UNIVERSITÄT BONN

Rheinische Friedrich-Wilhelms-Universität Bonn
Mathematisch-Naturwissenschaftliche Fakultät

Algorithmen und Berechnungskomplexität II

Skript SS 2018

Elmar Langetepe

Bonn, 18. Juli 2018

Anregungen der Studierenden sind uns sehr willkommen, bitte per Mail an
`elmar.langetepe@informatik.uni-bonn.de`.

Inhaltsverzeichnis

1 Einführung **1**

1.1 Modellierung und Kodierung von Problemen 2

2 Turingmaschinen **5**

2.1 Formale Definition 5

2.2 Berechnungen und Beispiele 7

2.3 Erweiterte Programmiertechniken 10

2.4 Mehrbandmaschinen und Simulation 11

2.5 Universelle Turingmaschine 15

3 Berechenbarkeit **19**

3.1 Entscheidbare und rekursiv aufzählbare Sprachen 19

3.2 Eigenschaften 20

3.3 Turingmaschinen-Eigenschaften als Sprachen 21

3.4 Existenz unentscheidbare Probleme 22

3.5 Konkrete unentscheidbare Probleme 24

3.6 Unentscheidbarkeit des Halteproblems 25

3.7 Reduktionen 25

3.8 Churchsche These und Registermaschine 28

3.9 Der Satz von Rice 33

3.10 Unentscheidbare Probleme ohne Selbstbezug 34

4 Komplexität und Klassen P und NP **41**

4.1 Entscheidungs- und Optimierungsprobleme 42

4.2 Zeitkomplexität von Turingmaschinen 43

4.3 Die Klasse P 45

4.4 Nichtdeterministische Turingmaschinen 46

4.5 Die Klasse NP 48

4.5.1 Beispiele für Probleme aus NP 49

4.6 Klasse NP und Zertifikate 52

4.6.1 P und NP 54

4.7 NP-Vollständigkeit . . . 54
4.7.1 Polynomielle Reduktion . . . 54
4.7.2 Definition der NP-Vollständigkeit . . . 55
4.7.3 Beispiele polynomieller Reduktionen . . . 56
4.7.4 Der Satz von Cook und Levin . . . 58
4.7.5 NP-Vollständigkeit arithmetischer Probleme . . . 64

5 Approximation und Randomisierung 67
5.1 Randomisierte Algorithmen . . . 67
5.2 Approximationen für NP-schwere Probleme . . . 71
5.2.1 Approximationsschemata PTAS und FPTAS . . . 72
5.2.2 Approximationsalgorithmus für TSP . . . 72
5.2.3 Ein Approximationsschemata des Rucksackproblem . . . 74
5.2.4 Approximation von Bin Packing . . . 76
5.3 Fixed Parameter Tractability . . . 77
5.3.1 Einfache Betrachtungen . . . 78
5.3.2 Verbesserungen durch Kernelisation . . . 79

Bibliography 81

Kapitel 1

Einführung

Im ersten Teil der Vorlesung *Algorithmen und Berechnungskomplexität* haben wir im vergangenen Semester verschiedene klassische algorithmische Fragestellungen (Sortieren, Flussprobleme, Graphenprobleme, Bin-Packing, Knapsack, etc.) untersucht, Lösungspläne dafür entworfen und die Effizienz dieser Pläne untersucht.

Die Lösungspläne beruhten dabei auf sehr unterschiedlichen Vorgehensweisen (Divide-and-Conquer, Inkrementelle Konstruktion, Greedy, Dynamische Programmierung, etc.). Diese Paradigmen haben wir für verschiedene Fragestellungen zur Anwendung gebracht.

Für die Analyse der Lösungen wurden dann die wesentlichen Berechnungsschritte asymptotisch abgeschätzt. Neben der formalen Beschreibung dieser Abschätzungen (O- und Ω-Notation) haben wir auch verschiedene Methoden der Analyse (Rekursionsgleichungen, Induktionsbeweise, strukturelle Eigenschaften, Rekursionstiefe, etc.) kennengelernt und angewendet. Außerdem haben wir gesehen, dass zur rechnergestützten Umsetzung der Lösungspläne eine geeignete, effiziente Repräsentation der Daten (Datenstrukturen) im Rechner notwendig ist. Ein wesentlicher Bestandteil der Vorlesung war die formale Analyse der vorgestellten Techniken und Strukturen. Dadurch konnten wir Gütegarantien abgeben. Die Formulierung der Algorithmen wurde im Wesentlichen durch Pseudocode dargestellt, der sich im Prinzip in jeder höheren Programmiersprache umsetzen lässt.

Neben diesen klassischen algorithmischen Prinzipien haben wir außerdem bereits eine formale Beschreibung von Sprachen vorgenommen. Formale Sprachen und Grammatiken werden beispielsweise benutzt, um Programmiersprachen syntaktisch korrekt zu beschreiben und semantisch richtig zu interpretieren. Wir hatten bereits festgestellt, welche Sprachen sich durch welche Automaten exakt beschreiben lassen oder genauer welche Sprachen durch welchen Automaten entschieden werden können. Damit wurde insgesamt bereits ein Schritt zu den in dieser Vorlesung zu behandelnden Fragestellungen getan: Gegeben ist ein formales Automaten-Konzept, welche Sprachen lassen sich entscheiden und für welche Sprachen ist das Konzept nicht mehr ausreichend? Diese Betrachtungsweisen werden uns in dieser Vorlesung sicher helfen.

Nicht alle vorgestellten Algorithmen waren wirklich effizient. Beispielsweise hat der Algorithmus von Ford und Fulkerson eine Laufzeit von $O(C \cdot m)$ benötigt oder beim Rucksackproblem ergab die Dynamische Programmierung eine Laufzeit von $O(n \cdot G)$ hierbei gehörten C (Summe aller Kantengewichte) und G (Gesamtkapazität des Rucksackes) eigentlich nicht zur Eingabegröße der m Kanten und n Gegenstände. Bei solchen Ergebnissen spricht man auch von *pseudopolynomieller* Laufzeit. Gibt es Algorithmen, die ohne eine solche Beschränkungen auskommen? Oder kann man zeigen, dass bestimmte Probleme wohl gar nicht besser gelöst werden können? Gibt es Probleme, die grundsätzlich gar nicht mit modernen Computern

gelöst werden können?

Es gibt erstaunlicherweise tatsächlich Problemstellungen die mit modernen Computern nicht gelöst werden können. Dazu gehört beispielsweise das sogenannte *Halteproblem*. Es lässt sich kein Java-Programm schreiben, dass als Eingabe ein beliebiges anderes Java-Programm erhält und für dieses Programm entscheidet, ob es terminiert oder nicht. Hier ist eine Grenze der *Berechenbarkeit* erreicht. Und zwar nicht, weil bislang noch Niemand einen passenden Trick gefunden hat, das Problem zu lösen. Nein, es lässt sich formal beweisen, dass es so ein Programm nicht geben kann. Es geht also in dieser Vorlesung um die Berechenbarkeit und auch um eine Charakterisierung von Schwere-Klassen der Problemstellungen je nach Komplexität.

Zunächst brauchen wir dazu ein formales Modell eines Rechners. Wir müssen sicher beschreiben können, was wir unter der Berechnung einer Problemlösung mit einem Computer verstehen wollen.

Dazu verwenden wir die nach Alan Turing (Britischer Mathematiker) benannten Turingmaschinen. Alles was im intuitiven Sinne (zum Beispiel durch ein Java-Programm) berechenbar ist, kann auch durch eine Turingmaschine mit entsprechender Kodierung berechnet werden und umgekehrt. Unser formales Konzept ist also gar keine Einschränkung, gibt uns lediglich sicheren Boden für unsere Beweise.

Wir beginnen mit der formalen Modellierung von Problemen durch geeignete Funktionen $f : \mathbb{N}^r \to \mathbb{N}$ auf den natürlichen Zahlen. Außerdem werden wird die Zahlen binär kodieren.

Das vorliegende Skript setzt sich aus verschiedenen Quellen zum Thema zusammen und versucht für diese eine einheitliche Bezeichnung durchzuhalten. Da das Skript erst im Laufe des Semesters entsteht, sind Hinweise der Studierenden über Fehler oder Ungereimtheiten sehr willkommen.

1.1 Modellierung und Kodierung von Problemen

Alle bisher von uns betrachteten Berechnungsprobleme lassen sich wie folgt klassifizieren. Gegeben ist eine Eingabe und dafür soll eine Ausgabe berechnet werden. Ein- und Ausgaben sind dabei im wesentlichen Wörter über einem festgelegten Alphabet Σ. Es ist für uns keine Einschränkung, wenn wir grundsätzlich das Alphabet $\Sigma = \{0, 1\}$ verwenden und die Ein- und Ausgaben darüber definieren. Zur Beschreibung der Mengen von Wörtern und deren Längen verweisen wir auf die vorhergehende Veranstaltung.

Anhand von ein paar Beispielen wollen wir zunächst sehen, wie wir uns die Kodierung der Ein- und Ausgabe eine Problems durch Binärzahlen vorstellen können. Dabei fangen wir sehr simpel an. Es kommt hier im wesentlichen nur darauf an, zu erkennen, dass sich jedes beliebiges Rechenproblem entsprechend kodieren lässt. Wie geschickt oder effizient die Kodierung ist, spielt zunächst keine Rolle.

Beispiel 1: Addition zweier Zahlen

Eingabe: Zwei binär kodierte Zahlen a und b

Ausgabe: Die binär kodierte Zahl $c = a + b$

Dieses Problem können wir direkt mit einer Funktion $f : \Sigma^* \to \Sigma^*$ beschreiben. Um bei der Eingabe a und b zu trennen, erweitern wir das Alphabet durch ein Symbol # und können somit die beiden Eingabewerte voneinander trennen. Mit $\text{bin}(a)$ bezeichnen wir die Binärdarstellung von a. Beispielsweise ist $\text{bin}(7) = 111$ und $\text{bin}(1) = 1$. Das Ergebnis für die Eingabe $\text{bin}(7)\#\text{bin}(1)$ ist dann $\text{bin}(8) = 1000$.

Klassischerweise werden in der Komplexitätstheorie zunächst *Entscheidungsprobleme* gelöst. Genauer wird dann als Ausgabe lediglich eine Ja/Nein Antwort benötigt, diese lässt sich durch 1 respektive 0 kodieren.

Beispiel 2: Einfache Rundtour in einem Graphen (Hamilton Circle)

Eingabe: Ein binär kodierter Graph $G = (V, E)$ und ein Startknoten v

Ausgabe: Das Zeichen 1, falls es einen Weg in V gibt, der bei v startet und endet und jeden Knoten von V exakt einmal besucht. Das Zeichen 0, falls ein solcher Weg nicht existiert.

Den Graphen können wir bequem durch das Alphabet $\Sigma = \{0, 1, \#\}$ beschreiben indem wir zum Beispiel zunächst die Zahl $\text{bin}(n)$ für die Anzahl der Knoten eintragen und dann die Werte einer $n \times n$ Matrix sukzessive (spalten- oder reihenweise) füllen. Eine 1 beschreibt dann für einen Eintrag (i, j), dass die Kante zwischen Knoten i und j exitiert. Entsprechend beschreibt eine 0, dass die Kante nicht in E liegt.

Beispiel 3: Tiefensuche in einem Graphen

Eingabe: Ein binär kodierter Graph $G = (V, E)$ und ein Startknoten v

Ausgabe: Die Folge von Knoten, die von v aus per Tiefensuche sukzessive erreicht werden.

Der Graph wird wie im vorherigen Beispiel kodiert, die Knoten werden durch ihre binären Identifier sukzessive angegeben.

Die in diesem Abschnitt verwendete allgemeine Kodierung werden wir im nächsten Abschnitt für die Beschreibung eines allgemeinen Rechners benutzen. Wir wissen jetzt, dass wir jedes Berechnungsproblem im Prinzip als eine Funktion $f : \Sigma^* \to \Sigma^*$ betrachten können. Genau genommen betrachten wir Funktionen vom Typ $f : \mathbb{N}^r \to \mathbb{N}^l$.

Kapitel 2

Turingmaschinen

Wir führen ein formales Modell eines Computers ein. Einen solchen Rechner kann man natürlich nicht im nächsten Laden kaufen. Das Modell der *Turingmaschine* wurde 1936 von Alan Turing als Modell für die Beschreibung von *Berechenbarkeit* entworfen und zwar bereits *vor* der Entwicklung erster elektronischer Rechenanlagen. Das Modell mag zunächst sehr eingeschränkt erscheinen, wird sich aber als sehr mächtig herausstellen.

2.1 Formale Definition

Als Speicher dient der einfachen Turingmaschine (TM) ein nach rechts offenes unendliches Band auf dem Zeichen in einzelnen Zellen geschrieben werden können. Diese Zeichen entstammen einem endlichen Bandalphabet Σ. Nach den obigen Vorüberlegungen fordern wir $\{0, 1, \#\} \subseteq \Sigma$. Außerdem ist es sinnvoll, die Zeichen \$ und $\sqcup$ für den Beginn des Bandes und für leere Zellen zu reservieren. Die Turingmaschine verwendet eine sogenannte *endliche Kontrolle*. Diese besteht aus einem Lese-/Schreibkopf für die Zellen des Bandes. Es lassen sich die Zeichen der Zellen auslesen und verändern. Der Kopf kann schrittweise über das Band bewegt werden. Die Kontrolle handelt deterministisch und die Maschine befindet sich zu jedem Zeitpunkt in einem Zustand q aus einer endlichen Zustandsmenge Q. Mittels einer Zustandsübergangsfunktion δ regelt die Kontrolle die Arbeit der Turingmaschine. Der Lese-/Schreibkopf kann in einem Schritt nach links oder rechts auf die Nachbarzellen bewegt werden oder auf der Zelle verweilen. Diese Verhalten kodieren wir entsprechend durch die Zahlen -1, 1 und 0 oder gelegentlich aus Bequemlichkeit auch durch L (links), R (rechts) und N (neutral).

Die Abbildung 2.1 zeigt die schematische Darstellung einer Turingmaschine während der Arbeit. Formal wird die Maschine wie folgt definiert.

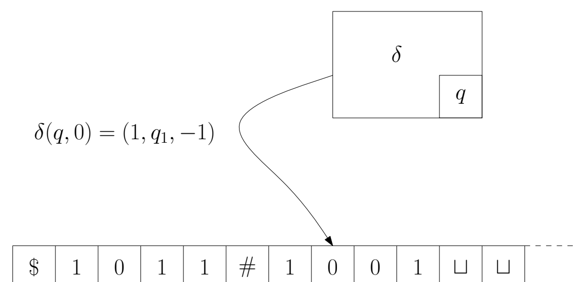

Abbildung 2.1: Die Darstellung einer Turingmaschine M.

Definition 1 *Eine deterministische Turingmaschine (DTM) M ist ein 5-Tupel der Form $M = (\Sigma, Q, \delta, q_0, F)$ mit:*

1. *Σ ist ein endliches Bandalphabet mit $\{0, 1, \#, \$, \sqcup\} \subseteq \Sigma$.*
2. *Q ist die endliche Zustandsmenge mit $Q \cap \Sigma = \emptyset$.*
3. *$q_0 \in Q$ ist der Startzustand.*
4. *$\delta : Q \times \Sigma \to Q \times \Sigma \times \{-1, 1, 0\}$ die Zustandsübergangsfunktion.*
5. *$F := \{q \in Q | \forall a \in \Sigma : \delta(q, a)$ ist undefiniert$\}$ ist die Menge der Endzustände.*

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

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

Die folgende Abbildung zeigt den Nachfolgezustand der DTM aus Abbildung 2.1 falls $\delta(q, 0) = (1, q_1, -1)$ gilt.

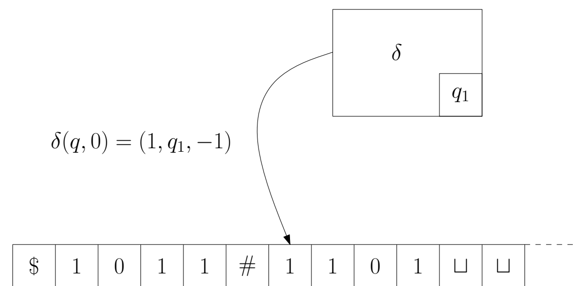

Abbildung 2.2: Die Turingmaschine M nach der Ausführung von $\delta(q, 0) = (1, q_1, -1)$.

Weiterhin legen wir zunächst folgende Konventionen fest. Wenn das Zeichen $ gelesen wird, dann darf die Maschine im nächsten Schritt den Lese-/Schreibkopf nur nach rechts bewegen und auch das Zeichen $ nicht überschreiben. Andere Zustandsübergänge für das Zeichen $ sind in δ verboten.

Der Kopf bewegt sich gemäß der oben angegebenen Aktion. Ein einzelner solcher Schritt wird als *Rechenschritt* bezeichnet. Die Anzahl der Rechenschritte, bis die Maschine in einem Endzustand landet, wird als *Laufzeit* bezeichnet. Die Anzahl der insgesamt aktiv verwendeten Zellen ergibt den benötigten *Speicherplatz*. Die Maschine *terminiert*, wenn ein Endzustand erreicht wird. Zu Beginn steht der Lese-/Schreibkopf auf dem ersten Zeichen hinter dem Bandanfang $.

Unter der *Konfiguration* einer DTM verstehen wir eine Momentaufnahme der Rechnung zu einem bestimmten Zeitpunkt. Die Konfiguration beschreibt die gesamte Situation der Maschine, den Zustand, die Position des Lese-/Schreibkopfes und die gesamte signifikante Bandinschrift.

Formal können wir die Konfiguration als String definieren und eine Schreibweise für Folgekonfigurationen festlegen.

1. Die Konfiguration einer DTM ist ein String $\alpha\, q\, \beta$ mit $q \in Q$, $\alpha, \beta \in \Sigma^*$. Auf den Band steht $\alpha\beta$ eingerahmt links vom Startzeichen \$ und rechts vom ersten $\sqcup$ Zeichen auf dem Band. Der Kopf steht auf dem ersten Element von β und der Zustand ist q.

2. Die Konfiguration $\alpha'\, q'\, \beta'$ ist die Nachfolgekonfiguration von $\alpha\, q\, \beta$, falls $\alpha'\, q'\, \beta'$ durch einen Rechenschritt aus $\alpha\, q\, \beta$ entsteht. Wir schreiben $\alpha\, q\, \beta \vdash \alpha'\, q'\, \beta'$.

3. Ist $\alpha''\, q''\, \beta''$ eine Nachfolgekonfiguration, die aus endlich vielen Schritten aus $\alpha\, q\, \beta$ entsteht so schreiben wir $\alpha\, q\, \beta \vdash^* \alpha''\, q''\, \beta''$ bzw. $\alpha\, q\, \beta \vdash^k \alpha''\, q''\, \beta''$, falls genau k Rechenschritte zwischen $\alpha\, q\, \beta$ und $\alpha''\, q''\, \beta''$ liegen.

In der Literatur finden sich alternative Beschreibungen von Turingmaschinen, bei denen Beispielsweise die Menge der Spezialbuchstaben $\{\sqcup, \#, \$\}$ strikt vom eigentlichen Alphabet getrennt wird. Dann wird zwischen Bandalphabet (Menge aller Zeichen auf dem Band) und dem Eingabealphabet (Alphabet für die explizite Eingabe) unterschieden. Wir machen diese Unterscheidung nicht, dafür haben wir aber andere Konventionen gewählt, beispielsweise, um das Überschreiben des Bandbeginns zu verhindern. Weiterhin finden sich gelegentlich Beschreibungen von Turingmaschinen mit einer expliziten Angabe von akzeptierenden und nicht-akzeptierenden Endzuständen. Wir verzichten auf diese Einteilung und werden das Akzeptieren oder Verwerfen explizit durch das Ergebnis der Berechnung festlegen. Außerdem werden Turingmaschinen gelegentlich auch durch ein beidseitiges unendliches Band definiert. Die Eingabe ist dann rechts und links mit jeweils einem $\sqcup$-Zeichen eingeschlossen.

Für den Leser sei hier erwähnt, dass die verschiedenen Konzepte allesamt äquivalent sind.

2.2 Berechnungen und Beispiele

Eine DTM, die für jede Eingabe terminiert berechnet eine totale Funktion $f_M : \Sigma^* \to \Sigma^*$. Das berechnete Ergebnis können wir uns beispielsweise wie folgt vorstellen. Die letzte Position des Lese-/Schreibkopfes ist der Anfang des Ergebnisses und das Ergebnis setzt sich nach rechts fort, bis das erste $\sqcup$-Zeichen erscheint. Steht der Lese-/Schreibkopf auf einem $\sqcup$-Zeichen, gilt auch das leere Wort ϵ als Antwort. Bei Terminierung befindet sich die Maschine in einem Endzustand.

Es kann passieren, dass die Maschine gar nicht terminiert, in diesen Fall ist das Ergebnis nicht definiert und die Funktion $f_M : \Sigma^* \to \Sigma^*$ ist partiell definiert. In diesem Fall können wir auf ein Symbol $\perp$ verweisen und machen die Funktion durch $f_M : \Sigma^* \to \Sigma^* \cup \{\perp\}$ total. Falls die DTM für eine Eingabe x nicht terminiert, setzen wir $f_M(x) = \perp$

Definition 2 *Eine Funktion $f : \Sigma^* \to \Sigma^* \cup \{\perp\}$ heißt* DTM-berechenbar *(oder auch* rekursiv*) falls es eine DTM M gibt mit $f_M = f$.*

Der alternative Begriff der Rekursivität geht darauf zurück, dass durch bestimmte einfache Regeln (ohne ein Rechnermodell) eine Klasse von Funktionen entworfen werden kann, die genau der Klasse der DTM-berechenbaren Funktionen entspricht. Diese Regeln beinhalten auch rekursives Verschachteln. Man kann dann formal zeigen, dass die Funktionenklassen identisch sind. Wir verzichten hier auf die Einführung dieser rekursiven Funktionen, wollen aber DTM-berechenbar und rekursiv als Synonym verwenden.

Bei Terminierung hält die Maschine in einem der Endzustände. Da wir es wie bereits erwähnt in der Komplexitätstheorie vorrangig mit Entscheidungsproblemen zu tun haben, wird die Ausgabe 1 für die Antwort Ja verwendet und auch als *Akzeptieren* bezeichnet. Wenn die

Maschine terminiert und das Ergebnis nicht 1 ist (in der Regel 0) sprechen wir dann von *Verwerfen* (Ablehnen) und meinen damit auch die Antwort Nein. Bei Terminierung wird in der Regel das Eingabeband gelöscht und die Antwort 1 oder 0 für Akzeptieren bzw. Verwerfen(Ablehnen) in einem Endzustand aufs Band geschrieben. Wie bereits erwähnt kann das Akzeptieren und Verwerfen(Ablehnen) auch explizit über entsprechende Zustände q_a und q_v geregelt werden.

Mit diesem Sprachgebrauch können wir festlegen, was es für eine Maschine heißt, dass sie eine Sprache $L \subseteq \Sigma^*$ akzeptiert bzw. sogar entscheidet.

Definition 3 *Sei $M = (\Sigma, Q, \delta, q_0, F)$ eine DTM.*

Die Menge aller von M akzeptierten $w \in \Sigma^*$ ist die von M akzeptierte Sprache L. *Wir schreiben auch $L = L(M)$.*

Die DTM M entscheidet *die von ihr akzeptierte Sprache $L(M)$, falls M alle Worte, die nicht in $L(M)$ liegen, ablehnt.*

Umgekehrt können wir nun definieren, was es heißt, dass eine Sprache L entscheidbar ist.

Definition 4 *Eine Sprache $L \subseteq \Sigma^*$ heißt* DTM-entscheidbar *(oder* rekursiv*), wenn es eine DTM M gibt, die die Sprache L entscheidet.*

Entscheidungsprobleme können nun prinzipiell als Sprachen L verstanden werden. Eine DTM löst dann ein Entscheidungsproblem L, wenn die Maschine die Sprache L entscheidet. Dazu betrachten wir nun ein paar explizite Beispiele.

Beispiel 1: Wir betrachten die Sprache $L = \{w | w1 \in \{0,1\}^*\}$ aller Binärwörter, die mit einer 1 enden. Das Entscheidungsproblem für diese Sprache L lösen wir mit einer DTM $M = \{\{0, 1, \#, \$, \sqcup\}, \{q_1, q_2, q_3\}, \delta, q_1, \{q_3\}\}$, wobei δ durch die folgende Tabelle beschrieben wird.

Q	0	1	$\sqcup$
q_1	$(q_1, 0, 1)$	$(q_2, 1, 1)$	$(q_3, 0, 0)$
q_2	$(q_1, 0, 1)$	$(q_2, 1, 1)$	$(q_3, 1, 0)$

Die Maschine liest sukzessive die Zeichen von links nach rechts und bleibt oder wechselt in den Zustand q_1, wenn das gelesene Zeichen eine 0 war. Ist das zuletzt gelesene Zeichen eine 1, so verweilt oder wechselt die Maschine in den Zustand q_2. Falls das Endymbol $\sqcup$ des Wortes gelesen wird, können wir anhand des vorherigen Zustandes erkennen, welcher Buchstabe zuletzt gelesen wurde und notieren die richtige Antwort (1 für Zeichen 1 bzw. Zustand q_2 und 0 für Zeichen 0 bzw. Zustand q_1). Ist das Band zu Beginn schon leer, hilft uns der Startzustand q_1.

Eine Wechsel von Konfigurationen können wir nun beispielsweise wie folgt beschreiben. Falls wir eine Eingabe 101 bearbeiten erhalten wir:

$$q_1 101 \vdash 1q_2 01 \vdash 10q_1 1 \vdash 101q_2 \vdash 101q_3 1$$

Sinnvoll ist es, sich die Vorgehensweise einer DTM anhand eines Stufenplanes oder einer Idee zurechtzulegen und danach entsprechend umzusetzen. Ein spezielles Zeichen kann man sich leicht durch einen entsprechenden Zustand merken.

Beispiel 2: Ein Palindrom ist ein Wort $w = w_1 w_2 \cdots w_{n-1} w_n$ ist ein Wort, dass von hinten nach vorne gelesen wieder w ergibt, also $w_n w_{n-1} \cdots w_2 w_1 = w$. Isnbesondere ist $w_i = w_{n-i+1}$ für $i = 1, \ldots, n$. Gesucht ist eine DTM M, die $L = \{w \in \{0,1\}^* | w \text{ ist Palindrom } \}$ entscheidet.

Unser Plan bzw. unsere Idee ist die Folgende. Hier haben wir die Zustände ebenfalls notiert.

1. Lese das erste Zeichen (rechts) in w im Zustand q_0. Falls es ein $\sqcup$ ist notieren wir das positive Ergebnis im Zustand q_6.
2. Sonst merke Dir dieses Zeichen in einem Zustand q_1 für 0 oder q_2 für 1 und ersetze das Zeichen durch ein $\sqcup$-Zeichen.
3. Gehe weiter nach rechts zum nächsten Zeichen, das ein $\sqcup$ ist. Überprüfe ob das vorherige Zeichen mit dem vorab gemerkten Zeichen übereinstimmt, Zustand q_3 für 0 und Zustand q_4 für 1. Ersetze es im positiven Fall durch einen $\sqcup$ oder notiere das negative Ergebnis in Zustand q_6. Falls gar kein Zeichen mehr da war (ungerade Anzahl) notiere das positive Ergebnis in Zustand q_6.
4. Im positiven Fall gehe durch Zustand q_5 nach links zum ersten $\sqcup$ Zeichen und Zustand q_0. Wiederhole Schritt 1.

Wir benötigen eine DTM $M = \{\{0, 1, \#, \$, \sqcup\}, \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}, \delta, q_0, \{q_6\}\}$ und verwenden eine Zustandsübergangsfunktion δ wie folgt:

Q	0	1	$\sqcup$
q_0	$(q_1, \sqcup, R)$	$(q_2, \sqcup, R)$	$(q_6, 1, N)$
q_1	$(q_1, 0, R)$	$(q_1, 1, R)$	$(q_3, \sqcup, L)$
q_2	$(q_2, 0, R)$	$(q_2, 1, R)$	$(q_4, \sqcup, L)$
q_3	$(q_5, \sqcup, L)$	$(q_6, 0, N)$	$(q_6, 1, N)$
q_4	$(q_6, 0, N)$	$(q_5, \sqcup, L)$	$(q_6, 1, N)$
q_5	$(q_5, 0, L)$	$(q_5, 1, L)$	$(q_0, \sqcup, R)$

Beispielsweise geht der Übergang von 0100 (kein Palindrom) wie folgt. Links wird das Wort immer von \$ begrenzt, rechts vom ersten $\sqcup$ hinter dem Restwort.

$q_0 0100 \vdash \sqcup q_1 100 \vdash \sqcup 1 q_1 00 \vdash \sqcup 10 q_1 0 \vdash \sqcup 100 q_1 \vdash \sqcup 10 q_3 0 \vdash \sqcup 1 q_5 0 \vdash \sqcup q_5 10 \vdash q_5 \sqcup 10 \vdash \sqcup q_0 10 \vdash \sqcup\sqcup q_2 0 \vdash \sqcup\sqcup 0 q_2 \vdash \sqcup\sqcup q_4 0 \vdash \sqcup\sqcup q_6 0$

Manchmal ist es sinnvoll, beim Stufenplan die Zustände zunächst wegzulassen, um die allgemeine Idee zu verdeutlichen. Das ganze ist sehr ähnlich zum Programmieren. Zunächst überlegt man sich eine allgemeinen Vorgehensweise, die dann durch die zur Verfügung stehenden Konstrukte umgesetzt werden.

Beispiel 3: Wir wollen eine DTM konstruieren, die die Sprache $L = \{0^n 1^n | n \geq 1\}$ entscheidet. Wir lösen das Problem in zwei Schritten:

1. Stelle fest, ob die Eingabe von der Form $0^i 1^j$ für $i \geq 0$ und $j \geq 1$ ist.
2. Teste dann ob auch $i = j$ gilt.

Für die erste Phase benutzen wir die Zustände q_0, q_1, q_2 und q_3, wobei q_2 den erfolgreichen Übergang in Phase 2. beschreibt und q_3 als (negativer) Endzustand gedacht ist. Der Übergang sieht wie folgt aus:

Q	0	1	$\sqcup$
q_0	$(q_0, 0, R)$	$(q_1, 1, R)$	$(q_3, 0, N)$
q_1	$(q_3, 0, N)$	$(q_1, 1, R)$	$(q_2, \sqcup, L)$

Die Maschine läuft in dieser Phase von links nach rechts über das Band und verbleibt in Zustand q_0 solange Nullen gelesen werden. Wenn dieses Einlesen mit einem Blank endet wird das Wort verworfen. Endet die Phase mit einer Eins, geht es in den Zustand q_1. Wird hier eine Null gelesen, können wir das Wort ebenfalls verwerfen. Erst wenn wir nach dem Lesen lauter Einser an einem Blank enden, ist klar, dass das Wort die gewünschte Form 0^i1^j für $i \geq 0$ und $j \geq 1$ hat und wir wechseln in die zweite Phase mit Zustand q_2.

Für die zweite Phase betrachten wir den folgenden Übergang. Die Maschine steht im Zustand q_2 zu Beginn mit dem Kopf rechts und muss nun die Anzahl der Einsen mit der Anzahl der Nullen vergleichen. Analog zum vorherigen Beispiel löschen wir eine Eins, gehen zum linken Rand und löschen dort eine Null. Dann gehen wir wieder nach rechts und starten mit der Prozedur erneut. Falls irgendwann das Wort leer ist, sind wir fertig. Sonst fehlt uns entweder eine Eins auf der rechten Seite oder eine Null auf der linken Seite. In diesen Fällen verwerfen wir das Wort.

Q	0	1	$\sqcup$	\$
q_2	$(q_3, 0, N)$	$(q_4, \sqcup, L)$	$(q_3, 0, N)$	
q_4	$(q_4, 0, L)$	$(q_4, 1, L)$	$(q_5, \sqcup, R)$	$(q_5, \$, R)$
q_5	$(q_6, \sqcup, R)$	$(q_3, 0, N)$	$(q_3, 0, N)$	
q_6	$(q_7, 0, R)$	$(q_7, 1, R)$	$(q_3, 1, N)$	
q_7	$(q_7, 0, R)$	$(q_7, 1, R)$	$(q_2, \sqcup, L)$	

2.3 Erweiterte Programmiertechniken

Wie wir bereits an den vorangegangenen Beispielen gesehen haben, kann die konkrete Beschreibung einer Turingmaschine und die Argumentation über ihre korrekte Arbeitsweise sehr mühsam sein. Im Folgenden wollen wir uns klar machen, dass wir mit dem Konzept der Turingmaschine sehr leicht auch Strukturen höherer Programmiersprachen entwerfen können.

Wir hatten bereits festgestellt, dass wir uns durch das Verwenden spezieller Zustände etwas *merken* können. Zum Beispiel ob eine 1 oder eine 0 gelesen wurde, siehe Beispiel 2. Im Prinzip haben wir dabei eine binäre **Variable** mit einem Wert belegt. Das könnten wir auch direkt machen. Wir können eine Zustandsraum Q leicht zu einen Zustandsraum $Q' = Q \times \{0, 1\}$ erweitern und haben dann stets eine binäre Variable zur Verfügung. Im Beispiel 2. hätten wir die Zustände $(q_1, 1)$ uns $(q_1, 0)$ statt q_1 und q_2 verwenden dürfen.

Etwas allgemeiner könnten wir sogar einen **endlichen Zähler** mit Werten zwischen 0 und k durch $Q' = Q \times \{0, \dots, k\}$ verwenden. Hierbei muss k allerdings eine feste Konstante sein, die nicht von der Eingabegröße abhängt. Es gibt nämlich in der DTM nur endlich viele Zustände.

Analog können wir den folgenden Trick auf dem Eingabeband durch die Erweiterung des Bandalphabets verwenden. Statt eines einzelnen Zeichens verwenden wir ein Zeichentupel das vom Lese-/Schreibkopf vollständig gelesen werden kann. Dadurch hat unsere Maschine quasi mehrere Spuren die gleichzeitig gelesen werden. In der Bandzelle steht also ein k-Tupel aus Zeichen.

Beispiel für mehrere Spuren: Wollen wir beispielsweise zwei Zahlen miteinander addieren, so haben in Abschnitt 1.1 bereits erwähnt, dass wir die Zahlen b_1 und b_2 binärkodiert durch $\text{bin}(b_1)\#\text{bin}(b_2)$ auf das Band schreiben können. Zur Berechnung wollen wir drei Spuren verwenden und deshalb Verwenden wir das Alphabet

$$\Sigma = \left\{ 0, 1, \#, \$, \begin{pmatrix}0\\0\\0\end{pmatrix}, \begin{pmatrix}0\\0\\1\end{pmatrix}, \begin{pmatrix}0\\1\\0\end{pmatrix}, \ldots, \begin{pmatrix}1\\1\\1\end{pmatrix} \right\}$$

mit insgesamt 12 *Zeichen*. Für das Aufaddieren von 1101 und 1001 schreiben wir nun stattdessen

$$\$ \begin{pmatrix}0\\0\\0\end{pmatrix} \begin{pmatrix}1\\1\\0\end{pmatrix} \begin{pmatrix}1\\0\\0\end{pmatrix} \begin{pmatrix}0\\0\\0\end{pmatrix} \begin{pmatrix}1\\1\\0\end{pmatrix} \sqcup$$

auf das Band und das Ergebnis soll am Ende in der dritten Spur eingetragen werden. Die Berechnung wird durch das klassische Additionsverfahren mit Überträgen realisiert. Die Überträge können wir uns beispielsweise in Zuständen q_j und q_n merken. Zunächst bewegt die Maschine den Lese-/Schreibkopf auf das am weitesten rechts liegende Element. Dann realisieren wir den Übergang zum Beispiel wie folgt:

$$\begin{aligned}
\begin{pmatrix}0\\0\\0\end{pmatrix} \begin{pmatrix}1\\1\\0\end{pmatrix} \begin{pmatrix}1\\0\\0\end{pmatrix} \begin{pmatrix}0\\0\\0\end{pmatrix} q_n \begin{pmatrix}1\\1\\0\end{pmatrix} &\vdash \begin{pmatrix}0\\0\\0\end{pmatrix} \begin{pmatrix}1\\1\\0\end{pmatrix} \begin{pmatrix}1\\0\\0\end{pmatrix} q_j \begin{pmatrix}0\\0\\0\end{pmatrix} \begin{pmatrix}1\\1\\0\end{pmatrix} \\
&\vdash \begin{pmatrix}0\\0\\0\end{pmatrix} \begin{pmatrix}1\\1\\0\end{pmatrix} q_n \begin{pmatrix}1\\0\\0\end{pmatrix} \begin{pmatrix}0\\0\\1\end{pmatrix} \begin{pmatrix}1\\1\\0\end{pmatrix}
\end{aligned}$$

Als Ergebnis erwarten wir dann zunächst

$$q \begin{pmatrix}0\\0\\1\end{pmatrix} \begin{pmatrix}1\\1\\0\end{pmatrix} \begin{pmatrix}1\\0\\1\end{pmatrix} \begin{pmatrix}0\\0\\1\end{pmatrix} \begin{pmatrix}1\\1\\0\end{pmatrix}$$

für einen Zwischenzustand q.

Möchte man dann das Ergebnis wieder rechts von einem Endzustand schreiben, muss die letzte Zeile noch entsprechend auf das Band übertragen werden. Das bereitet aber prinzipiell keine Schwierigkeiten.

Wir können also simple Arithmetik auf der Maschine bequem durch mehrere Spuren realisieren. Andere Konstrukte aus höheren Programmiersprachen lassen sich ebenfalls nachbilden. Beispielsweise kann für eine *FOR*-**Schleife** die Anzahl der Schleifendurchläufe auf die erste Spur geschrieben werden und diese Zahl wird dann sukzessive inkrementiert. Oder aber, falls **Unterprogramme** aufgerufen werden sollen, so kann ebenfalls dafür eine Spur des Bandes reserviert werden, durch den ein Prozedurstack realisiert werden kann.

Insgesamt halten wir fest, dass wir mit Turing-Maschinen auch bekannte Algorithmen wie das **Sortieren von Zahlen** oder das **Traversieren von Graphen** im Prinzip umsetzen können.

2.4 Mehrbandmaschinen und Simulation

Anschließend wollen wir die Turingmaschinen etwas erweitern und erlauben mehrere Lese-/Schreibköpfe, die sich sogar jeweils in unterschiedliche Richtungen bewegen dürfen. Dadurch

lassen sich viele Programme einfacher formulieren. Beispielsweise können einzelne Bänder für die Eingabe oder die Ausgabe reserviert werden. Es wird sich zeigen, dass diese k-Band Turingmaschinen nicht mächtiger sind als eine einfache 1-Band Maschine.

Definition 5 *Eine deterministische k-Band Turingmaschine (k-Band DTM) M ist eine Verallgemeinerung der DTM mit k statt einem Band. Jedes Band hat einen unabhängigen Lese-/Schreibkopf. Das erste Band wird als* Eingabeband *verwendet. Die verbleibenden $k-1$ Bänder sind zu Beginn mit $\sqcup$-Zeichen gefüllt. Die Köpfe stehen zu Beginn jeweils auf der ersten Zelle nach dem \$ Zeichen.*

Die Zustandsübergangsfunktion ist nun von der Form:

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

Die Funktionsweise der Maschine sowie die Laufzeit und der Speicherplatzbedarf sind analog.

Auch bei der k-Band Maschine hat man etwas Gestaltungsspielraum. Häufig wird verlangt, dass das Ergebnis der Berechnung am Ende auch auf das erste Band geschrieben wird, die anderen Bänder *gelöscht* werden und die Köpfe in die Ausgangsposition zurückkehren. Im folgenden Beispiel wollen wir das so realisieren.

Beispiel 2-Band DTM: Wir wollen die Nachfolgefunktion $f(x) = x + 1$ für alle $x \in \mathbb{N}$ berechnen. Die Idee der 2-Band Maschine M ist die Folgende:

- Kopiere Band 1 auf Band 2. Lösche Band 1.
- Addiere binär 1 zum Inhalt von Band 2 von rechts nach links. Übertrag durch q_j, kein Übertrag durch q_n.
- Falls am Ende q_j, schreibe 1 auf Band 1.
- Hänge Inhalt von Band 2 an Band 1. Lösche Band 2.
- Bringe die Köpfe nach vorne.

Wir benutzen zur konkreten Beschreibung eine etwas andere Tabellenschreibweise die selbsterklärend ist.

Band 1 auf Band 2 kopieren und Band 1 löschen.

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

Kopf 1 nach links, Kopf 2 bleibt rechts.

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

Addiere 1 zu Band 2, q_j Übertrag, q_n kein Übertrag.

q	z_1	z_2	q'	z'_1	z'_2	m_1	m_2
q_j	⊔	1	q_j	⊔	0	0	−1
q_j	⊔	0	q_n	⊔	1	0	−1
q_n	⊔	0/1	q_n	⊔	0/1	0	−1
q_n	⊔	$	q_4	⊔	$	0	−1

Falls am Ende q_j (dann Input $111\dots1$), schreibe führende 1 auf Band 1.

q	z_1	z_2	q'	z'_1	z'_2	m_1	m_2
q_j	⊔	$	q_4	1	$	1	1

Hänge Band 2 an Band 1 an. Lösche Band 2.

q	z_1	z_2	q'	z'_1	z'_2	m_1	m_2
q_4	⊔	0/1	q_4	0/1	⊔	1	1
q_4	⊔	⊔	q_5	⊔	⊔	−1	−1

Kopf 1 nach links und Kopf 2 nach links.

q	z_1	z_2	q'	z'_1	z'_2	m_1	m_2
q_5	0/1	⊔	q_5	0/1	⊔	−1	0
q_5	$	⊔	q_6	$	⊔	1	0
q_6	0/1	⊔	q_6	0/1	⊔	0	−1
q_6	0/1	$	q_7	0/1	$	0	1

Da wir nun die Funktionsweisen der jeweiligen Maschinen kennen, wollen wir zeigen, dass die Maschinen die gleichen Funktionen berechnen können. Dafür verwenden wir den Begriff der *Simulation*. Wir *simulieren* die Vorgehensweise einer k-Band Maschine auf einer 1-Band Maschine. Das heißt, jeder Rechenschritt der k-Band Maschine wird auf der 1-Band Maschine nachgebildet. Die Laufzeit ändert sich dadurch allerdings entsprechend.

Theorem 6 *Eine k-Band DTM M, die mit Rechenzeit $t(n)$ und Platz $s(n)$ auskommt, kann durch eine 1-Band DTM M' mit Rechenzeit $O(t^2(n))$ und Speicherplatz $O(s(n))$ simuliert werden.*

Wird die Sprache $L \subseteq \Sigma^*$ von einer k-Band DTM entschieden (akzeptiert), so gibt es auch eine 1-Band DTM, die die Sprache L entscheidet (akzeptiert).

Beweis. Die DTM M' verwendet statt k Bänder $2k$ Spuren, derart, dass

- die ungeraden Spuren $1, 3, \dots, 2k-1$ enthalten den Inhalt der Bänder $1, \dots, k$ von M.
- in den geraden Spuren $2, 4, \dots, 2k$ wird die Kopfposition von M auf den Bändern $1, \dots, k$ durch das Zeichen # festgehalten.

Die Abbildung 2.3 zeigt eine Momentaufnahme der Simulation einer 3-Band Maschine durch eine 1-Band Maschine mit 6 Spuren. Wir nutzen aus, dass k eine feste Konstante ist.

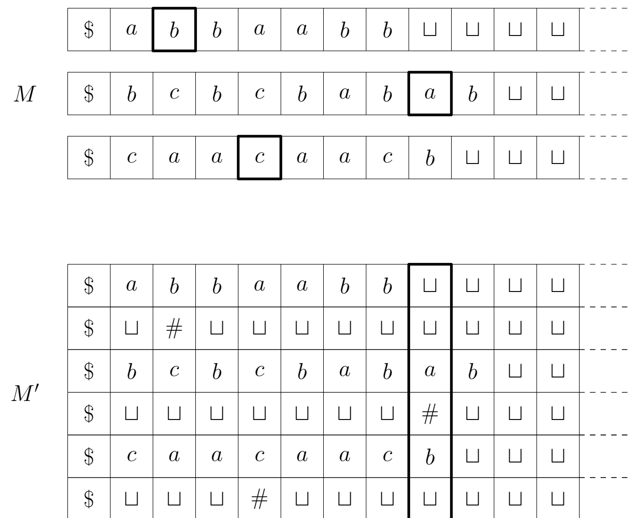

Abbildung 2.3: Die Maschine M' simuliert die Maschine M. Falls M sich im Zustand q befindet, befindet sich M' nach dem Durchlauf zum rechtesten $\#$ beispielsweise im Zustand $(q, 3, (b, a, c))$ und simuliert dann den Übergang von M durch einen Durchlauf nach links. Fall $\delta(q, (b, a, c)) = (q', (b, a, a), (0, 0, 1))$ gilt, muss die Übergangsfunktion δ' von M' sicherstellen, dass beim Durchlauf nach links auf der fünften Spur der Buchstabe c durch a überschrieben und dann das $\#$-Zeichen auf der sechsten Spur nach rechts verschoben wird. Das kann beispielsweise durch einen lokalen Übergang $\delta'((q', 2, (b, a, c)), (\cdot, \cdot, \cdot, \cdot, c, \#)) = ((q'_{(r,6)}, 1, (b, a, c)), (\cdot, \cdot, \cdot, \cdot, a, \sqcup), 1)$ realisiert werden, wobei $q'_{(r,6)}$ festhält, das gleich nur ein Blank auf der sechsten Spur durch ein $\#$ ersetzt wird und dann nach links weitergegangen wird, wobei dann wieder in den Zustand $(q', 1, (b, a, c))$ gewechselt wird, der auf das letzte $\sqcup$-Zeichen wartet.

Simulation eines Rechenschrittes:

1. Der Kopf vom M' steht auf dem linkesten #-Zeichen. Der Zustand von M ist bekannt.
2. Die Maschine läuft zum rechtesten #-Zeichen. Wir merken uns die Anzahl der überschrittenen #-Zeichen durch eine Zustandserweiterung (Zähler, $Q' = Q \times \{1, \ldots, k\}$). Außerdem werden sich die dabei gelesenen k Zeichen und das jeweilige Band durch eine weitere Zustandserweiterung ($\cdots \times \Sigma^k$) gemerkt, siehe dazu auch die Abbildung 2.3.
3. Am rechtesten # angekommen, kennt die Maschine M' den Bandinhalt und den Zustand der Maschine M. Die Übergangsfuktion von M kann dann ausgewertet werden. Die Maschine M' *weiß* nun bereits, wie die Spuren zu ändern sind. Die Übergangsfunktion vom M ist eindeutig.
4. Der Kopf der Maschine M' läuft zurück und manipuliert dabei lokal die Werte auf den Bändern gemäß der Übergangsfunktion von M. Dabei werden auch die # Positionen entsprechend verschoben. Das wird in der Übergangsfunktion von M' festgelegt. Lokal werden dazu (für die Verschiebung der #-Zeichen) nur konstant viele Hin-und-Her Bewegungen gebraucht, die durch eine konstante Anzahl von Zustände realisiert werden. Von rechts kommend wird ein Zeichen # mit seiner Position gelesen und bevor wir weiter nach links gehen muss das Zeichen ggf. noch nach rechts verschoben werden. Siehe dazu auch die Abbildung 2.3 und das in der Bildunterschrift angegebene Beispiel.
5. Am Ende steht der Kopf der Maschine M' wieder am linkesten #-Zeichen und kennt den aktuellen Zustand von M. Der nächste Schritt kann simuliert werden.

Alles, was sich die Maschine M' hierbei merken musste, hängt nicht von der Länge des aktuell vorhandenen Bandinhaltes vom M ab, sondern nur von k. Also können wir uns alle $O(k)$ Informationen in einem Zustand von M' merken. Die Zustandsmenge erhöht sich insgesamt auch nur endlich, das ist aber kein Effizienzkriterium an dieser Stelle. Die Anzahl der Zustände hängt aber nicht von der Anzahl der Rechenschritte ab. Für einen in M verwendete Zustand müssen wir einige zusätzliche Zustände generieren, deren Anzahl von k und Σ abhängen. Das gilt auch für die oben beschriebenen lokalen Hin-und-Her Bewegungen.

Für die Laufzeitanalyse ist eigentlich nur wichtig, wie viele Bewegungen der Kopf der 1-Band Maschine in einem Rechenschritt macht. Im wesentlichen läuft der Kopf (abgesehen von maximal k vielen lokalen Hin-und-Her Läufen) zweimal zwischen den beiden am weitesten entfernten #-Zeichen hin und her. Falls d die Anzahl der Bandzellen zwischen den am weitesten entferneten #-Zeichen ist, beträgt die Laufzeit eines Schrittes vom M' somit $O(d)$.

Wie weit können die #-Zeichen also auseinander liegen? Nach t Rechenschritten können sich je zwei #-Zeichen maximal $2t$ voneinander entfernt haben. Jede Markierung kann sich in einem Schritt nur um eine Zelle nach links oder rechts bewegen. Der Abstand ist also direkt mit der Laufzeit verbunden. Für $t(n)$ Schritte der Maschine M kann die Machine maximal $t(n)$-mal $C \times t(n)$ Arbeitsschritte gemacht haben. Folglich hat M' die Laufzeit $O(t^2(n))$.

Die Maschine M' benötigt nicht mehr als $O(s(n))$ Bandzellen, da für jede Bandzelle, die M' besucht, auch mindestens eine Bandzelle aus M gehört, man bedenke, dass k konstant ist. □

2.5 Universelle Turingmaschine

Die Turingmaschinen die wir bislang betrachtet haben wurden stets zur speziellen Lösung genau eines festen Problems formuliert. Diese spezialisierte Vorgehensweise entspricht der Historie. Zunächst wurden im Umfeld von Alan Turing durch den Einsatz der ersten Rechner

sehr spezielle (militärische) Aufgaben gelöst. Beispielsweise war Turing an der Dechiffrierung des ENIGMA-Codes beteiligt. Auch die ersten Maschinen von Zuse in Deutschland bearbeiteten spezielle (teilweise militärische) Problemstellungen.

Heutige Rechnern lassen sich beliebig programmieren. Die Eingabe besteht somit aus einem Programm (eine spezielle Maschine) und einer Eingabe für das die Maschine. Es lassen sich beliebige Maschinen und Eingaben verarbeiten. In diesem Sinne sind heutige Rechner universell.

Wir wollen nun auch diese universellen Rechner durch Turingmaschinen formalisieren. Dazu benötigen wir eine eindeutige Beschreibung einer einzelnen speziellen Maschine mit seiner Eingabe.

Genau genommen wird unsere universelle Turingmaschine U die Arbeit einer beliebigen Maschine M und mit einer Eingabe w simulieren.

Wir betrachten ohne Einschränkung eine 1-Band Turingmaschine mit n Zuständen $\{q_1, q_2, \ldots, q_n\}$ wobei q_1 der Startzustand und q_n der Endzustand ist. Insgesamt gibt es also $n-1$ signifikante Zustände.

Zunächst werden wir einen binären String $\langle M\rangle w$ kodieren, der die Maschine M und die Eingabe $w \in \{0,1\}^*$ eindeutig charakterisiert. Diesen Vorgang nennt man Gödelnummerierung.

Definition 7 *Als Gödelnummerierung bezeichnen wir eine injektive Abbildung aus der Menge der DTMs in die Menge* $\{0,1\}^*$.

Wir stellen eine Gödelnummerierung vor und müssen dabei im wesentlichen die Übergangsfunktion kodieren. Implizit ist dadurch auch die Anzahl der Zustände gegeben.

Wir beschränken uns auf die 5 Zeichen $x_1 = 0$, $x_2 = 1$, $x_3 = \sqcup$ $x_4 = \$$ und $x_5 = \#$ und die Bewegungen $b_1 = 1$, $b_2 = -1$ und $b_3 = 0$. Für n Zustände $\{q_1, q_2, \ldots, q_n\}$ sei q_1 der Startzustand und q_n der einzige Endzustand. Insgesamt gibt es also $n-1$ signifikante Zustände.

Der Kodierungsstring startet und endet mit der Ziffernfolge 111, die Übergänge werden durch Strings 11 getrennt. Jeder Übergang $\delta(q_i, x_j) = (q_k, x_l, b_m)$ wird durch den Binärstring $0^i\ 1\ 0^j\ 1\ 0^k\ 1\ 0^l\ 1\ 0^m$ kodiert. Die Übergänge werden durchnummeriert und code(i) beschreibt die Binärkodierung des i-ten Übergangs. Eine Turingmaschine M wird dann durch

$$\langle M\rangle := 111\ \text{code}(1)\ 11\ \text{code}(2)\ 11 \cdots 11\ \text{code}(s)\ 111$$

eindeutig kodiert.

Beispiel: Die Sprache $L = \{w | w1 \in \{0,1\}^*\}$ aller Binärwörter, die mit einer 1 enden wurde durch die DTM $M = \{\{0, 1, \#, \$, \sqcup\}, \{q_1, q_2, q_3\}, \delta, q_1, \{q_3\}\}$ mit Übergängen δ durch die folgende Tabelle beschrieben.

Q	0	1	$\sqcup$
q_1	$(q_1, 0, 1)$	$(q_2, 1, 1)$	$(q_3, 0, 0)$
q_2	$(q_1, 0, 1)$	$(q_2, 1, 1)$	$(q_3, 1, 0)$

In der Tabelle wurden nur die relevanten Übergänge dargestellt. Da die Maschine zu Beginn das erste Zeichen rechts vom \$-Zeichen liest, wird das Zeichen \$ nie gelesen.

Die Kodierung der Übergänge geht nun wie folgt:

Nummer i des Übergangs	Übergang i	code(i)
1	$\delta(q_1, 0) = (q_1, 0, 1)$	0 1 0 1 0 1 0 1 0
2	$\delta(q_1, 1) = (q_2, 1, 1)$	0 1 00 1 00 1 00 1 0
3	$\delta(q_1, \sqcup) = (q_3, 0, 0)$	0 1 000 1 000 1 0 1 000
4	$\delta(q_2, 0) = (q_1, 0, 1)$	00 1 0 1 0 1 0 1 0
5	$\delta(q_2, 1) = (q_2, 1, 1)$	00 1 00 1 00 1 00 1 0
3	$\delta(q_2, \sqcup) = (q_3, 1, 0)$	00 1 000 1 000 1 00 1 000

Die Maschine wird nun durch

$$\begin{aligned}\langle M\rangle \;=\; & 111\;\; 0\,1\,0\,1\,0\,1\,0\,1\,0\;\; 11\;\; 0\,1\,00\,1\,00\,1\,00\,1\,0\;\; 11\\ & 0\,1\,000\,1\,000\,1\,0\,1\,000\;\; 11\;\; 00\,1\,0\,1\,0\,1\,0\,1\,0\;\; 11\\ & 00\,1\,00\,1\,00\,1\,00\,1\,0\;\; 11\;\; 00\,1\,000\,1\,000\,1\,00\,1\,000\;\; 111\end{aligned}$$

kodiert.

Übungsaufgabe: Zeigen Sie, dass es sich bei der oben angegebenen Kodierung um eine Gödelnummerierung handelt.

Definition 8 *Eine Turingmaschine M_U heißt* universell*, falls für jede* 1*-Band DTM M und für jedes $w \in \{0,1\}^*$ gilt:*

- *M_U gestartet mit $\langle M\rangle w$ hält genau dann, wenn M gestartet mit w hält.*
- *Falls M gestartet mit w hält, dann berechnet M_U gestartet mit $\langle M\rangle w$ die gleiche Ausgabe wie M gestartet mit w.*

Theorem 9 *Es existiert eine universelle Turingmaschine M_U.*

Beweis. Wir geben M_U als 3-Band Turingmaschine an, die die Arbeitsweise jeder beliebigen Maschine M mit Eingabe w simuliert.

Dazu wird zunächst überprüft, dass es sich bei $\langle M\rangle w$ um eine korrekte Turingmaschine mit korrekter Eingabe handelt. Der String $\langle M\rangle w$ steht auf dem ersten Band und wird mit der Turingmaschine überprüft.

- Zunächst wird geprüft ob zwei Teilfolgen 111 existieren, die ein Wort w' einschließen, dass keine Teilfolge 111 enthält und mit 0 beginnt und endet. Dann wird geprüft, ob nach dem zweiten Vorkommen von 111 ein Wort $w \in \{0,1\}^*$ steht. Falls keine solche zwei Substrings existieren oder das Wort w nicht die entsprechende Form hat, wird das gesamte Wort verworfen.
- Zwischen den ersten beiden Teilfolgen der Form 111 wird geprüft ob zwischen je zwei Teilfolgen 11 jeweils ein String $0^i\ 1\ 0^j\ 1\ 0^k\ 1\ 0^l\ 1\ 0^m$ mit $i, j, k, l, m \geq 1$ steht, also mit 4 Einsen und zwischen den Einsen wiederum eine positive Anzahl an Nullen.
- Falls die obigen Anforderungen erfüllt sind, hat der String das richtige Format. Nun überprüfen wir noch, ob die Übergänge die richtige Form haben. Zum Beispiel soll nach dem Lesen von \$ der Kopf nicht nach links gehen dürfen. Das heißt, in einem String $0^i\ 1\ 0^j\ 1\ 0^k\ 1\ 0^l\ 1\ 0^m$ mit $j = 4$ darf nicht $m = 2$ sein.

Es ist leicht einzusehen, dass es eine Turingmaschine gibt, die alle notwendigen Überprüfungen durchführt. Der Zeitbedarf liegt in $O(n)$, wenn n die Länge der Eingabe $\langle M\rangle w$ ist.

Nun kopieren wir den String $\langle M\rangle$ auf das zweite Band und schreiben w linksbündig auf das erste Band. Alle Köpfe rücken nach links. Wir verwenden folgende Idee:

- Band 1 simuliert die Maschine M.
- Band 2 enthält $\langle M \rangle$.
- Band 3 speichert den aktuellen Zustand von M.

Zur Simulation eines Schrittes von M durch M_U wird wie folgt realisiert.

- Auf Band 1 wird aktuell ein Zeichen gelesen.
- Auf Band 3 wird der Zustand gelesen.
- Auf Band 2 wird der entsprechende Übergang gesucht.
- Die Inschrift auf Band 1 wird gemäß Übergang aktualisiert.
- Der Kopf auf Band 1 wird gemäß Übergang geändert.
- Der Zustand auf Band 3 wird gemäß Übergang geändert.
- Köpfe von Band 2 und 3 kehren zurück auf die Ausgangsposition (links).

Die Laufzeit eines solchen Schrittes ist proportional zur Größe der Eingabe $\langle M \rangle$. □

Die Simulation kann natürlich auch durch eine 1-Band DTM erfolgen, da wir in Theorem 6 gezeigt haben, dass jede k-Band Maschine durch eine 1-Band Maschine simuliert werden kann.

Kapitel 3

Berechenbarkeit

Nachdem wir uns im vorherigen Abschnitt mit einem konkreten Maschinenmodell befasst haben, wollen wir uns nun überlegen, was mit diesem Modell prinzipiell berechnet werden kann und wie mächtig das betrachtete Konzept insgesamt ist. Den Begriff der Berechenbarkeit oder Entscheidbarkeit werden wir wiederum über die Sprachen führen.

Zunächst präzisieren wir nochmal den Unterschied zwischen dem *Akzeptieren einer Sprache* und dem *Entscheiden einer Sprache*. Danach betrachten wir strukturelle Eigenschaften der entsprechenden Sprachen. Mittels der Gödelnummerierung können wir Eigenschaften von Turingmaschinen auch unter dem Aspekt von entscheidbaren oder rekursiv aufzählbaren Sprachen untersuchen.

Danach werden wir feststellen, dass es Sprachen gibt, die nicht entscheidbar sind. Das wichtiges Beispiel ist dabei das sogenannte *Halteproblem*. Die Sprache

$$H := \{\langle M\rangle x | M \text{ ist } DTM, \text{ die bei Eingabe von } x \text{ hält}\}$$

ist nicht entscheidbar.

Außerdem betrachten wir noch eine alternative Beschreibung eines Rechners, die sich als äquivalent herausstellen wird. Insgesamt führt uns dass zu der These, dass die im intuitiven Sinne berechenbaren Funktionen genau die Funktionen sind, die auch eine Turingmaschine berechnen kann (*Church-Turing These*).

3.1 Entscheidbare und rekursiv aufzählbare Sprachen

Wir hatten zwar bereits den Begriff DTM-entscheidbar definiert, greifen diesen Ausdruck hier aber nochmal auf und gehen etwas genauer auf den Unterschied zwischen akzeptieren und entscheiden ein.

Definition 10 *Sei Σ eine endliche Menge und $L \subseteq \Sigma^*$.*

- *Die Sprache L heißt* rekursiv aufzählbar *genau dann, wenn es eine DTM gibt, die L akzeptiert.*
- *Die Sprache L heißt* entscheidbar *genau dann, wenn es eine DTM gibt, die L entscheidet.*

Nach Theorem 6 können wir in der obigen Definition beliebige Mehrbandmaschinen verwenden. Wie bereits erwähnt wird entscheidbar gelegentlich auch *rekursiv* genannt. Für eine Sprache $L \subseteq \Sigma^*$ sei $\overline{L} := \Sigma^* \setminus L$ die *Komplementärsprache* von L.

3.2 Eigenschaften entscheidbarer und rekursiv aufzählbarer Sprachen

In diesem Abschnitt wollen wir zeigen, inwieweit die Begriffe entscheidbar und rekursiv aufzählbar *abgeschlossen* bezüglich klassischer Mengenoperationen wie Schnitt, Vereinigung und Komplementbildung sind. Diese Eigenschaften wollen wir dann beispielsweise für die *sprachliche* Betrachtung von Turingmaschinen verwenden.

Lemma 11 *Seien L_1 und L_2 entscheidbare Sprachen, dann gilt:*

1. $L_1 \cap L_2$ *ist entscheidbar*
2. $L_1 \cup L_2$ *ist entscheidbar*
3. $\overline{L}$ *ist entscheidbar*

Beweis.

Zu 1.: Seien $M_1 = (\Sigma_1, Q_1, \delta_1, q^1, F_1)$ und $M_2 = (\Sigma_2, Q_2\delta_2, q^2, F_2)$ die jeweiligen 1-Band DTMs, die L_1 und L_2 entscheiden. Wir konstruieren eine 2-Band DTM M, die $L_1 \cap L_2$ entscheidet. Die Maschine arbeitet wie folgt:

- Bei Eingabe von w wird das Verhalten von M_1 auf Band 1 simuliert und das Verhalten von M_2 auf Band 2.
- Falls M_1 und M_2 das Wort w akzeptieren, so auch M. Sonst wird das Wort verworfen.

Konkreter müssen wir dazu zunächst die Eingabe w auch auf das zweite Band kopieren. Dazu benötigen wir ein paar extra Zustände. Darüberhinaus enthält die Maschine M die Zustandsmenge $Q_1 \times Q_2$. Die Zustandsübergangsfunktion δ von M für die Simulation kann konkret wie folgt beschrieben werden. Für $q_1 \in Q_1$, $q_2 \in Q_2$, $a_1 \in \Sigma_1$, $a_2 \in \Sigma_2$ und $d_1, d_2 \in \{1, -1, 0\}$ sei $\delta((q_1, q_2), a_1, a_2) := ((p_1, p_2), b_1, b_2, d_1, d_2)$, falls $\delta_1(q_1, a_1) = (p_1, b_1, d_1)$ und $\delta_1(q_2, a_2) = (p_2, b_2, d_2)$ gilt.

Ein paar weitere Regeln und Zustände sind notwendig, wenn M_1 oder M_2 in einen akzeptierenden oder ablehnenden Zustand hält und die andere Maschine noch weiterlaufen muss. Abschließend akzeptiert M das Wort w, falls bei der Simulation, sowohl M_1 als auch M_2 das Wort akzeptieren. Sonst wird das Wort verworfen.

Zu 2.: Leichte Übungsaufgabe für den Leser.

Zu 3.: Sei $M = (\Sigma, Q, \delta, q, F)$ die Maschine, die L entscheidet. Für $\overline{M}$ ändern wir lediglich die Ausgabe für die akzeptierenden Endzustände auf 0 und für die ablehnenden Endzustände auf 1.

Da M auf jeder Eingabe hält, hält auch $\overline{M}$ auf jeder Eingabe und akzeptiert genau die Wörter aus Σ^*, die M verwirft. □

Lemma 12 *Seien L_1 und L_2 rekursiv aufzählbare Sprachen, dann gilt:*

1. $L_1 \cap L_2$ *ist rekursiv aufzählbar*

2. $L_1 \cup L_2$ ist rekursiv aufzählbar

Beweis. Der Beweis kann analog wie in Lemma 11 geführt werden. Der Unterschied ist lediglich, dass beispielsweise eine Maschine M_i, die simuliert wird, nicht zwangsläufig halten muss. In diesem Fall hält beispielsweise beim Schnitt, die simulierende Maschine ebenfalls nicht. Bei der Vereinigung darf die simulierende Maschine halten, wenn eine der simulierten Maschinen M_i hält. □

Jetzt wollen wir die intuitive Beziehung zwischen rekursiv aufzählbar und entscheidbar formalisieren.

Lemma 13 *L ist genau dann entscheidbar, wenn L und $\overline{L}$ rekursiv aufzählbar sind.*

Beweis. "$\Rightarrow$": Es ist zu zeigen, dass L und $\overline{L}$ rekursiv aufzählbar sind. Dass L rekursiv aufzählbar ist, ist trivial, da entscheidbar die strengere Eigenschaft ist. Nach Lemma 11 (3.) ist auch $\overline{L}$ entscheidbar und somit ebenfalls rekursiv aufzählbar.

"$\Leftarrow$": Seien hier M_1 und M_2 die zugehörigen 1-Band DTMs, die L respektive $\overline{L}$ akzeptieren. Wir lassen M_1 und M_2 wie im Beweis von Lemma 11 (1.) parallel laufen. Die zugehörige Maschine M hält in jedem Fall für mindestens eine simulierte Maschine in einem akzeptierenden Zustand, da jede Eingabe w entweder zu L oder $\overline{L}$ gehört. Falls die Maschine M_1 akzeptiert, akzeptiert M das Wort und hält. M verwirft das Wort und hält, falls die Maschine M_2 das Wort akzeptiert. □

Eine weitere Folgerung ist:

Lemma 14 *Ist L nicht entscheidbar, dann ist auch $\overline{L}$ nicht entscheidbar.*

Beweis. Der Beweis ergibt sich durch Umkehrung aus Lemma 11 (3.).

Übungsaufgabe: Führen Sie einen Widerspruchsbeweis explizit über DTMs. □

3.3 Turingmaschinen-Eigenschaften als Sprachen

Insbesondere durch die eindeutige Kodierung von Turingmaschinen und mittels der Verwendung der universellen Turingmaschine M_U können wir auch die Eigenschaften von Turingmaschinen durch die Begriffe entscheidbar und rekursiv aufzählbar bequem charakterisieren. Als Beispiele verwenden wir hier das Halteproblem H. Etwas einfacher ist folgendes Problem Z.

$$Z := \{\langle M\rangle n | M \text{ ist eine } DTM \text{ mit mindestens } n \text{ Zuständen}\}$$

Lemma 15 *Die Sprache Z ist entscheidbar.*

Beweis. In einem ersten Schritt wird überprüft, ob das Wort $\langle M\rangle n$ der Kodierung einer Turingmaschine mit angehängten binär kodierten $n \in \mathbb{N}$ entspricht. Falls das nicht der Fall sein sollte, wird das Wort verworfen. Ansonsten überprüfen wir im String $\langle M\rangle$ ob es dort $n-1$ verschiedene Zustände gibt. Der Endzustand wird bekanntlich nicht kodiert. Dazu muss in den 11 getrennten Strings, jeweils die Anzahl der Nullen im *Zustandsteil* gezählt werden, das Maximum gespeichert und mit $n-1$ verglichen werden. □

Ein Ziel unserer Betrachtungen wird es sein, zu zeigen, dass das aus praktischen Sicht sehr wichtige Halteproblem nicht entscheidbar ist, die zugehörige Sprache läßt sich allerdings rekursiv aufzählen.

Lemma 16 *Die Sprache H ist rekursiv aufzählbar.*

Beweis. Wir konstruieren eine DTM $\overline{M}$ und verwenden dabei intern als Unterprogramm die universelle Turingmaschine M_U, die stets zunächst überprüft, ob der String einer Kodierung entspricht. Die Vorgehensweise ist somit wie folgt:

- Simuliere M mit Eingabe x gemäß Beweis von Theorem 9 mittels M_U. Falls die Eingabe nicht die Form $w = \langle M \rangle x$ hatte, lehnt die Maschine das Wort ab.
- Falls die Maschine M_U die Eingabe $\langle M \rangle x$ (und damit M die Eingabe x) verwirft oder akzeptiert, akzeptiere $\langle M \rangle x$.

Zu zeigen ist, dass $\overline{M}$ nur die Wörter akzeptiert, die in H liegen. Falls $w \in H$ gilt, hält die Maschine M für die Eingabe x (akzeptieren oder verwerfend) und somit hält auch $\overline{M}$ für die Eingabe $\langle M \rangle x$ und akzeptiert das Wort w. Falls $w \notin H$ gilt, dann hält M für die Eingabe x nicht und somit terminiert auch die Maschine $\overline{M}$ für die Eingabe $\langle M \rangle x$ nicht. □

3.4 Existenz unentscheidbare Probleme

Zunächst wollen wir nun zeigen, dass es tatächlich Sprachen gibt. die nicht entscheidbar sind. Das Ergebnis werden wir dann für unser wichtiges Problem H verwenden. Die Tatsache, dass es nicht entscheidbare Sprachen gibt, werden wir über ein Abzählargument führen.

Definition 17 *Eine Menge M heißt* abzählbar*, falls es eine surjektive Abbildung $f : \mathbb{N} \to M$ gibt. Nicht abzählbare Mengen heißen* überabzählbar*.*

Jede endliche Menge ist abzählbar. Jede surjektive Abbildung $f : \mathbb{N} \to M$ liefert automatisch eine *Nummerierung* der Menge M. Dabei lassen wir doppelte Vorkommen aus M einfach weg. Bei unendlichen Mengen M liefert die surjektive Abbildung $f : \mathbb{N} \to M$ eine *laufende Nummerierung* der Menge M. Mathematisch gesehen erhalten wir so eine Bijektion f' zwischen $\mathbb{N}$ und M. Dann spricht man auch von einer *abzählbar unendlichen* Menge M. In diesem Sinne sind dann M und $\mathbb{N}$ dann gleichmächtig.

Beispiele:

- Die Menge aller rationalen Zahlen $\mathbb{Q}$ abzählen. Dazu benutzen wir beispielsweise zunächst eine surjektive Abbildung durch Paare
  $(1, \frac{0}{1}), (2, \frac{1}{1}), (3, -\frac{1}{1}), (4, \frac{1}{2}), (5, -\frac{1}{2}), (6, \frac{2}{2}), (7, -\frac{2}{2}), (8, \frac{2}{1}), (9, -\frac{2}{1}),$
  $(10, \frac{1}{3}), (11, -\frac{1}{3}), (12, \frac{2}{3}), (13, -\frac{2}{3}), (14, \frac{3}{3}) \ldots$ und zählen dann tatsächlich nur die nicht vollständig gekürzten Brüche, also
  $(1, \frac{0}{1}), (2, \frac{1}{1}), (3, \frac{1}{2}), (4, -\frac{1}{2}), (5, \frac{1}{3}), \ldots$
- Die Menge der Wörter über $\{0, 1\}$ ist abzählbar unendlich, eine natürliche Aufzählung ist:

  $0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, 1000, \ldots$
- Die Menge aller Turingmaschinen ist abzählbar unendlich, weil wir die Maschinen durch ihre Gödelnummern beschreiben können und es sich dabei um eine Teilmenge von $\{0, 1\}^*$ handelt.

Wir betrachten die Menge aller Abbildungen $f : \mathbb{N} \to \{0,1\}$ und bezeichnen diese mit $\{0,1\}^{\mathbb{N}}$.

Theorem 18 *Die Menge $\{0,1\}^{\mathbb{N}}$ ist überabzählbar.*

Beweis. Offensichtlich ist $\{0,1\}^{\mathbb{N}}$ unendlich. Angenommen, die Menge $\{0,1\}^{\mathbb{N}}$ sei abzählbar unendlich, dann gibt es eine Bijektion $\varphi : \mathbb{N} \to \{0,1\}^{\mathbb{N}}$, die jedem $n \in \mathbb{N}$ eine Funktion $f_n : \mathbb{N} \to \{0,1\}$ zuordnet. Wir schreiben dann die abzählbar vielen Funktionen und die Funktionswerte abgezählt in eine Tabelle. Für die Nummerierung verwenden wir eine nach Annahme existierende Bijektion zwischen $\mathbb{N}$ und $\{0,1\}^{\mathbb{N}}$.

$$\begin{pmatrix} f_1(1) & f_1(2) & f_1(3) & f_1(4) & f_1(5) & \ldots \\ f_2(1) & f_2(2) & f_2(3) & f_2(4) & f_2(5) & \ldots \\ f_3(1) & f_3(2) & f_3(3) & f_3(4) & f_3(5) & \ldots \\ f_4(1) & f_4(2) & f_4(3) & f_4(4) & f_4(5) & \ldots \\ f_5(1) & f_5(2) & f_5(3) & f_5(4) & f_5(5) & \ldots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ldots \end{pmatrix}$$

Nun definieren wir eine Funktion f_{diag} aus $\{0,1\}^{\mathbb{N}}$ mit

$$f_{\text{diag}}(i) := \begin{cases} 0 & \text{falls } f_i(i) = 1 \\ 1 & \text{falls } f_i(i) = 0 \end{cases}$$

Falls $\{0,1\}^{\mathbb{N}}$ abzählbar ist, so gibt es ein k mit $f_k = f_{\text{diag}}$. Andererseits ist aber $f_k(k) \neq f_{\text{diag}}(k)$. Die Bijektion kann nicht existieren. Ein Widerspruch, $\{0,1\}^{\mathbb{N}}$ ist überabzählbar. □

Den obigen Trick im Beweis nennt man auch *Diagonalisierungstrick*. Wir können jetzt auch leicht zeigen, dass die Potenzmenge (Menge aller Teilmengen) von $\mathbb{N}$, kurz $P(\mathbb{N})$ ebenfalls überabzählbar ist.

Lemma 19 *Die Menge $P(\mathbb{N})$ ist überabzählbar.*

Beweis. Es existiert eine einfache Bijektion zwischen $P(\mathbb{N})$ und $\{0,1\}^{\mathbb{N}}$. Für jedes f in $\{0,1\}^{\mathbb{N}}$ sei $M_f = \{j | f(j) = 1\}$. Offensichtlich existiert zu jedem f aus $\{0,1\}^{\mathbb{N}}$ genau ein M_f und umgekehrt. □

Genauso gilt die folgende Aussage bezüglich aller Sprachen über $\{0,1\}$.

Lemma 20 *Die Menge aller Sprachen über $\{0,1\}$ ist überabzählbar.*

Beweis. Jede Sprache über $\{0,1\}$ ist eine binär kodierte Teilmenge von $\mathbb{N}$ und umgekehrt. Also gibt es eine einfache Bijektion zwischen $P(\mathbb{N})$ und der Menge der Sprachen über $\{0,1\}$. □

Jetzt ziehen wir den finalen Schluss. Da die Menge aller Turingmaschinen abzählbar unendlich ist, die Menge aller Sprachen über $\{0,1\}$ aber nicht, gibt es mehr Sprachen als Maschinen und deshalb muss es zwangsläufig Sprachen geben, die nicht entschieden werden können.

Theorem 21 *Es existieren Sprachen, die unentscheidbar sind.*

3.5 Konkrete unentscheidbare Probleme

Wir wollen den obigen Trick der Diagonalisierung zu Nutze machen, um konkrete unentscheidbare Probleme zu konstruieren. Aus der obigen Betrachtung wissen wir bereits, dass wir die Menge der Turingmaschinen und die Menge der Wörter über $\{0, 1\}$ abzählen können. Für die Menge $\{0, 1\}^*$ ist bereits eine konkrete Aufzählung durch die Binärkodierung gegeben. Dass Wort w_i sei somit die Binärkodierung von i. Wir wollen auch die Turingmaschinen aufzählen. Dazu verwenden wir die Gödelnummerierungen der Maschinen. Falls $\text{bin}(i)$ keine Kodierung einer Turingmaschine ist, verwenden wir eine einfache Maschine M_R, die alle Wörter ablehnt und nur das leere Wort akzeptiert.

Sei also

$$M_i = \begin{cases} M & \text{falls bin}(i) \text{ Gödelnummer von } M \text{ ist} \\ M_R & \text{falls bin}(i) \text{ keine Gödelnummer ist} \end{cases} \tag{3.1}$$

Wir betrachten die folgende Diagonalsprache:

$$\text{Diag} := \{w \in \{0, 1\}^* | w = w_i \text{ und } M_i \text{ akzeptiert } w_i \text{nicht}\}$$

Konkret ist das i-te Wort aus $\{0, 1\}^*$ in Diag enthalten, falls es von der i-ten Maschine M_i nicht akzeptiert wird. Die Zugehörigkeit der Wörter w_i zu Diag können wir aus der Diagonalen einer Matrix A ablesen, daher auch der Name der Sprache.

$$A_{ij} = \begin{cases} 1 & \text{falls } M_i \text{ akzeptiert } w_i \\ 0 & \text{sonst} \end{cases}$$

$$\begin{array}{c} \\ M_0 \\ M_1 \\ M_2 \\ M_3 \\ M_4 \\ \\ \end{array} \begin{array}{c} \begin{matrix} w_0 & w_1 & w_2 & w_3 & w_4 & w_5 \end{matrix} \\ \begin{pmatrix} \mathbf{0} & 1 & 0 & 0 & 1 & \dots \\ 1 & \mathbf{1} & 1 & 0 & 1 & \dots \\ 1 & 0 & \mathbf{1} & 1 & 1 & \dots \\ 0 & 0 & 1 & \mathbf{1} & 0 & \dots \\ 0 & 1 & 1 & 1 & \mathbf{0} & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \dots \end{pmatrix} \end{array}$$

Es ist leicht zu sehen, dass $\text{Diag} = \{w_i | A_{ii} = 0\}$ gilt.

Theorem 22 *Die Sprache Diag ist nicht entscheidbar.*

Beweis. Angenommen Diag ist entscheidbar. Dann gibt es eine DTM M_j, die Diag entscheidet. Wir wenden M_j auf w_j an.

Falls $w_j \in \text{Diag}$ gilt, dann akzeptiert M_j das Wort w_j. Dann kann aber w_j nach Definition nicht in Diag gewesen sein. Ein Widerspruch!

Falls $w_j \notin \text{Diag}$ gilt, dann verwirft M_j das Wort w_j. Dann muss w_j aber nach Definition in Diag liegen, ein Widerspruch!

Insgesamt kann es keine TM geben, die Diag entscheidet. □

Lemma 23 *Das Komplement $\overline{\text{Diag}}$ der Sprache Diag ist nicht entscheidbar.*

Beweis. Folgt direkt aus Lemma 14 und Theorem 22. □

3.6 Unentscheidbarkeit des Halteproblems

Kommen wir nun zu unserer zentralen Aussage.

Theorem 24 *Das Halteproblem*

$$H := \{\langle M\rangle x | M \text{ ist } DTM, \text{ die bei Eingabe von } x \text{ hält}\}$$

ist nicht entscheidbar.

Beweis. Angenommen H wäre entscheidbar. Dann sei M_H die zugehörige Maschine, die H entscheidet. Mit dieser Maschine können wir dann aber auch $\overline{\text{Diag}}$ durch die folgende Maschine $M_{\overline{\text{Diag}}}$ entscheiden, sei dabei w die Eingabe für $M_{\overline{\text{Diag}}}$.

- Die Eingabe w entspricht einem w_i. Berechne i und die Gödelnummer $\langle M_i\rangle$ der i-ten Maschine.
- Starte die Maschine M_H mit der Eingabe $\langle M_i\rangle w$.
- Falls M_H die Eingabe verwirft, verwerfe w.
- Falls M_H die Eingabe akzeptiert, so verwende die universelle Maschine M_U und simuliere das Verhalten von M_i bezüglich w. Übernehme die Ausgabe!

Die Maschine $M_{\overline{\text{Diag}}}$ verwendet M_H und M_U als Unterprogramme.

Nun zeigen wir, dass unter den obigen Annahmen, die Maschine $M_{\overline{\text{Diag}}}$ die Sprache $\overline{\text{Diag}}$ entscheidet, ein Widerspruch zu Lemma 23.

Falls $w \in \overline{\text{Diag}}$ liegt, dann gilt dass M_i das Wort $w = w_i$ akzeptiert. Somit akzeptiert M_H das Wort $\langle M_i\rangle w$ und die Simulation durch M_U führt dazu, dass $M_{\overline{\text{Diag}}}$ das Wort w akzeptiert.

Falls $w \notin \overline{\text{Diag}}$ liegt, dann gilt dass M_i das Wort $w = w_i$ verwirft oder nicht hält. Falls M_i nicht hält, verwirft M_H das Wort $\langle M_i\rangle w$ und $M_{\overline{\text{Diag}}}$ verwirft das Wort w dann auch. Falls M_i hält und verwirft, hält zunächst M_H aber durch Simulation wird $M_{\overline{\text{Diag}}}$ Wort w verwerfen.

$\overline{\text{Diag}}$ wäre entscheidbar! □

3.7 Reduktionen

Im vorhergehenden Abschnitt haben wir gesehen, dass das Halteproblem nicht entscheidbar ist, dabei haben wir die Unterprogrammtechnik verwendet. Falls das Halteproblem entscheidbar ist, dann kann mittels der universellen Maschine M_U und mit der dann existierenden Maschine M_H auch die Sprache $\overline{\text{Diag}}$ entschieden werden. Ein Widerspruch entstand.

Wir wollen hier eine formalere Methode vorstellen, durch die sich Komplexitätsaussagen ableiten lassen. Wir wollen ein Problem in funktioneller Weise auf ein bekanntes Problem übertragen. Haben wir beispielsweise eine Maschine, die für zwei Zahlen $a, b \in \mathbb{Z}$ das Produkt $a \cdot b$ berechnet, so können wir das Problem der Berechnung von a^3 ebenfalls leicht lösen, in dem wir aus der Eingabe a zunächst das Produkt $a' = a \cdot a$ berechnen und dann nochmal das Produkt $a \cdot a'$. In diesem Sinne ist a^3 intuitiv *leichter* zu lösen als $a \cdot b$ oder anders ausgedrückt a^3 wird auf das Problem $a \cdot b$ *reduziert*. Das Problem $a \cdot b$ ist mindestens so schwer zu lösen wie das Problem a^3. Wenn ich $a \cdot b$ lösen kann, so auch a^3. Die Umkehrung gilt nämlich

offensichtlich nicht. Aus einer Berechnung von a^3 läßt sich direkt keine Berechnung von $a \cdot b$ ableiten.

Wir hatten es bislang in erster Linie mit Entscheidungsproblemen und den dazugehörigen Sprachen zu tun. Im obigen Sinne muss es bei einer Reduktion eine geeignete *Übertragung* von Wörtern einer Sprache L' in eine Sprache L geben. Formal definieren wir eine Reduktion wie folgt.

Definition 25 *Seien L' und L Sprachen über $\{0,1\}$. Die Sprache L' heißt* reduzierbar auf *L, falls es eine totale Funktion $f : \{0,1\}^* \to \{0,1\}^*$ gibt, für die gilt:*

1. *Für alle $w \in \{0,1\}^*$ gilt:* $\quad w \in L' \Leftrightarrow f(w) \in L$
2. *Die Funktion f ist DTM-berechenbar, d.h. es gibt eine DTM M_f, die f berechnet (siehe auch Definition 2).*

Die Funktion f wird dann auch als Reduktion *von L' auf L bezeichnet. Ist L' auf L reduzierbar, so schreiben wir $L' \leq L$. Wir sprechen insgesamt davon, dass L' auf L* mittels f *reduzierbar ist.*

Für diese (und nicht nur für diese) Definition ist es wichtig, dass stets die Bedingungen vollständig überprüft werden. Wir verlangen eine totale DTM-berechenbare Funktion $f : \{0,1\}^* \to \{0,1\}^*$, die entsprechende Maschine hält also auf jedem Input und ist nicht durch eine Erweiterung des Wertebereiches auf $\Sigma^* \cup \{\perp\}$ total ergänzt worden. Es reicht andererseits auch nicht aus, nur $w \in L' \Rightarrow f(w) \in L$ zu zeigen, das ist ja stets für $L = \{0,1\}^*$ erfüllt. Es genügt insgesamt nicht, intuitiv *mindestens so schwer wie* als Argument zu verwenden.

Bevor wir zu einem Beispiel übergehen, zweigen wir, dass die obige Relation $\leq$ schöne Eigenschaften hat.

Lemma 26 *Die Relation $\leq$ aus Definition 25 ist transitiv und reflexiv. Es gilt für alle Sprachen $L, L' \in \{0,1\}^*$:*

1. *$L \leq L$ (reflexiv)*
2. *Aus $L' \leq L$ und $L \leq L''$ folgt $L' \leq L''$ (transitiv)*
3. *$L' \leq L$ mittels $f \Leftrightarrow \overline{L'} \leq \overline{L}$ mittels f*

Beweis.

Zu 1. Hier kann die Funktion $id : \{0,1\}^* \to \{0,1\}^*$ mit $id(x) = x$ verwendet werden.

Zu 2. Sei $L' \leq L$ mittels f und $L \leq L''$ mittels g, dann gilt $L' \leq L''$ mittels $g \circ f$, denn für jedes $w \in \{0,1\}^*$ gilt:

$$w \in L' \Leftrightarrow f(w) \in L \Leftrightarrow g(f(w)) \in L'' \Leftrightarrow (g \circ f)(w) \in L''\,.$$

Zu 3. Die Aussage

$$\forall w \in \{0,1\}^* w \in L' \Leftrightarrow f(w) \in L$$

ist äquivalent zu

$$\forall w \in \{0,1\}^* w \notin L' \Leftrightarrow f(w) \notin L\,.$$

Daraus folgt die Behauptung. □

Das Ziel der Reduktion ist es, Berechenbarkeitsaussagen zu übertragen.

Lemma 27 *Seien L' und L Sprachen über $\{0,1\}$. Es gilt:*

1. *Falls L entscheidbar ist und $L' \leq L$ gilt, ist auch L' entscheidbar.*
2. *Falls L rekursiv aufzählbar ist und $L' \leq L$ gilt, ist auch L' rekursiv aufzählbar.*

Beweis.

Zu 1.: Da L entscheidbar ist, gibt es eine DTM M_L die L entscheidet. Die Maschine M_L hält auf jedem Input. Sei $L' \leq L$ mittels f. Dann existiert eine DTM M_f, die die totale Funktion f berechnet und somit auf jeder Eingabe hält. Daraus konstruieren wir eine Maschine $M_{L'}$, die eine Eingabe w wie folgt verarbeitet.

- Berechene $v = f(w)$ mit M_f.
- Lasse M_L auf v laufen. Falls M_L das Wort v akzeptiert, akzeptiere das Wort w. Falls M_L das Wort v verwirft, verwerfe das Wort w.

Es gilt:

$$\begin{aligned} M_{L'} \text{ akzeptiert } w \Leftrightarrow M_L \text{ akzeptiert } v = f(w) &\Leftrightarrow f(w) \in L \\ &\Leftrightarrow w \in L' \end{aligned}$$

Zu 2.: Analog zu 1. Der Unterschied ist, dass M_L nicht auf allen Eingaben hält und somit $M_{L'}$ ebenfalls nicht. □

Insgesamt läßt sich festhalten, dass die Reduktion eine spezielle Form der Unterprogrammtechnik darstellt. Natürlicherweise lassen sich die Konklusionen negieren.

Lemma 28 *Seien L' und L Sprachen über $\{0,1\}$ mit $L' \leq L$. Es gilt:*

1. *Falls L' nicht entscheidbar ist, ist auch L nicht entscheidbar.*
2. *Falls L' nicht rekursiv aufzählbar ist, ist auch L nicht rekursiv aufzählbar.*

Wir betrachten nun ein Beispiel. Die Sprache

$$A := \{\langle M\rangle x | M \text{ ist } DTM, \text{ die die Eingabe von } x \text{ akzeptiert}\}$$

sei das sogenannte *Akzeptanzproblem.*

Wir wollen zeigen, dass $H \leq A$ gilt, das Halteproblem läßt sich auf das Akzeptanzproblem reduzieren. Aus Lemma 28 und Theorem 24 folgt dann, dass die Sprache A nicht entscheidbar ist.

Dazu definieren wir eine totale DTM-berechenbare Funktion f wie folgt:

$$f(w) = \begin{cases} w & : \text{falls } w \text{ nicht von der Form } \langle M\rangle x \text{ für DTM } M \\ \langle\overline{M}\rangle x & : \text{falls } w = \langle M\rangle x \text{ für DTM } M, \\ & \text{wobei die DTM } \overline{M} \text{ sich aus } M \text{ wie folgt ergibt} \end{cases}$$

$\overline{M}$ arbeitet bei Eingabe $y \in \{0,1\}^*$ wie folgt:

- Simuliere M mit Eingabe y
- Falls M Eingabe y akzeptiert, akzeptiere y
- Falls M Eingabe y ablehnt, akzeptiere y

Falls $w \in \{0,1\}^*$ nicht von der Form $\langle M\rangle x$ ist, dann gilt offensichtlich gleichzeitig $w \notin H$ und $w = f(w) \notin A$. Es bleibt zu zeigen, dass

$$w = \langle M\rangle x \in H \Leftrightarrow f(w) = f(\langle M\rangle x) = \langle\overline{M}\rangle x \in A$$

gilt.

"$\Rightarrow$": Aus $w = \langle M\rangle x \in H$ folgt, dass M bei Eingabe von x hält, nach Definition von $\overline{M}$ akzeptiert $\overline{M}$ die Eingabe x und es ist $\langle\overline{M}\rangle x \in A$.

"$\Leftarrow$": Aus $\langle\overline{M}\rangle x \in A$ folgt, dass $\overline{M}$ die Eingabe x akzeptiert. Dann muss aber M auf der Eingabe x halten. Dann ist $\langle M\rangle x \in H$.

Insgesamt haben wir gerade das folgende Theorem bewiesen.

Theorem 29 *Das Halteproblem läßt sich auf das Akzeptanzproblem reduzieren, $H \leq A$. Das Akzeptanzproblem ist nicht entscheidbar.*

Ähnliche Reduktionen bezüglich des Halteproblems gibt es für die Sprachen

$$H_\epsilon := \{\langle M\rangle | M \text{ ist } DTM, \text{ die bei Eingabe von } \epsilon \text{ hält}\}$$

$$H_{\text{all}} := \{\langle M\rangle | M \text{ ist } DTM, \text{ die bei Eingabe jedes } x \in \{0,1\}^* \text{ hält}\}$$

Übungsaufgabe: Formulieren Sie die Reduktionsbeziehungen unter H, H_ϵ und H_{all} und beweisen Sie diese durch die Angabe entsprechender Funktionen.

3.8 Churchsche These und Registermaschine

Die Chruchsche These wurde in den 30er Jahtren von dem Mathematiker Alonzo Church aufgestellt und besagt, dass davon ausgegangen werden kann, dass alle im intuitiven Sinne berechenbaren Funktionen genau die Funktionen sind, die sich durch die Turingmaschinen berechnen lassen.

Diese These ist nicht beweisbar, das sich der Begriff *im intuitiven Sinne berechenbar* gar nicht formalisieren läßt. Es ist nicht klar, wo dieser Begriff enden soll. Sobald man ein formales Rechen- oder Funktionenmodell entwickelt hat, hat man sich darauf festgelegt. Da

der Phantasie keinen Grenzen gesetzt sind müsste man de facto jede andere vernünftige Beschreibung von Funktionen oder Maschinen darauf zurückführen können. Es gibt dafür aber unendlich viele Beschreibungsmöglichkeiten. Also begnügen wir uns damit, dass schon sehr viele Äquivalenzen gezeigt wurden und die These dadurch immer weiter untermauert werden konnte. Es wird stets eine These bleiben.

Wir hatten bislang die Äquivalenz zwischen 1-Band und Mehrbandmaschinen bewiesen und darauf hingewiesen, dass man sich der Berechenbarkeit auch durch die formale Beschreibung von rekursiven Funktionen nähern kann (was dann wiederum zu den DTM-berechenbaren Funktion führt). Wir haben das hier nicht getan aber wie bereits bemerkt ist die Bezeichnung der rekursiven (entscheidbaren) Sprachen historisch damit verknüpft.

Wir wollen uns nochmal ein alternatives, etwas moderneres Maschinenmodell kurz anschauen und die Äquivalenz zu den Turingmaschinen skizzieren. Das macht die These für uns ggf. noch etwas glaubhafter.

Chursche These: *Die durch eine Turingmaschine berechenbaren Funktionen und sind genau die Funktionen, die sich im intuitiven Sinne berechnen lassen.*

Wir betrachten eine sogenannte *Registermaschine* RAM (Random Access Maschine), die der maschinennahen Assembler-Sprache auf modernen Rechnern nachempfunden ist. Auf dem maschinennahem Assembler-Sprachen basieren dann wiederum auf einer Meta-Ebene alle modernen Programmiersprachen. Die RAM hat einen unbeschränkten Speicher der durch Registerzellen $R(0), R(1), R(2), \ldots$ realisiert wird. In jedem Register kann eine ganze Zahlen abgelegt werden. Die Maschine arbeitet auf einem *Programm* mit einer beliebig aber festen Anzahl an durchnummerierten Programmzeilen mit entsprechenden Befehlen aus einer vorgegebenen Befehlsmenge. Ein *Befehlszähler* b der Maschine legt fest, welche Befehlszeile als nächstes ausgeführt wird. Die Befehle dürfen den Befehlszähler und die Registerinhalte ändern. Zu Beginn steht der Befehszähler b auf 1. Ein Programm und Parameter können zu Beginn in der Maschine geladen werden; siehe Abbildung 3.1.

Wie bei den Turingmaschinen, gibt es verschiedene Arten von Registermaschinen. Es handelt sich dabei wieder um Abwandlungen, die durch Konventionen bestimmt sind. Beispielsweise werden häufig die Registerzellen $R(1), R(2), R(3), \ldots, R(n)$ zu Beginn als jeweilige Eingabe vorsehen. Eine konkrete Rechnung findet dann immer nur im Register $R(0)$ (dann auch *Akkumulator* genannt) statt; siehe Abbildung 3.1. Die restlichen Register können aber überschrieben werden. Die Berechnungsmöglichkeiten auf dem Akkumulator sind Addititition, Subtraktion, Multiplikation und ganzzahlige Division. Die Register können auch relativ angesprochen. Der Befehlssatz und die zugehörige Semantik der hier betrachteten Maschine befindet ist in der folgenden Tabelle 3.8 zusammengefaßt. Es gibt Befehle zum Speichern und Laden, für die Arithmetik, für Programmsprünge und das Ende.

Das folgende Registerprogramm berechnet die Summe von n Zahlen a_i für $i = 1, \ldots, n$. Die Registereingabe ist dabei $R(1) = n$ und in den Registern $R(4), R(5), \ldots, R(4 + R(1))$ sind die Zahlen $R(i) = a_{i-3}$ für $i = 4, \ldots n + 3$ abgelegt. Das Ergebnis befindet sich am Ende in $R(2)$ und $R(3)$ ist jeweils die Adresse des nächsten Summanden.

Übungsaufgabe: Schreiben Sie ein Registerprogramm, dass (i) das Maximum von n Zahlen berechnet und (ii) n Zahlen sortiert ausgibt.

Jetzt Vergleichen wir die beiden doch sehr unterschiedlichen Rechnermodelle. Die RAM beendet die Berechnung sobald ein ENDE Befehl erreicht wird. Wir stellen zunächst fest, dass die Registermaschinen Funktionen der Art $f : \mathbb{Z}^k \to \mathbb{Z}^l \cup \{\perp\}$ berechnen, wobei hier wie bei der DTM das Zeichen $\perp$ für das Nichtterminieren einer Berechnung steht. Da wir die ganzen

Syntax	Registeränderung	Befehlszähler
LOAD i	$R(0) := R(i)$	$b := b + 1$
CLOAD i	$R(0) := i$	$b := b + 1$
INDLOAD i	$R(0) := R(R(i))$	$b := b + 1$
STORE i	$R(i) := R(0)$	$b := b + 1$
INDSTORE i	$R(R(i)) := R(0)$	$b := b + 1$
ADD i	$R(0) := R(0) + R(i)$	$b := b + 1$
CADD i	$R(0) := R(0) + i$	$b := b + 1$
INDADD i	$R(0) := R(0) + R(R(i))$	$b := b + 1$
SUB i	$R(0) := R(0) - R(i)$	$b := b + 1$
CSUB i	$R(0) := R(0) - i$	$b := b + 1$
INDSUB i	$R(0) := R(0) - R(R(i))$	$b := b + 1$
MULT i	$R(0) := R(0) \cdot R(i)$	$b := b + 1$
CMULT i	$R(0) := R(0) \cdot i$	$b := b + 1$
INDMULT i	$R(0) := R(0) \cdot R(R(i))$	$b := b + 1$
DIV i	$R(0) := \lfloor R(0)/R(i) \rfloor$	$b := b + 1$
CDIV i	$R(0) := \lfloor R(0)/i \rfloor$	$b := b + 1$
INDDIV i	$R(0) := \lfloor R(0)/R(R(i)) \rfloor$	$b := b + 1$
GOTO j		$b := j$
IF $R(0)$ rel x GOTO j		$b := \begin{cases} j & : \text{ falls } R(0) \text{ rel } x \\ & : \ b+1 \text{ sonst} \end{cases}$
END	Ende der Berechnung	

Tabelle 3.1: Der Befehlssatz einer Registermaschine mit Relation rel $\in \{<, >, =, \leq, \geq\}$. In der IF-Anweisung ist x eine feste Konstante.

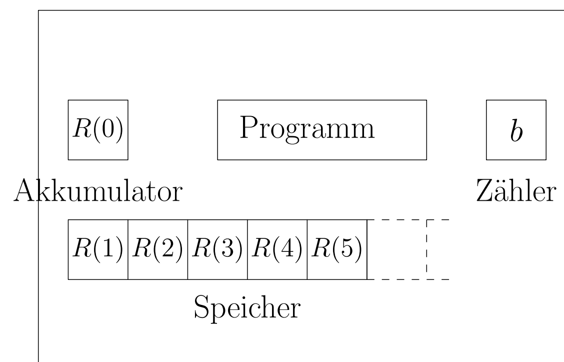

Abbildung 3.1: Die Komponenten einer Registermaschine mit Akkumulator, Programme und Parameter in den Registern können geladen werden.

Zeile	Befehl	Kommentar
1	CLOAD 0	Initialisierung
2	STORE 2	
3	CLOAD 4	
4	STORE 3	
5	LOAD 2	Schleife
6	INDADD 3	Nächster Summand hinzu
7	STORE 2	Abspeichern
8	LOAD 1	Zähler
9	IF $R(0) > 1$ GOTO 11	Letzter Summand?
10	END	Fertig
11	CSUB 1	Zähler inkrementieren
12	STORE 1	Abspeichern
13	LOAD 3	Summand Index
14	CADD 1	Index erhöhen
15	STORE 3	Abspeichern
16	GOTO 5	Schleifenanfang

Tabelle 3.2: Ein RAM-Programm zur Berechnung der Summe von n Zahlen.

Zahlen auch eins zu eins binär kodieren können, gibt es bezüglich der Funktionsbeschreibung zwischen DTMs und RAMs keinen Unterschied.

Ein Unterschied herrscht allerdings im Speicherverhalten. Die Registermaschine darf beliebig große Zahlen abspeichern. Wie wir bereits wissen, kann man dann in einem Register zum Beispiel eine vollständige DTM kodieren. Diesen Vorteil müssen wir bezüglich eines Laufzeitvergleichs ein bisschen abschwächen. Zur Kodierung einer ganzen Zahl der Größe n brauchen wir in der Turingmaschine $\log n$-Bits, die wir auch noch lesen müssen. Deshalb haben wir folgende verschiedene Kostenmaße:

- Uniformes Kostenmaß: Jeder Rechenschritt zählt eine Zeiteinheit.
- Logarithmisches Kostenmaß: Die Laufzeitkosten eines Rechenschrittes sind proportional zur binären Länge der Zahlen der dabei verwendeten Register.

Die RAM ist offensichtlich mächtig genug, um die klassischen Algorithmen, die wir im ersten Teil der Veranstaltung kennengelernt haben, zu realisieren. Schleifen, rekursive Aufrufe und auch komplexere Datenstrukturen lassen sich offensichtlich abbilden.

Theorem 30 *Jede $t(n)$-laufzeitbeschränkte DTM M mit Eingabegröße n kann durch eine RAM simuliert werden, die im uniformen Kostenmodell $O(t(n) + n)$-laufzeitbeschränkt und im logarithmischen Kostenmodell $O((t(n) + n) \log(t(n) + n))$-laufzeitbeschränkt ist.*

Beweis. Wir skizzieren ein RAM-Programm, das eine 1-Band Maschine simuliert. Das erste Register speichert die Kopfposition als Nummer. das zweite Register speichert den Zustand als Nummer. Ab Register 3 werden die aktuellen Bandinhalte der DTM an den Positionen $1, 2, 3, \ldots$ exakt in $R(3), R(4), R(5), \ldots$ abgespeichert.

Die Simulation eines Schrittes wird wie folgt durchgeführt. Falls die DTM in den Endzustand geraten ist, beendet die RAM die Berechung ebenfalls. Ein einfacher Test im Akkumulator bezüglich der Zustandsnummer reicht dazu aus. Ansonsten muss der Berechnungschritt uniform in konstanter Zeit ausgeführt werden. Zunächst gibt es $|Q|$ viele IF-Abfragen bezüglich

des Zustandes und $|\Sigma|$ viele Abfragen bezüglich des Zeichens der entsprechenden beiden Register. Je nach Ausgang dieser Abfragen, wir Register 2 (Zustand), Register 1 (Bandposition) und der Registerinhalt der Bandposition geändert. Dazu sind jeweils konstant viele Operationen mit dem Akkumulator notwendig.

Die Initialisierung benötigt $O(n)$ viel Zeit, jeder Rechenschritt der DTM wird in konstanter Zeit umgesetzt. Daraus ergeben sich die Laufzeiten $O(t(n)+n)$ (uniform) und im logarithmischen Kostenmodell $O((t(n)+n)\log(t(n)+n))$. Beachten Sie, dass die Konstanten von $|Q|$ und $|\Sigma|$ abhängen. □

Die umgekehrte Simulation ist ein wenig komplizierter, die Laufzeiten ändern sich polynomiell.

Theorem 31 *Jede im logarithmischen Kostenmaß $t(n)$-laufzeitbeschränkte RAM mit Gesamteingabegröße n (Inhalt aller Register im logarithmischen Maß) kann für ein Polynom P durch eine $O(P(t(n)+n))$-laufzeitbeschränkte DTM M simuliert werden.*

Beweis. Wir verwenden eine 2-Band DTM M mit Zustandsmenge $Q = Q_0 \cup Q_1 \cup Q_2 \cup \cdots \cup Q_m$ wobei m die Anzahl der Programmzeilen der RAM ist. Durch die Zustandsmenge Q_0 erzeugen wir die Ausgangssituation, die Zustandsmenge Q_m stellt die Ausgabe zur Verfügung. Jede Programmzeile hat eine Zustandsmenge Q_b und durch eine Übergangsfunktion wird die Arbeitsweise der Programmzeile simuliert. Durch den jeweiligen Zustandsbereich wird sich automatisch auch der Befehlszähler b gemerkt, unabhängig von der Eingabe. Die Bänder sind wie folgt belegt:

- Band 1 enthält die aktuelle Registerbelegung in der binär kodierten Form $\#\#\#\text{bin}(R(0)\#\#\text{bin}(R(i_0))\#\text{bin}(i_0)\#\#\text{bin}(R(i_1))\#\text{bin}(i_1)\#\#\cdots$ $\#\#\text{bin}(i_k)\#\text{bin}(i_k)\#\#\#$, wobei $i_0, i_1, \ldots, i_k$ die laufenden Nummern der verwendeten Register darstellt.
- Band 2 enthält stets die aktuell betroffenen Registerinhalte und die dazugehörigen Indizes. Hier wird die Rechnung simuliert (Akkumulator).

Wir simulieren nun die Konfigurationsänderungen der RAM wie folgt. Wir befinden uns für Befehlszähler b in Zustandsmenge Q_b und simulieren durch ein Unterprogramm M_b die Registermaschine:

1. M_b kopiert die Inhalte der in Befehlszeile b angesprochenen Register und deren Nummern auf Band 2.
2. M_b führt die notwendigen Veränderungen auf Band 2 aus (Umsetzung des Befehls).
3. M_b kopiert das Ergebnis zurück auf die Registerkonfiguration auf Band 1 und löscht Band 2.
4. M_b wechselt in die Folgezustandsmenge Q_a für die nächste Befehszeile a.

Offensichtlich läßt sich die RAM durch die o.a. DTM simulieren. Die Länge des ersten Bandes ist stets durch $O(n + t(n))$ beschränkt, da wir für jedes neu generierte Bit in der RAM eine Zeiteinheit berechnen können. Auch muss die RAM die gesamte Eingabe lesen. Alle Unterprogramme vollziehen eine konstante Anzahl von Rechenschritten, die im schlimmsten Fall von der maximalen Länge des ersten Bandes abhängt aber polynomiell beschränkt ist (zum Beispiel für eine Multiplikation). Ein einzelnes Unterprogramm macht somit nicht mehr als $c_b \times (n + t(n))^{k_b}$ Rechenschritte für feste Konstanten c_b und k_b. Diese Konstanten hängen

nicht von n, sondern fest von der Befehlszeile b ab. Für nicht mehr als $t(n)$ Rechenschritte der RAM gibt es dann insgesamt stets feste Konstanten C und k, so dass die Laufzeit der DTM $C \times (n + t(n))^k$ nicht übersteigt. □

In einem weiteren Schritt könnten wir nun die Vergleichbarkeit von Turingmaschinen mit modernen Programmiersprachen untersuchen. Das wird in einigen Quellen auch gemacht. Wir verzichten hier darauf. Zunächst ist klar, dass die höheren Programmiersprachen im wesentlichen auf Assembler-Code beruhen und deshalb nicht mächtiger sein können als eine RAM oder eine DTM. Deshalb wird in der Regel *nur* untersucht, ob sich mit den Kernregeln der jeweiligen Sprache auch die DTM-berechenbaren Funktionen beschreiben lassen. Eine solchen Analyse beschränkt sich auf die wesentlichen Programmierelemente, komfortable Befehle für Ein- und Ausgabe sind beispielsweise nicht relevant für die eigentliche *Rechenleistung*. Beispielsweise werden WHILE-Programme untersucht, deren spartanische Syntax eine gezielte Analyse erlaubt.

3.9 Der Satz von Rice

Gödel's Unvollständigkeitssätze besagen, dass es für keine komplexe mathematische Theorie einen Algorithmus gibt, der alle darin formulierbaren Aussagen auf Richtigkeit beweist. So ist beispielsweise die Prädikatenlogik zweiter Ordnung (mit Quantifizierung über Relationen) unvollständig, während es für die einfache Aussagenlogik ein einfaches Kalkül gibt, das jede Aussage entscheidet.

Übertragen auf Aussagen über das Verhalten von Turingmaschinen bzw. über die Eigenschaften der von den DTMs berechneten Funtkionen ergibt sich der Satz von Rice. Die nicht-trivialen Eigenschaften der von Turingmaschinen berechneten Funktionen sind nicht entscheidbar.

Jede Turingmaschine M berechnet formal gesehen eine (möglicherweise partielle) Funktion $f_M : \{0,1\}^* \rightarrow \{0,1\}^*$, die wir leicht auf eine totale Funktion $f_M : \{0,1\}^* \rightarrow \{0,1\}^* \cup \{\perp\}$ erweitern können, wobei $\perp$ das Nichthalten der Maschine interpretiert.

Sei also $R := \{f_M : \{0,1\}^* \rightarrow \{0,1\}^* \cup \{\perp\} | M \text{ ist eine Turingmaschine}\}$ die Menge aller von DTMs berechneten Funktionen. Jede Sprache einer nicht-trivialen Teilmenge S von R ist unentscheidbar. Zum Beispiel sind

- $T := \{\langle M \rangle | M \text{ ist DTM, die auf jeder Eingabe hält}\}$
- $E_k := \{\langle M \rangle | M \text{ ist DTM, die auf genau } k \text{ Eingaben hält}\}$

nichttriviale Teilmengen.

Theorem 32 *Sei $S \subset R$ mit $S \neq R$ und $S \neq \emptyset$. Die Sprache*

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

ist nicht entscheidbar.

Beweis. Wir benutzen eine Reduktion und nutzen aus, dass in einer Übungsaufgabe bereits gezeigt wurde, dass

$$H_\epsilon := \{\langle M \rangle | M \text{ ist } DTM, \text{ die bei Eingabe von } \epsilon \text{ hält}\}$$

nicht entscheidbar ist.

Wir nehmen zunächst an, dass die nirgendwo definierte Funktion undef in S enthalten ist. Da $S \neq R$ gilt, gibt es eine Funktion $g \in R \setminus S$. Sei M_g die Maschine, die g berechnet.

Wir konstruieren die folgende Maschine M_w für eine Eingabe w:

1. Falls w keine korrekte Gödelnummer ist, halte bei keiner Eingabe x
2. Sonst simuliere $w = \langle M \rangle$ auf der leeren Eingabe. Hält diese Simulation, wird danach die Maschine M_g für beliebige Eingaben x simuliert.

Das bedeutet: M_w kann auf Wörter x angewendet werden und berechnet die folgende Funktion f_w für w:

$$f_w = \begin{cases} \text{undef} & : \text{ falls } M \text{ auf leerem Band nicht hält} \\ & \text{ oder } w \text{ keine TM darstellt, also } w \notin H_\epsilon \\ g & : \text{ sonst, also } w \in H_\epsilon \end{cases}$$

Die Funktion $h : \{0,1\}^* \to \{0,1\}^*$, die jedem w die Gödelnummer der Maschine M_w zuordnet ist DTM-berechenbar. Der erste Teil der Maschine wird aus der universellen Maschine M_U gebaut, die die Maschine $w = \langle M \rangle$ simuliert, der zweite Teil besteht aus der Maschine M_g. Nun ist sogar $M_w = M_{h(w)}$.

Wir zeigen, dass $w \in H_\epsilon \Leftrightarrow h(w) \notin L(S)$ gilt.

$$\begin{aligned} w \in H_\epsilon &\Rightarrow f_w = g \\ &\Rightarrow \text{ von } M_{h(w)} \text{ berechnete Funktion liegt nicht in } S \\ &\Rightarrow h(w) \notin L(S) \end{aligned}$$

$$\begin{aligned} w \notin H_\epsilon &\Rightarrow f_w = \text{undef} \\ &\Rightarrow \text{ von } M_{h(w)} \text{ berechnete Funktion in } S \\ &\Rightarrow h(w) \in L(S) \end{aligned}$$

Nun gilt aber auch $w \in \overline{H_\epsilon} \Leftrightarrow h(w) \in L(S)$. Nach Definition 25, Theorem 14 und Lemma 27 gilt, dass $\overline{H_\epsilon} \leq L(S)$ und $L(S)$ ist nicht entscheidbar.

Der Fall undef $\notin S$ geht analog! □

Übungsaufgabe: Führen Sie den obigen Beweis auch für undef $\notin S$.

Der Satz hat sehr starke Konsequenzen. Insbesondere kann man keine automatische Programmverifikation programmieren. Ein solcher Algorithmus müsste für alle Programme und einer gegebene Spezifikation (Funktion) entscheiden, ob das Programm diese Funktion berechnet. Vorsicht! Im Einzelfall ist das natürlich immer möglich.

3.10 Unentscheidbare Probleme ohne Selbstbezug

Alle bisher betrachteten unentscheidbaren Probleme hatten eigentlich immer etwas mit einer Aussage über bestimmte (Turing)Programme zu tun. Insgesamt resultierte daraus der Satz von Rice, der besagt, dass alle allgemeinen nicht-trivialen Aussagen über Turingmaschinen nicht entscheidbar sind.

Wir stellen hier ein paar Beispiele von klassischen unenscheidbaren Problemen vor, die zunächst (scheinbar?) gar nicht aus dem Umfeld von Turingmaschinen stammen. Es sind in der Tat

wichtige *Allerweltsprobleme*, die ebenfalls unentscheidbar sind. Die Forschungsrichtung, die sich mit solchen Problemen befaßt nennt man auch *Rekursionstheorie*. Natürlicherweise geht es dabei letztendlich immer um Aussagen über die Berechenbarkeit von Funktionen.

Das *Postsche Korrespondenzproblem*, kurz PKP, ist ein Entscheidungsproblem, das an ein spezielles Domino erinnert. Gegeben ist eine Menge von Paaren $\{(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)\}$, wobei x_i und y_i Wörter über einem endlichen Alphabet Σ (bei uns wiederum $\Sigma = \{0, 1\}$) sind. Es soll entschieden werden, ob eine Folge von Indizes $i_1, i_2, \ldots, i_n \in \{1, 2, \ldots, k\}$ existiert, so dass $x_{i_1} x_{i_2} \cdots x_{i_n} = y_{i_1} y_{i_2} \cdots y_{i_n}$ gilt. Die Folge von Indizes soll mindestens ein Element enthalten, die Elemente dürfen sich sich aber wiederholen. Die Elemente sind in diesem Sinne verwendbare Bausteine.

Für eine Menge w solcher Paare, die die Eigenschaft erfüllt, schreiben wir dann auch $w \in PKP$. Das Wort w gehört zur Sprache aller Wörter, die eine entsprechende Bausequenz ermöglichen. Offensichtlich läßt sich die Menge der Bausteine auch problemlos binär kodieren.

Bildlich lassen sich die Tupel als Bausteine interpretieren, die in einer bestimmten Reihenfolge hintereinandergelegt, oben und unten das gleiche Wort ergeben sollen; siehe Abbildung 3.2.

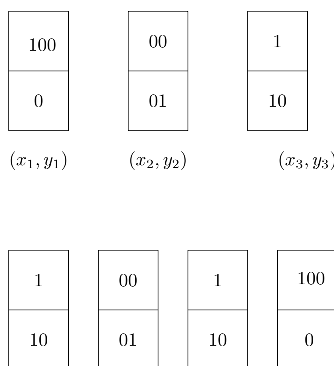

Abbildung 3.2: Eine Instanz des PKPs mit drei Bausteinen und eine Lösung.

Eine beliebte Modifikation bzw. Vereinfachung dieses Problems ist die Festlegung, dass die Folge mit einem fest vorgegebenen Startelement (x_1, y_1) beginnen muss. Beim *Modifizierten Postschen Korrespondenzproblem*, kurz MPKP, ist eine Menge von Paaren $\{(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)\}$, und ein Startpaar (x_1, y_1) gegeben, wobei x_i und y_i Wörter über einem endlichen Alphabet Σ sind. Es soll entschieden werden, ob eine Folge von Indizes $i_2, \ldots, i_n \in \{2, \ldots, k\}$ existiert, so dass $x_1 x_{i_2} \cdots x_{i_n} = y_1 y_{i_2} \cdots y_{i_n}$ gilt. Analog schreiben wir auch $w \in MPKP$.

Wir zeigen zunächst, dass sich MPKP auf PKP im Sinne der Sprachen reduzieren läßt.

Lemma 33 *Das Modifizierte Postsche Korrespondenzproblem läßt sich auf das Postsche Korrespondenzproblem reduzieren, d.h. $MPKP \leq PKP$.*

Beweis. Eine Instanz des MKPK wollen wir mittels einer Funktion f in eine Instanz des PKPs überführen, so dass die Eigenschaft der Konstruktion einer entsprechenden Bausequenz erhalten bleibt.

Wir verwenden zwei zusätzliche Symbole # und \$ und transformieren die Bausteine mittels einer Funktion f wie folgt.

Der Startbaustein (x_1, y_1) mit $x_1 = x_1^1 x_1^2 \cdots x_1^{l_{x_1}}$ und $y_1 = y_1^1 y_1^2 \cdots y_1^{l_{y_1}}$ wird zu

$$(x_1^*, y_1^*) := (\#x_1^1\#x_1^2\#\cdots\#x_1^{l_{x_1}}\#, \#y_1^1\#y_1^2\#\cdots\#y_1^{l_{y_1}})$$

transformiert. Der einzige Baustein, der mit # in x_i und y_i beginnt und somit als Startbaustein des PKPs gewählt werden *muss*.

Weiterhin werden alle Bausteine (x_i, y_i) für $i = 1, 2, \ldots, k$ mit $x_i = x_i^1 x_i^2 \cdots x_i^{l_{x_i}}$ und $y_i = y_i^1 y_i^2 \cdots y_i^{l_{y_i}}$ zu

$$(x_i', y_i') := (x_i^1 \# x_i^2 \# \cdots \# x_i^{l_{x_i}} \#, \# y_1^1 \# y_1^2 \# \cdots \# y_1^{l_{y_1}})$$

transformiert.

Für den Abschluss der Sequenz erstellen wir einen Extrabaustein

$$(x_{k+1}', y_{k+1}') = (\$, \#\$)\,.$$

Er ist der einzige Baustein, der mit dem gleichen Buchstaben in x_i und y_i endet und somit als Endbaustein des PKPs gewählt werden muss.

Zunächst ist klar, dass die Funktion f DTM-berechenbar ist. Für eine (kodierte) Menge von Bausteinen, können wir eine (kodierte) Menge von neuen Bausteinen konstruieren. Die zugehörige Funktion ist total. Jedes falsch kodiertes Wort wird auf ein falsch kodiertes Wort übertragen.

Es bleibt zu zeigen: $w \in \text{MPKP} \Leftrightarrow f(w) \in \text{PKP}$ gilt.

⇒: Für $w \in \text{MPKP}$ existiert eine Sequenz von Indizes $i_2, \ldots, i_n \in \{1, \ldots, k\}$ mit $x_1^* x_{i_2} \cdots x_{i_n} = y_1^* y_{i_2} \cdots y_{i_n}$.
Offensichtlich gilt: $x_1' x_{i_2}' \cdots x_{i_n}' x_{k+1}' = y_1' y_{i_2}' \cdots y_{i_n}' y_{k+1}'$ und somit ist $f(w) \in \text{PKP}$.

⇐: Für $f(w) \in \text{PKP}$ kann die Sequenz per Konstruktion nur mit x_1^* und y_1* beginnen und mit x_{k+1}' und y_{k+1}' enden. Also gibt es Indizes $1, i_2, \ldots, i_n, k+1 \in \{1, \ldots, k+1\}$ mit $x_1^* x_{i_2}' \cdots x_{i_n}' x_{k+1}' = y_1^* y_{i_2}' \cdots y_{i_n}' y_{k+1}'$. In beiden Sequenzen wechseln sich bis auf das Endzeichen \$ die # Zeichen mit den Zeichen aus $\{0, 1\}$ jeweils ab. Falls die Zeichen # und \$ jeweils entfernt werden, dann ergibt sich $x_1 x_{i_2} \cdots x_{i_n} = y_1 y_{i_2} \cdots y_{i_n}$, also $w \in \text{MPKP}$. □

Jetzt wollen wir noch $A \leq \text{MPKP}$ zeigen, aus der Transitivität der Reduktion folgt dann sogleich auch $A \leq \text{PKP}$ und damit ist dann die Unentscheidbarkeit von PKP gezeigt. Zur Erinnerung:

$$A := \{\langle M \rangle x | M \text{ ist } DTM, \text{ die die Eingabe von } x \text{ akzeptiert}\}\,.$$

Lemma 34 *Es gilt: $A \leq$ MPKP.*

Beweis. Für die Reduktion ist eine geeignete Transformation des Akzeptanzproblems auf ein MPKP Problem vermöge einer Funktion f notwendig. Konkret wird für eine Maschine $\langle M \rangle$ die Übergangsfunktion für eine gegebene Eingabe x in eine Reihe von Bausteinen übersetzt, so dass jeder akzeptierende Übergang genau durch eine Lösung des korrespondierenden MPKP gegeben ist. Wir werden dabei den Konfigurationen der Maschine folgen, die Startkonfiguration ist stets $q_1 x$ für den Startzustand q_1

Wir nehmen der Einfachheit halber an, dass der Buchtstabe # in der DTM zur Kodierung nicht benutzt wird und Blanks nur für das Ende der Eingabe verwendet werden. Das Konzept läßt sich leicht erweitern. Wir benutzen # als Hilfsbuchstabe zur Konstruktion der Bausteine (auch aus Konsistenzgründen zu andere Literaturquellen). Im allgemeinen sei $\langle M \rangle x$ gegeben und q_1 der Startzustand und q_n der Endzustand. Der *Startbaustein* ist:

$$\left[\frac{\#}{\#q_1x\#}\right]$$

Wir erzwingen dadurch, das oben als erstes q_1x hinzugefügt werden muss. Versetzt darunter werden wir dann die Konfigurationen nachbilden. Da wir die Konfiguration schrittweise nachbauen wollen, gibt es *Kopierbausteine* für die Elemente:

$$\left[\frac{\#}{\#}\right]\left[\frac{q_i}{q_i}\right]\left[\frac{a}{a}\right] \text{ für } a \in \{0,1\}$$

Nur das Kopieren wird nicht ausreichen, da dann unten immer wieder eine neue zusätzliche Kopie entsteht. Deshalb müssen die Übergänge während des Kopierens unten bereits mitkodieren. Für jedes $\delta(q,a) = (q',b,N)$ mit $q \neq q_n$ konstruieren wir $(qa, q'b)$, für jedes $\delta(q,a) = (q',b,R)$ mit $q \neq q_n$ konstruieren wir (qa, bq') und für jedes $\delta(q,a) = (q',b,L)$ mit $q \neq q_n$ konstruieren wir $(cqa, q'cb)$ für alle $a,b,c \in \{0,1\}$.

Übergangsbausteine für $q \neq q_n$, $a,b,c \in \{0,1\}$:

$$\left[\frac{qa}{q'b}\right]\left[\frac{qa}{bq'}\right]\left[\frac{cqa}{q'cb}\right]$$

Am Ende des Bandes müssen auch Blanks behandelt werden, das machen wir implizit durch Weglassen. Für jedes $\delta(q,\sqcup) = (q',b,N)$ mit $q \neq q_n$ konstruieren wir $(q\#, q'b\#)$, für jedes $\delta(q,\sqcup) = (q',b,R)$ mit $q \neq q_n$ konstruieren wir $(q\#, bq'\#)$ und für jedes $\delta(q,\sqcup) = (q',b,L)$ mit $q \neq q_n$ konstruieren wir $(cq\#, q'cb\#)$ für alle $a,b,c \in \{0,1\}$.

Spezielle Übergangsbausteine für $q \neq q_n$, $a,b,c \in \{0,1\}$:

$$\left[\frac{q\#}{q'b\#}\right]\left[\frac{q\#}{bq'}\right]\left[\frac{cq\#}{q'cb\#}\right]$$

Mit den bisherigen Bausteinen können wir die Konfigurationswechsel unten realisieren. Falls wir unten eine Endkonfiguration erreicht haben, muss das Wort oben noch damit komplettiert werden, dazu benötigen wir die *Löschbausteine* für den Endzustand q_n für $a \in \{0,1\}$:

$$\left[\frac{aq_n}{q_n}\right]\left[\frac{q_na}{q_n}\right]$$

Um die Simulation abzuschließen, ist der *Abschlussbaustein* notwendig:

$$\left[\frac{\#q_n\#}{\#}\right]$$

Wir verdeutlichen die Konstruktion an einem Beispiel. Betrachte dazu die folgende DTM $M = \{\{0,1,\#,\$,\sqcup\},\{q_1,q_2,q_3\},\delta,q_1,\{q_3\}\}$, die die Sprache $L = \{w|w1 \in \{0,1\}^*\}$ entscheidet. Die Sprache aller Binärwörter, die mit einer 1 enden.

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

Eine akzeptierende Konfiguration sieht wie folgt aus:

$$q_1101 \vdash 1q_201 \vdash 10q_11 \vdash 101q_2 \vdash 101q_31$$

Übersetzt in die Bausteinwelt erhalten wir dafür:

$$\left[\frac{\#}{\#q_1 101\#}\right] \qquad \begin{array}{l} \left[\frac{q_1 1}{1 q_2}\right] \left[\frac{0}{0}\right] \left[\frac{1}{1}\right] \left[\frac{\#}{\#}\right] \\ \left[\frac{1}{1}\right] \left[\frac{q_2 0}{0 q_1}\right] \left[\frac{1}{1}\right] \left[\frac{\#}{\#}\right] \\ \left[\frac{1}{1}\right] \left[\frac{0}{0}\right] \left[\frac{q_1 1}{1 q_2}\right] \left[\frac{\#}{\#}\right] \\ \left[\frac{1}{1}\right] \left[\frac{0}{0}\right] \left[\frac{1}{1}\right] \left[\frac{q_2 \#}{q_3 1 \#}\right] \end{array}$$

Wir sehen, dass für die Berechnungsschritte die Konfiguration jeweils *unten* textuell aufgeführt wird und die Vorgängerkonfiguration jeweils *oben* steht. Damit die Konstruktion erfolgreich beendet werden kann, werden die Löschbausteine verwendet:

$$\cdots \qquad \begin{array}{l} \left[\frac{1}{1}\right] \left[\frac{0}{0}\right] \left[\frac{1}{1}\right] \left[\frac{q_2 \#}{q_3 1 \#}\right] \\ \left[\frac{1}{1}\right] \left[\frac{0}{0}\right] \left[\frac{1}{1}\right] \left[\frac{q_3 1}{q_3}\right] \left[\frac{\#}{\#}\right] \\ \left[\frac{1}{1}\right] \left[\frac{0}{0}\right] \left[\frac{1 q_3}{q_3}\right] \left[\frac{\#}{\#}\right] \\ \left[\frac{1}{1}\right] \left[\frac{0 q_3}{q_3}\right] \left[\frac{\#}{\#}\right] \\ \left[\frac{1 q_3}{q_3}\right] \quad \left[\frac{\# q_3 \#}{\#}\right] \end{array}$$

Es ist allgemein zu zeigen, dass $A \leq$ MPKP gilt. Zunächst ist klar, dass die Funktion f DTM-berechenbar ist, jedes Wort $\langle M \rangle x$ kann in das entsprechende MPKP Problem überführt werden. Wir müssen nun allgemein zeigen, dass $\langle M \rangle x \in A \Leftrightarrow f(\langle M \rangle x) \in$ MPKP gilt.

$\Rightarrow$: Für $\langle M \rangle x \in A$ gibt es eine Folge von Konfigurationen $K_1 \vdash K_2 \vdash \cdots K_{t-1} \vdash K_t$ wobei K_1 im Startzustand q_1 und K_t im Endzustand q_n liegt.

Wir können wie oben erwähnt beginnend mit dem Startbaustein die Konfigurationsfolge unten mit Hilfe der Kopier- und Übergangsbausteine durch den String

$$\#K_1\#K_2\#\cdots\#K_{t-1}\#K_t\#$$

nachbauen. Oben steht dann nach Konstruktion der String

$$\#K_1\#K_2\#\cdots\#K_{t-1}\#\,.$$

Dieser läßt sich nun durch die Löschbausteine zum ersten String bis auf den fehlenden Abschluss $\#q_n\#$ erweitern. Nach Hinzufügen von $\left[\frac{\#q_n\#}{\#}\right]$ sind beide Strings identisch, also $f(\langle M \rangle x) \in$ MPKP.

$\Leftarrow$: Sei nun $\langle M\rangle x \notin A$. Wenn $\langle M\rangle$ keine gültige Maschine ist, kann f auf eine Instanz des MPKP abbilden, die nicht lösbar ist und wir sind fertig.

Sonst kann nach Konstruktion von $f(\langle M\rangle x)$ eine Lösung nur mit dem Startbaustein beginnen und muss dann irgendwann entweder den Endbaustein oder einen ersten Löschbaustein enthalten. Ansonsten kann die Anzahl der Buchstaben der beiden Strings niemals gleich sein.

Sei also $1, i_1, i_2, \ldots i_s$ die Teilfolge einer MPKP Lösung, so dass i_s der erste Lösch- oder Abschlussbaustein ist. Hier taucht im oberen String zum ersten Mal der Endzustand auf.

Dazwichen kann es dann nur Kopier- oder Übergangsbausteine gegeben haben. Diese Bausteine sind so konzipiert, dass sie nur richtige Konfigurationsfolgen versetzt um eine Konfiguration sukzessive *schreiben* können. Der untere String ist stets eine Konfiguration weiter. Dann muss die Maschine eine Konfigurationsfolge haben, die in einem Endzustand gerät. Somit hält die Maschine auf der Eingabe, ein Widerspruch. □

Weitere unendscheidbare Probleme sind beispielsweise das *Game of Life*. Dabei geht es um die Frage, ob Vorhersehbar ist, wie sich eine Population mit einfachen Veränderungsregeln entwickeln wird.

Ein anderes mathematisch motiviertes unentscheidbares Problem ist das Folgende:

Gegeben sind fünf affine Abbildungen $f_1, f_2, \ldots, f_5$. Eine affine Abbildung f ist definiert durch eine 2×2 Matrix A und einem Vector B mit jeweils rationalen Koeffizienten durch $f : \mathbb{Q}^2 \rightarrow \mathbb{Q}^2$ mit $f(v) = A \cdot v + B$.

Das zugehörige unentscheidbare Entscheidungsproblem lautet: Seien $q = (q_x, q_y)$ und $q' = (q_x, q_y)$. Gibt es ein endliches Produkt von Abbildungen aus $\{f_1, f_2, \ldots, f_5\}$ so dass $q = (q_x, q_y)$ auf $q' = (q_x, q_y)$ abgebildet werden kann. Wie man sich ggf. leicht vorstellen kann, gibt es eine Reduktion von PKP auf dieses Problem.

Kapitel 4

Komplexität und die Klassen P und NP

Nachdem wir uns in den vorherigen Abschnitten zunächst konkrete Maschinenmodelle eingeführt und danach formell gezeigt haben, dass es Problemstellungen gibt, die grundsätzlich nicht durch ein Computerprogramm gelöst oder entschieden werden können, wenden wir uns nun der Frage der Zeitkomplexität entscheidbarer Probleme zu. Entscheidbare Problemstellungen können so *schwer* zu lösen sein, so dass aus praktischer Sicht keine effiziente (bezüglich Laufzeit und/oder Speicherplatz) Implementierung möglich ist.

Im Vorlesungsteil Algorithmen und Datenstrukturen hatten wir als Zeit- und Platzkomplexitätsmaß die O-Notation eingeführt, Algorithmen im Pseudocode formuliert und einzelne Berechnungsschritte durch Konstanten abgeschätzt. Wir haben dabei festgestellt, dass viele klassische Problemstellungen in *Polynomialzeit* in Bezug auf die Eingabegröße gelöst werden können.

Übertragen auf unsere Maschinenmodelle entspricht diese Vorgehensweise im wesentlichen einer RAM im uniformen Kostenmaß. Solang die verwendeten Objekte und Zahlen nicht zu groß werden, ist die Annahme, dass eine einzelne Rechenoperation in konstanter Zeit ausgeführt werden kann insgesamt sinnvoll. Die Programme, die wir im Pseudocode formulieren können, lassen sich stets durch eine RAM oder eine Turingmaschine in annähernd (polynomieller Faktor) gleicher Laufzeit realisieren und umgekehrt. Deshalb verwenden wir im Folgenden auch gelegentlich Pseudocode. Im Vorlesungsteil Algorithmen und Datenstrukturen waren obere und untere polynomielle Laufzeitschranken von Interesse. Das wird hier unerheblich sein, wir wollen eher wissen, ob es polynomielle obere Schranken gibt oder nicht. Wir können daher großzügiger mit den Laufzeiten umgehen.

Es ist außerdem aus praktischer Sicht sinnvoll anzunehmen, dass eine einzelne Rechnenoperation für angemessen große Zahlen in konstanter Zeit ausführbar ist. Alle modernen Rechner verwenden beispielsweise eine fest eingebaute Floating-Point Arithmetik nach IEEE Standard die in einem vorgegebenen relativ großen Zahlenbereich sehr effizient arbeitet.

Vorsicht ist allerdings geboten. Es ist nicht erlaubt, beliebig große Zahlen zu verwenden. Dadurch ließen sich vollständige (exonentiell große) Lösungen in einzelnen Zahlen *konstant* kodieren. Wenn wir also beispielsweise eine RAM verwenden, müssen wir im Prinzip das logarithmische Kostenmaß verwenden.

4.1 Entscheidungs- und Optimierungsprobleme

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

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

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

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

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

TSP-Entscheidungsproblem: Gegeben $G = (V, E)$ mit Kantengewichtsfunktion $c : E \to \mathbb{N}$ und $t \in \mathbb{N}$.
Frage: Gibt es eine Rundreise mit Gesamtkosten $\leq t$?

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

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

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

1: Min := 0; Max := k;
2: **while** Max − Min ≥ 1 **do**
3: $\quad t := \left\lfloor \frac{\text{Min}+\text{Max}}{2} \right\rfloor$
4: $\quad$ *Löse Entscheidungsproblem mit* t;
5: $\quad$ **if** Antwort JA **then**
6: $\quad\quad$ Max := t;
7: $\quad$ **else**
8: $\quad\quad$ Min := $t + 1$;
9: $\quad$ **end if**
10: **end while**

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

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

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

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

1: *Bestimme optimale Kosten* t_0;
2: **for** alle $e \in E$ **do**
3: $\quad c(e) := c(e) + 1$;
4: $\quad$ *Bestimme optimale Kosten* t'_0;
5: $\quad$ **if** $t'_o > t_0$ **then**
6: $\quad\quad c(e) := c(e) - 1$; // Kante e wird gebraucht
7: $\quad$ **end if**
8: **end for**

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

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

Übungsaufgabe: Formulieren Sie für das Sortieren von n natürlichen Zahlen ein Entscheidungsproblem und eine Reduktion. Das Sortieren soll als funktionales Problem mittels des Entscheidungsproblems gelöst werden.

4.2 Zeitkomplexität von Turingmaschinen

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

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

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

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

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

Betrachten wir zum Beispiel nochmal die Maschine aus Beispiel 3 im Abschnitt 2.2, die die Sprache $L = \{0^k 1^k | k \geq 1\}$ entscheidet. Wir verzichten im Folgenden auf die genaue Ma-

schinenbeschreibung bzw. auf Implementierungsdetails und erläutern systematisch, was die Maschine tut.

1. Durchlaufe die Eingabe. Falls eine Null *nach* einer Eins auftaucht, lehne ab.
2. Wiederhole die folgenden Schritte, solange noch eine Eins *und* eine Null auf dem Band stehen.
3. Durchlaufe das Band von links nach rechts, lösche dabei die letzte Eins und die erste Null. Gehe nach rechts zum letzten Zeichen.
4. Falls alle Zahlen gelöscht wurden, akzeptiere die Eingabe. Falls nur noch Einsen und keine Null mehr auf dem Band steht oder falls nur noch Nullen und keine Eins mehr auf dem Band steht, lehne das Wort ab.

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

Definition 35 *Sei $t : \mathbb{N} \to \mathbb{N}$ eine monoton wachsende Funktion. Die Klasse DTIME$(t(n))$ ist definiert als:*

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

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

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

Beschreibung einer DTM M:

1. Durchlaufe die Eingabe. Falls eine Null *nach* einer Eins auftaucht, lehne ab. Sonst geh zurück zum Anfang des Bandes.
2. Wiederhole die folgenden Schritte, solange noch eine Eins *und* eine Null auf dem Band stehen.
3. Durchlaufe das Band von rechts nach links und stelle dabei fest, ob die Anzahl Einsen und Nullen beide gerade oder beide Ungerade sind. Lehne ab, falls dies nicht der Fall ist. Sonst gehe zurčk zum Beginn der Eingabe.
4. Durchlaufe das Band und streich jede zweite Null und jede zweite Eins, beginnend mit der ersten Null und der ersten Eins.
5. Falls nur noch Einsen und keine Null mehr auf dem Band steht oder falls nur noch Nullen und keine Eins mehr auf dem Band steht, lehne das Wort ab.

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

4.3 Die Klasse P

Die Theoreme 30 und 31 zeigen, dass die folgende Definition effizient lösbarer Probleme unabhängig vom jeweiligen Maschinenmodell (1-Band DTM, Mehrband DTM, RAM) betrachtet werden kann.

Definition 36 *Die Klasse P ist definiert als*

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

Die Klasse P ist also die Menge aller Sprachen, für die es ein beliebiges aber festes k gibt, so dass eine DTM M existiert, die die Sprache L in Laufzeit $O(n^k)$ entscheidet. Die Klasseneinteilung ist sinnvoll, es handelt sich um Problemstellungen für die

- $\cdots$ wir eine mathematisch konkrete polynomielle Abschätzung vornehmen.
- $\cdots$ die aus praktischer Sicht effizient lösbar sind.

Für die effiziente Lösbarkeit aus praktischer Sicht ist es notwendig, dass der zugehörige konstante Exponent k nicht zu groß wird. $k = 1000$ ist zum Beispiel aus praktischer Sicht für große n definitiv nicht mehr effizient. Es zeigt sich aber aus Erfahrung, dass für alle klassischen Problemstellungen auch relativ kleine konstante $k = 2, 3, 4, \ldots$ gefunden werden können. Beispiele für (funktionale) Problemstellungen aus P sind:

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

Wir wollen zwei Beispiele nochmal explizit betrachten und dabei unsere bisher vorgestellten Maschinenmodelle verwenden.

Beispiel 1: Gegeben ist ein gerichteter Graph $G = (V, E)$ und zwei Knoten s, t aus V. Gibt es in G einen gerichteten Pfad von s nach t.

Zunächst kann $G = (V, E)$ mit s und t binär als $\langle V, E, s, t\rangle$ kodiert werden. Als Sprache ausgedrückt suchen wir eine DTM M, die die Sprache

$$\text{PATH} := \left\{ \langle V, E, s, t\rangle \;\middle|\; \begin{array}{l} G = (V, E) \text{ ist ein gerichteter Graph mit } s, t \in V \\ \text{und einem gerichteten Pfad von } s \text{ nach } t \end{array} \right\}$$

entscheidet.

Lemma 37 *PATH $\in P$.*

Beweis. Wir geben eine DTM M an, die PATH entscheidet. Diese DTM implementiert eine Breitensuche.

1. Verwalte eine Menge $S = \{s\}$ und markiere s.
2. Wiederhole die folgenden Schritte, bis die Menge S leer ist oder t erreicht wird.
3. Für alle $a \in S$ betrachte alle Kanten $(a, b) \in E$ in denen b nicht markiert ist, füge alle diese bs in S ein, markiere alle diese bs und lösche die as aus S.
4. Falls $t \in S$ zu einem Zeitpunkt gilt, akzeptiere, sonst lehne ab.

Offensichtlich werden alle von s aus erreichbaren Knoten irgendwann in S geführt, wobei beim i-ten Durchlauf von Schritt 3. alle Knoten die mit i Schritten von s aus erreichbar sind, in S eingefügt und markiert werden. Falls diese Knoten später nocheinmal erreicht werden sollten, werden die Kanten ignoriert.

Die relevanten Informationen müssen sinnvoll auf ein Band gespeichert werden, um Schritt 3. auszuführen. Für jeden Knoten wird markiert, ob er aktuell in der Menge S liegt und nach Schritt 3. liegen wird. Für jede Kante wird notiert, ob der Endknoten markiert wurde.

Schritt 1. und Schritt 4. werden nur einmal durchlaufen. In Schritt 3. werden im schlimmsten Fall immer alle Kanten $|E| = m$ und alle Knoten $|V| = n$ betrachtet. Wir müssen dann stets die gesamte Eingabe der Größe $O(n+m)$ für alle aktuellen Knoten durchlaufen. Jeder Knoten ist aber nur genau einmal aktuell. Also ergibt sich eine Laufzeit von $O(n(n + m))$ für eine Eingabe der Größe $O(n + m)$. □

Beispiel 2 (Sortieren): Gegeben N binär kodierte Zahlen $a_1, a_2, \ldots, a_N \in \mathbb{N}$. Gesucht ist die aufsteigend sortierte Folge dieser Zahlen.

Lemma 38 *Sortieren liegt in P.*

Beweis. Wir verwenden eine RAM und bilden beispielsweise Insertionsort-Algorithmus durch die RAM ab. Sei dazu n die Eingabelänge, also die Anzahl aller Bits zur Kodierung der N Eingabezahlen $a_1, a_2, \ldots, a_N$.

Im uniformen Kostenmaß erhalten wir eine Laufzeit von $O(N^2)$ im worst-case. Für $l = \max_{1 \leq i \leq N} \log a_i$ kann jeder uniforme Schritt in Zeit $O(l)$ auf der RAM ausgeführt werden. Dann ist die Laufzeit der RAM im logarithmischen Zeitmaß $O(l \cdot N^2)$. Aus $l \leq n$ und $N \leq n$ folgt $l \cdot N^2 \leq n^3$. Die Laufzeit auf der RAM ist somit polynomiell beschränkt. □

4.4 Nichtdeterministische Turingmaschinen

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

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

.

Es ist kein polynomieller Algorithmus bekannt, der die Sprache Clique entscheidet. Fallls wir aber einen Kandidaten von k Knoten haben, für den wir die Cliqueneigenschaft überprüfen wollen, dann geht das sehrwohl in polynomieller Zeit. Alle momentan bekannten Algorithmen zur Lösung des Cliquenproblems testen im wesentlichen die Beweiskandidaten. Nehmen wir also an, wir hätten einen Mechanismus, der für uns den richtigen Beweiskandidaten *rät*. Dann können wir das Problem in polynomieller Zeit lösen. Das führt zur Definition der nichtdeterministischen Turingmaschinen NTM.

Definition 39 *Eine nichtdeterministische k-Band Turingmaschine (NTM) ist ein 5-Tupel $N = (\Sigma, Q, \delta, q_1, F)$ wobei Σ, Q, q_0 und F genauso definiert sind wie bei der DTM und die Zustandsübergangsfunktion ist von der Form*

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

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

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

Wir betrachten das folgende **Beispiel** eine NTM N. Wie bislang ist q_1 der Startzustand und q_3 der Endzustand.

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

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

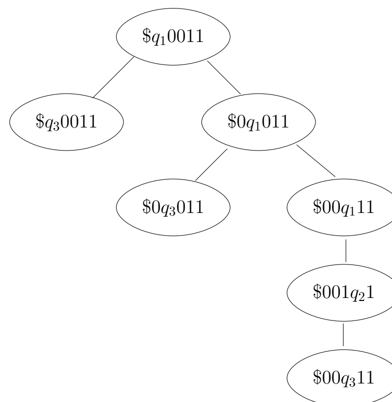

Abbildung 4.1: Der Berechnungsbaum der Maschine N.

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

Definition 40 *Eine NTM N akzeptiert die Eingabe $x \in \Sigma^*$, falls es mindestens eine Berechnung gibt, die in eine akzeptierende Endkonfiguration führt.*

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

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

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

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

Definition 41 *Sei N eine NTM. Die Laufzeit $T_N(w)$ von N bei einer Eingabe $w \in \Sigma^*$ ist definiert durch:*

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

.

Für $n \in \mathbb{N}$ ist dann

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

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

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

Definition 42 *Sei $t : \mathbb{N} \to \mathbb{N}$ eine monoton wachsende Funktion. Die Klasse NTIME($t(n)$) ist definiert als:*

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

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

4.5 Die Klasse NP

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

Definition 43 *Die Klasse NP ist definiert als*

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

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

Lemma 44 *Clique* $\in NP$.

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

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

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

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

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

Das Problem der Clique haben wir mit Hilfe des Nichtdeterminismus gelöst. Dazu wurde der richtige Beweiskandidat geraten. Ein generierter Beweiskandidat kann also in diesem Sinn auch als *Zertifikat* bezeichnet werden, dass wir in polynomieller Zeit mit der jeweiligen Maschine überprüfen können. Diese Sicht der Dinge werden wir im Folgenden präzisieren.

Vorher formulieren wir noch einige Beispiele für Sprachen bzw. Problemstellungen, die in NP liegen.

4.5.1 Beispiele für Probleme aus NP

Rucksackproblem: Gegeben ist eine Menge von Gewichten $w_1, w_2, \ldots, w_N \in \{1, 2, \ldots, b\}$ mit einem Rucksack der Kapazität $b \in \mathbb{N}$ und Preisen $p_1, p_2, \ldots, p_N$.
Gesucht ist eine zulässige Lösung mit maximalen Preis, also eine Teilmenge $K \in \{1, 2, \ldots, N\}$ mit $\sum_{i\in K} w_i \leq K$ mit maximalen Wert $\sum_{i\in K} p_i$ unter allen solchen Teilmengen K.

Das Rucksackproblem kann auch als Entscheidungsproblem definiert werden, dabei wird entschieden, ob es für die Eingabe eine Lösung einer bestimmten Güte gibt oder nicht. Wie in jedem Fall kann durch die Lösung des Entscheidungsproblems auch eine Lösung des Optimierungsproblems oder des obigen Funktionalen Optimierungsproblems gefunden werden. Der *zusätzliche* Aufwand ist polynomiell beschränkt.

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

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

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

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

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

Für eine gegebene Belegung gilt folgende Interpretation:

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

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

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

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

Ein weiterer Spezialfall ist:

Max-k-SAT: Gegeben: Ein aussagenlogischer Ausdruck $\alpha = k_1 \wedge k_2 \wedge \cdots \wedge k_\ell$ mit Klauseln vom Grad $\leq k$ in konjunktiver Normalform mit insgesamt m verwendeten Variablen $x_1, x_2, \ldots, x_m$,

und eine Zahl t.
Frage: Gibt es eine Belegung der Variablen, so dass mindestens t Klauseln aus $k_1, k_2, \ldots, k_l$ true sind?

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

Übungsaufgabe: Zeigen Sie 2-SAT liegt in P.

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

Auch für Set-Cover gibt es Spezialfälle:

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

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

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

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

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

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

Wir zeigen nochmal exemplarisch an zwei Beispielen, warum die beschriebenen Probleme in NP liegen.

Lemma 45 *SAT liegt in NP.*

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

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

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

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

Lemma 46 *Subset-Sum liegt in NP.*

Beweis. Durch eine nichtdeterministische NTM können wir die richtige Teilmenge T in maximal N Schritten raten.

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

4.6 Klasse NP durch Zertifikate charakterisieren

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

Definition 47 *Für eine Sprache L heißt eine DTM V* polynomieller Verifizierer *von L, falls es Polynome p und q gibt mit der folgenden Eigenschaft existiert:*

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

Die Sprache L heißt dann polynomiell verifizierbar.

Lemma 48 *Eine Sprache L liegt genau dann in NP, falls die Sprache polynomiell verifizierbar ist.*

Beweis. Sei $L \in NP$. Dann gibt es einen NTM N, die $x \in L$ mit $|x| = n$ in polynomieller Zeit $p(n)$ akzeptiert. Wir vereinfachen die nichtdeterministische Übergangsfunktion dadurch,

so dass immer maximal zwei nichtdeterministische Übergänge existieren. Die beiden alternativen Übergänge bezeichnen wir dann respektive mit 0 oder 1. (Die Anzahl der Übergänge ist allgemein stets durch die konstante Anzahl von Elementen aus Σ und Q beschränkt, deshalb lässt sich diese Betrachtung entsprechend ohne Einschränkung realisieren. Der Verzweigungsgrad der Maschine ist durch eine Konstante k beschränkt und hierfür würden $\log k$ Bits reichen. Die unten angesprochene Länge des Zertifikats y und die Laufzeit von V bleiben polynomiell.)

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

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

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

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

□

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

Wir wenden das Lemma an einem Beispiel an.

Lemma 49 *Das TSP-Entscheidungsproblem liegt in NP.*

Beweis. Als Sprache formulieren wir das Problem durch:

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

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

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

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

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

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

4.6.1 P und NP

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

Das vielleicht bekannteste Problem der Informatik lautet wie die Frage

$$P = NP?$$

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

Im Prinzip stellt sich die Frage, ob man durch eine intelligente Vorgehensweise auf die Generierung exponentiell vieler Beweiskandidatn verzichten kann, eine solche Vorgehensweise ist unbekannt. Für die Lösung des Problems kann man mit einer Belohnung von 1 Millionen US Dollar rechnen.

4.7 NP-Vollständigkeit

Aus $P \subseteq NP$ folgt, dass nicht alle Probleme in NP schwer zu lösen sind. Deshalb möchten wir gerne eine Aufteilung der Probleme in NP vornehmen und der Begriff der NP-Vollständigkeit wird diesen Aspekt abdecken und NP vollständig sind dann die *schweren* Probleme aus der Klasse NP. Das Konzept wird dadurch beschrieben, dass es ausreicht nur für eine NP-vollständige Sprache einen deterministischen polynomiellen Algorithmus zu entwerfen, um vollständig alle anderen solche Probleme polynomiell zu lösen. Wir werden zeigen das SAT NP-vollständig ist und daraus die NP-Vollständigkeit weiterer Problem folgern wie zum Beispiel TSP oder Clique.

4.7.1 Polynomielle Reduktion

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

Definition 50 *Seien L_1 und L_2 zwei Sprachen über $\{0,1\}$.*

L_1 ist polynomiell reduierbar *auf L_2, wenn es eine Reduktion von L_1 nach L_2 mittels einer Funktion f mit $f : \Sigma^* \to \Sigma^*$ gibt und f in polynomieller Zeit berechenbar ist. Wir schreiben dann auch $L_1 \leq_p L_2$.*

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

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

gilt und für f gibt es eine DTM M, die f in polynomieller Laufzeit berechnet. Die Definition ermöglicht es, einfache Übertragungen von Komplexitäten vorzunehmen.

Lemma 51 *Falls $L_1 \leq_p L_2$ und $L_2 \in P$, dann gilt auch $L_1 \in P$.*

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

Dieser deterministische Algorithmus B ist in der Laufzeit polynomiell beschränkt. Zunächst ist f durch ein Polynom p beschränkt. Das heißt auch $|f(x)| \leq p(|x|) + |x|$, zumindest die Eingabe muss berücksichtigt werden und jedes Schreiben einer weiteren Ausgabe kostet auch eine Zeiteinheit. Die Laufzeit von A wiederum ist polynomiell in der Größe der Eingabe $|f(x)|$ durch ein Polynom q beschränkt. Die Gesamtlaufzeit von B für eine Eingabe x der Größe $|x|$ ist dann beschränkt durch $p(|x|) + q(p(|x|) + |x|) \leq C \cdot h(|x|)$ für eine Polynom h. □

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

Beispiele für polynomielle Reduktionen werden wir im übernächsten Abschnitt behandeln.

4.7.2 Definition der NP-Vollständigkeit

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

Definition 52 *Ein Problem L heißt NP-schwer (NP-hard im Englischen), falls gilt:*

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

Daraus folgt sofort:

Theorem 53 *L NP-schwer und $L \in P$, dann folgt $P = NP$.*

Beweis. Sei L' ein beliebiges Problem aus NP. Es gilt $L' \leq_p L$. Aus $L \in P$ folgt nach Lemma 51, dass auch $L' \in P$ gilt. Dann gilt $NP \subseteq P$ und somit $NP = P$. □

Definition 54 *Ein Problem L heißt NP-vollständig (NP-complete im Englischen), falls gilt:*

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

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

4.7.3 Beispiele polynomieller Reduktionen

In diesem Abschnitt zeigen wir für ein paar ausgewählte Probleme, dass sie NP-vollständig sind gemäß Definition 54. Grundsätzlich muss man hierfür für ein Problem $L \in NP$ zeigen, das *alle* anderen Probleme $L' \in NP$ polynomiell auf L reduziert werden können. Nutzen wir die Aussage aus obiger Übungsaufgabe (nach Lemma 51), dann genügt es zu zeigen, dass ein beliebiges NP-schweres Problem L' auf L reduziert werden kann. Das Theorem 57 wird uns sagen, dass SAT NP-vollständig ist. Im folgenden Beweis wird dann zunächst $SAT \leq_p$ 3-SAT gezeigt. Den Beweis des Theorems 57 führen wir im Anschluss an diesen Abschnitt.

Wir beweisen nun das folgende Theorem. Die angegebenen Reduktionen können auch in dem Buch [1] von Blum nachgelesen werden.

Theorem 55 *Diese Probleme sind NP-vollständig:*

1. 3-SAT
2. *Clique*
3. *Knotenüberdeckung*

Beweis. Wir reduzieren zunächst $SAT \leq_p$ 3-SAT, dann 3-$SAT \leq_p$ *Clique* und schließlich *Clique* $\leq_p$ *Knotenüberdeckung*. Der Beweis des Theorems folgt dann aus Theorem 57.

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

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

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

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

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

   wobei α, α' aussagenlogische Formeln in konjunktiver Normalform sind, jeweils bestehend aus Klauseln vom Maximalgrad k bzw. 3. Sei $\alpha = k_1 \wedge \cdots \wedge k_\ell$ eine gültige SAT-Instanz über m Variablen $x_1, \ldots, x_m$, und sei $k_i = L_1 \vee \cdots \vee L_n$, $4 \leq n \leq k$, eine beliebige Klausel in α. Wir führen $n-3$ neue Variablen $A_0, \ldots, A_{n-4}$ ein, und ersetzen k_i durch eine Konjunktion k von $n-2$ Klauseln vom Grad 3, über den Variablen $x_1, \ldots, x_n, A_0, \ldots, A_{n-4}$. Durch die Konstruktion wird sichergestellt sein, dass k genau dann durch eine Belegung der Variablen $A_0, \ldots, A_{n-4}$ erfüllt werden kann, wenn k_i bereits durch eine Belegung der Variablen $x_1, \ldots, x_n$ erfüllt ist. Wir definieren nun

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

   Da die Konstruktion von k offensichtlich in Polynomialzeit möglich ist, genügt es nun zu zeigen, dass genau dann wenn k_i durch eine Belegung der Variablen $x_1, \ldots, x_m$ erfüllt

ist, k durch eine Belegung der Variablen $A_0, \ldots, A_{n-4}$ erfüllt werden kann. Wir zeigen zunächst, dass wenn k_i nicht erfüllt ist, k auch nicht erfüllt werden kann.

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

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

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

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

2. Wir haben *Clique* $\in NP$ bereits zuvor bewiesen. Wir zeigen nun, dass *Clique* NP-schwer ist durch die Reduktion 3-$SAT \leq_p$ *Clique*. Wir konstruieren nun aus einer gegebenen 3-SAT Instanz $\alpha = k_1 \wedge \cdots \wedge k_\ell$ einen Graphen $G = (V, E)$, so dass G genau dann eine l-Clique enthält wenn α erfüllbar ist. Falls in α eine Klausel weniger als drei Literale enthält, duplizieren wir zunächst in jeder dieser Klauseln ein Literal, so dass wir annehmen können, dass jede Klausel genau drei Literale enthält. Nun nummerieren wir die Literale derart, dass in Klausel k_i genau die Literale $L_{3i-2}, L_{3i-1}, L_{3i}$ enthalten sind. In der ersten Klausel sind folglich die Literale L_1, L_2, L_3 enthalten, in der zweiten Klausel die Literale L_4, L_5, L_6 und so weiter. Hierbei steht jedes Literal L_j jeweils entweder für einen Ausdruck x_r oder $\neg x_r$, wobei der Index r natürlich in j variieren kann.

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

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

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

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

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

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

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

Definition 56 *Für einen beliebigen Graphen $G = (V, E)$ bezeichne $\overline{G} = (V, \{V \times V\} \setminus E)$ den Komplementärgraphen von G.*

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

G enthält eine k-Clique C

$\Leftrightarrow$

Je zwei Knoten in C sind in G durch eine Kante verbunden

$\Leftrightarrow$

Keine zwei Knoten in C sind in $\overline{G}$ durch eine Kante verbunden

$\Leftrightarrow$

$V \setminus C$ ist eine Knotenüberdeckung von $\overline{G}$ der Größe $|V| - k$

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

□

4.7.4 Der Satz von Cook und Levin

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

Theorem 57 (Satz von Cook und Levin) *Das Problem SAT ist NP-vollständig.*

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

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

- Für die Variablenmenge $V = \{x_1.x_2, x_3, \ldots\}$ legt φ mit $\varphi : V \to \{0, 1\}$ eine Belegung fest.
- Sei y ein Literal, dann ist

$$\varphi(y) := \begin{cases} \varphi(x_i) & \text{falls } y = x_i \\ 1 - \varphi(x_i) & \text{falls } y = \neg x_i \end{cases}$$

- Für Literale $y_1, y_2, \ldots y_k$ und die Klausel $(y_1 \vee y_2 \vee \cdots \vee y_k)$ gilt:

$$\varphi((y_1 \vee y_2 \vee \cdots \vee y_k)) := \max\{\varphi(y_i) | 1 \leq i \leq k\}\,.$$

- Für den aussagenlogischen Ausdruck $k_1 \wedge k_2 \wedge \cdots \wedge k_l$ für Klausel $k_1, k_2, \ldots, k_l$ gilt:

$$\varphi(k_1 \wedge k_2 \wedge \cdots \wedge k_l) := \min\{\varphi(k_j) | 1 \leq j \leq l\}\,.$$

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

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

Lemma 58 *Für die Variablenmenge $V = \{x_1, x_2, \ldots, x_l\}$ ist der aussagenlogische Ausdruck*

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

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

Übungsaufgabe: Beweisen Sie das Lemma 58.

Beweis.(Theorem 57, siehe auch [1]) Offensichtlich gilt, dass SAT in NP liegt, siehe Lemma 45.

Es muss gezeigt werden, dass $L \leq_p SAT$ für alle $L \in NP$ gilt. Sei also L ein beliebiges Problem aus NP. Dann existiert eine NTM N mit $L = L(N)$ und polynomieller Laufzeit.

Wir nehmen an, dass es sich bei N um eine 1-Band NTM handelt. Analog zum Beweis von Theorem 6 über die Simulation einer k-Band DTM durch eine 1-Band DTM ist diese Annahme bezüglich der Laufzeiten gestattet. Wir benutzen zwei ausgewiesene Endzustände für das Akzeptieren und Verwerfen, das erleichtert die Konstruktion und ist keine Einschränkung. Sei nun $p(n) = C \cdot n^d$ die zu $N = (\Sigma, Q, \delta, q_0, F)$ und L zugehörige Laufzeitfunktion und

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

Tabelle 4.1: Bedeutung der Variablen. Die Anzahl der Variablen liegt in $O(p(n)^2)$, da r und m unabhängig von n durch Konstanten beschränkt sind.

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

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

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

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

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

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

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

Jetzt wollen wir die Konfigurationsübergänge durch eine Formel in konjuktiver Normalform eindeutig beschreiben, dafür benötigen wir Variablen, die in der Tabelle 4.7.4 angegeben sind. Die Anzahl der Variablen ist polynomiell beschränkt.

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

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

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

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

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

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

Zunächst kann die Startkonfiguration leicht durch

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

beschrieben werden.

Für einen Zeitpunkt t können die Randbedingungen $R(t)$ nach Lemma 58 durch

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

beschrieben werden. Nach Lemma 58 können diese Ausdrücke mit insgesamt $h(u,r,m,n) := (u+1)^2 + p^2(n) + (p(n)+1)r^2 + m^2 \in O(p^2(n))$ Literalen beschrieben werden. Insgesamt ergibt sich daraus der Ausdruck

$$R := \bigwedge_{0 \leq t \leq p(n)} (R_{\text{zustand}}(t) \wedge R_{\text{position}}(t) \wedge R_{\text{inhalt}}(t) \wedge R_{\text{übergang}}(t))$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Falls die Maschine N das Wort x verwirft, dann wird als Ausdruck $A(x)$ ein nicht-erfüllbarer Ausdruck zurückgegeben. Wir können dabei auch die Funktion $p(n)$ verwenden, um die Konstruktion nach mehr als $p(n)$ Schritten abzubrechen.

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

Es bleibt zu zeigen:

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

"$\Longrightarrow$":

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

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

Für das Wort $A(x)$ können wir eine Belegung φ der Variablen angeben, die $A(x)$ erfüllt. Wir verwenden exakt die in Tabelle 4.7.4 intendierte Belegung, diese erfüllt den Ausdruck $A(x)$ gemäß Konstruktion.

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

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

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

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

"$\Longleftarrow$":

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

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

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

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

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

- Übergang: $((q_{k(t)}, a_{j_{l(t)}}), (q_{\overline{k}(t)}, a_{\overline{j}_{l(t)}}, v_{l(t)}))$

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

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

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

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

□

4.7.5 NP-Vollständigkeit arithmetischer Probleme

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

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

Theorem 59 *Subset-Sum ist NP-vollständig.*

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

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

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

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

Wie geben ein Beispiel für die Formel

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

an.

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

Genau dann gibt es eine Auswahl T von Zahlen aus der gesamten Zahlenmenge

$$\{y_1, z_1, \ldots, y_n, z_n, s_1, t_1, \ldots, s_m, t_m\}$$

mit $\sum_{x \in T} x = b$, wenn die zugehörige 3-SAT-Formel erfüllbar ist.

Beachten Sie, dass die Konstruktion polynomiell in $(n+m)$ ist. Die Eingabelänge der 3-SAT-Formel hatte gerade diese Länge. Die konstruierten Zahlen sind nun zwar nicht zur Basis 2 binär kodiert aber zur Basis 10 kodiert und sind somit im Vergleich zu ihrer Zahlengröße in logarithmischer Größe (und nicht unär) gespeichert. □

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

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

Theorem 60 *Partition ist NP-vollständig.*

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

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

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

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

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

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

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

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

□

Kapitel 5

Approximation und Randomisierung

Da wir nun das Prinzip der NP-Vollständigkeit kennen gelernt haben, stellt sich unweigerlich die Frage, welche Vorgehensweise ratsam ist, wenn ein Problem schwer zu lösen ist. Unter *lösen* verstehen wir dabei, eine Lösung *deterministisch exakt* zu berechnen. Ist nicht klar, wie die Lösung eines Problems effizient berechnet werden kann, dann bieten sich hier zwei Alternativen an, nämlich die Entwicklung eines *randomisierten Algorithmus* und / oder eines *Approximationsalgorithmus*. Die folgenden beiden Abschnitte basieren auf entsprechenden Kapiteln aus dem Buch [1] von Blum.

5.1 Randomisierte Algorithmen

Im Gegensatz zu einem deterministischen Algorithmus kann ein randomisierter Algorithmus Zufallsentscheidungen verwenden. Der Verlauf einer Berechnung ist somit nicht eindeutig durch die Eingabe bestimmt. Ein naheliegender Ansatz ist, durch Randomisierung im Mittel eine gute Laufzeit beim Lösen eines Problems zu erhalten. Dies motiviert die folgende Definition.

Definition 61 *Ein randomisierter Algorithmus, dessen Laufzeit möglicherweise von seinen Zufallsentscheidungen abhängt, der aber stets eine korrekte Lösung berechnet, heißt* Las-Vegas-Algorithmus.

Weiterhin gibt es die Möglichkeit, dass ein Algorithmus zwar effizient ist, aber nicht immer eine korrekte Lösung berechnet.

Definition 62 *Ein randomisierter Algorithmus, der manchmal – mit geringer Wahrscheinlichkeit – ein falsches Ergebnis produzieren kann, heißt Monte-Carlo-Algorithmus.*

Wir stellen nun zunächst einen Las-Vegas-Algorithmus vor, und im Anschluss einen Monte-Carlo-Algorithmus. Wir beginnen mit dem Las-Vegas-Algorithmus *Randomisiertes QuickSort.* Es handelt sich hierbei um einen Sortieralgorithmus.

Da wir schon optimale deterministische Sortieralgorithmen kennen, stellt sich zunächst die Frage, warum wir einen randomisierten Sortieralgorithmus betrachten. Es handelt sich jedoch beim randomisierten QuickSort um einen

- einfach zu implementierenden Algorithmus,

- der (in einer optimierten Variante) real implementiert deutlich schneller läuft als deterministische Algorithmen, und
- der mit wenig zusätzlichem Speicherplatz auskommt.

Randomisierter Algorithmus **QuickSort(Z)**
Eingabe: n Zahlen $Z = z_1, \ldots, z_n$
Ausgabe: Die Zahlen in Z aufsteigend sortiert.

- Wähle ein $z_i \in Z$
- **return** QuickSort($Z_{<z_i}$), $Z_{=z_i}$, QuickSort($Z_{>z_i}$),

wobei

- $Z_{<z_i} = \{z \in Z \mid z < z_i\}$
- $Z_{=z_i} = \{z \in Z \mid z = z_i\}$
- $Z_{>z_i} = \{z \in Z \mid z > z_i\}$

Die Korrektheit des Algorithmus ist offensichtlich, daher analysieren wir jetzt die *erwartete* Laufzeit des Algorithmus. Zunächst überlegen wir uns, dass die Laufzeit nicht geringer wird, wenn alle Zahlen in Z verschieden sind. Dann können die rekursiv zu sortierenden Teilmengen nur größer und die Laufzeit entsprechend länger sein.

Wir stellen zunächst eine Rekursionsgleichung für die Laufzeit auf für den Fall, dass das Element z_i deterministisch gewählt wird, und leiten dann eine Abschätzung für die erwartete Laufzeit her. Wir betrachten im Folgenden die Anzahl der Vergleiche die durch den Algorithmus ausgeführt werden, da diese die Laufzeit dominieren. Wenn das gewählte Element z_i das k-kleinste Element von Z ist, dann ist (bei entsprechender Implementierung)

$$T(n) \quad = \quad T(k-1) + T(n-k-1) + (n+1) \tag{5.1}$$

die Laufzeit von QuickSort. Wir beobachten zunächst, dass wenn immer $k \approx n/2$ gilt, auch $T(n) \in O(n \log n)$ gilt. Wenn allerdings immer k nicht größer als eine Konstante C ist, dann ist $T(n) \in \Omega(n^2)$. Da eine zufällige Auswahl des Elements z_i einen zufälligen Wert von k impliziert, ist die Frage wie die Laufzeit bei zufällig gewähltem k ist.

Da jedes Element $z_i \in Z$ mit der gleichen Wahrscheinlichkeit $1/n$ gewählt wird, erhalten wir für die *erwartete* Laufzeit T_e des randomisierten QuickSort Algorithmus, wegen Gleichung 5.1,

$$T_e(0) \quad = \quad T_e(1) \quad = \quad 0 \tag{5.2}$$

$$T_e(n) \quad = \quad \frac{1}{n} \sum_{k=1}^{n} (n+1) + T_e(k-1) + T_e(n-k) \tag{5.3}$$

Multiplizieren wir beide Seiten der Gleichung 5.3 mit n, erhalten wir

$$n \cdot T_e(n) \quad = \quad n(n+1) + 2 \sum_{k=0}^{n-1} T_e(k) \quad . \tag{5.4}$$

Da wir einen geschlossenen Term für $T_e(n+1)$ aufstellen möchten, beginnen wir nun mit der Differenz

$$(n+1) \cdot T_e(n+1) - n \cdot T_e(n)$$

Nach Gleichung 5.4 gilt

$$\begin{aligned}(n+1)\cdot T_e(n+1) - n\cdot T_e(n) &= (n+1)(n+2) + 2\sum_{k=0}^{n} T_e(k) - \left[n(n+1) + 2\sum_{k=0}^{n-1} T_e(k)\right] \\ &= 2(n+1) + 2\cdot T_e(n)\end{aligned}$$

Dies ist äquivalent zu

$$T_e(n+1) = 2 + \frac{n+2}{n+1}\cdot T_e(n) \tag{5.5}$$

Per Induktion kann man nun leicht zeigen, dass für $0 \le k \le n-1$

$$T_e(n) = \left(\sum_{j=1}^{k+1} 2\cdot\frac{n+1}{n-k+j}\right) + \frac{n+1}{n-k}\cdot T_e(n-k-1) \tag{5.6}$$

Somit gilt, für $k = n-2$

$$T_e(n) = 2(n+1)\sum_{j=1}^{n-1}\frac{1}{2+j} \tag{5.7}$$

$$= 2(n+1)\sum_{j=3}^{n+1}\frac{1}{j} \tag{5.8}$$

Die Zahl $H_k = \sum_{j=1}^{k}\frac{1}{j}$ ist die k-te harmonische Zahl, und es gilt $H_k \le 1 + \ln(k)$, und somit

$$T_e(n) \le 2(n+1)\cdot\left(H_{k+1} - \frac{3}{2}\right) \tag{5.9}$$

$$\le 2(n+1)\ln(n+1) \tag{5.10}$$

Wir haben somit das folgende Ergebnis bewiesen.

Theorem 63 *Die erwartete Laufzeit des randomisierten QuickSort ist $O(n\log n)$.*

Nun stellen wir den Monte-Carlo-Algorithmus *Primzahltest* vor.

Ein naives Verfahren würde beispielsweise testen, ob für eine gegebene Zahl n eine Division von n mit jeder ganzen Zahl aus $2,\ldots,\sqrt{n}$ einen Rest 0 ergibt. Ist der Rest immer 0, dann ist n prim, andernfalls ist n zusammengesetzt. Für große Zahlen n ist diese Vorgehensweise jedoch nicht effizient.

Unser randomisierter Algorithmus Primzahltest verwendet folgenden Test als Subroutine.

Algorithmus **Test(n,a)**
Eingabe: Eine ungerade natürliche Zahl n, und eine Zahl $a \in \{2,\ldots,n-2\}$.
Ausgabe: *prim* oder *nicht prim*

- Da n ungerade ist, gilt $n-1 = 2^s d$ für *genau* ein $s \in \mathbb{N}$, wobei d ungerade ist.
  - Ist $a^d \not\equiv 1 \mod n$ und
  - $\forall\, r \in \{0,\ldots,s-1\}$ gilt $a^{2^r d} \not\equiv -1 \mod n$

  $\Rightarrow$ dann gebe *nicht prim* zurück, ansonsten *prim*.

Wir verwenden die folgenden zwei Theoreme.

Theorem 64 *Ist n prim, dann ist das Ergebnis von Test(n, a)* prim*, unabhängig von der Eingabe $a \in \{2, \ldots, n-2\}$.*

Das Theorem geht darauf zurück, dass zunächst für eine Primzahl p gilt, dass $a^{p-1} \equiv 1 \mod p$ für jede Zahl $a \in \{2, 3, \ldots, p-2\}$ gilt (Kleiner Fermatscher Satz). Deshalb gilt in jedem Fall $a^{2^s d} \equiv 1 \mod n$ falls n eine Primzahl ist. Darüberhinaus fängt somit die Folge

$$\left(a^{2^s d}, a^{2^{s-1} d}, a^{2^{s-2} d}, \ldots, a^{2d}, a^d\right)$$

modulo n gerechnet mit einer 1 an. Sei nun $x^2 \equiv 1 \mod n$, dann gilt für den *Nachfolger* in der Liste, dass entweder $x \equiv 1 \mod n$ oder $x \equiv -1 \mod n$ ist, falls n eine Primzahl ist! Letzteres ist äquivalent zu $x \equiv n - 1 \mod n$. Also testet man, ob für das letzte Element a^d bereits $a^d \equiv 1 \mod n$ gilt und somit eine Folge von Einsen möglich ist, oder ob zumindest ein $x \equiv n - 1 \mod n$ in der Liste als Nachfolger existiert. Falls das der Fall ist, haben wir ein gutes Zeugnis davon, dass es sich bei n um eine Primzahl handeln könnte. Ansonsten ist klar, dass n keine Primzahl sein kann.

Leider kann es auch falsche Zeigen geben, aber nicht so viele!

Theorem 65 *Sei $n > 9$ eine ungerade zusammengesetzte Zahl. Dann gibt es in $\{1, \ldots, n-1\}$ höchstens 1/4(n-1) Zahlen mit $ggT(a, n) = 1$ und das Ergebnis von Test(n, a) ist* prim.

Die obigen Theoreme motivieren die folgende Vorgehensweise. Es wird k Mal zufällig eine Zahl aus $\{2, \ldots, n-2\}$ ausgewählt. Gilt für eine dieser Zahlen $ggT(a, n) > 1$, dann ist n nicht prim. Andernfalls betrachten wir für jedes a den Test Test(n, a). Ist das Ergebnis eines Tests *nicht prim*, dann ist n nicht prim. Sagt jeder Test *prim*, dann ist das Ergebnis unseres Primzahltests ebenfalls *prim*.

Es gilt, dass wenn n prim ist, dass jeder Test *Test*(n, a) auch das Ergebnis *prim* zurückgibt. Wir wollen nun die Wahrscheinlichkeit abschätzen, dass wenn n nicht prim ist, alle k Tests Test(n, a) das Ergebnis *prim* zurückgeben. Nach obigem Theorem 65 ist für jedes beliebige a die Wahrscheinlichkeit für die Aussage *prim* maximal 1/4. Somit ist die Wahrscheinlichkeit, dass alle Tests das Ergebnis *prim* zurückgeben, maximal $(1/4)^k$. Wählen wir die Konstante k entsprechend groß, dann können wir für den folgenden Algorithmus Primzahltest eine beliebig große Erfolgswahrscheinlichkeit garantieren.

Monte-Carlo-Algorithmus **Primzahltest(n)**
Eingabe: Eine (große) Zahl n
Ausgabe: *prim* oder *nicht prim*, je nachdem ob n eine Primzahl ist oder nicht (mit hoher Wahrscheinlichkeit).

- Ist $n \leq 9$, dann gebe *prim* aus wenn $n \in \{2, 3, 5, 7\}$, ansonsten *nicht prim*
- Ist n gerade, dann **return** *nicht prim*
- **for** $i = 1$ **to** k **do**
  - Wähle zufällig ein $a \in \{2, \ldots, n-2\}$
  - **if** $(ggT(a, n) > 1)$ **return** *nicht prim*
  - **if** (Test(n, a) = *nicht prim*) **return** *nicht prim*
- **return** *prim*

Beispiel $n = 325$ nicht prim: Für $n = 325$ und für $a = 7 \in \{2, \ldots, 323\}$ erhalten wir zunächst $ggT(a, n) = 1$. Der Aufruf von Test(n, a) führt zur Darstellung $n-1 = 324 = 2^2 * 81$ und es gilt: $(a^{81} \mod 325) = 307, (a^{81*2} \mod 325 = 324 = n-1), a^{81*2*2} = a^{324} \mod 325 = 1)$, wobei nach dem zweiten Wert bereits klar war, dass alle nachfolgenden nun 1 werden würden. Der obige Vektor hat also die Form $(307, 324, 1)$ und der Monte Carlo Algorithmus wird fälschlicherweise *prim* liefern.

Wählen wir allerdings $a = 207 \in \{2, \ldots, 323\}$ so erhalten wir auch $ggT(a, n) = 1$ aber für die Folge $(a^{81} \mod 325 = 207 \neq 324 = n-1), (a^{81*2} \mod 325 = 274 \neq 324 = n-1), (a^{81*2*2} = a^{324} \mod 325 = 1)$ und der Vektor zeigt $(207, 274, 1)$. Der Algorithmus liefert *nicht prim*. Hier sieht man auch, dass aus $x^2 \equiv 1 \mod n$ nicht $x \equiv 1 \mod n$ oder $x \equiv -1 \mod n$ folgen muss, wenn n keine Primzahl ist.

Für $a = 25 \in \{2, \ldots, 323\}$ geht bereits der Test $ggT(a, n) = 25$ schief.

Beispiel $n = 41$ prim: Hier ist es egal, welches $a \in \{2, \ldots, 39\}$ wir wählen, der Algorithmus liefert stets *prim*. Zunächst ist $n-1 = 40 = 2^3 \cdot 5$. Für $a = 15$ erhalten wir $(a^5 \mod 41 = 14 \neq 40 = n-1), (a^{5*2} \mod 41 = 32 \neq 40 = n-1), (a^{5*2*2} \mod 41 = 40 = n-1), (a^{5*2*2*2} = a^{n-1} \mod 41 = 1)$ und der Test liefert $(13, 32, 40, 1)$ und verständlicherweise *prim* nach der dritten Auswertung.

Es kann auch schneller gehen, beispielsweise für $a = 37$ ist bereits $(a^5 \mod 41 = 1)$ und das bleibt dann natürlich so! Der Test liefert sofort *prim*.

Wir fassen die obige Analyse in folgendem Theorem zusammen.

Theorem 66 *Algorithmus* Primzahltest *ist immer korrekt wenn die Eingabe n eine Primzahl ist, und er ist mit Wahrscheinlichkeit $\geq 1 - (1/4)^k$ korrekt, falls n zusammengesetzt (keine Primzahl) ist.*

5.2 Approximationen für NP-schwere Probleme

Für uns ist ein Optimierungsproblem NP-schwer, wenn das zugehörige Entscheidungsproblem NP-schwer ist. Wir haben schon das TSP kennengelernt, das in NP liegt. Einen Beweis der NP-Vollständigkeit dieses Problems findet sich beispielsweise in [8].

Außerdem haben wir im Abschnitt 4.7.5 gezeigt, dass Subset-Sum und Partition NP-vollständige Probleme. Daraus werden wir ableiten, dass auch die bereits im ersten Teil der Vorlesung behandelten Problemstellungen *Rucksackproblem* (Knapsack) und *Optimales Bepackens von Behälter* (Bin Packing) NP-vollständig sind.

Wie bereits im ersten Teil dieser Vorlesung erähnt, sprechen wir von einer Approximationslösung, wenn sich die berechnete Lösung eines Algorithmus stets nur um einen Faktor von der optimalen Lösung unterscheidet. Dabei müssen wir die eigentliche Lösung bei der Analyse manchmal gar nicht genau kennen und können trotzdem eine Performanzaussage beweisen. Sei $\text{opt}(I)$ die Optimale Lösung einer Instanz I eines Problems Π.

Wir müssen zwischen Minimierungs- und Maximierungsproblemen unterscheiden.

- Sei $\alpha > 1$: Ein Algorithmus A berechnet eine α-Approximation für ein Minimierungsproblem Π, falls A für jedes $I \in \Pi$ eine zulässige Lösung mit Wert höchstens $\alpha \cdot \text{opt}(I)$ berechnet.
- Sei $\alpha < 1$: Ein Algorithmus A berechnet eine α-Approximation für ein Maximierungsproblem Π, falls A für jedes $I \in \Pi$ eine zulässige Lösung mit Wert mindestens $\alpha \cdot \text{opt}(I)$

berechnet.

Gelegentlich lässt sich lediglich zeigen, dass ein Algorithmus (beispielsweise für eine Minimierungsproblem) stets einen Unterschied von $\alpha \cdot \mathrm{opt}(I) + c$ zum Optimum für eine feste Konstante c garantiert. Asymptotisch wird dann eine Approximatiosngüte α garantiert und eine solche erweiterte Definition ist zulässig.

5.2.1 Approximationsschemata PTAS und FPTAS

Neben oben angegebenen festen Werten der Approximationen, gibt es gelegentlich auch die Möglichkeit, eine beliebig gute Approximationsgüte eines NP-schweren Problems zu erzielen. Die Güte der Approximation beeinflusst dabei die Laufzeit des Algorithmus. Wir sprechen dann auch von einem *Approximationsschemata* und die gewünschte Approximation ist dann durch einen Parameter ϵ beschrieben.

Genauer: Bei Eingabe einer Probleminstanz I und des Güteparameters ϵ wollen wir bei einem Maximierungsproblem eine zulässige Lösung berechnen, die nur um den Faktor $(1 - \epsilon)$ von der optimalen Lösung abweicht. Entsprechend wird bei einem Minimierungsproblem eine Abweichung von um den Faktor $(1 + \epsilon)$ gefordert. Typischerweise (bei NP-schweren Problemen) geht dabei der Parameter ϵ mit in die Laufzeitbetrachtung ein.

Es werden zwei verschiedene Varianten unterschieden:

- FPTAS (fully polynomial time approximation scheme): Ein Approximationsschemata, dessen Laufzeit polynomiell von n und von $\frac{1}{\epsilon}$ abhängt.
- PTAS (polynomial time approximation scheme): Ein Approximationsschemata, dessen Laufzeit für Konstante ϵ polynomiell von n abhängt.

Offensichtlich ist das Vorhandensein eines FPTAS die stärkere Aussage. Laufzeiten in O-Notation von $O((1/\epsilon)^2 n \log n)$, $O(n^2/\epsilon)$ oder auch $O(n^2 + 1/\epsilon^2)$ verdeutlich die polynomielle Abhängigkeit bei einem FPTAS. Dahingegen sind Laufzeiten für PTAS der Größenordungen $O(2^{1/\epsilon} n^2)$ oder $O(n^{1/\epsilon} + n^2)$ exponentiell wachsend in $1/\epsilon$. Dadurch werden Approximationen schnell sehr *teuer*.

5.2.2 Approximationsalgorithmus für TSP

In diesem Abschnitt werden wir einen Approximationsalgorithmus vorstellen, der eine Rundtour entlang der gegebenen Punktmenge berechnet, die nicht viel schlechter ist als eine optimale Tour. Allerdings setzen wir voraus, dass die Kostenfunktion c, welche die Kosten für die Reise von einem zum anderen Punkt festlegt, gewisse Eigenschaften erfüllt. Wir definieren zunächst zwei Spezialfälle des allgemeinen TSP.

Definition 67 *Beim Δ-TSP ist die Aufgabe, eine günstigste Rundreise zu bestimmen, die jeden von n Punkten $P = p_1, \ldots, p_n$ genau ein Mal besucht, wobei gilt:*

- *Symmetrie:* $c(p_i, p_j) = c(p_j, p_i)$
- *Dreiecksungleichung:* $c(p_i, p_j) \leq c(p_i, p_k) + c(p_k, p_j)$

Definition 68 *Eine TSP-Instanz heißt* Euklidisch*, falls es Punkte in einem $\mathbb{R}^d$ gibt, deren Euklidische Abstände denen der Kostenfunktion c entsprechen.*

Wir merken noch an, dass die Abstände zwischen den Punkten als Teil der Eingabe betrachtet werden, und diese nicht implizit durch Angabe der Funktion c – zum Beispiel in Abhängigkeit der Punktkoordinaten – angegeben werden. Weiterhin gilt, dass die Kosten $c(p_i, p_j)$ für eine Reise von p_i nach p_j ganzzahlig und nicht-negativ sind.

Zunächst machen wir zwei einfache Beobachtungen.

1. Jede Euklidische TSP-Instanz ist auch eine Δ-TSP-Instanz.

2. Wenn es bei einer Δ-TSP-Instanz überhaupt eine Rundtour mit endlichen Kosten gibt, dann sind die Reisekosten zwischen je zwei beliebigen Punkten $< \infty$.

Da es sich beim Euklidischen TSP um einen Spezialfall des allgemeinen TSP handelt, könnte man meinen, es wäre einfacher, eine optimale Rundreise zu bestimmen. Dies ist aber nicht der Fall, wie Papadimitriou festgestellt hat [6]:

Theorem 69 *Das Euklidische TSP ist NP-vollständig.*

Wir beschreiben nun den Approximationsalgorithmus für das Δ-TSP.

Algorithmus **Approx-Δ-TSP**

1. Bestimme einen minimalen Spannbaum T der Punktmenge P (die Knoten in T entsprechen folglich den Punkten in P).

2. Von Knoten p_1 aus bestimme eine Besuchsreihenfolge R der Knoten in T gemäß einer Tiefensuche.

3. Modifiziere R zu einer Tour $\overline{R}$ wie folgt:

   - Streiche aus R alle Vorkommen von p_1, abgesehen vom ersten und letzten Vorkommen am Anfang und am Ende von R.
   - Für alle $p \in \{p_2, \ldots, p_n\}$ streiche alle Vorkommen in R bis auf das Erste (d.h. wenn die Tiefensuche den Knoten das erste Mal besucht).

Wir beobachten zunächst, dass R genau die Kosten $2 \cdot |T|$ hat, wenn $|T|$ das Gewicht des minimalen Spannbaums T bezeichnet. Aber R ist noch keine Rundtour, welche jeden Knoten genau einmal besucht. Wir betrachten das Streichen eines Vorkommens eines Punktes p_i in R als das Ersetzen der Bewegung entlang der Kanten (p_j, p_i) und (p_i, p_k) durch die Bewegung entlang der Kante (p_j, p_k), wobei p_j und p_k der Vorgänger bzw. Nachfolger des Vorkommens von p_i in der aktuellen Tour R ist. Gemäß der Dreiecksungleichung haben wir somit die Kosten von R nicht erhöht, und weiterhin ist die resultierende Tour $\overline{R}$ eine Rundtour, welche jeden Punkt aus P genau einmal besucht. Weiterhin gilt, dass wenn wir aus einer beliebigen optimalen Rundreise eine Kante entfernen, ein Spannbaum von P resultiert. Da alle Kantengewichte nicht-negativ sind, folgt hieraus, dass die Kosten einer optimalen Tour mindestens $|T|$ sein müssen. Somit sind die Kosten von $\overline{R}$ maximal doppelt so groß wie die einer optimalen Rundtour.

Theorem 70 *Der Algorithmus Approx-Δ-TSP berechnet für jede Δ-TSP-Instanz in polynomieller Zeit eine 2-Approximation.*

5.2.3 Ein Approximationsschemata des Rucksackproblem

Wir wollen nun eine FPTAS für das Rucksackproblem vorstellen. Wir betrachten das Problem des optimalen Befüllens eines Rucksackes mit Gesamtgewichtskapazität G. Dazu sei eine Menge von n Gegenständen a_i mit Gewicht g_i und Wert w_i gegeben. Wir wollen den Rucksack optimal füllen. Wir gehen davon aus, dass alle Daten natürliche Zahlen sind.

Wir hatten im ersten Teil der Vorlesung bereits einen Algorithmus vorgestellt, der das Problem in Laufzeit $O(n \cdot G)$ Rechenschritten gelöst hat, wobei wir dabei mit dynamischer Programmierung eine Matrix mit $n \cdot G$ Einträgen gefüllt haben, siehe [7]. Da G aber binär kodiert nur $\log G$ Eingabeplatz verbraucht, ist ein solcher Algorithmus bezüglich der Eingabe nicht(!) polynomiell und wir hatten schon damals ohne weiteres Wissen von einem *pseudopolynomiellen* Algorithmus gesprochen. Jetzt wissen wir, was das heißen soll. Ist G konstant, dann haben wir lineare Laufzeit. Im logarithmischen Kostenmodell hat der Algorithmus exponentielle Laufzeit.

Zunächst wollen wir zeigen, dass das zugehörige Entscheidungsproblem NP-vollständig ist. Die Entscheidungsvariante des Problems bedeutet, dass gefragt wird, ob es eine Teilmenge mit zulässigem Gewicht G und Wert mindestens W gibt?

Theorem 71 *Die Entscheidungsvariante des Rucksackproblems ist ein NP-vollständiges Problem.*

Beweis. Wir reduzieren Subset-Sum auf die Entscheidungsvariante. Dazu wird für die Instanz $\{a_1, a_2, \ldots, a_n\}$ mit $a_i \in \mathbb{N}$ und $b \in \mathbb{N}$ des Subset-Sum Problems $g_i = w_i = a_i$ gesetzt und $W = G = b$.

Falls das Subset-Sum Problem eine Lösung hat existiert eine Teilmenge der Zahlen $\{a_1, a_2, \ldots, a_n\}$ mit Gewicht höchstens $G = b$ und Wert mindestens $W = b$.

Falls umgekehrt eine Teilmenge der Zahlen $\{a_1, a_2, \ldots, a_n\}$ mit Gewicht höchstens $G = b$ und Wert mindestens $W = b$ existiert, muss dieser Wert schon exakt erreicht werden, da Gewichte und Wert identisch sind. □

Wir geben jetzt eine weiteren pseudopolynomiellen Algorithmus für das Entscheidungsproblem des Rucksackproblem an.

Theorem 72 *Das Rucksackproblem kann pseudopolynomiell in $O(n^2 \cdot W)$ (uniformen) Rechenschritten für $W = \max_{i=1,\ldots,n} w_i$ gelöst werden.*

Beweis. Wir schlagen folgenden dynamische Programmierung vor. Der optimale Wert kann nur zwischen 0 und $n \cdot W$ liegen. Für $i \in \{0, \ldots, n\}$ und $w \in \{0, \ldots, n \cdot W\}$ sei nun $A(i, w)$ das kleinstmögliche Gesamtgewicht, mit dem der Wert w exakt erreicht werden kann, wobei man nur Objekte aus $\{a_1, \ldots, a_i\}$ verwenden darf.

Wenn der Wert w gar nicht durch $\{a_1, \ldots, a_i\}$ erreicht werden kann, dann setzen wir $A(i, w) = \infty$. Außerdem sei $A(0, w) = \infty$ für $w > 0$ und $A(i, w) = \infty$ für $w < 0$ und $i \in \{0, \ldots, n\}$ Es sei $A(i, 0) = 0$ für $i = \{1, \ldots, n\}$.

Für $i \in \{1, \ldots, n\}$ und $w \in \{1, \ldots, n \cdot W\}$ gilt dann die folgende Rekursionsgleichung:

$$A(i + 1, w) = \min(A(i, w), A(i, w - w_{i+1}) + g_{i+1})\,.$$

Den Wert w erreichen wir mit kleinstmöglichem Gesamtgewicht entweder durch eine Lösung, in der der Gegenstand a_{i+1} eine Rolle spielt, oder aber der Gegenstand wird nicht verwendet.

Die Lösung mit dem kleinerem Gesamtgewicht wird ausgewählt. Entweder kann man also die Lösung für $A(i, w)$ bereits verwenden oder aber $A(i, w - w_{i+1}) + g_{i+1}$ muss berechnet werden. Dabei wird der Wert um w_{i+1} reduziert. Dieser wird dann durch den Gegenstand a_{i+1} mit w_{i+1} genau auf w vergrößert.

Wie *üblich* können wir von links nach rechts und von oben nach unten eine Matrix mit den entsprechenden Werten füllen. Jeder Eintrag kostet nur konstant viele Operationen. Die Laufzeit entspricht einer Tabelle der Größe $O(n^2 \cdot W)$ und der optimal zu erreichenden Wert ist

$$\max\{w | A(n, w) \leq G\}.$$

Dieser kann aus der Tabelle abgelesen werden. Man kann außerdem zeigen, dass die optimale Rucksackbelegung ebenfalls leicht mit abgespeichert werden kann. Wir merken uns eine Tabelle von Zeigern, die die optimale Lösung wiedergibt. □

Wir wollen einen FPTAS entwerfen. Der Algorithmus hängt linear von den Werten w_i ab, diese Werte sind natürliche Zahlen und der Algorithmus kann nur mit natürlichen Zahlen arbeiten. Beachten Sie, dass im Prinzip nur sehr, sehr große Werte w_i für uns problematisch sind. Sofern die w_is in der Größenordnung von n liegen, haben wir gar kein Problem mit der Laufzeit.

Wollen wir die Laufzeit verringern, müssen wir die Werte nach unten skalieren und dabei auf jeden Fall runden. Die Idee ist, alle Werte w_i um dem gleichen Faktor α (z.B. $\alpha = 0.005$) kleiner zu machen und nach unten abzurunden.

Die Lösung ist dann ggf. nicht mehr optimal aber die Laufzeit hat sich auch um einen Faktor von mindestens α verringert. Wir wollen eine $(1 - \epsilon)$ erreichen, deshalb muss der Skalierungsfaktor α geschickt in Abhängigkeit von n, W und ϵ gewählt werden.

Theorem 73 *Für das Rucksackproblems existiert ein FPTAS. In (uniformer) Laufzeit $O(1/\epsilon \cdot n^3)$ kann eine $(1 - \epsilon)$-Approximation der optimalen Wertes berechnet werden. Die Laufzeit ist auch im logarithmischen Kostenmodell polynomiell in $1/\epsilon$ und n beschränkt.*

Beweis. Für den FPTAS skalieren wir die Werte mit dem Faktor $\alpha = \frac{n}{\epsilon W}$ mit $W = \max_{i=1,\ldots,n} w_i$ und runden dann ab. Also $w'_i = \lfloor \alpha w_i \rfloor$. Dann berechnen wir eine optimale Indexmenge $L \subseteq \{1, \ldots, n\}$ mittels dynamischer Programmierung für die Werte $w'_1, w'_2, \ldots, w'_n$. Es gilt $W' = \lfloor \alpha W \rfloor = \lfloor \frac{n}{\epsilon} \rfloor$ und der Algorithmus hat eine Laufzeit von $O(n^2 \cdot n/\epsilon) = O(1/\epsilon \cdot n^3)$ im uniformen Kostenmodell. Die im Algorithmus verwendeten Additionen und Vergleiche erhöhen die Laufzeit allerdings nur um einen Faktor, der in der logarithmischen Größe der Eingabe liegt. Also bleibt die Laufzeit polynomiell in $1/\epsilon$ und n.

Wir müssen noch den Approximationsfaktor von $(1-\epsilon)$ beweisen. Sei dazu $L^* \subseteq \{1, \ldots, n\}$ die optimale Indexmenge des Ausgangsproblems. Wir wollen die Lösung von L^* und L vergleichen. Für eine Teilmenge $S \subseteq \{1, \ldots, n\}$ sei $w(S) = \sum_{i \in S} w_i$ der Wert dieser Teilmenge. Wir müssen

$$w(L) \geq (1 - \epsilon) w(L^*)$$

beweisen. Für den Beweis schauen wir uns eine *Zwischenlösung* an. Wir skalieren die Werte w'_i wieder hoch durch den Faktor $1/\alpha$, den Rundungsfehler lassen wir allerdings nun weg. Das heißt konkret $w''_i := \frac{w'_i}{\alpha} = \frac{\lfloor \alpha w_i \rfloor}{\alpha}$.

Die Belegung L ist optimal für die Werte w'_i und somit auch für die Werte w''_i, da die Werte w'_i alle um den gleichen Faktor α angehoben wurden. Im Vergleich zur optimalen Lösung

L hat der Algorithmus aber durch die Rundungsfehler eine andere Auswahl getroffen. Die Rundungsfehler können wir abschätzen:

$$w_i - w_i'' = w_i - \frac{\lfloor \alpha w_i \rfloor}{\alpha} \leq w_i - \frac{\alpha w_i - 1}{\alpha} = \frac{1}{\alpha}\,.$$

Sei nun $w''(S) = \sum_{i \in S} w_i''$, dann lässt sich der Rundungsfehler einer optimalen Lösung L^* abschätzen durch

$$w(L^*) - w''(L^*) \leq \sum_{i \in L} \frac{1}{\alpha} \leq \frac{n}{\alpha} = \epsilon\, W \leq \epsilon\, w(L^*)\,.$$

Die letzte Ungleichung folgt daraus, dass der Gegenstand mit dem maximalen Wert auf jeden Fall auch in den Rucksack passt, sonst hätten wir diesen auch sofort weglassen können.

Jetzt haben wir

$$w(L^*) - w''(L^*) \leq \epsilon\, w(L^*) \Leftrightarrow w''(L^*) \geq (1-\epsilon)\, w(L^*)$$

und wir wissen bereits, dass $w''(L) \geq w''(L^*)$ ist. Da die Ausgangswerte w_i größer sind als die w_i'' gilt auch $w(L) \geq w''(L)$, es ist insgesamt:

$$w(L) \geq w''(L) \geq w''(L^*) \geq (1-\epsilon) w(L^*)$$

und die berechnete Lösung liefert mit den Ausgangswerten die gewünschte Approximationsgüte. □

Die hier verwendete Skalierung und Abschätzung ist ein klassischer Trick zum Design und zur Analyse von Approximationsgüten.

5.2.4 Approximation von Bin Packing

Das Problem des optimalen Bepackens von mehreren Behältern gleicher Gewichtskapazität G mit verschiedenen Paketen (Bin Packing) hatten wir bereits im letzten Semester betrachtet.

Die Pakete p haben keinen Preis sondern lediglich verschiedene Gewichte g. Insgesamt haben wir zunächst n Behälter b_j für $j = 1, \ldots, n$ mit Kapazität G und n Gegenstände p_i mit Gewichten g_i für $i = 1, \ldots, n$. Wir können $g_i \leq G$ annehmen, sonst brauchen wir einen Gegenstand gar nicht zu betrachten.

Ziel der Aufgabenstellung ist es, die Anzahl der benötigten Behälter zu minimieren. Im Prinzip bestimmen wir eine Zuordnung $Z : \{1, \ldots, n\} \mapsto \{1, \ldots, k\}$ die jedem p_i einen Behälter b_j über $Z(i) = j$ zuweist. Wir suchen das minimale k, für das es eine gültige Zuordnung Z gibt. Gültig heißt, dass die Summe aller g_i mit $Z(i) = j$ nicht größer als G ist.

Aufgabenstellung: Minimiere k, so dass eine Zuordnung $Z : \{1, \ldots, n\} \mapsto \{1, \ldots, k\}$ existiert mit

$$\sum_{Z(i)=j} g_i \leq G \text{ für alle } i \in \{1, \ldots, n\}\,. \tag{5.11}$$

Der Algorithmus First-Fit liefert eine 2-Approximation, das nehmen wir als bekanntes Ergebnis aus der letzten Vorlesung zur Kenntnis, siehe [7].

First-Fit: Unter allen Behältern wähle den ersten aus, in den a_i noch hineinpasst. Genauer: Finde kleinstes j, so dass $g_i + \sum_{1 \leq l \leq (i-1), Z(l)=j} g_l \leq G$ gilt.

Wir zeigen nun lediglich, dass es sich um ein NP-vollständiges Problem handelt und die Approximation somit gerechtfertigt ist.

Theorem 74 *Die Entscheidungsvariante von Bin Packing ist NP-vollständig.*

Beweis. Die Entscheidungsvariante lautet, ob für eine vorgegebene Anzahl von Behälter K eine zulässige Zuordnung existiert.

Wir reduzieren Partition auf dieses Problem. Dafür setzen wir $g_i = a_i$ für $i = 1, \ldots, n$, setzen $K = 2$ und $G := \frac{1}{2} \sum_{i=1}^{n} g_i$.

Genau dann, wenn es eine Teilmenge $T \subseteq \{1, \ldots, n\}$ mit $\sum_{i \in T} g_i = G$ gibt, kann das Bin Packing Problem mit 2 Behältern gelöst werden. □

5.3 Fixed Parameter Tractability

Der bisherige Ausweg bei NP-schweren Problemem war durch eine Approximation oder eine wahrscheinlich richtige Antwort geprägt. Abschließend wollen wir noch eine weitere Möglichkeit zur Behandlung der Problematik erläutern.

Dazu betrachten wie wiederum das Vertex-Cover Problem aus Abschnitt 4.5.1. Wir hatten in Abschnitt 4.7.3 festgestellt, dass das Problem NP-vollständig ist.

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

Die Frage nach einer minimalen Knotenüberdeckung ist ebenfalls NP-vollständig, da wir beispielsweise durch binäre Suche ein Optimum finden können. Aus praktischer Sicht könnten wir das Minimierungsproblem wie folgt betrachten. Für eine Straßennetzwerk möchten wir die minimale Anzahl an Kameras ermitteln, so dass wir von den Verkehrsknotenpunkten jede Straße *einsehen* können. Wir hatten bereits in einer Übungsaufgabe festgestellt, dass es einen einfachen polynomiellen Algorithmus mit einer 2-Approximation gibt, wir möchten aber möglicherweise doch das Minimum ermitteln, auch wenn dadurch die Laufzeit steigt.

Wir halten somit den Parameter k fest und wollen einen Algorithmus entwerfen, der das Optimum (Minimum) liefert, falls der minimale Vertex-Cover kleiner-gleich k ist und *false* sonst. Der Algorithmus soll also das folgende liefern!

Vertex-Cover(k): Geben ist ein Graph $G = (V, E)$ und eine Zahl $k \in \mathbb{N}$. Sei OPT der minimale Vertex-Cover von G.
Falls $|\text{OPT}| \leq k$ berechne das Ergebnis OPT.
Falls $|\text{OPT}| > k$, gebe *false* als Ergebnis zurück.

Nun gehört in der obigen Beschreibung der Parameter k sowie ein Parameter n für die Größe des Graphen zur Aufgabenstellung. Suchen wir nach Algorithmen, die polynomiell in n sind, so könnten wir erlauben, dass die Laufzeit zudem durch eine Funktion in k (unabhängig von n) beschrieben wird, die nicht polynomiell sein muss. Für einen *festen Parameter* k hätten wir somit eine polynomielle Laufzeit. Das fassen wir in folgender Definition zusammen:

Definition 75 *Eine Problemstellung mit zwei verschiedenen Eingabegrößen k und n bezeichnen wir als* fixed-parameter-tractable*, falls ein Algorithmus zur Lösung des Problems mit Laufzeit $O(f(k) \cdot n^l)$ existiert, wobei $l \in \mathbb{N}$ eine feste Konstante unabhängig von k und f eine Funktion unabhängig von n ist.*

Beachte, dass f für das Vertex-Cover Problem vermutlich nicht polynomiell in k sein kann, da wir dann $P = NP$ zeigen könnten.

5.3.1 Einfache Betrachtungen

Für das obige Vertex-Cover Problem unter Vorgabe von k ist es denkbar, für alle k-elementigen Teilmengen der n Knoten, die Vertex-Cover Eigenschaft zu überprüfen. Diese einfache Idee ergibt eine Laufzeit von

$$\binom{n}{k} = \frac{n!}{(n-k)!k!} \in O(n^k)$$

und erfüllt zudem die obige fixed-parameter-tractable Eigenschaft nicht, da die polynomielle Laufzeit in n mit wachsendem k ansteigt und nicht durch eine feste konstante $l \in N$ unabhängig von k begrenzt wird. Für einen echten FPT-Algorithmus benötigen wir Laufzeiten der Art $2^k \cdot n^2, k! \cdot n^4, \ldots$

Nun stellen wir einen einfachen FPT-Algorithmus vor, der mittels eines Suchbaumes eine Laufzeit von $2^k \cdot n$ zur Lösung des obigen Vertex-Cover Problems garantiert. Die einfache Idee beruht darauf, dass wir für (V, E) eine beliebige Kante $e = (v, w)$ wählen, und für beide Endpunkte rekursiv testen, ob eine Lösung für $(G - v) := (V \setminus \{v\}, E \setminus \{v\})$ oder für $(G - w) := (V \setminus \{v\}, E \setminus \{w\})$ für $k - 1$ existiert, wobei $E \setminus \{v\}$ (respektive $E \setminus \{w\}$) nur die Kanten aus E beinhaltet die nicht v (respektive nicht w) als Endpunkt besitzen.

Wir betrachten zwei durch eine Kante miteinander verbundene Knoten v und w und wählen die Knoten alternativ als Kandidaten aus. Gemäß Aufgabenstellung muss einer der beiden Knoten v oder w zum Vertex-Cover gehören sonst wir die Kante nicht abgedeckt. Nehmen wir ohne Einschränkung an, es wäre der Knoten v. Zu diesem Knoten benachbarte Knoten und auch der Knoten w können durchaus noch zum Vertex-Cover beitragen, aber nur dann, wenn Sie einen Knoten abdecken müssen, der nicht bereits mit v verbunden ist. Deshalb können wir in diesem Fall alle Kanten zu v entfernen, aber natürlich nicht die beteiligten Knoten. Im verbleibenden Baum $G' = (G - v)$ suchen wir nach einem Vertex-Cover mit $k' = k - 1$ Kanten. In diesem Sinne ist der Ansatz korrekt, da beide Alternativen v oder w betrachtet werden.

SimpleFPT$((V, E), k)$

```
1: if k = 0 and E ≠ ∅ then
2:    Return false;
3: else if E = ∅ then
4:    Return 0;
5: else
6:    Choose e = (v, w) ∈ E;
7:    K_v := SimpleFPT((G − v), k − 1);
8:    K_w := SimpleFPT((G − w), k − 1);
9:    Return min(K_v, K_w) + 1;
10: end if
```

Insgesamt ergibt sich durch das rekursive Verfahren ein binärer Suchbaum wie in Abbildung 5.1 dargestellt. Der Rekursionsaufruf wird mit dem Ergebnis *false* gestoppt, falls $k' = 0$ wird aber der Graph G' noch nicht leer ist. Falls der Graph G' für $k' \geq 0$ leer ist, wird der Wert 0 zurückgegeben, weitere Knoten werden nicht gebraucht. Die Ergebnisse der Teilbäume werden verglichen und der kleinste Wert zurückgegeben. Für einen solchen erfolgreichen Pfad könnten wir uns sukzessive auch die verwendeten Knoten des Covers ausgeben, durch die Rekursionstiefe ist auch das zugehörige k' bekannt. Somit wird die Aufgabenstellung vollständig gelöst und die Methode SimpleFPT$((V, E), k)$ liefert das gewünschte Ergebnis (abgesehen von den Knoten des Covers, wie gerade bemerkt).

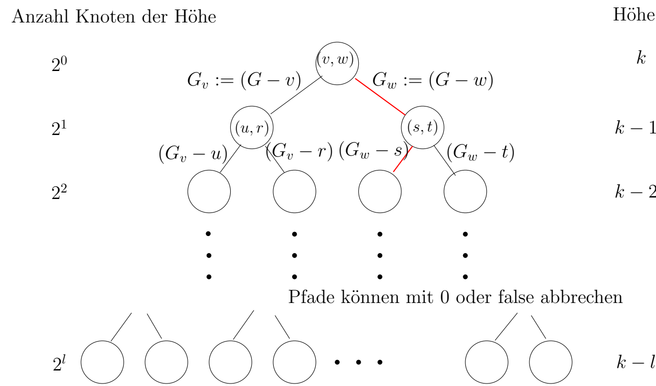

Abbildung 5.1: Solange der Restgraph G' für $k' > 0$ noch Kanten enthält, wird eine Kante $e = (v, w)$ aus E' ausgewählt und es werden die beiden Alternativen $(G' - v)$ und $(G' - w)$ mit $k'' = k' - 1$ aufgerufen. Der Pfad zu einem *erfolgreichen* Knoten minimaler Höhe $h \geq 0$, ergibt einen optimalen Vertex-Cover.

Das Entfernen von Kanten bezüglich eines gegeben Knotens kann in $O(m)$ Zeit für $|E| = m$ durchgeführt werden. Man kann aber auch zu Beginn bereits alle Mehrfachkanten vernachlässigen und von *einfachen* Graphen ausgehen. Gegebenenfalls werden alle Mehrfachkanten vorab mit Laufzeit $O(m)$ für $|E| = m$ entfernt. Danach werden in jedem einzelnen Schritt maximal $O(n)$ Kanten entfernt, der Graph hat Knotengrad $\leq n$.

Deshalb können wir die Laufzeit des Verfahren durch eine Rekursionsformel $T(n, k)$ für einen einfachen Graphen $G = (V, E)$ mit $|V| = n$ Knoten und einen Parameter k so beschreiben:

$$T(n, k) := 2 \cdot T(n - 1, k - 1) + O(n) \in O(2^k \cdot n) . \tag{5.12}$$

Die Laufzeit ist offensichtlich, da sich die Anzahl der Teilprobleme jeweils verdoppelt und das insgesamt nur k mal.

Theorem 76 *Der Algorithmus SimpleFPT berechnet für einen Graphen $G = (V, E)$ mit $|E| = m$ und $|V| = n$ und ein $k \in \mathbb{N}$ in Zeit $O(m + 2^k \cdot n)$ einen optimalen Vertex-Cover der Größe $\leq k$ falls dieser existiert und liefert* false*, falls es keinen Vertex-Cover dieser Größe gibt.*
Die Problemstellung ist fixed-parameter-tractable.

5.3.2 Verbesserungen durch Kernelisation

Eine naheliegende Idee ist es, den Teil eines Graphen zu betrachten, der den *Kern* des Problems ausmacht, das heißt den schweren Teil des Problems zu isolieren und für diesen dann die Laufzeit $f(k)$ zu manifestieren. Dazu wollen wir den Parameter k nutzen.

Beobachtung: Wenn ein Knoten v mehr als k Nachbarn hat, dann muss dieser Knoten unweigerlich zum Vertex-Cover der Größe $\leq k$ dazugehören, sonst müssen alle $k+1$ ausgehenden Kanten abgedeckt werden, ein Widerspruch zur Lösung mit $\leq k$.

Das gilt völlig unabhängig von anderen Knoten und somit können wir *alle* Knoten mit Grad $\geq k+1$ bereits zum Vertex-Cover hinzufügen und auch die entsprechenden Kanten entfernen. Dann bleibt ein *Kern* des Problems übrig. Sei C die Menge aller Knoten mit Grad $\geq k + 1$, betrachte:

$$(G - C) := (V \setminus \{C\}, E \setminus \{C\})$$

wobei die Kantenmenge $E\setminus\{C\}$ aus E durch das Entfernen aller Kanten, die mit Knoten aus C verdunden sind, entsteht. Es muss gezeigt werden, dass dieser Kern $G-C$ *klein* ist. Man sollte beachten, dass beim Entfernen der Kanten isolierte Knoten entstehen, diese müssen aber für einen Vertex-Cover des Restgraphen gar nicht berücksichtigt werden, da ja die Kanten durch Knoten abgedeckt werden.

Lemma 77 *Sei C die Menge aller Knoten mit Grad $\geq k+1$ und sei $(G-C) = (V', E')$ der zugehörige Kern. Falls $|OPT| \leq k$ für den minimalen Vertex-Cover OPT von G ist, dann gilt $|E'| \leq k^2$.*

Beweis. Jeder Knoten von G' kann maximal k Kanten abdecken. Der Vertex-Cover von G' muss $\leq k$ sein.

□

Nun schlagen wir folgenden Algorithmus vor. Zunächst betrachten wir alle Knoten vom Grad $\geq k+1$, fügen diese Knotenmenge C zum Vertex-Cover hinzu und betrachten $(G-C)$. Dabei entfernen wir zunächst wieder die Mehrfachkanten, und entfernen dann die Kanten für Knoten mit Grad $\geq k+1$. Der Aufwand dafür wird insgesamt mit $O(m)$ abgeschätzt. Danach wenden wir den einfachen SimpleFPT-Algorithmus auf dem Restgraphen für $k - |C|$ mit maximal $O(k^2)$ Kanten an und erhalten die Laufzweit $O(k^2 + 2^k \cdot k^2) \in O(2^k \cdot k^2)$. Die Laufzeit beträgt insgesamt $O(m + 2^k \cdot k^2)$ und die Funktion $g(k) := 2^k \cdot k^2$ beeinflusst die Laufzeit *nur* additiv!

Theorem 78 *Mit Hilfe einer Kernelisation erhalten wir folgendes Ergebnis für einen Graphen $G = (V, E)$ mit $|E| = m$ und $|V| = n$ und ein $k \in \mathbb{N}$. In Zeit $O(m + 2^k \cdot k^2)$ wird ein optimaler Vertex-Cover der Größe $\leq k$, falls dieser existiert oder es wird berichtet, dass es keinen Vertex-Cover dieser Größe gibt.*

Nach den obigen Betrachtungen könnte man auf die Idee kommen, dass ein Greedy-Algorithmus, der sukzessive den Knoten mit maximalen Grad auswählt, eine gute Approximation liefern müsste. Das ist aber nicht so, da wir beispielsweise Knoten vom Grad k für den Vertex-Cover der Größe k besser nicht auswählen. In der Vorlesung wurde ein kleines Beispiel gezeigt, dass sich leicht zu einem Faktor $\Omega(\log n)$ für n Knoten erweitern lässt.

Literaturverzeichnis

[1] Norbert Blum. Algorithmen und Datenstrukturen: Eine anwendungsorientierte Einführung. Oldenbourg Verlag, 2004.

[2] Berthold Vöcking Berechenbarkeit und Komplexität Vorlesungsskript, RWTH Aachen 2009.

[3] Johannes Blömer Einführung in Berechenbarkeit, Komplexität und formale Sprachen Vorlesungsskript, Universität Paderborn 2012.

[4] Uwe Schöning Theoretische Informatik – kurzgefasst Teubner, 2004.

[5] C. H. Papadimitirou Computational Complexity Addison-Wesley 1994.

[6] C. H. Papadimitirou The Euclidean Traveling Salesman Problem is NP-Complete Theoretical Computer Science, 1977. URL: http://dx.doi.org/10.1016/0304-3975(77)90012-3

[7] Elmar Langetepe und Heiko Röglin Algorithmen und Berechnungskomplexität I Vorlesungsskript, Universität Bonn 2013.

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