \in T'$ mit $e_j \in T_{i_j}$.

Knotenüberdeckung (Vertex-Cover) Geben ist ein Graph $G = (V, E)$ und eine Zahl $k \in \mathbb{N}$.
Frage: Gibt es eine Kontenmenge $V' \subseteq V$ mit $|V'| = k$, sodass für jede Kante $(v, w) \in E$ gilt: $v \in V'$ oder $w \in V'$ (oder beide)?

Knotenfärbung (Coloring) Gegeben ist ein Graph $G = (V, E)$ und eine Zahl $k \in \{1, 2, \dots, |V|\}$.
Frage: Gibt es eine *Färbung* $c : V \to \{1, 2, \dots, k\}$ der Knoten von V mit k *Farben*, sodass benachbarte Knoten verschiedene Farben haben, d.h., für alle $(v, w) \in E$ gilt $c(v) \neq c(w)$?

Probleme aus *NP* können auch explizit etwas mit Zahlenberechnungen zu tun haben. Als Beispiele gelten die folgenen Problemstellungen:

Partition Gegeben ist eine Menge $\{a_1, a_2, \dots, a_N\}$ von N natürlichen Zahlen mit $a_i \in \mathbb{N}$.
Frage: Gibt es eine Teilmenge T, sodass $\sum_{i \in T} a_i = \sum_{j \in \{1,2,\dots,N\} \setminus T} a_j$ gilt, also eine Aufteilung der Zahlen, sodass die Summen identisch sind?

Subset-Sum Gegeben ist eine Menge $\{a_1, a_2, \dots, a_N\}$ aus N natürlichen Zahlen mit $a_i \in \mathbb{N}$ und ein $b \in \mathbb{N}$.
Frage: Gibt es eine Teilmenge $T \subseteq \{1, 2, \dots, N\}$, sodass $\sum_{i \in T} a_i = b$ gilt?

An zwei Beispielen zeigen wir exemplarisch, warum die beschriebenen Probleme in *NP* liegen.

Lemma 15. *SAT liegt in NP.*

Beweis. Eine NTM mit der Eingabe $k_1 \wedge k_2 \wedge \dots \wedge k_l$ mit Klauseln vom Grad $\leq k$ und insgesamt m Variablen kann zunächst eine Belegung durch den Nichtdeterminismus in Laufzeit $O(m)$ raten.

Danach werden sukzessive die Klauseln überprüft. In jeder Klausel wird jede Variable abgefragt. Man könnte für diese Überprüfung beispielsweise eine 2-Band DTM verwenden, die auf dem zweiten Band die geratene Belegung enthält und auf dem ersten Band die Eingabe. Sowohl die Belegung als auch die Eingabe wird zur Kodierung der Variablenindizes zusätzlich einen log-Faktor benötigen.

Für jede Variable, die auf dem ersten Band besucht wird, wird im zweiten Band die Belegung durchsucht. Falls eine Klausel nicht erfüllt ist, wird abgelehnt. Falls alle Klauseln erfüllt sind, wird akzeptiert.

Die Laufzeit beträgt $O(m \cdot (k \cdot l))$ mit $m \leq k \cdot l$ und ist polynomiell durch die Eingabegröße $C \cdot k \cdot l$ beschränkt. Berücksichtigen wir die Länge der Kodierung der Variablenindizes, so wird ein zusätzlicher $\log^2$ Faktor benötigt. Die Laufzeit ist aber polynomiell. □

Lemma 16. *Subset-Sum liegt in NP.*

Beweis. Mit einer NTM können wir die richtige Teilmenge T in maximal N Schritten raten. Anschließend addieren wir die Elemente in deterministischer Weise, beispielsweise durch eine RAM auf. Im uniformen Kostenmaß geht das auch in Zeit $O(N)$. Eine weitere Operation und ein Vergleich mit b liefert die Antwort. Im logarithmischen Kostenmaß muss $l := \max_{\{1,2,\ldots,N\}} \log(a_i)$ betrachtet werden. Jeder Rechenschritt der RAM wird in $O(l)$ ausgeführt. Da $n = \sum_{\{1,2,\ldots,N\}} \log(a_i) + \log b \geq l$ die Eingabegröße ist, wird insgesamt eine Laufzeit von $C \cdot l \cdot N \in O(n^2)$ erzielt. □

3.7 Klasse *NP* durch Zertifikate charakterisieren

Bei der Einführung der Klasse *NP* hatten wir bereits über das Verifizieren von Beweiskandidaten gesprochen und in den eben geführten Beweisen, haben wir stets zunächst einen Kandidaten geraten und diesen dann deterministisch überprüft. Diese Beweisidee lässt sich schön verallgemeinern, ein klassisches Prinzip in der Theorie.

Definition 17. Für eine Sprache L heißt eine DTM V *polynomieller Verifizierer* von L, falls es Polynome p und q gibt mit

$$\begin{aligned} x \in L \quad \Leftrightarrow \quad & \exists y \in \{0,1\}^* \text{ mit } |y| \leq p(|x|) : \\ & V \text{ akzeptiert } y\#x \text{ in Laufzeit } q(|y\#x|). \end{aligned}$$

Die Sprache L heißt in diesem Fall *polynomiell verifizierbar*. Die geratenen Bits y werden dabei als Input übergeben.

Lemma 17. *Eine Sprache L liegt genau dann in NP, falls die Sprache polynomiell verifizierbar ist.*

Beweis. Sei $L \in NP$. Dann gibt es eine NTM N, die $x \in L$ mit $|x| = n$ in polynomieller Zeit $p(n)$ akzeptiert. Wir verändern die nichtdeterministische Übergangsfunktion so, dass immer maximal zwei nichtdeterministische Übergänge existieren. Die beiden alternativen Übergänge bezeichnen wir dann respektive mit 0 oder 1. (Die Anzahl der Übergänge ist allgemein stets durch die konstante Anzahl von Elementen aus Σ und Q beschränkt, deshalb lässt sich diese Betrachtung entsprechend ohne Einschränkung realisieren. Der Verzweigungsgrad der Maschine ist durch eine Konstante k beschränkt und hierfür würden $\log k$ Bits reichen. Die unten angesprochene Länge des Zertifikats y und die Laufzeit von V bleiben polynomiell.)

Bei einer Eingabe x mit $|x| = n$ verwenden wir ein Zertifikat $y \in \{0,1\}^{p(n)}$mit $|y| \leq p(|x|)$. Der zu erstellende Verifizierer V erhält als Eingabe $y\#x$ und simuliert einen Rechenweg deterministisch für die Starteingabe x, indem im i-ten Übergang der Übergang y_i auf x angewendet wird. Für jeden Simulationsschritt muss die Eingabe $|y\#x|$ konstant oft durchlaufen werden. Deshalb kann der Verifizierer V die Eingabe $y\#x$ in polynomieller Zeit $q(|y\#x|)$ akzeptieren:

$$\begin{aligned} x \in L &\Leftrightarrow N \text{ akzeptiert } x \text{ in polynomieller Zeit} \\ &\Leftrightarrow \exists y \in \{0,1\}^* \text{ mit } |y| \leq p(|x|) : \\ &\quad\ V \text{ akzeptiert } y\#x \text{ in Laufzeit } q(|y\#x|). \end{aligned}$$

Umgekehrt existiere ein Verifizierer V von L mit der angegebenen Eigenschaft. Wir wollen $L \in NP$ zeigen. Wir konstuieren eine NTM N, die $x \in L$ in polynomieller Zeit akzeptiert. Dabei rät N das Zertifikat $y \in \{0,1\}^*$ mit $|y| \leq p(|x|)$ in Zeit $p(n)$ und führt dann V auf dem Wort $y\#x$ deterministisch in Zeit $q(|y\#x|)$ aus, falls V das Wort $y\#x$ auch akzeptiert. Die Maschine N akzeptiert die Eingabe x genau dann, wenn mindestens einer der möglichen Rechenwege akzeptiert. Also gilt:

$$\begin{aligned} x \in L &\Leftrightarrow \exists y \in \{0,1\}^* \text{ mit } |y| \leq p(|x|) : \\ &\quad\ V \text{ akzeptiert } y\#x \text{ in Laufzeit } q(|y\#x|) \\ &\Leftrightarrow N \text{ akzeptiert } x \text{ in polynomieller Zeit.} \end{aligned}$$

□

Übungsaufgabe: Präzisieren Sie den obigen Beweis für beliebige Übergangsfunktionen mit maximal k Verzweigungen.

Nun wenden das Lemma auf ein Beispiel an.

Lemma 18. *Das TSP-Entscheidungsproblem liegt in NP.*

Beweis. Als Sprache formulieren wir das Problem durch:

$$\text{TSP} := \left\{ \langle V, E, c, t\rangle \;\middle|\; \begin{array}{l} G = (V,E) \text{ Graph mit Kantenkosten } c(i,j) \in \mathbb{N} \\ \text{für } (i,j) \in E \text{ und es gibt eine Rundreise durch} \\ V \text{ mit Kosten } \leq t \end{array} \right\}.$$

Der Verifizierer V_{TSP} für diese Sprache könnte wie folgt aussehen. Eine beliebige Rundreise kann durch eine Permutation π mit der Knotenfolge $\pi(1), \pi(2), \ldots, \pi(|V|)$ beschrieben werden. Die Länge dieser Beschreibung liegt in $O(|V| \log |V|)$.

Der deterministische Verifizierer V erhält als Eingabe die Permutation $\langle \pi \rangle \# x$ und überprüft zunächst, ob π eine Permutation ist. Da x die Beschreibung des Graphen darstellt, gilt $|\langle \pi \rangle| \in O(|V| \log |V|) \leq |x| =: n$. Danach wird

$$\sum_{i=1}^{|V|-1} c(\pi(i), \pi(i+1)) + c(\pi(|V|), \pi(1)) \leq t$$

getestet und akzeptiert, falls die Aussage stimmt, sonst wird abgelehnt.

Offensichtlich ist V ein polynomieller Verifizierer für L mit $p(n) \in O(n)$ und mit einer Laufzeit $q(|\langle \pi \rangle \# x|)$. Wir können nämlich annehmen, dass für $l = \sum_{(i,j) \in E} \log(c(i,j)) \leq n$ die Addition und Subtraktion in Zeit $C \cdot l^2 \in O(n^2)$ durchgeführt werden kann. Für das jeweilige Finden der Kosten in x wird jeweils maximal $O(|\langle \pi \rangle \# x|)$ Zeit benötigt. Somit ist $q(n) \in O(n^3)$. □

3.7.1 *P* und *NP*

Offensichtlich gilt $P \subseteq NP$, da jede deterministische Maschine ein Spezialfall einer nichtdeterministischen Maschine ist.

Das vielleicht bekannteste Problem der Informatik ist die Frage, ob

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

gilt. Ist also P eine echte Teilmenge von NP ($P \subset NP$) oder sind die Klassen identisch. Darüber wurde in den letzten Jahrzehnten intensiv diskutiert. Die Mehrheit der Forscher geht heute eher davon aus, dass die Klassen nicht identisch sind aber es gibt bislang keinen Beweis dafür. Es gibt auch eine Gruppe von Forschern, die glauben, dass diese Frage vielleicht gar nicht beweisbar sein kann bzw. die notwendige Theorie zur genaue Beschreibung des Sachverhaltes noch nicht gefunden wurde. Wieder andere glauben, dass polynomielle Algorithmen mit in der Praxis völlig irrelevanten großen Exponenten arbeiten werden.

Im Prinzip stellt sich die Frage, ob man durch eine intelligente Vorgehensweise auf die Generierung exponentiell vieler Beweiskandidaten verzichten kann, eine solche Vorgehensweise ist unbekannt. Für die Lösung des Problems kann man mit Weltruhm und einer Belohnung von einer Million US Dollar rechnen.

3.8 *NP*-Vollständigkeit

Aus $P \subseteq NP$ folgt, dass nicht alle Probleme in NP schwer zu lösen sind. Wir möchten eine weitere Aufteilung der Probleme in NP vornehmen. Dazu führen wir den Begriff der NP-Vollständigkeit ein. Als NP-vollständig bezeichnen wir die Probleme aus NP mit maximalem Schwierigkeitsgrad. Das Konzept wird dadurch beschrieben, dass es ausreicht nur für eine NP-vollständige Sprache einen deterministischen polynomiellen Algorithmus zu entwerfen, um vollständig alle anderen solche Probleme polynomiell zu lösen. Wir werden zeigen das SAT NP-vollständig ist und daraus die NP-Vollständigkeit weiterer Problem folgern, wie zum Beispiel TSP oder Clique.

3.8.1 Polynomielle Reduktion

Reduktionen hatten wir bereits im Umfeld der Sprachen und der Frage der Entscheidbarkeit verwendet, um die Komplexität der Berechenbarkeit von Sprachen zu vergleichen. Wir wollen dieses Konzept nun durch eine Zeitkomplexitätskomponente erweitern und sprechen dann von *polynomiellen Reduktionen.*

Definition 18. Seien L_1 und L_2 zwei Sprachen über $\{0,1\}$. L_1 ist *polynomiell reduzierbar* auf L_2, wenn es eine Reduktion von L_1 nach L_2 mittels einer Funktion $f : \Sigma^* \to \Sigma^*$ gibt, die in polynomieller Zeit berechenbar ist. Wir schreiben dann auch $L_1 \leq_p L_2$.

Wenden wir den Begriff der Reduktion mittels f an, so müssen wir für $L_1 \leq_p L_2$ eine totale, DTM-berechenbare Funktion $f : \Sigma^* \to \Sigma^*$ finden, für die

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

gilt. Für f muss es eine DTM M geben, die f in polynomieller Laufzeit berechnet. Die Definition ermöglicht es, einfache Übertragungen von Komplexitäten vorzunehmen.

Lemma 19. *Falls $L_1 \leq_p L_2$ und $L_2 \in P$, dann gilt auch $L_1 \in P$.*

Beweis. Für L_2 gibt es einen polynomiellen Algorithmus A und es gibt eine Funktion f, die L_1 auf L_2 reduziert. Daraus entwerfen wir einen Algorithmus B. Dieser berechnet zunächst aus einer Eingabe x für L_1 die Eingabe $f(x)$ für L_2, startet dann A auf $f(x)$ und übernimmt das Akzeptanzverhalten.

Dieser deterministische Algorithmus B ist in der Laufzeit polynomiell beschränkt. Zunächst ist f durch ein Polynom p beschränkt. Das heißt auch $|f(x)| \leq p(|x|) + |x|$, zumindest die Eingabe muss berücksichtigt werden und jedes Schreiben einer weiteren Ausgabe kostet auch eine Zeiteinheit. Die

Laufzeit von A wiederum ist polynomiell in der Größe der Eingabe $|f(x)|$ durch ein Polynom q beschränkt. Die Gesamtlaufzeit von B für eine Eingabe x der Größe $|x|$ ist dann beschränkt durch $p(|x|)+q(p(|x|)+|x|) \leq C \cdot h(|x|)$ für eine Polynom h. □

Übungsaufgabe: Ist $A \leq_p B$ und $B \leq_p C$ so gilt auch $A \leq_p C$.

Beispiele für polynomielle Reduktionen werden wir im übernächsten Abschnitt behandeln.

3.8.2 Definition der *NP*-Vollständigkeit

Wir definieren zunächst die Klasse schwerer Probleme, auf diese Probleme lassen sich alle Probleme aus *NP* reduzieren. Es handelt sich dann quasi um einen Stellvertreter der Komplexität.

Definition 19. Ein Problem L heißt *NP*-schwer (*NP*-hard im Englischen), falls gilt:

$$\text{Für alle } L' \in NP \text{ gilt: } L' \leq_p L.$$

Übungsaufgabe: Ist L_1 *NP*-schwer und gilt $L_1 \leq_p L_2$ so ist auch L_2 *NP*-schwer.

Die Folgerung aus der Übungsaufgabe wird häufig verwendet, um für ein Problem zu zeigen, dass es *NP*-schwer ist. Außerdem gilt:

Theorem 13. *Ist L NP-schwer und $L \in P$, dann folgt $P = NP$.*

Beweis. Sei L' ein beliebiges Problem aus *NP*. Es gilt $L' \leq_p L$. Aus $L \in P$ folgt nach Lemma 19, dass auch $L' \in P$ gilt. Dann gilt $NP \subseteq P$ und somit $NP = P$. □

Definition 20. Ein Problem L heißt *NP*-vollständig (*NP*-complete im Englischen), falls gilt:

1. $L \in NP$
2. L ist *NP*-schwer.

Wir haben also eine Teilmenge der Probleme aus *NP* beschrieben, die die schweren Problem aus *NP* darstellen. Gelegentlich gibt es auch Problemstellungen, die nur *NP*-schwer sind, also die Tatsache, dass diese Probleme auch in *NP* liegen, ist nicht unbedingt gegeben.

3.8.3 Beispiele polynomieller Reduktionen

In diesem Abschnitt zeigen wir für ein paar ausgewählte Probleme, dass sie *NP*-vollständig sind gemäß Definition 20. Grundsätzlich muss man hierfür für ein Problem $L \in NP$ zeigen, das *alle* anderen Probleme $L' \in NP$ polynomiell auf L reduziert werden können. Mit der Aussage der obigen Übungsaufgabe genügt es jedoch zu zeigen, dass ein beliebiges *NP*-schweres Problem L' auf L reduziert werden kann. Mit Theorem 15 zeigen wir, dass SAT *NP*-vollständig ist. Im folgenden Beweis wird dann zunächst $SAT \leq_p$ 3-SAT gezeigt. Den Beweis des Theorems 15 führen wir im Anschluss an diesen Abschnitt. Die angegebenen Reduktionen finden sich auch in [1].

Theorem 14. *Folgende Probleme sind NP-vollständig:*

1. 3-SAT
2. *Clique*
3. *Knotenüberdeckung (Vertex-Cover)*

Beweis. Wir reduzieren zunächst $SAT \leq_p$ 3-SAT, dann 3-$SAT \leq_p$ *Clique* und schließlich *Clique* $\leq_p$ *Knotenüberdeckung*. Der Beweis des Theorems folgt dann aus Theorem 15.

Für eine Reduktion $L' \leq_p L$ muss eine durch eine polynomiell zeitbeschränkte Turingmaschine berechenbare Funktion f angegeben werden, für die für alle $w \in L'$ gilt

$$w \in L' \Leftrightarrow f(w) \in L \ .$$

Wir gehen im Folgenden davon aus, dass in Polynomialzeit überprüft werden kann, ob w eine gültige Instanz für die Sprache L' kodiert, und beschränken uns daher auf die Beschreibung der Funktion f für den Fall, dass w tatsächlich eine solche Instanz kodiert.

1. Der Beweis 3-$SAT \in NP$ kann analog zum Beweis $SAT \in NP$ geführt werden. Wir zeigen nun, dass 3-SAT *NP*-schwer ist durch die Reduktion $SAT \leq_p$ 3-SAT. Unser Ziel ist die Angabe einer polynomiell berechenbaren Funktion f, sodass gilt

   $(\alpha, k) \in SAT$ ist erfüllbar $\Leftrightarrow (\alpha', 3) = f((\alpha, k)) \in$ 3-SAT ist erfüllbar,

   wobei α, α' aussagenlogische Formeln in konjunktiver Normalform sind, jeweils bestehend aus Klauseln vom Maximalgrad g bzw. 3. Sei $\alpha = k_1 \wedge \cdots \wedge k_\ell$ eine SAT-Instanz über m Variablen $x_1, \ldots, x_m$, und sei $k_i = L_1 \vee \cdots \vee L_n$, $4 \leq n \leq k$, eine beliebige Klausel in α. Wir führen $n-3$ neue Variablen $A_0, \ldots, A_{n-4}$ ein, und ersetzen k_i durch

eine Konjunktion k_i' von $n-2$ Klauseln vom Grad 3, über den Variablen $x_1, \ldots, x_n, A_0, \ldots, A_{n-4}$. Durch die Konstruktion wird sichergestellt sein, dass k genau dann durch eine Belegung der Variablen $A_0, \ldots, A_{n-4}$ erfüllt werden kann, wenn k_i bereits durch eine Belegung der Variablen $x_1, \ldots, x_n$ erfüllt ist. Wir definieren nun

$$\begin{aligned} k_i' &= (L_1 \vee L_2 \vee A_0) \wedge (\neg A_0 \vee L_3 \vee A_1) \wedge \ldots \\ &\wedge \quad (\neg A_{j-3} \vee L_j \vee A_{j-2}) \wedge \ldots \\ &\wedge \quad (\neg A_{n-5} \vee L_{n-2} \vee A_{n-4}) \wedge (\neg A_{n-4} \vee L_{n-1} \vee L_n) \end{aligned}$$

Da die Konstruktion von k_i' offensichtlich in Polynomialzeit möglich ist, genügt es nun zu zeigen, dass genau dann wenn k_i durch eine Belegung der Variablen $x_1, \ldots, x_m$ erfüllt ist, auch k_i' durch eine Belegung der Variablen $A_0, \ldots, A_{n-4}$ erfüllt werden kann. Wir zeigen zunächst, dass wenn k_i nicht erfüllt ist, k_i' auch nicht erfüllt werden kann.

Wir nehmen für einen Widerspruch an, dass eine Belegung B der x_i und A_j, $1 \leq i \leq n$ und $0 \leq j \leq n-3$ gibt, sodass k_i nicht erfüllt ist wohingegen k_i' erfüllt ist. Da durch B alle L_i false sind, folgt zunächst, dass A_0 true sein muss, da sonst die erste Klausel von k nicht erfüllt ist. Folglich ist $\neg A_0$ false, also muss weiterhin A_1 true sein. Wiederholen wir dieses Argument, so müssen letztendlich auch A_{n-5} und A_{n-4} true sein. Da nach Voraussetzung alle L_i false sind für $1 \leq i \leq n$, ist die Klausel $(\neg A_{n-4} \vee L_{n-1} \vee L_n)$ nicht erfüllt. Dies ist ein Widerspruch zur Annahme k_i' sei durch Belegung B erfüllt. Somit ist k_i' nicht erfüllbar, wenn k_i durch die Belegung der Variablen $x_1, \ldots, x_m$ nicht erfüllt ist.

Nehmen wir jetzt an, dass k_i durch eine Belegung B der Variablen $x_1, \ldots, x_m$ erfüllt ist. Es bleibt zu zeigen, dass k_i' dann auch durch eine Belegung der Variablen $A_0, \ldots, A_{n-4}$ erfüllt werden kann. Da k_i durch Belegung B erfüllt ist, muss es einen Index j, $1 \leq j \leq n$ geben mit $L_j = \text{true}$ gemäß B. Wir verfahren nun wie folgt.

- $j \in \{1, 2\}$: Setze $A_i = \text{false}$ für $0 \leq i \leq n-4$.
- $j \in \{n-1, n\}$: Setze $A_i = \text{true}$ für $0 \leq i \leq n-4$.
- $2 < j < n-1$: Durch L_j wird die Klausel $(\neg A_{j-3} \vee L_j \vee A_{j-2})$ erfüllt. Setze daher alle Variablen $A_0, \ldots, A_{j-3} = \text{true}$ und alle Variablen $A_{j-2}, \ldots, A_{n-4} = \text{false}$.

Wir können leicht verifizieren, dass k_i' durch die Belegung der Variablen $A_0, \ldots, A_{n-4}$ erfüllt wird.

2. Wir haben $Clique \in NP$ bereits zuvor bewiesen. Wir zeigen nun, dass *Clique* NP-schwer ist durch die Reduktion $3\text{-}SAT \leq_p Clique$. Wir konstruieren nun aus einer gegebenen 3-SAT Instanz $\alpha = k_1 \wedge \cdots \wedge k_\ell$ einen Graphen $G = (V, E)$, sodass G genau dann eine k-Clique

enthält wenn α erfüllbar ist. Falls in α eine Klausel weniger als drei Literale enthält, duplizieren wir zunächst in jeder dieser Klauseln ein Literal, sodass wir annehmen können, dass jede Klausel genau drei Literale enthält. Nun nummerieren wir die Literale derart, dass in Klausel k_i genau die Literale $L_{3i-2}, L_{3i-1}, L_{3i}$ enthalten sind. In der ersten Klausel sind folglich die Literale L_1, L_2, L_3 enthalten, in der zweiten Klausel die Literale L_4, L_5, L_6 und so weiter. Hierbei steht jedes Literal L_j jeweils entweder für einen Ausdruck x_r oder $\neg x_r$, wobei der Index r natürlich in j variieren kann.

Wir konstruieren nun zunächst die Knotenmenge V des Graphen G. Für jede Klausel k_i, $1 \leq i \leq k$ erstellen wir eine Knotenmenge $\mathcal{K}_i = \{L_{3i-2}, L_{3i-1}, L_{3i}\}$. Schließlich sei $V = \bigcup_{i=1}^{k} \mathcal{K}_i$. Die Kantenmenge E wird wie folgt konstruiert. Zwei Knoten $u \in \mathcal{K}_i$ und $v \in \mathcal{K}_j$ werden genau dann durch eine Kante $e \in E$ verbunden, wenn gilt

(a) $i \neq j$ (d.h. u und v stammen aus verschiedenen Klauseln) und

(b) $(u \wedge v)$ kann erfüllt werden durch eine Belegung der Variablen $x_1, \ldots, x_m$ (d.h., angenommen $i \neq j$, wenn $u = x_r$, dann wird u mit v verbunden falls $v \neq \neg x_r$, und wenn $u = \neg x_r$, dann wird u mit v verbunden falls $v \neq x_r$).

Wir zeigen nun, dass α genau dann erfüllbar ist, wenn G eine k-Clique enthält.

Sei zunächst α erfüllt. Dann konstruieren wir eine Belegung B der Variablen $x_1, \ldots, x_m$ welche α erfüllt. Da jede Klausel in α erfüllt ist, können wir in jeder Klausel k_i einen Knoten $v_i \in \mathcal{K}_i$ wählen, dessen zugehöriges Literal gemäß B erfüllt ist. Nennen wir das zu Knoten v_i zugehörige Literal $L_{\ell(i)}$. Da durch B die Formel $\alpha' = (L_{\ell(1)} \wedge \cdots \wedge L_{\ell(k)})$ erfüllt ist, und keine zwei in α' vorkommenden Literale aus derselben Klausel in α stammen, sind in G auch alle Knotenpaare mit Knoten aus $\{v_1, \ldots, v_k\}$ durch eine Kante verbunden. Somit existiert in G eine k-Clique.

Nehmen wir nun an, dass G eine k-Clique C enthält. Nach Definition der Kantenmenge E stammen alle Knoten in C aus verschiedenen Klauseln, da andernfalls zwei Knoten aus einer Klausel mit einer Kante verbunden wären – im Widerspruch zur Definition von E. Bezeichnen wir somit die Knoten in C mit $v_1, \ldots, v_k$ wobei v_i ein Knoten ist, dessen Literal in der i-ten Klausel von α enthalten ist, für $1 \leq i \leq k$. Wir bezeichnen weiterhin wie zuvor mit $L_{\ell(i)}$ das Literal, welches dem Knoten v_i zugeordnet ist. Wir konstruieren nun eine Belegung B der Variablen $x_1, \ldots, x_m$, welche alle Literale $L_{\ell(1)}, \ldots, L_{\ell(k)}$ erfüllt. Gemäß B ist dann auch α erfüllt, da jede Klausel in α ein Literal aus der Menge $\{L_{\ell(1)}, \ldots, L_{\ell(k)}\}$ enthält.

Für jeden Knoten $v_i \in C$ gilt entweder (i) $L_{\ell(i)} = x_r$ oder (ii) $L_{\ell(i)} = \neg x_r$ für ein $r \in \{1, \ldots, m\}$. Da v_i in G mit allen Knoten in C durch eine Kante verbunden ist, gilt (i) $\neg x_r \notin \{L_{\ell(1)}, \ldots, L_{\ell(k)}\}$ oder (ii) $x_r \notin \{L_{\ell(1)}, \ldots, L_{\ell(k)}\}$. Wählen wir folglich $L_{\ell(i)} = \text{true}$, dann wird kein Literal der Menge $\{L_{\ell(1)}, \ldots, L_{\ell(k)}\}$ false; ggf. werden neben $L_{\ell(i)}$ noch weitere dieser Literale ebenfalls true. Somit können wir alle den Konten $v_1, \ldots, v_k$ entsprechenden Literale durch eine Belegung der Variablen $x_1, \ldots, x_m$ erfüllen. Variablen x_r mit $\{x_r, \neg x_r\} \cap \{L_{\ell(1)}, \ldots, L_{\ell(k)}\} = \emptyset$, d.h. die in C nicht vorkommen, können beliebig true oder false gesetzt werden. Da nach Konstruktion jede einzelne Klausel in α erfüllt ist, haben somit insbesondere eine Belegung B der Variablen $x_1, \ldots, x_k$ konstruiert, welche α erfüllt.

3. Eine nichtdeterministische Turingmaschine kann (nichtdeterministisch) eine beliebige k-elementige Knotenmenge des gegebenen Graphen G auswählen und verifizieren, ob diese eine Knotenüberdeckung von G ist. Somit gilt Knotenüberdeckung $\in$ *NP*. Wir zeigen nun durch die Reduktion $\mathit{Clique} \leq_p$ Knotenüberdeckung, dass Knotenüberdeckung *NP*-vollständig ist.

   Definition 21. Für einen beliebigen Graphen $G = (V, E)$ bezeichne $\overline{G} = (V, \{V \times V\} \setminus E)$ den *Komplementärgraphen* von G.

   Zwei Knoten im Komplementärgraphen $\overline{G}$ von G sind somit genau dann in $\overline{G}$ durch eine Kante verbunden, wenn sie in G *nicht* adjazent sind. Es gelten daher folgende Äquivalenzen, wobei C immer eine k-elementige Teilmenge von V bezeichnet.

   $$\begin{gathered} G \text{ enthält eine } k\text{-Clique } C \\ \Leftrightarrow \\ \text{Je zwei Knoten in } C \text{ sind in } G \text{ durch eine Kante verbunden} \\ \Leftrightarrow \\ \text{Keine zwei Knoten in } C \text{ sind in } \overline{G} \text{ durch eine Kante verbunden} \\ \Leftrightarrow \\ V \setminus C \text{ ist eine Knotenüberdeckung von } \overline{G} \text{ der Größe } |V| - k \end{gathered}$$

   Für jede *Clique*-Instanz $\langle V, E, k \rangle$ kann in Polynomialzeit sowohl der Komplementärgraph $\overline{G}$ von $G = (V, E)$, als auch die Zahl $|V| - k$ berechnet werden. Wir haben somit das Clique-Problem auf das Problem Knotenüberdeckung reduziert.

□

3.8.4 Der Satz von Cook und Levin

Die zentrale Aussage, dass *SAT NP*-vollständig ist, wurde unabhängig von Stephen Cook (1971) und Leonid Levin (1973) bewiesen. Er beinhaltet eine Reduktion, die zeigt, dass jedes Problem aus *NP* in polynomieller Zeit auf das Erfüllbarkeitsproblem reduziert werden kann.

Theorem 15 (Cook und Levin)**.** *Das Problem SAT ist NP-vollständig.*

Bevor wir mit dem Beweis beginnen, beschreiben wir ein paar Hilfskonstrukte zur besseren Beweisführung. Der folgende Beweis stammt aus Blum [1].

Die Erfüllbarkeit einer aussagenlogischen Formel A mit Variablenmenge $V = \{x_1, x_2, x_3, \ldots\}$ und einer Belegung beschreiben wir rekursiv durch eine Funktion $\varphi(A)$, die entweder 0 (falsch) oder 1 (wahr) ausgibt.

- Für die Variablenmenge $V = \{x_1, x_2, x_3, \ldots\}$ legt φ mit $\varphi : V \to \{0, 1\}$ eine Belegung fest.
- Sei y ein Literal, dann ist
$$\varphi(y) := \begin{cases} \varphi(x_i) & \text{falls } y = x_i \\ 1 - \varphi(x_i) & \text{falls } y = \neg x_i \end{cases}$$
- Für Literale $y_1, y_2, \ldots y_k$ und die Klausel $(y_1 \vee y_2 \vee \cdots \vee y_k)$ gilt:
$$\varphi((y_1 \vee y_2 \vee \cdots \vee y_k)) := \max\{\varphi(y_i) | 1 \leq i \leq k\}\,.$$
- Für den aussagenlogischen Ausdruck $k_1 \wedge k_2 \wedge \cdots \wedge k_l$ für Klausel $k_1, k_2, \ldots, k_l$ gilt:
$$\varphi(k_1 \wedge k_2 \wedge \cdots \wedge k_l) := \min\{\varphi(k_j) | 1 \leq j \leq l\}\,.$$

Der aussagenlogische Ausdruck A ist genau dann *erfüllbar*, falls eine Belegung φ existiert mit $\varphi(A) = 1$.

Desweiteren benötigen wir das folgende Hilfslemma. Wir wollen für eine Variablenmenge $\{x_1, x_2, \ldots, x_l\}$ einen Ausdruck A in konjunktiver Normalform mit l^2 Literalen konstruieren, der genau dann erfüllbar ist, falls genau eine dieser Variablen wahr ist. Der Beweis des Lemmas ist eine Übungsaufgabe.

Lemma 20. *Für die Variablenmenge $V = \{x_1, x_2, \ldots, x_l\}$ ist der aussagenlogische Ausdruck*

$$\mathit{ExactOne}(x_1, x_2, \ldots, x_l) := \left(\bigwedge_{1 \leq i < j \leq l} (\neg x_i \vee \neg x_j) \right) \wedge (x_1 \vee x_2 \vee \cdots x_l)$$

genau dann erfüllbar, wenn genau eine Variable aus V wahr ist. Der Ausdruck $\mathit{ExactOne}(x_1, x_2, \ldots, x_l)$ enthält genau l^2 Literale.

Nun zum Beweis von Theorem 15:

Beweis. (Theorem 15, siehe auch [1]) Nach Lemma 15 wissen wir bereits, dass SAT in NP liegt. Es muss also noch gezeigt werden, dass $L \leq_p SAT$ für alle $L \in NP$ gilt. Sei also L ein beliebiges Problem aus NP. Dann existiert eine NTM N mit $L = L(N)$ und polynomieller Laufzeit.

Wir nehmen an, dass es sich bei N um eine 1-Band NTM handelt. Analog zum Beweis von Lemma 7 über die Simulation einer k-Band DTM durch eine 1-Band DTM ist diese Annahme bezüglich der Laufzeiten gestattet. Wir benutzen zwei ausgewiesene Endzustände für das Akzeptieren und Verwerfen, das erleichtert die Konstruktion und ist keine Einschränkung. Sei nun $p(n) = C \cdot n^d$ die zu $N = (\Sigma, Q, \delta, q_0, F)$ und L zugehörige Laufzeitfunktion und

- $Q = \{q_0, q_1, \ldots, q_u\}$
- $\Sigma = \{a_1, a_2, \ldots, a_r\}$ mit $a_1 = \sqcup$
- $F = \{q_{u-1}, q_u\}$ wobei q_u den akzeptierenden und q_{u-1} den verwerfenden Zustand beschreibt.

Für eine Eingabe $x \in \Sigma^*$ für N wollen wir einen aussagenlogischen Ausdruck $A(x)$ in polynomieller Zeit konstruieren, sodass "$x \in L \iff A(x)$ ist erfüllbar" gilt. Für die polynomielle Konstruktion spielt die Laufzeitfunktion p von N ein entscheidende Rolle. Die Laufzeitabschätzung von N über die Funktion p ist eine *obere* Schranke, mehr wissen wir leider über einzelne Akzeptanzpfade der Wörter nicht. Deshalb lassen wir die Maschine stets mit maximaler Laufzeit weiterlaufen. Das realisieren wir, indem wir die Haltekonfigurationen der beiden Zustände q_u und q_{u-1} beliebig oft wiederholen lassen.

- Für $\delta(q_i, a_j) = \emptyset$ und $i = u, u-1$ füge $\delta(q_i, a_j) = \{(q_i, a_j, 0)\}$ zur Übergangsfunktion hinzu.

Bemerkung: Falls es in δ andere Haltezustände ohne definierte Übergänge gibt, können diese prinzipiell genauso erweitert werden.

Sei nun $x \in \Sigma^*$ eine Eingabe für N mit $|x| = n$. Nun gilt $x \in L(N)$ genau dann, wenn es Konfigurationen $K_0, K_1, \ldots, K_{p(n)}$ gibt, mit folgenden Bedingungen:

- K_0 ist die Startkonfiguration von N bei Eingabe x.
- K_{i+1} ist eine Folgekonfiguration von K_i für $0 \leq i < p(n)$.
- Der Zustand in $K_{p(n)}$ ist q_u.

Typ	Variable	Intendierte Bedeutung
Zustände	$q_{t,k}$ $0 \leq t \leq p(n)$, $0 \leq k \leq u$	$q_{t,k} = 1 \Leftrightarrow q_k$ Zustand von K_t
Inhalt	$a_{t,i,j}$ $0 \leq t, i \leq p(n)$, $1 \leq j \leq r$	$a_{t,i,j} = 1 \Leftrightarrow a_j$ Inhalt i-tes Bandquadrat in K_t
Kopfpos.	$s_{t,i}$ $0 \leq t, i \leq p(n)$	$s_{t,i} = 1 \Leftrightarrow$ In K_t steht L/S Kopf auf Bandpos. i
Übergang	$b_{t,l}$ $0 \leq t < p(n)$, $1 \leq l \leq m$	$b_{t,l} = 1 \Leftrightarrow$ Übergang l wird von K_t nach K_{t+1} angewendet

Tabelle 3.1: Bedeutung der Variablen. Die Anzahl der Variablen liegt in $O(p(n)^2)$, da r und m unabhängig von n durch Konstanten beschränkt sind.

Die Übergangsfunktion von N kann als Relation mit einer endlichen Anzahl m an Tupeln aus $(Q \times \Sigma) \times (Q \times \Sigma \times \{-1, 1, 0\})$ aufgefasst werden. Wir nummerieren diese m verschiedenen Tupel durch und können somit eindeutig auf die Übergänge durch eine Nummer zugreifen.

Jetzt wollen wir die Konfigurationsübergänge durch eine Formel in konjuktiver Normalform eindeutig beschreiben, dafür benötigen wir Variablen, die in der Tabelle 3.8.4 angegeben sind. Die Anzahl der Variablen ist polynomiell beschränkt.

Jetzt wollen wir den gesamten Konfigurationswechsel durch Formeln in konjunktiver Normalform beschreiben. Insgesamt wird für ein Eingabewort $x = a_{j_1} a_{j_2} \ldots a_{j_n}$ mit der Länge n und für die Maschine N ein aussagenlogischer Ausdruck

$$A(x) = S \wedge R \wedge U \wedge q_{p(n),u}$$

mit folgenden Bedeutungen entworfen. Zunächst beschreibt $q_{p(n),u}$ den akzeptierenden Endzustand für $x \in L$. $A(x)$ kann nur dann erfüllt werden, wenn $q_{p(n),u} = 1$ gilt. Die weiteren Bedeutungen sind:

Startbedingung S : Die Startkonfiguration K_0 mit Zustand q_0. Der L/S-Kopf steht auf dem ersten Bandquadrat und $x\sqcup^{p(n)-n}$ ist der Bandinhalt. Durch die Hinzunahme der $\sqcup$-Zeichen stellen wir sicher, dass nur wirklich verwendete Bandquadrate betrachtet werden müssen. Die Maschine kann bei $p(n)$ Schritten nur maximal bis zum letzten $\sqcup$-Zeichen vorlaufen.

Randbedingungen R : Für jedes K_t mit $1 \leq t \leq p(n)$: Die Maschine N befindet sich in genau einem **Zustand**, der L/S-Kopf steht auf genau einer **Kopfposition**, jedes signifikante (siehe oben) Bandquadrat enthält genau einen **Inhalt**. Falls $t < p(n)$ gilt, gibt es für den **Übergang**

zu K_{t+1} genau ein Tupel aus δ, das anwendbar ist.

Übergangsbedingungen U : Für $1 \leq t < p(n)$ wird der Zustand, die Kopfposition und die Bandinschrift für die Anwendung des Übergangs von K_t nach K_{t+1} festgelegt.

Zunächst kann die Startkonfiguration leicht durch

$$S := q_{0,0} \wedge s_{0,1} \wedge a_{0,1,j_1} \wedge a_{0,2,j_2} \wedge \cdots \wedge a_{0,n,j_n} \wedge a_{0,n+1,1} \wedge a_{0,n+2,1} \wedge \cdots \wedge a_{0,p(n),1}$$

beschrieben werden.

Für einen Zeitpunkt t können die Randbedingungen $R(t)$ nach Lemma 20 durch

$$\begin{aligned} R_{\text{ZUSTAND}}(t) &:= \text{ExactOne}(q_{t,0}, q_{t,1}, \ldots, q_{t,u}) \\ R_{\text{POSITION}}(t) &:= \text{ExactOne}(s_{t,1}, s_{t,2}, \ldots, s_{t,p(n)}) \\ R_{\text{INHALT}}(t) &:= \bigwedge_{0 \leq i \leq p(n)} \text{ExactOne}(a_{t,i,1}, a_{t,i,2}, \ldots, a_{t,i,r}) \\ R_{\text{ÜBERGANG}}(t) &:= \text{ExactOne}(b_{t,1}, b_{t,2}, \ldots, b_{t,m}) \end{aligned}$$

beschrieben werden. Nach Lemma 20 können diese Ausdrücke mit insgesamt $h(u, r, m, n) := (u+1)^2 + p^2(n) + (p(n)+1)r^2 + m^2 \in O(p^2(n))$ Literalen beschrieben werden. Insgesamt ergibt sich daraus der Ausdruck

$$R := \bigwedge_{0 \leq t \leq p(n)} (R_{\text{ZUSTAND}}(t) \wedge R_{\text{POSITION}}(t) \wedge R_{\text{INHALT}}(t)) \wedge R_{\text{ÜBERGANG}}(t))$$

mit $(p(n)+1) \cdot h(u, r, m, n) \in O(p^3(n))$ vielen Literalen.

Analog wollen wir nun mit $U(t)$ einen Ausdruck beschreiben, der den korrekten Übergang von K_t nach K_{t+1} für $0 \leq t < p(n)$ beschreibt.

Folgendes muss zum Zeitpunkt t für ein Bandquadrat i geleistet werden.

1. Falls N das i-te Bandquadrat nicht liest, darf in diesem Bandquadrat nichts geändert werden.

2. Falls N das i-te Bandquadrat liest und beim Übergang zur Konfiguration K_{t+1} das l-te Tupel $((q_{k_l}, a_{j_l}), (q_{\bar{k}_l}, a_{\bar{j}_l}, v_l))$ der Übergangsfunktion anwendet, dann ist der Übergang wie folgt:

   Zeitpunkt t : q_{k_l} ist der **Zustand** und a_{j_l} ist der gelesene **Inhalt** im Bandquadrat i.

   Zeitpunkt $t+1$: $q_{\bar{k}_l}$ ist der **Zustand**, $a_{\bar{j}_l}$ der **Inhalt** im Bandquadrat i und $i + v_l$ die **Kopfposition**.

$U(t,i)$ muss beispielweise ausdrücken, dass $(\neg s_{t,i} \wedge a_{t,i,j}) \rightarrow a_{t+1,i,j}$ gültig ist. Implikationen der Art $(X \rightarrow Y)$ ersetzen wir äquivalent durch $\neg X \vee Y$ und somit $(\neg s_{t,i} \wedge a_{t,i,j}) \rightarrow a_{t+1,i,j}$ durch die äquivalente Klausel $(s_{t,i} \vee \neg a_{t,i,j} \vee a_{t+1,i,j}$.

Die Bedingung 1. von $U(t)$ für ein spezielles i lautet somit:

$$U_1(t,i) := \bigwedge_{1 \leq j \leq r} (\neg s_{t,i} \wedge a_{t,i,j}) \rightarrow a_{t+1,i,j} = \bigwedge_{1 \leq j \leq r} (s_{t,i} \vee \neg a_{t,i,j} \vee a_{t+1,i,j})$$

Solche Implikationen lassen sich nun auch für die Bedingung 2. und das Tupel l umsetzen. Die Implikationen haben die Form $(s_{t,i} \wedge b_{t,l}) \rightarrow \cdot)$ und werden entsprechend umgesetzt durch $(\neg s_{t,i} \vee \neg b_{t,l} \vee \cdot)$

$$\begin{aligned} U_2(t,i) \quad := \quad & \bigwedge_{1 \leq l \leq m} \\ & [(\neg s_{t,i} \vee \neg b_{t,l} \vee q_{t,k_l}) \\ & \wedge (\neg s_{t,i} \vee \neg b_{t,l} \vee a_{t,i,j_l}) \\ & \wedge (\neg s_{t,i} \vee \neg b_{t,l} \vee q_{t+1,\overline{k}_l}) \\ & \wedge (\neg s_{t,i} \vee \neg b_{t,l} \vee a_{t+1,i,\overline{j}_l}) \\ & \wedge (\neg s_{t,i} \vee \neg b_{t,l} \vee s_{t+1,i+v_l})] \end{aligned}$$

Für einen Zeitpunkt t muss dass für alle Bandquadrate i geprüft werden:

$$U(t) := \bigwedge_{0 \leq i \leq p(n)} U_1(t,i) \wedge U_2(t,i)\,.$$

Da $U_1(t,i)$ aus $3r$ Literalen und $U_2(t,i)$ aus $15m$ Literalen besteht, besteht $U(t)$ aus $(p(n)+1) \times (3r+15m)$ Literalen.

$U(t)$ muss wiederum für alle Zeitpunkte gelten. Insgesamt ist dann

$$U = \bigwedge_{0 \leq t < p(n)} U_1(t) \wedge U_2(t)$$

und enthält für $p(n)+1$ Zeitpunkte insgesamt $(p(n)+1)^2 \times (3r+15m)$ Literale.

Aus den obigen Konstruktionen ergibt sich, dass $S \wedge R \wedge U \wedge q_{p(n),u}$ aus $O(p^3(n))$ vielen Literalen besteht. Offensichtlich ist es auch möglich aus der Maschine N und dem existierenden nichtdeterministischen Rechenweg das Wort $A(x)$ in polynomieller Zeit abhängig von $n = |x|$ zu berechnen. Wir müssen die Maschine kodieren und beim Ablauf das Wort erstellen.

Falls die Maschine N das Wort x verwirft, dann wird als Ausdruck $A(x)$ ein nicht-erfüllbarer Ausdruck zurückgegeben. Wir können dabei auch die

Funktion $p(n)$ verwenden, um die Konstruktion nach mehr als $p(n)$ Schritten abzubrechen.

Insgesamt gibt es eine DTM-berechenbare Funktion f, die jede Eingabe x in einen Ausdruck $A(x)$ überträgt. Die Konstruktion kann in polynomielle Laufzeit durchgeführt werden und benötigt $O(p^3(n) \log n)$ viel Platz.

Es bleibt zu zeigen:

$$x \in L \Leftrightarrow A(x) \in SAT\,.$$

"$\Longrightarrow$":

Falls $x \in L$ liegt, existiert eine Berechnung von N bezüglich x, die x in $p(|x|)$ Schritten akzeptiert. Das heißt, dass eine Folge von Konfigurationen mit $K_0, K_1, \ldots, K_{p(n)}$ existiert mit:

- K_0 ist die Startkonfiguration von N bei Eingabe x.
- K_{i+1} ist die Folgekonfiguration von K_i für $0 \leq i < p(n)$.
- Der Zustand in $K_{p(n)}$ ist q_u.

Für das Wort $A(x)$ können wir eine Belegung φ der Variablen angeben, die $A(x)$ erfüllt. Wir verwenden exakt die in Tabelle 3.8.4 intendierte Belegung, diese erfüllt den Ausdruck $A(x)$ gemäß Konstruktion.

Für $0 \leq t \leq p(n)$, $1 \leq k \leq u$ $\quad q_{t,k} = \begin{cases} 1 & \text{falls } q_k \text{ Zustand von } K_t \\ 0 & \text{sonst} \end{cases}$

Für $0 \leq t, i \leq p(n)$, $1 \leq j \leq r$ $\quad a_{t,i,j} = \begin{cases} 1 & \text{falls } a_j \text{ Inhalt } i\text{-tes} \\ & \text{Bandquadrat in } K_t \\ 0 & \text{sonst} \end{cases}$

Für $0 \leq t, i \leq p(n)$ $\quad s_{t,i} = \begin{cases} 1 & \text{In } K_t \text{ steht} \\ & \text{L/S Kopf auf Bandpos. } i \\ 0 & \text{sonst} \end{cases}$

Für $0 \leq t < p(n)$, $1 \leq l \leq m$ $\quad b_{t,l} = \begin{cases} 1 & \text{Übergang } l \text{ wird von } K_t \\ & \text{nach } K_{t+1} \text{ angewendet} \\ 0 & \text{sonst} \end{cases}$

"$\Longleftarrow$":

Sei nun umgekehrt $A(x)$ konstruiert worden. Sei φ eine Belegung der Variablen, die $A(x)$ erfüllt.

Da $\varphi(R) = 1$ gilt, gilt für jedes $0 \leq t \leq p(n)$:

- ExactOne$(q_{t,0}, q_{t,1}, \ldots, q_{t,u})$

- ExactOne$(s_{t,1}, s_{t,2}, \ldots, s_{t,p(n)})$
- $\bigwedge_{0 \leq i \leq p(n)}$ ExactOne$(a_{t,i,1}, a_{t,i,2}, \ldots, a_{t,i,r})$
- ExactOne$(b_{t,1}, b_{t,2}, \ldots, b_{t,m})$

Dann gibt für jedes t genau ein $k(t)$ mit $\varphi(q_{t,k(t)}) = 1$ und genau ein $i(t)$ mit $\varphi(s_{t,i(t)}) = 1$ und genau ein $l(t)$ mit $\varphi(b_{t,l(t)}) = 1$ und für $1 \leq i \leq p(n)$ genau ein $j(t,i)$ mit $\varphi(a_{t,i,j(t,i)}) = 1$. Dann beschreiben diese wahren Variablen genau die Konfiguration K_t und ein Tupel der Übergangsrelation:

- Zustand: $q_{k(t)}$
- Kopfposition: $s_{t,i(t)}$
- Bandinhalt: $(a_{j(t,1)}, a_{j(t,2)}, \ldots, a_{j(t,p(n))})$
- Übergang: $((q_{k(t)}, a_{j_{l(t)}}), (q_{\overline{k}(t)}, a_{\overline{j}_{l(t)}}, v_{l(t)}))$

Wegen $\varphi(S) = 1$ ist $k(0) = 0$ und $i(0) = 1$ etc. also K_0 die Startkonfiguration. Wegen $\varphi(q_{p(n),u}) = 1$ ist $K_{p(n)}$ eine akzeptierende Konfiguration.

Da auch $\varphi(U) = 1$ gilt, ist $U(t) = 1$ für alle $0 \leq t < p(n)$. Dann folgt aus $U_1(t)$, dass $j(t+1,i) = j(t,i)$ für alle $i \neq i(t)$ ist. Somit wird die Bandinschrift stets nur unter der Position $i(t)$ geändert.

Für $i(t)$ und $l(t)$ folgt aus $\varphi(s_{t,i(t)}) = 1$ und $\varphi(b_{t,l(t)}) = 1$, dass in $U_2(t)$ nun $\varphi(a_{t+1,i(t),\overline{j}_{l(t)}}) = 1$ und $\varphi(s_{t+1,i+v_{l(t)}}) = 1$ gelten muss. Also ist das $l(t)$-te Tupel der Übergangsfunktion anwendbar und findet beim Übergang von K_t nach K_{t+1} eine korrekte Anwendung.

Insgesamt ist $K_0, K_1, \ldots, K_{p(n)}$ eine akzeptierende Konfigurationsfolge für x bezüglich N und es gilt $x \in L$.

□

3.8.5 *NP*-Vollständigkeit arithmetischer Probleme

Zunächst möchten wir hier noch die *NP*-Vollstänigkeit zweier arithmetischer Probleme zeigen, sodass wir im nächsten Abschnitt leicht zeigen können, dass die Optimierungsprobleme des *Rucksackproblems* und des *Bepackens von Behältern* ebenfalls schwere Probleme sind, für die wir bestenfalls schnell Approximationen berechnen können.

Subset-Sum: Gegeben ist eine Menge $\{a_1, a_2, \ldots, a_N\}$ von N natürlichen Zahlen mit $a_i \in \mathbb{N}$ und ein $b \in \mathbb{N}$.
Frage: Gibt es eine Teilmenge $T \subseteq \{1, 2, \ldots, N\}$ sodass $\sum_{i \in T} a_i = b$ gilt?

Theorem 16. *Subset-Sum ist NP-vollständig.*

Beweis. (Skizze) Zunächst ist klar, dass Subset-Sum in *NP* liegt, da die Menge T als Zertifikat verwendet werden kann. Wir beschreiben eine Polynomialzeitreduktion von 3-SAT. Dazu muss eine Formel in konjunktiver Normalform mit n Variablen $\{x_1, \ldots, x_n\}$ und m Klauseln $\{c_1, \ldots, c_m\}$ in polynomieller Zeit in eine Subset-Sum Konfiguration überführt werden.

Danach muss gelten: Die Klausel ist genau dann erfüllbar, wenn das zugehörige Subset-Sum Problem eine Lösung hat.

Zur Reduktion benutzen wir (2n+2m) Dezimal-Nummern der Länge $n+m$, die Zahl b besteht aus n Einsen gefolgt von m Dreien.

Für jede Variable x_i wird eine Dezimal-Zahl y_i und für $\neg x_i$ eine Dezimal-Zahl z_i verwendet. Die Belegung 1 an der Stelle i legt das jeweils fest. Die Ziffern von $n+1$ bis $n+m$ legen durch 0 und 1 fest, ob das Literal in c_j vorkommt oder nicht vorkommt. In jedem c_j kommen maximal drei Literale vor. Da nur eins davon erfüllt sein muss, führen wir Dezimalzahlen s_i und t_i für jedes c_i ein, sodass in jedem Fall an dieser Stelle die Spaltensumme 3 erzielt werden kann.

Wie geben ein Beispiel für die Formel

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

an.

	x_1	x_2	x_3	c_1	c_2
y_1	1	0	0	1	0
z_1	1	0	0	0	1
y_2	0	1	0	1	1
z_2	0	1	0	0	0
y_3	0	0	1	0	0
z_3	0	0	1	1	1
s_1	0	0	0	1	0
t_1	0	0	0	1	0
s_2	0	0	0	0	1
t_2	0	0	0	0	1
b	1	1	1	3	3

Genau dann gibt es eine Auswahl T von Zahlen aus der gesamten Zahlenmenge $\{y_1, z_1, \ldots, y_n, z_n, s_1, t_1, \ldots, s_m, t_m\}$ mit $\sum_{x \in T} x = b$, wenn die zugehörige 3-SAT-Formel erfüllbar ist.

Beachten Sie, dass die Konstruktion polynomiell in $(n+m)$ ist. Die Eingabelänge der 3-SAT-Formel hatte gerade diese Länge. Die konstruierten Zahlen

sind nun zwar nicht zur Basis 2 binär kodiert aber zur Basis 10 kodiert und sind somit im Vergleich zu ihrer Zahlengröße in logarithmischer Größe (und nicht unär) gespeichert. □

Partition: Gegeben ist eine Menge $\{a_1, a_2, \ldots, a_N\}$ von N natürlichen Zahlen mit $a_i \in \mathbb{N}$.
Frage: Gibt es eine Teilmenge $T \subseteq \{1, 2, \ldots, N\}$, sodass die Teilsummen $\sum_{i \in T} a_i$ und $\sum_{j \in \{1,2,\ldots,N\} \setminus T} a_j$ identisch sind.

Partition ist ein Spezialfall von Subset-Sum, für $b = \frac{1}{2} \sum_{j \in \{1,2,\ldots,N\}} a_j$ und ist nicht einfacher zu lösen als Subset-Sum.

Theorem 17. *Partition ist NP-vollständig.*

Beweis. Partition liegt offensichtlich in *NP*, da es ein Spezialfall von Subset-Sum ist.

Wir zeigen Subset-Sum $\leq_p$ Partition mit polynomieller Reduktion p.

Für $\{a_1, a_2, \ldots, a_N\}$ mit $a_i \in \mathbb{N}$ und $b \in \mathbb{N}$ sei $A := \sum_{i=1}^{N} a_i$. Wir konstruieren eine Eingabe für Partition durch $a'_1, a'_2, \ldots, a'_{N+2}$ mit

- $a'_i := a_i$ für $i = 1, \ldots, N$
- $a'_{N+1} := 2A - b$, und
- $a'_{N+2} := A + b$

Es folgt: $\sum_{i=1}^{N+2} a'_i = 4A$. Dann fragt Partition danach, ob es eine Aufteilung von $\{a'_1, a'_2, \ldots, a'_{N+2}\}$ gibt, die jeweils $2A$ ergibt oder analog, ob es eine Teilmenge von $\{a'_1, a'_2, \ldots, a'_{N+2}\}$ gibt, die die Summe $2A$ ergibt. In polynomieller Zeit lässt sich diese Reduktion berechnen. Beachten Sie, dass die Zahlen binär kodiert sind.

Es gilt offensichtlich: Genau dann hat das Partition-Problem eine Lösung, wenn auch das Subset-Sum Problem eine Lösung hat.

⇒: Hat das Partition-Problem eine Lösung, so können $a'_{N+1} := 2A - b$ und $a'_{N+2} := A + b$ nicht in einer gemeinsamen Teilmenge sein, dann wäre die Summe zu groß. Die Zahlen aus $\{a'_1, a'_2, \ldots, a'_N\}$ die sich mit $a'_{N+1} := 2A - b$ in einer Teilmenge befinden, summieren sich zu dann zu b auf und ergeben eine Subset-Sum Lösung.

⇐: Wenn eine Teilmenge von $\{a_1, a_2, \ldots, a_N\} = \{a'_1, a'_2, \ldots, a'_N\}$ den Summenwert b ergibt, so können wir $a'_{N+1} = 2A - b$ hinzufügen und erhalten eine Lösung für das Partition-Problem. □

Literaturverzeichnis

[1] Norbert Blum. Theoretische Informatik: Eine anwendungsorientierte Einführung. Oldenbourg Verlag, 2001.

[2] Uwe Schöning. Theoretische Informatik – kurzgefasst. Spektrum, Akad. Verl., 2009.

[3] Uwe Schöning. Ideen der Informatik – Grundlegende Modelle und Konzepte der Theoretischen Informatik. Oldenbourg, 2009.

[4] C. H. Papadimitriou. Computational Complexity. Addison-Wesley 1994.

[5] Katrin Erk, Lutz Priese. Theoretische Informatik. Springer, 2000.

[6] M. R. Garey, D. S. Johnson Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman, 1979.

[7] Ingo Wegener. Theoretische Informatik. Teubner, 2005.