Skript zur Vorlesung

Berechenbarkeit und Komplexität

an der RWTH Aachen

Prof. Dr. Berthold Vöcking
Lehrstuhl Informatik 1

November 2010

Inhaltsverzeichnis

Kapitel 1

Einführung

Das vorliegende Skript ist über mehrere Semester entstanden. An der Erstellung des Skriptes haben insbesondere Helge Bals, Nadine Bergner, Dirk Bongartz, Marcel Ochel und Alexander Skopalik mitgewirkt.

In den meisten Vorlesungen des Informatikstudiums wird vermittelt, wie verschiedenste Probleme mit dem Computer gelöst werden können. Man könnte den Eindruck gewinnen, dass die Möglichkeiten des Computers unbeschränkt sind. Diese Vorlesung ist anders: Wir werden die Grenzen des Computers erkunden und zahlreiche scheinbar einfache Probleme kennen lernen, über die wir beweisen können, dass sie nicht durch den Computer gelöst werden können.

- Im ersten Teil der Vorlesung beschäftigen wir uns mit fundamentalen Fragestellungen aus dem Gebiet der *Rekursions-* bzw. *Berechenbarkeitstheorie*. Hier geht es um die absoluten Grenzen des Rechners. Wir lernen Probleme kennen, die nicht durch einen Rechner gelöst werden können, egal wieviel Rechenzeit und Speicherplatz wir zur Verfügung stellen.

- Das Thema des zweiten Teils der Vorlesung ist die *Komplexitätstheorie*. Wir untersuchen, welche Probleme einen effizienten Algorithmus zulassen, also welche Probleme sich in akzeptabler Zeit durch einen Rechner lösen lassen. Zu diesem Zweck werden wir sogenannte Komplexitätsklassen einführen, insbesondere die Klassen P und NP, auf deren Basis wir die Schwierigkeit verschiedener Probleme charakterisieren können.

Um formale Aussagen über *Probleme* und *Computer* bzw. *Rechner* machen zu können, benötigen wir zunächst eine mathematische Modellierung dieser bei-

den Konzepte. Wir beginnen damit zu erläutern, was wir unter einem Problem verstehen und stellen dann verschiedene Rechnermodelle vor.

1.1 Modellierung von Problemen

Algorithmische Probleme kann man spezifizieren, indem man festlegt, welche Ausgaben zu welchen Eingaben berechnet werden sollen. Ein- und Ausgaben sind dabei endliche Wörter über einem jeweils festgelegtem endlichem Alphabet Σ. Typischerweise gilt $\Sigma = \{0,1\}$. Die Menge aller Wörter der Länge k über Σ bezeichnen wir als Σ^k. Beispielsweise gilt

$$\{0,1\}^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}.$$

Das sogenannte *leere Wort*, also das Wort der Länge 0, bezeichnen wir mit ϵ, d.h. $\Sigma^0 = \{\epsilon\}$. Sei $\mathbb{N} = \{0, 1, 2, \ldots\}$ die Menge der natürlichen Zahlen. Die Menge $\Sigma^* = \bigcup_{k \in \mathbb{N}} \Sigma^k$ ist der sogenannte *Kleenesche Abschluß* von Σ und enthält alle Wörter über Σ, die wir z.B. der Länge nach aufzählen können:

$$\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, \ldots \ .$$

Diese Reihenfolge wird im Folgenden als *kanonische Reihenfolge* bezeichnet.

Probleme sind „Aufgaben“ oder „Fragestellungen“, die wir mit dem Rechner lösen oder beantworten wollen. Um zu verdeutlichen, welche Arten von Aufgaben wir meinen, geben wir einige einfache Beispiele. Beispielsweise möchte man einen Primfaktor zu einer gegebenen Zahl bestimmen.

Problem 1.1 (Primfaktorbestimmung)

Eingabe: *eine binär kodierte Zahl* $q \in \mathbb{N}$, $q \geq 2$

Ausgabe: *ein binär kodierter Primfaktor von* q

Hier sind beispielsweise zur Eingabe 6 die Ausgaben 2 oder 3 möglich, genauer gesagt natürlich die Binärdarstellungen dieser Zahlen. Die Binärdarstellung einer natürlichen Zahl k bezeichnen wir mit $bin(k)$. Das Problem entspricht dann der Relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$ mit

$$\begin{aligned} R \ = \ \{(x,y) \in \{0,1\}^* \times \{0,1\}^* \mid\ & x = bin(q), y = bin(p), \\ & p, q \in \mathbb{N}, q \geq 2, p \text{ prim}, p \text{ teilt } q\} \ . \end{aligned}$$

Beim Problem der Primfaktorbestimmung gibt es zu einer Eingabe in der Regel mehrere mögliche korrekte Ausgaben. Deshalb haben wir dieses Problem in Form einer Relation modelliert. Für viele Probleme gibt es zu jeder Eingabe eine eindeutige Ausgabe, wie beispielsweise bei der Multiplikation.

Problem 1.2 (Multiplikation)

Eingabe: *zwei binär kodierte Zahlen* $a, b \in \mathbb{N}$

Ausgabe: *die binär kodierte Zahl* $c \in \mathbb{N}$ *mit* $c = a \cdot b$

Derartige Probleme können wir als Funktion $f : \Sigma^* \to \Sigma^*$ beschreiben. Die zur Eingabe $x \in \Sigma^*$ gesuchte Ausgabe ist $f(x) \in \Sigma^*$. Im Fall der Multiplikation werden die Zahlen a und b binär kodiert. Um die Zahlen a und b in der Eingabe voneinander trennen zu können, erweitern wir das Alphabet um ein Trennsymbol, d.h. wir setzen $\Sigma = \{0, 1, \#\}$. Die kodierte Eingabe ist dann $bin(a)\#bin(b)$ und die Ausgabe ist $f(bin(a)\#bin(b)) = bin(c)$.

Viele Probleme lassen sich als „Ja-Nein-Fragen“ formulieren. Derartige *Entscheidungsprobleme* sind von der Form $f : \Sigma^* \to \{0, 1\}$, wobei wir 0 als „Nein“ und 1 als „Ja“ interpretieren. Sei $L = f^{-1}(1) \subseteq \Sigma^*$ die Menge derjenigen Eingaben, die mit „Ja“ beantwortet werden. Die Menge L ist eine Teilmenge der Wörter über dem Alphabet Σ, d.h. $L \subseteq \Sigma^*$. Eine solche Teilmenge wird als *Sprache* bezeichnet. Wir geben ein Beispiel.

Problem 1.3 (Graphzusammenhang)

Eingabe: *die Kodierung eines Graphen* $G = (V, E)$

Ausgabe: *das Zeichen 1, falls* G *zusammenhängend ist, ansonsten das Zeichen 0.*

Den Graphen können wir über dem Alphabet $\{0, 1\}$ kodieren, indem wir seine Adjazenzmatrix angeben. In der obigen Problembeschreibung haben wir bereits implizit angenommen, dass die Eingabe wirklich der korrekten Kodierung eines Graphen entspricht. Natürlich gibt es auch Eingaben, die keiner Adjazenzmatrix entsprechen, etwa wenn die Länge des Binärstrings keine quadratische Zahl ist. Bei der Beschreibung des Problems als Sprache müssen wir dies explizit berücksichtigen. Sei $\mathcal{G}$ die Menge aller Graphen und $\mathcal{G}_z$ die Menge aller zusammenhängender Graphen und $code(G)$ die Kodierung eines Graphen $G \in \mathcal{G}$. Dann ist

$$L \;=\; \{w \in \{0,1\}^* \mid \exists G \in \mathcal{G}_z : w = code(G)\} \;.$$

Die Sprache L enthält alle Eingaben, die einen zusammenhängenden Graphen kodieren. Das *Komplement* von L ist definiert als $\bar{L} = \{0,1\}^* \setminus L$. Die Sprache $\bar{L}$ enthält Eingaben, die keiner korrekten Kodierung eines Graphen entsprechen, also syntaktisch inkorrekt sind, oder Eingaben, die einen nicht zusammenhängenden Graphen kodieren.

Im ersten Teil der Vorlesung werden wir uns größtenteils mit Entscheidungsproblemen beschäftigen. Wir untersuchen, welche Entscheidungsprobleme berechenbar sind, also von Rechnern gelöst werden können, und welche nicht berechenbar sind, also nicht von Rechnern gelöst werden können, egal wieviel Rechenzeit oder Speicherplatz zur Verfügung stehen.

1.2 Computermodelle

1.2.1 Die Turingmaschine

Wir präsentieren nun ein sehr einfaches Modell eines Computers bzw. Rechners, das Modell der Turingmaschine, welches von Alan Turing im Jahr 1936 eingeführt wurde. Dieses Modell sieht auf den ersten Blick sehr rudimentär und eingeschränkt aus. Wir werden im Laufe des Kapitels einsehen, dass dieses einfache Modell sehr mächtig ist und all diejenigen Probleme lösen kann, die auch heute existierende Rechner lösen können, selbst wenn wir optimistischer Weise annehmen, dass diesen Rechnern eine unbeschränkte Rechenzeit und ein unbeschränkter Speicher zur Verfügung steht.

Definition des Maschinenmodells

Eine *Turingmaschine (TM)* arbeitet auf einem beidseitig unendlichem *Speicherband*. Dieses Band besteht aus einzelnen *Zellen*, die jeweils ein Element eines vorgegebenen, endlichen *Bandalphabets* Γ enthalten. In vielen Fällen ist es sinnvoll, $\Gamma = \{0, 1, B\}$ zu setzen, wobei die Buchstaben 0 und 1 ein 0-Bit bzw. ein 1-Bit repräsentieren. Der Buchstabe B heißt *Leerzeichen (Blank)* und repräsentiert eine leere, unbeschriebene Zelle. Die Turingmaschine besitzt einen Schreib-/Lesekopf, der über das Band läuft und dabei die Inschriften in einzelnen Zellen lesen und verändern kann. Neben dem Speicherband hat die Turingmaschine noch einen aktuellen *Zustand* q, der Element einer endlichen Zustandsmenge Q ist.

Am Anfang der Rechnung befindet sich die TM im *Anfangszustand* $q_0 \in Q$, auf dem Band steht von links nach rechts die *Eingabe* in benachbarten Zellen,

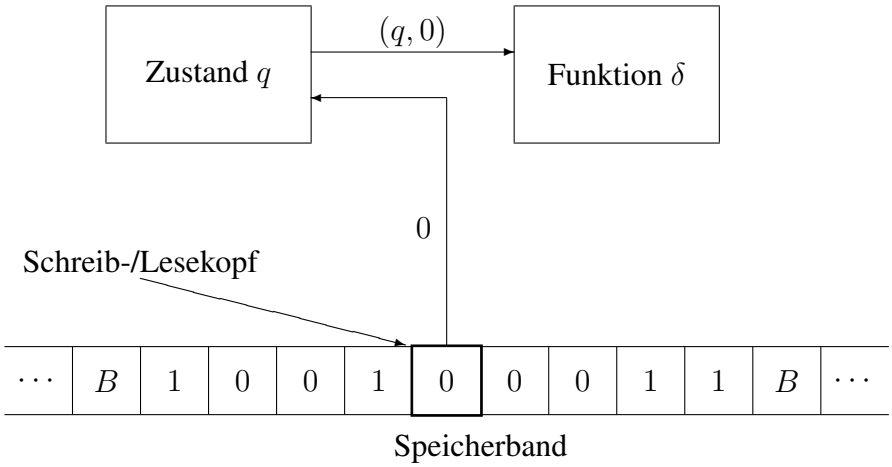

Abbildung 1.1: Vor dem Zustandsübergang $\delta(q, 0) = (q', 1, L)$.

alle anderen Zellen links und rechts der Eingabe enthalten das Leerzeichen, und der Kopf steht unter dem ersten Zeichen der Eingabe. Das Eingabealphabet Σ ist eine Teilmenge vom Bandalphabet Γ, die insbesondere nicht das Leerzeichen B enthält. Die Arbeitsweise der TM wird durch eine *Zustandsübergangsfunktion* $\delta : (Q \setminus \{\bar{q}\}) \times \Gamma \to Q \times \Gamma \times \{R, L, N\}$ beschrieben. Hierbei bezeichnet $\bar{q}$ den sogenannten *Stoppzustand*. Wenn $\delta(q, a) = (q', a', d)$ gilt, hat das folgende Interpretation: Wenn der aktuelle Zustand der Turingmaschine q ist und der Schreib-/Lesekopf auf eine Speicherzelle mit Inhalt a zeigt, dann schreibt die Turingmaschine den Buchstaben a' in die aktuelle Speicherzelle, wechselt in den Zustand q', und der Schreib-/Lesekopf geht eine Position nach rechts oder links, oder bleibt stehen, je nachdem ob $d = R$, $d = L$ oder $d = N$ gilt. Dieser Vorgang wird als *Rechenschritt* bezeichnet. Es wird solange ein Rechenschritt nach dem anderen ausgeführt, bis der Stoppzustand $\bar{q}$ erreicht ist.

In den Abbildungen 1.1 und 1.2 ist ein solcher Rechenschritt dargestellt. Dabei visualisiert Abbildung 1.1 die Situation vor einem bestimmten Zustandsübergang und Abbildung 1.2 die Situation nach dem Zustandsübergang. Die in diesen Abbildungen dargestellte Turingmaschine hat das Bandalphabet $\Gamma = \{0, 1, B\}$ und das Eingabealphabet $\Sigma = \{0, 1\}$. Die Zustandsmenge Q enthält unter anderem die Zustände q und q'. In Abbildung 1.1 befindet sich die Turingmaschine im Zustand q, auf dem Speicherband stehen Elemente aus Γ, und der Schreib-/Lesekopf steht auf einem Speicherplatz mit Inhalt a. In einem Rechenschritt wird der Inhalt des

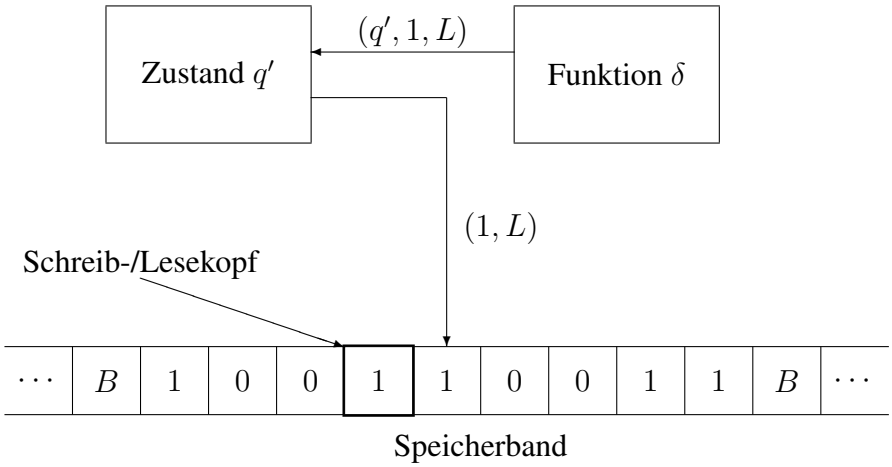

Abbildung 1.2: Nach dem Zustandübergang.

aktuellen Speicherplatzes, also a, ausgelesen, und zusammen mit dem aktuellen Zustand, also q, in die Übergangsfunktion δ „hineingesteckt“. In unserem Beispiel sei nun $\delta(q, 0) = (q', 1, L)$, d.h. die TM geht in den Zustand q' über, schreibt das Zeichen 1 an die aktuelle Kopfposition und bewegt dann den Schreib-/Lesekopf um eine Zelle nach links. Das Ergebnis des Rechenschrittes ist in Abbildung 1.2 dargestellt.

Wir fassen zusammen. Eine TM wird durch Angabe der folgenden Komponenten beschrieben:

- Q, die endliche Zustandsmenge
- $\Sigma \supseteq \{0, 1\}$, das endliche Eingabealphabet
- $\Gamma \supset \Sigma$, das endliche Bandalphabet
- $B \in \Gamma \setminus \Sigma$, das Leerzeichen (Blank)
- $q_0 \in Q$, der Anfangszustand
- $\bar{q} \in Q$, der Stoppzustand
- die Zustandsübergangsfunktion

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

Die TM ist definiert durch das 7-Tupel $(Q, \Sigma, \Gamma, B, q_0, \bar{q}, \delta)$. Die sogenannte Konfiguration einer TM entspricht einem „Schnappschuss“ der Rechnung zu einem bestimmten Zeitpunkt. Die Konfiguration beschreibt alle Details, die benötigt werden, um die Rechnung fortzusetzen, also den Zustand, die Kopfposition und die Bandinschrift. Wir formalisieren dies folgendermaßen.

- Eine *Konfiguration* einer TM ist ein String $\alpha q \beta$, mit $q \in Q$ und $\alpha, \beta \in \Gamma^*$. Bedeutung: auf dem Band steht $\alpha\beta$ eingerahmt von Blanks, der Zustand ist q, und der Kopf steht unter dem ersten Zeichen von β.
- $\alpha' q' \beta'$ ist *direkte Nachfolgekonfiguration* von $\alpha q \beta$, falls $\alpha' q' \beta'$ in einem Rechenschritt aus $\alpha q \beta$ entsteht. Wir schreiben $\alpha q \beta \vdash \alpha' q' \beta'$.
- $\alpha'' q'' \beta''$ ist *Nachfolgekonfiguration* von $\alpha q \beta$, falls $\alpha'' q'' \beta''$ in endlich vielen Rechenschritten aus $\alpha q \beta$ entsteht. Wir schreiben $\alpha q \beta \vdash^* \alpha'' q'' \beta''$. Es gilt insbesondere $\alpha q \beta \vdash^* \alpha q \beta$.

Die Rechnung *terminiert*, sobald der Stoppzustand $\bar{q}$ erreicht ist. In diesem Fall gibt es keine Nachfolgekonfiguration. Die *Laufzeit* einer TM-Rechnung ist die Zahl der Rechenschritte, die die TM bis zur Terminierung ausführt. Falls die Rechnung nicht terminiert, also den Stoppzustand nie erreicht, so ist die Laufzeit unbeschränkt. Der *Platzbedarf* ist die Zahl der besuchten Speicherzellen. Auch der Platzbedarf kann unbeschränkt sein.

TM-berechenbare Funktionen und Sprachen

Eine TM M, die für jede Eingabe terminiert, berechnet eine totale Funktion $f_M : \Sigma^* \to \Sigma^*$. Wenn der Stoppzustand $\bar{q}$ erreicht ist, so kann das Ergebnis, also $f_M(x)$, folgendermaßen vom Speicherband abgelesen werden: Der erste Buchstabe des Ergebnisses steht unter dem Schreib-/Lesekopf. Das Ergebnis wird nach rechts durch den ersten Buchstaben begrenzt, der nicht in Σ liegt, z.B. den Blankbuchstaben B. Insbesondere ist das Ergebnis das leere Wort ϵ, wenn der Kopf zum Schluss der Rechnung auf eine Symbol aus $\Gamma \setminus \Sigma$ zeigt. Es kann jedoch passieren, dass eine TM nicht immer terminiert. Im Allgemeinen berechnet eine TM M deshalb eine Funktion $f_M : \Sigma^* \to \Sigma^* \cup \{\perp\}$. Das Symbol $\perp$ (Bottom) steht dabei für „nicht definiert“. Eine Eingabe $x \in \Sigma^*$ wird auf $\perp$ abgebildet, wenn M auf der Eingabe x nicht terminiert.

Definition 1.4 *Eine Funktion $f : \Sigma^* \to \Sigma^* \cup \{\perp\}$ heißt* TM-berechenbar *bzw.* rekursiv*, wenn es eine TM M mit $f = f_M$ gibt.*

Der Begriff der *Rekursivität* geht darauf zurück, dass man die Klasse der TM-berechenbaren Funktionen auch durch eine Menge von induktiven Kompositionsregeln ohne direkten Bezug zu einem Maschinenmodell definieren kann, die unter anderem auch das rekursive Verschachteln von Funktionen umschließt.

Beachte, dass die Menge der TM-berechenbaren bzw. rekursiven Funktionen auch partielle Funktionen einschließt, also solche Funktionen, bei denen einige Eingaben keine definierten Ausgaben besitzen bzw. auf das Zeichen $\perp$ abgebildet werden.

Wie bereits in Abschnitt 1.1 erwähnt, will man häufig nur Entscheidungsprobleme lösen, d.h. man erwartet von der Turingmaschine eine Ja-Nein-Antwort. Wenn die Rechnung terminiert und das Ergebnis mit einer 1 beginnt, gilt das als Ja-Antwort (*Akzeptieren*). Wenn die Rechnung terminiert aber das Ergebnis nicht mit einer 1 beginnt, so gilt das als Nein-Antwort (*Verwerfen*). Für den Spezialfall von Entscheidungsproblemen (Sprachen) können wir den Begriff der Rekursivität wie folgt fassen.

Definition 1.5 *Eine Sprache $L \subseteq \Sigma^*$ heißt* TM-entscheidbar *bzw.* rekursiv*, wenn es eine TM gibt, die auf allen Eingaben stoppt und die Eingabe w akzeptiert, wenn $w \in L$ ist, und die Eingabe w verwirft, wenn $w \notin L$ ist.*

Beispiele für Turingmaschinen

Erstes Beispiel: Wir betrachten die Sprache $L = \{w0 \,|\, w \in \{0,1\}^*\}$. Diese Sprache enthält alle Binärwörter, die mit einer 0 enden. Das Entscheidungsproblem L wird gelöst durch die TM

$$M = (\{q_0, q_1, \bar{q}\}, \{0,1\}, \{0,1,B\}, B, q_0, \bar{q}, \delta) \ ,$$

wobei δ durch folgende Tabelle gegeben ist:

δ	0	1	B
q_0	$(q_0, 0, R)$	$(q_1, 1, R)$	$(\bar{q}, 1, N)$
q_1	$(q_0, 0, R)$	$(q_1, 1, R)$	$(\bar{q}, 0, N)$

Am Anfang steht der Kopf der TM auf dem ersten Buchstaben der Eingabe. Die TM geht in jedem Schritt nach rechts, sofern sie nicht stoppt. Dabei merkt sie sich in ihrem Zustand, ob der jeweils zuletzt gelesene Buchstabe eine 1 war (dann ist der Zustand q_1) oder nicht (dann ist der Zustand q_0). Wenn die TM das erste Mal auf einen Blank trifft, ist das Ende der Eingabe erreicht. Die TM kann dann an

ihrem Zustand erkennen, ob das Eingabewort in der Sprache L liegt oder nicht, und dementsprechend durch Schreiben einer 1 akzeptieren oder durch Schreiben einer 0 verwerfen.

Die Rechnung für eine bestimmte Eingabe kann nachvollzogen werden, indem man die Konfigurationsfolge angibt. Beispielsweise ergibt sich für die Eingabe 0110 die folgende Konfigurationsfolge:

$$q_0 0110 \vdash 0q_0 110 \vdash 01q_1 10 \vdash 011q_1 0 \vdash 0110q_0 B \vdash 0110\bar{q}1$$

Da der Kopf am Ende unter (vor) einer 1 steht bedeutet dies, dass die Maschine die Eingabe 0110 akzeptiert.

Zweites Beispiel: Als weiteres, etwas umfangreicheres Beispiel entwickeln wir eine TM für die Sprache $L = \{0^n 1^n \mid n \geq 1\}$. Wir setzen $Q = \{q_0, \ldots, q_6\}$, $\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, B\}$ und überlegen uns nun eine korrekte Zustandsübergangsfunktion. Unsere TM arbeitet in zwei Phasen:

- *Phase 1:* Teste, ob die Eingabe von der Form $0^i 1^j$ für $i \geq 0$ und $j \geq 1$ ist.
- *Phase 2:* Teste, ob $i = j$ gilt.

Phase 1 verwendet die Zustände q_0 und q_1 und wechselt bei Erfolg nach q_2. Phase 2 verwendet die Zustände q_2 bis q_6 und akzeptiert bei Erfolg. Eingeschränkt auf q_0 und q_1 hat δ folgende Gestalt:

δ	0	1	B
q_0	$(q_0, 0, R)$	$(q_1, 1, R)$	$(\bar{q}, 0, N)$
q_1	$(\bar{q}, 0, N)$	$(q_1, 1, R)$	(q_2, B, L)

Wieso implementiert dieser Teil von δ die erste Phase? Die TM läuft von links nach rechts über die Eingabe, bis ein Zeichen ungleich 0 gefunden wird. Falls dieses Zeichen eine 1 ist, geht sie über in den Zustand q_1. Wenn dieses Zeichen ein Blank ist, so ist die Eingabe von der Form 0^i mit $i \geq 0$, und die TM verwirft. Im Zustand q_1 geht die TM weiter nach rechts bis zum ersten Zeichen ungleich 1. Wenn das soeben gelesene Zeichen eine 0 ist, hat die Eingabe die Form $0^i 1^j 0\{0, 1\}^*$ mit $i \geq 0$ und $j \geq 1$, und die TM verwirft. Ist das soeben gelesene Zeichen ein Blank, so ist die erste Phase erfolgreich beendet, und die TM geht nach links, so dass der Kopf auf der rechtesten 1 steht. Außerdem wechselt die TM in den Zustand q_2 und startet somit die zweite Phase.

Die zweite Phase spielt sich auf den Zuständen $\{q_2, \ldots, q_6\}$ ab. Eingeschränkt auf diese Zustände hat δ folgende Gestalt:

δ	0	1	B
q_2	$(\bar{q}, 0, N)$	(q_3, B, L)	$(\bar{q}, 0, N)$
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_4, B, R)
q_4	(q_5, B, R)	$(\bar{q}, 0, N)$	$(\bar{q}, 0, N)$
q_5	$(q_6, 0, R)$	$(q_6, 1, R)$	$(\bar{q}, 1, N)$
q_6	$(q_6, 0, R)$	$(q_6, 1, R)$	(q_2, B, L)

In der zweiten Phase löscht die TM wiederholt die rechteste 1 und die linkeste 0, bis das Band leer ist. Wenn zu einer 1 keine entsprechende 0 gefunden wird oder umgekehrt, unterscheidet sich die Anzahl der Nullen von der Anzahl der Einsen, und die TM verwirft. Wenn zu jeder 1 eine entsprechende 0 gefunden wird, ist die Anzahl der Einsen gleich der Anzahl der Nullen, und die TM akzeptiert. Wir verdeutlichen die genaue Implementierung dieses Vorgehens, indem wir die Bedeutung der einzelnen Zustände erklären:

q_2 : Der Kopf steht auf dem rechtesten Nichtblank. Falls dieses Zeichen eine 1 ist, so lösche es, gehe nach links und wechsel in den Zustand q_3. Sonst verwirf die Eingabe.

q_3 : Bewege den Kopf auf das linkeste Nichtblank. Wechsel nach q_4.

q_4 : Falls das gelesene Zeichen eine 0 ist, ersetze es durch ein Blank und gehe nach q_5, sonst verwirf die Eingabe.

q_5 : Wir haben jetzt die linkeste 0 und die rechteste 1 gelöscht. Falls das Restwort leer ist, dann akzeptiere, sonst gehe in den Zustand q_6.

q_6 : Laufe wieder zum rechtesten Nichtblank und starte erneut in q_2.

Im Zustand q_2 muss der Kopf auf dem rechtesten Nichtblank stehen. Wie oben erläutert, ist dies auch am Anfang der zweiten Phase der Fall. Somit arbeitet die Turingmaschine korrekt.

High-Level-Beschreibungen von Turingmaschinen

Die präsentierten Beispiele von konkreten Turingmaschinen haben gezeigt, dass die Beschreibung der Arbeitsweise einer TM durch die konkrete Angabe der Übergangsfunktion sehr umständlich sein kann. In diesem Abschnitt überlegen wir uns, wie man die Arbeitsweise einer Turingmaschine auf sinnvolle Art und Weise auf

einem höheren Abstraktionsniveau beschreiben kann, so dass einerseits die Beschreibung relativ einfach und intuitiv verständlich ist, aber andererseits klar bleibt wie man sie technisch umsetzen kann.

Als erstes zeigen wir, dass man bei der Beschreibung des Programmes einer TM davon ausgehen kann, dass man eine Variable mit konstant großem Zustandsbereich im Zustandsraum abspeichern kann. Wenn man im Programm einer Turingmaschine beispielsweise auf ein Speicherbit zugreifen möchte, so kann man dies technisch realisieren indem man den Zustandsraum Q erweitert zu $Q' = Q \times \{0, 1\}$. Ganz analog kann man einen Zähler mit Werten zwischen 0 und einer Konstante k implementieren, indem man den Zustandsraum Q erweitert zu $Q' = Q \times \{0, \ldots, k\}$. Die Zahl k muss eine Konstante sein, weil der Zustandsraum einer TM endlich ist. Insbesondere kann man nicht die Länge der Eingabe im Zustandsraum speichern.

Ein weiterer Programmiertrick: Bei einer k-spurigen TM handelt es sich um eine TM, bei der das Band in k sogenannte Spuren eingeteilt ist, d.h. in jeder Bandzelle stehen k Zeichen, die der Kopf gleichzeitig einlesen kann. Das können wir erreichen, indem wir das Bandalphabet um k-dimensionale Vektoren erweitern, z.B.

$$\Gamma' := \Sigma \cup \Gamma^k \ .$$

Die Verwendung einer mehrspurigen TM erlaubt es häufig, Algorithmen einfacher zu beschreiben. Wir verdeutlichen dies am Beispiel der Addition. Aus der Eingabe $bin(i_1)\#bin(i_2)$ für $i_1, i_2 \in \mathbb{N}$ soll $bin(i_1 + i_2)$ berechnet werden. Wir verwenden eine 3-spurige TM mit den Alphabeten $\Sigma = \{0, 1, \#\}$ und

$$\Gamma = \left\{ 0, 1, \#, \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \ldots, \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, B \right\} \ .$$

Unsere Implementierung für das Addieren zweier Zahlen arbeitet in drei Phasen. In Phase 1 wird die Eingabe so zusammengeschoben, dass die Binärkodierungen von i_1 und i_2 in der ersten und zweiten Spur rechtsbündig übereinander stehen. Aus der Eingabe $0011\#0110$ wird beispielsweise

$$B^* \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} B^* \ .$$

In Phase 2 addiert die TM die beiden Zahlen nach der Schulmethode, indem der Kopf das Band von rechts nach links durchläuft. Überträge werden im Zustand

gespeichert (Wieso geht das?). Als Ergebnis auf Spur 3 ergibt sich in unserem Beispiel

$$B^* \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} B^* \ .$$

In Phase drei wird das Ergebnis der Rechnung ins Einspur-Format zurücktransformiert. Dabei werden die Inhalte der ersten beiden Spuren ignoriert, auf dem Band steht der Inhalt der dritten Spur, kodiert durch Σ. Die TM muss schließlich noch zum linkesten Buchstaben des Ergebnisses laufen, weil dies von TMn, die Funktionen berechnen, so gefordert wird.

Programmkonstrukte aus höheren Programmiersprachen können auch auf TMn implementiert werden:

- *Schleifen* haben wir bereits implizit in den Beispielen für TMn gesehen.
- *Variablen* können realisiert werden, indem wir pro Variable eine Spur des Bandes reservieren oder auch mehrere Variablen auf derselben Spur abspeichern. Auf diese Art können wir beispielsweise auch Felder (Arrays) abspeichern.
- *Unterprogramme* können implementiert werden, indem wir eine Spur des Bandes zur Realisierung eines Prozedurstacks reservieren.

Diese Techniken erlauben es uns, auch auf bekannte Algorithmen beispielsweise zum Sortieren von Daten zurückzugreifen.

Mehrband-Turingmaschinen

In diesem Abschnitt wird zunächst die Mehrband-TM als Verallgemeinerung der TM eingeführt. Es zeigt sich jedoch, dass die Berechnungskraft durch das Hinzufügen zusätzlicher Bänder nicht steigt.

Definition 1.6 (k-Band TM) *Eine k-Band-TM ist eine Verallgemeinerung der Turingmaschine und verfügt über k Speicherbänder mit jeweils einem unabhängigen Kopf. Die Zustandsübergangsfunktion ist von der Form*

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

Band 1 fungiert als Ein-/Ausgabeband wie bei der (1-Band) TM. Die Bänder $2, \ldots, k$ sind initial mit B^* beschrieben. Die sonstige Funktionsweise einer k-Band TM ist analog zur TM. Die Laufzeit und der Platzbedarf einer k-Band TM sind ebenfalls analog zur TM definiert.

Bei der in Abschnitt 1.2.1 besprochenen mehrspurigen TM handelte es sich lediglich um eine hilfreiche Interpretation der (1-Band) TM. Die k-Band TM ist hingegen eine echte Verallgemeinerung. Der wesentliche Unterschied liegt darin, dass die Lese-/Schreib-Köpfe der einzelnen Bänder sich unabhängig voneinander bewegen können. Dennoch ist eine k-Band-TM nicht wirklich mächtiger als eine normale TM, wie der folgende Satz zeigt:

Satz 1.7 *Eine k-Band TM M, die mit Rechenzeit $t(n)$ und Platz $s(n)$ auskommt, kann von einer (1-Band) TM M' mit Zeitbedarf $O(t^2(n))$ und Platzbedarf $O(s(n))$ simuliert werden.*

Beweis: Die TM M' verwendet $2k$ Spuren. Nach Simulation des t-ten Schrittes für $0 \leq t \leq t(n)$ gilt

- Die ungeraden Spuren $1, 3, \ldots, 2k-1$ enthalten den Inhalt der Bänder $1, \ldots, k$ von M.
- Auf den geraden Spuren $2, 4, \ldots, 2k$ sind die Kopfpositionen von M auf diesen Bändern mit dem Zeichen $\#$ markiert.

Diese Invarianten sind in Abbildung 1.3 visualisiert. Die Initialisierung der Spuren ist in konstanter Zeit möglich.

Jeder Rechenschritt von M wird durch M' wie folgt simuliert: Am Anfang steht der Kopf von M' auf dem linkesten $\#$ und M' kennt den Zustand von M. Der Kopf von M' läuft nach rechts bis zum rechtesten $\#$, wobei die k Zeichen an den mit $\#$ markierten Spurpositionen im Zustand abgespeichert werden. Es ist kein Problem für M', das linkeste (bzw. rechteste) $\#$ zu erkennen, da es nur konstant viele $\#$-Markierungen gibt und M' sich somit in seinem Zustand merken kann, wie viele Markierungen jeweils links (bzw. rechts) von der aktuellen Kopfposition stehen. Am rechtesten $\#$-Zeichen angekommen, kann M' die Übergangsfunktion von M auswerten und kennt den neuen Zustand von M sowie die erforderlichen Bewegungen auf den k Bändern. Nun läuft der Kopf von M' zurück, verändert dabei die Bandinschriften an den mit $\#$ markierten Stellen und verschiebt, falls erforderlich, auch die $\#$-Markierungen um eine Position nach links oder rechts. Man mache sich klar, dass alles, was M' sich merken muss, konstante Größe hat und somit im Zustand von M' gespeichert werden kann.
Laufzeitanalyse: Wieviele Bandpositionen können zwischen der linkesten und der rechtesten $\#$-Markierung liegen? Nach t Schritten ($t \geq 1$) können diese Markierungen höchstens $2t$ Positionen auseinanderliegen, da jede der beiden Markierungen pro Schritt höchstens um eine Position nach außen wandert. Also ist der

die 2 *Bänder* der simulierten 2-Band TM

$\cdots$	B	b	l	a	$-$	b	a	n	d	1	B	$\cdots$

$\cdots$	B	B	b	a	n	d	2	B	B	B	B	$\cdots$

die 4 *Spuren* der simulierenden TM

$\cdots$	B	b	l	a	$-$	b	a	n	d	1	B	$\cdots$
$\cdots$	B	B	B	B	B	B	B	$\#$	B	B	B	$\cdots$
$\cdots$	B	B	b	a	n	d	2	B	B	B	B	$\cdots$
$\cdots$	B	B	$\#$	B	B	B	B	B	B	B	B	$\cdots$

Abbildung 1.3: Simulation von k Bändern mit $2k$ Spuren, hier $k = 2$. Die Kopfpositionen sind durch fette Umrandungen angedeutet.

Abstand zwischen diesen Zeichen und somit auch die Laufzeit zur Simulation eines Schrittes durch $O(t(n))$ beschränkt. Insgesamt ergibt das zur Simulation von $t(n)$ Schritten eine Laufzeitschranke von $O(t(n)^2)$.
Analyse des Platzbedarfs: Der Platzbedarf ist linear, weil zu jeder Speicherzelle, die die simulierende TM besucht, mindestens eine Speicherzelle gehört, die die simulierte k-Band TM besucht. □

Die Universelle Turingmaschine

Bisher haben wir für jedes Problem eine eigene TM entworfen, einen *special purpose* Rechner. Real existierende Maschinen sind jedoch programmierbare *general purpose* Rechner. Wir konstruieren jetzt eine programmierbare Variante der TM, die sogenannte *universelle TM*.

Die universelle TM U simuliert eine beliebige andere TM M auf einer belie-

bigen Eingabe w. Als Eingabe erhält U einen String der Form $\langle M \rangle w$, wobei der String $\langle M \rangle$ die simulierte Turingmaschine M kodiert. Der String $\langle M \rangle$ wird als *Gödelnummer* bezeichnet. Wir definieren gleich, wie eine *Gödelnummer* genau aussieht. Die universelle TM simuliert das Verhalten der TM M auf der Eingabe w und übernimmt deren Akzeptanzverhalten. Bei inkorrekter Eingabe (d.h. die Eingabe beginnt nicht mit einer Gödelnummer) gibt U eine Fehlermeldung aus.

Definition 1.8 *Als* Gödelnummerierung *bezeichnet man eine injektive Abbildung von der Menge aller TMn in die Menge* $\{0,1\}^*$.

Jeder TM wird durch die Gödelnummerierung somit ein Binärstring zugeordnet. Natürlich können wir den String als Zahl interpretieren, daher die Bezeichnung „Nummerierung".

Die Definition erlaubt viele verschiedene Gödelnummerierungen, und es wird deutlich werden, dass es auch viele verschiedene Gödelnummerierungen gibt. Wir werden aber im Folgenden stets von *der* Gödelnummer einer TM reden und diese als $\langle M \rangle$ schreiben. Diese Notation setzt implizit voraus, dass wir uns auf eine bestimmte Gödelnummerierung festgelegt haben.

Wir gehen von einer *präfixfreien* Kodierung aus. *Präfixfreiheit* bedeutet, dass keine Gödelnummer das Präfix (Anfangsteilwort) einer anderen Gödelnummer sein darf. Präfixfreiheit können wir beispielsweise erreichen, indem alle Gödelnummern auf 111 enden und ansonsten der Teilstring 111 nicht in der Kodierung vorkommt.

Wir zeigen nun, wie eine geeignete Kodierung einer TM aussehen könnte. O.B.d.A. beschränken wir uns auf TMn der folgenden Form:

- Die Menge der Zustände ist $Q = \{q_1, \ldots, q_t\}$.
- Der Anfangszustand ist q_1 (statt wie sonst q_0) und der Stoppzustand ist q_2 (d.h. $\bar{q} = q_2$).
- Das Bandalphabet ist $\Gamma = \{0, 1, B\}$. Wir nummerieren das Alphabet durch, indem wir $X_1 = 0$, $X_2 = 1$ und $X_3 = B$ setzen.
- Auch die möglichen Kopfbewegungen nummerieren wir, indem wir $D_1 = L$, $D_2 = N$ und $D_3 = R$ setzen.

Zur Beschreibung von TM dieser Form müssen wir nun die Übergangsfunktion als Binärstring kodieren. Der Übergang $\delta(q_i, X_j) = (q_k, X_\ell, D_m)$ wird kodiert durch

den Binärstring $0^i10^j10^k10^\ell10^m$. Die Kodierung des t-ten Übergangs bezeichnen wir mit *code*(t). Die Gödelnummer einer TM M mit s vielen Übergängen ist dann

$$\langle M \rangle \;=\; \textit{code(1)}\,11\,\textit{code(2)}\,11\ldots11\,\textit{code}(s)\,111\ .$$

Als Eingabe erhält die universelle TM U ein Wort der Form $\langle M\rangle w$ für beliebiges $w \in \{0,1\}^*$. Wir implementieren U zunächst in Form einer 3-Band TM. Zur Initialisierung überprüft U zunächst, ob die Eingabe mit einer korrekten Gödelnummer beginnt. Falls nicht, wird ein Fehler ausgegeben. Dann kopiert U die Gödelnummer auf Band 2 und schreibt die Binärkodierung des Anfangszustands auf Band 3. Jetzt bereitet U Band 1 so vor, dass es nur das Wort w enthält. Der Kopf steht unter dem ersten Zeichen von w. Die Initialisierung benötigt Zeit $O(1)$, wobei wir die Kodierungslänge von M als Konstante ansehen.

Während der Simulation übernehmen die 3 Bänder von U die folgenden Aufgaben:

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

Zur Simulation eines Schritts von M sucht U zu dem Zeichen an der Kopfposition auf Band 1 und dem Zustand auf Band 3 die Kodierung des entsprechenden Übergangs von M auf Band 2. Wie in der Übergangsfunktion beschrieben,

- aktualisiert U die Inschrift auf Band 1,
- bewegt U den Kopf auf Band 1, und
- verändert U den auf Band 3 abgespeicherten Zustand von M.

Die Laufzeit eines Schrittes ist konstant, also simuliert U die TM M mit konstantem Zeitverlust. Können wir dieses Ergebnis auch mit einer (1-Band) TM erreichen? Natürlich können wir die beschriebene 3-Band TM auf der 1-Band TM mit drei Spuren simulieren. Aber bei Verwendung der Simulation aus Satz 1.7 handeln wir uns einen quadratischen Zeitverlust ein. Wir erhalten eine universelle 1-Band TM mit konstantem Zeitverlust, wenn wir sowohl die Gödelnummer auf Spur 2 als auch den Zustand auf Spur 3 mit dem Kopf der TM M mitführen.

Übungsaufgabe 1.9 *Sei $M = (Q, \Sigma, \Gamma, B, q_0, \bar{q}, \delta)$ eine 1-Band TM, deren Speicherplatzbearf für eine Eingabe der Länge n maximal $s(n)$ beträgt. Zeige: Wenn M auf einer Eingabe w der Länge n hält, dann hält M auf w nach spätestens $(|Q| - 1) \cdot |\Gamma|^{s(n)} \cdot s(n) + 1$ Schritten.*

Übungsaufgabe 1.10 *Zeige, dass jede (1-Band) TM mit Laufzeitschranke $t(n)$ durch eine TM mit Laufzeitschranke $O(t(n))$ simuliert werden kann, die ein einseitig beschränktes Speicherband hat, d.h. nur Zellen rechts der initialen Kopfposition benutzt.*

1.2.2 Die Registermaschine

Die TM scheint nicht viel mit heute existierenden Rechnern zu tun zu haben. Wir führen jetzt ein Rechnermodell ein, dass eher den Vorstellungen eines Computers entspricht. Dann vergleichen wir die Berechnungskraft dieses Modells mit der Berechnungskraft der TM.

Die *Registermaschine (RAM)* ist ein Maschinenmodell, das der Assemblersprache eines modernen Rechners recht nahe kommt. Die Abkürzung RAM steht für die im englischen Sprachgebrauch verwendete Bezeichnung „Random Access Machine“. Der Speicher der RAM ist unbeschränkt und besteht aus den Registern $c(0), c(1), c(2), c(3), \ldots$. Das Register $c(0)$ heißt Akkumulator und spielt eine Sonderrolle, da die Rechenbefehle den Inhalt von $c(0)$ als implizites Argument benutzen und das Ergebnis von Rechenoperationen stets in $c(0)$ gespeichert wird. Die Inhalte der Register sind ganze Zahlen, die beliebig groß sein können. Außerdem enthält die RAM einen Befehlszähler b, der initial auf 1 steht. In Abbildung 1.4 ist eine RAM graphisch veranschaulicht.

Eine RAM arbeitet ein endliches Programm mit durchnummerierten Befehlen ab. In jedem Schritt wird der Befehl in Programmzeile b abgearbeitet. Danach wird beim Befehl GOTO i der Befehlszähler auf i gesetzt, beim Befehl END stoppt die Rechnung, ansonsten wird der Befehlszähler um eins inkrementiert. Syntax und Semantik der RAM-Befehle sind in Tabelle 1.1 definiert.

Die Eingabe steht am Anfang der Rechnung in den Registern $c(1), \ldots, c(k)$ für ein $k \in \mathbb{N}$. Die Ausgabe befindet sich nach dem Stoppen in den Registern $c(1), \ldots, c(\ell)$ für ein $\ell \in \mathbb{N}$. Der Einfachheit halber nehmen wir an, dass k und ℓ zu Beginn der Rechnung festliegen. Die RAM berechnet somit eine Funktion der Form $f : \mathbb{Z}^k \to \mathbb{Z}^\ell \cup \{\perp\}$, wobei das Zeichen $\perp$ wie bei der TM für eine nicht terminierende Rechnung steht. Die Tatsache, dass Registermaschinen Funktionen

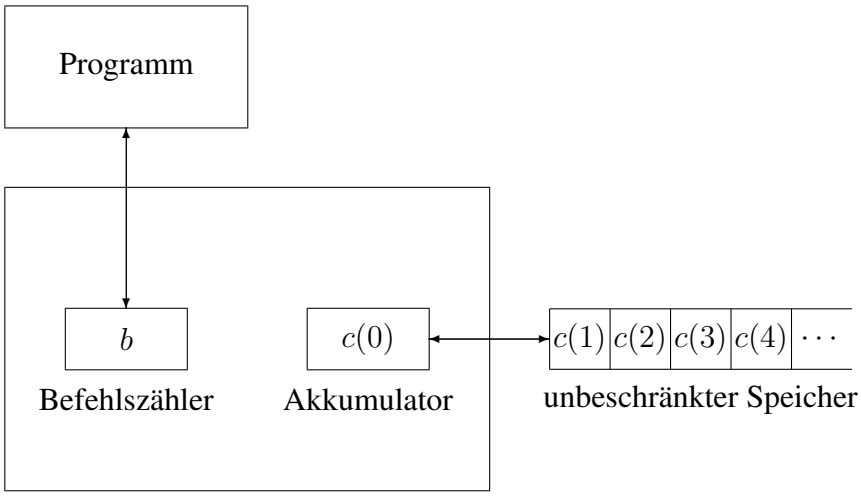

Abbildung 1.4: Graphische Veranschaulichung der RAM

über den ganzen Zahlen berechnen, stellt keinen grundsätzlichen Unterschied zum TM-Modell dar, da wir Zahlen und Binärwörter ineinander überführen können.

Auf der Registermaschine können wir offensichtlich Konstrukte wie beispielsweise Schleifen und Rekursionen, die wir von höheren Programmiersprachen gewohnt sind, realisieren. Auch komplexere Datentypen sind realisierbar. Rationale Zahlen lassen sich beispielsweise als Nenner und Zähler in zwei Registern darstellen. Die Mächtigkeit des Maschinenmodells würde sich auch nicht verändern, wenn einzelne Register die Fähigkeit hätten, rationale Zahlen zu speichern.

Eine Rechnung *terminiert*, sobald der Befehl END erreicht wird. Für die Bemessung der *Laufzeit* einer Rechnung gibt es unterschiedliche Modelle:

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

Sollte die Rechnung nicht terminieren, so ist die Laufzeit unbeschränkt.

1.2.3 Vergleich der beiden Computermodelle

Wir zeigen nun, dass sich TM und RAM gegenseitig simulieren können. Diese beiden Rechnermodelle sind also gleich mächtig, d.h. jede Funktion, die in einem

Syntax	Zustandsänderung	Änderung von b
LOAD i	$c(0) := c(i)$	$b := b + 1$
CLOAD i	$c(0) := i$	$b := b + 1$
INDLOAD i	$c(0) := c(c(i))$	$b := b + 1$
STORE i	$c(i) := c(0)$	$b := b + 1$
INDSTORE i	$c(c(i)) := c(0)$	$b := b + 1$
ADD i	$c(0) := c(0) + c(i)$	$b := b + 1$
CADD i	$c(0) := c(0) + i$	$b := b + 1$
INDADD i	$c(0) := c(0) + c(c(i))$	$b := b + 1$
SUB i	$c(0) := c(0) - c(i)$	$b := b + 1$
CSUB i	$c(0) := c(0) - i$	$b := b + 1$
INDSUB i	$c(0) := c(0) - c(c(i))$	$b := b + 1$
MULT i	$c(0) := c(0) \cdot c(i)$	$b := b + 1$
CMULT i	$c(0) := c(0) \cdot i$	$b := b + 1$
INDMULT i	$c(0) := c(0) \cdot c(c(i))$	$b := b + 1$
DIV i	$c(0) := \lfloor c(0)/c(i) \rfloor$	$b := b + 1$
CDIV i	$c(0) := \lfloor c(0)/i \rfloor$	$b := b + 1$
INDDIV i	$c(0) := \lfloor c(0)/c(c(i)) \rfloor$	$b := b + 1$
GOTO j	-	$b := j$
IF $c(0) = x$ GOTO j	-	$b := \begin{cases} j \text{ falls } c(0) = x \\ b + 1 \text{ sonst} \end{cases}$
IF $c(0) < x$ GOTO j	-	$b := \begin{cases} j \text{ falls } c(0) < x \\ b + 1 \text{ sonst} \end{cases}$
IF $c(0) \leq x$ GOTO j	-	$b := \begin{cases} j \text{ falls } c(0) \leq x \\ b + 1 \text{ sonst} \end{cases}$
END	Ende der Rechnung	-

Tabelle 1.1: Syntax und Semantik der RAM-Befehle

dieser beiden Modelle berechnet werden kann, kann auch im anderen Modell berechnet werden. Wenn man TM und RAM bezüglich des logarithmischen Kostenmaßes vergleicht, so sind auch die Laufzeiten in diesen beiden Modellen ähnlich, denn die gegenseitigen Simulationen haben nur einen polynomiellen Zeitverlust.

Satz 1.11 *Jede im logarithmischen Kostenmaß $t(n)$-zeitbeschränkte RAM kann für ein Polynom q durch eine $O(q(n+t(n)))$-zeitbeschränkte TM simuliert werden.*

Beweis: Im Beweis können wir für die Simulation eine 2-Band-TM statt einer (1-Band) TM verwenden. Wir erläutern, warum das geht: Seien $\alpha, \beta, \gamma \in \mathbb{N}$ geeignet gewählte Konstanten. Wir werden zeigen, dass die Laufzeit der Simulation der RAM mit Laufzeitschranke $t(n)$ durch eine 2-Band TM nach oben beschränkt ist durch $t'(n) = \alpha(n+t(n))^\beta$. Gemäß Satz 1.7 können wir die 2-Band TM mit Laufzeitschranke $t'(n)$ nun wiederum durch eine (1-Band) TM mit Laufzeitschranke $t''(n) = \gamma(t'(n))^2$ simulieren. Für die Simulation der RAM auf der (1-Band) TM ergibt sich somit eine Laufzeitschranke von

$$t''(n) \;=\; \gamma(t'(n))^2 \;=\; \gamma\left(\alpha(n+t(n))^\beta\right)^2 \;=\; \gamma\alpha^2 \cdot (n+t(n))^{2\beta}.$$

Diese Laufzeitschranke ist polynomiell in $n + t(n)$, weil sowohl der Term $\gamma\alpha^2$ als auch der Term 2β konstant ist. Wir fassen zusammen:

Beobachtung 1.12 *Die Klasse der Polynome ist gegen Hintereinanderausführung abgeschlossen. Deshalb können wir eine* konstante *Anzahl von Simulationen, deren Zeitverlust jeweils polynomiell nach oben beschränkt ist, ineinanderschachteln und erhalten dadurch wiederum eine Simulation mit polynomiell beschränktem Zeitverlust.*

Wir beschreiben nun die Simulation einer RAM durch eine 2-Band TM. Das Programm P der RAM bestehe aus p Programmzeilen. Für jede Programmzeile schreiben wir ein TM-Unterprogramm. Sei M_i das Unterprogramm für Programmzeile i, $1 \leq i \leq p$. Außerdem spezifizieren wir ein Unterprogramm M_0 für die Initialisierung der TM und M_{p+1} für die Aufbereitung der Ausgabe des Ergebnisses. Die *Registermaschinenkonfiguration* wird durch den Befehlszähler b und den Inhalt der Register beschrieben. Den Befehlszähler $b \in \{1, \ldots, p\}$ kann die TM im Zustand abspeichern, da p konstant ist (d.h. unabhängig von der Eingabe). Der Inhalt der benutzten Register ist allerdings nicht durch eine Konstante beschränkt.
Speicherung der Register auf einem Band der TM: Seien $0, i_1, \ldots, i_m$ die zu einem Zeitpunkt benutzten Register, also die Indizes des Akkumulators und derjenigen

anderen Register, die Teile der Eingabe enthalten oder in einem der vorhergehenden Rechenschritte angesprochen wurden. Wir verwenden eine 2-Band-TM und speichern diese Indizes und den Inhalt der entsprechenden Register in der folgenden Form auf Band 2 ab:

$$\#\#0\#bin(c(0))\#\#bin(i_1)\#bin(c(i_1))\#\#\ldots\#\#bin(i_m)\#bin(c(i_m))\#\#\# \ .$$

Beobachtung 1.13 *Die Länge des Speicherinhalts auf Band 2 ist durch $O(n + t(n))$ beschränkt, weil die RAM für jedes neue Bit, das sie erzeugt, mindestens eine Zeiteinheit benötigt.*

Beachte, dass diese Beobachtung auch für die auf dem Band gespeicherten Adressen gilt, da jede, nicht konstante Adresse, die von der RAM benutzt wird mindest einmal in den Akkumulator geladen werden muss.

Rechenschritt für Rechenschritt simuliert die TM nun die Konfigurationsveränderungen der RAM. Dazu ruft sie das durch den Programmzähler b beschriebene Unterprogramm M_b auf.

Das Unterprogramm M_b arbeitet wie folgt: M_b kopiert den Inhalt der in Programmzeile b angesprochenen Register auf Band 1, führt die notwendigen Operationen auf diesen Registerinhalten durch, kopiert dann das Ergebnis in das in Programmzeile b angegebene Register auf Band 2 zurück, und aktualisiert zuletzt den Programmzähler b.

Laufzeitanalyse: Die Initialisierung erfordert Zeit $O(n)$. Alle anderen Unterprogramme haben eine Laufzeit, die polynomiell in der Länge der Bandinschrift auf Band 2 beschränkt ist, also eine polynomielle Laufzeit in $n + t(n)$. Die Gesamtlaufzeit der Simulation ist somit auch polynomiell in $n + t(n)$ beschränkt. □

Satz 1.14 *Jede $t(n)$-zeitbeschränkte TM kann durch eine RAM simuliert werden, die uniform $O(t(n) + n)$ und logarithmisch $O((t(n) + n)\log(t(n) + n))$ zeitbeschränkt ist.*

Beweis: Wie schon bei den anderen Simulationen müssen wir die Konfiguration der simulierten Maschine (in diesem Fall der TM) im Speicher der simulierenden Maschine (der RAM) abbilden. In Übungsaufgabe 1.10 wird gezeigt, dass wir annehmen können, dass es sich um eine TM mit einseitig beschränktem Band handelt, deren Zellen mit $0, 1, 2, 3, \ldots$ durchnummeriert sind. Die Zustände und Zeichen werden ebenfalls durchnummeriert und mit ihren Nummern identifiziert,

so dass sie in den Registern abgespeichert werden können. Register 2 speichert den aktuellen Zustand. Register 1 speichert den Index der Kopfposition. Die Register $3, 4, 5, 6, \ldots$ speichern die Inhalte der Bandpositionen $0, 1, 2, 3, \ldots$, soweit diese Positionen zur Eingabe der TM gehören oder zuvor vom Kopf der TM besucht worden sind.

Die TM wird nun Schritt für Schritt durch die RAM simuliert. Zunächst wird überprüft, ob die TM im aktuellen Schritt terminiert. Falls ja, so terminiert auch die RAM mit der entsprechenden Ausgabe. Ansonsten muss der Zustandsübergang durchgeführt werden: Auf einer ersten Stufe von $|Q|$ vielen IF-Abfragen wird dazu der aktuelle Zustand selektiert. Für jeden möglichen Zustand gibt es dann eine zweite Stufe von $|\Gamma|$ vielen IF-Abfragen, die das gelesene Zeichen selektieren. Je nach Ausgang der IF-Abfragen wird dann der Zustand in Register 2 geändert, die Bandinschrift im Register mit der Adresse $c(1)$ überschrieben, und die Bandposition in Register 1 aktualisiert.
Laufzeitanalyse im uniformen Kostenmodell: Die Initialisierung kann in Zeit $O(n)$ durchgeführt werden. Die Simulation jedes einzelnen TM-Schrittes hat konstante Laufzeit. Insgesamt ist die Simulationszeit somit $O(n + t(n))$.
Laufzeitanalyse im logarithmischen Kostenmodell: Die in den Registern gespeicherten Zahlen repräsentieren Zustände, Zeichen und Bandpositionen. Zustände und Zeichen haben eine konstante Kodierungslänge. Die Bandpositionen, die während der Simulation angesprochen werden, sind durch $\max\{n, t(n)\} \leq n + t(n)$ beschränkt. Die Kodierungslänge dieser Positionen ist also $O(\log(t(n)+n))$. Damit kann die Simulation jedes einzelnen TM-Schrittes in Zeit $O(\log(t(n) + n))$ durchgeführt werden. Insgesamt ergibt sich somit eine Simulationszeit von $O((t(n) + n) \log(t(n) + n)$. □

Fazit: Die Mehrband-TM kann mit quadratischem Zeitverlust durch eine (1-Band) TM simuliert werden. TMn und RAMs (mit logarithmischen Laufzeitkosten) können sich gegenseitig mit polynomiell beschränktem Zeitverlust simulieren. Wenn es uns also nur um Fragen der Berechenbarkeit von Problemen oder um ihre Lösbarkeit in polynomieller Zeit geht, können wir wahlweise auf die TM, die Mehrband-TM oder die RAM zurückgreifen. Für positive Ergebnisse (wie etwa obere Laufzeitschranken) ist es häufig sinnvoll, Algorithmen für die RAM zu spezifizieren. Für den Nachweis negativer Ergebnisse werden wir meistens auf die TM zurückgreifen, da dieses Modell strukturell einfacher ist.

Übungsaufgabe 1.15

- *Zeige, dass jede TM durch eine RAM simuliert werden kann, die nur eine konstante Anzahl von Registern für natürliche Zahlen benutzt. Bestimme den Zeitverlust der Simulation im logarithmischen Kostenmaß.*
- *Zeige, dass der Befehlssatz der RAM für die Simulation auf die Befehle LOAD, CLOAD, STORE, CADD, CSUB, GOTO, IF* $c(0) \neq 0$ *GOTO, END eingeschränkt werden kann. Welchen Einfluss hat diese Einschränkung auf den Zeitverlust?*

Kapitel 2

Berechenbarkeit

In diesem Teil der Vorlesung werden wir untersuchen, welche Probleme durch einen Computer gelöst werden können. Wir werden Probleme kennenlernen, die beweisbar nicht durch einen Rechner gelöst werden können, selbst wenn beliebig viel Rechenzeit und Speicherplatz zur Verfügung steht.

2.1 Die Church-Turing-These

In Abschnitt 1.2.1, Definition 1.4 haben wir die Klasse der *TM-berechenbaren* bzw. *rekursiven* Funktionen definiert. Unsere bisherigen Ergebnisse zeigen, dass die Berechnungskraft einer RAM genauso groß ist wie die einer TM. Also stimmt die Klasse der TM-berechenbaren bzw. rekursiven Funktionen überein mit der Klasse derjenigen Funktionen, die durch eine RAM berechnet werden kann.

Tatsächlich wurden in den Anfängen der Rekursionstheorie (also der Theorie der Berechenbarkeit) zahlreiche Versuche unternommen, um die Klasse der berechenbaren Funktionen zu charakterisieren. Keines der in diesem Zusammenhang vorgeschlagenen Berechnungsmodelle hat sich als mächtiger als das TM-Modell herausgestellt. Diese Einsicht hat Church zu der Formulierung der folgenden These veranlasst. Die These trägt heute den Namen *Church-Turing-These*, da sie zwar von Church formuliert aber weitgehend auf Erkenntnisse von Turing zurückgeht.

These 2.1 (Church-Turing-These) *Die Klasse der rekursiven Funktionen stimmt mit der Klasse der intuitiv berechenbaren Funktionen überein.*

Die Church-Turing These kann grundsätzlich nicht bewiesen werden, da der Begriff der *intuitiven Berechenbarkeit* im Gegensatz zum Begriff der *Rekursivität*

nicht formal definiert ist. Die Klasse der intuitiv berechenbaren Funktionen interpretieren wir als die Menge aller Funktionen, die in realistischen Rechnermodellen berechnet werden können. Prinzipiell ist es vorstellbar, dass jemand ein realistisches Rechnermodell erdenkt, das es erlaubt eine größere Klasse von Funktionen zu berechnen als das TM- bzw. RAM-Modell. Die Church-Turing-These ist also zwar nicht beweisbar aber sie ist falsifizierbar. Kaum jemand glaubt jedoch, dass dies möglich ist. Deshalb ist die Church-Turing-These die Grundlage unserer Untersuchungen zur Berechenbarkeit.

Aufgrund der Church-Turing-These sprechen wir im Folgenden nicht mehr von TM-berechenbaren Funktionen, sondern nur noch von *berechenbaren* Funktionen. *TM-entscheidbare* Sprachen bezeichnen wir vereinfacht als *entscheidbare* Sprachen. Allgemein verwenden wir den Sammelbegriff der *rekursiven* Probleme.

Im Folgenden werden wir mehrere Probleme kennen lernen, die trotz ihrer einfachen Formulierung, nicht rekursiv sind. Das bedeutet, dass es für keines dieser Problem einen Algorithmus gibt, der dieses Problem löst, unabhängig vom Rechnermodell für den der Algorithmus spezifiziert wird. Dies schließt auch moderne Rechnerkonzepte wie z.B. Quantencomputer oder phantastische Konzepte wie nicht-deterministische Turingmaschinen, die wir später noch kennen lernen werden, ein.

2.2 Nicht rekursive Probleme

Wir kommen nun zum zentralen Punkt dieses Teils der Vorlesung: Wir beweisen die Existenz nicht rekursiver Probleme, zunächst durch ein einfaches Zählargument. Dann lernen wir einige konkrete Probleme kennen, die nicht rekursiv sind.

2.2.1 Existenz unentscheidbarer Probleme

Wir zeigen die Existenz unentscheidbarer Probleme, indem wir beweisen, dass die Menge aller Sprachen überabzählbar ist, während die Menge aller TMn abzählbar ist.

Definition 2.2 (Abzählbare Menge) *Eine Menge M heißt* abzählbar*, wenn es eine surjektive Funktion $c : \mathbb{N} \to M$ gibt. Nicht abzählbare Mengen heißen* überabzählbar.

Jede endliche Menge M ist offensichtlich abzählbar.

Im Fall einer abzählbar unendlichen Menge M gibt es immer auch eine bijektive Abbildung $c : \mathbb{N} \to M$, denn Wiederholungen können bei der Abzählung offensichtlich ausgelassen werden. Eine solche Abbildung liefert eine *Nummerierung* der Elemente von M. Abzählbar unendliche Mengen haben also dieselbe Mächtigkeit wie die Menge der natürlichen Zahlen $\mathbb{N}$.

Einige Beispiele für abzählbare Mengen sind:

- die Menge der ganzen Zahlen $\mathbb{Z}$:

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

- die Menge der Wörter über $\{0,1\}^*$:

$$\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, \ldots \quad .$$

- die Menge der TMn, weil jede TM durch eine Gödelnummer beschrieben wird, und die Menge der Gödelnummern eine Teilmenge der Wörter über $\{0,1\}^*$ ist.

Ein Beispiel für eine überabzählbare Menge ist die Potenzmenge der natürlichen Zahlen, d.h. die Menge aller Teilmengen von $\mathbb{N}$:

Satz 2.3 *Die Menge $\mathcal{P}(\mathbb{N})$ ist überabzählbar.*

Beweis: Zum Zweck des Widerspruchs nehmen wir an, dass $\mathcal{P}(\mathbb{N})$ abzählbar ist. Wir nehmen also an, dass es eine Nummerierung der Mengen in $\mathcal{P}(\mathbb{N})$ gibt. Mit S_i bezeichnen wir die i-te Menge gemäß dieser Nummerierung. Wir definieren eine zweidimensionale unendliche Matrix $(A_{i,j})_{i\in\mathbb{N},j\in\mathbb{N}}$ mit

$$A_{i,j} \;=\; \begin{cases} 1 & \text{falls } j \in S_i \\ 0 & \text{sonst} \end{cases}$$

Die Matrix A könnte etwa folgendermaßen aussehen:

	0	1	2	3	4	5	6	
S_0	**0**	1	1	0	1	0	1	$\cdots$
S_1	1	**1**	1	0	1	0	1	$\cdots$
S_2	0	0	**1**	0	1	0	1	$\cdots$
S_3	0	1	1	**0**	0	0	1	$\cdots$
S_4	0	1	0	0	**1**	0	1	$\cdots$
S_5	0	1	1	0	1	**0**	0	$\cdots$
S_6	1	1	1	0	1	0	**1**	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		

Wir definieren die Menge $S_{diag} = \{i \in \mathbb{N} \mid A_{i,i} = 1\}$. Das Komplement dieser Menge ist

$$\bar{S}_{diag} = \mathbb{N} \setminus S_{diag} = \{i \in \mathbb{N} \mid A_{i,i} = 0\} \ .$$

Auch $\bar{S}_{diag}$ ist eine der Mengen aus $\mathcal{P}(\mathbb{N})$. In der Nummerierung von $\mathcal{P}(\mathbb{N})$ nehme $\bar{S}_{diag}$ den k-ten Platz ein, d.h. $\bar{S}_{diag} = S_k$. Jetzt gibt es zwei Fälle, die jeweils zum Widerspruch führen.
Fall 1: Falls $A_{k,k} = 1$ gilt, so liegt k in S_k. Wegen $S_k = \bar{S}_{diag}$ liegt k aber nicht in S_{diag}. Es folgt $A_{k,k} = 0$, Widerspruch!
Fall 2: Falls $A_{k,k} = 0$ gilt, so liegt k nicht in S_k. Wegen $S_k = \bar{S}_{diag}$ liegt k aber in S_{diag}. Es folgt $A_{k,k} = 1$, auch Widerspruch! □

Wir beobachten nun, dass die Menge aller Sprachen über $\{0, 1\}$ bijektiv zur Potenzmenge von $\mathbb{N}$ ist. Dies liegt daran, dass die Menge $\{0, 1\}^*$ dieselbe Mächtigkeit wie $\mathbb{N}$hat, und dass jede Sprache einer Teilmenge von $\{0, 1\}^*$ entspricht.

Korollar 2.4 *Die Menge der Sprachen über dem Alphabet $\{0, 1\}$ ist überabzählbar.*

Wir fassen zusammen: Die Menge der TMn ist abzählbar. Die Menge der Sprachen ist überabzählbar. Also gibt es mehr Sprachen als TMn, und somit gibt es Sprachen, die unentscheidbar sind.

2.2.2 Unentscheidbarkeit der Diagonalsprache

Wir wissen jetzt, dass es nicht rekursive Probleme gibt, kennen aber noch kein konkretes nicht rekursives Problem. Die *Diagonalsprache* D ist ein erstes Beispiel für ein derartiges Problem. Diese Sprache ist folgendermaßen definiert.

$$D \ = \ \{\, w \in \{0, 1\}^* \mid w = w_i \text{ und } M_i \text{ akzeptiert } w \text{ nicht}\} \ .$$

Anders gesagt, das i-te Wort bzgl. der kanonischen Reihenfolge, also w_i, ist genau dann in D, wenn die i-te TM, also M_i, dieses Wort nicht akzeptiert.

Zunächst versuchen wir ein wenig Intuition für diese selsame Sprache zu gewinnen. Warum heißt diese Sprache *Diagonalsprache*? Man betrachte eine unendliche binäre Matrix A mit

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

Die Matrix könnte beispielsweise so aussehen:

	w_0	w_1	w_2	w_3	w_4	
M_0	$\mathbf{0}$	1	1	0	1	$\cdots$
M_1	1	$\mathbf{0}$	1	0	1	$\cdots$
M_2	0	0	$\mathbf{1}$	0	1	$\cdots$
M_3	0	1	1	$\mathbf{1}$	0	$\cdots$
M_4	0	1	0	0	$\mathbf{0}$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		

Die Diagonalsprache läßt sich auf der Diagonalen der Matrix ablesen. Es ist

$$D = \{w_i \mid A_{i,i} = 0\} \ .$$

Satz 2.5 *Die Diagonalsprache D ist nicht rekursiv.*

Beweis: Wir führen einen Widerspruchsbeweis und nehmen an, D ist rekursiv. Dann gibt es eine TM M_j, die D entscheidet. Wir wenden M_j auf w_j an. Es ergeben sich zwei Fälle, die jeweils direkt zum Widerspruch führen.
Fall 1: Falls w_j in D liegt, dann akzeptiert M_j die Eingabe w_j, weil M_j die Sprache D entscheidet. Wegen der Definition von D kann w_j somit aber nicht in D liegen, Widerspruch!
Fall 2: Falls w_j nicht in D liegt, dann verwirft M_j die Eingabe w_j, weil M_j die Sprache D entscheidet. Wegen der Definition von D muss w_j somit aber in D liegen, auch Widerspruch! □

Auch das Komplement der Diagonalsprache ist nicht berechenbar. Dies folgt aus dem folgenden Hilfssatz.

Lemma 2.6 *Sei L eine unentscheidbare Sprache. Dann ist auch $\bar{L}$, das Komplement von L, unentscheidbar.*

Beweis: Zum Widerspruch nehmen wir an, es gibt eine TM $M_{\bar{L}}$, die die Sprache $\bar{L}$ entscheidet. Gemäß Definition 1.5 hält $M_{\bar{L}}$ auf jeder Eingabe w und akzeptiert genau dann, wenn $w \in \bar{L}$. Wir konstruieren nun eine TM M, die $M_{\bar{L}}$ als Unterprogramm verwendet: M startet $M_{\bar{L}}$ auf der vorliegenden Eingabe und negiert anschließend die Ausgabe von $M_{\bar{L}}$. Die TM M entscheidet nun offensichtlich L. Ein Widerspruch zur Unentscheidbarkeit von L. □

Korollar 2.7 *Das Komplement $\bar{D}$ der Diagonalsprache ist nicht rekursiv.*

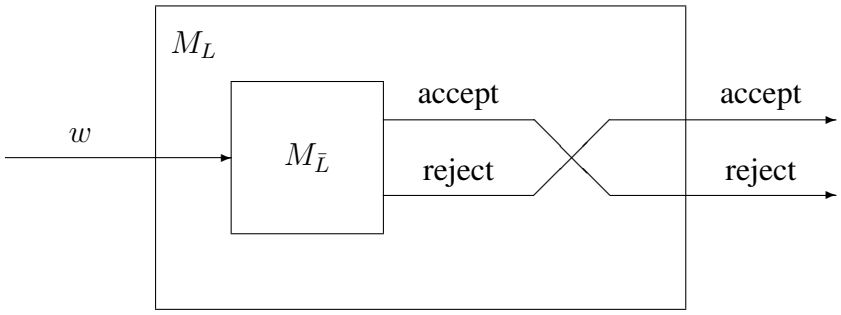

Abbildung 2.1: Aus $M_{\bar{D}}$ konstruieren wir M_D.

2.2.3 Unentscheidbarkeit des Halteproblems

Die Diagonalsprache scheint doch recht konstruiert und wenig praxisrelevant zu sein. Man könnte die Hoffnung haben, dass alle unentscheidbaren Probleme ohne Bedeutung für die Praxis sind. Leider werden wir sehen, dass diese Hoffnung sich nicht bestätigt.

Beim *Halteproblem* geht es darum, zu entscheiden, ob ein Programm auf einer bestimmten Eingabe terminiert. In unserer Notation ergibt sich die folgende formale Problemdefinition.

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

Es wäre äußerst hilfreich, wenn Compiler das Halteproblem entscheiden könnten. Leider zeigt der folgende Satz, das dies nicht möglich ist.

Satz 2.8 *Das Halteproblem H ist nicht rekursiv.*

Beweis: Zum Zweck des Widerspruchs nehmen wir an, dass H entscheidbar ist. Dann gibt es eine TM M_H, die H entscheidet. Wir zeigen nun, wie man mit Hilfe von M_H eine TM $M_{\bar{D}}$ konstruiert, die $\bar{D}$ entscheidet. Dieses Problem ist jedoch nach Korollar 2.7 unentscheidbar. Also kann es M_H nicht geben, und somit ist der Satz gezeigt.

Das Programm der TM $M_{\bar{D}}$ mit Unterprogramm M_H sieht folgendermaßen aus:

1. Auf Eingabe w, berechne i, so dass gilt $w = w_i$.
2. Berechne nun die Gödelnummer der i-ten TM, also $\langle M_i \rangle$.

3. Starte M_H als Unterprogramm mit Eingabe $\langle M_i \rangle w$.

4. Falls M_H akzeptiert, so simuliere das Verhalten von M_i auf w (genau wie die universelle TM dies tun würde) und übernehme die Ausgabe.

5. Falls M_H verwirft, so verwirf die Eingabe.

Wir müssen die Korrektheit der Konstruktion nachweisen. Sei $w = w_i$. Es gilt

$$\begin{aligned} w \in \bar{D} &\Rightarrow M_H \text{ akzeptiert } \langle M_i \rangle w \text{ und } M_i \text{ akzeptiert } w \\ &\Rightarrow M_{\bar{D}} \text{ akzeptiert } w \end{aligned}$$

$$\begin{aligned} w \notin \bar{D} &\Rightarrow M_H \text{ verwirft } \langle M_i \rangle w \text{ oder } M_i \text{ verwirft } w \\ &\Rightarrow M_{\bar{D}} \text{ verwirft } w \end{aligned}$$

□

2.2.4 Die Unterprogrammtechnik (Turing Reduktion)

Die Beweise von Lemma 2.6 und Satz 2.8 basieren auf dem gleichen Beweisansatz: Um nachzuweisen, dass eine bestimmte Sprache, nennen wir sie X, nicht rekursiv ist, haben wir zum Zwecke des Widerspruchs angenommen, dass eine TM M_X existiert, die X entscheidet. Wir haben dann eine TM M_Y konstruiert, die M_X als Unterprogramm aufruft und auf diese Art eine Sprache Y entscheidet, von der wir bereits wissen, dass sie nicht rekursiv ist. Aus dieser Konstruktion haben wir gefolgert, dass M_X nicht existieren kann und somit X unentscheidbar ist. Dieses Beweiskonzept bezeichnen wir als *Unterprogrammtechnik*.[1]

Wir demonstrieren die Unterprogrammtechnik an einem weiteren Beispiel. Das *spezielle Halteproblem* ist definiert durch

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

Satz 2.9 *Das spezielle Halteproblem H_ϵ ist nicht rekursiv.*

[1] In der Literatur wird die Unterprogrammtechnik häufig auch als *Turing Reduktion* bezeichnet. Wir behalten uns den Begriff Reduktion jedoch für eine spezielle, restriktivere Art der Unterprogrammtechnik vor, die wir später in Abschnitt 2.5 einführen.

Beweis: Wir nutzen die Unterprogrammtechnik. Zum Zwecke des Widerspruchs nehmen wir an es gibt eine Orakelmaschine M_ϵ, die H_ϵ entscheidet. Mit der Orakelmaschine als Unterprogramm konstruieren wir eine TM M_H, die das nicht rekursive Halteproblem entscheidet, und deshalb nicht existieren kann. Also ist H_ϵ nicht entscheidbar.

Die TM M_H mit Unterprogramm M_ϵ arbeitet wie folgt:

1. Falls die Eingabe nicht mit einer korrekten Gödelnummer beginnt, verwirft M_H die Eingabe.

2. Sonst, also auf Eingaben der Form $\langle M \rangle w$, berechnet M_H die Gödelnummer einer TM M_w^* mit den folgenden Eigenschaften:

   - Falls die TM M_w^* auf der leeren Eingabe startet, schreibt sie das Wort w aufs Band und simuliert die TM M auf der Eingabe w.
   - Auf anderen Eingaben kann sich die TM beliebig verhalten.

3. Nachdem M_H die Gödelnummer $\langle M_w^* \rangle$ auf das Band geschrieben hat, startet sie M_ϵ auf dieser Eingabe, und akzeptiert genau dann, wenn M_ϵ akzeptiert.

Unsere Konstruktion ist korrekt, denn es gilt:

$$\begin{aligned} \langle M \rangle w \in H &\Rightarrow M \text{ hält auf } w \\ &\Rightarrow M_w^* \text{ hält auf } \epsilon \\ &\Rightarrow M_\epsilon \text{ akzeptiert } \langle M_w^* \rangle \\ &\Rightarrow M_H \text{ akzeptiert } \langle M \rangle w \end{aligned}$$

$$\begin{aligned} \langle M \rangle w \notin H &\Rightarrow M \text{ hält nicht auf } w \\ &\Rightarrow M_w^* \text{ hält nicht auf } \epsilon \\ &\Rightarrow M_\epsilon \text{ verwirft } \langle M_w^* \rangle \\ &\Rightarrow M_H \text{ verwirft } \langle M \rangle w \end{aligned}$$

□

2.2.5 Der Satz von Rice

Da TMn nicht auf jeder Eingabe halten, berechnen sie partielle Funktionen. Die von einer TM M berechnete Funktion ist deshalb von der Form

$$f_M : \{0,1\}^* \to \{0,1\}^* \cup \{\perp\} \ .$$

Das Zeichen $\perp$ steht dabei für *nicht definiert* und bedeutet, dass die Maschine nicht hält. Im Fall von Entscheidungsproblemen vereinfacht sich die Form der Funktionen zu

$$f_M : \{0,1\}^* \to \{0,1,\perp\} \ .$$

Dabei steht 0 für *Verwerfen*, 1 für *Akzeptieren* und $\perp$ für *Nicht-Halten*. Sei

$$\mathcal{R} \ = \ \{f_M : \{0,1\}^* \to \{0,1\}^* \cup \{\perp\} \mid \text{M ist eine TM}\} \ .$$

$\mathcal{R}$ bezeichnet also die Menge der von TM berechenbaren Funktionen.

Satz 2.10 (Satz von Rice) *Sei S eine Teilmenge von $\mathcal{R}$ mit $\emptyset \neq S \neq \mathcal{R}$. Dann ist die Sprache*

$$L(S) = \{\, \langle M \rangle \mid M \textit{ berechnet eine Funktion aus } S\}$$

nicht rekursiv.

In Worten: Nicht-triviale Eigenschaften der von einer TM berechneten Funktion sind nicht entscheidbar.

Beweis: Wir benutzen die Unterprogrammtechnik. Aus einer TM $M_{L(S)}$, die $L(S)$ entscheidet, konstruieren wir eine TM M_ϵ, die das spezielle Halteproblem H_ϵ entscheidet, und damit im Widerspruch zu Satz 2.9 steht.

Sei u die überall undefinierte Funktion. Wir unterscheiden zwei Fälle. Im ersten Fall nehmen wir an, dass $u \notin S$ gilt. Sei $f \neq u$ eine Funktion aus S. Die Funktion f existiert aufgrund unserer Voraussetzung $S \neq \emptyset$.

Die TM M_ϵ mit Unterprogramm $M_{L(S)}$ arbeitet wie folgt:

1. Falls die Eingabe nicht aus einer korrekten Gödelnummer besteht, verwirft M_ϵ die Eingabe.

2. Sonst konstruiert M_ϵ aus der Eingabe $\langle M \rangle$ die Gödelnummer einer TM M^* mit folgendem Verhalten:

(a) Ignoriere die Eingabe x. Simuliere das Verhalten von M bei Eingabe ϵ auf einer für diesen Zweck reservierten Spur.

(b) Berechne $f(x)$, d.h. simuliere M_f auf Eingabe x.

Beobachtung: Die konstruierte TM M^* berechnet genau dann die Funktion $f(x)$, wenn die gegebene TM M auf ϵ hält.

3. Nachdem M_ϵ die Gödelnummer $\langle M^* \rangle$ auf das Band geschrieben hat, startet sie $M_{L(S)}$ auf dieser Eingabe, und akzeptiert genau dann, wenn $M_{L(S)}$ akzeptiert.

Wir zeigen nun die Korrektheit unserer Konstruktion. Falls die Eingabe keine Gödelnummer ist, so verwirft M_ϵ korrekterweise die Eingabe. Bei Eingaben der Form $w = \langle M \rangle$ gilt:

$$\begin{aligned} w \in H_\epsilon &\Rightarrow M \text{ hält auf } \epsilon \\ &\Rightarrow M^* \text{ berechnet } f \\ &\Rightarrow \langle M^* \rangle \in L(S) \\ &\Rightarrow M_{L(S)} \text{ akzeptiert } \langle M^* \rangle \\ &\Rightarrow M_\epsilon \text{ akzeptiert } w \end{aligned}$$

$$\begin{aligned} w \notin H_\epsilon &\Rightarrow M \text{ hält nicht auf } \epsilon \\ &\Rightarrow M^* \text{ berechnet } u \\ &\Rightarrow \langle M^* \rangle \notin L(S) \\ &\Rightarrow M_{L(S)} \text{ verwirft } \langle M^* \rangle \\ &\Rightarrow M_\epsilon \text{ verwirft } w \end{aligned}$$

Damit ist die Korrektheit nachgewiesen.

Im zweiten Fall, den wir noch untersuchen müssen, gilt $u \in S$. Jetzt wählen wir f aus $\mathcal{R} \setminus S \neq \emptyset$. Der Rest des Beweis für diesen Fall ist fast analog zu Fall 1. Wir invertieren lediglich das Akzeptanzverhalten der TM M_ϵ in Schritt 3. $\square$

Fazit: Der Satz von Rice hat viele Konsequenzen für die Praxis. Zum Beispiel kann es keinen Compiler geben, der entscheidet, ob ein gegebenes TM- oder RAM-Programm auf allen Eingaben terminiert, weil der Compiler dazu entscheiden müßte, ob das Programm eine Funktion der Form $f : \Sigma^* \to \Sigma^*$ berechnet. Insbesondere kann es keine automatische Verifikation von TM- oder RAM-

Programmen geben, weil dieses einen Algorithmus erfordern würde, der entscheidet, ob ein gegebenes Programm eine Funktion berechnet, die einer gegebenen Spezifikation S entspricht.

2.3 Semi-Entscheidbarkeit und Rekursive Aufzählbarkeit

Definition 2.11 *Eine Sprache L wird von einer TM M* entschieden*, wenn M auf jeder Eingabe hält, und genau die Wörter aus L akzeptiert.*

Diese Definition hängt eng zusammen mit der Definition der Entscheidbarkeit (vgl. Definition 1.5): Eine Sprache ist *entscheidbar* bzw. *rekursiv* genau dann, wenn es eine TM gibt, die die Sprache entscheidet.

Definition 2.12 *Eine Sprache $L \subseteq \Sigma^*$ wird von einer TM M* erkannt*, wenn M jedes Wort aus L akzeptiert und M kein Wort akzeptiert, das nicht in L enthalten ist. Auf Eingaben, die nicht in L enthalten sind, muss M nicht halten.*

Definition 2.13 *Eine Sprache L heißt* semi-entscheidbar*, wenn es eine TM gibt, die L erkennt.*

Beispielsweise ist das Halteproblem zwar nicht entscheidbar aber semi-entscheidbar. Eine TM kann die Sprache H wie folgt erkennen: Wenn die Eingabe x nicht die Form $\langle M \rangle w$ für eine TM M hat, so wird verworfen. Ansonsten wird M auf Eingabe w simuliert, und es wird akzeptiert, wenn M auf w hält. Wenn M nicht auf w hält, terminiert die Simulation nicht, aber genau dies ist in der Definition der Semi-Entscheidbarkeit erlaubt.

Ein *Aufzähler* für eine Sprache $L \subseteq \Sigma^*$ ist eine Variante einer TM mit einem angeschlossenen *Drucker* im Sinne eines zusätzlichen Ausgabebandes, auf dem sich der Kopf nur nach rechts bewegt. Gestartet mit leerem Speicherband, gibt der Aufzähler alle Wörter aus L auf dem Drucker aus, d.h.

- der Aufzähler gibt ausschließlich Wörter aus L aus, und
- jedes Wort aus L wird irgendwann ausgegeben.

Die ausgegebenen Wörter sind dabei durch ein Zeichen getrennt, das nicht in Σ enthalten ist. Die wiederholte Ausgabe von Wörtern aus L ist nicht explizit verboten. Abbildung 2.2 visualisiert einen Aufzähler.

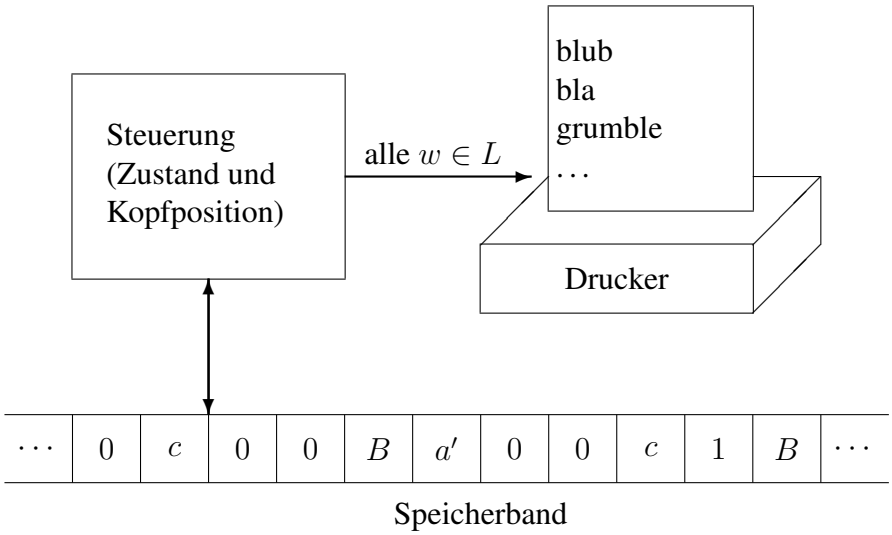

Abbildung 2.2: Illustration eines Aufzählers für eine Sprache L

Definition 2.14 *Eine Sprache L heißt* rekursiv aufzählbar*, wenn es einen Aufzähler für L gibt.*

Wir zeigen nun, dass Semi-Entscheidbarkeit und Rekursive Aufzählbarkeit gleichbedeutend sind.

Satz 2.15 *Eine Sprache L ist genau dann semi-entscheidbar, wenn L rekursiv aufzählbar ist.*

Beweis: Als erstes zeigen wir, wie man aus einem Aufzähler für L eine TM M konstruieren kann, die L erkennt. Die TM M simuliert den Aufzähler mit Hilfe einer Spur, die die Rolle des Druckers übernimmt. Immer wenn ein neues Wort ausgegeben worden ist, vergleicht M dieses Wort mit w und akzeptiert bei Übereinstimmung. Falls $w \in L$, so wird w irgendwann durch den Aufzähler ausgegeben und somit von M akzeptiert. Falls $w \notin L$, so wird w nicht akzeptiert. Also erkennt M die Sprache L.

Zum Beweis der anderen Richtung zeigen wir nun, wie man aus einer TM M, die L erkennt, einen Aufzähler konstruiert. Seien $w_1, w_2, w_3, \ldots$ die Wörter aus Σ^* in kanonischer Reihenfolge. Der Aufzähler arbeitet wie folgt: Für $i = 1, 2, 3, \ldots$

- Simuliere i Schritte von M auf jedem Wort aus $w_1, \ldots, w_i$.

- Wird dabei eines der Worte akzeptiert, so drucke es aus.

Falls das Wort w_k in L enthalten ist, so wird es von M nach einer endlichen Anzahl von Schritten, sagen wir nach t_k vielen Schritten, akzeptiert. In jeder Iteration $i \geq \max\{k, t_k\}$ druckt der Aufzähler dieses Wort somit aus. Ein Wort $w \notin L$ wird hingegen nicht von M akzeptiert und somit auch nicht vom Aufzähler ausgedruckt. □

2.4 Eigenschaften rekursiver und rekursiv aufzählbarer Sprachen

Wir untersuchen nun, ob die rekursiven oder rekursiv aufzählbaren Sprachen gegenüber den üblichen Mengenoperationen wie Schnitt, Vereinigung und Komplement abgeschlossen sind. Zunächst zeigen wir, dass sowohl die Menge der rekursiven als auch die Menge der rekursiv aufzählbaren Sprachen gegen Schnittmengenbildung abgeschlossen sind.

Satz 2.16

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

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

Beweis: a) Seien M_1 und M_2 zwei TMn, die L_1 bzw. L_2 entscheiden. Wir konstruieren eine TM M, die $L_1 \cap L_2$ entscheidet. Die TM M arbeitet so:

- Auf Eingabe w simuliert M zunächst das Verhalten von M_1 auf w und dann das Verhalten von M_2 auf w.

- Falls M_1 und M_2 akzeptieren, so akzeptiert auch M.

Die TM M akzeptiert offensichtlich die Eingaben aus $L_1 \cap L_2$. Zudem hält M auch auf jeder Eingabe, da sowohl M_1 als auch M_2 auf jeder Eingabe halten. Also entscheidet M die Sprache $L_1 \cap L_2$.

b) Wir verwenden dieselbe Konstruktion wie in (a). Alle Aussagen, bis auf die zur Terminierung, gelten wie zuvor, wenn wir jedes Vorkommen des Begriffes „entscheiden“ durch den Begriff „erkennen“ ersetzen. □

Ein analoger Satz gilt für die Vereinigung von rekursiven bzw. rekursiv aufzählbaren Sprachen:

Satz 2.17

a) Wenn die Sprachen L_1 und L_2 rekursiv sind, so ist auch die Sprache $L_1 \cup L_2$ rekursiv.

b) Wenn die Sprachen L_1 und L_2 rekursiv aufzählbar sind, so ist auch die Sprache $L_1 \cup L_2$ rekursiv aufzählbar.

Beweis: a) Seien M_1 und M_2 zwei TMn, die L_1 bzw. L_2 entscheiden. Wir konstruieren eine TM M, die $L_1 \cup L_2$ entscheidet. Die TM M arbeitet so:

- Auf Eingabe w, simuliert M zunächst das Verhalten von M_1 auf w und dann das Verhalten von M_2 auf w.
- Falls M_1 oder M_2 akzeptieren, so akzeptiert auch M.

Die TM M akzeptiert offensichtlich die Eingaben aus $L_1 \cup L_2$. Zudem hält M auf jeder Eingabe, da sowohl M_1 als auch M_2 auf jeder Eingabe halten. Also entscheidet M die Sprache $L_1 \cup L_2$.

b) Wir können diesmal die Konstruktion aus (a) nicht ohne Weiteres auf Maschinen M_1 und M_2 übertragen, die die Sprachen L_1 und L_2 nur erkennen, aber nicht entscheiden. Das Problem ist, dass möglicherweise M_2 die Eingabe akzeptiert, aber M das niemals feststellt, da M_1 nicht terminiert.

Um dieses Problem zu umgehen, verwenden wir statt der „sequentiellen“ Simulation von M_1 und M_2 eine „parallele“ Simulation der beiden TMn durch eine 2-Band TM M, die jeweils ein Band für die Simulation von M_1 und M_2 einsetzt.

Wir nehmen an, dass M_1 und M_2 die Zustandsmengen Q_1 bzw. Q_2 verwenden. Die Zustandsmenge von M enthält die Tupel aus $Q_1 \times Q_2$.

Seien δ_1 und δ_2 die Übergangsfunktionen von M_1 und M_2. Dann ist die Übergangsfunktion δ von M folgendermaßen definiert. Für je zwei Zustände $q_1 \in Q_1$ und $q_2 \in Q_2$, die keine Endzustände seien, und zwei Zeichen $a_1, a_2 \in \Gamma$ des Bandalphabets mit $\delta_1(q_1, a_1) = (q_1', b_1, k_1)$ für $q_1' \in Q_1, b_1 \in \Gamma, k_1 \in \{L, R, N\}$ und $\delta_2(q_2, a_2) = (q_2', b_2, k_2)$ für $q_2' \in Q_2, b_2 \in \Gamma, k_2 \in \{L, R, N\}$ setzen wir

$$\delta((q_1, q_2)(a_1, a_2)) := ((q_1', q_2'), (b_1, b_2), (k_1, k_2)) \ .$$

Das Akzeptanzverhalten definieren wir wie folgt: M akzeptiert, sobald die Rechnung von M_1 auf Band 1 oder die Rechnung von M_2 auf Band 2 akzeptiert. Diese Simulation stellt sicher, dass M akzeptiert, wenn M_1 akzeptiert *oder* M_2 akzeptiert. M erkennt somit die Sprache $L_1 \cup L_2$. □

Bei der Komplementbildung unterscheiden sich rekursive und rekursiv aufzählbare Sprachen aber voneinander:

Satz 2.18 *Wenn eine Sprache L rekursiv ist, so ist auch $\bar{L}$ rekursiv.*

Beweis: Sei M_L eine TM, die L entscheidet. Wir erhalten eine TM $M_{\bar{L}}$, die $\bar{L}$ entscheidet, indem wir das Akzeptanzverhalten von M_L invertieren. Also ist $\bar{L}$ rekursiv. □

Bemerkung 2.19 *Im Gegensatz zur Menge der rekursiven Sprachen ist die Menge der rekursiv aufzählbaren Sprachen nicht gegen Komplementbildung abgeschlossen.*

Beweis: Das Halteproblem belegt die Aussage. Die Sprache H ist rekursiv aufzählbar (siehe oben). Wäre $\bar{H}$ ebenfalls rekursiv aufzählbar, so wäre H nach Lemma 2.20 (siehe unten) rekursiv. Ein Widerspruch zu Satz 2.8. Also ist $\bar{H}$ nicht rekursiv aufzählbar. □

Lemma 2.20 *Sind $L \subseteq \Sigma^*$ und $\bar{L} = \Sigma^* \setminus L$ rekursiv aufzählbar, so ist L rekursiv.*

Beweis: Seien M und $\bar{M}$ Maschinen, die L bzw. $\bar{L}$ erkennen. Die TM M' entscheidet L durch eine parallele Simulation von M und $\bar{M}$ auf der Eingabe w:

- M' akzeptiert w, sobald M akzeptiert.
- M' verwirft w, sobald $\bar{M}$ akzeptiert.

Da entweder $w \in L$ oder $w \notin L$ gilt, tritt eines dieser Ereignisse nach endlicher Zeit ein, so dass die Terminierung von M' sichergestellt ist. □

Aus diesem Lemma folgt:

Korollar 2.21 *L ist nicht rekursiv genau dann, wenn mindestens eine der beiden Sprachen L oder $\bar{L}$ nicht rekursiv aufzählbar ist.*

2.5 Die Technik der Reduktion

Die Reduktion ist eine Technik zum Nachweis, ob eine Sprache rekursiv bzw. rekursiv aufzählbar oder auch nicht rekursiv bzw. nicht rekursiv aufzählbar ist. Die von uns verwendete Art der Reduktion bildet die Eingaben eines Problems auf die Eingaben eines anderen Problems ab, und wird deshalb auch als *Eingabe-Eingabe-Reduktion* oder *Many-One Reduktion* bezeichnet.

Definition 2.22 *Es seien L_1 und L_2 Sprachen über Σ. Dann heißt L_1 auf L_2* reduzierbar*, Notation $L_1 \leq L_2$, wenn es eine berechenbare Funktion $f : \Sigma^* \to \Sigma^*$ gibt, so dass für alle $x \in \Sigma^*$ gilt*

$$x \in L_1 \;\Leftrightarrow\; f(x) \in L_2 \;\;.$$

Wie die Notation bereits andeutet, bedeutet $L_1 \leq L_2$, dass L_1 unter dem Gesichtspunkt der Berechenbarkeit nicht schwieriger als L_2 ist. Dies wird formalisiert durch das folgende Lemma:

Lemma 2.23 *Falls $L_1 \leq L_2$ und L_2 rekursiv [rekursiv aufzählbar] ist, so ist auch L_1 rekursiv [rekursiv aufzählbar].*

Beweis: Wir konstruieren eine TM M_1, die L_1 entscheidet [erkennt], durch Unterprogrammaufruf einer TM M_2, die L_2 entscheidet [erkennt]:

- Die TM M_1 berechnet $f(x)$ aus ihrer Eingabe x.
- Dann simuliert M_1 die TM M_2 mit der Eingabe $f(x)$ und übernimmt das Akzeptanzverhalten.

Die TM M_1 arbeitet korrekt, denn

$$\begin{aligned} M_1 \text{ akzeptiert } x &\;\Leftrightarrow\; M_2 \text{ akzeptiert } f(x) \\ &\;\Leftrightarrow\; f(x) \in L_2 \\ &\;\Leftrightarrow\; x \in L_1 \;\;. \end{aligned}$$

Falls L_2 rekursiv ist, so ist die Terminierung von M_1 auf jeder Eingabe gesichert. Falls L_2 rekursiv aufzählbar ist, so ist die Terminierung von M_1 auf Eingaben aus L_1 gesichert. □

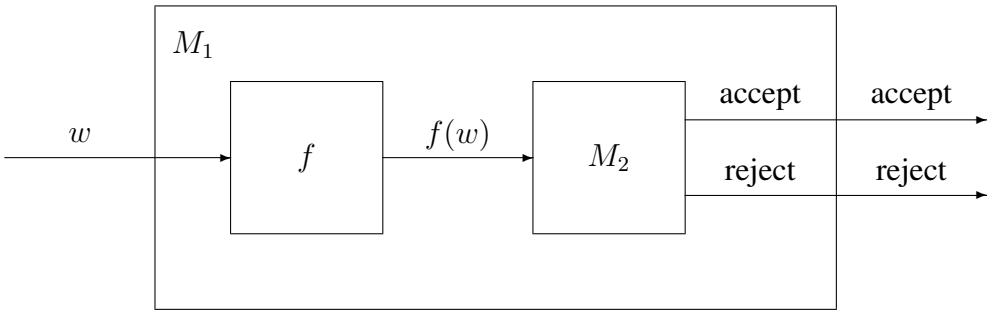

Abbildung 2.3: Im Beweis von Lemma 2.23 konstruieren wir M_1 aus M_2.

Der Beweis von Lemma 2.23 wird in Abbildung 2.3 visualisiert. Diese Abbildung macht auch deutlich, dass das Akzeptanzverhalten des Unterprogramms unverändert übernommen wird. Man kann erkennen, dass die Reduktion eigentlich nur eine spezielle Variante der Unterprogrammtechnik ist, bei der nach dem Unterprogrammaufruf keine Manipulation der Ausgabe mehr möglich ist. Insbesondere ist es nicht möglich nach dem Unterprogrammaufruf das Akzeptanzverhalten zu invertieren, also Ja- und Nein-Antworten zu vertauschen. Dies scheint aber beim Unterprogrammaufruf einer rekursiv aufzählbaren Sprache ohnehin wenig sinnvoll zu sein, da das Unterprogramm für Eingaben, die nicht zur Sprache gehören, möglicherweise nicht terminiert, so dass nach dem Invertieren der Ausgabe, nicht notwendigerweise alle Eingaben, die zur Sprache gehören, akzeptiert werden.

Wir demonstrieren das Konzept der Reduktion anhand der Sprache H_{all}. Diese Sprache ist definiert durch

$$H_{\text{all}} \;=\; \{\langle M\rangle \mid M \text{ hält auf allen Eingaben}\} \;.$$

Die Sprache H_{all} ist als *allgemeines Halteproblem* bekannt. Sie ist ein Beispiel für eine Sprache, die nicht rekursiv aufzählbar ist und deren Komplement ebenfalls nicht rekursiv aufzählbar ist. Wir werden diese Eigenschaft mit Hilfe von zwei Reduktionen in Verbindung mit Lemma 2.23 nachweisen. Man beachte, dass alle Sprachen, die wir zuvor untersucht haben, die Eigenschaft haben, dass entweder die Sprache selbst rekursiv aufzählbar ist, oder aber ihr Komplement rekursiv aufzählbar ist.

Satz 2.24 *Die Sprache $\bar{H}_{all}$ ist nicht rekursiv aufzählbar.*

Beweis: Aufgrund von Lemma 2.23 reicht es, eine nicht rekursiv aufzählbare Sprache auf $\bar{H}_{\text{all}}$ zu reduzieren. Zur Erinnerung: Das spezielle Halteproblem ist

definiert durch (vergleiche Abschnitt 2.2.3)

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

Die Sprache H_ϵ ist nicht rekursiv (Satz 2.9), aber offensichtlich rekursiv aufzählbar. Mit Lemma 2.20 ergibt sich, dass $\bar{H}_\epsilon$ nicht rekursiv aufzählbar ist. Daher zeigen wir $\bar{H}_\epsilon \leq \bar{H}_{\text{all}}$ bzw. die äquivalente Aussage $H_\epsilon \leq H_{\text{all}}$.

Gemäß der Definition von Reduktionen konstruieren wir eine berechenbare Funktion f, die Ja-Instanzen von H_ϵ auf Ja-Instanzen von H_{all} und Nein-Instanzen von H_ϵ auf Nein-Instanzen von H_{all} abbildet. Die Funktion f ist folgendermaßen definiert. Sei w die Eingabe für H_ϵ.

- Wenn w keine gültige Gödelnummer ist, so sei $f(w) = w$.
- Falls $w = \langle M\rangle$ für eine TM M, so sei $f(w)$ die Gödelnummer einer TM M^*_ϵ mit der folgenden Eigenschaft: M^*_ϵ ignoriert die Eingabe und simuliert M mit der Eingabe ϵ.

Die Funktion f ist offensichtlich berechenbar.

Es bleibt zu zeigen, dass f Gödelnummern korrekt transformiert. Falls w keine Gödelnummer ist, so gilt $w \notin H_\epsilon$ und $f(w) = w \notin H_{\text{all}}$, wie erforderlich. Sei nun $w = \langle M\rangle$ für eine TM M. In diesem Fall gilt $f(w) = \langle M^*_\epsilon\rangle$. Es folgt

$$\begin{aligned} w \in H_\epsilon \;&\Rightarrow\; M \text{ hält auf der Eingabe } \epsilon \\ &\Rightarrow\; M^*_\epsilon \text{ hält auf jeder Eingabe} \\ &\Rightarrow\; f(w) = \langle M^*_\epsilon\rangle \in H_{\text{all}} \;\;, \end{aligned}$$

$$\begin{aligned} w \notin H_\epsilon \;&\Rightarrow\; M \text{ hält nicht auf Eingabe } \epsilon \\ &\Rightarrow\; M^*_\epsilon \text{ hält auf keiner Eingabe} \\ &\Rightarrow\; f(w) = \langle M^*_\epsilon\rangle \notin H_{\text{all}} \;\;. \end{aligned}$$

Also gilt $w \in H_\epsilon \;\Leftrightarrow\; f(w) \in H_{\text{all}}$ und somit ist die Funktion f korrekt konstruiert. □

Satz 2.25 *Die Sprache H_{all} ist nicht rekursiv aufzählbar.*

Beweis: Analog zum vorherigen Satz zeigen wir $\bar{H}_\epsilon \leq H_{\text{all}}$.

Wir konstruieren eine berechenbare Funktion f, die Ja-Instanzen von $\bar{H}_\epsilon$ auf Ja-Instanzen von H_{all} und Nein-Instanzen von $\bar{H}_\epsilon$ auf Nein-Instanzen von H_{all} abbildet. Sei w die Eingabe für $\bar{H}_\epsilon$.

- Wenn w keine gültige Gödelnummer ist, liegt w in $\bar{H}_\epsilon$. Daher bildet f ungültige Gödelnummern auf irgendein festgelegtes $w' \in H_{\text{all}}$ ab.
- Falls $w = \langle M \rangle$ für eine TM M gilt, so berechnet f die Gödelnummer $\langle M'_M \rangle$. Die TM M'_M verhält sich auf einer Eingabe x wie folgt: Falls $|x| = i$, so simuliert M'_M die ersten i Schritte von M auf Eingabe ϵ. Wenn M nach i Schritten hält, dann geht M'_M in eine Endlosschleife, ansonsten hält M'_M.

Die Funktion f ist offensichtlich berechenbar.

Es bleibt zu zeigen, dass f Gödelnummern korrekt transformiert. Wenn w keine gültige Gödelnummer ist die Korrektheit offensichtlich. Im Falle $w = \langle M \rangle$ gilt

$$\begin{aligned} w \in \bar{H}_\epsilon &\Rightarrow M \text{ hält nicht auf der Eingabe } \epsilon \\ &\Rightarrow \neg\exists i\text{: } M \text{ hält innerhalb von } i \text{ Schritten auf } \epsilon \\ &\Rightarrow \forall i\text{: } M'_M \text{ hält auf allen Eingaben der Länge } i \\ &\Rightarrow f(w) = \langle M'_M \rangle \in H_{\text{all}} \ , \end{aligned}$$

$$\begin{aligned} w \notin \bar{H}_\epsilon &\Rightarrow M \text{ hält auf der Eingabe } \epsilon \\ &\Rightarrow \exists i\text{: } M \text{ hält innerhalb von } i \text{ Schritten auf } \epsilon \\ &\Rightarrow \exists i\text{: } M'_M \text{ hält nicht auf Eingaben der Länge } i \\ &\Rightarrow f(w) = \langle M'_M \rangle \notin H_{\text{all}} \ . \end{aligned}$$

Also gilt $w \in \bar{H}_\epsilon \Leftrightarrow f(w) \in H_{\text{all}}$ und somit ist die Funktion f korrekt konstruiert. □

Übungsaufgabe 2.26 *Zeige die Transitivität der Reduktion, d.h. aus $L_1 \leq L_2$ und $L_2 \leq L_3$ folgt $L_1 \leq L_3$ für beliebige Sprachen L_1, L_2, L_3.*

2.6 Klassische Probleme aus der Rekursionstheorie

Bei allen nicht rekursiven Problemen, die wir bisher kennengelernt haben, ging es darum, Aussagen über ein gegebenes Programm (in Form einer Gödelnummer) zu machen. Unsere Analyse gipfelte im Satz von Rice, der besagt, dass jede nicht-triviale Aussage über die durch ein Programm berechnete Funktion unentscheidbar ist.

Aber wie sieht es aus mit der Entscheidbarkeit anderer Probleme, die sich nicht unmittelbar auf Turingmaschinen beziehen? Wir werden in den folgenden beiden Abschnitten zwei Entscheidungsprobleme kennen lernen, die sich nicht auf Turingmaschinen beziehen und die trotz ihrer einfachen Formulierung unentscheidbar sind. Das bedeutet, diese Probleme können in ihrer Allgemeinheit nicht durch einen Algorithmus gelöst werden, egal wieviel Rechenzeit und Speicherplatz wir zur Verfügung stellen.

2.6.1 Hilberts zehntes Problem

Im Jahr 1900 präsentierte der Mathematiker David Hilbert 23 ungelöste mathematische Probleme auf einem Mathematiker-Kongreß in Paris. Hilbert formulierte das Problem (im Originalwortlaut) so:

Eine diophantische Gleichung mit irgendwelchen Unbekannten und mit ganzen rationalen Zahlenkoeffizienten sei vorgelegt: Man soll ein Verfahren angeben, nach welchem sich mittels einer endlichen Anzahl von Operationen entscheiden läßt, ob die Gleichung in den ganzen rationalen Zahlen lösbar ist.

Die „ganzen rationalen Zahlen“, von denen in diesem Problem die Rede ist, sind die ganzen Zahlen aus $\mathbb{Z}$, wie wir sie kennen. „Diophantische Gleichungen“ bezeichnen Gleichungen mit Polynomen in mehreren Variablen, die so definiert sind:

Ein *Term* ist ein Produkt aus Variablen mit einem konstanten Koeffizienten, z.B. ist

$$6 \cdot x \cdot x \cdot x \cdot y \cdot z \cdot z \quad \text{bzw.} \quad 6x^3yz^2$$

ein Term über den Variablen x, y, z mit dem Koeffizienten 6. Ein *Polynom* ist eine Summe von Termen, z.B.

$$6x^3yz^2 + 3xy^2 - x^3 - 10 \ .$$

Eine *diophantische Gleichung* setzt ein Polynom gleich Null. Die Lösungen der Gleichung entsprechen also den Nullstellen des Polynoms. Obiges Polynom hat

beispielsweise die Nullstelle

$$(x, y, z) = (5, 3, 0) \ .$$

Definition 2.27 (Hilberts zehntes Problem (in unseren Worten)) *Beschreibe einen Algorithmus, der entscheidet, ob ein gegebenes Polynom mit ganzzahligen Koeffizienten eine ganzzahlige Nullstelle hat.*

Die diesem Entscheidungsproblem zugrundeliegende Sprache ist

$$N \ = \ \{\, p \,|\, p \text{ ist ein Polynom mit einer ganzzahligen Nullstelle} \} \ .$$

Wir machen uns zunächst klar, dass N rekursiv aufzählbar ist: Gegeben sei ein Polynom p mit ℓ Variablen. Der Wertebereich von p entspricht der abzählbar unendlichen Menge $\mathbb{Z}^\ell$. Der folgende Algorithmus erkennt N:

- Zähle die ℓ-Tupel aus $\mathbb{Z}^\ell$ nacheinander auf und werte p für jedes dieser Tupel aus.
- Akzeptiere, sobald eine der Auswertungen den Wert *Null* ergibt.

Falls wir eine obere Schranke für die Absolutwerte der Nullstellen hätten, so bräuchten wir nur eine endliche Menge von ℓ-Tupeln aufzählen, und N wäre somit entscheidbar. Für Polynome über nur einer Variablen gibt es tatsächlich eine derartige obere Schranke: Für ein Polynom der Form

$$p(x) = a_k x^k + a_{k-1} x^{k-1} + \cdots + a_1 x + a_0$$

mit ganzzahligen Koeffizienten gilt

$$p(x) = 0, x \in \mathbb{Z} \Rightarrow x \text{ teilt } a_0. \text{ (Warum?)}$$

Also gibt es keine Nullstelle mit Absolutwert größer als $|a_0|$. Eingeschränkt auf Polynome mit nur einer Variable ist das Nullstellenproblem damit entscheidbar.

Für Polynome mit mehreren Variablen gibt es leider keine obere Schranke für die Absolutwerte der Nullstellen. Um das einzusehen, betrachte beispielsweise das Polynom $x + y$. Dieses Polynom hat die Nullstellen

$$(0, 0), (-1, 1), (1, -1), (-2, 2), (2, -2), \ldots \ .$$

Die Menge der Nullstellen ist also unbeschränkt. Aber vielleicht gibt es ja eine obere Schranke für diejenige Nullstelle mit den kleinsten Absolutwerten? Oder

vielleicht gibt es ganz andere Möglichkeiten, einem Polynom anzusehen, ob es eine ganzzahlige Nullstelle hat? Erst knapp siebzig Jahre, nachdem Hilbert sein Problem präsentiert hatte, konnte Yuri Matijasevič alle diese Fragen beantworten, und zwar negativ! Hilbert hatte die folgende Antwort nicht erwartet:

Satz 2.28 (Satz von Matijasevič (1970)) *Das Problem, ob ein ganzzahliges Polynom eine ganzzahlige Nullstelle hat, ist unentscheidbar.*

Damit ist Hilberts zehntes Problem unlösbar. Der Beweis des Satzes von Matijasevič beruht auf einer Kette von Reduktionen, durch die letztendlich das Halteproblem H auf das Nullstellenproblem N reduziert wird. Yuri Matijasevič hat „lediglich“ das letzte Glied dieser Kette geschlossen. Andere wichtige Beiträge zu diesem Ergebnis wurden zuvor von Martin Davis, Julia Robinson und Hilary Putnam erbracht. Wir werden nicht genauer auf den Beweis dieses Satzes eingehen, da er zu aufwendig ist, um ihn in dieser Grundstudiumsvorlesung zu besprechen.

Wir wollen zumindest für ein natürliches Problem, das sich nicht direkt mit Turingmaschinen beschäftigt, die Nichtberechenbarkeit nachweisen. Dazu untersuchen wir im folgenden Abschnitt ein scheinbar einfaches Puzzle.

2.6.2 Das Postsche Korrespondenzproblem

Das Postsche Korrespondenzproblem ist eine Art von Puzzle aus Dominos. Jedes Domino ist mit zwei Wörtern über einem Alphabet Σ beschrieben, ein Wort in der oberen Hälfte und eines in der unteren. Gegeben sei eine Menge K von Dominos, z.B.

$$K = \left\{ \left[\frac{b}{ca}\right], \left[\frac{a}{ab}\right], \left[\frac{ca}{a}\right], \left[\frac{abc}{c}\right] \right\} .$$

Die Aufgabe besteht darin, eine Folge von Dominos aus K zu ermitteln, so dass sich oben und unten dasselbe Wort ergibt. Die Folge soll aus mindestens einem Domino bestehen. Wiederholungen von Dominos sind erlaubt. Ein Beispiel für eine derartige *korrespondierende Folge* über K ist

$$\left[\frac{a}{ab}\right]\left[\frac{b}{ca}\right]\left[\frac{ca}{a}\right]\left[\frac{a}{ab}\right]\left[\frac{abc}{c}\right] .$$

Nicht für jede Menge K ist dies möglich, z.B. gibt es keine korrespondierende Folge für die Menge

$$K = \left\{ \left[\frac{abc}{ca}\right], \left[\frac{abca}{abc}\right], \left[\frac{abc}{bc}\right] \right\} ,$$

weil für jede Folge, die sich aus diesen Dominos bilden läßt, das obere Wort länger als das untere ist.

Definition 2.29 (Postsches Korrespondenzproblem (PKP)) *Eine Instanz des PKP besteht aus einer Menge*

$$K = \left\{ \left[\frac{x_1}{y_1}\right], \ldots, \left[\frac{x_k}{y_k}\right] \right\} ,$$

wobei x_i und y_i nichtleere Wörter über einem endlichen Alphabet Σ sind. Es soll entschieden werden, ob es eine korrespondierende Folge von Indizes $i_1, \ldots, i_n \in \{1, \ldots, k\}$, $n \geq 1$ gibt, also eine Folge, so dass gilt $x_{i_1} x_{i_2} \ldots x_{i_n} = y_{i_1} y_{i_2} \ldots y_{i_n}$.

Die Elemente der Menge K bezeichnen wir intuitiv als Dominos. Wir werden die Unentscheidbarkeit des PKP durch eine kurze Reduktionskette nachweisen, die einen Umweg über eine Variante des PKP nimmt. Die Modifikation liegt darin, dass wir einen Startdomino bestimmen, mit dem die korrespondierende Folge beginnen muss, nämlich $[\frac{x_1}{y_1}]$.

Definition 2.30 (Modifiziertes PKP (MPKP)) *Eine Instanz des MPKP besteht aus einer geordneten Menge*

$$K = \left(\left[\frac{x_1}{y_1}\right], \ldots, \left[\frac{x_k}{y_k}\right] \right) .$$

wobei x_i und y_i nichtleere Wörter über einem endlichen Alphabet Σ sind. Es soll entschieden werden, ob es eine korrespondierende Folge von Indizes $i_2, \ldots, i_n \in \{1, \ldots, k\}$, $n \geq 1$ gibt, so dass gilt $x_1, x_{i_2} \ldots x_{i_n} = y_1, y_{i_2} \ldots y_{i_n}$.

Wir werden die folgenden zwei Aussagen beweisen.

Lemma 2.31 $MPKP \leq PKP$.

Lemma 2.32 $H \leq MPKP$.

Aus der Transitivität der Reduktion (vgl. Übungsaufgabe 2.26) folgt dann $H \leq PKP$. Das Halteproblem ist nicht rekursiv, aber rekursiv aufzählbar. Das Komplement des Halteproblems ist nicht rekursiv aufzählbar. Dies zusammen mit Lemma 2.23 ergibt den folgenden Satz.

Satz 2.33 *Das PKP ist nicht rekursiv, aber rekursiv aufzählbar. Das Komplement des PKP ist nicht rekursiv aufzählbar.*

Wir müssen „nur“ noch die Lemmata 2.31 und 2.32 beweisen. Zunächst zeigen wir Lemma 2.31:
Beweis: Wir beschreiben die Funktion f: Seien $\#$ und $\$$ zwei Symbole, die nicht im Alphabet Σ des MPKP enthalten sind. Wir bilden

$$K = \left(\left[\frac{x_1}{y_1}\right], \ldots, \left[\frac{x_k}{y_k}\right] \right)$$

ab auf

$$f(K) \;=\; \left\{ \left[\frac{x'_0}{y'_0}\right], \left[\frac{x'_1}{y'_1}\right], \ldots, \left[\frac{x'_k}{y'_k}\right], \left[\frac{x'_{k+1}}{y'_{k+1}}\right] \right\},$$

wobei

- x'_i aus x_i (für $1 \leq i \leq k$) entsteht, indem wir hinter jedem Zeichen ein $\#$ einfügen, $x'_0 = \#x'_1$ und $x'_{k+1} = \$$.
- y'_i aus y_i (für $1 \leq i \leq k$) entsteht, indem wir vor jedem Zeichen ein $\#$ einfügen, $y'_0 = y'_1$ und $y'_{k+1} = \#\$$.

Syntaktisch inkorrekte Eingaben werden durch f auf das leere Wort $\epsilon \notin PKP$ abgebildet. Offensichtlich ist f berechenbar.

Wir geben zur Verdeutlichung ein Beispiel: Die MPKP-Instanz

$$K = \left(\left[\frac{ab}{a}\right], \left[\frac{c}{abc}\right], \left[\frac{a}{b}\right] \right)$$

wird abgebildet auf die PKP-Instanz

$$f(K) = \left\{ \left[\frac{\#a\#b\#}{\#a}\right], \left[\frac{a\#b\#}{\#a}\right], \left[\frac{c\#}{\#a\#b\#c}\right], \left[\frac{a\#}{\#b}\right], \left[\frac{\$}{\#\$}\right] \right\}.$$

Lösung des MPKP:

$$\left[\frac{ab}{a}\right] \left[\frac{a}{b}\right] \left[\frac{ab}{a}\right] \left[\frac{c}{abc}\right].$$

Lösung des PKP:

$$\left[\frac{\#a\#b\#}{\#a}\right] \left[\frac{a\#}{\#b}\right] \left[\frac{a\#b\#}{\#a}\right] \left[\frac{c\#}{\#a\#b\#c}\right] \left[\frac{\$}{\#\$}\right].$$

Zu zeigen: $K \in MPKP \Rightarrow f(K) \in PKP$. Sei $(1, i_2, \ldots, i_n)$ eine MPKP-Lösung für K, d.h.

$$x_1 x_{i_2} \ldots x_{i_n} = y_1 y_{i_2} \ldots y_{i_n} = a_1 a_2 \ldots a_s$$

für geeignet gewählte Symbole $a_1, \ldots, a_s$ aus Σ. Dann ist $(0, i_2, \ldots, i_n, k+1)$ eine PKP-Lösung für $f(K)$, denn

$$x'_0 x'_{i_2} \ldots x'_{i_n} \$ = \#a_1 \# a_2 \# \ldots \# a_s \# \$ = y'_0 y'_{i_2} \ldots y'_{i_n} \# \$ \,.$$

Gibt es also eine Lösung für K bzgl. MPKP, so gibt es auch eine Lösung für $f(K)$ bzgl. PKP. Somit haben wir gezeigt $K \in MPKP \Rightarrow f(K) \in PKP$.
Zu zeigen: $f(K) \in PKP \Rightarrow K \in MPKP$. Sei nun $(i_1, i_2, \ldots, i_n)$ eine PKP-Lösung *minimaler Länge* für $f(K)$.

- *Beobachtung 1:* Es gilt $i_1 = 0$ und $i_n = k+1$, weil nur x'_0 und y'_0 mit demselben Zeichen beginnen und nur x'_{k+1} und y'_{k+1} mit demselben Zeichen enden.

- *Beobachtung 2:* Es gilt $i_j \neq 0$ für $2 \leq j \leq n$, weil sonst zwei $\#$-Zeichen im oberen Wort direkt aufeinander folgen würden, was im unteren Wort unmöglich ist.

- *Beobachtung 3:* Es gilt $i_j \neq k+1$ für $1 \leq j < n$, denn würde das \$-Zeichen vorher auftreten, könnten wir die vorliegende minimale korrespondierende Folge nach dem ersten Vorkommen des \$-Zeichens abschneiden und hätten eine noch kürzere Lösung gefunden.

Aus den Beobachtungen folgt, dass unsere PKP-Lösung für $f(K)$ die Struktur

$$x'_0 x'_{i_2} \ldots x'_{i_n} = \#a_1 \# a_2 \# \ldots \# a_s \# \$ = y'_0 y'_{i_2} \ldots y'_{i_n}$$

hat für geeignet gewählte Symbole $a_1, \ldots, a_s$ aus Σ. Daraus ergibt sich die folgende MPKP-Lösung für K:

$$x_1 x_{i_2} \ldots x_{i_{n-1}} = a_1 a_2 \ldots a_s = y_1 y_{i_2} \ldots y_{i_{n-1}} \,.$$

Somit gilt $f(K) \in PKP \Rightarrow K \in MPKP$. □

Scheinbar haben Dominos wenig mit Turingmaschinen zu tun. In Lemma 2.32 wird dennoch behauptet, dass man mit Hilfe eines Puzzles aus Dominos das Halteproblem für Turingmaschinen entscheiden kann. Bevor wir in den Beweis des

Lemmas einsteigen, möchten wir auf der Basis eines umfangreichen Beispiels illustrieren, wie die Rechnung einer Turingmaschine durch ein Puzzle aus Dominos „simuliert“ werden kann. Betrachte die folgende TM M:

$$\Sigma = \{0,1\}, \Gamma = \{0,1,B\}, Q = \{q_0, q_1, q_2, \bar{q}\},$$

wobei q_0 der Anfangszustand ist und $\bar{q}$ der Endzustand. Die Überführungsfunktion δ sei gegeben durch

δ	0	1	B
q_0	$(q_0, 0, R)$	$(q_1, 1, R)$	$(\bar{q}, 1, N)$
q_1	$(q_2, 0, R)$	$(q_1, 1, R)$	$(\bar{q}, 1, N)$
q_2	$(q_2, 0, R)$	$(q_2, 1, R)$	(q_2, B, R)

Die TM M erkennt, ob das Eingabewort von der Form 0^i1^j, $i, j \geq 0$, ist. Bei Eingabe eines Wortes dieser Form terminiert M im Zustand $\bar{q}$ und akzeptiert, ansonsten läuft der Kopf im Zustand q_2 weiter und weiter nach rechts. Die Rechnung der TM auf einer gegebenen Eingabe kann durch eine Konfigurationsfolge beschrieben werden. Auf der Eingabe $w = 0011$ ist die Konfigurationsfolge beispielsweise

$$q_00011 \vdash 0q_0011 \vdash 00q_011 \vdash 001q_11 \vdash 0011q_1B \vdash 0011\bar{q}1 \ .$$

Wir möchten die Rechnung einer TM auf einer Eingabe durch ein Puzzle aus Dominos simulieren. Dieses Puzzle entspricht dem MPKP. Als Startdomino für das MPKP wählen wir ein Domino bei dem das untere Wort aus der Anfangskonfiguration mit ein paar zusätzlichen Trennsymbolen besteht:

$$\left[\frac{\#}{\#\#q_00011\#}\right].$$

Das Puzzle für unsere Beispielrechnung (M, w) enthält unter anderem jeweils ein Domino für jedes Zeichen aus $\Gamma \cup \{\#\}$:

$$\left[\frac{0}{0}\right], \left[\frac{1}{1}\right], \left[\frac{B}{B}\right], \left[\frac{\#}{\#}\right].$$

Wir erweitern diese Liste *erlaubter* Dominos um je ein Domino für jeden Eintrag in der Tabelle der Überführungsfunktion δ, der den jeweiligen Übergang inklusive der Kopfbewegung beschreibt.

$$\left[\frac{q_00}{0q_0}\right], \left[\frac{q_01}{1q_1}\right], \left[\frac{q_0B}{\bar{q}\ 1}\right], \left[\frac{q_10}{0q_2}\right], \left[\frac{q_11}{1q_1}\right], \left[\frac{q_1B}{\bar{q}\ 1}\right], \left[\frac{q_20}{0q_2}\right], \left[\frac{q_21}{1q_2}\right], \left[\frac{q_2B}{Bq_2}\right]$$

Wir werden später noch weitere Steine zur Liste erlaubter Dominos hinzufügen.

Beobachtung 2.34 *Wenn wir das Startdomino mit einer Folge von Dominos aus der Liste der erlaubten Dominos derart ergänzen, dass der obere String ein Präfix des unteren Strings ist, so*

- *rekonstruieren wir im unteren String die Konfigurationsfolge von M auf w, und*
- *der obere String folgt dem unteren mit einer Konfiguration im Rückstand.*

Die Rekonstruktion der Konfigurationsfolge in unserem Beispiel sieht so aus: Die ersten Dominos in der Lösung des Puzzles sind

$$\left[\frac{\#}{\#\#q_00011\#}\right] \quad \begin{array}{l} \left[\frac{\#}{\#}\right]\left[\frac{q_00}{0q_0}\right]\left[\frac{0}{0}\right]\left[\frac{1}{1}\right]\left[\frac{1}{1}\right]\left[\frac{\#}{\#}\right] \\ \left[\frac{\#}{\#}\right]\left[\frac{0}{0}\right]\left[\frac{q_00}{0q_0}\right]\left[\frac{1}{1}\right]\left[\frac{1}{1}\right]\left[\frac{\#}{\#}\right] \\ \left[\frac{\#}{\#}\right]\left[\frac{0}{0}\right]\left[\frac{0}{0}\right]\left[\frac{q_01}{1q_1}\right]\left[\frac{1}{1}\right]\left[\frac{\#}{\#}\right] \\ \left[\frac{\#}{\#}\right]\left[\frac{0}{0}\right]\left[\frac{0}{0}\right]\left[\frac{1}{1}\right]\left[\frac{q_11}{1q_1}\right]\left[\frac{\#}{\#}\right] \\ \left[\frac{\#}{\#}\right]\left[\frac{0}{0}\right]\left[\frac{0}{0}\right]\left[\frac{1}{1}\right]\left[\frac{1}{1}\right]\left[\frac{q_1\#}{\bar{q}\ 1\#}\right] \ \ldots \end{array}$$

Vielleicht ist es aufgefallen, dass wir im letzten Schritt ein wenig gemogelt haben. Wir haben ein Domino verwendet, das nicht in der zuvor spezifizierten Liste erlaubter Dominos enthalten ist. Tatsächlich ergänzen wir die Liste erlaubter Dominos um die folgenden Elemente:

$$\left[\frac{q_0\#}{\bar{q}\ 1\#}\right], \left[\frac{q_1\#}{\bar{q}\ 1\#}\right].$$

Die Aufgabe dieser Dominos ist es, Überführungen zu realisieren, die auf ein implizites Blank-Symbol am Ende der Konfiguration zurückgreifen, das virtuell zwischen dem Zeichen q_0 bzw. q_1 und dem $\#$ steht.

Wie können wir nun erreichen, dass der obere String seinen Rückstand am Ende der Rechnung aufholt? Zu diesem Zweck ergänzen wir die Liste der erlaubten Dominos um die folgenden Elemente:

$$\left[\frac{\bar{q}0}{\bar{q}}\right], \left[\frac{\bar{q}1}{\bar{q}}\right], \left[\frac{\bar{q}B}{\bar{q}}\right], \left[\frac{0\bar{q}}{\bar{q}}\right], \left[\frac{1\bar{q}}{\bar{q}}\right], \left[\frac{B\bar{q}}{\bar{q}}\right]$$

Desweiteren fügen wir noch ein *Abschlußdomino* hinzu.

$$\left[\frac{\#\bar{q}\#\#}{\#}\right].$$

Man beachte, dass diese Dominos nur dann zum Einsatz kommen können, wenn der Endzustand $\bar{q}$ erreicht ist, also nur wenn die Rechnung der TM terminiert.

Wir setzen die Rekonstruktion der Konfigurationsfolge in unserem Beispiel fort:

$$\begin{array}{ll} \ldots & \left[\frac{\#}{\#}\right]\left[\frac{0}{0}\right]\left[\frac{0}{0}\right]\left[\frac{1}{1}\right]\left[\frac{1}{1}\right]\left[\frac{q_1\#}{\bar{q}\ 1\#}\right] \\ & \left[\frac{\#}{\#}\right]\left[\frac{0}{0}\right]\left[\frac{0}{0}\right]\left[\frac{1}{1}\right]\left[\frac{1}{1}\right]\left[\frac{\bar{q}\ 1}{\bar{q}}\right]\left[\frac{\#}{\#}\right] \\ & \left[\frac{\#}{\#}\right]\left[\frac{0}{0}\right]\left[\frac{0}{0}\right]\left[\frac{1}{1}\right]\left[\frac{1\bar{q}}{\bar{q}}\right]\left[\frac{\#}{\#}\right] \\ & \left[\frac{\#}{\#}\right]\left[\frac{0}{0}\right]\left[\frac{0}{0}\right]\left[\frac{1\bar{q}}{\bar{q}}\right]\left[\frac{\#}{\#}\right] \\ & \left[\frac{\#}{\#}\right]\left[\frac{0}{0}\right]\left[\frac{0\bar{q}}{\bar{q}}\right]\left[\frac{\#}{\#}\right] \\ & \left[\frac{\#}{\#}\right]\left[\frac{0\bar{q}}{\bar{q}}\right]\left[\frac{\#}{\#}\right] \qquad \left[\frac{\#\bar{q}\#\#}{\#}\right]. \end{array}$$

Jetzt stimmt der obere mit dem unteren String überein. Skeptiker vergleichen jeweils den unteren String in einer Zeile mit dem oberen String in der darunterliegenden Zeile.

Die Idee hinter der obigen Konstruktion ist es, eine Eingabe für das Halteproblem in ein MPKP-Puzzle zu transformieren, so dass das Puzzle genau dann eine Lösung hat, wenn die im Halteproblem betrachtete TM auf ihrer Eingabe hält. Unser Beispiel hat erläutert, wie eine derartige Transformation für eine bestimmte Eingabe des Halteproblems aussehen könnte. Der folgende Beweis für Lemma 2.32 verallgemeinert und formalisiert das Vorgehen aus unserem Beispiel.

Beweis: Wir beschreiben eine berechenbare Funktion f, die eine syntaktisch korrekte Eingabe für H der Form $(\langle M\rangle, w)$ auf eine syntaktisch korrekte Instanz $K = f((\langle M\rangle, w))$ für das MPKP abbildet, so dass gilt

$$M \text{ hält auf } w \ \Leftrightarrow\ K \text{ hat eine Lösung } .$$

Syntaktisch nicht korrekte Eingaben für H werden auf syntaktisch nicht korrekte Eingaben für MPKP abgebildet. Das Alphabet, das wir für die MPKP-Instanz verwenden ist $\Gamma \cup Q \cup \{\#\}$, wobei gelte $\# \notin \Gamma \cup Q$.
Konstruktion der Funktion f: Gegeben sei das Tupel $(\langle M \rangle, w)$. Wir beschreiben, welche Dominos die Menge $K = f((\langle M \rangle, w))$ enthält. Das *Startdomino* ist von der Form

$$\left[\frac{\#}{\#\#q_0w\#}\right].$$

Desweiteren enthalte K die folgenden Arten von Dominos:
Kopierdominos:

$$\left[\frac{a}{a}\right] \text{ für alle } a \in \Gamma \cup \{\#\}$$

Überführungsdominos:

$$\left[\frac{qa}{q'c}\right] \quad \text{falls } \delta(q,a) = (q',c,N), \text{ für } q \in Q \setminus \{\bar{q}\},\ a \in \Gamma$$

$$\left[\frac{qa}{cq'}\right] \quad \text{falls } \delta(q,a) = (q',c,R), \text{ für } q \in Q \setminus \{\bar{q}\},\ a \in \Gamma$$

$$\left[\frac{bqa}{q'bc}\right] \quad \text{falls } \delta(q,a) = (q',c,L), \text{ für } q \in Q \setminus \{\bar{q}\},\ a,b \in \Gamma$$

Spezielle *Überführungsdominos*, die implizite Blanks berücksichtigen:

$$\left[\frac{\#qa}{\#q'Bc}\right] \quad \text{falls } \delta(q,a) = (q',c,L), \text{ für } q \in Q \setminus \{\bar{q}\},\ a \in \Gamma$$

$$\left[\frac{q\#}{q'c\#}\right] \quad \text{falls } \delta(q,B) = (q',c,N), \text{ für } q \in Q \setminus \{\bar{q}\}$$

$$\left[\frac{q\#}{cq'\#}\right] \quad \text{falls } \delta(q,B) = (q',c,R), \text{ für } q \in Q \setminus \{\bar{q}\}$$

$$\left[\frac{bq\#}{q'bc\#}\right] \quad \text{falls } \delta(q,B) = (q',c,L), \text{ für } q \in Q \setminus \{\bar{q}\},\ b \in \Gamma$$

$$\left[\frac{\#q\#}{\#q'Bc\#}\right] \quad \text{falls } \delta(q,B) = (q',c,L), \text{ für } q \in Q \setminus \{\bar{q}\}$$

Löschdominos:

$$\left[\frac{a\bar{q}}{\bar{q}}\right] \text{ und } \left[\frac{\bar{q}\ a}{\bar{q}}\right] \text{ für } a \in \Gamma$$

Abschlußdomino:

$$\left[\frac{\#\bar{q}\#\#}{\#}\right]$$

Dies sind alle Dominos in der MPKP-Instanz. Zur intuitiven Bedeutung der unterschiedlichen Arten von Dominos verweisen wir auf das vorab präsentierte ausführliche Beispiel. Die Beschreibung der Funktion f ist somit abgeschlossen.

Wir beweisen nun die Korrektheit der Konstruktion. Offensichtlich ist f berechenbar. Wir müssen noch nachweisen, dass M genau dann auf w hält, wenn die Eingabe K zur Sprache $MPKP$ gehört.

Zu zeigen: M hält auf $w \;\Rightarrow\; K \in MPKP$.

Wenn M auf w hält, so entspricht die Rechnung von M auf w einer endlichen Konfigurationsfolge der Form

$$k_0 \vdash k_1 \vdash \cdots \vdash k_{t-1} \vdash k_t \;\;,$$

wobei k_0 die Startkonfiguration und k_t die Endkonfiguration im Zustand $\bar{q}$ ist.

In diesem Fall können wir, beginnend mit dem Startdomino, nach und nach Kopier- und Überführungsdominos hinzulegen, so dass

- der untere String die vollständige Konfigurationsfolge von M auf w in der folgenden Form darstellt

$$\#\# \, k_0 \, \#\# \, k_1 \, \#\# \cdots \#\# \, k_{t-1} \, \#\# \, k_t \, \# \;\;,$$

- und der obere String ein Präfix des unteren Strings ist, nämlich

$$\#\# \, k_0 \, \#\# \, k_1 \, \#\# \cdots \#\# \, k_{t-1} \, \# \;\;.$$

Durch Hinzufügen von Löschdominos kann jetzt der Rückstand des oberen Strings fast ausgeglichen werden. Danach sind beide Strings identisch bis auf ein Suffix der Form

$$\#\bar{q}\# \;\;.$$

Dieses Suffix fehlt im oberen String. Nach Hinzufügen des Abschlußdominos

$$\left[\frac{\#\bar{q}\#\#}{\#}\right]$$

sind beide Strings somit identisch. Wenn M auf w hält, gilt somit $K \in MPKP$.

Zu zeigen: M hält nicht auf $w \;\Rightarrow\; K \notin MPKP$.

Zum Zweck des Widerspruchs nehmen wir an, dass M nicht auf w hält, aber $K \in MPKP$.

Beobachtung 2.35 *Jede korrespondierende Folge enthält zumindest einen Lösch- oder Abschlußdomino, denn sonst wäre der untere String länger als der obere, weil beim Startdomino der obere String kürzer als der untere ist, und bei den Kopier- und Überführungsdominos der obere String niemals länger als der untere ist.*

Sei nun $1, i_2, \ldots, i_n$ eine korrespondierende Folge für K. Die Teilfolge

$$1, i_2, \ldots, i_{s-1}$$

bestehe nur aus dem Startdomino sowie folgenden Kopier- und Überführungsdominos. Der Domino i_s sei der erste Lösch- oder Abschlußdomino in der Folge. Zunächst betrachten wir die Teilfolge $1, i_2, \ldots, i_{s-1}$.

- Die Kopier- und Überführungsdominos sind so definiert, dass bei Einhaltung der Übereinstimmung zwischen dem oberen und dem unteren String die Konfigurationsfolge der Rechnung von M auf w entsteht.
- Der obere String folgt dabei dem unteren String mit Rückstand einer Konfiguration.
- Da die Rechnung von M auf w nicht terminiert, kann in der Konfigurationsfolge nicht der Zustand $\bar{q}$ auftauchen.

Der Lösch- oder Abschlußdomino i_s enthält jedoch im oberen Wort den Zustand $\bar{q}$. Das Hinzufügen dieses Dominos verletzt somit die Übereinstimmung zwischen den beiden Strings. Dies steht jedoch im Widerspruch zur Annahme, dass eine korrespondierende Folge vorliegt. □

2.7 Mächtigkeit von Programmiersprachen

Lässt sich die Mächtigkeit einer TM mit der Mächtigkeit einer Programmiersprache wie z.B. Java vergleichen? – Ein Programm in Java kann in Assemblercode für eine RAM übersetzt werden. Die Berechnungskraft einer derartigen Programmiersprache kann also nicht größer als die einer RAM sein und somit auch nicht größer als die einer TM, die ja eine RAM simulieren kann. Möglicherweise nutzt eine Programmiersprache jedoch nicht die volle Berechnungskraft dieser Modelle aus.

Definition 2.36 *Eine Programmiersprache wird als* Turing-mächtig *bezeichnet, wenn jede berechenbare Funktion auch durch ein Programm in dieser Programmiersprache berechnet werden kann.*

Eine praktische Programmiersprache wie Java hat alle möglichen Features. Die meisten dieser Features dienen der Bequemlichkeit des Programmierens. In diesem Abschnitt untersuchen wir, welche Arten von Befehlen wirklich benötigt werden, um eine *Turing-mächtige* Programmiersprache zu erhalten.

2.7.1 WHILE-Programme

Die Programmiersprache WHILE ist ein Beispiel für eine Turing-mächtige Programmiersprache, die auf das Wesentliche reduziert ist. Syntaktisch besteht ein WHILE-Programm aus

- einer konstanten Anzahl Variablen $x_0 \;\; x_1 \;\; x_2 \ldots$
- den drei Konstanten $-1 \;\; 0 \;\; 1$
- den vier Symbolen ; := + $\neq$
- den drei Schlüsselwörtern WHILE DO END

Die Syntax ist induktiv definiert: Für jedes $c \in \{-1, 0, 1\}$ ist die Zuweisung

$$x_i \;:=\; x_j + c$$

ein WHILE-Programm. Falls P_1 und P_2 WHILE-Programme sind, dann ist auch

$$P_1; P_2$$

ein WHILE-Programm. Falls P ein WHILE-Programm ist, dann ist auch

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

ein WHILE-Programm.

Wir kommen nun zur Semantik dieser Programme: WHILE-Programme berechnen k-stellige Funktionen der Form $f : \mathbb{N}^k \to \mathbb{N}$. Da wir natürliche Zahlen in Bit-Strings transformieren können und umgekehrt, ist dies kein grundlegender Unterschied zu den Funktionen, die von TMn berechnet werden. Die Eingabe

eines WHILE-Programmes ist in den Variablen $x_1, \ldots, x_k$ enthalten. Alle anderen Variablen werden mit 0 initialisiert. Programme der Form $x_i := x_j + c$ sind Zuweisungen des Wertes $x_j + c$ an die Variable x_i, wobei der Wert 0 nicht unterschritten werden kann (d.h. $0 + (-1) = 0$). In einem WHILE-Programm P_1; P_2 wird zunächst P_1 und dann P_2 ausgeführt. Das Programm WHILE $x_i \neq 0$ DO P END hat die Bedeutung, dass P solange ausgeführt wird bis x_i den Wert 0 erreicht hat. Das Resultat eines WHILE-Programms ist die Zahl, die sich am Ende der Rechnung in der Variable x_0 ergibt.

Satz 2.37 *Die Programmiersprache WHILE ist Turing-mächtig.*

Beweis: In Übungsaufgabe 1.15 haben wir gezeigt, dass jede TM durch eine RAM simuliert werden kann, die nur eine konstante Anzahl von Registern für natürliche Zahlen benutzt und mit dem eingeschränkten Befehlssatz LOAD, CLOAD, STORE, CADD, CSUB, GOTO, IF $c(0) \neq 0$ GOTO, END auskommt. Deshalb müssen wir jetzt nur noch zeigen, wie eine beliebige Funktion, die durch eine derartig eingeschränkte RAM berechnet werden kann, durch ein WHILE-Programms berechnet werden kann.

Sei Π ein beliebiges RAM-Programm mit eingeschränktem Befehlssatz, das aus ℓ Zeilen besteht und k Register für natürliche Zahlen benutzt. Wir speichern den Inhalt von Register $c(i)$, für $0 \leq i \leq k$, in der Variable x_i des WHILE-Programms. In der Variable x_{k+1} speichern wir zudem den Befehlszähler b der RAM ab, und die Variable x_{k+2} verwenden wir, um eine Variable zu haben, die immer den initial gesetzen Wert 0 enthält. Die oben aufgelisteten RAM-Befehle werden nun in Form von konstant vielen Zuweisungen der Form $x_i := x_j + c$ mit $c \in \{0, 1\}$ implementiert. Der RAM-Befehl LOAD i wird beispielsweise in das WHILE-Programm

$$x_0 := x_i + 0;\ x_{k+1} := x_{k+1} + 1$$

transformiert, wobei die zweite der beiden Zuweisungen die Änderung des Befehlszählers durchführt. Der RAM-Befehl CLOAD i wird analog in das WHILE-Programm

$$x_0 := x_{k+2} + 0;\ \underbrace{x_0 := x_0 + 1;\ \ldots;\ x_0 := x_0 + 1;}_{i \text{ mal}}\ x_{k+1} := x_{k+1} + 1$$

transformiert, d.h. wir weisen x_0 zunächst den Wert 0 zu, addieren i Mal eine 1 hinzu und aktualisieren dann den Befehlszähler. Die RAM-Befehle STORE, CADD, CSUB und GOTO lassen sich leicht auf ähnliche Art realisieren. Der RAM-Befehl IF $c(0) \neq 0$ GOTO j wird durch das WHILE-Programm

$x_{k+1} := x_{k+1} + 1;$ ($b := b + 1$)
$x_{k+3} := x_0 + 0;$ (*help* $:= c(0)$)
WHILE $x_{k+3} \neq 0$ DO (while *help* $\neq 0$ do)
$x_{k+1} := x_{k+2} + 0;\ \underbrace{x_{k+1} := x_{k+1} + 1; \cdots + 1;}_{j \text{ mal}}$ ($b := j$)
$x_{k+3} := x_{k+2} + 0$ (*help* $:= 0$)
END (end of while)

ersetzt, wobei die Variable x_{k+1} dem Befehlszähler b entspricht und die Variable x_{k+3} als Hilfsvariable *help* fungiert. Den RAM-Befehl END ersetzen wir durch das WHILE-Programm $x_{k+1} = 0$, d.h. der Befehlszähler b wird auf 0 gesetzt.

Jede Zeile des RAM-Programms wird nun wie oben beschrieben in ein WHILE-Programm transformiert. Das WHILE-Programm für Zeile i bezeichnen wir mit P_i. Das Programm P_i soll nur ausgeführt werden, wenn der Befehlszähler, also die Variable x_{k+1}, den Wert i hat. Deshalb betten wir P_i in ein WHILE-Programm P_i' mit der folgenden Semantik ein:

$$\text{Falls } x_{k+1} = i \text{ dann führe } P_i \text{ aus.}$$

Übungsaufgabe 2.38 *Beschreibe eine mögliche Implementierung des WHILE-Programms P_i'.*

Nun fügen wir die WHILE-Programme $P_1', \ldots, P_\ell'$ zu einem WHILE-Programm P der Form

$$x_{k+1} := 1; \text{ WHILE } x_{k+1} \neq 0 \text{ DO } P_1'; \ldots; P_\ell' \text{ END}$$

zusammen. Die äußere Schleife stellt sicher, dass bei Ausführung des WHILE-Programms die Befehle in genau der Reihenfolge abgearbeitet werden, wie es die RAM vorgibt. Somit berechnet P dieselbe Funktion, wie das gegebene RAM-Programm Π. □

Offensichtlich kann man mit Programmiersprachen wie C, C++, Java oder auch Haskell alles implementieren, was man in WHILE-Programmen implementieren kann. Folglich sind alle diese Programmiersprachen Turing-mächtig. Interessanterweise haben selbst TEX und PostScript diese Eigenschaft. Im nächsten Abschnitt sehen wir allerdings ein Beispiel für eine Programmiersprache, die diese Eigenschaft nicht hat, obwohl diese Programmiersprache nur eine kleine, scheinbar harmlose Änderung zu WHILE-Programmen aufweist.

2.7.2 LOOP-Programme

LOOP-Programme sind definiert wie WHILE-Programme bis auf die folgende Änderung. Wir ersetzen das WHILE-Konstrukt durch ein LOOP-Konstrukt der folgenden Form:

$$\text{LOOP } x_i \text{ DO } P \text{ END} \ ,$$

wobei die Variable x_i nicht in P vorkommen darf. Die Semantik dieses Konstruktes besagt, dass das Programm P sooft ausgeführt wird, wie es der Wert der Variable x_i vorgibt.

Es ist leicht zu sehen, dass man jedes LOOP-Konstrukt durch ein WHILE-Konstrukt ersetzen kann. Jedoch kann nicht jedes WHILE-Konstrukt durch ein LOOP-Konstrukt ersetzt werden. Offensichtlich kann man WHILE-Konstrukte konstruieren, die einer Endlosschleife entsprechen, also nicht terminieren. Mit dem LOOP-Konstrukt ist dies nicht möglich, da die Anzahl der Iterationen der LOOP-Schleife schon beim Betreten der Schleife bestimmt ist. Im Gegensatz zu WHILE-Programmen (und auch TM- oder RAM-Programmen) terminieren LOOP-Programme immer. Dies ist natürlich eine gute Eigenschaft von LOOP-Programmen. Allerdings stellt sich die Frage, ob durch diese positive Eigenschaft die Berechnungskraft eingeschränkt ist.

Wir bezeichnen die Klasse der durch LOOP-Programme berechenbaren Funktionen als *LOOP-berechenbare Funktionen*. Man könnte hoffen, dass alle berechenbaren totalen Funktionen auch LOOP-berechenbar sind. Ackermann hat jedoch 1928 eine totale Funktion vorgestellt, die zwar berechenbar ist, nicht aber durch LOOP-Programme berechnet werden kann. Die *Ackermannfunktion* $a : \mathbb{N}^2 \to \mathbb{N}$ ist folgendermaßen definert:

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

Die Ackermannfunktion kann durch eine TM berechnet werden, denn die TM kann rekursive Aufrufe in Form von dynamischen Unterprogrammaufrufen realisieren (vgl. Abschnitt 1.2.1).

Die Ackermannfunktion ist streng monoton wachsend in beiden Parametern. Eine für uns wichtige Eigenschaft der Ackermannfunktion ist, dass sie sehr schnell

wächst. Durch einfache Induktion nach n erhält man beispielsweise

- $A(1,n) = n+2$,
- $A(2,n) = 2n+3$,
- $A(3,n) = 8 \cdot 2^n - 3$,
- $A(4,n) = \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{\substack{n+2 \text{ viele} \\ \text{Potenzen}}} - 3$,

Die Funktion $A(5,n)$ wächst so gigantisch schnell, dass uns auf Anhieb keine anschauliche Darstellung zur Beschreibung dieser Funktion einfällt.

Um nachzuweisen, dass die Ackermannfunktion nicht LOOP-berechenbar ist, werden wir zeigen, dass die Ackermannfunktion schneller wächst als das maximal mögliche Wachstum der Variableninhalte in einem beliebigen LOOP-Programm. Dazu müssen wir zunächst eine Funktion definieren, die das Wachstum der Variableninhalte eines LOOP-Programms P beschreibt. Seien $x_0, x_1, \ldots, x_k$ die Variablen von P. Wenn die Variablen initial die Werte $a = (a_0, \ldots, a_k) \in \mathbb{N}^{k+1}$ haben, dann sei $f_P(a)$ das $(k+1)$-Tupel der Variablenwerte nach Ausführung von P und $|f_P(a)|$ die Summe dieser Werte. Wir definieren nun die Funktion $F_P : \mathbb{N} \to \mathbb{N}$ durch

$$F_P(n) = \max \left\{ |f_P(a)| \,\middle|\, a = (a_0, \ldots, a_k) \in \mathbb{N}^{k+1} \text{ mit } \sum_{i=0}^{k} a_i \leq n \right\} .$$

Intuitiv beschreibt die Funktion F_P das maximale Wachstum der Variablenwerte in einem LOOP-Programm P. Wir zeigen nun, dass $F_P(n)$ für alle $n \in \mathbb{N}$ echt kleiner ist als $A(m,n)$, wenn der Parameter m genügend groß in Abhängigkeit von P gewählt wird. Beachte, für ein festes Programm P ist der Parameter m eine Konstante.

Lemma 2.39 *Für jedes LOOP-Programm P gibt es eine natürliche Zahl m, so dass für alle n gilt: $F_P(n) < A(m,n)$.*

Beweis: Betrachte ein beliebiges LOOP-Programm P. Zum Beweis des Lemmas inklusive der Bestimmung der Konstante m verwenden wir eine strukturelle Induktion über den Aufbau von P.

Falls P die Form $x_i := x_j + c$ für $c \in \{-1, 0, 1\}$ hat, so gilt $F_P(n) \leq 2n + 1$, da beim Aufruf des Programmes $x_j \leq n$ gilt. Es folgt $F_P(n) < A(2, n)$. Damit ist der Induktionsanfang gemacht.

Falls P von der Form $P_1; P_2$ ist, dann können wir als Induktionsannahme voraussetzen, dass es eine natürliche Zahl q gibt, so dass $F_{P_1}(\ell) < A(q, \ell)$ und $F_{P_2}(\ell) < A(q, \ell)$ für alle $\ell \in \mathbb{N}$. Es folgt

$$F_P(n) \;\leq\; F_{P_2}(F_{P_1}(n)) \;<\; A(q, A(q, n)) \;\; .$$

Übungsaufgabe 2.40 *Zeige durch eine Induktion nach n, dass gilt $A(q, n) \leq A(q + 1, n - 1)$.*

Wir können somit $A(q, n)$ durch $A(q + 1, n - 1)$ substituieren und erhalten

$$F_P(n) \;<\; A(q, A(q + 1, n - 1)) \;=\; A(q + 1, n) \;\; ,$$

wobei die letzte Gleichung unmittelbar aus der rekursiven Definition von A folgt. In diesem Fall können wir also $m = q + 1$ wählen.

Wir untersuchen nun den Fall, dass P von der Form LOOP x_i DO Q END ist. Die Funktion $F_P(n)$ wird durch Maximumbildung über initiale Variablenbelegungen $a_0, \ldots, a_k$ mit der Eigenschaft $a_0 + a_1 + \ldots + a_k \leq n$ gebildet. Da die Variable x_i in Q nicht vorkommt, ist die Anzahl der Durchläufe der LOOP-Schleife gleich a_i. Für ein vorgegebenes $n \in \mathbb{N}$ sei $\alpha(n) \in \mathbb{N}$ derjenige Wert für a_i, der das Maximum von $F_P(n)$ bildet. Zur Vereinfachung der Notation schreiben wir α statt $\alpha(n)$. Nun gilt

$$F_P(n) \;\leq\; F_Q(F_Q(\ldots F_Q(F_Q(n - \alpha))\ldots)) + \alpha \;\; ,$$

wobei die Funktion $F_Q(\cdot)$ hier α-fach ineinander eingesetzt ist. Als Induktionsannahme setzen wir voraus, dass es eine natürliche Zahl q mit der Eigenschaft $F_Q(\ell) < A(q, \ell)$ für alle $\ell \in \mathbb{N}$ gibt. Wegen der Ganzzahligkeit der betrachteten Funktionen folgt $F_Q(\ell) \leq A(q, \ell) - 1$. Diese Ungleichung wenden wir zunächst nur auf die äußerste Schachtelung $F_Q(\ldots)$ an und erhalten

$$F_P(n) \;\leq\; A(q, F_Q(\ldots F_Q(F_Q(n - \alpha))\ldots)) + \alpha - 1 \;\; .$$

Wenn wir dasselbe Prinzip auf alle Verschachtelungen anwenden, ergibt sich

$$F_P(n) \;\leq\; A(q, A(q, \ldots A(q, A(q, n - \alpha))\ldots)) \;\; .$$

Die Monotonie der Ackermannfunktion liefert nun

$$F_P(n) \;\leq\; A(q, A(q, \ldots A(q, A(q+1, n-\alpha))\ldots)) \;\; .$$

Aus der Definition der Ackermannfunktion ergibt sich $A(q+1, y) = A(q, A(q+1, y-1))$. Auf die innere Verschachtelung angewendet ergibt sich somit

$$F_P(n) \;\leq\; A(q, A(q, \ldots A(q+1, n-\alpha+1)\ldots)) \;\; ,$$

wobei die Funktion $A(q, \cdot)$ jetzt nur noch $(\alpha-1)$-fach ineinander verschachtelt ist. Wenn wir diese Art der Substitution noch weitere $(\alpha - 2)$ mal anwenden, ergibt sich

$$F_P(n) \;\leq\; A(q+1, n-1) \;<\; A(q+1, n) \;\; .$$

Also genügt es auch in diesem Fall $m = q + 1$ zu wählen. Damit haben wir Lemma 2.39 durch strukturelle Induktion nachgewiesen. □

Satz 2.41 *Die Ackermannfunktion ist nicht LOOP-berechenbar.*

Beweis: Angenommen die Ackermannfunktion ist LOOP-berechenbar, dann ist die Funktion $B(n) = A(n, n)$ ebenfalls LOOP-berechenbar. Sei P ein LOOP-Programm für B. Es gilt $B(n) \leq F_P(n)$. Zu P wählen wir nun gemäß Lemma 2.39 die natürliche Zahl m, so dass für alle n gilt $F_P(n) < A(m, n)$. Für $n = m$ ergibt sich nun

$$B(n) \;\leq\; F_P(n) \;<\; A(m, n) \;=\; A(n, n) \;=\; B(n) \;\; .$$

Dies ist offensichtlich ein Widerspruch, woraus folgt dass die Ackermannfunktion nicht LOOP-berechenbar ist. □

2.8 Primitiv rekursive und μ-rekursive Funktionen

Historisch wurde die Klasse der LOOP-berechenbaren Funktionen nicht in Form von LOOP-Programmen vorgestellt, sondern wie auch die berechenbaren (rekursiven) Funktionen in Form einer Menge von Kompositionsregeln. Diese nachfolgend vorgestellten Regeln bilden durch induktive Anwendung die Klasse der sogenannten *primitiv rekursiven* Funktionen.

Wurzel zur Bildung der primitiv rekursiven Funktionen sind die folgenden *Basisfunktionen*:

- *Nullfunktionen* $o : \mathbb{N}^k \to \mathbb{N}$ für alle $k \in \mathbb{N}$ mit

$$o(x_1, \ldots, x_k) = 0 \qquad \text{für alle } x_1, \ldots, x_k \in \mathbb{N}$$

- *Projektionen* für alle $1 \leq i \leq k \in \mathbb{N}$:

$$p_{|i}(x_1, \ldots, x_i, \ldots, x_k) = x_i \quad \text{für alle } x_1, \ldots, x_k \in \mathbb{N}$$

- *Nachfolgerfunktion* $s : \mathbb{N} \to \mathbb{N}$ mit

$$s(n) = n + 1 \qquad \text{für alle } n \in \mathbb{N}$$

Neben den Basisfunktionen sind Funktionen, welche durch Anwendung der folgenden Bildungsregeln entstehen, ebenfalls primitiv rekursiv.

- *Kompositionen* primitiv rekursiver Funktionen $h, g_1, \ldots, g_n$ der Form

$$f(x_1, \ldots, x_k) = h(g_1(x_1, \ldots, x_k), \ldots, g_n(x_1, \ldots, x_k))$$

  sind selber auch primitiv rekursiv.

- *Primitive Rekursionen* primitiv rekursiver Funktionen g, h der Form

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

  sind wiederum primitiv rekursiv.

Diese beiden Regeln lassen sich sehr ergiebig zur Bildung neuer Funktionen aus den Basisfunktionen anwenden. Mit ihrer Hilfe lassen sich fast alle grundlegenden Funktionen definieren. Bis zur Entdeckung der Ackermannfunktion vermutete man daher, dass die Menge der primitiv rekursiven Funktionen mit der Menge der berechenbaren totalen Funktionen übereinstimmt.

Tatsächlich gleicht die Menge der primitiv rekursiven Funktionen aber genau der Menge der LOOP-berechenbaren Funktionen. Bevor wir den Beweis dazu angeben, wollen wir uns die Definition primitiv rekursiver Funktionen anhand einiger Beispiele veranschaulichen.

- Die Addition $add : \mathbb{N}^2 \to \mathbb{N}$ lässt sich schreiben als

$$\begin{aligned} add(0, x) &= x \\ add(n+1, x) &= s(p_{|1}(add(n, x), n, x)) \end{aligned}$$

  Statt der formalen Angabe $s(p_{|1}(add(n, x), n, x))$ benutzen wir auch die verkürzende Schreibweise $s(add(n, x))$.

- Die Multiplikation $mult : \mathbb{N}^2 \to \mathbb{N}$ definieren wir durch

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

- Die Subtraktion $sub : \mathbb{N}^2 \to \mathbb{N}$ schreiben wir als

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

  wobei u durch $u(0) = 0$ und $u(n+1) = n$ definiert ist.

Satz 2.42 *Die Klasse der primitiv rekursiven Funktionen stimmt genau mit der Klasse der LOOP-berechenbaren Funktionen überein.*

Beweis: Wir zeigen zunächst, dass alle primitiv rekursiven Funktionen als LOOP-Programme implementiert werden können. Dazu führen wir einen Induktionsbeweis über den Aufbau der Funktion.

Basisfunktionen sind offensichtlich LOOP-berechenbar und dienen uns als Induktionsanfang. Auch Kompositionen primitiv rekursiver Funktionen können wir durch Hintereinanderausführung der entsprechenden LOOP-Programme simulieren. Es bleibt also nur zu zeigen, dass, falls f durch primitive Rekursion definiert ist, d.h. in der Form

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

gegeben ist, ebenfalls eine Implementierung möglich ist.

Laut Induktionsannahme existieren zwei LOOP-Programme P_g und P_h, welche die Funktionen g und h berechnen. Somit ist das folgende LOOP-Programm wohldefiniert und berechnet die gewünschte Funktion.

$y := P_g(x_1, \ldots, x_k)$;
LOOP n DO $y := P_h(y, n-1, x_1, \ldots, x_k)$ END

Für die Umkehrrichtung benutzen wir eine bijektive Abbildung $encode : \mathbb{N}^k \to \mathbb{N}$ (z.B. die Cantorsche Tupelfunktion), um Modifikationen auf k Variablen durch eine einstellige Funktion darzustellen.

Man kann sich leicht veranschaulichen, dass die in der Übung vorgestellte Cantorsche Paarfunktion $\pi : \mathbb{N}^2 \to \mathbb{N}$ mit

$$\pi(x, y) = \binom{x + y + 1}{2} + y$$

und ihre Erweiterung $\pi^{(k+1)} : \mathbb{N}^{k+1} \to \mathbb{N}$ auf $(k+1)$-Tupel natürlicher Zahlen durch

$$\pi^{(k+1)}(x_0, \ldots, x_k) = \pi(\pi^{(k)}(x_0, \ldots, x_{k-1}), x_k)$$

primitiv rekursiv ist.

Ebenso kann man die Umkehrfunktionen d_i mit

$$d_i(encode(x_0, \ldots, x_i, \ldots, x_k)) = x_i$$

zum Dekodieren der einzelnen Elemente primitiv rekursiv definieren.

Nun sind wir in der Lage, LOOP-Programme als primitiv rekursive Funktionen anzugeben. Ein LOOP-Programm P der Form $x_i := x_j \pm c$ kann als primitiv rekursive Funktion

$$g_P(z) = encode(d_0(z), \ldots, d_{i-1}(z), d_i(z) \pm c, d_{i+1}(z), \ldots, d_k(z))$$

geschrieben werden.

Für die Hintereinanderausführung zweier Programme $Q; R$ verwenden wir die Komposition $g_R(g_Q(z))$.

Das Programm LOOP x_i DO Q END wird durch

$$g_{\text{LOOP}}(z) = h(d_i(z), z)$$

mit $h(0, z) = z$ und $h(n+1, z) = g_Q(h(n, z))$ angegeben. □

Aufgrund der gerade gezeigten Äquivalenz zwischen der Klasse der LOOP-berechenbaren Funktionen auf der einen und der Klasse der primitiv rekursiven Funktionen auf der anderen Seite, werden wir die beiden Begriffe oft auch synonym füreinander verwenden.

Aus Abschnitt 2.7.2 wissen wir bereits, dass nicht alle rekursiven (also Turing- bzw. WHILE-berechenbaren) Funktionen primitiv rekursiv sind. Wir möchten uns daher im Folgenden einer Charakterisierung der WHILE-berechenbaren Funktionen zuwenden.

Durch Hinzunahme einer einzelnen weiteren Kompositionsregel können wir die induktive Definition der primitiv rekursiven Fuktionen so erweitern, dass sie alle berechenbaren Funktionen umfasst.

- Die *μ-Rekursion* μf einer Funktion f ist definiert durch

$$\mu f := \min\{n \mid f(n, x_1, \ldots, x_k) = 0 \text{ und für alle } m < n \text{ ist } f(m, x_1, \ldots, x_k) \text{ definiert}\},$$

  wobei $\min\{\emptyset\} := \perp$.

Die Klasse der *μ-rekursiven Funktionen* setzt sich nun induktiv aus den Basisfunktionen, Kompositionen, primitiven Rekursionen und μ-Rekursionen μ-rekursiver Funktionen zusammen.

Satz 2.43 *Die Klasse der μ-rekursiven Funktionen stimmt genau mit der Klasse der WHILE-/TURING-berechenbaren Funktionen überein.*

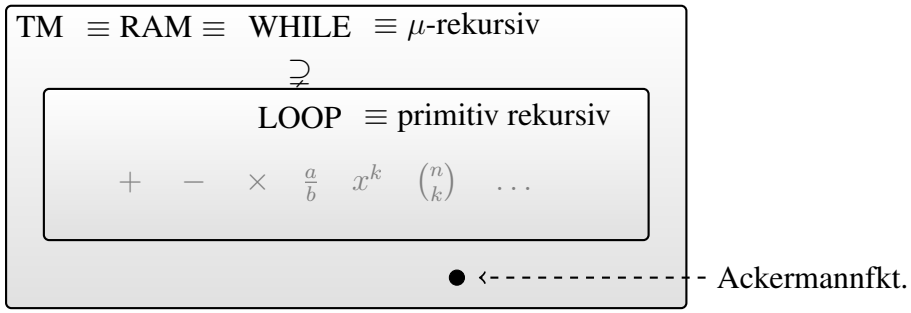

Abbildung 2.4: Hierarchie der vorgestellten Funktionsklassen

Beweis: Da wir im Vergleich zu den primitiv rekursiven Funktionen (welche wir, wie gesehen, als LOOP- und somit auch als WHILE-Programm darstellen können) lediglich die zusätzliche Verwendung des μ-Operators erlauben, müssen wir nur ein diesem Fall entsprechendes WHILE-Konstrukt angeben. Wir implementieren μf durch das WHILE-Programm

$x_0 := 0;$
$y := f(0, x_1, \ldots, x_k);$
WHILE $y \neq 0$ DO
$x_0 := x_0 + 1;$
$y := f(x_0, x_1, \ldots, x_k);$
END

Umgekehrt ersetzen wir ein WHILE-Programm P der Form

WHILE $x_i \neq 0$ DO Q END

durch die Funktion

$$g_P(z) = h(\mu(d_i h)(z), z)$$

wobei $h(n, z)$ wie im vorigen Beweis definiert ist und den Zustand der Programmvariablen $z = encode(x_0, \ldots, x_k)$ nach n Ausführungen von Q wiedergibt. Alle anderen WHILE-Konstrukte können wir wie im vorigen Beweis bereits als primitiv rekursive Funktionen darstellen. □

Kapitel 3

Komplexität

Die Komplexitätstheorie beschäftigt sich mit der „Komplexität von Problemen“. Die Komplexität eines Problems ist ein Maß dafür, wie schwierig es ist, das Problem zu lösen. Sie wird beschrieben durch Angabe der Ressourcen, die man zur Lösung des Problems auf einem Rechner benötigt. Bei diesen Ressourcen handelt es sich typischerweise um Rechenzeit und Speicherplatz; in dieser Vorlesung vornehmlich um die Rechenzeit. Die in der Komplexitätstheorie wohl am intensivsten untersuchte Fragestellung dreht sich darum, welche Probleme durch einen „effizienten Algorithmen“ gelöst werden können.

3.1 Die Komplexitätsklasse P

3.1.1 Motivation und Definition

Algorithmen werden häufig umgangssprachlich oder in einer Art *Pseudocode* beschrieben, der an eine Programmiersprache angelehnt ist. Bei der Analyse der Algorithmen wird in der Praxis häufig auf das uniforme Kostenmaß der Registermaschine (Random Access Machine – RAM) zurückgegriffen. Das uniforme Kostenmaß der RAM erlaubt es aber Rechenoperationen in konstanter Zeit durchzuführen, die in der Praxis nicht in konstanter Zeit möglich sind, z.B. die Addition oder Multiplikation beliebig großer Zahlen. In vielen (aber nicht allen) praktischen Anwendungen liegen häufig Zahlen mit eingeschränktem Wertebereich vor, die tatsächlich durch einen Rechner in einem Schritt verarbeitet werden können. Deshalb kann die Verwendung des uniformen Kostenmaßes für die praktische Algorithmenentwicklung durchaus sinnvoll sein. Wenn es aber darum geht

eine Theorie zu entwickeln, die die Komplexität von Problemen im Allgemeinen erfasst, möchten wir insbesondere auch die Komplexität des Rechnens mit großen Zahlen berücksichtigen. Wir greifen deshalb auf ein genaueres Kostenmaß zurück.

Auch wir werden in dieser Vorlesung Algorithmen umgangssprachlich oder in Pseudocode beschreiben. Die Algorithmenbeschreibung sollte so gefasst sein, dass implizit klar ist, wie der Algorithmus auf einer RAM ausgeführt werden kann. Für die Laufzeitanalyse legen wir das logarithmische Kostenmaß der RAM zugrunde, bei dem die Laufzeitkosten eines Rechenschrittes proportional zur Anzahl der angesprochenen Bits sind. Solange nichts anderes festgelegt ist, werden alle Zahlen, die in der Eingabe vorkommen oder während der Rechnung erzeugt werden, als binär kodiert vorausgesetzt. Die Laufzeit bemessen wir als Funktion der *Eingabelänge*, die wir ebenfalls in Bits messen.

Definition 3.1 (worst case Laufzeit eines Algorithmus) *Die* worst case Laufzeit $t_A(n)$, $n \in \mathbb{N}$, *eines Algorithmus A entspricht den maximalen Laufzeitkosten auf Eingaben der Länge n bezüglich des logarithmischen Kostenmaßes der RAM.*

Wenn wir in dieser Vorlesung die Laufzeit eines Algorithmus analysieren, untersuchen wir immer die worst case Laufzeit, auch wenn wir bisweilen den Zusatz *worst case* aus Bequemlichkeit nicht explizit erwähnen.

Definition 3.2 (Polynomialzeitalgorithmus) *Wir sagen, die worst case Laufzeit $t_A(n)$ eines Algorithmus A ist* polynomiell beschränkt*, falls gilt*

$$\exists \alpha \in \mathbb{N} : t_A(n) = O(n^\alpha) \ .$$

Einen Algorithmus mit polynomiell beschränkter worst case Laufzeit bezeichnen wir als Polynomialzeitalgorithmus*.*

Bereits im ersten Teil der Vorlesung wurde gezeigt, dass Turingmaschine (TM) und RAM sich gegenseitig mit polynomiellem Zeitverlust simulieren können. Wenn ein Problem also einen Algorithmus mit polynomiell beschränkter Laufzeit auf der RAM hat, so kann es auch mit einer polynomiellen Anzahl Schritten auf einer TM gelöst werden, und umgekehrt. Dies gilt nicht nur für die RAM und die TM, sondern auch für zahlreiche andere sinnvolle Maschinenmodelle, wie z.B. die Mehrband-TM. Die Klasse der Probleme, die in polynomieller Zeit gelöst werden können, ist also in einem gewissen Sinne *robust* gegenüber der Wahl des Maschinenmodells.

Definition 3.3 (Komplexitätsklasse P) *P ist die Klasse der Probleme, für die es einen Polynomialzeitalgorithmus gibt.*

Polynomialzeitalgorithmen werden häufig auch als „effiziente Algorithmen“ bezeichnet. Die Gleichsetzung von Effizienz mit polynomieller Rechenzeit ist sicherlich stark vereinfachend, aber als erste Näherung für den Begriff der Effizienz ganz gut geeignet. P ist in diesem Sinne die Klasse derjenigen Probleme, die effizient gelöst werden können.

3.1.2 Beispiele für Probleme in P

Von den folgenden Problem wissen wir, dass sie in P sind, weil wir einen Polynomialzeitalgorithmus für sie kennen.

- Sortieren
- Kürzeste Wege
- Minimaler Spannbaum
- Graphzusammenhang
- Maximaler Fluss
- Maximum Matching
- Lineare Programmierung
- Größter Gemeinsamer Teiler
- Primzahltest

Natürlich ist diese Liste nicht vollständig. Zur Illustration betrachten wir zwei dieser Probleme etwas genauer.

Problem 3.4 (Sortieren)

Eingabe: N (binär kodierte) Zahlen $a_1, \ldots, a_N \in \mathbb{N}$

Ausgabe: aufsteigend sortierte Folge der Eingabezahlen

Satz 3.5 *Sortieren* $\in$ P.

Beweis: Sei n die Eingabelänge, d.h. die Anzahl Bits die zur Kodierung der N Eingabezahlen $a_1, \ldots, a_N$ benötigt wird. Im uniformen Kostenmaß können wir in Zeit $O(N \log N)$ sortieren, z.B. durch Mergesort. Wir müssen zeigen, dass die Laufzeit von Mergesort auch bezüglich des logarithmischen Kostenmaßes polynomiell beschränkt ist.

- Jeder uniforme Schritt von Mergesort kann in Zeit $O(\ell)$ durchgeführt werden, wobei $\ell = \max_{1 \leq i \leq N} \log(a_i)$.
- Die Laufzeit von Mergesort ist somit $O(\ell \cdot N \log N)$.
- Aus $\ell \leq n$ und $N \leq n$, folgt $\ell \cdot N \log N \leq n^2 \log n \leq n^3$.

Damit ist die worst case Laufzeit von Mergesort polynomiell beschränkt, nämlich durch $O(n^3)$. □

Wenn man genauer hinschaut, kann man auch eine Schranke von $O(n \log n)$ für die Laufzeit von Mergesort im logarithmischen Kostenmaß herleiten, aber das ist für uns hier unerheblich.

Das zweite Problem, dass wir genauer betrachten ist ein Entscheidungsproblem.

Problem 3.6 (Graphzusammenhang)

Eingabe: Graph $G = (V, E)$

Frage: Ist G zusammenhängend?

Bei Graphproblemen gehen wir der Einfachheithalber grundsätzlich davon aus, dass der Graph in Form einer Adjazenzmatrix eingegeben wird.

Satz 3.7 *Graphzusammenhang* $\in \mathsf{P}$.

Beweis:. Die Eingabelänge ist $n = |V|^2 \geq |E|$. *Graphzusammenhang* löst man beispielsweise mit Hilfe einer Tiefensuche.

- Die Tiefensuche benötigt $O(|V| + |E|)$ uniforme Rechenschritte.
- Die Kosten für jeden Rechenschritt sind durch $O(\log |V|)$ (z.B. für den Zugriff auf Knotenindizes) beschränkt.
- Die Gesamtlaufzeit ist somit $O((|V| + |E|) \log |V|) = O(n \log n)$.

Somit ist die worst case Laufzeit der Tiefensuche polynomiell beschränkt, nämlich durch $O(n^2)$. □

3.2 Die Komplexitätsklasse NP

3.2.1 Definition und Erläuterung

Um nachzuweisen, dass ein Problem in P enthalten ist, genügt es einen Polynomialzeitalgorithmus für das Problem anzugeben. Wie kann man aber nachweisen, dass ein Problem nicht in P enthalten ist? – Diese Frage wird uns noch ein wenig beschäftigen. Dazu machen wir einen vielleicht zunächst etwas obskur erscheinenden Umweg über nichtdeterministische Turingmaschinen.

Definition 3.8 (Nichtdeterministische Turingmaschine – NTM) *Eine nichtdeterministische Turingmaschine (NTM) ist definiert wie eine deterministische Turingmaschine (TM), nur die Zustandsüberführungsfunktion wird zu einer Relation*

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

Eine Konfiguration K' ist direkter Nachfolger einer Konfiguration K, geschrieben $K \vdash K'$, falls K' durch einen der in δ beschriebenen Übergänge aus K hervorgeht. Das bedeutet zu einer Konfiguration kann es mehrere Nachfolgekonfigurationen geben. Jede mögliche Konfigurationsfolge, die mit der Startkonfiguration beginnt und jeweils mit einer der direkten Nachfolgekonfigurationen fortgesetzt wird bis sie eine Endkonfiguration im Zustand $\bar{q}$ erreicht, wird als Rechenweg der NTM bezeichnet. Der Verlauf der Rechnung ist also nicht eindeutig bestimmt.

Definition 3.9 (Akzeptanzverhalten der NTM) *Eine NTM M akzeptiert die Eingabe $x \in \Sigma^*$, falls es mindestens einen Rechenweg von M gibt, der in eine akzeptierende Endkonfiguration führt. Die von M erkannte Sprache $L(M)$ besteht aus allen von M akzeptierten Wörtern.*

Bei der Festlegung der Laufzeitkosten der NTM M berücksichtigen wir nur die Eingaben $x \in L(M)$. Für jedes $x \in L(M)$ gibt es mindestens einen Rechenweg auf dem M zu einem akzeptierenden Endzustand gelangt. Wir stellen uns vor, M wählt den kürzesten dieser Rechenwege. In anderen Worten, die NTM hat die magische Fähigkeit immer den kürzesten Rechenweg zu raten. Alternativ können wir uns auch vorstellen, die NTM kann allen Rechenwegen gleichzeitig folgen und stoppt, sobald einer der Wege erfolgreich ist.

Definition 3.10 (Laufzeit der NTM) *Sei M eine NTM. Die Laufzeit von M auf einer Eingabe $x \in L(M)$ ist definiert als*

$$T_M(x) := \textit{Länge des kürzesten akzeptierenden Rechenweges von } M \textit{ auf } x \ .$$

Für $x \notin L(M)$ definieren wir $T_M(x) = 0$. Die worst case Laufzeit $t_M(n)$ für M auf Eingaben der Länge $n \in \mathbb{N}$ ist definiert durch

$$t_M(n) := \max\{T_M(x) \mid x \in \Sigma^n\} \ .$$

Wir behaupten nicht, dass man eine derartige Maschine bauen kann. Wir benutzen die NTM nur als Modell für die Klassifikation der Komplexität von Problemen.

Definition 3.11 (Komplexitätsklasse NP) NP *ist die Klasse der Entscheidungsprobleme, die durch eine NTM M erkannt werden, deren worst case Laufzeit $t_M(n)$ polynomiell beschränkt ist.*

NP steht dabei für *nichtdeterministisch polynomiell*. Wir geben ein erstes Beispiel für ein Problem in NP.

Problem 3.12 (Cliquenproblem – CLIQUE)

Eingabe: Graph $G = (V, E)$, $k \in \{1, \ldots, |V|\}$

Frage: Gibt es eine k-Clique?

Eine *Clique* ist dabei eine Knotenteilmenge $C \subseteq V$, die die Bedingung erfüllt, dass jeder Knoten aus C mit jedem anderen Knoten aus C durch eine Kante verbunden ist. Eine *k-Clique* ist eine Clique der Größe k. Der Graph $G = (V, E)$ wird durch seine Adjazenzmatrix beschrieben.

Satz 3.13 *CLIQUE* $\in$ NP.

Beweis: Wir beschreiben eine NTM M mit $L(M) =$ CLIQUE. M arbeitet wie folgt.

1. Falls die Eingabe nicht von der Form (G, k) ist, so verwirft M die Eingabe. Sonst fährt M wie folgt fort.

2. Sei $G = (V, E)$. Sei N die Anzahl der Knoten. O.B.d.A. $V = \{1, \ldots, N\}$. M schreibt hinter die Eingabe den String $\#^N$. Der Kopf bewegt sich unter das erste $\#$.

3. M läuft von links nach rechts über den String $\#^N$ und ersetzt ihn nichtdeterministisch durch einen String aus $\{0,1\}^N$. Diesen String nennen wir $y = (y_1, \ldots, y_N)$.

4. Sei $C = \{i \in V \mid y_i = 1\} \subseteq V$. M akzeptiert, falls C eine k-Clique ist.

Die Anweisungen 1, 2 und 4 sind deterministisch. Diese Anweisungen können offensichtlich auch von einer (deterministischen) TM in einer polynomiell beschränkten Anzahl von Schritten ausgeführt werden. Nur in der dritten Anweisung greifen wir auf Nichtdeterminismus zurück: M „rät“ einen String $y \in \{0,1\}^N$ in N nicht deterministischen Schritten.

Wir zeigen $L(M) =$ CLIQUE. Zunächst nehmen wir an, die Eingabe ist von der Form (G, k) und G enthält eine k-Clique. In diesem Fall gibt es mindestens einen String y, der dazu führt, dass die Eingabe in Anweisung 4 akzeptiert wird. Falls die Eingabe also in CLIQUE enthalten ist, so gibt es einen akzeptierenden Rechenweg polynomiell beschränkter Länge. Falls die Eingabe hingegen nicht in CLIQUE enthalten ist, so gibt es keinen akzeptierenden Rechenweg.

Fazit: M erkennt die Sprache CLIQUE (vgl. dazu Def 3.9) und hat eine polynomiell beschränkte worst case Laufzeit (vgl. dazu Def 3.10). Also ist CLIQUE $\in$ NP (vgl. dazu Def 3.11). □

Bei der Beschreibung der NTM für das Cliquenproblem haben wir den Nichtdeterminismus nur dazu verwendet einen String y zu raten, der eine Knotenteilmenge C beschreibt, für die man ohne weiteres überprüfen kann, ob sie eine k-Clique ist. Wir können y als ein *Zertifikat* ansehen, so dass wir mit diesem Zertifikat die Zugehörigkeit zu CLIQUE in polynomieller Zeit überprüfen können. Die Schwierigkeit des Cliquenproblems scheint also nur darin zu liegen ein geeignetes Zertifikat zu finden, dass die Zugehörigkeit zur Sprache CLIQUE belegt. Diese Erkenntnis verallgemeinern wir nun auf alle Probleme in NP.

3.2.2 Alternative Charakterisierung der Klasse NP

Die Klasse NP kann beschrieben werden, ohne auf nichtdeterministische Turingmaschinen zurückzugreifen: Eine Sprache L gehört genau dann zu NP, wenn es für die Eingaben aus L ein Zertifikat polynomieller Länge gibt, mit dem sich die Zugehörigkeit zu L in polynomieller Zeit verifizieren lässt. Diese umgangssprachliche Charakterisierung von NP wird durch den folgenden Satz präzisiert. Dabei übernimmt y die Rolle des *Zertifikats*.

Satz 3.14 *Eine Sprache $L \subseteq \Sigma^*$ ist genau dann in* NP*, wenn es einen Polynomialzeitalgorithmus V (einen sogenannten* Verifizierer*) und ein Polynom p mit der folgenden Eigenschaft gibt:*

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

Beweis: Sei $L \in$ NP. Sei M eine NTM, die L in polynomieller Zeit erkennt. Die Laufzeit von M sei nach oben beschränkt durch ein Polynom q. Wir konstruieren nun einen Verifizierer V mit den gewünschten Eigenschaften. Zunächst vereinfachen wir, und nehmen an, dass die Überführungsrelation δ immer genau zwei mögliche Übergänge vorsieht, die wir mit 0 und 1 bezeichnen. Bei einer Eingabe $x \in \Sigma^n$, verwenden wir ein Zertifikat $y \in \{0,1\}^{q(n)}$, das den Weg durch die Rechnung von M weist: Der Verifizierer V erhält als Eingabe $y\#x$ und simuliert einen Rechenweg der NTM M für die Eingabe x, wobei er im i-ten Rechenschritt von M den Übergang y_i wählt. Es gilt

$$x \in L \;\Leftrightarrow\; M \text{ akzeptiert } x \;\Leftrightarrow\; \exists y \in \{0,1\}^{q(n)} : V \text{ akzeptiert } y\#x.$$

Die Laufzeit des Verifizierers V ist polynomiell beschränkt. Somit erfüllt V die im Satz geforderten Eigenschaften.

Sollte die Überführungsrelation δ Verzweigungen mit einem größeren Grad als zwei vorsehen, so fassen wir mehrere Bits zusammen um einen Ast der Verzweigung zu selektieren: Bei einem maximalen Verzweigungsgrad von k verwenden wir $\lceil \log k \rceil$ Bits für jede Verzweigung. Falls ein nicht vorhandener Ast selektiert wurde, weil der Grad einzelner Verzweigungen kleiner als k ist, so wird die Eingabe verworfen. Wir beobachten, dass der maximale Verzweigungsgrad k nicht größer als $3 \cdot |Q| \cdot |\Gamma|$ sein kann. Somit ist k eine Konstante. Deshalb sind weiterhin sowohl die Länge des Zertifikates y als auch die Laufzeit von V polynomiell beschränkt.

Nun zeigen wir die Umkehrrichtung. Dazu nehmen wir an, es liegt ein Verifizierer V für L und ein Polynom p vor, wie sie im Satz beschrieben sind. Es ist zu zeigen: $L \in$ NP. Wir konstruieren eine NTM M, die L in polynomieller Zeit erkennt. M arbeitet folgendermaßen:

- M rät ein Zertifikat $y \in \{0,1\}^*, |y| \leq p(n)$.
- M führt V auf $y\#x$ aus und akzeptiert, falls V akzeptiert.

Die Laufzeit von M ist polynomiell beschränkt, und M erkennt L, denn

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

weil M die Eingabe x genau dann akzeptiert, wenn mindestens einer der möglichen Rechenwege einen akzeptierenden Endzustand erreicht. □

3.2.3 Beispiele für Probleme in NP

Typische Vertreter für Probleme aus der Klasse NP sind Entscheidungsvarianten von Optimierungsproblemen. Wir beschreiben Optimierungsprobleme indem wir *Eingaben*, *zulässige Lösungen* und *Zielfunktion* angeben. Die Ausgabe, die bei einem Optimierungsproblem berechnet werden soll, ist eine zulässige Lösung, die die Zielfunktion optimiert. Wenn es mehrere optimale Lösungen gibt, so soll genau eine von diesen Lösungen ausgegeben werden.

Problem 3.15 (Rucksackproblem, Knapsack Problem – KP) *Beim KP suchen wir eine Teilmenge K von N gegebenen Objekten mit Gewichten $w_1, \ldots, w_N$ und Nutzenwerten bzw. Profiten $p_1, \ldots, p_N$, so dass die Objekte aus K in einen Rucksack mit Gewichtsschranke b passen und dabei der Nutzen maximiert wird.*

Eingabe: $b \in \mathbb{N}$, $w_1, \ldots, w_N \in \{1, \ldots, b\}$, $p_1, \ldots, p_N \in \mathbb{N}$

zulässige Lösungen: $K \subseteq \{1, \ldots, N\}$, *so dass* $\sum_{i \in K} w_i \leq b$

Zielfunktion: Maximiere $\sum_{i \in K} p_i$

Zu einem Optimierungsproblem können wir grundsätzlich auch immer eine Entscheidungsvariante angegeben, in der wir eine Zielvorgabe machen und fragen, ob diese Vorgabe erreicht werden kann. Bei der *Entscheidungsvariante vom* KP ist zur oben beschriebenen Eingabe zusätzlich eine Zahl $p \in \mathbb{N}$ gegeben. Es soll entschieden werden, ob es eine zulässige Teilmenge der Objekte mit Nutzen mindestens p gibt.

Problem 3.16 (Bin Packing Problem – BPP) *Beim BPP suchen wir eine Verteilung von N Objekten mit Gewichten $w_1, \ldots, w_N$ auf eine möglichst kleine Anzahl von Behältern mit Gewichtskapazität jeweils b. Die Anzahl der Behälter wird mit k bezeichnet.*

Eingabe: $b \in \mathbb{N}$, $w_1, \ldots, w_N \in \{1, \ldots, b\}$

zulässige Lösungen: $k \in \mathbb{N}$ *und Funktion* $f : \{1, \ldots, N\} \to \{1, \ldots, k\}$*,*
so dass

$$\forall i \in \{1, \ldots, k\} : \sum_{j \in f^{-1}(i)} w_j \leq b$$

Zielfunktion: Minimiere k

Bei der *Entscheidungsvariante vom BPP* ist $k \in \mathbb{N}$ Teil der Eingabe. Es ist zu entscheiden, ob es eine zulässige Verteilung der Objekte auf höchstens k Behälter gibt.

Problem 3.17 (Traveling Salesperson Problem – TSP) *Beim TSP ist ein vollständiger Graph aus* N *Knoten (Orten) mit Kantengewichten gegeben. Gesucht ist eine Rundreise (ein* Hamiltonkreis*, eine* Tour*) mit kleinstmöglichen Kosten.*

Eingabe: $c(i, j) \in \mathbb{N}$ *für* $i, j \in \{1, \ldots, N\}$*, wobei gilt* $c(j, i) = c(i, j)$

zulässige Lösungen: Permutationen π *auf* $\{1, \ldots, N\}$

Zielfunktion: Minimiere $\sum_{i=1}^{N-1} c(\pi(i), \pi(i+1)) + c(\pi(N), \pi(1))$

Bei der *Entscheidungsvariante vom TSP* ist zusätzlich eine Zahl $b \in \mathbb{N}$ gegeben und es ist zu entscheiden, ob es eine Rundreise mit Kosten höchstens b gibt.

Satz 3.18 *Die Entscheidungsvarianten von KP, BPP und TSP sind in* NP.

Beweis: Wir verwenden zulässige Lösungen dieser Probleme als Zertifikat. Dazu müssen wir nachweisen, dass diese Lösungen eine polynomiell in der Eingabelänge beschränkte Kodierungslänge haben und durch einen Algorithmus, dessen Laufzeit ebenfalls polynomiell in der Eingabelänge des Problems beschränkt ist, verifiziert werden können.

- KP: Die Teilmenge K der ausgewählten Objekte kann mit N Bits kodiert werden, und es kann in polynomieller Zeit überprüft werden, ob die Gewichtsschranke b des Rucksacks eingehalten wird und zusätzlich der geforderte Nutzenwert p erreicht wird.

- BPP: Die Abbildung $f : \{1, \ldots, N\} \to \{1, \ldots, k\}$, die die Verteilung der Objekte auf die Behälter beschreibt, kann mit $O(N \log k)$ Bits kodiert werden. Die Kodierungslänge der Lösungen ist somit polynomiell in der Eingabelänge beschränkt. Es kann in polynomieller Zeit überprüft werden, ob die vorgegebene Anzahl Behälter und die Gewichtsschranke je Behälter eingehalten werden.

- TSP: Für die Kodierung einer Permutation π werden $O(N \log N)$ Bits benötigt, und es kann in polynomieller Zeit überprüft werden, ob die durch π beschriebene Rundreise die vorgegebene Kostenschranke b einhält.

Somit sind die Entscheidungsvarianten von KP, BPP und TSP ebenfalls in der Klasse NP enthalten. □

Mit Hilfe eines Algorithmus, der ein Optimierungsproblem löst, kann man offensichtlich auch die Entscheidungsvariante lösen. Häufig funktioniert auch der umgekehrte Weg: Aus einem Polynomialzeitalgorithmus für die Entscheidungsvariante können wir einen Polynomialzeitalgorithmus für die Optimierungsvariante konstruieren. Optimierungs- und Entscheidungsvariante haben also typischerweise dieselbe Komplexität modulo polynomieller Faktoren. Wir demonstrieren dies für das Rucksackproblem.

Satz 3.19 *Wenn die Entscheidungsvariante von KP in polynomieller Zeit lösbar ist, dann auch die Optimierungsvariante.*

Beweis: Neben der Entscheidungs- und Optimierungsvariante betrachten wir die folgende *Zwischenvariante:* Gesucht ist der Zielfunktionswert einer optimalen Lösung, ohne dass die optimale Lösung selber ausgegeben werden muss.

Aus einem Algorithmus A für die Entscheidungsvariante, konstruieren wir zunächst einen Algorithmus B für die Zwischenvariante. Algorithmus B berechnet den maximal möglichen Profit p, indem er eine Binärsuche über dem Wertebereich $\{0, \ldots, P\}$ ausführt, wobei $P = \sum_{i=1}^{N} p_i$ eine triviale obere Schranke für p ist. In dieser Binärsuche werden die Entscheidungen über diejenige „Hälfte“ des Wertebereichs auf der die Suche fortgesetzt wird mit Hilfe von Algorithmus A getroffen. Die Binärsuche hat einen Wertebereich der Größe $P + 1$. Falls $P + 1$ eine Zweierpotenz ist, so benötigt die Suche genau $\log(P + 1)$ Iterationen, da sich der Wertebereich von Iteration zu Iteration halbiert. Ist der Wertbereich keine Zweierpotenz, so gilt eine obere Schranke von $\lceil \log(P + 1) \rceil$ für die Anzahl der Aufrufe von Algorithmus A.

Diese Laufzeitschranke müssen wir in Beziehung zur Eingabelänge setzen. Wir nehmen an, alle Eingabezahlen werden binär kodiert. Die Kodierungslänge einer natürlichen Zahl $a \in \mathbb{N}$ bezeichnen wir mit $\kappa(a)$. Es gilt $\kappa(a) = \lceil \log(a + 1) \rceil$. Die Binärdarstellung der Summe zweier Zahlen ist kürzer als die Summe der

Algorithmus C

1) $K := \{1, \ldots, N\}$;

2) $p^* := B(K)$;

3) **for** i := 1 **to** N **do**

   if $B(K \setminus \{i\}) = p^*$ **then** $K := K - \{i\}$;

4) **Ausgabe** K.

Abbildung 3.1: Pseudocode für einen Algorithmus C für KP, der eine optimale Rucksackbepackung berechnet und zwar mit Hilfe eines Unterprogramms B, das lediglich den Wert einer optimalen Lösung zurückgibt. Der Wert $p^* = B(K)$ entspricht dem optimalen Lösungswert.

Längen ihrer Binärdarstellungen, d.h. für $a, b \in \mathbb{N}$ gilt $\kappa(a+b) \leq \kappa(a) + \kappa(b)$. Die Eingabelänge ist also mindestens

$$\sum_{i=1}^{N} \kappa(p_i) \;\geq\; \kappa\left(\sum_{i=1}^{N} p_i\right) \;=\; \kappa(P) \;=\; \lceil \log(P+1) \rceil \;,$$

wobei wir beobachten, dass der letzte Term genau der oberen Schranke für die Anzahl Iterationen der Binärsuche entspricht. Die Anzahl der Aufrufe von Algorithmus A ist somit linear in der Eingabelänge beschränkt. Falls also die Laufzeit von A polynomiell beschränkt ist, so ist auch B ein Polynomialzeitalgorithmus.

Als nächstes zeigen wir, wie aus einem Algorithmus B für die Zwischenvariante, ein Algorithmus C für die Optimierungsvariante konstruiert werden kann. Algorithmus C wird durch Angabe des Pseudocodes in Abbildung 3.1 beschrieben. Der Algorithmus betrachtet die Objekte nacheinander. Für jedes Objekt $i \in \{1, \ldots, N\}$ testet er mit Hilfe von Algorithmus B, ob es auch eine optimale Lösung ohne Objekt i gibt. Falls Objekt i nicht für die optimale Lösung benötigt wird, so wird es gestrichen. Auf diese Art und Weise bleibt am Ende eine Menge von Objekten übrig, die zulässig ist, also die Gewichtsschranke einhält, und den optimalen Zielfunktionswert erreicht.

Die Laufzeit von Algorithmus C wird im Wesentlichen durch die $N + 1$ Unterprogrammaufrufe von Algorithmus B bestimmt. Falls also die Laufzeit von B polynomiell beschränkt ist, so ist auch C ein Polynomialzeitalgorithmus. □

3.3 P versus NP

Das wohl bekannteste offene Problem in der Informatik ist die Frage

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

Genau genommen macht die Frage in dieser Form wenig Sinn, denn wir haben P als Menge von Problemen (z.B. Sortieren) und NP als Menge von Entscheidungsproblemen (Sprachen) definiert. Wenn E die Menge aller Entscheidungsprobleme ist, so müsste die Frage eigentlich lauten

$$\mathsf{P} \cap E = \mathsf{NP}?$$

Zur Verwirrung aller, die in die Komplexitätstheorie einsteigen, wird aber die Bezeichnung P doppeldeutig verwendet, mal für die Klasse aller Probleme, die einen Polynomialzeitalgorithmus haben, und mal für die Klasse der Entscheidungsprobleme, die in polynomieller Zeit entschieden werden können. Wir treffen die folgende Vereinbarung: Immer wenn wir P mit NP oder anderen Klassen, die nur für Entscheidungsprobleme definiert sind, vergleichen, schränken wir P auf die Klasse der Entscheidungsprobleme, die in polynomieller Zeit entschieden werden können, ein.

Da eine deterministische TM auch als eine nichtdeterministische TM aufgefasst werden kann, die vom Nichtdeterminismus keinen Gebrauch macht, gilt (eingeschränkt auf Entscheidungsprobleme) offensichtlich

$$\mathsf{P} \subseteq \mathsf{NP}.$$

Die Klasse NP scheint wesentlich mächtiger als die Klasse P zu sein, da die NTM die „magische“Fähigkeit hat, den richtigen Rechenweg aus einer Vielzahl von Rechenwegen zu raten. Satz 3.14 beschreibt die Klasse NP in Form eines Polynomialzeitalgorithmus, der ein Zertifikat umsonst geschenkt bekommt und nur noch die „JA-Antworten“ verifizieren muss. Natürlich können wir alle möglichen Zertifikate ausprobieren, um den Nichtdeterminismus zu simulieren. Dieses naive Vorgehen zeigt zunächst einmal, dass die Probleme in NP zumindest rekursiv sind. Allerdings liefert dieser Ansatz nur eine exponentielle Laufzeitschranke.

Satz 3.20 *Für jedes Entscheidungsproblem $L \in \mathsf{NP}$ gibt es einen Algorithmus A, der L entscheidet, und dessen worst case Laufzeit durch $2^{q(n)}$ nach oben beschränkt ist, wobei q ein geeignetes Polynom ist.*

Beweis: Der Verifizierer V und das Polynom p seien definiert wie in Satz 3.14. Um die Eingabe $x \in \{0,1\}^n$ zu entscheiden, generiert Algorithmus A nacheinander alle möglichen Zertifikate $y \in \{0,1\}^{p(n)}$ und startet den Verifizierer V mit der Eingabe $y\#x$. Falls V eines der generierten Zertifikate akzeptiert, so akzeptiert auch A. Falls V keines der Zertifikate akzeptiert, so verwirft A.

Die Korrektheit von A folgt direkt aus Satz 3.14. Wir analysieren nun die Laufzeit von A. Sei p' eine polynomielle Laufzeitschranke für den Verifizierer V. Wir definieren das Polynom $q(n) = p(n) + p'(p(n) + 1 + n)$. Die Laufzeit von Algorithmus A ist nach oben beschränkt durch

$$2^{p(n)} \cdot p'(p(n) + 1 + n) \;\leq\; 2^{p(n)} \cdot 2^{p'(p(n)+1+n)} \;=\; 2^{q(n)} \;.$$

□

Mit der Charakterisierung aus Satz 3.14 im Hinterkopf können wir die Frage $\mathsf{P} = \mathsf{NP}?$ auch folgendermaßen interpretieren: Ist das Herleiten eines Beweises (Zertifikates) wesentlich schwieriger als das Nachvollziehen eines gegebenen Beweises? - Möglicherweise hat das Herleiten exponentielle Komplexität während das Nachvollziehen in polynomieller Zeit möglich ist. Die Meisten, die sich mit dieser Fragestellung auseinander gesetzt haben, glauben wohl daran, dass es tatsächlich schwieriger ist einen Beweis herzuleiten als ihn nur nachzuvollziehen, was zu der Hypothese $\mathsf{P} \neq \mathsf{NP}$ führt. Solange keiner das Gegenteil beweist, arbeiten wir mit dieser Hypothese. Die Bedeutung dieser Hypothese für die Algorithmik können wir an dieser Stelle aber noch nicht wirklich ermessen. Dazu benötigen wir noch mehr Hintergrundwissen, welches wir uns im Folgenden aneignen werden.

3.4 NP-Vollständigkeit

Die Klasse NP enthält viele Probleme, für die es trotz intensivster Bemühungen bis heute nicht gelungen ist, einen Polynomialzeitalgorithmus zu entwickeln. Insbesondere in der Optimierung treten immer wieder scheinbar harmlose Probleme mit Anwendungen zum Beispiel aus den Bereichen Produktion, Logistik und Ökonomie auf, die auch tatsächlich einfach zu lösen wären, hätten wir einen Rechner, der mit so etwas wie „Intuition“ ausgestattet wäre, um den richtigen Rechenweg aus einer Vielzahl von Rechenwegen zu „erraten“. Leider gibt es aber derartige Rechner nur in theoretischen Modellen, wie etwa dem Modell der nicht deterministischen Turingmaschine. Derartig verschrobene theoretische Modelle

können uns bei der praktischen Arbeit aber wohl kaum behilflich sein, oder vielleicht doch?

TSP ist ein Beispiel für eines derjenigen Probleme, die auf den ersten Blick recht harmlos erscheinen, über die sich aber schon viele Algorithmenentwickler vergeblich den Kopf zerbrochen haben. Wie ist das möglich, wo wir doch effiziente Algorithmen für scheinbar sehr ähnliche Problemstellungen wie etwa dem Kürzeste-Wege-Problem oder dem Minimalen-Spannbaum-Problem kennen? – Wenn uns jemand den Auftrag erteilen würde, einen effizienten Algorithmus für TSP zu entwerfen, würde er womöglich damit rechnen, dass wir ihm nach einer Woche die fertige Implementierung vorlegen. Stattdessen kommen wir voraussichtlich mit der enttäuschenden Nachricht zurück, dass wir die Aufgabe nicht erfüllen konnten. Hoffentlich haben wir keinen Vertrag unterschrieben, der hohe Konventionalstrafen für den Fall unseres Versagens vorsieht. Allemal sehen wir schlecht gegenüber unserem Mitbewerber aus, der die Implementierung für das Kürzeste-Wege-Problem in Null-Komma-Nichts vorgelegt hat. Das einzige was uns jetzt noch retten kann, ist eine wirklich stichhaltige Begründung, warum die Berechnung einer kürzesten Rundreise tatsächlich wesentlich schwieriger ist als etwa die Berechnung kürzester Wege.

Die NP-Vollständigkeitstheorie liefert eine derartige Begründung. Sie erlaubt es uns Probleme bezüglich ihrer Schwierigkeit miteinander zu vergleichen. Wir werden zeigen, dass beispielsweise TSP zu den schwierigsten Problemen in NP gehört. Unter der Hypothese $\mathsf{P} \neq \mathsf{NP}$ hat TSP deshalb beweisbar keinen Polynomialzeitalgorithmus. Wir werden sehen, dass TSP keine Ausnahme ist, sondern diese Eigenschaft mit zahlreichen anderen Problemen aus NP teilt, den sogenannten „NP-vollständigen“ Problemen.

Wir beginnen nun mit den grundlegenden Definitionen zu dieser Theorie.

3.4.1 Polynomielle Reduktionen

Wir vergleichen die Schwierigkeit von Entscheidungsproblemen mittels polynomieller Reduktion.

Definition 3.21 (Polynomielle Reduktion) *L_1 und L_2 seien zwei Sprachen über Σ_1 bzw. Σ_2. L_1 ist* polynomiell reduzierbar *auf L_2, wenn es eine Reduktion von L_1 nach L_2 gibt, die in polynomieller Zeit berechenbar ist. Wir schreiben $L_1 \leq_p L_2$.*

Zur Erinnerung: Eine Reduktion $L_1 \leq L_2$ ist definiert durch eine berechenbare Funktion $f : \Sigma_1^* \to \Sigma_2^*$ mit der Eigenschaft, dass für alle $x \in \Sigma_1^*$ gilt

$x \in L_1 \Leftrightarrow f(x) \in L_2$. Bei der Polynomialzeitreduktion $L_1 \leq_p L_2$ verlangen wir also lediglich zusätzlich, dass f in polynomieller Zeit berechnet werden kann.

Falls es für das Problem L_2 einen Polynomialzeitalgorithmus A gibt und ferner eine Funktion f existiert, mit der das Problem L_1 polynomiell auf L_2 reduziert werden kann, so können wir aus diesen Komponenten auch einen Algorithmus B für L_1 konstruieren: B berechnet aus der Eingabe x für L_1 die Eingabe $f(x)$ für L_2, startet A auf $f(x)$ und übernimmt das Akzeptanzverhalten von A. Wir behaupten, dass auch B ein Polynomialzeitalgorithmus ist. Diese Behauptung impliziert das folgende Lemma.

Lemma 3.22 $L_1 \leq_p L_2$, $L_2 \in P \Rightarrow L_1 \in P$.

Beweis: Die Korrektheit des oben beschriebenen Algorithmus B folgt direkt aus der Eigenschaft $x \in L_1 \Leftrightarrow f(x) \in L_2$. Wir müssen nur noch die polynomielle Laufzeit von Algorithmus B nachweisen.

Sei p ein Polynom, dass die Laufzeit für die Berechnung von f beschränkt. Beachte, es gilt $|f(x)| \leq p(|x|) + |x|$, da das Schreiben jedes Ausgabebits, das nicht blind aus der Eingabe übernommen wird, mindestens eine Zeiteinheit kostet. Die Laufzeit von A ist durch ein Polynom in der Länge von $f(x)$ beschränkt. Dieses Polynom bezeichnen wir mit q. O.B.d.A. seien alle Koeffizienten von p und q nicht negativ. Wir erhalten somit die folgende Laufzeitschranke für Algorithmus B:

$$p(|x|) + q(|f(x)|) \;\leq\; p(|x|) + q(p(|x|) + |x|) \;\;.$$

Die Laufzeit von B ist also polynomiell in die Eingabelänge beschränkt. □

Intuitiv bedeutet $L_1 \leq_p L_2$, dass L_1 „nicht schwieriger als“ L_2 ist – modulo einer polynomiellen Eingabetransformation.

3.4.2 Beispiel einer polynomiellen Reduktion

Die eigentliche Stärke des Reduktionsprinzips ist es, dass man Probleme unterschiedlichster Art aufeinander reduzieren kann. Wir präsentieren ein Beispiel in dem ein Graphproblem auf ein Problem der Aussagenlogik reduziert wird.

Problem 3.23 (Knotenfärbung (COLORING))

Eingabe: Graph $G = (V, E)$, *Zahl* $k \in \{1, \ldots, |V|\}$

Frage: Gibt es eine Färbung $c : V \to \{1, \ldots, k\}$ der Knoten von G mit k Farben, so dass benachbarte Knoten verschiedene Farben haben, d.h. $\forall\{u, v\} \in E : c(u) \neq c(v)$?

Problem 3.24 (Erfüllbarkeitsproblem (Satisfiability – SAT))

Eingabe: Aussagenlogische Formel ϕ in KNF

Frage: Gibt es eine erfüllende Belegung für ϕ?

Eine aussagenlogische Formel in *konjunktiver Normalform (KNF)* besteht aus $\wedge$-verknüpften Klauseln, die wiederum aus $\vee$-verknüpften Literalen bestehen. Ein Literal entspricht dabei einer Variablen x_i oder ihrer Negierung $\bar{x}_i$. Wir geben ein Beispiel für eine derartige Formel mit vier Klauseln.

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

Eine *Belegung* weist jeder Variablen einen Wert zu, entweder 1 für *wahr* oder 0 für *falsch*. Ein *unnegiertes Literal ist erfüllt*, wenn die zugehörige Variable den Wert 1 hat, und ein *negiertes Literal ist erfüllt*, wenn die zugehörige Variable den Wert 0 hat. Eine *Klausel ist erfüllt*, wenn sie mindestens ein erfülltes Literal enthält. Die *Formel ist erfüllt*, wenn alle ihre Klauseln erfüllt sind. Gefragt ist nach der Existenz einer Belegung, die die Formel erfüllt, einer sogenannten *erfüllenden Belegung*. Beispielsweise ist $x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 0$ eine erfüllende Belegung für die obige Formel.

Satz 3.25 *COLORING $\leq_p$ SAT.*

Beweis: Wir beschreiben eine polynomiell berechenbare Funktion f, die eine Eingabe (G, k) für das COLORING-Problem auf eine Formel ϕ für das SAT-Problem abbildet. Diese Abbildung soll die Eigenschaft

$$G \text{ hat eine } k\text{-Färbung } \Leftrightarrow \quad \phi \text{ ist erfüllbar}$$

haben. Wir definieren die Abbildung folgendermaßen: Die Formel ϕ hat für jede Knoten-Farb-Kombination (v, i) mit $v \in V$ und $i \in \{1, \ldots, k\}$ eine Variable x_v^i. Die Formel für (G, k) lautet

$$\phi = \bigwedge_{v \in V} \underbrace{(x_v^1 \vee x_v^2 \vee \ldots \vee x_v^k)}_{\text{Knotenbedingung}} \wedge \bigwedge_{\{u,v\} \in E} \bigwedge_{i \in \{1,\ldots,k\}} \underbrace{(\bar{x}_u^i \vee \bar{x}_v^i)}_{\text{Kantenbedingung}} .$$

Die $\bigwedge$-Quantoren sind dabei nur Akkürzungen, die wir im Beweis verwenden um eine kompakte Darstellung der Formel zu haben. Der erste Quantor $\bigwedge_{v \in V}$ steht beispielsweise dafür, dass die Formel ϕ die Knotenbedingung für jeden Knoten enthält. Die Anzahl der Literale in der Formel ist somit $O(k \cdot |V| + k \cdot |E|) = O(|V|^3)$. Die Länge der Formel ist deshalb polynomiell in der Größe der Eingabe des COLORING-Problems beschränkt und die Formel kann in polynomieller Zeit konstruiert werden.

Zum Nachweis der Korrektheit müssen wir zum einen zeigen, dass gilt

$$G \text{ hat eine } k\text{-Färbung } \Rightarrow \ \phi \text{ ist erfüllbar } .$$

Sei c eine k-Färbung für G. Für jeden Knoten v mit $c(v) = i$ setzen wir $x_v^i = 1$ und alle anderen Variablen auf 0. Dadurch ist die Knotenbedingung offensichtlich erfüllt. Die Kantenbedingung ist ebenfalls erfüllt, denn es gilt: Für jede Farbe i und jede Kante $\{u, v\}$ gilt $\bar{x}_u^i \vee \bar{x}_v^i$, denn sonst hätten u und v beide die Farbe i. Also erfüllt diese Belegung die Formel ϕ.

Zum anderen müssen wir zeigen, es gilt

$$\phi \text{ ist erfüllbar } \Rightarrow \ G \text{ hat eine } k\text{-Färbung } .$$

Dazu fixiere eine beliebige erfüllende Belegung für ϕ. Wegen der Knotenbedingung gibt es für jeden Knoten v mindestens eine Farbe mit $x_v^i = 1$. Für jeden Knoten wähle eine beliebige derartige Farbe aus. Sei $\{u, v\} \in E$. Wir behaupten $c(u) \neq c(v)$. Zum Widerspruch nehmen wir an, $c(u) = c(v) = i$. Dann wäre $x_u^i = x_v^i = 1$ und die Kantenbedingung $\bar{x}_u^i \vee \bar{x}_v^i$ wäre verletzt. $\square$

Die polynomielle Reduktion von COLORING auf SAT impliziert die folgenden beiden äquivalenten Aussagen.

Korollar 3.26

a) Wenn SAT einen Polynomialzeitalgorithmus hat, so hat auch COLORING einen Polynomialzeitalgorithmus.

b) Wenn COLORING keinen Polynomialzeitalgorithmus hat, so hat auch SAT keinen Polynomialzeitalgorithmus.

In dieser Vorlesung werden wir die polynomielle Reduktion in der Regel im Sinne von Aussage b) anwenden, d.h. wir verwenden polynomielle Reduktionen um zu belegen, dass ein Problem A keinen Polynomialzeitalgorithmus hat, wenn ein anderes Problem B ebenfalls keinen Polynomialzeitalgorithmus hat.

3.4.3 Definitionen von NP-Härte und NP-Vollständigkeit

Definition 3.27 (NP-Härte) *Ein Problem L heißt* NP-hart*, falls gilt*

$$\forall L' \in \mathsf{NP} : L' \leq_p L \ .$$

Satz 3.28 *L NP-hart, $L \in \mathsf{P} \ \Rightarrow \ \mathsf{P} = \mathsf{NP}$.*

Beweis: Sei L' ein beliebiges Problem aus NP. Da L NP-hart ist, gilt $L' \leq_p L$. Aus $L \in \mathsf{P}$ folgt somit gemäß Lemma 3.22, dass $L' \in P$ gilt. Also ist jedes Problem aus NP auch in P enthalten, so dass gilt $\mathsf{P} = \mathsf{NP}$. □

Unter der Hypothese $\mathsf{P} \neq \mathsf{NP}$ gibt es für NP-harte Probleme deshalb keinen Polynomialzeitalgorithmus.

Definition 3.29 (NP-Vollständigkeit) *Ein Problem L heißt* NP-vollständig*, falls gilt*

1. *$L \in$ NP und*
2. *L ist NP-hart.*

Intuitiv sind NP-vollständige Probleme somit die schwierigsten unter den Problemen in NP.

Definition 3.30 NPC *bezeichnet die Klasse der* NP*-vollständigen Probleme.*

Alle Probleme in NPC können als gleich schwierig – modulo polynomieller Eingabetransformationen – angesehen werden, da alle Probleme in NPC gegenseitig aufeinander polynomiell reduziert werden können.

3.5 Der Satz von Cook und Levin

Der erste NP-Vollständigkeitsbeweis wurde unabhängig von Stephen Cook (1971) und Leonid Levin (in einer Veröffentlichung von 1973) geführt. Cook und Levin zeigten durch eine sogenannte *Master-Reduktion*, dass alle Probleme aus NP in polynomieller Zeit auf das Erfüllbarkeitsproblem (SAT, vgl. Abschnitt 3.4.2) reduziert werden können.

Satz 3.31 *SAT ist* NP*-vollständig.*

Beweis: Eine aussagenlogische Formel ϕ kann bezüglich einer gegebenen Belegung in polynomieller Zeit ausgewertet werden. Somit können wir eine erfüllende Belegung als Zertifikat einsetzen, so dass gilt SAT $\in$ NP. Der schwierige Teil des Beweises liegt im Nachweis der NP-Härte. Dazu müssen wir in einer Master-Reduktion jedes Problem aus NP polynomiell auf SAT reduzieren.

Sei $L \subseteq \Sigma^*$ ein beliebiges Problem aus NP. Für jedes $x \in \Sigma^*$ müssen wir in polynomieller Zeit eine Formel ϕ konstruieren, die die folgende Eigenschaft erfüllt.

$$x \in L \;\Leftrightarrow\; \phi \in \text{SAT}$$

Über L haben wir dabei leider nur wenig Informationen zur Verfügung: Es handelt sich um ein Entscheidungsproblem, für das es eine NTM M gibt, die das Problem in polynomieller Zeit erkennt. Die Laufzeitschranke von M werde dabei durch ein Polynom p in der Eingabelänge beschrieben. Basierend auf diesen Informationen beschreiben wir nun einen Polynomialzeitalgorithmus, der für jedes $x \in \Sigma^*$ eine aussagenlogische Formel ϕ mit der folgenden Eigenschaft konstruiert.

$$M \text{ akzeptiert } x \;\Leftrightarrow\; \phi \text{ ist erfüllbar}$$

Die Zustandsmenge der NTM bezeichnen wir wie üblich mit Q, das Bandalphabet mit Γ und die Überführungsrelation mit δ. Um unsere Argumentation ein wenig zu vereinfachen, treffen wir o.B.d.A. die folgenden Vereinbarungen. Die NTM M bewege den Kopf niemals an eine Position links der Startposition (vgl. Übungsaufgabe 1.10). Die Zustandsmenge Q von M enthalte genau einen akzeptierenden Zustand, den wir mit q_{accept} bezeichnen. Wenn dieser Zustand erreicht wird, so terminiert M nicht, sondern verbleibt in einer Endlosschleife im Zustand q_{accept}.

Sei $x \in \Sigma^*$ eine beliebige Eingabe und $n = |x|$. Sei $K_0 = q_0 x$ die Startkonfiguration von M. Unsere Aufgabe ist die Konstruktion einer Formel ϕ, die genau dann erfüllbar ist, wenn es eine mögliche Konfigurationsfolge

$$K_0 \vdash K_1 \vdash \cdots \vdash K_{p(n)}$$

für M auf der Eingabe x gibt, bei der $K_{p(n)}$ eine Konfiguration im Zustand q_{accept} ist. Zu diesem Zweck definieren wir zunächst eine geeignete Menge von booleschen Variablen, die es uns ermöglicht, einzelne Konfigurationen und Konfigurationsübergänge in einer aussagenlogischen Formel zu kodieren.

Definition der Variablen: Wir unterscheiden drei Arten von Variablen, die den Zustand, die Kopfposition, und die Bandinschrift zu jedem Zeitpunkt innerhalb

der Laufzeitschranke beschreiben sollen. Beachte, dass die Laufzeitschranke für M auch eine Schranke für die maximal erreichbare Kopfposition darstellt, da sich der Kopf nur um eine Position je Schritt bewegen kann. Deshalb müssen wir nur Kopfpositionen und Bandinschriften von Index 0 bis Index $p(n)$ betrachten. Im Einzelnen verwenden wir die folgenden Variablen.

- $Q(t, k)$ für $t \in \{0, \ldots, p(n)\}$ und $k \in Q$
- $H(t, j)$ für $t, j \in \{0, \ldots, p(n)\}$
- $S(t, j, a)$ für $t, j \in \{0, \ldots, p(n)\}$ und $a \in \Gamma$

Die Belegung $Q(t, k) = 1$ soll besagen, dass sich die Rechnung zum Zeitpunkt t im Zustand k befindet; $H(t, j) = 1$ steht dafür, dass sich der Kopf zum Zeitpunkt t an Bandposition j befindet; und die Belegung $S(t, j, a) = 1$ bedeutet, dass zum Zeitpunkt t an Bandposition j das Zeichen a geschrieben steht. Aufgrund der Tatsache, dass wir nur an polynomiell vielen Rechenschritten und deshalb auch nur an polynomiell vielen Bandpositionen interessiert sind, benötigen wir auch nur eine polynomiell beschränkte Anzahl an Variablen.

Kodierung einzelner Konfigurationen: Für jedes $t \in \{0, \ldots, p(n)\}$ beschreiben wir nun eine Formel ϕ_t, die nur dann erfüllt ist, wenn die Belegung der auf den Zeitpunkt t eingeschränkten Variablenmenge $Q(t, k)$, $H(t, j)$, $S(t, j, a)$ eine korrekte Konfiguration beschreibt, d.h.

- es gibt genau einen Zustand $k \in Q$ mit $Q(t, k) = 1$,
- es gibt genau eine Bandposition $j \in \{0, \ldots, p(n)\}$ mit $H(t, j) = 1$, und
- für jedes $j \in \{0, \ldots, p(n)\}$ gibt es jeweils genau ein Zeichen $a \in \Gamma$ mit $S(t, j, a) = 1$.

Für eine beliebige Variablenmenge $\{y_1, \ldots, y_m\}$ besagt das Prädikat

$$(y_1 \vee \ldots \vee y_m) \ \wedge \bigwedge_{i \neq j} (\neg y_i \vee \neg y_j) \ ,$$

dass genau eine der Variablen y_i den Wert 1 annimmt und alle anderen Variablen den Wert 0. Die Länge dieses Prädikates ist $O(m^2)$, wobei wir mit der *Länge* die Anzahl der Literale im Prädikat meinen. Auf diese Art kodieren wir die obigen Anforderungen für den Zeitpunkt t. Es ergibt sich eine Teilformel in KNF der Länge $O(p(n)^2)$, die wir mit ϕ_t bezeichnen.

Kodierung von Konfigurationsübergängen: Wir betrachten nun nur noch Belegungen, die die Formeln $\phi_0, \ldots, \phi_{p(n)}$ erfüllen und somit Konfigurationen K_0, $\ldots, K_{p(n)}$ beschreiben, die allerdings nicht notwendigerweise einer möglichen Konfigurationsfolge der NTM M entsprechen. Als nächstes konstruieren wir eine Formel ϕ'_t für $1 \leq t \leq p(n)$, die nur durch solche Belegungen erfüllt wird, bei denen K_t eine direkte Nachfolgekonfiguration von K_{t-1} ist.

Zunächst beschreiben wir lediglich eine Teilformel, welche festlegt, dass die Bandinschrift von K_t an allen Positionen außer der Kopfposition (zum Zeitpunkt $t-1$) mit der Inschrift von K_{t-1} übereinstimmt, da sich beim Konfigurationsübergang die Bandinschrift ja nur an der Kopfposition verändern darf.

$$\bigwedge_{i=0}^{p(n)} \bigwedge_{z \in \Gamma} ((S(t-1,i,z) \wedge \neg H(t-1,i)) \Rightarrow S(t,i,z))$$

Diese Teilformel erklärt sich von selbst, wenn man sie von links nach rechts liest: „Für alle Bandpositionen i und alle Zeichen z gelte, falls zum Zeitpunkt $t-1$ das Band an Position i das Zeichen z enthält und die Kopfposition nicht i ist, so enthalte das Band auch zum Zeitpunkt t an Position i das Zeichen z.“ Nur der besseren Lesbarkeit wegen verwenden wir in der Formel die Implikation „$\Rightarrow$“. Der Ausdruck „$A \Rightarrow B$“ kann durch den äquivalenten Ausdruck „$\neg A \vee B$“ ersetzt werden. Anschließend wenden wir die De Morganschen Regeln an, die besagen, dass ein Ausdruck der Form „$\neg(C \wedge D)$“ äquivalent zu „$\neg C \vee \neg D$“ ist. Dadurch ergibt sich die folgende KNF-Teilformel

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

Die Länge dieser Formel ist $O(p(n))$.

Die Überführungsrelation δ beschreibt die Veränderungen, die beim Konfigurationsübergang erlaubt sind. Unter anderem wird durch δ beschrieben, wie sich der Kopf bewegt. Die Kopfbewegung implementieren wir, indem wir eine Zahl $\kappa \in \{-1, 0, 1\}$ auf die Kopfposition aufaddieren, d.h. wir identifizieren L mit -1, R mit 1 und N mit 0. Die folgende Teilformel beschreibt die Veränderungen, die beim Übergang von K_{t-1} nach K_t erlaubt sind, falls sich die Rechnung zum Zeitpunkt $t-1$ im Zustand k befindet und der Kopf an Position j das Zeichen a

liest.

$$(Q(t-1,k) \wedge H(t-1,j) \wedge S(t-1,j,a)) \;\Rightarrow \bigvee_{(k,a,k',a',\kappa)\in\delta} (Q(t,k') \wedge H(t,j+\kappa) \wedge S(t,j,a'))$$

Diese Teilformel ist ebenfalls selbst erklärend: „Wenn zum Zeitpunkt $t-1$ die Rechnung im Zustand k, der Kopf an Position j ist und an dieser Position das Zeichen a geschrieben steht, so führe in Schritt t eine der möglichen δ-Überführungen bezüglich Zustand, Kopfposition und geschriebenem Zeichen durch." Die Länge dieser Teilformel hängt nur vom Verzweigungsgrad der Übergangsrelation δ ab und ist somit $O(1)$. Für jede aussagenlogische Formel der Länge $O(1)$ gibt es auch eine äquivalente Formel in KNF der Länge $O(1)$. Wir können somit die erlaubten Bandveränderungen an der Kopfposition durch eine KNF-Teilformel konstanter Länge beschreiben.

Wir fassen diese Teilformeln zur Teilformel ϕ'_t in KNF zusammen, die einen korrekten Konfigurationsübergang von K_{t-1} nach K_t sicherstellt. Diese Teilformel hat die Länge $O(p(n))$.

Zusammensetzen der Formel. Zu den oben beschriebenen Teilformeln ϕ_t und ϕ'_t fügen wir noch weitere einfache Teilformeln hinzu, die festlegen, dass die Rechnung in der Startkonfiguration q_0x startet und sich nach $p(n)$ Schritten im Zustand q_{accept} befindet. Die so entstehende Formel ϕ in KNF lautet

$$Q(0,q_0) \;\wedge\; H(0,0) \;\wedge\; \bigwedge_{i=0}^{n} S(0,i,x_i) \;\wedge\; \bigwedge_{i=n+1}^{p(n)} S(0,i,B) \wedge \bigwedge_{i=0}^{p(n)} \phi_i \;\wedge\; \bigwedge_{i=1}^{p(n)} \phi'_i \;\wedge\; Q(p(n),q_{\text{accept}}) \;.$$

Diese Formel hat die Länge $O(p(n)^3)$. Aus unserer Beschreibung ergibt sich, dass die Formel in polynomiell beschränkter Zeit konstruiert werden kann.

Wir haben die Formel ϕ derart konstruiert, dass jede akzeptierende Konfigurationsfolge von M genau einer erfüllenden Belegung von ϕ entspricht. Somit ist ϕ genau dann erfüllbar, wenn eine akzeptierende Konfigurationsfolge von M auf x existiert, also im Falle $x \in L$. Wir haben bereits argumentiert, dass ϕ in polynomieller Zeit aus x konstruiert werden kann. Damit ist die Master-Reduktion abgeschlossen. □

3.6 NP-Vollständigkeit von 3SAT und CLIQUE

Nachdem Cook sein Ergebnis über die NP-Vollständigkeit von SAT vorgestellt hatte, war die Bedeutung dieses Ergebnisses für die Algorithmik noch weitgehend unklar. Ein Jahr später, also 1972, hat Richard Karp demonstriert, dass man die NP-Härte von SAT dazu verwenden kann, um auch für andere Probleme relativ leicht nachzuweisen, dass sie NP-hart sind. Karps Ansatz ist bis heute die wichtigste Methode geblieben um zu zeigen, dass ein Problem – zumindest unter der Hypothese $\mathsf{P} \neq \mathsf{NP}$ – keinen Polynomialzeitalgorithmus zulässt. In seiner bahnbrechenden Arbeit hat Karp die Anwendbarkeit dieser Methode an 21 fundamentalen Problemen aus verschiedensten Bereichen demonstriert, von denen wir hier auch einige präsentieren werden. Inzwischen gibt es hunderte (wahrscheinlich auch tausende) von Problemen, deren NP-Vollständigkeit nach demselben Muster nachgewiesen wurde. Eine Standardreferenz für die wichtigsten NP-vollständigen Probleme ist bis heute das Buch *Computers and Intractability: A Guide to the Theory of NP-Completeness* von Garey und Johnson aus dem Jahre 1979. Mit über 3300 in *Citeseer* gelisteten Zitationen ist dieses Buch tatsächlich das meist zitierte Dokument der Informatik.

Die von Karp geführten NP-Vollständigkeitsbeweise verwenden das folgende Prinzip. Um zu zeigen, dass ein Entscheidungsproblem L NP-vollständig ist, muss zum einen gezeigt werden, dass $L \in \mathsf{NP}$ ist und zum anderen, dass L NP-hart ist, also dass gilt $\forall L' \in \mathsf{NP} : L' \leq_p L$. Die Zugehörigkeit zu NP kann dabei meist leicht durch Angabe eines geeigneten Zertifikates nachgewiesen werden. Die Schwierigkeit liegt in der Regel darin die NP-Härte zu beweisen. Dazu müssen wir jedoch nicht jedesmal eine neue Master-Reduktion führen, sondern es genügt ein bekanntes NP-hartes Problem L^* (wie etwa SAT) auf L zu reduzieren.

Lemma 3.32 *L^* NP-hart, $L^* \leq_p L \Rightarrow L$ NP-hart.*

Beweis: Gemäß Voraussetzung gilt $\forall L' \in \mathsf{NP} : L' \leq_p L^*$ und $L^* \leq_p L$. Aufgrund der Transitivität der polynomiellen Reduktion folgt somit $\forall L' \in \mathsf{NP} : L' \leq_p L$. □

SAT wird uns also als Ausgangspunkt für weitere Härtebeweise dienen. Um SAT auf andere Probleme reduzieren zu können, ist es hilfreich, wenn wir annehmen können, dass aussagenlogische Formeln in einer möglichst strukturierten Form vorliegen. Bei einer Formel in 3KNF enthalten alle Klauseln genau drei Literale. Wenn wir SAT auf aussagenlogische Formeln dieser Form beschränken, nennen wir das Problem 3SAT.

Problem 3.33 (3SAT)

Eingabe: Aussagenlogische Formel ϕ in 3KNF

Frage: Gibt es eine erfüllende Belegung für ϕ?

3SAT ist ein Spezialfall von SAT und deshalb wie SAT in NP. Um zu zeigen, dass 3SAT ebenfalls NP-vollständig ist, müssen wir also „nur“ noch die NP-Härte von 3SAT nachweisen. Wir zeigen

Lemma 3.34 *SAT $\leq_p$ 3SAT.*

Beweis: Gegeben sei eine Formel ϕ in KNF. Wir müssen beschreiben, wie ϕ in polynomieller Zeit in eine *erfüllbarkeitsäquivalente* Formel ϕ' in 3KNF überführt werden kann, d.h.

$$\phi \text{ ist erfüllbar } \Leftrightarrow \ \phi' \text{ ist erfüllbar .}$$

Diejenigen Klauseln aus ϕ, in denen weniger als drei Literale vorkommen, können wir einfach in logisch äquivalente Klauseln mit drei Literalen umformen, indem wir eines der Literale in der Klausel wiederholen. Sei nun

$$C \ := \ \ell_1 \vee \ell_2 \vee \ldots \vee \ell_{k-1} \vee \ell_k$$

eine Klausel aus ϕ mit $k > 3$ Variablen, wobei ℓ_i jeweils für ein negiertes oder unnegiertes Literal steht. In einer *Klauseltransformation* ersetzen wir C durch die Teilformel

$$\underbrace{(\ell_1 \vee \ell_2 \vee \ldots \vee \ell_{k-2} \vee h)}_{=:\, C'} \wedge \underbrace{(\bar{h} \vee \ell_{k-1} \vee \ell_k)}_{=:\, C''} \ ,$$

wobei h eine zusätzlich eingeführte Hilfvariable bezeichnet. Aus der Klausel C der Länge $k > 3$ haben wir somit eine Klausel C' der Länge $k-1$ und eine Klausel C'' der Länge drei erzeugt. Diese Art der Klauseltransformation wenden wir solange auf ϕ an, bis eine Formel ϕ' in 3KNF vorliegt, wobei wir in jeder Klauseltransformation eine neue Hilfsvariable erzeugen. Um die Korrektheit unserer Konstruktion nachzuweisen, müssen wir zeigen, dass jede einzelne Klauseltransformation die Erfüllbarkeitsäquivalenz bewahrt.

- Sei B eine erfüllende Belegung für eine Formel, die die Klausel C enthält. Wir müssen zeigen, dass die Formel auch dann noch erfüllbar ist, wenn wir C durch $C' \wedge C''$ ersetzen. Aus B erhalten wir eine erfüllende Belegung für die neue Formel, wenn wir wie folgt eine geeignete Belegung für h hinzufügen. B erfüllt mindestens eines der Literale aus C. Falls dieses Literal

in der Klausel C' enthalten ist, so ist C' erfüllt, und wir setzen $h = 0$, so dass auch C'' erfüllt ist. Falls dieses Literal in der Klausel C'' enthalten ist, so ist C'' erfüllt, und wir setzen $h = 1$, so dass auch C' erfüllt ist.

- Sei nun B' eine erfüllende Belegung für die Formel, die wir erhalten, wenn wir C durch $C' \wedge C''$ ersetzen. Wir zeigen B' erfüllt auch die Ausgangsformel, denn B' erfüllt C: Falls $h = 0$ gilt, so muss B' eines der Literale $\ell_1, \ldots, \ell_{k-2}$ aus C' erfüllen. Falls hingegen $h = 1$ gilt, so muss B' eines der Literale ℓ_{k-1} oder ℓ_k aus C'' erfüllen. In beiden Fällen, erfüllt die Belegung B' die Klausel C und somit die Ausgangsformel.

Jede einzelne Klauseltransformation kann in Zeit polynomiell in der Klausellänge durchgeführt werden. Für jede Klausel der Länge k in der Formel ϕ benötigen wir $k - 2$ Transformationen um schließlich nur noch Klauseln der Länge 3 zu haben. Die Gesamtanzahl der benötigten Klauseltransformationen ist also durch die Anzahl der in ϕ vorkommenden Literale nach oben beschränkt. Die Reduktion kann deshalb in polynomieller Zeit berechnet werden. □

Durch Anwendung von Lemma 3.32 ergibt sich somit das folgende Ergebnis.

Satz 3.35 *3SAT ist* NP*-vollständig.* □

Die Reduktion von SAT auf 3SAT ist nicht besonders spektakulär, da die beiden Probleme offensichtlich miteinander verwandt sind. Die eigentliche Stärke der Reduktionstechnik liegt darin, dass sie Probleme aus unterschiedlichsten Bereichen zueinander in Beziehung setzen kann. Im Folgenden veranschaulichen wir dies indem wir 3SAT nun auf verschiedene Graph- und Zahlprobleme reduzieren und damit die NP-Vollständigkeit dieser Probleme nachweisen.

Da wir schon wissen, dass das Cliquenproblem in NP ist, müssen wir zum Nachweis der NP-Vollständigkeit nur noch die NP-Härte nachweisen. Dazu führen wir eine Polynomialzeitreduktion von 3SAT auf CLIQUE durch.

Satz 3.36 *CLIQUE ist* NP*-vollständig.*

Beweis: Gegeben sei eine KNF-Formel ϕ. Wir beschreiben eine Transformation von ϕ in einen Graphen $G = (V, E)$ und eine Zahl $k \in \mathbb{N}$ mit der folgenden Eigenschaft.

$$\phi \text{ ist erfüllbar } \Leftrightarrow \ G \text{ hat eine } k\text{-Clique}$$

Seien $C_1, \ldots, C_m$ die Klauseln von ϕ und $\ell_{i,1}, \ell_{i,2}, \ldots, \ell_{i,3}$ die Literale in Klausel C_i. Wir identifizieren die Knoten des Graphen mit den Literalen der Formel, d.h. wir setzen

$$V \;=\; \{\ell_{i,j} \mid 1 \leq i \leq m, 1 \leq j \leq 3\} \;\; .$$

Die Kantenmenge E definieren wir wie folgt. Ein Knotenpaar wird durch eine Kante verbunden, es sei denn

1) die mit den Knoten assoziierten Literale gehören zur selben Klausel oder

2) eines der beiden Literale ist die Negierung des anderen Literals.

Den Parameter k des Cliquenproblems setzen wir gleich m, also gleich der Anzahl der Klauseln in ϕ. Diese einfache Eingabetransformation kann offensichtlich in polynomieller Zeit berechnet werden. Wir müssen noch nachweisen, dass der Graph G genau dann eine k-Clique hat, wenn die Formel ϕ erfüllbar ist.

- Zunächst nehmen wir an ϕ hat eine erfüllende Belegung. Diese Belegung erfüllt in jeder Klausel mindestens ein Literal. Pro Klausel wählen wir eines dieser erfüllten Literale beliebig aus. Wir behaupten, die zugehörigen $k = m$ Knoten bilden eine Clique in G. *Begründung:* Alle selektierten Literale gehören zu verschiedenen Klauseln. Es kann also keine Kante aufgrund von Ausnahmeregel 1 fehlen. Alle selektierten Literale werden gleichzeitig erfüllt. Somit kann kein Paar von sich widersprechenden Literalen dabei sein. Ausnahmeregel 2 findet also ebenfalls keine Anwendung. Also sind alle selektierten Knoten miteinander verbunden, und G enthält eine k-Clique.

- Jetzt nehmen wir an G enthält eine k-Clique. Aufgrund von Ausnahmeregel 1 müssen die Knoten in dieser Clique zu verschiedenen Klauseln gehören. Aus $k = m$ folgt somit, dass die k-Clique genau ein Literal pro Klausel selektiert. Diese Literale können alle gleichzeitig erfüllt werden, da sie sich wegen Ausnahmeregel 2 nicht widersprechen. Also ist ϕ erfüllbar.

□

3.7 NP-Vollständigkeit des Hamiltonkreisproblems

Ein *Hamiltonkreis* ist ein Kreis in einem Graphen, der jeden Knoten genau einmal besucht. Wenn man einen Startknoten und eine Richtung festlegt entlang derer

man dem Kreis folgt, erhält man die Spezifizierung einer *Rundreise* in Form einer Permutation der Knoten des Graphen.

Problem 3.37 (Hamiltonkreis (Hamiltonian Circuit – HC))

Eingabe: Graph $G = (V, E)$

Frage: Gibt es einen Hamiltonkreis in G?

Spricht man von einem *gerichteten Hamiltonkreis* so meint man eigentlich einen *Hamiltonkreis in einem gerichteten Graphen*, also einen Kreis im gerichteten Graphen, der alle Knoten genau einmal besucht.

Problem 3.38 (Gerichteter Hamiltonkreis (Directed HC – DHC))

Eingabe: gerichteter Graph $G = (V, E)$

Frage: Gibt es einen Hamiltonkreis in G?

Diese beiden Probleme sind aufeinander polynomialzeit-reduzierbar. Die eine Richtung ist dabei trivial. Wenn ein ungerichteter Graph G vorliegt, so können wir ihn in einen gerichteten Graphen G' transformieren, indem wir jede ungerichtete Kante durch zwei entgegengesetzte gerichtete Kanten ersetzen. G hat offensichtlich genau dann einen Hamiltonkreis, wenn auch G' einen Hamiltonkreis hat.

Die Umkehrrichtung ist interessanter. Gegeben sei nun ein gerichteter Graph $G = (V, E)$. Wir konstruieren einen ungerichteten Graphen $G' = (V', E')$, der genau dann einen Hamiltonkreis hat, wenn auch G einen Hamiltonkreis hat. Dazu ersetzen wir jeden Knoten $v \in V$ mit seinen Eingangs- und Ausgangskanten durch einen Pfad von drei Knoten $v_{\text{in}}-v-v_{\text{out}}$, wobei wir die Eingangskanten des Knoten mit v_{in} und die Ausgangskanten mit v_{out} verbinden und die Richtung der Kanten ansonsten ignorieren. Diese lokale Ersetzung wird in Abbildung 3.2 illustriert. Wir fassen den Knoten v als *Zimmer* mit zwei Türen auf. Die Knoten v_{in} und v_{out} bezeichnen wir als *Eingangs-* bzw. *Ausgangstür* zum Zimmer v.

Eine Rundreise in G' betritt ein Zimmer v möglicherweise durch seine Ausgangstür v_{out}. Dann muss diese Rundreise das Zimmer jedoch durch seine Eingangstür v_{in} wieder verlassen, weil sonst v_{out} zweimal besucht würde. Da v_{in} jedoch nur mit Ausgangstüren anderer Zimmer verbunden ist, muss das nächste von der Rundreise besuchte Zimmer ebenfalls durch die Ausgangstür betreten und die Eingangstür verlassen werden. Per Induktion lässt sich dieses Argument auf alle Zimmer verallgemeinern. Wenn wir eine derartige Rundreise nun in entgegengesetzter Richtung durchlaufen, so erhalten wir eine Rundreise, bei der alle Zimmer

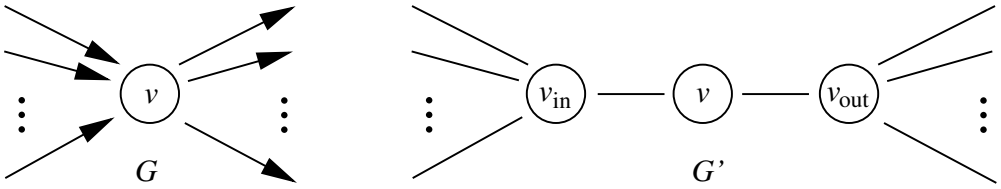

Abbildung 3.2: Lokale Ersetzung bei der Reduktion von DHC auf HC

durch Eingangstüren betreten und Ausgangstüren verlassen werden. Wenn G' also einen Hamiltonkreis hat, so gibt es auch immer eine Richtung in der dieser Kreis durchlaufen werden kann, so dass eine Rundreise entsteht, bei der alle Türen im Sinne ihrer Bezeichnung verwendet werden. Eine derartige Rundreise im ungerichteten Graphen G' gibt es nun offensichtlich genau dann, wenn der gerichtete Graph G einen Hamiltonkreis hat. Somit wurde das folgende Lemma bewiesen.

Lemma 3.39 *HC $\leq_p$ DHC und DHC $\leq_p$ HC.* □

Wir zeigen nun, dass diese beiden Probleme keinen Polynomialzeitalgorithmus haben, es sei denn $\mathsf{P} = \mathsf{NP}$.

Satz 3.40 *HC und DHC sind* NP*-vollständig.*

Beweis: Beide Probleme sind offensichtlich in NP, da man die Kodierung eines Hamiltonkreises in polynomieller Zeit auf ihre Korrektheit überprüfen kann. Da HC und DHC beidseitig aufeinander polynomiell reduzierbar sind, genügt es die NP-Härte eines der beiden Probleme nachzuweisen. Wir zeigen die NP-Härte von DHC, durch eine polynomielle Reduktion von SAT. Dazu beschreiben wir eine polynomiell berechenbare Funktion, die eine KNF-Formel ϕ in einen gerichteten Graphen $G = (V, E)$ transformiert, der genau dann einen Hamiltonkreis hat, wenn ϕ erfüllbar ist.

Seien $x_1, \ldots, x_N$ die Variablen der Formel und $c_1, \ldots, c_M$ die Klauseln. Für jede Variable x_i enthält der Graph G eine Struktur G_i, wie sie in Abbildung 3.3 dargestellt ist. Diese Struktur nennen wir *Diamantengadget*[1]. Diese N Gadgets werden miteinander verbunden, indem wir die Knoten t_i und s_{i+1} $(1 \leq i \leq N-1)$ sowie t_N und s_1 miteinander identifizieren. In dem so entstehenden Graphen besucht jede Rundreise, die beim Knoten s_1 startet, die Gadgets in der Reihenfolge

[1] *Gadget* (engl.) bedeutet *Apparatur, Dingsbums* (sl.), *Gerät,* oder *Vorrichtung* und wird im Zusammenhang von Reduktionen auf Graphprobleme im Sinne von *Graphstruktur, die eine Entität des Ausgangsproblems repräsentiert*, verwendet.

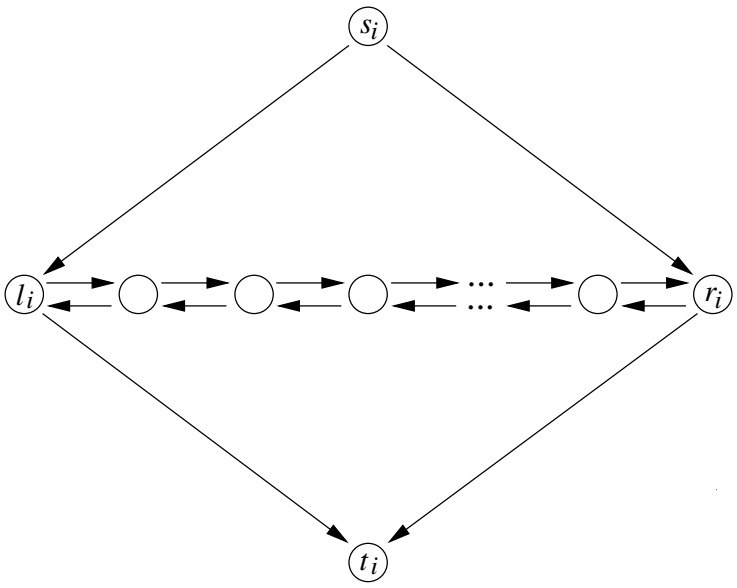

Abbildung 3.3: Diamantengadget G_i für Variable x_i

$G_1, G_2, \ldots, G_N$. Die Rundreise hat dabei für jedes Gadget G_i die Freiheit das Gadget *von links nach rechts*, also von l_i nach r_i, oder *von rechts nach links*, also von r_i nach l_i, zu durchlaufen. Die erste Variante interpretieren wir als Variablenbelegung $x_i = 1$, die zweite als Variablenbelegung $x_i = 0$.

Wir können somit Belegungen der Variablen $x_1, \ldots, x_N$ repräsentieren, aber die Zusammensetzung der Formel ϕ ist bisher bei der Konstruktion des Graphen unberücksichtigt geblieben. Jetzt fügen wir einen weiteren Knoten für jede Klausel c_j ein. Diesen *Klauselknoten* bezeichnen wir wie die Klausel selbst mit c_j. Falls die Variable x_i negiert oder unnegiert als Literal in Klausel c_j enthalten ist, so fügen wir Kanten hinzu, die das Gadget G_i auf geeignete Art und Weise mit dem Klauselknoten c_j verbinden. Das Gadget G_i enthalte zu diesem Zweck in der doppelt verketteten Liste zwischen l_i und r_i ein benachbartes Knotenpaar für jede Klausel c_j in der die Variable x_i enthalten ist. Den linken Knoten in diesem Paar bezeichnen wir mit a_{ij}, den rechten mit b_{ij}. Die Knotenpaare für verschiedene Klauseln sind jeweils durch einen Zwischenknoten (hier grau markiert) getrennt. Auch die beiden Paare, die in der Kette am weitesten links und rechts stehen, trennen wir durch einen Zwischenknoten von l_i bzw. r_i. Ist die Variable x_i unnegiert in c_j enthalten, so fügen wir die Kanten (a_{ij}, c_j) und (c_j, b_{ij}) hinzu, wie in Abbildung 3.4 beschrieben. Ist die Variable x_i hingegen negiert in c_j enthalten, so fügen wir die Kanten (b_{ij}, c_j) und (c_j, a_{ij}) hinzu, wie in Abbildung 3.5 beschrieben. Damit ist die Beschreibung des Graphen G abgeschlossen. G kann offensichtlich in

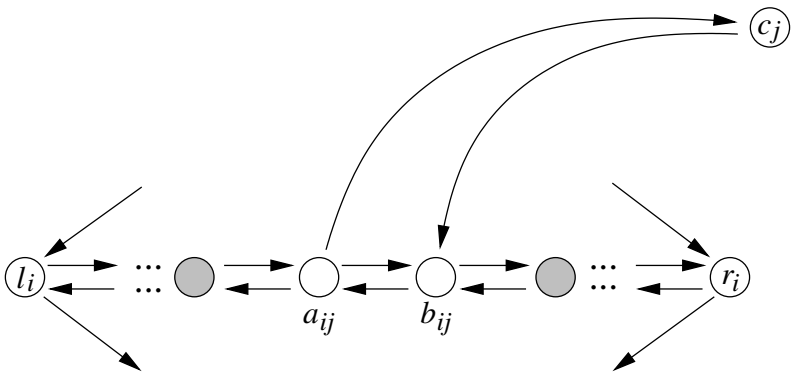

Abbildung 3.4: Einbindung von Klauselknoten c_j in das Gadget G_i für den Fall, dass die Klausel c_j das Literal x_i enthält. Bei den grauen Knoten handelt es sich um die Zwischenknoten, die zwischen den Knotenpaaren für verschiedene Klauseln eingefügt werden.

polynomieller Zeit aus ϕ konstruiert werden.

Durch die Hinzunahme der Knoten c_j ist leider nicht mehr sofort klar, dass die Rundreise die Gadgets tatsächlich nacheinander besucht. Ist es vielleicht möglich, dass die Tour aus dem Gadget G_i zum Knoten c_j geht und dann von dort zu einem anderen Gadget $G_{i'}$ mit $i' \neq i$ wechselt? – Wir behaupten, dass dieses nicht passieren kann. Zum Widerspruch nehmen wir an, die Rundreise wechselt während des Besuchs von Gadget G_i zu c_j und kehrt dann nicht direkt zum Gadget G_i zurück. O.B.d.A. ist x_i in unnegierter Form in der Klausel c_j enthalten, wie in Abbildung 3.4 beschrieben. Den Zwischenknoten links von a_{ij} bezeichnen wir mit u und den Zwischenknoten rechts von b_{ij} mit v. Wir unterscheiden zwei Fälle: Im ersten Fall wird der Knoten c_j über einen Weg $\ldots u, a_{ij}, c_j, w \ldots$ besucht, wobei w zu einem anderem Gadget gehört. Dann muss die Tour den Knoten b_{ij} über den Knoten u besuchen. An dieser Stelle steckt die Tour jedoch fest, denn nun hat b_{ij} keinen Nachbarn mehr, der noch nicht besucht wurde. Im zweiten Fall wird der Knoten c_j über den Weg $\ldots v, b_{ij}, a_{ij}, c_j, w \ldots$ besucht. In diesem Fall gibt es aus gleichen Gründen keine Möglichkeit den Knoten u zu durchlaufen. Also werden die Gadgets tatsächlich in der Reihenfolge $G_1, G_2, \ldots, G_N$ abgearbeitet. Somit können wir an unserer Interpretation für die Belegung der Variablen – entsprechend der Richtung in der die Gadgets durchlaufen werden – festhalten.

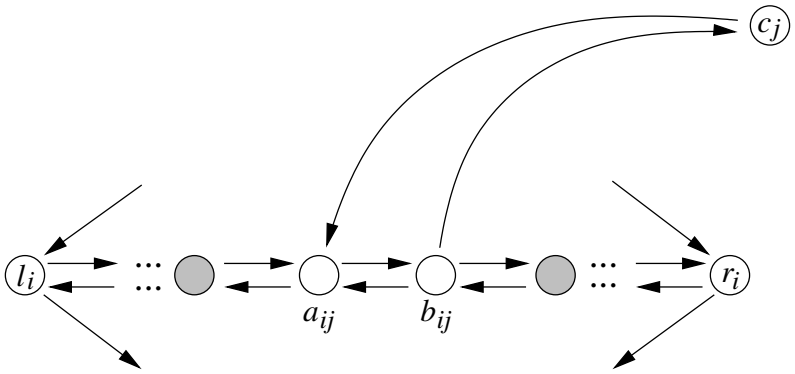

Abbildung 3.5: Einbindung von Klauselknoten c_j in das Gadget G_i für den Fall, dass die Klausel c_j das Literal $\bar{x}_i$ enthält.

Wir müssen zeigen, dass G genau dann einen Hamiltonkreis enthält, wenn ϕ eine erfüllenden Belegung hat.

- Eine Rundreise für G besucht auch alle Klauselknoten. Wird ein Klauselknoten c_j aus einem Gadget G_i heraus von links nach rechts durchlaufen, so muss gemäß unserer Konstruktion, die Klausel c_j das Literal x_i enthalten. Also wird diese Klausel durch die mit der Laufrichtung von links nach rechts assoziierten Belegung $x_i = 1$ erfüllt. Bei einer Laufrichtung von rechts nach links, die mit der Belegung $x_i = 0$ assoziiert ist, wird die Klausel ebenso erfüllt, weil sie in diesem Fall das Literal $\bar{x}_i$ enthält. Also erfüllt die mit der Rundreise assoziierte Belegung alle Klauseln.

- Eine Belegung beschreibt in welcher Richtung die Gadgets $G_1, \ldots, G_N$ jeweils durchlaufen werden. Die Klauselknoten können wir in die Rundreise einbauen, indem wir für jeden Klauselknoten c_j ($1 \leq j \leq M$) eine Variable x_i auswählen, die die zugehörige Klausel erfüllt, und beim Besuch des Gadgets G_i einen Umweg über den Klauselknoten c_j nehmen. Sollte die Klausel für $x_i = 1$ erfüllt sein, so ist x_i unnegiert in c_j enthalten, und somit ist ein Besuch von c_j beim Durchlaufen des Gadgets G_i von links nach rechts möglich. Sollte die Klausel hingegen für $x_i = 0$ erfüllt sein, so ist die Variable negiert in der Klausel enthalten, und der Besuch von c_j kann beim Durchlaufen des Gadgets von rechts nach links erfolgen. Also können alle

Klauselknoten in die Rundreise eingebunden werden. □

HC können wir nun auf die Entscheidungsvariante von TSP reduzieren, wobei wir nur auf zwei verschiedene Distanzwerte zwischen den Orten zurückgreifen müssen, nämlich Distanzen mit Wert 1 oder 2. Diese eingeschränkte TSP-Variante bezeichnen wir mit $\{1,2\}$-*TSP*. Gegeben sei ein ungerichteter Graph $G = (V, E)$ für HC. Wir definieren eine symmetrische Distanzmatrix für TSP, bei der wir für Knotenpaare zwischen denen eine Kante in G vorliegt, die Distanz 1 eintragen, und für alle anderen Paare die Distanz 2. G enthält nun einen Hamiltonkreis genau dann, wenn die Eingabe für $\{1,2\}$-TSP eine Rundreise der Länge $|V|$ enthält. Durch diese Polynomialzeitreduktion ergibt sich unmittelbar die Härte von $\{1,2\}$-TSP und somit auch die Härte des allgemeinen TSP.

Korollar 3.41 *Die Entscheidungsvariante von* $\{1,2\}$*-TSP ist* NP*-hart.* □

3.8 NP-Vollständigkeit einiger Zahlprobleme

Wir führen zunächst zwei elementare arithmetische Probleme ein: SUBSET-SUM und PARTITION. Wir reduzieren 3SAT polynomiell auf SUBSET-SUM und dann SUBSET-SUM auf PARTITION. Aus der NP-Härte von SUBSET-SUM folgt dann fast direkt die Härte des Rucksackproblems und aus der NP-Härte von PARTITION ebenso leicht die Härte des Bin-Packing-Problems.

3.8.1 NP-Vollständigkeit von SUBSET-SUM

Problem 3.42 (SUBSET-SUM)

Eingabe: $a_1, \ldots, a_N \in \mathbb{N}$, $b \in \mathbb{N}$

Frage: Gibt es $K \subseteq \{1, \ldots, N\}$ *mit* $\sum_{i \in K} a_i = b$*?*

Satz 3.43 *SUBSET-SUM ist* NP*-vollständig.*

Beweis: Da die Teilmenge K als Zertifikat verwendet werden kann, ist SUBSET-SUM offensichtlich in NP enthalten. Um die NP-Härte des Problems nachzuweisen, beschreiben wir eine Polynomialzeitreduktion von 3SAT. Gegeben sei eine

Formel ϕ in 3KNF. Diese Formel bestehe aus M Klauseln $c_1, \ldots, c_M$ über N Variablen $x_1, \ldots, x_N$. Für $i \in \{1, \ldots, N\}$ sei

$$\begin{aligned} S(i) &= \{j \in \{1, \ldots, M\} \mid \text{Klausel } c_j \text{ enthält Literal } x_i\} \ , \\ S'(i) &= \{j \in \{1, \ldots, M\} \mid \text{Klausel } c_j \text{ enthält Literal } \bar{x}_i\} \ . \end{aligned}$$

Aus der Formel ϕ erzeugen wir eine SUBSET-SUM-Eingabe. Wir beschreiben die Eingabe von SUBSET-SUM in Form von Dezimalzahlen, die aus $N + M$ Ziffern bestehen. Die k-te Ziffer einer Zahl a bezeichnen wir dabei mit $a(k)$. Für jede boolesche Variable x_i, $i \in \{1, \ldots, N\}$, enthält die SUBSET-SUM-Eingabe zwei Zahlen a_i und a'_i, deren Ziffern wie folgt definiert sind.

$$\begin{aligned} a_i(i) = 1 \quad \text{und} \quad \forall j \in S(i) : a_i(N + j) = 1 \ , \\ a'_i(i) = 1 \quad \text{und} \quad \forall j \in S'(i) : a'_i(N + j) = 1 \ . \end{aligned}$$

Alle anderen Ziffern setzen wir auf den Wert 0. Neben den jeweils zwei Zahlen pro Variable generieren wir zwei weitere Zahlen h_j und h'_j pro Klausel, die wir als *Füllzahlen* bezeichnen. Für $1 \leq j \leq M$ gelte $h_j = h'_j$, wobei nur die Ziffern an Position $N + j$ den Wert 1 haben, alle anderen Ziffern erhalten den Wert 0. Wir müssen noch den zur SUBSET-SUM-Eingabe gehörenden Summenwert b spezifizieren: Wir setzen $b(k) = 1$ für $1 \leq k \leq N$ und $b(k) = 3$ für $N + 1 \leq k \leq N + M$.

Die Reihenfolge bzw. Wertigkeit der einzelnen Ziffern spielt in unserer Konstruktion keine Rolle. Wichtig ist lediglich, dass es bei der Addition einer beliebigen Teilmenge der Zahlen a_i, a'_i, h_i, h'_i keinen Additionsübertrag von Ziffer zu Ziffer geben kann. Dies wird dadurch gewährleistet, dass pro Ziffernposition höchsten fünf dieser Zahlen eine 1 enthalten. Alle anderen Zahlen enthalten an dieser Position eine 0. (Es würde deshalb tatsächlich ausreichen, ein Zahlensystem zur Basis 6 statt des Dezimalsystems zu verwenden.) Zur besseren Intuition können wir uns vorstellen alle Zahlen der SUBSET-SUM-Eingabe in einer Tabelle abzutragen, in der für jede Ziffernposition eine Spalte vorgesehen ist. Abbildung 3.6 zeigt beispielsweise die Tabelle für die folgende 3KNF-Formel.

$$(x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_2 \vee x_3 \vee \bar{x}_N) \wedge ...$$

Die Zahlen in unserer Konstruktionsbeschreibung sind gigantisch groß. Sie bestehen aus $N + M$ Ziffern. Ihre Kodierungslänge ist dennoch polynomiell in der Eingabelänge der Formel ϕ beschränkt, deren Kodierung ja mindetens $N + M$

	1	2	3	$\cdots$	N	$N+1$	$N+2$	$\cdots$	$N+M$
a_1	1	0	0	$\cdots$	0	1	0	$\cdots$	$\cdots$
a'_1	1	0	0	$\cdots$	0	0	0	$\cdots$	$\cdots$
a_2	0	1	0	$\cdots$	0	0	1	$\cdots$	$\cdots$
a'_2	0	1	0	$\cdots$	0	1	0	$\cdots$	$\cdots$
a_3	0	0	1	$\cdots$	0	1	1	$\cdots$	$\cdots$
a'_3	0	0	1	$\cdots$	0	0	0	$\cdots$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
a_N	0	0	0	$\cdots$	1	0	0	$\cdots$	$\cdots$
a'_N	0	0	0	$\cdots$	1	0	1	$\cdots$	$\cdots$
h_1	0	0	0	$\cdots$	0	1	0	$\cdots$	0
h'_1	0	0	0	$\cdots$	0	1	0	$\cdots$	0
h_2	0	0	0	$\cdots$	0	0	1	$\cdots$	0
h'_2	0	0	0	$\cdots$	0	0	1	$\cdots$	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
h_M	0	0	0	$\cdots$	0	0	0	$\cdots$	1
h'_M	0	0	0	$\cdots$	0	0	0	$\cdots$	1
b	1	1	1	$\cdots$	1	3	3	$\cdots$	3

Abbildung 3.6: Illustration zur Reduktion von 3SAT auf SUBSET-SUM

Zeichen benötigt. Insbesondere kann die SUBSET-SUM-Eingabe in polynomieller Zeit aus der 3SAT-Eingabe konstruiert werden. Wir müssen zeigen, es gibt genau dann eine Teilmenge der Zahlen a_i, a'_i, h_i, h'_i mit Summenwert b, falls die Formel ϕ eine erfüllende Belegung hat.

- Falls es eine erfüllende Belegung x^* gibt, so nehmen wir diejenigen Zahlen a_i in unsere Teilmenge K auf, für die gilt $x^*_i = 1$, ansonsten nehmen wir a'_i auf. Sei A die Summe der ausgewählten Zahlen a_i und a'_i. Da für jedes $i \in \{1, \ldots, N\}$ entweder a_i oder a'_i aufgenommen wird, gilt $A(i) = 1$. Da jede Klausel mindestens ein erfülltes Literal enthält, gilt zudem $A(N + j) \geq 1$ für $1 \leq j \leq M$. Falls $A(N + j) < 3$ so können wir eine oder beide der Füllzahlen h_j und h'_j verwenden um exakt den geforderten Wert 3 an Ziffernposition $N + j$ der Summe zu erhalten. Also gibt es eine zulässige Lösung für SUBSET-SUM.

- Falls es nun eine Teilmenge K der Zahlen a_i, a'_i, h_i, h'_i mit Summenwert

b gibt, so enthält K für jedes $i \in \{1, \ldots, N\}$ entweder die Zahl a_i oder die Zahl a'_i, denn sonst würde sich in Ziffernposition i keine 1 ergeben. Setze $x^*_i = 1$, falls $a_i \in K$, und $x^*_i = 0$, falls $a'_i \in K$. Wir zeigen, x^* ist eine erfüllende Belegung: Sei A die Summe der Zahlen a_i und a'_i, die in K enthalten sind. Es gilt $A(N + j) \geq 1$ für $1 \leq j \leq M$, weil nur so die Ziffer an Position $N+j$ in der SUBSET-SUM-Lösung den Wert 3 annehmen kann, da nur zwei Füllzahlen für diese Position zur Verfügung stehen. Dadurch ist sichergestellt, dass jede Klausel mindestens ein Literal mit Wert 1 enthält, so dass x^* eine erfüllende Belegung ist.

□

3.8.2 NP-Vollständigkeit von PARTITION

Problem 3.44 (PARTITION)

Eingabe: $a_1, \ldots, a_N \in \mathbb{N}$

Frage: Gibt es $K \subseteq \{1, \ldots, N\}$ *mit* $\sum_{i \in K} a_i = \sum_{i \in \{1,\ldots,N\} \setminus K} a_i$*?*

PARTITION ist ein Spezialfall von SUBSET-SUM, da die gestellte Frage äquivalent zur Frage ist, ob es eine Teilmenge K mit Summenwert $\frac{1}{2} \sum_{i=1}^{N} a_i$ gibt. Wir zeigen, dass diese Spezialisierung des SUBSET-SUM-Problems nicht einfacher zu lösen ist, als das allgemeine SUBSET-SUM-Problem.

Satz 3.45 *PARTITION ist* NP*-vollständig.*

Beweis: PARTITION ist offensichtlich $\in$ NP, weil es ein Spezialfall von SUBSET-SUM ist. Um zu zeigen, dass PARTITION NP-hart ist, zeigen wir SUBSET-SUM $\leq_p$ PARTITION. Die Eingabe von SUBSET-SUM sei $a_1, \ldots, a_N \in \mathbb{N}$ und $b \in \mathbb{N}$. Es sei $A = \sum_{i=1}^{N} a_i$. Wir bilden diese Eingabe für SUBSET-SUM auf eine Eingabe für PARTITION ab, die aus den $N + 2$ Zahlen $a'_1, \ldots, a'_{N+2}$ bestehe. Dazu setzen wir

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

In der Summe ergeben diese $N + 2$ Zahlen den Wert $4A$. PARTITION fragt also danach, ob es eine Aufteilung der Zahlen $a'_1, \ldots, a'_{N+2}$ in zwei Teilmengen gibt, die in der Summe jeweils $2A$ ergeben oder äquivalent, ob es eine Teilmenge der Zahlen mit Summenwert $2A$ gibt.

Diese einfache Eingabetransformation ist in polynomieller Zeit berechenbar. Wir müssen zeigen, dass die Eingabe für SUBSET-SUM genau dann eine Lösung hat, wenn auch die Eingabe für PARTITION eine Lösung hat.

- Wenn es eine geeignete Aufteilung der Eingabezahlen für PARTITION gibt, so können a'_{N+1} und a'_{N+2} dabei nicht in derselben Teilmenge sein, denn $a'_{N+1} + a'_{N+2} = 3A$. Deshalb ergibt sich auch eine Lösung für SUBSET-SUM, denn diejenigen Zahlen aus $a'_1, \ldots, a'_N$ bzw. $a_1, \ldots, a_N$, die sich in derselben Teilmenge wie a'_{N+1} befinden, summieren sich auf zu $2A - a'_{N+1} = b$.
- Die Umkehrrichtung gilt ebenfalls, denn wenn es eine Teilmenge der Zahlen $a_1, \ldots, a_N$ bzw. $a'_1, \ldots, a'_N$ mit Summenwert b gibt, so können wir die Zahl a'_{N+1} zu dieser Teilmenge hinzufügen, und erhalten dadurch eine Teilmenge der Zahlen $a'_1, \ldots, a'_{N+2}$, deren Summenwert $2A$ ist.

□

3.8.3 Konsequenzen für KP und BPP

Das folgende Korollar zeigt, dass das Rucksackproblem (KP) und das Bin-Packing-Problem (BPP) keinen Polynomialzeitalgorithmus zulassen, es sei denn $\mathsf{P} = \mathsf{NP}$. Diese Ergebnisse folgen unmittelbar aus der NP-Vollständigkeit von SUBSET-SUM und PARTITION.

Korollar 3.46 *Die Entscheidungsvarianten von KP und BPP sind* NP*-vollständig.*

Beweis: Wir reduzieren SUBSET-SUM auf die Entscheidungsvariante von KP. Bei der Entscheidungsvariante von KP sind N Objekte mit Gewicht $w_1, \ldots, w_N$ und Nutzenwert $p_1, \ldots, p_N$ gegeben. Gefragt ist nach einer Teilmenge der Objekte mit Gewicht höchstens B und Nutzenwert mindestens P. Wir setzen $w_i = p_i = a_i$ für $1 \leq i \leq N$ und $B = P = b$. Eine Teilmenge der Zahlen $a_1, \ldots, a_N$ mit Summenwert b existiert genau dann, wenn es eine Rucksackbepackung mit Gewicht höchstens $B = b$ und Profit mindestens $P = b$ gibt.

Wir reduzieren PARTITION auf die Entscheidungsvariante von BPP. Bei der Entscheidungsvariante von BPP sind N Objekte mit Gewicht $w_1, \ldots, w_N$ gegeben. Die Frage lautet, ob diese Objekte in k Behälter mit Gewichtsschranke jeweils b passen. Wir setzen $w_i = a_i$ für $1 \leq i \leq N$, $k = 2$ und $b = \frac{1}{2}\sum_{i=1}^{N} a_i$. Die Zahlen $a_1, \ldots, a_N$ können genau dann in zwei gleich große Teilsummen aufgeteilt werden, wenn die Objekte mit den Gewichten $a_1, \ldots, a_N$ in zwei Behälter mit Gewichtsschranke $b = \frac{1}{2}\sum_{i=1}^{N} a_i$ passen.

Beide Reduktion können in polynomieller Zeit berechnet werden. □

3.9 Übersicht über die Komplexitätslandschaft

Wir fassen zusammen und geben eine Übersicht über Komplexitätsklassen, die wir kennen gelernt haben. Wir beschränken uns dabei auf Entscheidungsprobleme. P enthält die Probleme, die wir effizient lösen können, also in polynomieller Zeit in Rechenmodellen, die den existierenden Computern nahe kommen. NP enthält diejenigen Probleme, die wir in polynomieller Zeit auf nicht deterministischen Maschinen lösen können. Unter der Hypothese $\mathsf{P} \neq \mathsf{NP}$ können wir nicht alle Probleme aus NP effizient lösen, insbesondere die schwierigsten unter den Problemen aus NP, die NP-vollständigen Probleme, haben keinen Polynomialzeitalgorithmus. Diese fasst man in der Klasse NPC zusammen.

Sei PSPACE die Klasse derjenigen Probleme, die wir mit einem polynomiell beschränkten Band auf einer Turingmaschine lösen können. Im Gegensatz zur Zeitkomplexität kann man nachweisen (*Satz von Savitch*), dass diese Klasse sich nicht verändert, wenn wir zwischen deterministischen und nicht-deterministischen Turingmaschinen wechseln. Alternativ kann PSPACE auch über Registermaschinen definiert werden, ohne die Bedeutung der Klasse zu verändern. Im Beweis des Satzes von Cook und Levin haben wir nebenbei beobachtet, dass wir die Probleme in NP mit polynomiell großem Speicherplatz lösen können, da sich der Kopf einer Turingmaschine pro Zeitschritt nur eine Position bewegen kann. Also gilt $\mathsf{NP} \subseteq \mathsf{PSPACE}$.

Die Klasse der Probleme mit einer Laufzeitschranke $2^{p(n)}$ für ein Polynom p bezeichnen wir als EXPTIME. In Korollar 3.20 haben wir gezeigt, dass gilt $\mathsf{NP} \subseteq \mathsf{EXPTIME}$. In Übungsaufgabe 1.9 haben wir gezeigt, dass es bei einer Speicherplatzbeschränkung der Größe $s(n)$ nur $2^{O(s(n))}$ viele verschiedenen Konfigurationen für eine Turingmaschine gibt, so dass im Falle einer solchen Speicherplatzbeschränkung auch die Rechenzeit durch $2^{O(s(n))}$ beschränkt ist. Die Pro-

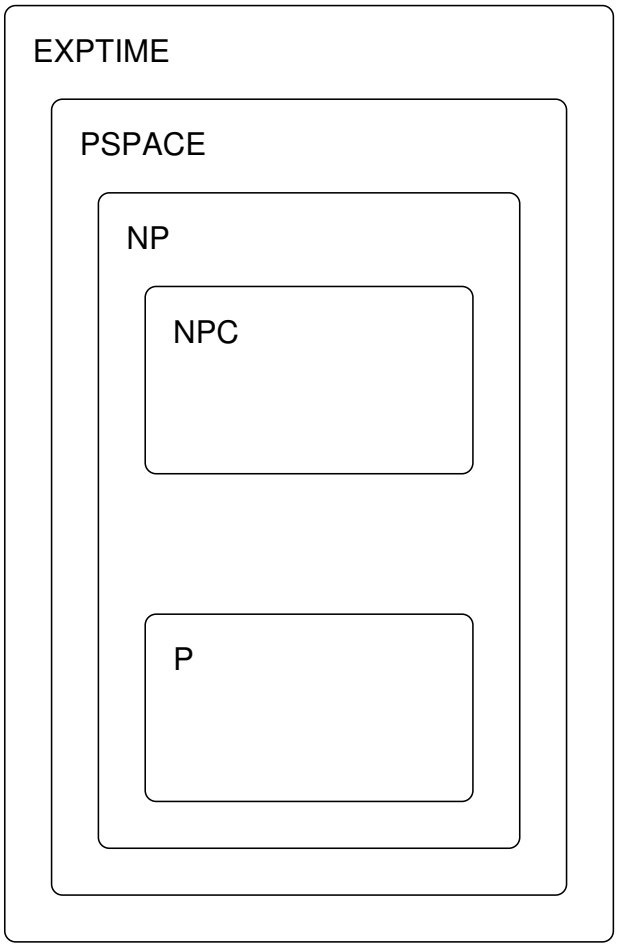

Abbildung 3.7: Überblick über die Komplexitätslandschaft unter der Hypothese $\mathsf{P} \subsetneq \mathsf{NP} \subsetneq \mathsf{PSPACE} \subsetneq \mathsf{EXPTIME}$.

bleme in PSPACE können deshalb in Zeit $2^{p(n)}$ gelöst werden, so dass tatsächlich gilt PSPACE $\subseteq$ EXPTIME. Wir fassen zusammen.

$$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXPTIME}$$

Es ist bekannt, dass Probleme existieren, die in EXPTIME aber nicht in P enthalten sind (*Hierarchiesätze*). Somit gilt P $\subsetneq$ EXPTIME. In allen anderen Fällen ist unklar, ob die angegebenen Inklusionen echt oder unecht sind, ob also etwa die Klassen P und NP oder sogar die Klassen P und PSPACE nicht doch zusammenfallen. Abbildung 3.7 beschreibt die Situation unter der verbreiteten Hypothese, dass diese Inklusionen alle echt sind. In der Vorlesung *Komplexitätstheorie* im Hauptstudium wird dieses Bild weiter verfeinert. Zwischen P und PSPACE wird beispielsweise die Komplexitätslandschaft durch die sogenannte *polynomielle Hierarchie* gegliedert, deren unterste Stufe durch die Klassen NP und Co-NP gebildet wird, wobei Co-NP die Komplemente der Sprachen aus NP enthält, also diejenigen Entscheidungsprobleme bei denen die „NEIN-Antworten“ polynomiell verifizierbar sind.

Wir verlassen an dieser Stelle die klassische Komplexitätstheorie und kümmern uns im verbleibenden Rest der Veranstaltung um einen alternativen Ansatz um mit schwierigen Optimierungsproblemen wie KP, BPP und TSP umzugehen, für deren Entscheidungsvarianten wir die NP-Vollständigkeit nachgewiesen haben.

3.10 Approximationsalgorithmen für NP-harte Probleme

Wir nennen ein Optimierungsproblem NP-hart, wenn die Entscheidungsvariante des Problems NP-hart ist. Unter der Hypothese P $\neq$ NP gibt es für NP-harte Optimierungsprobleme wie das Rucksackproblem (KP), das Bin-Packing-Problem (BPP) und das Traveling-Salesperson-Problem (TSP) keinen Polynomialzeitalgorithmus, der eine optimale Lösung berechnet. In der Praxis ist es häufig jedoch ausreichend eine Lösung zu berechnen, deren Zielfunktionswert vielleicht nicht optimal ist, aber das Optimum annähernd erreicht.

Sei Π ein Optimierungsproblem. Für eine Instanz I von Π bezeichnen wir den optimalen Zielfunktionswert mit $opt(I)$. Wir unterscheiden *Minimierungs-* und *Maximerungsprobleme* danach, ob der Zielfunktionswert wie etwa bei TSP minimiert oder wie beim Rucksackproblem maximiert werden soll.

- Ein α-*Approximationsalgorithmus*, $\alpha > 1$, für ein Minimierungsproblem Π berechnet für jede Instanz I von Π eine zulässige Lösung mit Zielfunktionswert höchstens $\alpha \cdot opt(I)$.

- Ein α-*Approximationsalgorithmus*, $\alpha < 1$, für ein Maximierungsproblem Π berechnet für jede Instanz I von Π eine zulässige Lösung mit Zielfunktionswert mindestens $\alpha \cdot opt(I)$.

Der Term α wird auch als *Approximationsfaktor* oder *Approximationsgüte* bezeichnet. Ein 2-Approximationsalgorithmus für TSP würde beispielsweise für jede TSP-Instanz eine Rundreise berechnen, die höchstens doppelt so teuer ist, wie eine optimale Rundreise. Ein 0.99-Approximationsalgorithmus für das Rucksackproblem würde für jede Rucksackinstanz eine zulässige Rucksackbepackung berechnen, die mindestens 99% des optimalen Nutzenwertes erreicht.

Nicht alle NP-harten Optimierungsprobleme haben einen Approximationsalgorithmus mit polynomiell beschränkter Laufzeit. Einge wenige NP-harte Optimierungsprobleme lassen sich allerdings beliebig gut approximieren. Diese Probleme haben ein sogenanntes Approximationsschema. Zum Abschluss der Vorlesung demonstrieren wir dieses Prinzip anhand des Rucksackproblems.

3.10.1 Ein Approximationsschema für das Rucksackproblem

Ein *Approximationsschema* A für ein Optimierungsproblem Π ist ein Algorithmus, der es ermöglicht, in polynomieller Zeit eine zulässige Lösung für Π mit beliebiger Approximationsgüte zu berechnen. Der gewünschte Approximationsfaktor ist dabei ein Parameter des Algorithmus: Bei Eingabe einer Probleminstanz I und einer Zahl $\epsilon > 0$, berechnet A eine zulässige Lösung mit Approximationsfaktor $1 + \epsilon$, falls Π ein Minimierungsproblem ist, bzw. $1 - \epsilon$, falls Π ein Maximierungsproblem ist. Die Laufzeit eines Approximationsschemas hängt typischerweise von der Eingabelänge n und dem Parameter ϵ ab. Wir unterscheiden zwei Varianten von Approximationsschemata bezüglich ihrer Laufzeitschranken.

- Ein Approximationschema A wird als *FPTAS (fully polynomial time approximation scheme)* bezeichnet, falls die Laufzeit von A polynomiell sowohl in n als auch in $\frac{1}{\epsilon}$ beschränkt ist.

- Ein Approximationschema A wird als *PTAS (polynomial time approximation scheme)* bezeichnet, falls die Laufzeit von A für jedes konstante $\epsilon > 0$ polynomiell in n beschränkt ist.

Typische Laufzeitschranken für ein FPTAS sind etwa $O(n^2/\epsilon)$ oder $O(n^3 + 1/\epsilon^2)$. Wir werden zeigen, dass das Rucksackproblem ein FPTAS hat. Die Anforderungen an ein PTAS sind schwächer, denn die Laufzeit muss nicht polynomiell von $\frac{1}{\epsilon}$ abhängen. Statt dessen wird ϵ als Konstante aufgefasst. Für ein PTAS sind Laufzeitschranken der Form $O(2^{1/\epsilon}n^2)$ oder auch $O(n^{1/\epsilon}\log n)$ erlaubt, die für ein FPTAS nicht zulässig wären.

Das FPTAS für das Rucksackproblem basiert auf einem dynamischen Programm. Sei $P = \max_{1\leq i\leq N} p_i$ der maximale Nutzenwert über die N gegebenen Objekte. Der Algorithmus findet eine optimale Rucksackbepackung in $O(N^2P)$ uniformen Rechenschritten, von denen jeder einzelne in polynomieller Zeit berechnet werden kann, wie wir in Lemma 3.47 zeigen werden. Beachte, die Laufzeitschranke $O(N^2P)$ ist nicht polynomiell in der Eingabelänge n beschränkt. Zwar gilt $N \leq n$ aber P kann exponentiell in n sein. Eine derartige Laufzeitschranke, die polynomiell lediglich in der Größe (statt der Kodierungslänge) der Eingabezahlen beschränkt ist, wird als *pseudopolynomiell* bezeichnet.

Lemma 3.47 *Für das Rucksackproblem gibt es einen Algorithmus, der eine optimale Lösung in $O(N^2P)$ uniformen Rechenschritten berechnet.*

Beweis: Der optimale Nutzenwert liegt offensichtlich zwischen 0 und $N \cdot P$. Für $i \in \{0,\ldots,N\}$ und $p \in \{0,\ldots,N\cdot P\}$, sei $A_{i,p}$ das kleinstmögliche Gewicht, mit dem man den Nutzenwert p exakt erreichen kann, wenn man nur Objekte aus der Menge $\{1,\ldots,i\}$ verwenden darf. Wir setzen $A_{i,p} = \infty$, falls der Nutzen p nicht durch eine Teilmenge von $\{1,\ldots,i\}$ erreicht werden kann. Für alle $i \in \{1,\ldots,N\}$ gilt $A_{i,0} = 0$. Außerdem gilt $A_{0,p} = \infty$ für alle $p > 0$ und $A_{i,p} = \infty$ für alle $p < 0$. Für $i \in \{1,\ldots,N\}$ und $p \in \{1,\ldots,N\cdot P\}$ gilt nun die folgende Rekursionsgleichung:

$$A_{i+1,p} \;=\; \min(A_{i,p},\; A_{i,p-p_{i+1}} + w_{i+1}) \;\; .$$

Wir verwenden den Ansatz der dynamischen Programmierung und berechnen Zeile für Zeile alle Einträge in der Tabelle bzw. Matrix $(A_{i,p})_{i\in\{0,\ldots,N\},p\in\{0,\ldots,NP\}}$. Für jeden Tabelleneintrag benötigen wir dabei eine konstante Anzahl von Rechenschritten. Die Laufzeit entspricht somit der Größe der Tabelle $O(N^2P)$. Der gesuchte, maximal erreichbare Nutzenwert ist

$$\max\{p \,|\, A_{N,p} \leq b\} \;\; .$$

Die zugehörige Rucksackbepackung lässt sich durch Abspeichern geeigneter Zusatzinformationen zu den einzelnen Tabelleneinträgen rekonstruieren, so dass die optimale Rucksackbepackung in Zeit $O(N^2P)$ berechnet werden kann. □

Die Laufzeitschranke hängt linear von der Größe der Nutzenwerte ab. Zu beachten ist, dass der Algorithmus davon ausgeht, dass die Nutzenwerte – wie in der Problemspezifikation festgelegt – natürliche Zahlen sind. Wir können die Größe der Eingabezahlen verringern, indem wir alle Nutzenwerte mit demselben Skalierungsfaktor α multiplizieren (z.B. $\alpha = 0.01$) und die dabei entstehenden Nachkommastellen streichen. Was passiert nun, wenn wir das dynamische Programm mit derart skalierten und gerundeten Nutzenwerten aufrufen?

- Einerseits verbessert sich die Laufzeit des Algorithmus um den Skalierungsfaktor α, da die Laufzeitschranke linear vom größten Nutzenwert abhängt, und sich dieser Wert um den Faktor α verringert hat.
- Andererseits können wir nicht mehr garantieren, dass der Algorithmus eine optimale Lösung berechnet, da er mit gerundeten Nutzenwerten arbeitet.

Wenn wir uns nun allerdings vorstellen, dass wir den Skalierungsfaktor derart wählen, dass die Nutzenwerte durch die Rundung nur leicht verfälscht werden, so sollte die vom Algorithmus berechnete Lösung doch einen Nutzenwert haben, der „nahe“ am optimalen Nutzenwert liegt. Wir zeigen im Folgenden, dass dieser einfache Skalierungstrick tatsächlich funktioniert und zu einem FPTAS führt, wenn wir den Skalierungsfaktor auf geeignete Art und Weise in Abhängigkeit von den Parametern N, P und ϵ wählen.

Satz 3.48 *Das Rucksackproblem hat ein FPTAS.*

Beweis: Das FPTAS arbeitet folgendermaßen:

1. Skaliere die Nutzenwerte mit dem Faktor $\alpha = \frac{N}{\epsilon P}$ und runde ab, d.h. für $i \in \{1, \ldots, N\}$ setze $p'_i = \lfloor \alpha p_i \rfloor$.
2. Berechne eine optimale Rucksackbepackung $K \subseteq \{1, \ldots, N\}$ für die Nutzenwerte $p'_1, \ldots, p'_N$ mit dem dynamischen Programm aus Lemma 3.47.

Durch die Skalierung sind die Nutzenwerte nach oben beschränkt durch $P' = \lfloor \alpha P \rfloor = \lfloor \frac{N}{\epsilon} \rfloor$. Aus Lemma 3.47 ergibt sich somit unmittelbar eine Laufzeitschranke von $O(N^2 P') = O(N^3/\epsilon)$ uniformen Rechenschritten. Im logarithmischen Kostenmaß ist noch die Zeit zu berücksichtigen, die für die einzelnen Rechenschritte benötigt wird. Diese Zeit ist jedoch offensichtlich polynomiell in der Eingabelänge beschränkt, da es sich bei den Rechenschritten nur um einfache Additionen und Vergleiche handelt. Um zu zeigen, dass der Algorithmus ein FPTAS ist, müssen wir also nun nur noch den Approximationsfaktor nachweisen.

Sei $K^* \subseteq \{1, \ldots, N\}$ eine optimale Rucksackbepackung. Die durch den Algorithmus berechnete Rucksackbepackung bezeichnen wir mit K. Es gilt zu zeigen

$$p(K) \geq (1 - \epsilon)\, p(K^*) \ .$$

Für den Beweis skalieren wir die gerundeten Nutzenwerte virtuell wieder herauf, allerdings ohne dabei den Rundungsfehler rückgängig zu machen, d.h. wir setzen $p_i'' = p_i'/\alpha$. Die Rucksackbepackung K ist optimal für die Nutzenwerte p_i' und somit auch optimal für die Nutzenwerte p_i''. Bezogen auf die ursprünglichen Nutzenwerte p_i macht der Algorithmus allerdings möglicherweise für jedes Objekt einen Rundungsfehler: Objekte sehen weniger profitabel aus als sie eigentlich sind. Der Rundungsfehler für Objekt i lässt sich abschätzen durch

$$p_i - p_i'' \ = \ p_i - \frac{\lfloor \alpha p_i \rfloor}{\alpha} \ \leq \ p_i - \frac{\alpha p_i - 1}{\alpha} \ = \ \frac{1}{\alpha} \ .$$

Für eine Teilmenge $S \subseteq \{1, \ldots, N\}$ sei $p(S) = \sum_{i \in S} p_i$ und $p''(S) = \sum_{i \in S} p_i''$. Der Rundungsfehler für eine optimale Lösung K^* lässt sich entsprechend abschätzen durch

$$p(K^*) - p''(K^*) \ \leq \ \sum_{i \in K^*} \frac{1}{\alpha} \ \leq \ \frac{N}{\alpha} \ = \ \epsilon\, P \ \leq \ \epsilon\, p(K^*) \ ,$$

wobei die Ungleichung $p(K^*) \geq P$ daraus folgt, dass der optimale Nutzen $p(K^*)$ mindestens so groß ist wie der Nutzen des Objektes mit dem maximalen Nutzenwert P. Aus der obigen Ungleichung folgt $p''(K^*) \geq (1 - \epsilon)\, p(K^*)$. Nun ergibt sich

$$p(K) \ \geq \ p''(K) \ \geq \ p''(K^*) \geq (1 - \epsilon)\, p(K^*) \ .$$

□

Eine wichtige Grundlage für das FPTAS des Rucksackproblems ist das dynamische Programm mit Laufzeit $O(N^2 P)$. Ein anderes dynamisches Programm, das vielleicht aus der Vorlesung *Datenstrukturen und Algorithmen* bekannt ist, hat Laufzeit $O(N^2 W)$, wobei W das maximale Gewicht ist. Dieser Algorithmus ist also pseudopolynomiell in den Gewichten statt in den Nutzenwerten. Um aus diesem Algorithmus ein FPTAS zu gewinnen, müsste man die Gewichte runden. Das ist aber problematisch, denn

- rundet man die Gewichte ab, so berechnet der Algorithmus möglicherweise ein unzulässige Lösung;

- rundet man die Gewichte hingegen auf, so ignoriert der Algorithmus möglicherweise einige sehr gute zulässige Lösungen.

Der Skalierungstrick, den wir für das Rucksackproblem vorgeführt haben, funktioniert auch für einige andere Optimierungsprobleme, die einen pseudopolynomiellen Algorithmus haben. Um ein FPTAS zu erhalten, sollte die Laufzeit, wie erläutert, pseudopolynomiell in der Zielfunktion und nicht in den Nebenbedingungen sein. Ist dies gegeben, so ist im Einzelfall zu überprüfen, ob und wie man durch geschicktes Runden zu einem FPTAS gelangen kann.

3.10.2 Stark und schwach NP-harte Probleme

Bisher sind wir durchgängig davon ausgegangen, dass Zahlen in der Eingabe eines Problems binär kodiert werden. Die Spezifikation des Eingabeformats ist Teil der Problembeschreibung. Wenn wir das Eingabeformat ändern, so erhalten wir ein neues Problem. Das Rucksackproblem mit binärer Kodierung haben wir mit KP bezeichnet. Sei u-KP die Variante des Rucksackproblems, bei der wir annehmen, dass die Eingabezahlen unär kodiert vorliegen, d.h. wir kodieren eine Zahl $k \in \mathbb{N}$ durch k aufeinander folgende Einsen. Die Eingabelänge von k ist dann k und nicht mehr $\lceil \log(k+1) \rceil$ wie bei der binären Kodierung. Wir wissen KP ist NP-hart, hat aber einen Algorithmus mit pseudopolynomieller Laufzeitschranke. Die Laufzeit dieses Algorithmus ist polynomiell in der Größe der Eingabezahlen beschränkt, was nichts anderes bedeutet, als dass die Laufzeit polynomiell in der unären Eingabelänge ist. Der pseudopolynomielle Algorithmus für KP ist also aus Sicht von u-KP ein Polynomialzeitalgorithmus.

> Somit ist das Rucksackproblem bei unärer Kodierung in P, obwohl es in der üblichen binären Kodierung NP-hart ist.

Wie läßt sich dieses scheinbare Paradoxon erklären? – Beim Wechsel von der binären zur unären Kodierung vergrößert sich die Eingabelänge exponentiell. Da wir die Komplexität eines Problems relativ zur Eingabelänge beschreiben, hat sich durch das Aufblasen der Eingabe die Komplexität des Problems verringert, obwohl eine gegebene Instanz natürlich nicht dadurch einfacher zu lösen ist, dass wir die Eingabezahlen unär hinschreiben. Wir unterscheiden nun zwei Arten von NP-harten Problemen, nämlich einerseits Probleme, die auch bei unärer Kodierung noch NP-hart sind, und andererseits Probleme, wie das Rucksackproblem, die lediglich in binärer Kodierung NP-hart sind.

Definition 3.49 *Ein* NP*-hartes Problem, das bei unärer Kodierung einen polynomiellen Algorithmus hat, wird als* schwach NP-hart *bezeichnet. Ein Problem, das auch bei unärer Kodierung der Eingabezahlen* NP*-hart bleibt, wird als* stark NP-hart *bezeichnet.*

Das Rucksackproblem ist schwach NP-hart, da es einen pseudopolynomiellen Algorithmus hat. Die Probleme SUBSET-SUM und PARTITION sind ebenfalls schwach NP-hart, da sie mit einer einfachen Variante des dynamischen Programms für das Rucksackproblem ebenfalls in pseudopolynomieller Zeit gelöst werden können.

Bemerkung 3.50 *KP, SUBSET-SUM und PARTITION sind schwach NP-hart.*

Das Cliquenproblem ist ein Beispiel für ein Problem, welches trivialer Weise stark NP-hart ist, weil sich die Eingabelänge nur um einen konstanten Faktor verändert, wenn wir die einzige in der Eingabe vorkommende Zahl, nämlich die Cliquengröße $k \in \{1, \ldots, |V|\}$, unär statt binär kodieren. Korollar 3.41 belegt die NP-Härte von $\{1, 2\}$-TSP und zeigt somit, dass TSP auch dann noch NP-hart ist, wenn alle Eingabezahlen klein sind. TSP ist somit ebenfalls stark NP-hart, obwohl sich bei diesem Problem die Eingabelänge exponentiell vergrößern kann, wenn Zahlen unär statt binär kodiert werden. Auch vom Bin-Packing-Problem (BPP) weiss man, dass es stark NP-hart ist.

Bemerkung 3.51 *CLIQUE, TSP und BPP sind stark NP-hart.*

Unter der Hypothese $\mathsf{P} \neq \mathsf{NP}$ haben diese Probleme somit keinen Algorithmus mit einer Laufzeitschranke, die polynomiell in der unären Kodierungslänge ist, also keinen pseudopolynomiellen Algorithmus.

Wir zeigen nun, dass stark NP-harte Optimierungsprobleme nicht nur keinen pseudopolynomiellen Algorithmus sondern auch kein FPTAS haben können, es sei denn $\mathsf{P} = \mathsf{NP}$. Der folgende Satz macht einige kompliziert klingende technische Annahmen, die aber für typische Optimierungsprobleme ohne weiteres erfüllt sind. Als Faustformel sollten wir uns merken, unter der Hypothese $\mathsf{P} \neq \mathsf{NP}$ haben stark NP-harte Probleme kein FPTAS.

Satz 3.52 *Sei Π ein Optimierungsproblem mit ganzzahliger, nicht-negativer Zielfunktion. Sei p ein geeignetes Polynom. Für eine Eingabe I bezeichne $opt(I) \in \mathbb{N}$ den Wert einer optimalen Lösung und $n_u(I)$ die unäre Eingabelänge. Es gelte $opt(I) < p(n_u(I))$ für jede Eingabe I. Falls Π stark* NP*-hart ist, so hat Π kein FPTAS, es sei denn* $\mathsf{P} = \mathsf{NP}$.

Beweis: Zum Zwecke des Widerspruchs nehmen wir an, Π ist stark NP-hart und hat ein FPTAS. Aus dem FPTAS werden wir einen pseudopolynomiellen Algorithmus für Π konstruieren. Einen derartigen Algorithmus kann es unter der Hypothese $\mathsf{P} \neq \mathsf{NP}$ nicht geben. Also hat Π kein FPTAS oder $\mathsf{P} = \mathsf{NP}$.

Wir nehmen der Einfachheit halber an, Π ist ein Minimierungsproblem. Der Beweis für Maximierungsprobleme funktioniert analog. Sei A ein FPTAS für Π. Bei Eingabe I mit unärere Länge $n_u(I)$ setzen wir $\epsilon = 1/p(n_u(I))$. Die von A berechnete Lösung hat damit den Wert höchstens

$$(1+\epsilon)\, opt(I) \;<\; opt(I) + \epsilon\, p(n_u(I)) \;=\; opt(I) + 1 \;\; .$$

Somit berechnet A eine Lösung mit Zielfunktionswert $z \in [opt(I), opt(I)+1)$. Aus der Ganzzahligkeit der Zielfunktion folgt, $z = opt(I)$. A berechnet also eine optimale Lösung. Die Laufzeit von A ist polynomiell in $\frac{1}{\epsilon} = p(n_u(I))$ beschränkt, also polynomiell in $n_u(I)$, und somit ist A ein pseudopolynomieller Algorithmus. Damit ist der gewünschte Widerspruch gezeigt. □

Aus dem obigen Satz folgt beispielsweise, dass TSP kein FPTAS hat, da es stark NP-hart ist. Die Umkehrung von Satz 3.52 gilt nicht. Es gibt Probleme mit pseudopolynomiellen Algorithmen, für die es unter der Hypothese $\mathsf{P} \neq \mathsf{NP}$ beweisbar kein FPTAS gibt.

In der Vorlesung *Effiziente Algorithmen* wird das Thema Approximationsalgorithmen intensiver vorgestellt. Es wird z.B. gezeigt, dass TSP in seiner allgemeinen Form überhaupt keinen Approximationsalgorithmus mit polynomiell beschränkter Laufzeit hat, es sei denn $\mathsf{P} = \mathsf{NP}$. Nur wenn man bestimmte Forderungen an die Distanzmatrix stellt, also die Eingabe einschränkt, kann man das TSP-Problem bis auf einen konstanten Faktor approximieren.

Kapitel 4

Schlussbemerkungen

4.1 Literaturhinweise

Die folgenden Bücher eignen sich als zusätzliche Literatur und stehen in der Informatikbibliothek zur Verfügung.

- M.R. Garey, D.S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman and Company 1979.
- J.E. Hopcroft, R. Motwani, J.D. Ullman. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley 2001.
- J. Hromkovic. Theoretische Informatik. Teubner 2004.
- U. Schöning. Theoretische Informatik - kurzgefaßt. Spektrum Akademischer Verlag 2001.
- M. Sipser. Introduction to the Theory of Computation. PWS Publishing 1997.
- I. Wegener. Theoretische Informatik - eine algorithmenorientierte Einführung. Teubner Verlag 1999.
- I. Wegener. Kompendium Theoretische Informatik - Eine Ideensammlung. Teubner 1996.

4.2 Danksagung

Das vorliegende Skript ist über mehrere Semester entstanden. An der Erstellung des Skriptes haben insbesondere Helge Bals, Nadine Bergner, Dirk Bongartz, Marcel Ochel und Alexander Skopalik mitgewirkt.