Algorithmen und Berechnungskomplexität II

Prof. Dr. Heiko Röglin
Institut für Informatik
Universität Bonn

23. Mai 2022

Inhaltsverzeichnis

Bitte senden Sie Hinweise auf Fehler im Skript und Verbesserungsvorschläge an die E-Mail-Adresse `roeglin@cs.uni-bonn.de`.

Kapitel 1

Einleitung

Wir haben im vergangenen Semester effiziente Algorithmen zur Lösung einer Vielzahl von Problemen entworfen. In dieser Vorlesung werden wir die Grenzen der Algorithmik kennenlernen und uns damit beschäftigen, für welche Probleme es keine effizienten Algorithmen gibt und welche überhaupt nicht algorithmisch gelöst werden können.

Wir starten mit einer Einführung in die *Berechenbarkeitstheorie*. Dieses Teilgebiet der theoretischen Informatik ist der Frage gewidmet, welche Probleme nicht durch Algorithmen in endlicher Zeit gelöst werden können und zwar unabhängig davon, wie leistungsfähig unsere Rechner in der Zukunft auch sein mögen. Die Existenz solcher *nicht berechenbaren* Probleme ist uns bereits aus der Vorlesung „Logik und diskrete Strukturen“ bekannt. Wir haben dort jedoch kein konkretes nicht berechenbares Problem angegeben, sondern nur ausgenutzt, dass die Menge der Funktionen überabzählbar ist, während es nur abzählbar unendlich viele Programme gibt. Dieses Ergebnis lässt die Möglichkeit offen, dass alle in der Praxis relevanten Probleme algorithmisch gelöst werden können und nicht berechenbare Probleme im Informatik-Alltag keine Rolle spielen.

Wir werden jedoch am Beispiel des *Halteproblems* demonstrieren, dass es auch ausgesprochen wichtige Probleme gibt, die nicht berechenbar sind. Bei diesem Problem soll für ein gegebenes Programm entschieden werden, ob es nach endlich vielen Schritten terminiert. Zur Verifikation der Korrektheit von Programmen wäre es hilfreich, einen Compiler zu haben, der bei nicht terminierenden Programmen eine Warnung ausgibt. Wir werden beweisen, dass es einen solchen Compiler nicht geben kann, da das Halteproblem nicht berechenbar ist. Anschließend werden wir Techniken kennenlernen, mit deren Hilfe man auch für viele weitere Probleme nachweisen kann, dass sie nicht berechenbar sind.

Im zweiten Teil der Vorlesung werden wir uns mit *Komplexitätstheorie* beschäftigen. In diesem Teilgebiet der theoretischen Informatik geht es um die Frage, welche Ressourcen notwendig sind, um bestimmte Probleme zu lösen. Wir konzentrieren uns insbesondere auf die Ressource Rechenzeit, denn für praktische Anwendungen ist die Laufzeit eines Algorithmus oft von entscheidender Bedeutung. Wer möchte schon einen Tag oder gar mehrere Jahre auf die Ausgabe seines Navigationsgerätes warten? Leider gibt es eine

ganze Reihe von Problemen, die zwar berechenbar sind, für die es aber vermutlich keine effizienten Algorithmen gibt. Ein solches Problem, das in vielen logistischen Anwendungen eine Rolle spielt, ist das *Problem des Handlungsreisenden (engl. Traveling Salesman Problem (TSP))*, bei dem eine Landkarte mit mehreren Städten gegeben ist und die kürzeste Rundreise durch diese Städte gesucht wird.

Das Problem des Handlungsreisenden gehört wie viele andere natürliche Probleme zu der Klasse der *NP-schweren* Probleme. Man vermutet, dass es für diese Probleme keine effizienten Algorithmen gibt. Diese sogenannte $P \neq NP$*-Vermutung* ist bis heute unbewiesen und eines der größten ungelösten Probleme der Mathematik und der theoretischen Informatik. Wir werden besprechen, was diese Vermutung genau besagt, und zahlreiche NP-schwere Probleme kennenlernen.

Zwar werden in dieser Vorlesung hauptsächlich negative Ergebnisse gezeigt, diese haben aber wichtige Auswirkungen auf die Praxis. Hat man beispielsweise bewiesen, dass ein Problem nicht effizient gelöst werden kann, so ist klar, dass die Suche nach einem effizienten Algorithmus eingestellt werden kann und dass man stattdessen über Alternativen (Abwandlung des Problems etc.) nachdenken sollte. Außerdem beruhen Kryptosysteme auf der Annahme, dass gewisse Probleme nicht effizient gelöst werden können. Wäre es beispielsweise möglich, große Zahlen effizient zu faktorisieren, so wäre RSA kein sicheres Kryptosystem. In diesem Sinne können also auch negative Ergebnisse gute Nachrichten sein.

Im letzten Teil der Vorlesung werden wir uns damit beschäftigen, wie man in Anwendungen mit NP-schweren Problemen umgehen kann. Der erste Ansatz, den wir dazu kennenlernen werden, sind *Approximationsalgorithmen*. Trifft die P $\neq$ NP-Vermutung zu, so können NP-schwere Probleme nicht effizient gelöst werden. Für solche Probleme ist es deshalb oft sinnvoll, die Anforderungen zu reduzieren und statt nach einer optimalen nur noch nach einer möglichst guten Lösung zu suchen. Ein *r-Approximationsalgorithmus* ist ein effizienter Algorithmus, der für jede Eingabe eine Lösung berechnet, die höchstens um den Faktor r schlechter ist als die optimale Lösung. Ein 2-Approximationsalgorithmus für das Problem des Handlungsreisenden ist also beispielsweise ein Algorithmus, der stets eine Rundreise berechnet, die höchstens doppelt so lang ist wie die kürzeste Rundreise. Wir werden für einige NP-schwere Probleme Approximationsalgorithmen entwerfen und die Grenzen dieses Ansatzes diskutieren.

Anschließend lernen wir die Technik der *linearen Programmierung* kennen. In einem linearen Programm geht es darum, Werte für eine Menge von reellwertigen Variablen zu wählen, die eine lineare Zielfunktion optimieren. Dabei gibt es eine Menge von linearen Nebenbedingungen, die eingehalten werden müssen. Wir werden sehen, dass es in polynomieller Zeit möglich ist, eine optimale reellwertige Belegung der Variablen zu finden. Da viele Probleme in Form eines linearen Programms dargestellt werden können, ist dies ein interessantes Ergebnis, das für viele Probleme direkt polynomielle Algorithmen liefert. Wenn man die erlaubten Werte der Variablen auf ganze statt reelle Zahlen einschränkt, so spricht man von einem *ganzzahligen linearen Programm*. Die optimale Wahl ganzzahliger Werte ist jedoch ein NP-schweres Problem und insbesondere kann man viele bekannte NP-schwere Probleme einfach als ganzzahlige lineare Programme formulieren. Dies ist deshalb interessant, da es zur Lösung ganzzahliger linearer Programme hoch optimierte Softwarepakete gibt, die für einige (aber nicht alle)

Eingaben trotz der NP-Schwere effizient optimale Lösungen finden. Ein heuristischer Ansatz zur Lösung eines NP-schweren Problems in Anwendungen besteht also darin, das Problem als ganzzahliges lineares Programm zu formulieren und dieses dann mithilfe eines der oben genannten Softwarepakete zu lösen. Das ist nicht immer effizient, kann aber je nachdem, welche Eingabe gelöst werden soll, funktionieren.

Zum Abschluss betrachten wir noch kurz *Exponentialzeitalgorithmen*. Ein Algorithmus, der jede Eingabe eines NP-schweren Problems optimal löst, benötigt im Worst Case vermutlich exponentielle Rechenzeit. Diese erreicht man bei den meisten Problemen bereits durch einen einfachen Brute-Force-Ansatz, der alle möglichen Lösungen ausprobiert. Wir werden allerdings sehen, dass es auch bei den Exponentialzeitalgorithmen noch Abstufungen gibt und man die Basis der exponentiellen Laufzeit durch geeignete Methoden oft verbessern kann. Dies führt dann zu Algorithmen, die zumindest für kleine Eingaben noch praktikabel sind.

Es gibt zahlreiche Bücher und Skripte, in denen die Inhalte dieser Vorlesung teilweise nachgelesen werden können (z. B. [1, 2, 3]). Einen hervorragenden Einstieg in das Thema Komplexität liefert das vierte Buch der „Algorithms Illuminated“-Reihe von Tim Roughgarden [4] (sofern man sich nicht daran stört, dass es nur eine englische Ausgabe gibt). Diese Vorlesung orientiert sich in Teilen insbesondere an dem Buch von Ingo Wegener [7] und dem Skript von Berthold Vöcking [6].

Kapitel 2

Grundlagen

Bevor wir uns der Berechenbarkeits- und Komplexitätstheorie widmen, führen wir in diesem Kapitel zunächst einige Grundlagen ein, die dafür notwendig sind. Wir definieren formal, was wir unter einem Problem verstehen, und diskutieren dann verschiedene Rechnermodelle.

2.1 Probleme und Funktionen

Ein *Algorithmus* ist eine Handlungsvorschrift, mit der Eingaben in Ausgaben transformiert werden können. Diese Handlungsvorschrift muss so präzise formuliert sein, dass sie von einem Computer ausgeführt werden kann. Unter einem *Problem* verstehen wir informell den gewünschten Zusammenhang zwischen der Eingabe und der Ausgabe. Beim Sortierproblem besteht die Eingabe beispielsweise aus einer Menge von Zahlen und die gewünschte Ausgabe ist eine aufsteigend sortierte Permutation dieser Zahlen.

Um den Begriff *Problem* zu formalisieren, gehen wir davon aus, dass eine beliebige endliche Menge Σ gegeben ist, die wir *Alphabet* nennen. Die Eingaben und Ausgaben sind als Zeichenketten über diesem Alphabet codiert. Denkt man an reale Rechner, so ist die Wahl $\Sigma = \{0, 1\}$ naheliegend. Für theoretische Betrachtungen ist es manchmal hilfreich (wenngleich nicht notwendig) Alphabete mit mehr als zwei Zeichen zu betrachten. Wenn nicht explizit anders erwähnt, so gehen wir in dieser Vorlesung stets davon aus, dass die Zeichen 0 und 1 im Alphabet Σ enthalten sind, d. h. $\{0, 1\} \subseteq \Sigma$. Wie üblich bezeichnen wir mit Σ^* die Menge aller Zeichenketten endlicher Länge, die sich über dem Alphabet Σ bilden lassen. Dazu gehört insbesondere das leere Wort ε der Länge 0. Für ein Wort $w \in \Sigma^*$ bezeichnen wir mit $|w|$ seine Länge. Außerdem bezeichnet w^R das gespiegelte Wort, das heißt für $w = w_1 \dots w_n$ ist $w^R = w_n \dots w_1$.

Unter einem *Problem* verstehen wir eine Relation $R \subseteq \Sigma^* \times \Sigma^*$ mit der Eigenschaft, dass es für jede Eingabe $x \in \Sigma^*$ mindestens eine Ausgabe $y \in \Sigma^*$ mit $(x, y) \in R$ gibt. Gibt es zu jeder Eingabe eine eindeutige Ausgabe, so können wir das Problem auch als Funktion $f\colon \Sigma^* \to \Sigma^*$ beschreiben, die jeder Eingabe $x \in \Sigma^*$ ihre Ausgabe $f(x) \in \Sigma^*$ zuweist. Ein Algorithmus *löst* ein Problem, das durch eine Relation R beschrieben

wird, wenn er zu jeder Eingabe $x \in \Sigma^*$ eine Ausgabe $y \in \Sigma^*$ mit $(x, y) \in R$ produziert. Ein Algorithmus löst ein Problem, das durch eine Funktion f beschrieben wird, wenn er zu jeder Eingabe $x \in \Sigma^*$ die Ausgabe $f(x)$ produziert. Wir sagen dann auch, dass der Algorithmus die Funktion f *berechnet*.

Wir legen ein besonderes Augenmerk auf Funktionen der Form $f\colon \Sigma^* \to \{0, 1\}$. Eine solche Funktion beschreibt ein sogenanntes *Entscheidungsproblem*, da für eine Eingabe $x \in \Sigma^*$ nur entschieden werden muss, ob $f(x) = 0$ oder $f(x) = 1$ gilt. Eine *Sprache* über dem Alphabet Σ ist eine Teilmenge von Σ^*. Zwischen Entscheidungsproblemen und Sprachen existiert eine eineindeutige Beziehung. Jedes Entscheidungsproblem $f\colon \Sigma^* \to \{0, 1\}$ kann als Sprache $L_f = \{x \in \Sigma^* \mid f(x) = 1\}$ aufgefasst werden. Ebenso kann jede Sprache $L \subseteq \Sigma^*$ als Funktion $f_L\colon \Sigma^* \to \{0, 1\}$ mit

$$f_L(x) = \begin{cases} 1 & \text{falls } x \in L \\ 0 & \text{falls } x \notin L \end{cases}$$

aufgefasst werden. Die Funktion f_L heißt die *charakteristische Funktion* der Sprache L.

Beispiele

Wir betrachten nun einige Beispiele für Probleme. In diesen Beispielen und im weiteren Verlauf der Vorlesung bezeichnen wir die Binärdarstellung einer Zahl $n \in \mathbb{N}_0$ ohne führende Nullen mit $\text{bin}(n)$ und wir bezeichnen mit $\text{val}(x) \in \mathbb{N}_0$ die Zahl, die durch $x \in \{0, 1\}^*$ binär codiert wird. Erlauben wir in der Binärdarstellung führende Nullen, so ist $\text{val}(x)$ für jede Zeichenkette $x \in \{0, 1\}^*$ eindeutig bestimmt.

- Das Problem, eine natürliche Zahl in Binärdarstellung zu quadrieren, lässt sich sowohl durch die Funktion $f\colon \{0, 1\}^* \to \{0, 1\}^*$ mit $f(x) = \text{bin}(\text{val}(x)^2)$ als auch durch die Relation

  $$R = \{(x, y) \mid \text{val}(y) = \text{val}(x)^2\} \subseteq \Sigma^* \times \Sigma^*$$

  beschreiben. Obwohl beides sinnvolle Modellierungen desselben Problems sind, sind sie formal nicht ganz identisch. Um dies einzusehen, betrachten wir einen Algorithmus, der das Quadrat der Eingabe korrekt berechnet, binär codiert ausgibt und an die Ausgabe zusätzlich noch eine führende Null anfügt. Dieser Algorithmus löst das Problem, das durch die Relation R beschrieben wird, er berechnet aber nicht die Funktion f.
- Das Problem, einen Primfaktor einer natürlichen Zahl zu bestimmen, lässt sich nicht als Funktion beschreiben, da es zu manchen Eingaben mehrere mögliche Ausgaben gibt. Stattdessen kann es aber durch die Relation

  $$R = \{(x, y) \mid \text{val}(y) \text{ ist eine Primzahl, die } \text{val}(x) \text{ teilt}\} \subseteq \Sigma^* \times \Sigma^*$$

  beschrieben werden.
- Wir betrachten als nächstes das Problem, für einen ungerichteten Graphen $G = (V, E)$ zu entscheiden, ob er zusammenhängend ist oder nicht. Um

dieses Problem formal beschreiben zu können, müssen wir zunächst festlegen, wie der Graph G codiert wird. Dazu gibt es zahlreiche Möglichkeiten. Wir entscheiden uns dafür, ihn als Adjazenzmatrix darzustellen, wobei wir die Zeilen dieser Matrix nacheinander schreiben. Für einen Graphen mit n Knoten besitzt diese Darstellung eine Länge von n^2. Somit codieren genau solche Eingaben $x \in \{0,1\}^*$ einen Graphen, deren Länge $|x|$ eine Quadratzahl ist. Ist $|x|$ eine Quadratzahl, so bezeichnen wir mit $G(x)$ den durch x codierten Graphen. Wir können das Zusammenhangsproblem als Funktion $f\colon \{0,1\}^* \to \{0,1\}$ mit

$$f(x) = \begin{cases} 1 & \text{falls } |x| = n^2 \text{ für ein } n \in \mathbb{N} \text{ und } G(x) \text{ ist zusammenhängend} \\ 0 & \text{sonst} \end{cases}$$

modellieren. Diese Funktion gibt den Wert 1 aus, wenn die Eingabe einen zusammenhängenden Graphen codiert, und den Wert 0, wenn die Eingabe entweder syntaktisch nicht korrekt ist oder einen nicht zusammenhängenden Graphen codiert. Dies können wir auch als Sprache

$$L = \{x \in \{0,1\}^* \mid |x| = n^2 \text{ für ein } n \in \mathbb{N} \text{ und } G(x) \text{ ist zusammenhängend}\}$$

ausdrücken.

2.2 Rechnermodelle

Bereits im vergangenen Semester haben wir gesehen, dass zum Entwurf und zur Analyse von Algorithmen ein *Rechnermodell* benötigt wird. Dabei handelt es sich um ein formales Modell, das festlegt, welche Operationen ein Algorithmus ausführen darf und welche Ressourcen (insbesondere welche Rechenzeit) er für diese Operationen benötigt. Wir haben das Modell der *Registermaschine* kennengelernt, das in etwa dieselben Operationen bietet wie eine rudimentäre Assemblersprache. Wir haben uns dann davon überzeugt, dass Registermaschinen das Verhalten realer Rechner gut abbilden, und deshalb für die allermeisten Anwendungen ein vernünftiges Rechnermodell darstellen.

In dieser Vorlesung werden wir jedoch stattdessen das Modell der *Turingmaschine* zugrunde legen. Dieses hat auf den ersten Blick deutlich weniger mit realen Rechnern gemein als Registermaschinen, es ist dafür aber strukturell einfacher und erleichtert deshalb die theoretischen Betrachtungen. Wir werden sehen, dass Turingmaschinen trotz ihrer Einfachheit genauso mächtig wie Registermaschinen sind. Damit sind auch Turingmaschinen ein gutes Modell für reale Rechner, auch wenn man das anhand ihrer Definition nicht direkt erkennen kann.

2.2.1 Turingmaschinen

Informell kann man eine Turingmaschine als einen endlichen Automaten beschreiben, der mit einem Band mit unendlich vielen Speicherzellen ausgestattet ist. In jeder Zelle

Abbildung 2.1: Illustration einer Turingmaschine: Links ist die Konfiguration zu Beginn dargestellt. Die Eingabe ist 1001 und die Turingmaschine befindet sich im Startzustand q_0. Rechts ist die Konfiguration am Ende dargestellt. Hier befindet sich die Turingmaschine im Endzustand $\bar{q}$ und die Ausgabe ist 00.

des Bandes steht eins von endlich vielen Zeichen. Dabei bezeichne Γ die endliche Menge von möglichen Zeichen und $\square \in \Gamma$ sei das Leerzeichen. Wie ein endlicher Automat befindet sich auch eine Turingmaschine zu jedem Zeitpunkt in einem von endlich vielen Zuständen, wobei wir die endliche Zustandsmenge wieder mit Q bezeichnen.

Darüber hinaus besitzt eine Turingmaschine einen Lese-/Schreibkopf, der zu jedem Zeitpunkt auf einer Zelle des Bandes steht und das dort gespeicherte Zeichen liest. Basierend auf dem gelesenen Zeichen und ihrem aktuellen Zustand kann die Turingmaschine in einem Rechenschritt das Zeichen an der aktuellen Kopfposition ändern, in einen anderen Zustand wechseln und den Kopf um eine Position nach links oder rechts verschieben. Zu Beginn befindet sich die Eingabe auf dem Band, der Kopf steht auf dem ersten Zeichen der Eingabe (oder auf einem Leerzeichen, wenn die Eingabe das leere Wort ε ist), alle Zellen links und rechts von der Eingabe enthalten das Leerzeichen $\square$ und die Turingmaschine befindet sich im Startzustand $q_0 \in Q$. Die Eingabe ist dabei keine Zeichenkette aus Γ^*, sondern aus Σ^* für ein Eingabealphabet $\Sigma \subseteq \Gamma$ mit $\square \notin \Sigma$.

Die Turingmaschine führt solange Rechenschritte der oben beschriebenen Form aus, bis sie einen bestimmten Endzustand $\bar{q} \in Q$ erreicht. Bezeichne $w_i \in \Gamma$ beim Erreichen des Endzustandes $\bar{q}$ den Inhalt von Zelle i und sei $j \in \mathbb{Z}$ die Kopfposition zu diesem Zeitpunkt. Dann ist die Zeichenkette $w_j \dots w_{k-1} \in \Sigma^*$ die Ausgabe, wobei $k \geq j$ den eindeutigen Index bezeichne, für den $w_j, \dots, w_{k-1} \in \Sigma$ und $w_k \in \Gamma \setminus \Sigma$ gilt. Die Ausgabe ist also die Zeichenkette aus Σ^*, die an der aktuellen Kopfposition beginnt und nach rechts durch das erste Zeichen aus $\Gamma \setminus \Sigma$ beschränkt ist. Abbildung 2.1 zeigt ein Beispiel.

Genauso wie bei endlichen Automaten sind die Rechenschritte einer Turingmaschine durch eine Zustandsüberführungsfunktion δ bestimmt. Diese erhält als Eingabe den aktuellen Zustand und das Zeichen an der aktuellen Kopfposition und liefert als Ausgabe den neuen Zustand, das Zeichen, durch das das Zeichen an der Kopfposition ersetzt werden soll, und ein Element aus der Menge $\{L, N, R\}$, das angibt, ob der Kopf an der aktuellen Position stehen bleiben soll (N), eine Position nach links (L) oder eine Position nach rechts (R) verschoben werden soll. Formal handelt es sich dabei um eine Funktion $\delta\colon (Q \setminus \{\bar{q}\}) \times \Gamma \to Q \times \Gamma \times \{L, N, R\}$.

Definition 2.1. *Eine* Turingmaschine (TM) *M ist ein 7-Tupel $(Q, \Sigma, \Gamma, \square, q_0, \bar{q}, \delta)$, das aus den folgenden Komponenten besteht.*

- *Q, die* Zustandsmenge*, ist eine endliche Menge von* Zuständen.

- $\Sigma \supseteq \{0,1\}$, *das* Eingabealphabet, *ist eine endliche Menge von Zeichen.*
- $\Gamma \supseteq \Sigma$, *das* Bandalphabet, *ist eine endliche Menge von Zeichen.*
- $\square \in \Gamma \setminus \Sigma$ *ist das* Leerzeichen.
- $q_0 \in Q$ *ist der* Startzustand.
- $\bar{q} \in Q$ *ist der* Endzustand.
- $\delta\colon (Q \setminus \{\bar{q}\}) \times \Gamma \to Q \times \Gamma \times \{L, N, R\}$ *ist die* Zustandsüberführungsfunktion.

Abbildung 2.2 zeigt exemplarisch die Rechenschritte einer Turingmaschine. Die Eingabe ist das Wort $1001 \in \Sigma^*$, welches links und rechts von Leerzeichen umschlossen ist. Der Kopf steht zu Beginn auf dem ersten Zeichen der Eingabe und die Turingmaschine befindet sich im Startzustand q_0. Die Zustandsüberführungsfunktion δ gibt vor, dass im Zustand q_0 beim Lesen des Zeichens 1 in den Zustand q gewechselt wird, das Zeichen 1 geschrieben wird und der Kopf um eine Position nach rechts bewegt wird. Im zweiten Schritt wird in Zustand q das Zeichen 0 gelesen, was gemäß δ dazu führt, dass in den Zustand q' gewechselt, das Zeichen 1 geschrieben und der Kopf um eine Position nach rechts bewegt wird. Im dritten und letzten Schritt wird das Zeichen 0 im Zustand q' gelesen, was gemäß δ dazu führt, dass in den Endzustand $\bar{q}$ gewechselt, das Zeichen 0 geschrieben und der Kopf nicht bewegt wird. Die Ausgabe der Turingmaschine ist in diesem Beispiel das Wort $01 \in \Sigma^*$.

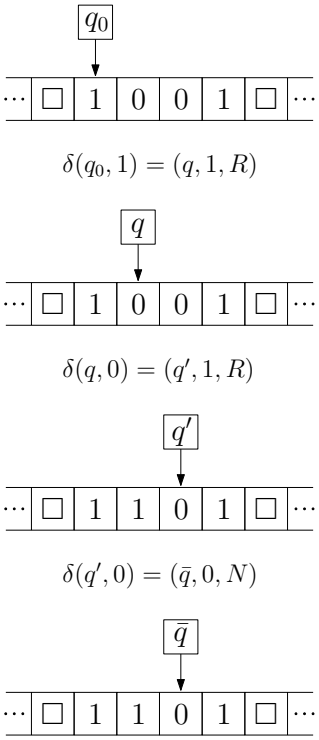


Abbildung 2.2: Beispiel für das Verhalten einer Turingmaschine

Mit jeder Turingmaschine M kann man eine Funktion $f_M\colon \Sigma^* \to \Sigma^* \cup \{\bot\}$ assoziieren, die für jede Eingabe $w \in \Sigma^*$ angibt, welche Ausgabe $f_M(w)$ die Turingmaschine bei dieser Eingabe produziert. In dem gerade besprochenen Beispiel gilt folglich $f_M(1001) = 01$. Erreicht die Turingmaschine M bei einer Eingabe w den Endzustand $\bar{q}$ nicht nach endlich vielen Schritten, so sagen wir, dass sie bei Eingabe w *nicht hält* (oder *nicht terminiert*), und wir definieren $f_M(w) = \bot$. Wir sagen, dass *die Turingmaschine M die Funktion f_M berechnet.*

Definition 2.2. *Eine Funktion $f\colon \Sigma^* \to \Sigma^*$ heißt* berechenbar *(oder* rekursiv[1]*), wenn es eine Turingmaschine M mit $f_M = f$ gibt. Eine solche Turingmaschine terminiert insbesondere auf jeder Eingabe.*

[1]Auf den ersten Blick hat das Wort „rekursiv“ in diesem Kontext nichts mit dem normalen Gebrauch dieses Wortes in der Informatik zu tun. Die Bezeichnung stammt von einer alternativen Charakterisierung berechenbarer Funktionen, die wir in dieser Vorlesung nicht besprechen werden.

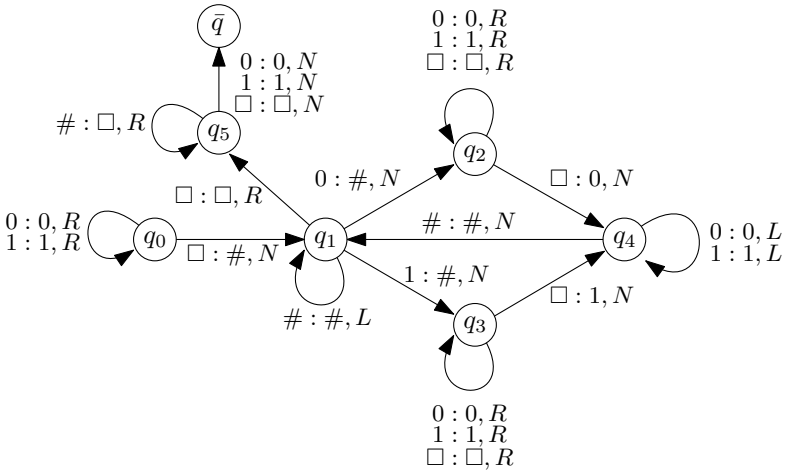


Abbildung 2.3: Darstellung der Turingmaschine aus dem Beispiel in Automatennotation: Ähnlich wie bei der Darstellung endlicher Automaten stellen wir die Zustandsüberführungfunktion als Graphen dar. Die Knoten entsprechen dabei den Zuständen und die Kanten stellen die möglichen Rechenschritte dar. Ist eine Kante von einem Zustand q zu einem Zustand q' mit $a : b, D$ beschriftet, so stellt dies den Rechenschritt $\delta(q, a) = (q', b, D)$ dar.

Wenn wir das Verhalten einer Turingmaschine beschreiben oder analysieren, so sprechen wir im Folgenden oft von der *Konfiguration* einer Turingmaschine. Darunter verstehen wir zu einem Zeitpunkt die aktuelle Kombination aus Bandinhalt, Kopfposition und Zustand.

Beispiel

Wir betrachten ein Beispiel für eine Turingmaschine $M = (Q, \Sigma, \Gamma, \square, q_0, \bar{q}, \delta)$. Es sei $Q = \{q_0, q_1, \ldots, q_5, \bar{q}\}$, $\Sigma = \{0, 1\}$ und $\Gamma = \{0, 1, \#, \square\}$. Die Zustandsüberführungsfunktion δ ist in der folgenden Tabelle dargestellt. Diese enthält an zwei Stellen das Symbol —, da die entsprechenden Kombinationen von Zustand und Zeichen nicht auftreten können (wie wir weiter unten argumentieren werden). Formal könnte man dort ein beliebiges Tripel aus $Q \times \Gamma \times \{L, N, R\}$ einsetzen, um die Tabelle zu vervollständigen.

	q_0	q_1	q_2	q_3	q_4	q_5
0	$(q_0, 0, R)$	$(q_2, \#, N)$	$(q_2, 0, R)$	$(q_3, 0, R)$	$(q_4, 0, L)$	$(\bar{q}, 0, N)$
1	$(q_0, 1, R)$	$(q_3, \#, N)$	$(q_2, 1, R)$	$(q_3, 1, R)$	$(q_4, 1, L)$	$(\bar{q}, 1, N)$
#	—	$(q_1, \#, L)$	$(q_2, \#, R)$	$(q_3, \#, R)$	$(q_1, \#, N)$	$(q_5, \square, R)$
$\square$	$(q_1, \#, N)$	$(q_5, \square, R)$	$(q_4, 0, N)$	$(q_4, 1, N)$	—	$(\bar{q}, \square, N)$

Anstatt in Tabellenform ist es oft übersichtlicher, die Zustandsüberführungsfunktion einer Turingmaschine grafisch darzustellen. Für die Turingmaschine aus diesem Beispiel erläutern wir diese Darstellung in Abbildung 2.3.

Um zu verstehen, welche Funktion diese Turingmaschine berechnet, kann man sich

zunächst ihr Verhalten auf einigen Beispielen anschauen. Diese kann man entweder von Hand durchgehen oder man schreibt ein Programm, mit dem man Turingmaschinen simulieren kann. Wir können aber auch versuchen, direkt zu verstehen, was die Turingmaschine in den einzelnen Zuständen macht. Die folgenden Eigenschaften können wir direkt aus der Zustandsüberführungsfunktion ableiten.

- q_0: M schiebt den Kopf nach rechts bis zum Ende der Eingabe (ohne den Bandinhalt dabei zu verändern) und schreibt anschließend rechts neben die Eingabe das Zeichen #. Danach wird in den Zustand q_1 gewechselt und Zustand q_0 wird nie wieder erreicht.
- q_1: M schiebt den Kopf nach links, solange an der aktuellen Position das Zeichen # steht. Das weitere Vorgehen ist abhängig vom ersten Zeichen ungleich #, das erreicht wird. Ist es 0 oder 1, so wird es durch # ersetzt und in den Zustand q_2 bzw. q_3 gewechselt. Ist es ein Leerzeichen, so wird in den Zustand q_5 gewechselt.
- q_2: M schiebt den Kopf nach rechts bis zum ersten Leerzeichen, welches sie durch 0 ersetzt. Anschließend wechselt sie in den Zustand q_4.
- q_3: M schiebt den Kopf nach rechts bis zum ersten Leerzeichen, welches sie durch 1 ersetzt. Anschließend wechselt sie in den Zustand q_4.
- q_4: M schiebt den Kopf nach links, bis sie das erste Mal das Zeichen # erreicht. Dann wechselt sie in den Zustand q_1.
- q_5: M schiebt den Kopf nach rechts, bis sie das erste Zeichen ungleich # erreicht. Alle Rauten, die sie dabei sieht, werden durch Leerzeichen ersetzt. Danach terminiert M.

Nun können wir einen Schritt weitergehen und uns überlegen, was aus diesen Eigenschaften folgt. Dazu betrachten wir das Verhalten der Turingmaschine M, wenn ihr Bandinhalt von der Form $xa\#\ldots\#y$ für $x, y \in \{0,1\}^*$ und $a \in \{0,1\}$ ist, der Kopf auf der Raute am weitesten rechts steht und die TM sich im Zustand q_1 befindet. Diese Konfiguration erreicht M wenn sie von q_0 in den Zustand q_1 wechselt (zu diesem Zeitpunkt entspricht xa noch der Eingabe, y ist das leere Wort und genau eine Raute steht auf dem Band). In den folgenden Abbildungen ist das Verhalten der Turingmaschine M in dieser Konfiguration dargestellt (links für $a = 0$ und rechts für $a = 1$). Dabei ist nicht jeder einzelne Schritt eingezeichnet, sondern nur diejenigen, in denen Zustandswechsel erfolgen.

$$
\begin{array}{llll}
 & & q_1 & \\
x_1 \cdots x_n\ 0\ \# \cdots \#\ \#\ y_1 \cdots y_m\ \square & & x_1 \cdots x_n\ 1\ \# \cdots \#\ \#\ y_1 \cdots y_m\ \square & \\
 & q_1 & & \\
x_1 \cdots x_n\ 0\ \# \cdots \#\ \#\ y_1 \cdots y_m\ \square & & x_1 \cdots x_n\ 1\ \# \cdots \#\ \#\ y_1 \cdots y_m\ \square & \\
 & q_2 & & \\
x_1 \cdots x_n\ \#\ \# \cdots \#\ \#\ y_1 \cdots y_m\ \square & & x_1 \cdots x_n\ \#\ \# \cdots \#\ \#\ y_1 \cdots y_m\ \square & \\
 & & & q_2 \\
x_1 \cdots x_n\ \#\ \# \cdots \#\ \#\ y_1 \cdots y_m\ \square & & x_1 \cdots x_n\ \#\ \# \cdots \#\ \#\ y_1 \cdots y_m\ \square & \\
 & & & q_4 \\
x_1 \cdots x_n\ \#\ \# \cdots \#\ \#\ y_1 \cdots y_m\ 0 & & x_1 \cdots x_n\ \#\ \# \cdots \#\ \#\ y_1 \cdots y_m\ 1 & \\
 & & q_4 & \\
x_1 \cdots x_n\ \#\ \# \cdots \#\ \#\ y_1 \cdots y_m\ 0 & & x_1 \cdots x_n\ \#\ \# \cdots \#\ \#\ y_1 \cdots y_m\ 1 & \\
 & & q_1 & \\
x_1 \cdots x_n\ \#\ \# \cdots \#\ \#\ y_1 \cdots y_m\ 0 & & x_1 \cdots x_n\ \#\ \# \cdots \#\ \#\ y_1 \cdots y_m\ 1 &
\end{array}
$$

Nach den dargestellten Schritten befindet sich die Turingmaschine wieder in fast derselben Konfiguration wie zu Beginn. Der einzige Unterschied ist, dass der Bandinhalt nun von der Form $x\# \ldots \#ya$ ist. Das heißt, der letzte Buchstabe auf der linken Seite wurde an die rechte Seite angehängt. Dies wird solange wiederholt, bis die linke Seite leer ist. Anschließend werden die Rauten im Zustand q_5 gelöscht. Danach steht der Kopf auf dem ersten Zeichen der rechten Seite und die Turingmaschine terminiert. Aus diesen Überlegungen folgt, dass M die Funktion $f(w) = w^R$ berechnet. Sie dreht also die Eingabe um.

Natürlich ist es nicht Sinn und Zweck dieser Vorlesung, zu lernen, wie man Turingmaschinen konstruiert. Es wird vermutlich nur sehr wenigen Spaß machen, eine Turingmaschine für das Zusammenhangsproblem oder andere nicht triviale Probleme zu konstruieren (auch wenn dies möglich wäre). Das Spielen mit dem Modell der Turingmaschine dient an dieser Stelle nur dazu, ein Gefühl dafür zu entwickeln, wie mächtig Turingmaschinen sind und wie man prinzipiell damit Probleme lösen und Funktionen berechnen kann.

Wir passen Definition 2.2 nun noch an Entscheidungsprobleme und Sprachen an. Bei einem solchen Problem besteht die Ausgabe immer aus genau einem Zeichen (0 oder 1), weshalb uns nur das Zeichen interessiert, das am Ende der Berechnung an der Kopfposition steht. Dies ist in folgender Definition formalisiert.

Definition 2.3. *Eine Turingmaschine M* akzeptiert *eine Eingabe $w \in \Sigma^*$, wenn sie bei Eingabe w terminiert und ein Wort ausgibt, das mit* 1 *beginnt. Sie* verwirft *eine Eingabe $w \in \Sigma^*$, wenn sie bei Eingabe w terminiert und ein Wort ausgibt, das nicht mit* 1 *beginnt.*

Eine Turingmaschine M entscheidet *eine Sprache $L \subseteq \Sigma^*$, wenn sie jedes Wort $w \in L$ akzeptiert und jedes Wort $w \in \Sigma^* \setminus L$ verwirft.*

Eine Sprache $L \subseteq \Sigma^*$ heißt entscheidbar *oder* rekursiv*, wenn es eine Turingmaschine M gibt, die L entscheidet. Wir sagen dann, dass M* eine Turingmaschine für die Sprache L ist*. Eine solche Turingmaschine terminiert insbesondere auf jeder Eingabe.*

Beispiel

Wir betrachten als Beispiel eine Turingmaschine $M = (Q, \Sigma, \Gamma, \square, q_0, \bar{q}, \delta)$ mit $Q = \{q_0, q_1, q_2, \bar{q}\}$, $\Sigma = \{0, 1\}$ und $\Gamma = \{0, 1, \square\}$. Die Zustandsüberführungsfunktion δ ist in der folgenden Tabelle dargestellt.

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

Wir beobachten zunächst, dass die Turingmaschine in jedem Schritt, in dem sie nicht in den Endzustand wechselt, den Kopf nach rechts bewegt und den Bandinhalt nicht ändert. Die Turingmaschine terminiert, sobald sie das erste Leerzeichen (also das Ende der Eingabe) erreicht hat. Die Turingmaschine verhält sich also wie ein endlicher Automat, der die Eingabe Zeichen für Zeichen von links nach rechts liest und dabei Zustandsübergänge durchführt. Sie akzeptiert die Eingabe genau dann, wenn sie das erste Leerzeichen im Zustand q_2 erreicht. Ansonsten verwirft sie die Eingabe. Betrachtet man die Zustandsübergänge, so sieht man, dass Zustand q_2 genau dann erreicht wird, wenn der bisher gelesene Teil der Eingabe mit 01 endet. Daraus folgt insgesamt, dass die Turingmaschine M genau die Wörter akzeptiert, die mit 01 enden. Alle anderen Wörter verwirft sie.

Das Modell der Turingmaschine wurde bereits 1937 von dem britischen Mathematiker Alan Turing eingeführt [5]. Turing geht in dieser wegweisenden Arbeit der Frage nach, welche Funktionen $f\colon \mathbb{N} \to \{0, 1\}$ von Computern berechnet werden können. Allerdings hatte der Begriff *Computer* damals eine andere Bedeutung als heute. Man verstand darunter einen Menschen, der mathematische Berechnungen gemäß fester Regeln durchführt, also gewissermaßen einen Menschen, der einen Algorithmus von Hand ausführt. Solche menschlichen Computer wurden einige Jahrhunderte lang für aufwendige Berechnungen in Wissenschaften wie beispielsweise der Astronomie eingesetzt. Letztlich spielt es bei der Frage, welche Funktionen durch Algorithmen berechnet werden können, aber keine Rolle, ob die Algorithmen von einem Menschen oder einem elektronischen Computer ausgeführt werden.

Techniken zum Entwurf von Turingmaschinen

Um ein besseres Gefühl dafür zu bekommen, wie man prinzipiell Turingmaschinen für nicht triviale Probleme entwerfen kann, ist es hilfreich, sich zu überlegen, wie man grundlegende Programmiertechniken auf Turingmaschinen anwenden kann.

1. Eine Variable, die Werte aus einer endlichen Menge annehmen kann, kann in der Zustandsmenge einer Turingmaschine gespeichert werden. Möchte man beispielsweise eine Variable realisieren, die für ein festes $k \in \mathbb{N}$ Werte aus der Menge $\{0, \ldots, k\}$ annehmen kann, so kann man die Zustandsmenge Q zu der Menge $Q' = Q \times \{0, \ldots, k\}$ erweitern. Ein Zustand aus Q' ist dann ein Paar (q, a),

wobei $q \in Q$ den eigentlichen Zustand der Turingmaschine beschreibt und $a \in \{0, \ldots, k\}$ den Wert der Variablen. Die Zustandsüberführungsfunktion kann diesen Wert berücksichtigten und ihn gegebenenfalls ändern. Es ist allerdings wichtig, dass k konstant ist, da die erweiterte Zustandsmenge Q' endlich sein muss. Das bedeutet insbesondere, dass wir die Variable nicht dazu einsetzen können, um zum Beispiel die Länge der Eingabe zu speichern. Analog können auch endlich viele Variablen mit endlichen Wertebereichen realisiert werden.

2. Oft ist es hilfreich, wenn das Band der Turingmaschine aus verschiedenen Spuren besteht. Das bedeutet, in jeder Zelle stehen k Zeichen aus Γ, die alle gleichzeitig in einem Schritt gelesen und geschrieben werden. Ist $k \in \mathbb{N}$ eine Konstante, so kann dies dadurch realisiert werden, dass das Bandalphabet Γ zu $\Gamma' = \Sigma \cup \Gamma^k$ erweitert wird. Dies ist für endliches Γ und endliches k eine endliche Menge. Die Zustandsüberführungsfunktion kann dann entsprechend angepasst werden und in jedem Schritt die k Zeichen verarbeiten, die sich an der aktuellen Kopfposition befinden. Wir werden im Folgenden ein Zeichen $a \in \Sigma$ mit dem Element $(a, \square, \square, \ldots, \square) \in \Gamma^k$ identifizieren.

   Um den Sinn von mehreren Spuren und das Abstraktionsniveau, auf dem wir zukünftig Turingmaschinen beschreiben wollen, zu verdeutlichen, betrachten wir das Problem, zwei gegebene Zahlen in Binärdarstellung zu addieren. Sei $\Sigma = \{0, 1, \#\}$ das Eingabealphabet und sei $\text{bin}(x)\#\text{bin}(y)$ für $x \in \mathbb{N}$ und $y \in \mathbb{N}$ eine Eingabe. Die Aufgabe ist es, $\text{bin}(x+y)$ zu berechnen. Dazu konstruieren wir eine Turingmaschine mit drei Spuren und dem Bandalphabet

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

   Die Turingmaschine arbeitet in mehreren Phasen. In der ersten Phase verschiebt sie die Binärdarstellungen $\text{bin}(x)$ und $\text{bin}(y)$ so, dass sie auf den ersten beiden Spuren rechtsbündig untereinander stehen (ggf. mit führenden Nullen, wenn die Darstellungen nicht dieselbe Länge haben). Die Eingabe 101#1100 wird in dieser Phase beispielsweise in den Bandinhalt

$$\ldots \square \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \square \ldots$$

   überführt. In der zweiten Phase geht die Turingmaschine die Bits von rechts nach links durch und führt die Addition bitweise nach der Schulmethode durch. Das Ergebnis wird dabei auf die dritte Spur geschrieben. Tritt in einem Schritt ein Übertrag auf, so merkt die Turingmaschine sich dies im Zustand. In dem obigen Beispiel ergibt sich nach dieser Phase der Bandinhalt

$$\ldots \square \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \square \ldots$$

In der dritten Phase wird das Ergebnis zurück in das (einspurige) Eingabealphabet Σ übersetzt. Dabei werden die Inhalte der ersten beiden Spuren gelöscht und nur der Inhalt der dritten Spur bleibt erhalten.

3. Mithilfe mehrerer Spuren können auch Variablen realisiert werden, die beliebig viele Zustände annehmen können. Diese können nicht mehr in den Zustand codiert werden, sondern sie müssen auf dem Band gespeichert werden. Man könnte zum Beispiel für jede Variable eine Spur benutzen oder mehrere durch Trennzeichen getrennte Variablen auf derselben Spur abspeichern.

4. Mit etwas Aufwand kann man beim Entwurf von Turingmaschinen auch Unterprogramme einsetzen. Dazu kann man beispielsweise für jedes Unterprogramm eine Teilmenge von Zuständen auszeichnen, die ausschließlich in diesem Unterprogramm angenommen werden. Ebenso kann man beim Aufruf eines Unterprogramms einen separaten (beispielsweise weit rechts liegenden) Speicherbereich auf dem Band reservieren, in dem die Berechnungen des Unterprogramms erfolgen. Es können dabei einige technische Probleme auftreten. Genügt der reservierte Speicherbereich beispielsweise nicht, so muss er gegebenenfalls verschoben werden. Ebenso muss ein Aufrufstack verwaltet werden. All dies ist möglich, aber relativ komplex und wenig elegant. Wir wollen die Diskussion hier nicht weiter vertiefen. Für die weiteren Betrachtungen ist lediglich wichtig zu wissen, dass Unterprogramme prinzipiell realisiert werden können.

5. Auch können for- und while-Schleifen in Turingmaschinen realisiert werden. Eine solche Schleife haben wir bereits implizit in der obigen Turingmaschine gesehen, die bei Eingabe w das gespiegelte Wort w^R berechnet. Letztendlich kann man Schleifen aber auch als eine spezielle Art von Unterprogrammen auffassen, sodass sie genauso wie andere Unterprogramme realisiert werden können.

Turingmaschinen mit mehreren Bändern

Wir haben oben diskutiert, dass Turingmaschinen mit mehreren Spuren durch eine einfache Anpassung des Bandalphabets simuliert werden können. Diese Maschinen besitzen aber weiterhin nur einen Lese-/Schreibkopf, der die k Zeichen unter der aktuellen Kopfposition gleichzeitig verarbeiten kann. Nun gehen wir einen Schritt weiter und betrachten Turingmaschinen mit einer konstanten Anzahl an Bändern. Bei diesen Maschinen besitzt jedes Band einen eigenen Lese-/Schreibkopf, der unabhängig von den anderen Köpfen bewegt werden kann.

Eine *k-Band-Turingmaschine* M ist ein 7-Tupel $(Q, \Sigma, \Gamma, \square, q_0, \bar{q}, \delta)$, bei dem die ersten sechs Komponenten die gleiche Bedeutung haben wie in Definition 2.1. Die Zustandsüberführungsfunktion δ ist von der Form

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

d. h. sie erhält als Eingabe den aktuellen Zustand und die k Zeichen, die sich an den aktuellen Kopfpositionen befinden. Als Ausgabe liefert sie den Nachfolgezustand, die k Zeichen, durch die die alten ersetzt werden sollen, und die Bewegungen der k Köpfe.

Das erste Band fungiert dabei wie bei einer normalen Turingmaschine als Ein- und Ausgabeband. Die anderen Bänder sind initial leer. Eine 1-Band-Turingmaschine ist somit identisch zu einer normalen Turingmaschine gemäß Definition 2.1.

Da sich die Köpfe unabhängig bewegen können, erscheint eine einfache Simulation durch eine 1-Band-Turingmaschine nicht ohne Weiteres möglich zu sein. Wir werden nun aber beweisen, dass k-Band-Turingmaschinen stets durch normale Turingmaschinen simuliert werden können, wobei allerdings ein gewisser Zeitverlust auftritt. Um dies quantifizieren zu können, definieren wir die Begriffe Rechenzeit und Platzbedarf zunächst formal.

Definition 2.4. *Es sei M eine k-Band-Turingmaschine. Die* Rechenzeit $t_M(w)$ von M auf Eingabe w *ist die Anzahl an Rechenschritten, die M bei Eingabe w bis zur Terminierung durchführt. Terminiert M auf w nicht, so ist die Rechenzeit unendlich. Der* Platzbedarf $s_M(w)$ von M auf Eingabe w *ist die Anzahl (summiert über alle Bänder) an verschiedenen Zellen, auf denen sich im Laufe der Rechnung mindestens einmal ein Lese-/Schreibkopf befunden hat. Terminiert M nicht, so kann der Platzbedarf unendlich sein.*

Die Rechenzeit $t_M(n)$ von M auf Eingaben der Länge $n \in \mathbb{N}$ *ist die maximale Rechenzeit, die bei Eingaben der Länge n auftritt, also $t_M(n) = \max_{w \in \Sigma^n} t_M(w)$. Analog ist der* Platzbedarf $s_M(n)$ von M auf Eingaben der Länge $n \in \mathbb{N}$ *als $s_M(n) = \max_{w \in \Sigma^n} s_M(w)$ definiert.*

Nun werden wir sehen, wie man k-Band-Turingmaschinen durch 1-Band-Turingmaschinen simulieren kann.

Theorem 2.5. *Eine k-Band Turingmaschine M mit Rechenzeit $t(n)$ und Platzbedarf $s(n)$ kann durch eine 1-Band-Turingmaschine M' mit Rechenzeit $O(t(n)^2)$ und Platzbedarf $O(s(n))$ simuliert werden.*

Beweis. Die Turingmaschine M' simuliert die Turingmaschine M Schritt für Schritt und verwendet dafür $2k$ Spuren. Nach der Simulation des t-ten Rechenschrittes von M durch M' mit $t \in \mathbb{N}_0$ sind die folgenden Invarianten erfüllt.

1. Die ungeraden Spuren $1, 3, \ldots, 2k-1$ enthalten den Inhalt der k Bänder von M.
2. Auf den geraden Spuren $2, 4, \ldots, 2k$ sind die Kopfpositionen von M mit dem Zeichen # markiert (Spur $2i$ enthält die Markierung für die Kopfposition auf Band i von M). Alle anderen Zellen auf diesen Bändern enthalten das Leerzeichen.
3. Der Kopf von M' steht an der Position am weitesten links, die auf einem der geraden Bänder mit # markiert ist.

Die Spuren können in konstanter Zeit initialisiert werden. Dazu muss lediglich das Zeichen # an der aktuellen Position (die zu Beginn dem ersten Zeichen der Eingabe entspricht) auf alle geraden Spuren geschrieben werden. Um nun einen Rechenschritt von M zu simulieren, geht die Turingmaschine M' wie folgt vor. Sie schiebt den Kopf

solange nach rechts, bis sie alle k Markierungen # gesehen hat. Dabei merkt sie sich im Zustand, welche k Zeichen sich an den aktuellen Kopfpositionen befinden. Nachdem sie die letzte Markierung erreicht und alle k Zeichen gelesen hat, wertet sie die Zustandsüberführungsfunktion von M aus und merkt sich im Zustand den neuen Zustand von M, durch welche Zeichen die k Zeichen an den Kopfpositionen ersetzt werden sollen und wie die Köpfe sich bewegen sollen. Anschließend schiebt sie den Kopf wieder nach links zurück und ändert dabei den Bandinhalt an den markierten Positionen und verschiebt die Markierungen entsprechend den Bewegungen von M. Sie stoppt, sobald sie alle k Markierungen gesehen und den Rechenschritt von M komplett simuliert hat. Danach ist die Invariante wieder hergestellt. Die Leserinnen und Leser sollten sich noch einmal klarmachen, dass M' sich bei dieser Konstruktion nur konstant viele Informationen im Zustand merken muss.

Die Anzahl an Schritten, die M' benötigt, um einen Schritt von M zu simulieren, ist proportional zu dem Abstand, den die beiden Markierungen am weitesten links und am weitesten rechts voneinander haben, da M' diese Entfernung zweimal zurücklegen muss. Da die Laufzeit von M durch $t(n)$ beschränkt ist, können diese beiden Markierungen zu jedem Zeitpunkt um maximal $2t(n)$ viele Positionen auseinander liegen (schlimmstenfalls geht ein Kopf von M in jedem Schritt nach links und ein anderer in jedem Schritt nach rechts). Jeder einzelne der $t(n)$ vielen Schritte kann somit in Zeit $O(t(n))$ simuliert werden, woraus direkt die behauptete Rechenzeit von $O(t(n)^2)$ folgt. Dass der Platzbedarf von M' in derselben Größenordnung wie der von M liegt, folgt direkt daraus, dass M' nur dann eine Zelle besucht, wenn M auf einem seiner Bänder ebenfalls diese Zelle besucht. □

2.2.2 Registermaschinen

Wir haben uns bereits im letzten Semester mit dem Modell der *Registermaschine* beschäftigt, das an eine rudimentäre Assemblersprache erinnert, die auf die wesentlichen Befehle reduziert wurde. In diesem Modell steht als Speicher eine unbegrenzte Anzahl an Registern zur Verfügung, die jeweils eine natürliche Zahl enthalten und auf denen grundlegende arithmetische Operationen durchgeführt werden können. Die Inhalte zweier Register können addiert, subtrahiert, multipliziert und dividiert werden. Ebenso können Registerinhalte kopiert werden und Register können mit Konstanten belegt werden. Darüber hinaus unterstützen Registermaschinen unbedingte Sprungoperationen (GOTO) und bedingte Sprungoperationen (IF), wobei die erlaubten Bedingungen stark eingeschränkt sind.

Formal besteht ein Registermaschinenprogramm aus einer endlichen durchnummerierten Folge von Befehlen, deren Syntax in Tabelle 2.1 zusammengefasst ist. Wird ein solches Programm ausgeführt, so wird ein Befehlszähler b verwaltet, der angibt, welcher Befehl als nächstes ausgeführt werden soll. Zu Beginn wird $b = 1$ gesetzt. Jeder Befehl ändert den Befehlszähler (ist es kein Sprungbefehl, so wird b lediglich um eins erhöht) und gegebenenfalls den Inhalt der Register. Wir bezeichnen mit $c(0), c(1), c(2), \ldots$ die Inhalte der Register. Dabei gilt stets $c(i) \in \mathbb{N}_0$. Zu Beginn gilt $c(0) = 0$, in den ersten N Registern $c(1), \ldots, c(N)$ steht für ein $N \in \mathbb{N}$ die Eingabe und für alle anderen

Register gilt $c(i) = 0$. Die Ausgabe findet sich am Ende der Rechnung in vorher festgelegten Registern, standardmäßig in Register $c(0)$. In Tabelle 2.1 ist auch die Semantik der einzelnen Befehle aufgeführt.

In einer Registermaschine kann jedes Register eine beliebig große natürliche Zahl enthalten. Im letzten Semester haben wir das *uniforme Kostenmaß* zugrunde gelegt, bei dem die Ausführung jedes Befehls unabhängig von der Größe der Zahlen eine Zeiteinheit benötigt. Werden in den Registern große Zahlen gespeichert, so ist es realistischer das *logarithmische Kostenmaß* einzusetzen. Bei diesem ist die Laufzeit eines Befehls proportional zu der Länge der Zahlen in den angesprochenen Registern in Binärdarstellung. Die Laufzeit eines Befehl ist also proportional zum Logarithmus der beteiligten Zahlen.

Analog zu Definition 2.4 interessiert uns auch bei Registermaschinen die Laufzeit in Abhängigkeit von der Eingabelänge. Diese messen wir im logarithmischen Kostenmaß durch die Summe der Längen der Binärdarstellungen der in den Eingaberegistern $c(1), \ldots, c(N)$ gespeicherten Zahlen. Wir nennen eine Registermaschine im logarithmischen Kostenmaß $t(n)$-*zeitbeschränkt*, wenn die Laufzeit im logarithmischen Kostenmaß für jedes $n \in \mathbb{N}$ und jede Eingabe der Länge n durch $t(n)$ nach oben beschränkt ist.

Beispiel

Als Beispiel entwerfen wir eine Registermaschine, die die Summe $\sum_{i=1}^{n} i$ berechnet, wobei die Zahl $n \in \mathbb{N}$ die Eingabe ist, die sich zu Beginn in Register $c(1)$ befindet. In einer höheren Programmiersprache können wir die Summe beispielsweise mit einer for-Schleife wie folgt ausrechnen:

```
s = 0
for i = 1 to n
  s = s + i
return s
```

In einem Registermaschinenprogramm können wir eine Schleife realisieren, indem wir Sprungbefehle verwenden. Wir müssen außerdem darauf achten, dass wir die notwendigen Informationen passend zwischen den Registern verschieben. Das folgende Registermaschinenprogramm speichert in Register $c(1)$ den Wert der Variable i, der anders als in der obigen for-Schleife von n bis 1 runtergezählt wird. In Register $c(2)$ ist der Wert der Variable s gespeichert.

```
1. LOAD(1)
2. STORE(2)
3. CSUB(1)
4. STORE(1)
5. ADD(2)
6. STORE(2)
7. LOAD(1)
```

```
 8. IF c(0)=0 GOTO 10
 9. GOTO 3
10. LOAD(2)
11. END
```

In ähnlicher Weise lässt sich eine Schleife auch in einer Assemblersprache realisieren. Das Programm führt $2 + 7n + 1 = 7n + 3$ Operationen durch, und wir können seine Laufzeit im logarithmischen Kostenmaß mit $O(n \log n)$ abschätzen, da die Werte aller Register durch $O(n^2)$ nach oben beschränkt sind und somit jede der $O(n)$ Operation Kosten $O(\log n)$ verursacht. Tatsächlich beträgt die Laufzeit sogar $\Theta(n \log n)$. Wir bemerken, dass dies im Vergleich zur Eingabelänge $\Theta(\log n)$ exponentiell ist. Tatsächlich lässt sich das Ergebnis wesentlich effizienter mithilfe der Gauß'schen Summenformel $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ berechnen:

```
1. LOAD(1)
2. CADD(1)
3. MULT(1)
4. CDIV(2)
5. END
```

Dieses Registermaschinenprogramm führt nur vier Operationen aus und hat im logarithmischen Kostenmaß eine Laufzeit von $\Theta(\log n)$, was linear in der Eingabelänge ist.

Um die Mächtigkeit des Modells der Turingmaschine zu verdeutlichen, diskutieren wir nun, dass Registermaschinen durch Turingmaschinen simuliert werden können und andersherum. Da wir Registermaschinen bereits im letzten Semester als realistisches Modell realer Rechner akzeptiert haben, bedeutet dies, dass auch Turingmaschinen ein realistisches Modell darstellen.

Theorem 2.6. *Jede im logarithmischen Kostenmaß $t(n)$-zeitbeschränkte Registermaschine kann durch eine Turingmaschine simuliert werden, deren Rechenzeit $O(q(n + t(n)))$ für ein Polynom q beträgt.*

Wir werden das Theorem nicht beweisen. Der Beweis ist zwar nicht schwer, aber wenig erhellend. Man kann dazu eine Turingmaschine mit 2-Bändern konstruieren, die für jede Zeile des Registermaschinenprogramms ein Unterprogramm enthält. Die Unterprogramme werden dabei auf dem ersten Band ausgeführt, während das zweite Band die Inhalte der Register in Binärdarstellung enthält. Die Turingmaschine simuliert dann Schritt für Schritt das Verhalten der Registermaschine. Die genaue Konstruktion dieser Turingmaschine ist allerdings sehr technisch.

Theorem 2.6 sagt nicht nur, dass jede Registermaschine durch eine Turingmaschine simuliert werden kann, sondern auch, dass die Laufzeit der entsprechenden Turingmaschine nur polynomiell größer ist als die der Registermaschine. Insbesondere ist die Laufzeit der Turingmaschine polynomiell, wenn die Laufzeit der Registermaschine polynomiell ist. Um dies einzusehen, nehmen wir an, dass die Laufzeit $t(n)$ der

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

Tabelle 2.1: Syntax und Semantik der Befehle einer Registermaschine [6]

Registermaschine ein Polynom vom Grad $d \geq 1$ ist. Laut dem Theorem existiert ein Polynom q, dessen Grad wir mit $d^\star$ bezeichnen, für das die Rechenzeit der Turingmaschine durch $O(q(n+t(n))) = O(q(t(n))) = O(q(n^d)) = O(n^{dd^\star})$ beschränkt ist. Damit ist auch die Laufzeit der Turingmaschine durch ein Polynom beschränkt.

Wir werden später in dieser Vorlesung noch ausführlich über die Bedeutung von polynomieller Laufzeit sprechen. An dieser Stelle sollten die Leserinnen und Leser die Erkenntnis mitnehmen, dass jedes Problem, das von einer Registermaschine im logarithmischen Kostenmaß in polynomieller Zeit gelöst werden kann, auch von einer Turingmaschine in polynomieller Zeit gelöst werden kann.

Auch umgekehrt kann jede Turingmaschine mit polynomiellem Zeitverlust durch eine Registermaschine im logarithmischen Kostenmaß simuliert werden.

Theorem 2.7. *Jede Turingmaschine, deren Rechenzeit durch $t(n)$ beschränkt ist, kann durch eine im logarithmischen Kostenmaß $O((t(n) + n)\log(t(n) + n))$-zeitbeschränkte Registermaschine simuliert werden.*

Zusammen implizieren die Theoreme 2.6 und 2.7, dass die Klasse der von Turingmaschinen berechenbaren Funktionen und die Klasse der von Registermaschinen berechenbaren Funktionen übereinstimmen. Darüber hinaus sind sogar die Rechenzeiten bis auf polynomielle Faktoren vergleichbar. Etwas präziser formuliert besagen die Theoreme insbesondere, dass die Klasse der von Turingmaschinen in polynomieller Zeit berechenbaren Funktionen und die Klasse der von Registermaschinen in polynomieller Zeit berechenbaren Funktionen übereinstimmen.

2.2.3 Die Church-Turing-These

In Definition 2.2 haben wir festgelegt, dass eine Funktion berechenbar ist, wenn es eine Turingmaschine gibt, die zu jeder Eingabe in endlich vielen Schritten die durch die Funktion beschriebene Ausgabe liefert. Der Begriff der Berechenbarkeit ist demnach zunächst an das Modell der Turingmaschine gekoppelt. Dies ist uns aber nicht allgemein genug, denn haben wir für eine Funktion nachgewiesen, dass sie nicht berechenbar ist, so wäre es zunächst ja durchaus denkbar, dass sie zwar nicht von einer Turingmaschine, aber von einem Java- oder C++-Programm berechnet werden kann. Geht auch dies nicht, so kann man die Funktion vielleicht mithilfe anderer Hardware (Quantencomputer, Analogrechner, etc.) berechnen. Falls dies wirklich der Fall wäre, so wäre ein Begriff der Berechenbarkeit, der sich wie Definition 2.2 nur auf Turingmaschinen stützt, nicht besonders interessant.

Tatsächlich haben wir aber bereits in Theorem 2.6 gesehen, dass Registermaschinen nicht mächtiger sind als Turingmaschinen. Registermaschinen wiederum sind ein realistisches Modell für heutige reale Rechner (unabhängig von der benutzten Programmiersprache). Ist eine Funktion also auf einem realen Rechner berechenbar, so ist sie auch auf einer Registermaschine und damit auch auf einer Turingmaschine berechenbar. Andersherum formuliert, kann eine Funktion, die nicht von einer Turingmaschine berechnet werden kann, auch von keinem heutigen realen Rechner berechnet werden.

Darüber hinaus gibt es neben Turingmaschinen und Registermaschinen eine ganze Reihe von weiteren theoretischen Modellen, mit denen man versucht hat, den Begriff der Berechenbarkeit zu formalisieren (WHILE-Programme, μ-rekursive Funktionen etc.). Diese Modelle wurden unabhängig voneinander entworfen, es hat sich aber im Nachhinein herausgestellt, dass sie alle zu derselben Klasse von berechenbaren Funktionen führen. Dies sind starke Indizien für die Vermutung, dass alle Funktionen, die man auf irgendeine algorithmische Art berechnen kann, auch von Turingmaschinen berechnet werden können. Dies ist die sogenannte *Church-Turing-These*.

These 2.8 (Church-Turing-These)**.** *Alle „intuitiv berechenbaren“ Funktionen können von Turingmaschinen berechnet werden.*

Diese These ist prinzipiell nicht beweisbar, da der Begriff „intuitiv berechenbar“ nicht formal definiert ist. Eine formalere Variante dieser These, die zwar prinzipiell beweisbar, aber noch unbewiesen ist, lautet wie folgt.

These 2.9 (Physikalische Church-Turing-These)**.** *Die Gesetze der Physik erlauben es nicht, eine Maschine zu konstruieren, die eine Funktion berechnet, die nicht auch von einer Turingmaschine berechnet werden kann.*

Auch für diese These gibt es eine ganze Reihe von Indizien. Beispielsweise weiß man, dass weder mit Quantencomputern noch mit Analogrechnern Funktionen berechnet werden können, die man nicht auch mit einer Turingmaschine berechnen kann. Tatsächlich gibt es mittlerweile unter dem Stichwort *unconventional computing* ein ganzes Kuriositätenkabinett an physikalischen, biologischen und chemischen Modellen, mit denen Berechnungen durchgeführt werden können. Keines der bekannten Modelle widerspricht jedoch der oben formulierten physikalischen Church-Turing-These.

Wir haben in Abschnitt 2.2.2 gesehen, dass Turingmaschinen und Registermaschinen nicht nur dieselbe Klasse von Funktionen berechnen können, sondern dass sogar die Klassen der in polynomieller Zeit berechenbaren Funktionen in beiden Modellen übereinstimmen. Die *erweiterte Church-Turing-These* besagt, dass dies sogar für alle realistischen Maschinenmodelle der Fall ist, dass also die Klasse der in polynomieller Zeit berechenbaren Funktionen unabhängig vom Maschinenmodell ist. Auch diese These ist mathematisch nicht präzise, da nicht formal definiert ist, was ein realistisches Maschinenmodell ist. Während die Church-Turing-These heutzutage kaum angezweifelt wird, verhält es sich mit der erweiterten Church-Turing-These anders, was insbesondere an Quantencomputern liegt. Es ist nämlich bekannt, dass Zahlen mithilfe von Quantencomputern in polynomieller Zeit faktorisiert werden können. Man glaubt jedoch, dass dies mit herkömmlichen Rechnern (und mit Turing- und Registermaschinen) nicht möglich ist.

Kapitel 3

Berechenbarkeitstheorie

In diesem Kapitel weisen wir für das Halteproblem nach, dass es nicht entscheidbar ist. Der Beweis beruht auf der Existenz einer sogenannten *universellen Turingmaschine*. Dabei handelt es sich um eine Turingmaschine, die beliebige andere Turingmaschinen simulieren kann. Sie erhält als Eingabe die Codierung einer Turingmaschine M und ein Wort w und simuliert das Verhalten von M auf w.

Die Existenz einer solchen universellen Turingmaschine ist bei genauer Betrachtung nicht besonders schwierig nachzuweisen. Sie ist aber von zentraler Bedeutung, denn sie besagt, dass man bei Turingmaschinen genauso wie bei realen Rechnern die Hardware vom auszuführenden Programmcode trennen kann. Bei der universellen Turingmaschine handelt es sich um einen programmierbaren Rechner, der mit derselben Hardware beliebige Programme ausführen kann. Denkt man an heutige Rechner, so ist diese Eigenschaft natürlich eine Selbstverständlichkeit. Im Jahre 1936 als Alan Turing seine revolutionäre Arbeit schrieb und das Modell der Turingmaschine eingeführt hat, gab es allerdings nur festverdrahtete Rechner für bestimmte Aufgaben und die Von-Neumann-Architektur war noch unbekannt.

Ist erst mal für eine Sprache $A \subseteq \Sigma^*$ nachgewiesen, dass sie nicht entscheidbar ist, so kann die Unentscheidbarkeit weiterer Sprachen durch *Reduktionen* gezeigt werden. Sei beispielsweise $B \subseteq \Sigma^*$ eine Sprache, deren Unentscheidbarkeit wir nachweisen wollen. Unter einer Reduktion von A auf B verstehen wir eine Turingmaschine zur Lösung von A, die eine (hypothetische) Turingmaschine zur Lösung von B als Unterprogramm einsetzt. Existiert eine solche Reduktion und eine Turingmaschine, die B entscheidet, so kann folglich auch A entschieden werden. Ist A nicht entscheidbar, folgt aus der Reduktion mit einem Widerspruchsbeweis, dass auch B nicht entscheidbar sein kann. Wir nutzen dies aus, indem wir zunächst die Unentscheidbarkeit einer speziell für diesen Zweck konstruierten künstlichen Sprache nachweisen und anschließend mithilfe von Reduktionen nachweisen, dass das Halteproblem und andere interessante Probleme nicht entscheidbar sind.

Zum Schluss dieses Kapitels lernen wir noch eine Abstufung innerhalb der Klasse der nicht entscheidbaren Sprachen kennen. Wir nennen eine Sprache L *rekursiv aufzählbar* oder *semi-entscheidbar*, wenn es eine Turingmaschine gibt, die alle Eingaben $x \in L$

akzeptiert und alle Eingaben $x \notin L$ entweder verwirft oder nicht auf ihnen terminiert. Wir werden kurz diskutieren, warum diese Klasse interessant ist und welcher Zusammenhang zu den entscheidbaren Sprachen besteht.

Bevor wir uns formal mit der Berechenbarkeitstheorie beschäftigen, zeigen wir, dass es nicht nur für die Informatik sondern auch für die Mathematik weitreichende und unglaubwürdige Konsequenzen hätte, wenn das Halteproblem entscheidbar wäre. Dazu betrachten wir den folgenden an Java angelehnten Pseudocode.

```
void main() {
  int n = 4;
  while (true) {
    boolean foundPrimes = false;
    for (int p = 2; p < n; p++) {
      if (prime(p) && prime(n-p)) foundPrimes = true;
    }
    if (!foundPrimes) return;
    n = n + 2;
  }
}
```

Wir gehen davon aus, dass die Methode `prime` für eine gegebene Zahl entscheidet, ob sie eine Primzahl ist oder nicht. Weiterhin gehen wir davon aus, dass n und p beliebig große Werte annehmen können, ohne dass Überläufe auftreten. Dies kann in Java beispielsweise durch die Verwendung der Klasse `java.math.BigInteger` erreicht werden. Aus Gründen der Übersichtlichkeit haben wir aber auf die Verwendung dieser Klasse verzichtet. In der while-Schleife wird über alle geraden Zahlen $n = 4, 6, 8, \ldots$ größer als zwei iteriert, solange bis in einer Iteration die Abbruchbedingung greift. Man sieht leicht, dass das Programm genau dann terminiert, wenn eine Zahl n erreicht wird, die sich nicht als Summe zweier Primzahlen schreiben lässt. Zusammengefasst lässt sich also Folgendes sagen.

Beobachtung 3.1. *Das obige Programm terminiert genau dann, wenn es eine gerade Zahl größer als zwei gibt, die sich nicht als Summe zweier Primzahlen schreiben lässt.*

In der Vorlesung „Logik und diskrete Strukturen“ haben wir bereits die *Goldbachsche Vermutung* kennengelernt, die besagt, dass sich jede gerade Zahl größer als zwei als Summe zweier Primzahlen schreiben lässt. Diese Vermutung ist trotz intensiver Bemühungen zahlreicher Mathematiker bis heute unbewiesen. Hätten wir einen Compiler zur Hand, der uns sagen könnte, ob das obige Programm terminiert, so könnten wir damit einfach die Goldbachsche Vermutung entscheiden: Terminiert das Programm, so ist die Goldbachsche Vermutung falsch, ansonsten ist sie korrekt.

3.1 Entwurf einer universellen Turingmaschine

In diesem Abschnitt entwerfen wir eine universelle Turingmaschine. Wir betrachten der Einfachheit halber nur Turingmaschinen mit dem Eingabealphabet $\Sigma = \{0, 1\}$

und dem Bandalphabet $\Gamma = \{0, 1, \square\}$, was (wie oben bereits erwähnt) keine echte Einschränkung ist. Eine universelle Turingmaschine erhält als Eingabe eine Zeichenkette der Form $\langle M \rangle w$, wobei $\langle M \rangle \in \{0,1\}^*$ die Codierung einer Turingmaschine M und $w \in \{0,1\}^*$ eine beliebige Zeichenkette ist. Die universelle Turingmaschine simuliert bei dieser Eingabe das Verhalten der Turingmaschine M auf der Eingabe w. Das heißt, sie terminiert auf der Eingabe $\langle M \rangle w$ genau dann, wenn die Turingmaschine M auf der Eingabe w terminiert, und sie akzeptiert (verwirft) die Eingabe $\langle M \rangle w$ genau dann, wenn M die Eingabe w akzeptiert (verwirft).

Wir müssen zunächst festlegen, wie die Codierung $\langle M \rangle$ einer Turingmaschine M aussehen soll. Eine solche Codierung nennen wir auch Gödelnummerierung (nach dem Mathematiker Kurt Gödel) und es gibt eine Vielzahl an Möglichkeiten, wie sie realisiert werden kann.

Definition 3.2. *Eine injektive Abbildung der Menge aller Turingmaschinen in die Menge* $\{0,1\}^*$ *heißt* Gödelnummerierung*. Für eine feste Gödelnummerierung bezeichnen wir mit* $\langle M \rangle \in \{0,1\}^*$ *die* Gödelnummer *der Turingmaschine* M*, also die Zeichenkette, auf die* M *gemäß der Gödelnummerierung abgebildet wird.*

Eine Gödelnummerierung heißt präfixfrei*, wenn kein echtes Präfix (Anfangsstück) einer Gödelnummer* $\langle M \rangle$ *selbst eine gültige Gödelnummer ist.*

Die Präfixfreiheit garantiert, dass wir bei einer Zeichenkette der Form $\langle M \rangle w$ stets eindeutig entscheiden können, an welcher Stelle die Gödelnummer aufhört und das Wort w beginnt. Wir geben nun noch eine konkrete präfixfreie Gödelnummerierung an und bezeichnen mit $\mathcal{G}$ die Menge aller gültigen Gödelnummern in dieser Nummerierung. Sprechen wir im Folgenden von *der* Gödelnummer einer Turingmaschine, so bezieht sich das immer auf diese konkrete Gödelnummerierung. Die Präfixfreiheit folgt direkt aus der Tatsache, dass jede gültige Gödelnummer in unserer Nummerierung mit der Zeichenkette 111 endet und die Zeichenkette 111 an keiner anderen Stelle in einer Gödelnummer vorkommt.

Es sei eine Turingmaschine $M = (Q, \Sigma, \Gamma, \square, q_1, q_2, \delta)$ gegeben, deren Komponenten die folgende Form haben.

- Es sei $Q = \{q_1, q_2, \ldots, q_t\}$ für ein $t \geq 2$.
- Es sei $\Sigma = \{X_1, X_2\}$ und $\Gamma = \Sigma \cup \{X_3\}$ für $X_1 = 0$, $X_2 = 1$ und $X_3 = \square$.
- Es sei q_1 der Startzustand und q_2 der Endzustand.

Wir nummerieren die drei möglichen Kopfbewegungen und setzen $D_1 = L$, $D_2 = N$ und $D_3 = R$. Einen Übergang $\delta(q_i, X_j) = (q_k, X_\ell, D_m)$ codieren wir mit der Zeichenkette $0^i 10^j 10^k 10^\ell 10^m$. Die Zustandsüberführungsfunktion besteht aus $3(t-1)$ vielen solchen Übergängen (jeweils drei für jeden Zustand ungleich q_2), deren Codierungen wir mit $\text{code}(1), \ldots, \text{code}(3(t-1))$ bezeichnen. Die Gödelnummer von M ist dann

$$\langle M \rangle = 11\,\text{code}(1)\,11\,\text{code}(2)\,11\,\ldots\,11\,\text{code}(3(t-1))\,111.$$

Theorem 3.3. *Es existiert eine universelle Turingmaschine U, die auf jeder Eingabe der Form $\langle M\rangle w \in \{0,1\}^*$ das Verhalten der Turingmaschine M auf der Eingabe $w \in \{0,1\}^*$ simuliert. Die Rechenzeit von U auf der Eingabe $\langle M\rangle w$ ist nur um einen konstanten Faktor (das heißt in diesem Kontext, dass der Faktor nur von M, nicht aber von w abhängt) größer als die Rechenzeit von M auf der Eingabe w.*

Beweis. Wir konstruieren die universelle Turingmaschine U als 3-Band-Turingmaschine. Dies genügt zum Beweis des Theorems, da wir gemäß Theorem 2.5 jede 3-Band-Turingmaschine durch eine normale 1-Band-Turingmaschine simulieren können. Zu Beginn steht die Eingabe $\langle M\rangle w$ auf dem ersten Band. Zunächst schreibt U die Gödelnummer $\langle M\rangle$ auf das zweite Band und löscht sie vom ersten. Auf dem ersten Band steht dann nur noch das Wort w und der Kopf befindet sich auf dem ersten Zeichen von w. Auf das dritte Band schreibt U eine Codierung des Startzustandes. Wir codieren dabei den Zustand q_i als 0^i. Das bedeutet, zu Beginn befindet sich auf Band 3 die Zeichenkette 0, da q_1 der Startzustand ist. Für jede gegebene Turingmaschine M kann diese Initialisierung in konstanter Zeit (d. h. unabhängig von w) durchgeführt werden.

Die universelle Turingmaschine U simuliert nun Schritt für Schritt die Berechnung von M auf w. Dabei werden die drei Bänder stets die folgenden Inhalte enthalten.

- Band 1 enthält den Bandinhalt von M nach den bereits simulierten Schritten. Die Kopfposition auf Band 1 stimmt ebenfalls mit der von M überein.
- Band 2 enthält die Gödelnummer $\langle M\rangle$.
- Band 3 codiert den Zustand von M nach den bereits simulieren Schritten. Ist dies q_i, so enthält Band 3 die Zeichenkette 0^i, die links und rechts von Leerzeichen eingeschlossen ist.

Zur Simulation eines Schrittes von M sucht U auf Band 2 nach einem passenden Übergang. Das bedeutet, U sucht nach der Zeichenkette 110^i10^j1, wobei q_i der aktuell auf Band 3 codierte Zustand sei und X_j das Zeichen, das sich auf Band 1 an der Kopfposition befindet. Findet die universelle Turingmaschine eine entsprechende Zeichenfolge auf Band 2, so kann sie entsprechend der oben beschriebenen Codierung den Übergang $\delta(q_i, X_j) = (q_k, X_\ell, D_m)$ rekonstruieren. Sie kann dann dementsprechend das Zeichen an der Kopfposition auf Band 1 durch X_ℓ ersetzen, den Kopf von Band 1 gemäß D_m bewegen und den Inhalt von Band 3 durch 0^k ersetzen.

Die universelle Turingmaschine simuliert auf die oben beschriebene Art und Weise Schritt für Schritt das Verhalten der Turingmaschine M, solange bis der Endzustand q_2 erreicht wird. Da die Ausgabe bei einer Mehrbandmaschine auf dem ersten Band zu finden ist, stimmen die Ausgaben von U auf $\langle M\rangle w$ und von M auf w überein.

Die beschriebene universelle Turingmaschine benötigt für die Simulation eines Schrittes von M nur eine konstante Anzahl an Schritten (also eine Anzahl, die zwar abhängig von M nicht jedoch von w ist). Bei der Simulation der beschriebenen 3-Band-Turingmaschine durch eine normale 1-Band-Turingmaschine gemäß Theorem 2.5 handeln wir uns einen quadratischen Zeitverlust ein. Dieser kann vermieden werden, indem die universelle Turingmaschine von vornherein als 3-Spur-Turingmaschine konstruiert

wird, bei der die Spuren dieselben Aufgaben erfüllen wie die Bänder in der oben beschriebenen Konstruktion. Bei dieser Konstruktion muss darauf geachtet werden, dass die Inhalte der Spuren 2 und 3, die zu jedem Zeitpunkt nur eine konstante Länge haben, mit dem Kopf der Turingmaschine mitverschoben werden. Auf die Details dieser Konstruktion werden wir hier allerdings nicht eingehen. □

Im Folgenden werden wir nur noch selten explizit über universelle Turingmaschinen sprechen, dennoch spielt Theorem 3.3 für den Rest dieses Kapitels implizit eine wichtige Rolle. Wir werden nämlich in vielen Beweisen Turingmaschinen konstruieren, die andere Turingmaschinen simulieren. Oft ist die Gödelnummer $\langle M \rangle$ einer Turingmaschine M gegeben und wir konstruieren eine Turingmaschine M', die M auf einer bestimmten Eingabe simuliert und anschließend die Ausgabe von M weiterverarbeitet. Auch wenn wir es normalerweise nicht dazu sagen, ist eine solche Simulation nur aufgrund der Existenz einer universellen Turingmaschine möglich.

3.2 Die Unentscheidbarkeit des Halteproblems

Wir werden in diesem Abschnitt das *Halteproblem* formal definieren und nachweisen, dass es nicht entscheidbar ist. Dazu werden wir die Technik der *Diagonalisierung* einsetzen, mit der wir bereits in der Vorlesung „Logik und diskrete Strukturen“ gezeigt haben, dass die Menge der reellen Zahlen überabzählbar ist. Anstatt diese Technik direkt auf das Halteproblem anzuwenden, werden wir zunächst eine Hilfssprache (die *Diagonalsprache*) definieren, für die mithilfe von Diagonalisierung leicht nachgewiesen werden kann, dass sie nicht entscheidbar ist. Dann werden wir durch eine *Reduktion* zeigen, dass aus der Unentscheidbarkeit der Diagonalsprache auch die des Halteproblems folgt.

Für die folgenden Betrachtungen ist es nützlich, eine Ordnung auf den Wörtern aus der Menge $\{0,1\}^*$ zu definieren. Für zwei Wörter $x = x_1 \dots x_\ell \in \{0,1\}^\ell$ und $y = y_1 \dots y_\ell \in \{0,1\}^\ell$ derselben Länge ℓ heißt x *lexikographisch kleiner* als y, wenn für das erste Zeichen x_i bzw. y_i, in dem sich die beiden Wörter unterscheiden, $x_i = 0$ und $y_i = 1$ gilt. Anders ausgedrückt, ist x lexikographisch kleiner als y, wenn ein Index i existiert, für den $x_1 \dots x_{i-1} = y_1 \dots y_{i-1}$ sowie $x_i = 0$ und $y_i = 1$ gilt. Nun können wir die *kanonische Ordnung* auf der Menge $\{0,1\}^*$ definieren. In dieser Ordnung kommt ein Wort $x \in \{0,1\}^*$ vor einem Wort $y \in \{0,1\}^*$, wenn $|x| < |y|$ gilt oder wenn $|x| = |y|$ gilt und x lexikographisch kleiner als y ist. Die ersten Wörter in der kanonischen Ordnung von $\{0,1\}^*$ sehen demnach wie folgt aus:

$$\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, 0000, \dots$$

Wir bezeichnen im Rest dieses Abschnittes für $i \in \mathbb{N}$ mit w_i das Wort aus $\{0,1\}^*$, das in der kanonischen Ordnung an der i-ten Stelle steht. Es gilt also beispielsweise $w_1 = \varepsilon$ und $w_5 = 01$.

Die kanonische Ordnung überträgt sich auf natürliche Art und Weise auf Teilmengen von $\{0,1\}^*$. Insbesondere können die Gödelnummern aus $\mathcal{G}$ gemäß der kanonischen

Ordnung sortiert werden. Im Rest dieses Abschnittes bezeichne M_i für $i \in \mathbb{N}$ die Turingmaschine, deren Gödelnummer $\langle M_i \rangle$ in der kanonischen Ordnung der Menge $\mathcal{G}$ an der i-ten Stelle steht.

Für das weitere Vorgehen ist die genaue Definition der kanonischen Ordnung nicht von Bedeutung. Letztlich ist nur wichtig, dass wir mithilfe dieser Ordnung sowohl die Menge aller Wörter aus $\{0,1\}^*$ als auch die Menge aller Turingmaschinen in einer festen Reihenfolge nummeriert haben (formal haben wir diese Mengen bijektiv auf die Menge der natürlichen Zahlen abgebildet). Ferner sollten sich die Leserinnen und Leser als Übung überlegen, dass es möglich ist, für ein gegebenes $i \in \mathbb{N}$ das Wort w_i und die Gödelnummer der Turingmaschine M_i zu berechnen. Außerdem ist es möglich, für ein gegebenes Wort $w \in \{0,1\}^*$ den Index i mit $w_i = w$ zu berechnen und für eine gegebene Gödelnummer $\langle M \rangle \in \mathcal{G}$ den Index $i \in \mathbb{N}$ mit $M_i = M$.

Definition 3.4. *Die Sprache*

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

heißt Diagonalsprache. *Die Diagonalsprache D enthält das Wort w_i, das in der kanonischen Ordnung von $\{0,1\}^*$ an der i-ten Stelle steht, also genau dann, wenn es von der Turingmaschine M_i, deren Gödelnummer in der kanonischen Ordnung von $\mathcal{G}$ an der i-ten Stelle steht, nicht akzeptiert wird.*

Die folgende Tabelle zeigt, warum die Diagonalsprache ihren Namen trägt. Jede Spalte entspricht einem Wort aus $\{0,1\}^*$ und jede Zeile entspricht einer Turingmaschine. Dabei sind sowohl die Wörter aus $\{0,1\}^*$ als auch die Turingmaschinen gemäß der kanonischen Ordnung sortiert. In einer Zelle, die zu einer Turingmaschine M_j und einem Wort w_i gehört, steht genau dann eine Eins, wenn M_j das Wort w_i akzeptiert, und ansonsten eine Null. Eine Null bedeutet also entweder, dass M_j das Wort w_i verwirft oder dass M_j auf w_i nicht hält. Die konkreten Werte in der Tabelle sind fiktiv gewählt. Ein Wort w_i gehört genau dann zu der Diagonalsprache D, wenn auf der Diagonalen in der Zelle, die zu der Turingmaschine M_i und dem Wort w_i gehört, eine Null steht. In dem Beispiel gilt also $w_1, w_3 \in D$, aber $w_2, w_4, w_5 \notin D$.

	w_1	w_2	w_3	w_4	w_5	$\dots$
M_1	**0**	1	1	0	1	$\dots$
M_2	1	**1**	0	0	0	$\dots$
M_3	1	0	**0**	1	1	$\dots$
M_4	1	0	0	**1**	0	$\dots$
M_5	0	1	1	1	**1**	$\dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Theorem 3.5. *Die Diagonalsprache D ist nicht entscheidbar.*

Beweis. Wir führen einen Widerspruchsbeweis und gehen davon aus, dass D entscheidbar ist. Dann gibt es per Definition eine Turingmaschine M, die D entscheidet. Die Turingmaschine M terminiert auf jeder Eingabe, akzeptiert alle $x \in D$ und verwirft alle $x \in \{0,1\}^* \setminus D$. Da die Gödelnummer $\langle M \rangle$ von M zu der Menge $\mathcal{G}$ gehört, gibt

es einen Index $i \in \mathbb{N}$ mit $M = M_i$. Das bedeutet, die Turingmaschine, die die Diagonalsprache D entscheidet, kommt in der Aufzählung aller Turingmaschinen gemäß der kanonischen Ordnung an der i-ten Stelle.

Uns interessiert nun, wie sich die Turingmaschine $M = M_i$ auf dem Wort w_i verhält, das in der Aufzählung aller Wörter aus $\{0,1\}^*$ gemäß der kanonischen Ordnung an der i-ten Stelle steht. Dazu unterscheiden wir zwei Fälle.

- Gilt $w_i \in D$, so akzeptiert M_i das Wort w_i, da M_i die Diagonalsprache D entscheidet. Aus der Definition von D folgt dann aber direkt, dass $w_i \notin D$ gilt.
- Gilt $w_i \notin D$, so verwirft M_i das Wort w_i, da M_i die Diagonalsprache D entscheidet. Aus der Definition von D folgt dann aber direkt, dass $w_i \in D$ gilt.

Da wir in beiden Fällen einen Widerspruch erhalten, kann es im Widerspruch zu der Annahme keine Turingmaschine geben, die die Diagonalsprache entscheidet. □

Für den Beweis von Theorem 3.5 war die spezielle Konstruktion der Diagonalsprache entscheidend. Deshalb ist ein solcher direkter Beweis der Unentscheidbarkeit einer Sprache für andere (weniger künstliche) Sprachen schwierig. Aus diesem Grund werden wir alle weiteren Unentscheidbarkeitsresultate mithilfe von Reduktionen zeigen.

Definition 3.6. *Die folgende Sprache nennen wir* Halteproblem[1]*:*

$$H = \{\langle M \rangle w \mid M \text{ hält auf } w\} \subseteq \{0,1\}^*.$$

Theorem 3.7. *Das Halteproblem H ist nicht entscheidbar.*

Beweis. Wir reduzieren die Diagonalsprache auf das Halteproblem. Das bedeutet, wir führen einen Widerspruchsbeweis und gehen davon aus, dass das Halteproblem H entscheidbar ist. Dann gibt es eine Turingmaschine M_H, die das Halteproblem entscheidet. Mithilfe dieser Turingmaschine M_H konstruieren wir eine Turingmaschine M_D, die die Diagonalsprache entscheidet. Da es eine solche Turingmaschine laut Theorem 3.5 nicht geben kann, haben wir damit die Annahme, dass M_H existiert, zum Widerspruch geführt.

Die Turingmaschine M_D führt auf einer beliebigen Eingabe $w \in \{0,1\}^*$ die folgenden Schritte aus.

Verhalten von Turingmaschine M_D auf Eingabe w

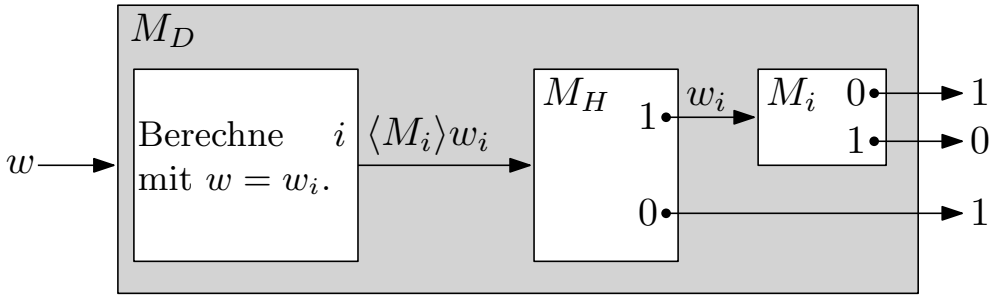
1. Berechne, welchen Index $i \in \mathbb{N}$ das Wort w in der kanonischen Ordnung von $\{0,1\}^*$ besitzt. Für diesen Index gilt per Definition $w_i = w$. Berechne die Gödelnummer $\langle M_i \rangle$, die in der kanonischen Ordnung von $\mathcal{G}$ an der i-ten Stelle steht.
2. Simuliere das Verhalten von M_H auf der Eingabe $\langle M_i \rangle w_i$.
3. **if** (M_H akzeptiert $\langle M_i \rangle w_i$ nicht)
4. akzeptiere w;

[1] Es wäre präziser, H als die Halte*sprache* zu bezeichnen. Der Name Halte*problem* ist aber geläufiger, weshalb wir ihn auch in dieser Vorlesung verwenden.

```
5   else
6       Simuliere das Verhalten von M_i auf der Eingabe w_i.
7       if (M_i akzeptiert w_i) verwirf w;
8       else akzeptiere w;
```

Wir haben bereits bei der Definition der kanonischen Ordnung angesprochen, dass die Berechnung des Index i und der Gödelnummer $\langle M_i \rangle$ in Schritt 1 möglich ist. Auch das Simulieren der Turingmaschine M_H in Schritt 2 ist kein Problem, da wir bereits in Abschnitt 3.1 bei der Diskussion der universellen Turingmaschine beschrieben haben, wie eine Turingmaschine eine andere simulieren kann. Da M_H das Halteproblem entscheidet, terminiert M_H insbesondere auf jeder Eingabe. Das bedeutet, dass auch die Simulation in Schritt 2 terminiert. Auch die Simulation von M_i auf der Eingabe w_i in Schritt 6 terminiert, da Schritt 6 nur dann erreicht wird, wenn M_H die Eingabe $\langle M_i \rangle w_i$ akzeptiert, d. h. wenn $\langle M_i \rangle w_i \in H$ gilt, was per Definition bedeutet, dass M_i auf der Eingabe w_i terminiert.

Die folgende Abbildung illustriert noch einmal das Verhalten der Turingmaschine M_D.



Wir haben oben bereits argumentiert, dass die Turingmaschine M_D auf jeder Eingabe terminiert. Es bleibt zu zeigen, dass sie die Eingabe w genau dann akzeptiert, wenn $w \in D$ gilt. Sei $i \in \mathbb{N}$ der Index, der in Schritt 1 berechnet wird. Dann gilt $w_i = w$.

- Gilt $w_i \in D$, so akzeptiert die Turingmaschine M_i die Eingabe w_i nicht (per Definition von D). In diesem Fall terminiert M_i auf der Eingabe w_i entweder nicht oder M_i terminiert und verwirft w_i.
  - Terminiert M_i nicht, so gilt $\langle M_i \rangle w_i \notin H$ (per Definition von H) und damit akzeptiert M_H die Eingabe $\langle M_i \rangle w_i$ nicht. Dann wird $w = w_i$ in Zeile 4 von M_D akzeptiert.
  - Terminiert M_i und verwirft w_i, so gilt $\langle M_i \rangle w_i \in H$ und damit wird Zeile 6 erreicht. Da M_i die Eingabe w_i verwirft, wird $w = w_i$ in Zeile 8 akzeptiert.
- Gilt $w_i \notin D$, so akzeptiert die Turingmaschine M_i die Eingabe w_i (per Definition von D). Das bedeutet insbesondere, dass M_i auf der Eingabe w_i terminiert. Es wird also Zeile 6 erreicht. Da M_i die Eingabe w_i akzeptiert, wird $w = w_i$ in Zeile 7 verworfen.

Damit ist der Beweis abgeschlossen, denn wir haben gezeigt, dass M_D jedes Wort $w \in D$ akzeptiert und jedes Wort $w \notin D$ verwirft. Da es gemäß Theorem 3.5 die Turingmaschine M_D nicht geben kann, kann die Annahme, dass eine Turingmaschine M_H für das Halteproblem H existiert, nicht gelten. □

3.3 Turing- und Many-One-Reduktionen

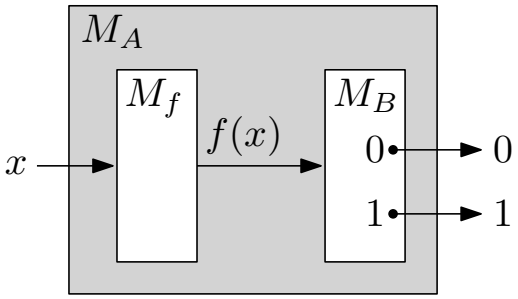
Wir haben in der Einleitung dieses Kapitels definiert, dass eine Reduktion einer Sprache A auf eine Sprache B eine Turingmaschine ist, die die Sprache A mithilfe eines (hypothetischen) Unterprogramms für die Sprache B löst. Diese Art der Reduktion, die wir auch im Beweis von Theorem 3.7 eingesetzt haben, wird in der Literatur auch als *Turing-Reduktion* oder *Unterprogrammtechnik* bezeichnet. Wir beschäftigen uns nun noch mit einer eingeschränkten Klasse von Turing-Reduktionen, den sogenannten *Many-One-Reduktionen*.

Definition 3.8. *Eine* Many-One-Reduktion[2] *einer Sprache $A \subseteq \Sigma_1^*$ auf eine Sprache $B \subseteq \Sigma_2^*$ ist eine berechenbare Funktion $f\colon \Sigma_1^* \to \Sigma_2^*$ mit der Eigenschaft, dass*

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

für alle $x \in \Sigma_1^*$ gilt. Existiert eine solche Reduktion, so heißt A auf B reduzierbar und wir schreiben $A \leq B$.

Eine Many-One-Reduktion f kann als Spezialfall einer Turing-Reduktion angesehen werden. Um die Sprache A mithilfe einer hypothetischen Turingmaschine M_B für die Sprache B zu entscheiden, genügt es, für eine gegebene Eingabe $x \in \Sigma_1^*$ den Funktionswert $f(x) \in \Sigma_2^*$ zu berechnen und dann die Turingmaschine M_B auf der Eingabe $f(x)$ zu simulieren. Es gilt $x \in A$ genau dann, wenn M_B die Eingabe $f(x)$ akzeptiert. Sei M_A die so konstruierte Turingmaschine für A und sei M_f eine Turingmaschine, die die Funktion f berechnet. Dann kann die Many-One-Reduktion schematisch wie folgt dargestellt werden.



Im folgenden Theorem halten wir die wesentlichen Konsequenzen einer Reduktion fest.

Theorem 3.9. *Es seien $A \subseteq \Sigma_1^*$ und $B \subseteq \Sigma_2^*$ zwei Sprachen, für die $A \leq B$ gilt. Ist B entscheidbar, so ist auch A entscheidbar. Ist A nicht entscheidbar, so ist auch B nicht entscheidbar.*

Beweis. Wir haben oben bereits beschrieben, dass man eine Turingmaschine M_A für die Sprache A konstruieren kann, wenn $A \leq B$ gilt und eine Turingmaschine M_B für B existiert. Damit ist der erste Teil des Theorems bewiesen.

Der zweite Teil folgt mit einem Widerspruchsbeweis direkt aus dem ersten, denn aus der Annahme, dass B entscheidbar ist, folgt mit $A \leq B$ direkt, dass auch A entscheidbar ist. □

[2]Wir werden den Begriff „Reduktion“ im Folgenden stets im Sinne von Many-One-Reduktion benutzen. Sprechen wir über Turing-Reduktionen, so werden wir dies explizit erwähnen.

Intuitiv besagt die Existenz einer Reduktion $A \leq B$ also, dass die Sprache A höchstens so schwer zu entscheiden ist wie die Sprache B. Wir definieren nun noch einige Abwandlungen des Halteproblems, deren Unentscheidbarkeit wir durch Reduktionen nachweisen werden.

Definition 3.10. *Das* spezielle Halteproblem H_ε *sei definiert durch*

$$H_\varepsilon = \{\langle M\rangle \mid M \text{ hält auf } \varepsilon\} \subseteq \{0,1\}^*.$$

Das vollständige Halteproblem H_{all} *sei definiert durch*

$$H_{all} = \{\langle M\rangle \mid M \text{ hält auf jeder Eingabe aus } \{0,1\}^*\} \subseteq \{0,1\}^*.$$

Die universelle Sprache U *sei definiert durch*

$$U = \{\langle M\rangle w \mid M \text{ akzeptiert } w\} \subseteq \{0,1\}^*.$$

Wir beschäftigen uns zunächst nur mit dem speziellen Halteproblem und der universellen Sprache. Auf das vollständige Halteproblem kommen wir erst in Abschnitt 3.5 zurück.

Theorem 3.11. *Die universelle Sprache U ist nicht entscheidbar.*

Beweis. Wir reduzieren das Halteproblem H auf die universelle Sprache. Dazu konstruieren wir eine Funktion $f\colon \{0,1\}^* \to \{0,1\}^*$, die Eingaben für das Halteproblem H auf Eingaben für die universelle Sprache U abbildet. Ist $x \in \{0,1\}^*$ nicht von der Form $\langle M\rangle w$ für eine Turingmaschine M (das heißt, x beginnt nicht mit einer gültigen Gödelnummer), so sei $f(x) = x$.

Gilt $x = \langle M\rangle w$ für eine Turingmaschine M und ein $w \in \{0,1\}^*$, so berechnen wir die Gödelnummer einer Turingmaschine $M^\star$ mit dem folgenden Verhalten. Die Turingmaschine $M^\star$ simuliert das Verhalten von M auf der gegebenen Eingabe Schritt für Schritt, solange bis M terminiert. Anschließend akzeptiert $M^\star$ die Eingabe unabhängig von der Ausgabe von M. Es sei dann $f(\langle M\rangle w) = \langle M^\star\rangle w$. Die Funktion f ist berechenbar, da die Turingmaschine $M^\star$ für gegebenes M leicht konstruiert werden kann.

Wir müssen nun noch zeigen, dass die oben definierte Funktion f tatsächlich eine Reduktion von H auf U darstellt. Wir zeigen die beiden zu der Äquivalenz $x \in H \iff f(x) \in U$ gehörenden Implikationen getrennt.

„⇒“ Sei $x \in H$. Dann muss $x = \langle M\rangle w$ für eine Turingmaschine M und ein Wort $w \in \{0,1\}^*$ gelten. Ferner folgt aus $x = \langle M\rangle w \in H$, dass die Turingmaschine M auf der Eingabe w hält. Da die Turingmaschine $M^\star$ das Verhalten von M simuliert, hält sie ebenfalls auf der Eingabe w. Per Definition akzeptiert $M^\star$ jede Eingabe, auf der sie hält. Dies bedeutet, dass $f(x) = \langle M^\star\rangle w \in U$ gilt.

„⇐“ Sei $x \notin H$. Dann ist x entweder nicht von der Form $\langle M \rangle w$ oder es gilt $x = \langle M \rangle w$, aber die Turingmaschine M hält nicht auf der Eingabe w. Ist x nicht von der Form $\langle M \rangle w$, so gilt $f(x) = x \notin U$. Gilt $x = \langle M \rangle w$ für eine Turingmaschine M, die auf der Eingabe w nicht hält, so hält die Turingmaschine $M^\star$ ebenfalls nicht auf der Eingabe w. Das bedeutet, dass sie die Eingabe w nicht akzeptiert, woraus $f(x) = \langle M^\star \rangle w \notin U$ folgt.

Damit haben wir gezeigt, dass $H \leq U$ gilt. Aus den Theoremen 3.7 und 3.9 folgt somit, dass U nicht entscheidbar ist. □

Der Beweis des nächsten Theorems ist sehr ähnlich zu dem gerade geführten Beweis für die Aussage, dass die universelle Sprache U nicht entscheidbar ist. Da Reduktionen aber ein sehr wichtiges Konzept in der theoretischen Informatik sind, kann es nicht schaden, ein weiteres Beispiel dafür zu sehen.

Theorem 3.12. *Das spezielle Halteproblem H_ε ist nicht entscheidbar.*

Beweis. Wir reduzieren das Halteproblem H auf das spezielle Halteproblem H_ε. Dazu konstruieren wir eine Funktion $f\colon \{0,1\}^* \to \{0,1\}^*$, die Eingaben für das Halteproblem H auf Eingaben für das spezielle Halteproblem H_ε abbildet. Ist $x \in \{0,1\}^*$ nicht von der Form $\langle M \rangle w$ für eine Turingmaschine M (das heißt, x beginnt nicht mit einer gültigen Gödelnummer), so sei $f(x) = x$.

Gilt $x = \langle M \rangle w$ für eine Turingmaschine M und ein $w \in \{0,1\}^*$, so berechnen wir die Gödelnummer einer Turingmaschine $M_w^\star$ mit dem folgenden Verhalten. Die Turingmaschine $M_w^\star$ löscht zunächst die Eingabe und ersetzt diese durch w. Anschließend simuliert sie das Verhalten von M auf der Eingabe w Schritt für Schritt. Es sei dann $f(\langle M \rangle w) = \langle M_w^\star \rangle$. Die Funktion f ist berechenbar, da die Turingmaschine $M_w^\star$ für gegebenes M und w leicht konstruiert werden kann.

Wir müssen nun noch zeigen, dass die oben definierte Funktion f tatsächlich eine Reduktion von H auf H_ε darstellt. Wir zeigen die beiden zu der Äquivalenz $x \in H \iff f(x) \in H_\varepsilon$ gehörenden Implikationen getrennt.

„⇒“ Sei $x \in H$. Dann muss $x = \langle M \rangle w$ für eine Turingmaschine M und ein Wort $w \in \{0,1\}^*$ gelten. Ferner folgt aus $x = \langle M \rangle w \in H$, dass die Turingmaschine M auf der Eingabe w hält. Da die Turingmaschine $M_w^\star$ bei jeder Eingabe das Verhalten von M auf w simuliert, hält sie auf jeder Eingabe (insbesondere auf der leeren Eingabe). Dies bedeutet, dass $f(x) = \langle M_w^\star \rangle \in H_\varepsilon$ gilt.

„⇐“ Sei $x \notin H$. Dann ist x entweder nicht von der Form $\langle M \rangle w$ oder es gilt $x = \langle M \rangle w$, aber die Turingmaschine M hält nicht auf der Eingabe w. Ist x nicht von der Form $\langle M \rangle w$, so beginnt x nicht mit der Gödelnummer einer Turingmaschine. Insbesondere ist x dann auch nicht von der Form $\langle M \rangle$ für eine Turingmaschine M. In diesem Fall gilt $f(x) = x \notin H_\varepsilon$. Gilt $x = \langle M \rangle w$ für eine Turingmaschine M, die auf der Eingabe w nicht hält, so hält die Turingmaschine $M_w^\star$ auf keiner Eingabe (insbesondere nicht auf der leeren Eingabe). Dies bedeutet, dass $f(x) = \langle M_w^\star \rangle \notin H_\varepsilon$ gilt.

Damit haben wir gezeigt, dass $H \leq H_\varepsilon$ gilt. Aus den Theoremen 3.7 und 3.9 folgt somit, dass H_ε nicht entscheidbar ist. □

Die Leserinnen und Leser sollten sich als Übung überlegen, dass die Reduktion von H auf H_ε im letzten Beweis auch eine Reduktion von H auf H_{all} ist und somit auch die Unentscheidbarkeit von H_{all} beweist.

Ein verbreiteter Fehler bei der Anwendung von Theorem 3.9 ist es, die Reduktion in die falsche Richtung durchzuführen. Möchte man für ein Problem L zeigen, dass es nicht entscheidbar ist, so muss man ein nicht entscheidbares Problem wie z. B. das Halteproblem auswählen und dieses auf L reduzieren und nicht umgekehrt.

3.4 Der Satz von Rice

Die Tatsache, dass das Halteproblem nicht entscheidbar ist, zeigt bereits, dass eine automatische Programmverifikation im Allgemeinen nicht möglich ist. In diesem Abschnitt zeigen wir sogar noch ein verschärftes Ergebnis: Für keine nicht triviale Menge von Funktionen kann entschieden werden, ob eine gegebene Turingmaschine eine Funktion aus dieser Menge berechnet. Dieses als *Satz von Rice*[3] bekannte Resultat werden wir nun formalisieren und beweisen.

Im Folgenden betrachten wir der Einfachheit halber wieder das Eingabealphabet $\Sigma = \{0, 1\}$. Es bezeichne

$$\mathcal{R} = \{f \colon \Sigma^* \to \Sigma^* \cup \{\bot\} \mid \exists \text{ Turingmaschine } M \text{ mit } f_M = f\}$$

die Menge aller von Turingmaschinen berechenbaren Funktionen. Für eine Teilmenge $S \subseteq R$ der berechenbaren Funktionen bezeichnen wir mit

$$L(S) = \{\langle M \rangle \mid f_M \in S\}$$

die Menge der Gödelnummern der Turingmaschinen, die eine Funktion aus der Menge S berechnen.

Theorem 3.13 (Satz von Rice)**.** *Es sei $S \subseteq \mathcal{R}$ mit $S \neq \emptyset$ und $S \neq \mathcal{R}$ eine Teilmenge der berechenbaren Funktionen. Dann ist die Sprache $L(S)$ nicht entscheidbar.*

Beweis. Sei $S \subseteq \mathcal{R}$ mit $S \neq \emptyset$ und $S \neq \mathcal{R}$ beliebig. Wir zeigen das Theorem mittels einer Turing-Reduktion des speziellen Halteproblems H_ε auf die Sprache $L(S)$. Dazu konstruieren wir eine Turingmaschine M_{H_ε}, die das spezielle Halteproblem H_ε mithilfe einer hypothetischen Turingmaschine $M_{L(S)}$ für die Sprache $L(S)$ als Unterprogramm entscheidet.

Es sei $u \colon \Sigma^* \to \Sigma^* \cup \{\bot\}$ die überall undefinierte Funktion mit $u(x) = \bot$ für alle $x \in \Sigma^*$. Die Funktion u gehört zu der Menge $\mathcal{R}$ der berechenbaren Funktionen, da sie von jeder

[3] Benannt nach dem Mathematiker Henry Gordon Rice, der dieses Ergebnis 1953 veröffentlichte.

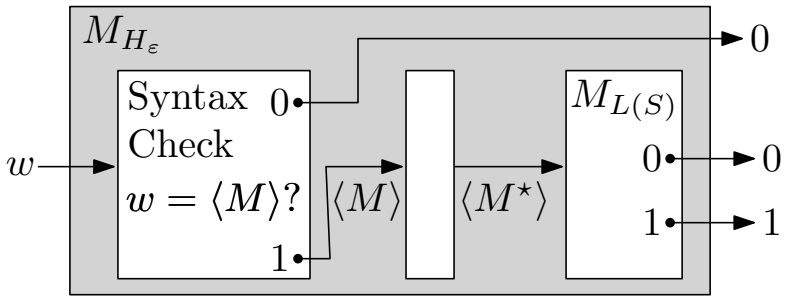
Turingmaschine, die auf keiner Eingabe hält, berechnet wird. Wir unterscheiden nun die beiden Fälle $u \in S$ und $u \notin S$.

Zunächst betrachten wir den Fall $u \notin S$. In diesem Fall sei $f \neq u$ eine beliebige Funktion aus S. Eine solche Funktion existiert, da S nicht leer ist. Sei M_f eine Turingmaschine, die f berechnet. Die Turingmaschine M_{H_ε} führt zunächst eine Überprüfung der Syntax der Eingabe w durch. Ist w keine gültige Gödelnummer, so gilt $w \notin H_\varepsilon$ und M_{H_ε} verwirft w direkt. Ansonsten sei $w = \langle M \rangle$ für eine Turingmaschine M. In diesem Fall konstruiert M_{H_ε} die Gödelnummer $\langle M^\star \rangle$ einer Turingmaschine $M^\star$ mit dem folgenden Verhalten.

> Erhält die Turingmaschine $M^\star$ eine Eingabe x, so simuliert sie in Phase 1 das Verhalten von M auf der leeren Eingabe ε. Anschließend simuliert $M^\star$ in Phase 2 die Turingmaschine M_f auf der Eingabe x und gibt $f(x)$ aus. Terminiert M_f auf der Eingabe x nicht, so terminiert auch $M^\star$ nicht auf x.

Die so konstruierte Turingmaschine $M^\star$ besitzt die folgende Eigenschaft: Hält M nicht auf der leeren Eingabe, so hält $M^\star$ auf keiner Eingabe und berechnet dementsprechend die überall undefinierte Funktion u. Hält M auf der leeren Eingabe, so erreicht $M^\star$ Phase 2 und berechnet die Funktion f.

Nachdem M_{H_ε} die Gödelnummer $\langle M^\star \rangle$ berechnet hat, simuliert sie das Verhalten von $M_{L(S)}$ auf der Eingabe $\langle M^\star \rangle$ und übernimmt die Ausgabe. Das Verhalten von M_{H_ε} ist schematisch in der folgenden Abbildung dargestellt.



Wir zeigen nun, dass die Turingmaschine M_{H_ε} das spezielle Halteproblem H_ε löst.

- Sei $w \in H_\varepsilon$. Dann muss $w = \langle M \rangle$ für eine Turingmaschine M gelten. Ferner folgt aus $w = \langle M \rangle \in H_\varepsilon$, dass die Turingmaschine M auf der leeren Eingabe ε hält. Dies bedeutet gemäß der obigen Beobachtung, dass die Turingmaschine $M^\star$ die Funktion f berechnet. Wegen $f \in S$ bedeutet dies, dass $\langle M^\star \rangle \in L(S)$ gilt. Dementsprechend akzeptiert die Turingmaschine $M_{L(S)}$ die Eingabe $\langle M^\star \rangle$. Damit akzeptiert M_{H_ε} die Eingabe w.

- Sei $w \notin H_\varepsilon$. Dann ist w entweder keine gültige Gödelnummer und wird direkt von M_{H_ε} verworfen oder es gilt $w = \langle M \rangle$ für eine Turingmaschine M, die nicht auf der leeren Eingabe ε hält. Letzteres bedeutet gemäß der obigen Beobachtung, dass die Turingmaschine $M^\star$ die Funktion u berechnet. Wegen $u \notin S$ bedeutet dies, dass $\langle M^\star \rangle \notin L(S)$ gilt. Dementsprechend verwirft die Turingmaschine $M_{L(S)}$ die Eingabe $\langle M^\star \rangle$. Damit verwirft M_{H_ε} die Eingabe w.

Damit haben wir für den Fall $u \notin S$ gezeigt, dass aus der Entscheidbarkeit von $L(S)$ die Entscheidbarkeit des speziellen Halteproblems H_ε im Widerspruch zu Theorem 3.12 folgt. Der Fall $u \in S$ kann ähnlich behandelt werden. In diesem Fall wählt man $f \in \mathcal{R} \setminus S$ beliebig und ändert lediglich im letzten Schritt das Akzeptanzverhalten von M_{H_ε} dahingehend, dass M_{H_ε} die Eingabe w genau dann akzeptiert, wenn $M_{L(S)}$ die Eingabe $\langle M^\star \rangle$ verwirft. Es ist eine gute Übung für die Leserin und den Leser, die Details dieses Falles noch einmal nachzuvollziehen. □

Der Satz von Rice hat weitreichende Konsequenzen für die Verifikation von Programmen. Er besagt, dass es unmöglich ist, automatisch zu verifizieren, dass ein gegebenes Programm ein bestimmtes Problem löst. Betrachten wir als Beispiel das Problem, zu testen, ob eine gegebene Zahl eine Primzahl ist. Dies können wir durch die Funktion $f\colon \{0,1\}^* \to \{0,1\}$ mit

$$f(x) = \begin{cases} 1 & \text{falls val}(x) \text{ eine Primzahl ist} \\ 0 & \text{sonst} \end{cases}$$

beschreiben. Wählen wir $S = \{f\}$, so besagt der Satz von Rice, dass nicht entschieden werden kann, ob eine gegebene Turingmaschine M die Funktion f berechnet. Bei Problemen, die durch Relationen beschrieben sind, müssen wir die Menge S anders definieren. Betrachten wir dazu noch einmal das Problem, eine gegebene Zahl in Binärdarstellung zu quadrieren. Dieses haben wir in Kapitel 2 durch die Relation

$$R = \{(x,y) \mid \text{val}(y) = \text{val}(x)^2\} \subseteq \{0,1\}^* \times \{0,1\}^*$$

modelliert. Wir definieren dann S als die Menge aller berechenbaren Funktionen, die bei jeder Eingabe $x \in \{0,1\}^*$ eine Ausgabe $y \in \{0,1\}^*$ mit $(x,y) \in R$ produzieren:

$$S = \{f \in \mathcal{R} \mid \forall x \in \{0,1\}^* : (x, f(x)) \in R\}.$$

Eine Turingmaschine löst das durch R beschriebene Problem genau dann, wenn sie eine Funktion aus S berechnet. Außerdem gilt $S \neq \emptyset$ und $S \neq \mathcal{R}$. Wieder besagt der Satz von Rice, dass nicht entschieden werden kann, ob eine gegebene Turingmaschine dieses Problem löst.

3.5 Rekursiv aufzählbare Sprachen

Eine Sprache L heißt gemäß Definition 2.3 entscheidbar oder rekursiv, wenn es eine Turingmaschine gibt, die auf jeder Eingabe hält und genau die Wörter aus der Sprache L akzeptiert. Bei vielen nicht entscheidbaren Problemen, die wir kennengelernt haben, besteht eine Asymmetrie zwischen Wörtern aus der Sprache und Wörtern, die nicht zu der Sprache gehören.

Um dies zu erläutern, konstruieren wir eine Turingmaschine M_H, die das Halteproblem zwar nicht löst, aber zumindest nie ein falsches Ergebnis ausgibt. Erhält M_H eine Eingabe $\langle M \rangle w$, so simuliert sie das Verhalten von M auf w. Terminiert diese Simulation

nach endlich vielen Schritten, so akzeptiert M_H anschließend die Eingabe $\langle M \rangle w$. Erhält M_H eine syntaktisch nicht korrekte Eingabe, so verwirft sie diese direkt. Die so definierte Turingmaschine M_H akzeptiert jede Eingabe $x \in H$. Jede Eingabe $x \notin H$ wird entweder verworfen (wenn sie syntaktisch nicht korrekt ist) oder M_H hält nicht auf x. Somit gibt M_H niemals ein falsches Ergebnis aus. Auf Eingaben, die zu H gehören, gibt sie sogar stets das richtige Ergebnis aus. Nur auf Eingaben, die nicht zu H gehören, terminiert sie nicht notwendigerweise.

Definition 3.14. *Eine Turingmaschine M* erkennt *eine Sprache $L \subseteq \Sigma^*$, wenn sie jedes Wort $w \in L$ akzeptiert und jedes Wort $w \in \Sigma^* \setminus L$ entweder verwirft oder darauf nicht terminiert. Eine Sprache $L \subseteq \Sigma^*$ heißt* semi-entscheidbar *oder* rekursiv aufzählbar*, wenn es eine Turingmaschine M gibt, die L erkennt.*

Aus der Definition folgt direkt, dass rekursive Sprachen auch rekursiv aufzählbar sind. Aber auch viele nicht rekursive Sprachen, die wir in dieser Vorlesung kennengelernt haben, sind rekursiv aufzählbar. Die Leserinnen und Leser sollten sich überlegen, dass dies insbesondere auf das Halteproblem H, das spezielle Halteproblem H_ε und die universelle Sprache U zutrifft. Wir haben allerdings auch Sprachen kennengelernt, bei denen auf den ersten Blick nicht klar ist, ob sie rekursiv aufzählbar sind. Ein Beispiel ist das vollständige Halteproblem H_{all}. Erhält man die Gödelnummer $\langle M \rangle$ einer Turingmaschine M, die auf jeder Eingabe hält, so existiert (anders als beim Halteproblem H, bei dem man eine gegebene Turingmaschine nur auf einer gegebenen Eingabe simulieren muss) kein offensichtliches Verfahren, um dies in endlicher Zeit festzustellen. Im Folgenden werden wir nachweisen, dass das vollständige Halteproblem H_{all} tatsächlich nicht rekursiv aufzählbar ist.

Zunächst werden wir uns aber noch mit den sogenannten *Abschlusseigenschaften* von rekursiv aufzählbaren Sprachen beschäftigen. Das heißt, wir werden untersuchen, bezüglich welcher Operationen (Schnitt, Vereinigung, Komplementbildung) die Menge der rekursiv aufzählbaren Sprachen abgeschlossen ist.

Theorem 3.15. *Es seien $L_1 \subseteq \Sigma^*$ und $L_2 \subseteq \Sigma^*$ zwei rekursiv aufzählbare Sprachen. Dann sind auch die Sprachen $L_1 \cup L_2$ und $L_1 \cap L_2$ rekursiv aufzählbar.*

Beweis. Es seien M_1 und M_2 Turingmaschinen, die die Sprachen L_1 bzw. L_2 erkennen. Gemäß Definition 3.14 bedeutet dies, dass die Turingmaschine M_i jedes Wort $x \in L_i$ akzeptiert und kein Wort $x \notin L_i$ akzeptiert.

Wir konstruieren zunächst eine Turingmaschine $M_\cap$ für den Schnitt $L_1 \cap L_2$. Die Turingmaschine $M_\cap$ simuliert bei einer Eingabe $x \in \Sigma^*$ zunächst die Turingmaschine M_1 auf x und anschließend (sofern die Simulation von M_1 auf x terminiert) die Turingmaschine M_2 auf x. Sie akzeptiert die Eingabe x genau dann, wenn sowohl M_1 als auch M_2 die Eingabe x akzeptiert haben.

Man überlegt sich leicht, dass die so konstruierte Turingmaschine $M_\cap$ den Schnitt $L_1 \cap L_2$ erkennt. Sie akzeptiert jedes Wort $x \in L_1 \cap L_2$, da jedes solche Wort auch von M_1 und M_2 akzeptiert wird. Sie akzeptiert kein Wort $x \notin L_1 \cap L_2$, da jedes solche Wort von mindestens einer der Turingmaschinen M_1 oder M_2 nicht akzeptiert wird.

Auf den ersten Blick erscheint es so, als könne man analog eine Turingmaschine $M_\cup$ konstruieren, die die Vereinigung $L_1 \cup L_2$ erkennt. Würde man allerdings erst M_1 und anschließend M_2 auf der Eingabe x simulieren und x akzeptieren, wenn mindestens eine der beiden Turingmaschinen x akzeptiert hat, so kann es Wörter $x \in L_1 \cup L_2$ geben, die nicht akzeptiert werden. Für Wörter $x \in L_1 \cup L_2$ mit $x \in L_2$ und $x \notin L_1$ kann es nämlich passieren, dass die Simulation von M_1 auf x nicht terminiert und die Simulation von M_2 gar nicht erreicht wird.

Die Lösung besteht darin, die Turingmaschinen M_1 und M_2 parallel zu simulieren. Konkret kann man $M_\cup$ als eine 2-Band-Turingmaschine konstruieren, die auf einem Band das Verhalten von M_1 und auf dem anderen das Verhalten von M_2 simuliert. Die Turingmaschine $M_\cup$ simuliert in jedem Rechenschritt parallel einen Schritt von M_1 und einen von M_2 auf dem entsprechenden Band. Sie akzeptiert die Eingabe, sobald eine von den beiden Simulationen terminiert und die Eingabe akzeptiert. Da jedes Wort $x \in L_1 \cup L_2$ nach endlich vielen Schritten von M_1 oder M_2 akzeptiert wird und kein Wort $x \notin L_1 \cup L_2$ von M_1 oder M_2 akzeptiert wird, erkennt die so konstruierte Turingmaschine die Vereinigung $L_1 \cup L_2$. □

Die Leserinnen und Leser mögen sich als kurze Übung überlegen, dass das obige Theorem auch analog für rekursive Sprachen gilt, dass also die Vereinigung und der Schnitt zweier rekursiver Sprachen wieder rekursiv sind. Das *Komplement* einer Sprache $L \subseteq \Sigma^*$ ist definiert als $\overline{L} = \Sigma^* \setminus L$. Eine weitere Übung ist es, zu zeigen, dass das Komplement jeder rekursiven Sprache ebenfalls rekursiv ist. Rekursiv aufzählbare Sprachen sind hingegen nicht abgeschlossen unter Komplementbildung, was aus dem folgenden Theorem folgt.

Theorem 3.16. *Sind eine Sprache $L \subseteq \Sigma^*$ und ihr Komplement $\overline{L}$ rekursiv aufzählbar, so ist L rekursiv.*

Beweis. Es seien M_L und $M_{\overline{L}}$ Turingmaschinen, die L bzw. $\overline{L}$ erkennen. Wir konstruieren nun eine Turingmaschine M, die L entscheidet. Die Turingmaschine M simuliert auf einer Eingabe x die Turingmaschinen M_L und $M_{\overline{L}}$ parallel (analog zu der Turingmaschine $M_\cup$ aus dem Beweis von Theorem 3.15). Sie stoppt, sobald eine der beiden Turingmaschinen M_L oder $M_{\overline{L}}$ die Eingabe x akzeptiert. Akzeptiert M_L die Eingabe x, so akzeptiert auch M die Eingabe x. Akzeptiert hingegen $M_{\overline{L}}$ die Eingabe x, so verwirft M die Eingabe x.

Um zu zeigen, dass M die Sprache L entscheidet, halten wir zunächst fest, dass jedes $x \in \Sigma^*$ entweder zu L oder zu $\overline{L}$ gehört und demnach von genau einer der beiden Turingmaschinen M_L oder $M_{\overline{L}}$ akzeptiert wird. Damit ist sichergestellt, dass die Turingmaschine M auf jeder Eingabe x terminiert. Da jede Eingabe $x \in L$ von M_L akzeptiert wird, akzeptiert M ebenfalls jede Eingabe aus L. Da jede Eingabe $x \notin L$ von $M_{\overline{L}}$ akzeptiert wird, verwirft M jede Eingabe, die nicht zu L gehört. Daraus folgt, dass M die Sprache L entscheidet. □

Das vorangegangene Theorem impliziert gemeinsam mit unserem Wissen über das Halteproblem, dass das Komplement einer rekursiv aufzählbaren Sprache im Allgemeinen

nicht rekursiv aufzählbar ist. Wir haben oben diskutiert, dass H rekursiv aufzählbar ist. Wäre das Komplement $\overline{H}$ ebenfalls rekursiv aufzählbar, so würde aus Theorem 3.16 folgen, dass H rekursiv ist, was im Widerspruch zu der Nichtentscheidbarkeit des Halteproblems (Theorem 3.7) steht.

Hat man bewiesen, dass eine Sprache nicht rekursiv aufzählbar ist, so kann man für weitere Sprachen wieder mittels Reduktionen nachweisen, dass sie ebenfalls nicht rekursiv aufzählbar sind. Das folgende Theorem lässt sich analog zu Theorem 3.9 beweisen.

Theorem 3.17. *Es seien $A \subseteq \Sigma_1^*$ und $B \subseteq \Sigma_2^*$ zwei Sprachen, für die $A \leq B$ gilt. Ist B rekursiv aufzählbar, so ist auch A rekursiv aufzählbar. Ist A nicht rekursiv aufzählbar, so ist auch B nicht rekursiv aufzählbar.*

Für die meisten Sprachen L, die wir in dieser Vorlesung kennengelernt haben, kann man relativ einfach zeigen, dass entweder L oder $\overline{L}$ rekursiv aufzählbar ist. Mithilfe von zwei Reduktionen zeigen wir nun, dass H_{all} nicht in diese Kategorie fällt.

Theorem 3.18. *Weder das vollständige Halteproblem H_{all} noch sein Komplement $\overline{H_{all}}$ sind rekursiv aufzählbar.*

Beweis. Wir haben nach dem Beweis von Theorem 3.12 angemerkt, dass die in diesem Beweis konstruierte Funktion auch eine Reduktion von H auf H_{all} darstellt. Die Leserinnen und Leser sollten sich überlegen, dass eine Reduktion einer Sprache A auf eine Sprache B auch stets eine Reduktion von $\overline{A}$ auf $\overline{B}$ ist. Dies folgt direkt aus Definition 3.8. Somit liefert der Beweis von Theorem 3.12 eine Reduktion von $\overline{H}$ auf $\overline{H_{\text{all}}}$. Da $\overline{H}$, wie oben diskutiert, nicht rekursiv aufzählbar ist, impliziert dies, dass auch $\overline{H_{\text{all}}}$ nicht rekursiv aufzählbar ist.

Zu zeigen ist also nur noch, dass auch H_{all} nicht rekursiv aufzählbar ist. Dazu geben wir eine Reduktion $f\colon \{0,1\}^* \to \{0,1\}^*$ von $\overline{H_\varepsilon}$ auf H_{all} an. Die Sprache $\overline{H_\varepsilon}$ ist nicht rekursiv aufzählbar, da ihr Komplement H_ε rekursiv aufzählbar aber nicht rekursiv ist. Eingaben x für $\overline{H_\varepsilon}$, die keine Gödelnummern sind, werden durch f auf eine beliebige Gödelnummer $f(x) = \langle M_{\text{all}} \rangle \in H_{\text{all}}$ abgebildet. Gilt $x = \langle M \rangle$ für eine Turingmaschine M, so sei $f(x) = \langle M^\star \rangle$ für eine Turingmaschine $M^\star$ mit dem folgenden Verhalten.

> Erhält die Turingmaschine $M^\star$ eine Eingabe w, so simuliert sie die Turingmaschine M auf der leeren Eingabe solange bis sie entweder hält oder $|w|$ viele Schritte gemacht hat. Falls M innerhalb dieser $|w|$ vielen simulierten Schritte hält, so geht $M^\star$ in eine Endlosschleife. Ansonsten terminiert $M^\star$.

Wir zeigen nun, dass $x \in \overline{H_\varepsilon} \iff f(x) \in H_{\text{all}}$ für alle $x \in \{0,1\}^*$ gilt.

- Sei $x \in \overline{H_\varepsilon}$. Dann ist x entweder keine gültige Gödelnummer oder es gilt $x = \langle M \rangle$ für eine Turingmaschine, die nicht auf der leeren Eingabe hält. Ist x keine gültige Gödelnummer, so gilt per Definition $f(x) = \langle M_{\text{all}} \rangle \in H_{\text{all}}$. Ist $x = \langle M \rangle \in \overline{H_\varepsilon}$, so folgt aus der obigen Konstruktion, dass die Turingmaschine $M^\star$ auf jeder Eingabe w terminiert. Damit gilt $f(x) = \langle M^\star \rangle \in H_{\text{all}}$.

- Sei $x \notin \overline{H_\varepsilon}$, also $x \in H_\varepsilon$. Dann gilt $x = \langle M \rangle$ für eine Turingmaschine M, die auf der leeren Eingabe hält. Es sei $t \in \mathbb{N}$ die Anzahl an Schritten, die M auf der leeren Eingabe benötigt. Aus der obigen Konstruktion folgt, dass die Turingmaschine $M^\star$ für jede Eingabe w mit $|w| > t$ in eine Endlosschleife gerät. Somit terminiert $M^\star$ nicht auf jeder Eingabe und es gilt $f(x) = \langle M^\star \rangle \notin H_{\text{all}}$.

Zusammengefasst haben wir in diesem Beweis die beiden Reduktionen $\overline{H} \leq \overline{H_{\text{all}}}$ und $\overline{H_\varepsilon} \leq H_{\text{all}}$ gezeigt. Da $\overline{H}$ und $\overline{H_\varepsilon}$ nicht rekursiv aufzählbar sind, ist somit weder H_{all} noch $\overline{H_{\text{all}}}$ rekursiv aufzählbar. □

Wir lernen nun noch eine alternative Charakterisierung von rekursiv aufzählbaren Sprachen kennen, die insbesondere die Namensgebung erklärt.

Definition 3.19. *Ein* Aufzähler *für eine Sprache $L \subseteq \Sigma^*$ ist eine Turingmaschine mit einem zusätzlichen Ausgabeband, das zu Beginn leer ist. Ein Aufzähler erhält keine Eingabe und er schreibt nach und nach Wörter aus L (durch Leerzeichen getrennt) auf das Ausgabeband. Er schreibt keine Wörter auf das Ausgabeband, die nicht zu L gehören, und zu jedem Wort $w \in L$ existiert ein Index $i_w \in \mathbb{N}$, sodass das Wort w nach i_w Schritten des Aufzählers auf dem Ausgabeband steht.*

Wir haben in der obigen Definition keinerlei Aussage darüber getroffen, in welcher Reihenfolge die Wörter aus L auf das Band geschrieben werden, und wir erlauben insbesondere, dass ein Wort aus L mehrfach auf das Ausgabeband geschrieben wird.

Theorem 3.20. *Eine Sprache L ist genau dann rekursiv aufzählbar, wenn ein Aufzähler für L existiert.*

Beweis. Mithilfe eines Aufzählers A für eine Sprache L kann leicht eine Turingmaschine M konstruiert werden, die L erkennt. Erhält diese Turingmaschine eine Eingabe w, so simuliert sie den Aufzähler A. Sie terminiert und akzeptiert die Eingabe w, sobald der Aufzähler das Wort w auf das Ausgabeband schreibt. Da jedes Wort $w \in L$ nach endlich vielen Schritten vom Aufzähler auf das Ausgabeband geschrieben wird, akzeptiert M jedes Wort aus L. Auf Wörtern $w \notin L$ terminiert M nicht, da diese vom Aufzähler nicht auf das Ausgabeband geschrieben werden. Somit erkennt M die Sprache L.

Sei nun andersherum eine Turingmaschine M gegeben, die die Sprache L erkennt. Mithilfe dieser Turingmaschine möchten wir nun einen Aufzähler A für die Sprache L konstruieren. Würde M die Sprache L entscheiden (also insbesondere auf jeder Eingabe terminieren), so könnte M einfach in einen Aufzähler umgebaut werden. Wir müssten dazu lediglich M nacheinander auf allen Wörtern aus Σ^* (zum Beispiel in kanonischer Reihenfolge) simulieren und genau die Wörter auf das Ausgabeband schreiben, die M akzeptiert. Da M die Sprache L aber nur erkennt (und somit auf Wörtern, die nicht zur Sprache gehören, im Allgemeinen nicht hält), schlägt dieser Ansatz fehl.

Stattdessen simulieren wir die Turingmaschine M in gewisser Weise für alle Wörter aus Σ^* parallel. Um dies zu präzisieren, sei wieder $w_1, w_2, w_3, \ldots$ die Aufzählung aller Wörter aus Σ^* in kanonischer Reihenfolge. Der Aufzähler A arbeitet wie folgt.

Aufzähler A für L

```
1  for i = 1, 2, 3, ...
2      Simuliere jeweils i Schritte von M auf den Eingaben w_1, ..., w_i.
3      Wird bei einer dieser Simulationen ein Wort w akzeptiert,
         so schreibe es auf das Ausgabeband.
```

Die so definierte Turingmaschine A ist ein Aufzähler für L. Sie schreibt nur Wörter auf das Ausgabeband, die von M akzeptiert werden. Somit schreibt sie keine Wörter, die nicht zu L gehören, auf das Ausgabeband. Für jedes Wort $w \in L$ gibt es ein t_w, sodass M die Eingabe w nach t_w vielen Schritten akzeptiert. Sei $w = w_j$. Dann gibt A das Wort $w \in L$ für $i = \max\{t_w, j\}$ (also nach endlich vielen Schritten) aus. □

Es ist essentiell, dass wir die Reihenfolge, in der die Wörter durch einen Aufzähler auf das Band geschrieben werden, nicht vorgeschrieben haben. Die Leserinnen und Leser sollten sich als Übung überlegen, dass die Existenz eines *kanonischen Aufzählers*, der die Wörter einer Sprache L in kanonischer Reihenfolge auf das Band schreibt, äquivalent zu der Entscheidbarkeit von L ist.

3.6 Weitere nicht entscheidbare Probleme

Bei allen nicht entscheidbaren Problemen, die wir bislang kennengelernt haben, tauchen Turingmaschinen und Gödelnummern explizit in der Definition auf. Zum Abschluss dieses Kapitels wollen wir noch kurz zwei weitere nicht entscheidbare Probleme vorstellen, die auf den ersten Blick nichts mit Turingmaschinen zu tun haben.

Der Mathematiker David Hilbert hat im Jahre 1900 eine Liste von 23 zentralen offenen Problemen der Mathematik präsentiert. Eines davon (*Hilberts zehntes Problem*) ist die Frage, mit welchem Algorithmus man feststellen kann, ob ein gegebenes Polynom eine ganzzahlige Nullstelle besitzt. Die Frage bezieht sich insbesondere auf multivariate Polynome, also Polynome mit mehreren Variablen. Ein solches Polynom ist die Summe von Monomen, wobei ein Monom das Produkt eines Koeffizienten mit Potenzen der Variablen ist. Sind also beispielsweise x, y und z die Variablen, so sind die Terme xy, x^2, $10xy^2$, $-2x^2y^3z$ sowie -7 Monome und $xy+x^2+10xy^2-2x^2y^3z-7$ ist ein Polynom.

Hilbert interessierte sich für einen Algorithmus, der entscheidet, ob es ganzzahlige Werte für die Variablen gibt, für die das Polynom den Wert 0 annimmt. 1970 hat Yuri Matiyasevich dieses Problem gelöst, allerdings anders als von Hilbert erwartet.

Theorem 3.21. *Hilberts zehntes Problem ist nicht entscheidbar.*

Den recht komplexen Beweis dieses Theorems werden wir in dieser Vorlesung nicht besprechen.

Ein anderes klassisches nicht entscheidbares Problem ist das *Postsche Korrespondenzproblem (PKP)*, das auf den ersten Blick an ein einfaches Puzzle erinnert. Eine

Eingabe für dieses Problem besteht aus einer endlichen Menge K von Paaren $K = \{(x_1, y_1), \ldots, (x_k, y_k)\}$ mit $x_i, y_i \in \Sigma^*$ für ein endliches Alphabet Σ. Anschaulich kann man sich ein Element (x_i, y_i) aus K als einen Dominostein vorstellen, der in der oberen Hälfte mit x_i und in der unteren Hälfte mit y_i beschriftet ist. Jedes Element aus K repräsentiert eine Klasse von Dominosteinen und wir gehen davon aus, dass von jeder Klasse beliebig viele Steine verfügbar sind.

Die Frage lautet nun, ob es möglich ist, Dominosteine auszuwählen und hintereinander zu legen, sodass oben und unten dieselbe Zeichenkette steht. Formaler ausgedrückt, soll entschieden werden, ob ein $n \geq 1$ und Indizes $i_1, \ldots, i_n \in \{1, \ldots, k\}$ existieren, sodass $x_{i_1} \ldots x_{i_n} = y_{i_1} \ldots y_{i_n}$ gilt.

Theorem 3.22. *Das Postsche Korrespondenzproblem ist nicht entscheidbar.*

Auch dieses Theorem werden wir aus Zeitgründen nicht in dieser Vorlesung beweisen. Sein Beweis ist allerdings deutlich einfacher als der von Theorem 3.21. Die wesentliche Idee besteht darin, dass man die Berechnung einer gegebenen Turingmaschine mithilfe geeignet gewählter Dominosteine simulieren kann. Insbesondere kann man die Dominosteine so wählen, dass es genau dann eine Lösung für das PKP gibt, wenn die gegebene Turingmaschine auf einer gegebenen Eingabe hält.

Kapitel 4

Komplexitätstheorie

In diesem Kapitel beschäftigen wir uns mit Problemen, bei denen einfach zu sehen ist, dass sie berechenbar sind. Für praktische Anwendungen ist es aber wichtig, Probleme nicht nur in endlicher Zeit, sondern auch möglichst effizient zu lösen, da man nicht beliebig lange auf die Ausgabe eines Algorithmus warten kann. Welche Laufzeit akzeptabel ist, hängt von der Anwendung ab. Während man die Ausgabe eines Navigationsgerätes innerhalb weniger Sekunden erwartet, akzeptiert man bei manchen großen Optimierungsproblemen in der Logistik vielleicht auch eine Laufzeit von mehreren Tagen.

Wir werden zunächst formal definieren, wann wir einen Algorithmus als *effizient* betrachten. Dabei interessiert uns in erster Linie, wie seine Laufzeit von der Eingabelänge abhängt. Es hat sich bewährt, Algorithmen als effizient zu betrachten, wenn ihre Laufzeit nur höchstens polynomiell mit der Eingabelänge wächst. Probleme, für die es solche Algorithmen gibt, gehören der *Komplexitätsklasse* P an.

Anschließend beschäftigen wir uns mit *nichtdeterministischen Turingmaschinen*. Dabei handelt es sich um Turingmaschinen, die nichtdeterministische Schritte machen dürfen (analog zu nichtdeterministischen endlichen Automaten, die die Leserin und der Leser sich noch einmal ins Gedächtnis rufen sollten). Wir definieren dann die Komplexitätsklasse NP als die Menge der Probleme, die von solchen nichtdeterministischen Turingmaschinen in polynomieller Zeit gelöst werden können. Genauso wie nichtdeterministische endliche Automaten sind nichtdeterministische Turingmaschinen nur ein Hilfsmittel für unsere theoretischen Betrachtungen und sie entsprechen (vermutlich) keinem physikalisch realisierbaren Rechner.

Aus der Definition folgt direkt, dass $\mathrm{P} \subseteq \mathrm{NP}$ gilt, es ist aber bis heute unklar, ob alle Probleme aus NP auf einer (deterministischen) Turingmaschine in polynomieller Zeit gelöst werden können, ob also $\mathrm{P} = \mathrm{NP}$ gilt. Man vermutet, dass dies nicht der Fall ist. Wir lernen insbesondere die Klasse der NP-*schweren* Probleme kennen. Ein NP-schweres Problem hat die Eigenschaft, dass die Existenz eines effizienten Algorithmus für dieses Problem implizieren würde, dass $\mathrm{P} = \mathrm{NP}$ gilt. Zeigt man für ein Problem also, dass es NP-schwer ist, so kann man dies als starkes Indiz dafür werten, dass es keinen effizienten Algorithmus für dieses Problem gibt.

4.1 Die Klassen P und NP

4.1.1 Die Klasse P

Um die Klasse P formal zu definieren, erinnern wir uns noch einmal an Definition 2.4. Dort wird die Laufzeit $t_M(n)$ einer Turingmaschine M auf Eingaben der Länge n als $t_M(n) = \max_{w \in \Sigma^n} t_M(w)$ definiert. Somit ist $t_M(n)$ die *Worst-Case-Laufzeit* von M auf Eingaben der Länge n.

Definition 4.1. *Ein Entscheidungsproblem L gehört genau dann zu der Komplexitätsklasse* P*, wenn es eine Turingmaschine M gibt, die L entscheidet, und eine Konstante $k \in \mathbb{N}$, für die $t_M(n) = O(n^k)$ gilt.*

Ein Entscheidungsproblem gehört also genau dann zu der Klasse P, wenn es einen Algorithmus für das Problem gibt, dessen Worst-Case-Laufzeit sich auf einer Turingmaschine polynomiell in der Eingabelänge beschränken lässt. Wir haben bereits in Abschnitt 2.2.2 diskutiert, dass die Klassen derjenigen Probleme, die von Turingmaschinen bzw. Registermaschinen im logarithmischen Kostenmaß in polynomieller Zeit gelöst werden können, übereinstimmen. Dementsprechend hätten wir in Definition 4.1 anstatt über Turingmaschinen auch über Registermaschinen im logarithmischen Kostenmaß sprechen können.

Die Klasse P wird gemeinhin als die Klasse der effizient lösbaren Probleme angesehen, und sprechen wir im Folgenden von einem *effizienten Algorithmus*, so meinen wir damit stets einen Algorithmus, dessen Laufzeit nur höchstens polynomiell mit der Eingabelänge wächst. Solche Algorithmen nennen wir auch *polynomielle Algorithmen*. Es bedarf sicherlich einer Diskussion, ob diese Festlegung des Effizienzbegriffes sinnvoll ist. Im Allgemeinen scheint sie zumindest fragwürdig, denn selbst ein Algorithmus mit einer linearen Laufzeit von $O(n)$ für Eingaben der Länge n kann in der Praxis zu langsam sein, wenn in der O-Notation sehr große Konstanten versteckt sind. Ebenso ist ein Algorithmus mit Laufzeit $\Theta(n^{100})$ für praktische Zwecke nicht zu gebrauchen, selbst wenn in der O-Notation keine großen Konstanten verborgen sind. Auf der anderen Seite ist ein Algorithmus mit einer exponentiellen Laufzeit von $O(1{,}00000001^n)$ gemäß der obigen Definition nicht effizient, obwohl er in der Praxis durchaus schnell sein kann. Ebenso gibt es Algorithmen mit einer exponentiellen Laufzeit von beispielsweise $\Omega(2^n)$, die in der Praxis dennoch auch für große Eingaben schnell sind, da dort oft gar keine Worst-Case-Eingaben auftreten, sondern Eingaben, die deutlich schneller gelöst werden können.

Wie bei der Diskussion, die wir im letzten Semester nach der Einführung der O-Notation geführt haben, müssen wir also auch hier zugeben, dass es Fälle gibt, in denen wir nicht das richtige Abstraktionsniveau getroffen haben. Dennoch hat es sich bewährt, Effizienz mit polynomieller Laufzeit gleichzusetzen, denn für fast alle interessanten Probleme, die in polynomieller Zeit gelöst werden können, gibt es effiziente Algorithmen, bei denen der Grad der polynomiellen Laufzeit klein ist. Das heißt, die meisten interessanten Probleme aus der Klasse P können auch tatsächlich in praktischen Anwendungen schnell gelöst werden. Man sollte aber in Erinnerung behalten,

dass es Fälle gibt, in denen es eine zu starke Vereinfachung ist, die effiziente Lösbarkeit eines Problems mit der Existenz eines polynomiellen Algorithmus gleichzusetzen.

Wir sollten ebenfalls die Einschränkung auf Entscheidungsprobleme in Definition 4.1 diskutieren. Zunächst besteht kein Grund für diese Einschränkung und man hätte P auch genauso gut als die Menge aller Probleme oder Funktionen definieren können, die in polynomieller Zeit gelöst bzw. berechnet werden können. Tatsächlich findet man solche abweichenden Definitionen auch in der Literatur. Die Einschränkung auf Entscheidungsprobleme bereitet schon auf die Definition der Klasse NP vor und sie ist nicht so gravierend, wie sie auf den erste Blick erscheinen mag. Um dies zu illustrieren betrachten wir drei Varianten des *Cliquenproblems*, das in dieser Vorlesung später noch mehrfach auftreten wird. Bei diesem Problem ist ein ungerichteter Graph $G = (V, E)$ gegeben. Eine *Clique* in einem solchen Graphen ist eine Teilmenge $V' \subseteq V$ der Knoten, für die es zwischen jedem Paar von verschiedenen Knoten $u \in V'$ und $v \in V'$ eine Kante in E gibt. Eine Clique V' mit $k = |V'|$ nennen wir auch *k-Clique von G*. Wir unterscheiden die folgenden Varianten des Cliquenproblems.

Varianten des Cliquenproblems

- *Optimierungsvariante*
  Eingabe: Graph $G = (V, E)$
  Aufgabe: Berechne eine Clique von G mit maximaler Kardinalität.

- *Wertvariante*
  Eingabe: Graph $G = (V, E)$
  Aufgabe: Berechne das größte $k^\star \in \mathbb{N}$, für das es eine $k^\star$-Clique in G gibt.

- *Entscheidungsvariante*
  Eingabe: Graph $G = (V, E)$ und ein Wert $k \in \mathbb{N}$
  Aufgabe: Entscheide, ob es in G eine Clique der Größe mindestens k gibt.

In den meisten Anwendungen, in denen das Cliquenproblem auftritt, sind wir an der Optimierungsvariante interessiert und wollen eine größtmögliche Clique finden. Wir zeigen nun aber, dass entweder alle drei Varianten in polynomieller Zeit gelöst werden können oder gar keine. Das bedeutet, es genügt, nur die Komplexität der Entscheidungsvariante zu untersuchen.

Theorem 4.2. *Entweder gibt es für alle drei Varianten des Cliquenproblems polynomielle Algorithmen oder für gar keine.*

Beweis. Mit einem polynomiellen Algorithmus für die Optimierungsvariante des Cliquenproblems können wir offensichtlich auch die Wertvariante in polynomieller Zeit lösen. Dazu müssen wir nur die Kardinalität der gefundenen größtmöglichen Clique ausgeben. Ebenso können wir mit einem polynomiellen Algorithmus für die Wertvariante auch die Entscheidungsvariante in polynomieller Zeit lösen, indem wir einfach die berechnete maximale Cliquengröße $k^\star$ mit der übergebenen Zahl k vergleichen. Nicht trivial sind lediglich die Rückrichtungen.

Zunächst zeigen wir, wie wir aus einem polynomiellen Algorithmus A für die Entscheidungsvariante des Cliquenproblems einen polynomiellen Algorithmus für die Wertvariante konstruieren können. Sei dazu G ein Graph mit N Knoten, für den wir die Größe $k^\star$ der maximalen Clique bestimmen wollen. Die genaue Eingabegröße hängt von der Codierung von G ab. Wir gehen der Einfachheit halber davon aus, dass G als Adjazenzmatrix gegeben ist und für die Eingabelänge dementsprechend $n = N^2$ gilt. Sei $A(G,k) \in \{0,1\}$ die Ausgabe von Algorithmus A bei Eingabe (G,k). Dabei bedeute $A(G,k) = 1$, dass es in dem Graphen G eine Clique der Größe mindestens k gibt. Um die Wertvariante des Cliquenproblems für den gegebenen Graphen G zu lösen, berechnen wir $k^\star = \max\{k \in \{1,\ldots,N\} \mid A(G,k) = 1\}$ mithilfe von maximal N Aufrufen des Algorithmus A. Das Maximum $k^\star$ ist per Definition der Wert, den wir suchen. Da A ein polynomieller Algorithmus ist, gibt es eine Konstante $\alpha \in \mathbb{N}$, für die wir die Laufzeit zur Berechnung von $k^\star$ mit $O(N \cdot (n')^\alpha)$ abschätzen können, wobei n' die Eingabegröße von (G,k) ist. Es gilt $n' \leq N^2 + \lceil \log_2(N) \rceil = O(N^2)$, da G mit N^2 Bits und $k \in \{1,\ldots,N\}$ mit $\lceil \log_2(N) \rceil$ Bits codiert werden kann. Dementsprechend beträgt die Laufzeit, die wir zur Berechnung von $k^\star$ benötigen, $O(N \cdot (N^2)^\alpha) = O(n^{\alpha+1})$. Diese Laufzeit ist, wie gewünscht, polynomiell in der Eingabelänge n.

Nun fehlt nur noch die Umwandlung eines polynomiellen Algorithmus für die Wertvariante in einen polynomiellen Algorithmus für die Optimierungsvariante. Sei dazu G ein Graph mit N Knoten, für den wir eine Clique mit maximaler Kardinalität bestimmen wollen. Sei nun A ein Algorithmus, der die Wertvariante des Cliquenproblems in polynomieller Zeit löst. Wir bezeichnen mit $A(G)$ die Ausgabe von Algorithmus A bei Eingabe G und konstruieren einen Algorithmus A_{opt}, der mithilfe von A die Optimierungsvariante löst. Dieser Algorithmus ist im folgenden Pseudocode dargestellt und er benutzt das Konzept eines *induzierten Teilgraphen*. Für einen Graphen $G = (V,E)$ ist der durch $V' \subseteq V$ induzierte Teilgraph der Graph $G' = (V',E')$ mit $E' = \{(u,v) \in E \mid u,v \in V'\}$. Es ist also der Teilgraph von G mit Knotenmenge V', der genau die Kanten aus E enthält, die innerhalb der Menge V' verlaufen.

$A_{\text{opt}}(G)$

```
1  k* = A(G);
2  V' := V = {v_1, ..., v_N};
3  for (i = 1; i ≤ N; i++)
4      G' sei induzierter Teilgraph von G mit Knotenmenge V' \ {v_i}.
5      if (A(G') == k*) V' := V' \ {v_i};
6  return V';
```

Man beweist leicht per Induktion die Invariante, dass es zu jedem Zeitpunkt eine $k^\star$-Clique gibt, die komplett in der aktuellen Knotenmenge V' enthalten ist. Ferner kann man leicht argumentieren, dass ein Knoten, der in Zeile 5 nicht aus der Menge V' entfernt wird, in jeder $k^\star$-Clique $V^\star \subseteq V'$ enthalten sein muss. Daraus folgt, dass die Menge V' am Ende genau aus einer $k^\star$-Clique besteht.

Die Laufzeit von A_{opt} ist durch $O(N \cdot (N^2)^\alpha)$ nach oben beschränkt, wenn die Laufzeit von A auf Graphen mit N Knoten durch $O((N^2)^\alpha)$ nach oben beschränkt ist. Somit ist die Laufzeit von A_{opt} polynomiell in der Eingabelänge $n = N^2$ beschränkt. □

Wir haben in dem obigen Beweis relativ genau die Anzahl an Bits angegeben, die benötigt werden, um eine Eingabe zu codieren. Man sollte sich aber klar machen, dass das vorangegangene Theorem robust gegenüber Änderungen der Codierung ist. Es gilt also beispielsweise auch dann, wenn die Graphen als Adjazenzlisten codiert sind oder wenn die Zahlen nicht binär sondern dezimal codiert sind. Für die allermeisten Optimierungsprobleme, die wir bislang kennengelernt haben, kann man analog drei verschiedene Varianten definieren (man denke zum Beispiel an das Kürzeste-Wege-Problem, an Flussprobleme oder an das Rucksackproblem). Genauso wie beim Cliquenproblem kann man für all diese Probleme nachweisen, dass die drei Varianten im Sinne polynomieller Berechenbarkeit äquivalent sind.

Nebenbei sei angemerkt, dass wir Theorem 4.2 mittels zweier spezieller Turing-Reduktionen bewiesen haben. Wir haben erst gezeigt, wie die Wertvariante auf die Entscheidungsvariante reduziert werden kann und anschließend wie die Optimierungsvariante auf die Wertvariante reduziert werden kann. Die wesentliche Eigenschaft dieser Reduktionen ist, dass sie nicht nur berechenbar sind, sondern auch in polynomieller Zeit berechnet werden können. Solche sogenannten *polynomiellen Reduktionen* werden wir später noch ausführlich besprechen.

Aus den oben diskutierten Gründen konzentrieren wir uns im Folgenden nur noch auf Entscheidungsprobleme und wir betrachten zum Abschluss dieses Abschnittes noch ein paar Beispiele.

Beispiele

- Wir haben uns im letzten Semester mit dem Problem beschäftigt, für einen Graphen zu testen, ob er zusammenhängend ist oder nicht. Mithilfe einer Tiefensuche kann man dies für Graphen mit n Knoten in Zeit $O(n^2)$ entscheiden, wenn der Graph als Adjazenzmatrix gegeben ist. Diese Laufzeit bezieht sich allerdings auf das uniforme Kostenmaß. Im logarithmischen Kostenmaß beträgt die Laufzeit $O(n^2 \log n)$, da wir $O(\log n)$ Bits benötigen, um den Index eines Knotens zu codieren. Diese Laufzeit ist polynomiell in der Eingabelänge n^2 beschränkt und damit gehört das Zusammenhangsproblem zu P. Generell trifft dies auf alle Graphenprobleme zu, die mit Algorithmen gelöst werden können, deren Laufzeit polynomiell in der Anzahl der Knoten und Kanten beschränkt ist.

- Ein weiteres Problem aus dem letzten Semester ist das Problem, einen minimalen Spannbaum zu berechnen. Bei diesem Problem besteht die Eingabe aus einem ungerichteten Graphen $G = (V, E)$ mit Kantengewichten $w\colon E \to \mathbb{N}$. Um dieses Problem in ein Entscheidungsproblem zu transformieren, erweitern wir die Eingabe um einen Wert $z \in \mathbb{N}$ und stellen die Frage, ob G einen Spannbaum mit Gewicht höchstens z besitzt. Die Leserinnen und Leser sollten sich überlegen, dass die Optimierungsvariante des Spannbaumproblems genau dann in polynomieller Zeit gelöst werden kann, wenn dies für die Entscheidungsvariante der Fall ist.

  Für zusammenhängende Graphen mit n Knoten und $m \geq n - 1$ Kanten löst der Algorithmus von Kruskal das Spannbaumproblem in Zeit $O(m \log m)$

im uniformen Kostenmaß, wenn der Graph als Adjazenzliste gegeben ist. Im logarithmischen Kostenmaß ist die Abschätzung der Laufzeit aufwendiger. Zunächst müssen die Kanten gemäß ihrem Gewicht sortiert werden. Benutzt man dazu Mergesort, so beträgt die Laufzeit für das Sortieren $O(m \log(m) \cdot \log(W))$ für $W = \max_{e \in E} w(e)$, da die Laufzeit durch die $O(m \log m)$ wesentlichen Vergleiche dominiert wird, die im logarithmischen Kostenmaß jeweils eine Laufzeit von $O(\log W)$ benötigen. Die Kosten für die restlichen Schritte des Algorithmus können wir im uniformen Kostenmaß durch $O(m \log m)$ und im logarithmischen Kostenmaß durch $O(m \log(m) \cdot \log(n))$ beschränken. Der zusätzliche Faktor resultiert wieder aus der Tatsache, dass wir $O(\log n)$ Bits benötigen, um den Index eines Knotens zu speichern. Insgesamt ergibt sich eine Laufzeit von $O(m \log m \cdot \max\{\log(W), \log(n)\})$. Die Eingabelänge beträgt $\Omega(m \log(n) + \log(W))$, da wir zur Codierung einer Kante $\Omega(\log(n))$ viele Bits benötigen und $\Omega(\log(W))$ Bits zur Codierung des maximalen Gewichtes W. Es gilt

$$m \log m \cdot \max\{\log(W), \log(n)\} = O((m \log(n) + \log(W))^2)$$

und somit ist die Laufzeit des Algorithmus von Kruskal polynomiell in der Eingabelänge beschränkt.

- Man kann die Entscheidungsvariante des Cliquenproblems lösen, indem man für einen gegebenen Graphen $G = (V, E)$ und eine gegebene Zahl $k \in \mathbb{N}$ alle Teilmengen von V der Größe k darauf testet, ob sie eine Clique bilden. Da es $\binom{n}{k} \geq (n/k)^k$ solcher Teilmengen gibt, besitzt dieser Algorithmus eine Laufzeit von $\Omega((n/k)^k)$, was sich nicht polynomiell in der Eingabelänge $O(n^2 + \log k)$ beschränken lässt. Damit ist natürlich nicht bewiesen, dass die Entscheidungsvariante des Cliquenproblems nicht zu der Klasse P gehört, sondern nur, dass der obige einfache Algorithmus keine polynomielle Laufzeit besitzt. Tatsächlich werden wir sehen, dass das Cliquenproblem NP-schwer ist. Es ist insofern bis heute unklar, ob das Cliquenproblem in polynomieller Zeit gelöst werden kann.

- Wir betrachten als letztes Beispiel die Entscheidungsvariante des Rucksackproblems. Sei dazu eine Eingabe mit Nutzenwerten $p_1, \ldots, p_N \in \mathbb{N}$, Gewichten $w_1, \ldots, w_N \in \mathbb{N}$ und einer Kapazität $t \in \mathbb{N}$ gegeben. Um ein Entscheidungsproblem zu erhalten, erweitern wir diese Eingabe um einen Wert $z \in \mathbb{N}$ und fragen, ob es eine Teilmenge $I \subseteq \{1, \ldots, N\}$ der Objekte mit $\sum_{i \in I} w_i \leq t$ und $\sum_{i \in I} p_i \geq z$ gibt. Wieder sollten sich die Leserinnen und Leser als Übung überlegen, dass die Optimierungsvariante des Rucksackproblems genau dann in polynomieller Zeit gelöst werden kann, wenn dies für die Entscheidungsvariante der Fall ist. Im vergangenen Semester haben wir die Optimierungsvariante des Rucksackproblems mithilfe dynamischer Programmierung in Zeit $O(N^2 W)$ für $W = \max_i w_i$ gelöst. Dementsprechend kann auch die Entscheidungsvariante in dieser Zeit gelöst werden. Auch diese Laufzeit be-

zog sich auf das uniforme Kostenmaß. Im logarithmischen Kostenmaß ergibt sich eine Laufzeit von $O(N^2 W \log P)$ für $P = \sum_i p_i$.

Ist dies eine polynomielle Laufzeit? Um diese Frage zu beantworten, müssen wir uns zunächst überlegen, wie die Größen N, W und P zu der Eingabelänge in Beziehung stehen. Typischerweise werden Zahlen in der Eingabe binär codiert. Betrachten wir den Spezialfall, dass alle Nutzenwerte p_i, alle Gewichte w_i und die Kapazität t Zahlen sind, die binär mit jeweils $N \in \mathbb{N}$ Bits codiert werden können. Dann kann W Werte bis 2^N-1 annehmen, P kann Werte bis $N(2^N-1)$ annehmen und die Eingabelänge beträgt $n = (2N+1)N$. Dieser Spezialfall zeigt bereits, dass die Laufzeit $O(N^2 W \log P) = O(N^3(2^N-1))$ im Allgemeinen nicht polynomiell in der Eingabelänge beschränkt ist. Wir werden sehen, dass auch das Rucksackproblem NP-schwer ist.

4.1.2 Die Klasse NP

Um die Komplexitätsklasse NP zu definieren, formalisieren wir zunächst den Begriff einer nichtdeterministischen Turingmaschine.

Definition 4.3. *Bei einer* nichtdeterministischen Turingmaschine (NTM) *handelt es sich um ein 7-Tupel* $(Q, \Sigma, \Gamma, \square, q_0, \bar{q}, \delta)$*, bei dem alle Komponenten bis auf* δ *identisch zu denen einer (deterministischen) Turingmaschine aus Definition 2.1 sind. Bei* δ *handelt es sich nun nicht mehr um eine Zustandsüberführungsfunktion, sondern um eine* Zustandsüberführungsrelation

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

Das Verhalten einer deterministischen Turingmaschine gemäß Definition 2.1 ist durch die Funktion δ eindeutig bestimmt. Liest die Turingmaschine in einem Zustand $q \in Q \setminus \{\bar{q}\}$ ein Zeichen $a \in \Gamma$, so gibt $\delta(q, a) = (q', a', D)$ mit $q' \in Q$, $a' \in \Gamma$ und $D \in \{L, N, R\}$ das Verhalten vor: die Turingmaschine ersetzt das Zeichen an der aktuellen Kopfposition durch a', wechselt in den Zustand q' und bewegt den Kopf gemäß D.

Das Verhalten einer nichtdeterministischen Turingmaschine M ist im Allgemeinen nicht eindeutig bestimmt. Für einen Zustand $q \in Q \setminus \{\bar{q}\}$ und ein Zeichen $a \in \Gamma$ kann es mehrere Tripel $(q', a', D) \in Q \times \Gamma \times \{L, N, R\}$ geben, für die $((q, a), (q', a', D)) \in \delta$ gilt. Die nichtdeterministische Turingmaschine kann dann eines davon wählen und den entsprechenden Schritt ausführen. Für eine gegebene Konfiguration gibt es bei einer nichtdeterministischen Turingmaschine also im Allgemeinen mehrere erlaubte Nachfolgekonfigurationen.

Es kann auch passieren, dass es für ein Paar (q, a) gar kein Tripel (q', a', D) gibt, für das $((q, a), (q', a', D)) \in \delta$ gilt. Dann gibt es für die aktuelle Konfiguration keine gültige Nachfolgekonfiguration. Dies definieren wir als gleichbedeutend damit, dass die Turingmaschine die Eingabe verwirft. Alternativ könnten wir auch für jedes solche Paar (q, a) das Tupel $((q, a), (\bar{q}, 0, N))$ zur Relation δ hinzufügen.

Da die Rechenschritte einer nichtdeterministischen Turingmaschine nicht eindeutig sind, kann es für eine Eingabe im Allgemeinen verschiedene mögliche Ausgaben geben. Insbesondere kann es bei einem Entscheidungsproblem Eingaben geben, für die es sowohl einen Rechenweg zu einer akzeptierenden Endkonfiguration (d. h. zu einer Konfiguration, bei der unter dem Kopf das Zeichen 1 steht) als auch einen Rechenweg zu einer verwerfenden Endkonfiguration gibt. Dabei ist ein *Rechenweg* eine Folge von gültigen Rechenschritten der Turingmaschine ausgehend von der Startkonfiguration. Wir erinnern uns daran, dass dieselbe Problematik schon bei nichtdeterministischen Automaten aufgetreten ist. Dort haben wir gesagt, dass ein nichtdeterministischer Automat ein Wort akzeptiert, wenn es mindestens einen Rechenweg gibt, der zu einem akzeptierenden Endzustand führt. Genau diese Definition übernehmen wir auch für Turingmaschinen.

Definition 4.4. *Eine nichtdeterministische Turingmaschine M akzeptiert eine Eingabe $w \in \Sigma^*$, wenn es mindestens einen Rechenweg von M gibt, der bei Eingabe w zu einer akzeptierenden Endkonfiguration führt (d. h. zu einer Konfiguration mit Zustand $\bar{q}$, bei der unter dem Kopf das Zeichen* 1 *steht). Wieder definieren wir $L(M) \subseteq \Sigma^*$ als die Menge der von M akzeptierten Eingaben und wir sagen, dass M die Sprache $L(M)$* entscheidet*, wenn sie für jede Eingabe auf jedem Rechenweg hält.*

Genauso wie die Ausgabe von den nichtdeterministischen Entscheidungen der Turingmaschine abhängen kann, kann auch die Laufzeit stark davon abhängen. Wir definieren die Laufzeit basierend auf dem längsten Rechenweg.

Definition 4.5. *Die Laufzeit $t_M(w)$ einer nichtdeterministischen Turingmaschine M auf einer Eingabe $w \in \Sigma^*$ ist definiert als die Länge des längsten Rechenweges von M bei Eingabe w. Gibt es bei Eingabe w einen Rechenweg, auf dem M nicht terminiert, so ist die Laufzeit unendlich. Wieder sei $t_M(n) = \max_{w \in \Sigma^n} t_M(w)$ die Worst-Case-Laufzeit für Eingaben der Länge $n \in \mathbb{N}$.*

Die Leserinnen und Leser werden sich an dieser Stelle vermutlich fragen, warum nichtdeterministische Turingmaschinen interessant sind und warum wir ihre Laufzeit auf die obige Art definiert haben. Wir erinnern noch einmal daran, dass das Modell der nichtdeterministischen Turingmaschine nur ein Hilfsmittel für die Analyse der Komplexität von Problemen ist und nicht mit dem Ziel definiert wurde, reale Rechner zu modellieren. Wir definieren nun die Klasse NP als das Äquivalent zu der Klasse P für nichtdeterministische Turingmaschinen.

Definition 4.6. *Ein Entscheidungsproblem L gehört genau dann zu der Komplexitätsklasse* NP*, wenn es eine nichtdeterministische Turingmaschine M gibt, die L entscheidet, und eine Konstante $k \in \mathbb{N}$, für die $t_M(n) = O(n^k)$ gilt.*

Zur Veranschaulichung betrachten wir zwei Beispiele für Probleme aus der Klasse NP.

Theorem 4.7. *Die Entscheidungsvarianten des Cliquenproblems und des Rucksackproblems liegen in* NP*.*

Beweis. Bei der Entscheidungsvariante des Cliquenproblems besteht die Eingabe aus einem ungerichteten Graphen $G = (V, E)$ mit n Knoten sowie einer Zahl $k \in \mathbb{N}$ und es soll entschieden werden, ob es in G eine Clique der Größe mindestens k gibt. Wir konstruieren eine nichtdeterministische Turingmaschine M für dieses Problem, die in zwei Phasen arbeitet. In der ersten Phase nutzt sie den Nichtdeterminismus und wählt eine beliebige Teilmenge der Knoten aus. In der zweiten Phase arbeitet sie deterministisch und überprüft, ob mindestens k Knoten ausgewählt wurden und diese eine Clique bilden. Ist dies der Fall, so akzeptiert sie, ansonsten verwirft sie die Eingabe.

Die erste Phase kann beispielsweise dadurch realisiert werden, dass die Turingmaschine zunächst deterministisch n Rauten auf das Band schreibt, den Kopf auf die erste Raute verschiebt und in einen speziellen Zustand q wechselt, in dem die Übergänge $((q, \#), (q, 0, R)) \in \delta$ und $((q, \#), (q, 1, R)) \in \delta$ erlaubt sind. Dies führt dazu, dass in den nächsten n Schritten eine beliebige Zeichenkette $x \in \{0, 1\}^n$ auf das Band geschrieben werden kann. Diese kann als Auswahl der Knoten interpretiert werden. Abgesehen von diesen Übergängen sind alle anderen Schritte der Turingmaschine deterministisch, d. h. der komplette Nichtdeterminismus ist in der Zeichenkette x enthalten.

Die soeben beschriebene Turingmaschine akzeptiert eine Eingabe genau dann, wenn es in dem Graphen $G = (V, E)$ eine Clique der Größe mindestens k gibt. Um dies zu sehen, betrachten wir zunächst eine Eingabe, für die dies der Fall ist. Sei $V' \subseteq V$ eine Clique der Größe mindestens k. Wählt die Turingmaschine diese Teilmenge in Phase 1 nichtdeterministisch aus, so akzeptiert sie die Eingabe in Phase 2. Enthält auf der anderen Seite der Graph G keine Clique der Größe mindestens k, so führt jede in Phase 1 ausgewählte Teilmenge dazu, dass die Eingabe verworfen wird, da sie entweder zu klein ist oder keine Clique bildet.

Betrachten wir nun die Laufzeit auf einer Eingabe (G, k). Phase 1 kann in polynomieller Zeit durchgeführt werden, da nur die Zeichenkette $\#^n$ auf das Band geschrieben wird und anschließend genau n nichtdeterministische Schritte erfolgen. Zu Beginn von Phase 2 ist bereits eine Teilmenge der Knoten ausgewählt. Es muss nur getestet werden, ob sie mindestens k Elemente enthält und zwischen jedem Paar von ausgewählten Knoten eine Kante verläuft. Auch dies geht in polynomieller Zeit. Damit ist insgesamt gezeigt, dass die Entscheidungsvariante des Cliquenproblems in NP liegt.

Der Beweis, dass die Entscheidungsvariante des Rucksackproblems in NP liegt, kann vollkommen analog geführt werden. Auch für dieses Problem kann eine nichtdeterministische Turingmaschine konstruiert werden, die in zwei Phasen arbeitet. In der ersten Phase wählt sie eine beliebige Teilmenge der Objekte nichtdeterministisch aus und in der zweiten Phase überprüft sie, ob das Gewicht dieser Teilmenge die Kapazität nicht überschreitet und ob die Teilmenge den vorgegebenen Nutzen erreicht. □

Sei $\textsc{Clique} \subseteq \{0, 1\}^*$ die Sprache aller Paare (G, k), wobei G ein Graph ist, der eine Clique der Größe mindestens k enthält. Wir gehen davon aus, dass das Paar (G, k) auf eine naheliegende Art über dem Alphabet $\{0, 1\}$ codiert ist. Den ersten Teil des vorangegangenen Theorems können wir dann auch kurz als $\textsc{Clique} \in \text{NP}$ schreiben. Im Beweis haben wir den Nichtdeterminismus nur dazu genutzt, eine Zeichenkette $x \in \{0, 1\}^n$ zu erzeugen, die eine Teilmenge der Knoten beschreibt. Für einen

gegebenen Graphen G, der eine Clique der Größe mindestens k enthält, kann man diese Zeichenkette als ein *Zertifikat* für $(G,k) \in$ Clique auffassen. Die zweite Phase kann man dann als Verifikation des Zertifikats betrachten. Die wesentlichen Eigenschaften sind, dass diese Verifikation in polynomieller Zeit durchgeführt werden kann, dass es für jede Eingabe $(G,k) \in$ Clique mindestens ein gültiges Zertifikat gibt und dass es für keine Eingabe $(G,k) \notin$ Clique ein gültiges Zertifikat gibt. Diese Einsicht kann man auf beliebige Sprachen aus NP übertragen und erhält die folgende Charakterisierung der Klasse NP, die ohne nichtdeterministische Turingmaschinen auskommt.

Theorem 4.8. *Eine Sprache $L \subseteq \Sigma^*$ ist genau dann in der Klasse* NP *enthalten, wenn es eine deterministische Turingmaschine V (einen* Verifizierer*) gibt, deren Worst-Case-Laufzeit polynomiell beschränkt ist, und ein Polynom p, sodass für jede Eingabe $x \in \Sigma^*$ gilt*

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

Dabei sei # ein beliebiges Zeichen, das zum Eingabealphabet des Verifizierers, aber nicht zu Σ gehört.

Beweis. Sei $L \in$ NP. Dann gibt es eine nichtdeterministische Turingmaschine $M = (Q, \Sigma, \Gamma, \square, q_0, \bar{q}, \delta)$, die L entscheidet, und ein Polynom r, für das $t_M(n) \leq r(n)$ gilt. Wir konstruieren nun einen deterministischen Verifizierer V für L, der die nichtdeterministische Turingmaschine M simuliert. Dabei gibt das Zertifikat y die nichtdeterministischen Entscheidungen vor.

In jeder Konfiguration kann die Turingmaschine M zwischen maximal $\ell := 3|Q||\Gamma|$ möglichen Rechenschritten wählen, da es für jedes Paar $(q,a) \in (Q \setminus \{\bar{q}\}) \times \Gamma$ maximal ℓ viele Tripel $(q', a', D) \in Q \times \Gamma \times \{L, N, R\}$ mit $((q,a),(q',a',D)) \in \delta$ gibt. Wir gehen davon aus, dass die Tripel aus $Q \times \Gamma \times \{L,N,R\}$ beliebig nummeriert sind. Dann kann jeder Rechenschritt durch eine Zahl aus der Menge $\{1, \ldots, \ell\}$ beschrieben werden. Eine solche Zahl kann mit $\ell^\star := \lceil \log_2 \ell \rceil$ Bits codiert werden, was für eine gegebene Turingmaschine M eine Konstante ist. Der Verifizierer V erhält die Eingabe $x\#y$ für ein Zertifikat $y \in \{0,1\}^m$. Ist die Länge m des Zertifikates nicht durch $\ell^\star$ teilbar oder echt größer als $\ell^\star r(|x|)$, so verwirft der Verifizierer die Eingabe $x\#y$. Ansonsten zerlegt er das Zertifikat y in eine Folge von Zeichenketten der Länge jeweils $\ell^\star$ und interpretiert diese als Folge $\alpha_1, \ldots, \alpha_{m'}$ von Zahlen aus der Menge $\{1, \ldots, \ell\}$, die jeweils einen Rechenschritt von M beschreiben. Beschreibt eine der Zeichenketten eine Zahl größer als ℓ, so verwirft der Verifizierer die Eingabe $x\#y$.

Der Verifizierer V simuliert das Verhalten von M auf der Eingabe x. In Schritt i versucht er den durch α_i beschriebenen Rechenschritt auszuführen. Ist dieser gemäß δ nicht erlaubt, so verwirft er die Eingabe $x\#y$. Sind alle durch $\alpha_1, \ldots, \alpha_{m'}$ beschriebenen Schritte erlaubt, so testet der Verifizierer V, ob nach diesen m' Schritten eine akzeptierende Endkonfiguration erreicht wird (d. h. die Turingmaschine M befindet sich im Zustand $\bar{q}$ und unter dem Kopf steht das Zeichen 1). Er akzeptiert die Eingabe $x\#y$, wenn dies der Fall ist, und verwirft sie ansonsten.

Wählen wir $p(n) = \ell^\star r(n) = O(r(n))$, so erfüllt dieser Verifizierer die gewünschten Eigenschaften. Zu einer Eingabe $x \in \Sigma^*$ existiert genau dann ein Zertifikat $y \in \{0,1\}^*$

mit $|y| \leq p(|x|)$, für das V die Eingabe $x\#y$ akzeptiert, wenn es einen Rechenweg der nichtdeterministischen Turingmaschine M der Länge höchstens $r(|x|)$ gibt, der bei Eingabe w zu einer akzeptierenden Endkonfiguration führt. Dies ist genau dann der Fall, wenn M die Eingabe x akzeptiert, was gleichbedeutend mit $x \in L$ ist.

Sei nun eine Sprache L gegeben, für die es ein Polynom p und einen Verifizierer V mit den geforderten Eigenschaften gibt. Wir konstruieren daraus eine nichtdeterministische Turingmaschine M für die Sprache L, die auf einer Eingabe x wie folgt arbeitet. In der ersten Phase erzeugt sie nichtdeterministisch ein beliebiges Zertifikat $y \in \{0,1\}^*$ mit $|y| \leq p(|x|)$. In der zweiten Phase simuliert sie den Verifizierer V auf der Eingabe $x\#y$ und akzeptiert genau dann die Eingabe x, wenn V die Eingabe $x\#y$ akzeptiert. Die Laufzeit der Turingmaschine M ist polynomiell beschränkt, da das Erzeugen von y in Zeit $O(p(|x|))$ erfolgt und der Verifizierer ebenfalls eine polynomielle Laufzeit besitzt.

Die Turingmaschine M akzeptiert ein Wort x genau dann, wenn es zu L gehört. Denn für jedes Wort $x \in L$ gibt es mindestens ein Zertifikat y (d. h. mindestens einen Rechenweg von M), der zu einer akzeptierenden Endkonfiguration führt. Andererseits gibt es für kein $x \notin L$ ein solches Zertifikat, das heißt, jedes solche Wort wird von M unabhängig von der nichtdeterministischen Wahl von y verworfen. Damit ist insgesamt gezeigt, dass M die Sprache L in polynomieller Zeit entscheidet. Somit gehört L zu NP. □

Intuitiv besagt die Charakterisierung von NP aus dem vorangegangenen Theorem, dass eine Sprache genau dann zu NP gehört, wenn es in polynomieller Zeit möglich ist, die Korrektheit eines Lösungsvorschlages zu überprüfen. Dies ist auf den ersten Blick eine deutlich schwächere Anforderung als bei der Klasse P. Bei einem Problem aus P muss es möglich sein, eine Lösung in polynomieller Zeit zu finden und nicht nur ihre Korrektheit zu überprüfen.

4.1.3 P versus NP

Wir haben die Klassen P und NP kennengelernt, aber bislang noch nichts über ihren Zusammenhang gesagt. Direkt aus den Definitionen folgt, dass $\mathrm{P} \subseteq \mathrm{NP}$ gilt, denn gibt es für eine Sprache eine polynomielle deterministische Turingmaschine M, so kann man diese auch als nichtdeterministische Turingmaschine auffassen, die vom Nichtdeterminismus keinen Gebrauch macht.

Es stellt sich nun die Frage, wie groß die Klasse NP ist. Da wir sie über nichtdeterministische Turingmaschinen definiert haben, die (vermutlich) kein physikalisch realisierbares Rechnermodell darstellen, wäre es prinzipiell sogar möglich, dass selbst nicht entscheidbare Sprachen zu NP gehören. Wir werden als erstes beweisen, dass dies nicht der Fall ist, da jede Sprache aus NP in exponentieller Zeit von einer deterministischen Turingmaschine entschieden werden kann.

Theorem 4.9. *Für jede Sprache $L \in \mathrm{NP}$ gibt es eine deterministische Turingmaschine M, die L entscheidet, und ein Polynom r, für das $t_M(n) \leq 2^{r(n)}$ gilt.*

Beweis. Es sei $L \in \text{NP}$ beliebig. Dann gibt es gemäß Theorem 4.8 ein Polynom p und einen polynomiellen Verifizierer V, sodass für jede Eingabe $x \in \Sigma^*$

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

gilt. Wir konstruieren nun eine deterministische Turingmaschine M für die Sprache L, die bei einer Eingabe x den Verifizierer V für alle möglichen Zertifikate $y \in \{0,1\}^*$ mit $|y| \leq p(|x|)$ auf der Eingabe $x\#y$ simuliert. Sie akzeptiert die Eingabe x genau dann, wenn der Verifizierer in mindestens einer dieser Simulationen akzeptiert. Aus der Eigenschaft des Verifizierers folgt direkt, dass die so konstruierte Turingmaschine M die Sprache L entscheidet. Sie akzeptiert eine Eingabe x nämlich genau dann, wenn ein gültiges Zertifikat y existiert.

Da es sich bei V um eine polynomielle deterministische Turingmaschine handelt, kann sie für eine gegebene Eingabe $x\#y$ mit $|y| \leq p(|x|)$ in Zeit $r'(|x|)$ für ein Polynom r' simuliert werden. Die Gesamtlaufzeit der Turingmaschine M ist dann durch $O(r'(|x|) \cdot 2^{p(|x|)})$ nach oben beschränkt, da es $\sum_{i=0}^{p(|x|)} 2^i = 2^{p(|x|)+1} - 1 = O(2^{p(|x|)})$ viele Zertifikate y gibt, die getestet werden. Für ein geeignetes Polynom r gilt $O(r'(|x|) \cdot 2^{p(|x|)}) \leq 2^{r(|x|)}$. □

Das vorangegangene Theorem zeigt, dass nichtdeterministische Turingmaschinen durch deterministische Turingmaschinen und damit auch durch reale Rechner simuliert werden können. Dabei kann sich die Laufzeit aber drastisch vergrößern und aus polynomiellen nichtdeterministischen Turingmaschinen können exponentielle deterministische Turingmaschinen werden. Es stellt sich nun die Frage, ob dies so sein muss, oder ob es eine bessere Möglichkeit gibt, nichtdeterministische Turingmaschinen durch deterministische zu simulieren, bei der kein so großer Zeitverlust auftritt.

Formaler formuliert ist dies die Frage, ob P = NP gilt oder ob es Sprachen aus NP gibt, die nicht in P enthalten sind. Dies ist die größte offene Frage der theoretischen Informatik. Auf ihre Lösung wurde vom Clay Mathematics Institute sogar ein Preisgeld in Höhe von einer Million US-Dollar ausgelobt. Wenn man die alternative Charakterisierung von NP in Theorem 4.8 betrachtet, scheint die Hypothese P ≠ NP naheliegend, da es intuitiv einfacher erscheint, ein Zertifikat auf Korrektheit zu prüfen als ein Zertifikat aus einer exponentiell großen Menge zu finden. Ein Beweis ist das aber natürlich nicht, obwohl auch die Mehrheit der Forscherinnen und Forscher im Bereich der theoretischen Informatik glaubt, dass P ≠ NP gilt.

4.2 NP-Vollständigkeit

Aus unserer bisherigen Diskussion geht noch nicht hervor, warum es überhaupt interessant ist, sich mit der Klasse NP zu beschäftigen. Wir wissen beispielsweise, dass die Entscheidungsvariante des Cliquenproblems zu NP gehört. Bislang können wir daraus als einzige Konsequenz ableiten, dass das Cliquenproblem gemäß Theorem 4.9 in exponentieller Zeit gelöst werden kann. Dies sieht man aber auch leicht direkt ein, ohne den Umweg über NP gehen zu müssen.

Die Bedeutung der Klasse NP ergibt sich erst dadurch, dass es Probleme aus NP gibt, auf die sich alle anderen Probleme aus NP in polynomieller Zeit reduzieren lassen (die Entscheidungsvariante des Cliquenproblems ist ein solches Beispiel). Diese Probleme nennt man NP-vollständige Probleme. Das bedeutet, dass ein polynomieller Algorithmus für ein solches Problem implizieren würde, dass man alle Probleme aus NP in polynomieller Zeit lösen kann. Geht man also von der Hypothese $\mathrm{P} \neq \mathrm{NP}$ aus und zeigt, dass ein gegebenes Problem NP-vollständig ist, so bedeutet das, dass man dieses Problem nicht effizient lösen kann. Dies werden wir im Folgenden präzisieren.

Definition 4.10. *Eine* polynomielle Reduktion *einer Sprache $A \subseteq \Sigma_1^*$ auf eine Sprache $B \subseteq \Sigma_2^*$ ist eine Many-One-Reduktion $f\colon \Sigma_1^* \to \Sigma_2^*$, die in polynomieller Zeit berechnet werden kann. Existiert eine solche Reduktion, so heißt A auf B* polynomiell reduzierbar *und wir schreiben $A \leq_p B$.*

Wir erinnern noch einmal daran, dass bei einer Many-One-Reduktion f

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

für alle $x \in \Sigma_1^*$ gilt. Dass die Many-One-Reduktion $f\colon \Sigma_1^* \to \Sigma_2^*$ in polynomieller Zeit berechnet werden kann, bedeutet formal, dass es eine Turingmaschine M und ein $k \in \mathbb{N}$ gibt, sodass M zu jedem $x \in \Sigma_1^*$ die Ausgabe $f(x)$ in Zeit $t_M(|x|) = O(|x|^k)$ berechnet.

Analog zu Theorem 3.9 können wir das folgende Resultat für polynomielle Reduktionen festhalten.

Theorem 4.11. *Es seien $A \subseteq \Sigma_1^*$ und $B \subseteq \Sigma_2^*$ zwei Sprachen, für die $A \leq_p B$ gilt. Ist $B \in \mathrm{P}$, so ist auch $A \in \mathrm{P}$. Ist $A \notin \mathrm{P}$, so ist auch $B \notin \mathrm{P}$.*

Beweis. Es genügt, den ersten Teil zu zeigen, da der zweite Teil direkt daraus folgt. Sei also $A \leq_p B$ mit der zugehörigen polynomiellen Reduktion $f\colon \Sigma_1^* \to \Sigma_2^*$ und sei $B \in \mathrm{P}$. Ferner sei M_B eine Turingmaschine, die die Sprache B in polynomieller Zeit entscheidet und es sei M_f eine Turingmaschine, die die Reduktion f in polynomieller Zeit berechnet. Wir konstruieren eine Turingmaschine M_A für A, die bei einer Eingabe x zunächst M_f simuliert und $f(x)$ berechnet und dann anschließend M_B auf $f(x)$ simuliert.

Wie schon im Beweis von Theorem 3.9 diskutiert, entscheidet die so konstruierte Turingmaschine M_A die Sprache A. Wir müssen lediglich noch zeigen, dass ihre Laufzeit polynomiell ist. Seien p und q Polynome, für die $t_{M_B}(n) \leq p(n)$ und $t_{M_f}(n) \leq q(n)$ gilt. Die Laufzeit von M_A bei einer Eingabe x der Länge n beträgt dann $O(q(n) + p(q(n) + n))$, da zunächst $f(x)$ in Zeit $O(q(n))$ berechnet wird und anschließend die Turingmaschine M_B auf der Eingabe $f(x)$ simuliert wird. Die Länge der Eingabe $f(x)$ können wir durch $q(n)+n$ nach oben abschätzen, da die Turingmaschine M_f pro Rechenschritt nur maximal ein Zeichen zur Ausgabe hinzufügen kann. Insofern können wir die Laufzeit zur Simulation von M_B auf der Eingabe $f(x)$ durch $O(p(q(n) + n))$ nach oben beschränken. Wir haben bereits nach Theorem 2.6 diskutiert, dass die Verschachtelung zweier Polynome wieder ein Polynom ist. Damit ist das Theorem bewiesen. □

Wir betrachten nun ein Beispiel für eine polynomielle Reduktion und führen dafür zunächst das *Vertex-Cover-Problem (VC)* ein. Bei diesem Problem sind als Eingabe ein ungerichteter Graph $G = (V, E)$ und eine Zahl $k \in \mathbb{N}$ gegeben. Es soll entschieden werden, ob es eine Teilmenge $V' \subseteq V$ der Knoten mit $|V'| \leq k$ gibt, sodass jede Kante aus E zu mindestens einem Knoten aus V' inzident ist. Eine solche Teilmenge nennen wir ein *Vertex Cover* von G.

Theorem 4.12. *Es gilt* Clique $\leq_p$ VC.

Beweis. Sei eine Eingabe für das Cliquenproblem bestehend aus einem ungerichteten Graphen $G = (V, E)$ und einer Zahl $k \in \mathbb{N}$ gegeben. Wir konstruieren daraus eine Eingabe für das Vertex-Cover-Problem bestehend aus einem Graphen $G' = (V', E')$ und einer Zahl $k' \in \mathbb{N}$. Dabei gilt $V' = V$ und E' enthält genau die Kanten, die E nicht enthält, d. h.

$$E' = \{\{x, y\} \mid x, y \in V, x \neq y, \{x, y\} \notin E\}.$$

Außerdem setzen wir $k' = n - k$ für $n = |V|$. Zusammengefasst entspricht die Reduktion der Funktion f mit $f(G, k) = (G', k')$. Diese Funktion ist in polynomieller Zeit berechenbar. Wir müssen nun Folgendes zeigen: G enthält genau dann eine Clique der Größe k, wenn G' ein Vertex Cover der Größe k' enthält. Die Reduktion ist in Abbildung 4.1 für ein Beispiel dargestellt.

Sei zunächst $C \subseteq V$ eine Clique der Größe k in G. Wir behaupten, dass dann $V \setminus C$ ein Vertex Cover in G' der Größe $k' = n - k$ ist. Es gilt $|V \setminus C| = |V| - |C| = n - k = k'$. Es bleibt zu zeigen, dass $V \setminus C$ ein Vertex Cover in G' ist. Angenommen, dies ist nicht der Fall. Dann existiert eine Kante $\{x, y\} \in E'$, die durch $V \setminus C$ nicht abgedeckt ist, was gleichbedeutend zu $x \notin V \setminus C$ und $y \notin V \setminus C$, also $x, y \in C$ ist. Da C eine Clique ist, gilt $\{x, y\} \in E$, was aufgrund der Definition von E' aber der Aussage $\{x, y\} \in E'$ widerspricht. Somit ist die erste Richtung der Äquivalenz gezeigt.

Sei nun andersherum $D \subseteq V' = V$ ein Vertex Cover der Größe k' in G'. Wir behaupten, dass dann $V \setminus D$ eine Clique in G der Größe k ist. Es gilt $|V \setminus D| = |V| - |D| = n - k' = k$. Es bleibt zu zeigen, dass $V \setminus D$ eine Clique in G ist. Angenommen, dies ist nicht der Fall. Dann existieren zwei Knoten $x, y \in V \setminus D$ mit $\{x, y\} \notin E$. Aufgrund der Definition von E' gilt $\{x, y\} \in E'$. Da es sich bei D um ein Vertex Cover von G' handelt, muss $x \in D$ oder $y \in D$ gelten im Widerspruch zu $x, y \in V \setminus D$. Damit ist auch die zweite Richtung der Äquivalenz gezeigt. □

Wir betrachten noch ein weiteres Beispiel für eine polynomielle Reduktion. Dazu führen wir zunächst eine Variante des Kürzeste-Wege-Problems ein, die wir *beschränktes Kürzeste-Wege-Problem* nennen. Genauso wie beim normalen Kürzeste-Wege-Problem enthält eine Eingabe für dieses Problem einen gerichteten Graphen $G = (V, E)$ mit Kantengewichten $w \colon E \to \mathbb{N}_0$ sowie einen Startknoten $s \in V$ und einen Zielknoten $t \in V$. Um aus dem Kürzeste-Wege-Problem ein Entscheidungsproblem zu machen, fügen wir der Eingabe zusätzlich eine Zahl $W \in \mathbb{N}_0$ hinzu, und fragen, ob es einen Weg P von s nach t gibt, dessen Gewicht $\sum_{e \in P} w(e)$ höchstens W beträgt. Beim beschränkten Kürzeste-Wege-Problem wird die Situation dadurch erschwert, dass es

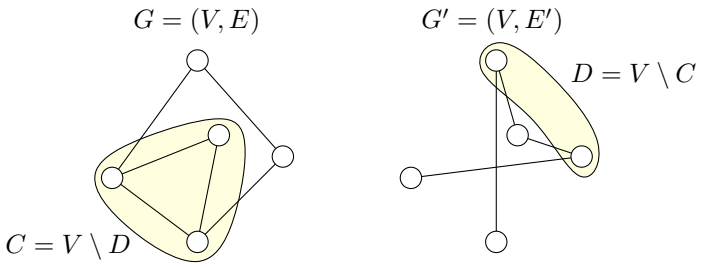


Abbildung 4.1: Illustration der Reduktion Clique $\leq_p$ VC an einem Beispiel. C ist eine Clique der Größe 3 in G und $D = V \setminus C$ ist ein Vertex Cover der Größe $5 - 3 = 2$ in G'.

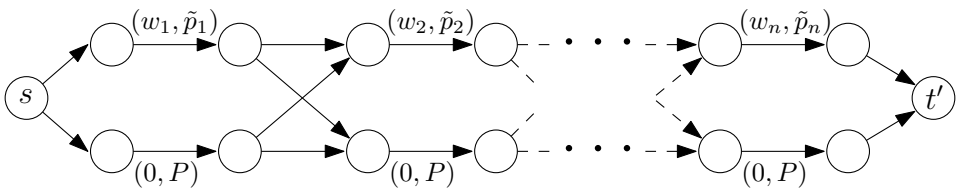
zusätzlich noch Kantenkosten $c\colon E \to \mathbb{N}_0$ und ein Budget $C \in \mathbb{N}_0$ gibt. Die Frage ist dann, ob es einen Weg P von s nach t gibt, dessen Gewicht $\sum_{e \in P} w(e)$ höchstens W beträgt und der das Budget einhält, d. h. für den $\sum_{e \in P} c(e) \leq C$ gilt. Dieses Problem tritt in vielen Anwendungen auf. Denkt man beispielsweise an Routenplanung, so könnte das Gewicht einer Kante angeben, wie lange es dauert, die entsprechende Strecke zurückzulegen, und die Kosten könnten der Maut entsprechen.

Wir bezeichnen im Folgenden mit KP die Entscheidungsvariante des Rucksackproblems und mit BKWP die soeben definierte Entscheidungsvariante des beschränkten Kürzeste-Wege-Problems. Sowohl bei KP als auch bei BKWP handelt es sich um Sprachen über geeignet gewählten endlichen Alphabeten. Wir können ohne Beschränkung der Allgemeinheit davon ausgehen, dass beide Sprachen über dem Alphabet $\{0, 1\}$ definiert sind.

Theorem 4.13. *Es gilt* KP $\leq_p$ BKWP.

Beweis. Sei $\mathcal{I} = (p_1, \ldots, p_n, w_1, \ldots, w_n, t, z)$ eine Eingabe für das Rucksackproblem, wobei wir davon ausgehen, dass alle Komponenten aus $\mathbb{N}_0$ sind. Diese Eingabe (bzw. eine geeignete Codierung $\text{code}(\mathcal{I}) \in \{0, 1\}^*$ von $\mathcal{I}$) gehört genau dann zu KP, wenn es eine Menge $I \subseteq \{1, \ldots, n\}$ mit $\sum_{i \in I} w_i \leq t$ und $\sum_{i \in I} p_i \geq z$ gibt. Der Einfachheit halber werden wir im Folgenden stets $\mathcal{I} \in \text{KP}$ statt $\text{code}(\mathcal{I}) \in \text{KP}$ schreiben. Dabei setzen wir eine beliebige sinnvolle Codierung voraus, deren Länge polynomiell in n und dem Logarithmus der größten Zahl in $\mathcal{I}$ beschränkt ist.

Wir transformieren nun die gegebene Eingabe für das Rucksackproblem in eine Eingabe für das beschränkte Kürzeste-Wege-Problem. Dazu konstruieren wir den Graphen G mit $4n + 2$ Knoten, der in der folgenden Abbildung dargestellt ist.



In diesem Graphen sei s der Startknoten und t' der Zielknoten. An einigen Kanten finden sich Beschriftungen der Form (a, b). Damit ist gemeint, dass die entsprechende

Kante ein Gewicht von a und Kosten in Höhe von b besitzt. Es sei $P = \max_i p_i$ und $\tilde{p}_i = P - p_i$. Für Kanten, die nicht beschriftet sind, sind sowohl das Gewicht als auch die Kosten Null. Man kann sich bei diesen Kanten also die Beschriftung $(0, 0)$ dazu denken. Schließlich setzen wir noch W auf die gegebene Kapazität t des Rucksacks und $C = nP - z$.

Wir haben somit aus der Instanz $\mathcal{I}$ für das Rucksackproblem eine Instanz $\mathcal{I}'$ des BKWP erzeugt, in der die Frage darin besteht, ob in dem obigen Graphen ein Weg von der Quelle s zur Senke t' existiert, dessen Gesamtgewicht höchstens $W = t$ beträgt und dessen Gesamtkosten höchsten $C = nP - z$ betragen. Ist dies der Fall, so schreiben wir $\mathcal{I}' \in \text{BKWP}$. Genau wie beim Rucksackproblem müssten wir formal korrekt eigentlich $\text{code}(\mathcal{I}') \in \text{BKWP}$ für eine geeignete Codierung von $\mathcal{I}'$ schreiben.

Die Instanz $\mathcal{I}'$ für das beschränkte Kürzeste-Wege-Problem können wir für eine gegebene Instanz $\mathcal{I}$ des Rucksackproblems in polynomieller Zeit erzeugen. Es bleibt zu zeigen, dass es sich um eine Reduktion handelt, dass also $\mathcal{I} \in \text{KP} \iff \mathcal{I}' \in \text{BKWP}$ gilt.

- Wir zeigen zunächst $\mathcal{I} \in \text{KP} \Rightarrow \mathcal{I}' \in \text{BKWP}$. Wegen $\mathcal{I} \in \text{KP}$ gibt es eine Menge $I \subseteq \{1, \ldots, n\}$ mit $\sum_{i \in I} w_i \leq t$ und $\sum_{i \in I} p_i \geq z$. Aus dieser Menge konstruieren wir einen Weg T von s nach t' in dem Graphen G. Dieser benutzt für jedes $i \in I$ die Kante mit Gewicht/Kostenkombination $(w_i, \tilde{p}_i)$ und für jedes $i \notin I$ die darunterliegende Kante mit Gewicht/Kostenkombination $(0, P)$. Ansonsten benutzt der Weg nur Kanten, die sowohl Gewicht als auch Kosten Null haben. Ein solcher Weg kann in dem gegebenen Graphen einfach konstruiert werden.

  Der Weg T besitzt ein Gesamtgewicht von

  $$\sum_{i \in I} w_i + \sum_{i \notin I} 0 = \sum_{i \in I} w_i \leq t = W$$

  und Gesamtkosten von

  $$\sum_{i \in I} \tilde{p}_i + \sum_{i \notin I} P = \sum_{i \in I} (P - p_i) + \sum_{i \notin I} P = nP - \sum_{i \in I} p_i \leq nP - z = C.$$

  Der Weg T beweist, dass $\mathcal{I}' \in \text{BKWP}$ gilt.

- Nun zeigen wir noch $\mathcal{I}' \in \text{BKWP} \Rightarrow \mathcal{I} \in \text{KP}$. Wegen $\mathcal{I}' \in \text{BKWP}$ existiert in G ein Weg T von s nach t' mit Gesamtgewicht höchstens $W = t$ und Gesamtkosten von höchstens $C = nP - z$. Wir konstruieren aus diesem Weg T eine Lösung $I \subseteq \{1, \ldots, n\}$ für das Rucksackproblem. Zunächst beobachten wir, dass aus der Struktur des Graphen G direkt folgt, dass für jedes $i \in \{1, \ldots, n\}$ entweder die Kante mit Gewicht/Kostenkombination $(w_i, \tilde{p}_i)$ oder die darunterliegende Kante mit Gewicht/Kostenkombination $(0, P)$ enthalten ist. Die Menge I enthält ein Objekt i genau dann, wenn der Weg T die Kante mit Gewicht/Kostenkombination $(w_i, \tilde{p}_i)$ enthält.

  Die Menge I besitzt ein Gesamtgewicht von

  $$\sum_{i \in I} w_i = \sum_{e \in T} w(e) \leq W = t.$$

Da T Gesamtkosten höchstens $C = nP - z$ besitzt und sich die Kosten von T als

$$\sum_{i \in I} \tilde{p}_i + \sum_{i \notin I} P = \sum_{i \in I} (P - p_i) + \sum_{i \notin I} P = nP - \sum_{i \in I} p_i$$

schreiben lassen, folgt

$$nP - \sum_{i \in I} p_i \leq C = nP - z,$$

was äquivalent zu

$$\sum_{i \in I} p_i \geq z$$

ist. Damit ist I, wie gewünscht, eine Auswahl der Objekte mit Gesamtgewicht höchstens t und Gesamtnutzen mindestens z.

Insgesamt folgt nun KP $\leq_p$ BKWP. □

Eine wesentliche Implikation von Theorem 4.13 ist, dass jeder effiziente Algorithmus für das beschränkte Kürzeste-Wege-Problem mithilfe der Reduktion in einen effizienten Algorithmus für das Rucksackproblem umgebaut werden kann. Könnten wir also beispielsweise den Algorithmus von Dijkstra so anpassen, dass er effizient das beschränkte Kürzeste-Wege-Problem löst, so hätten wir damit auch direkt einen effizienten Algorithmus für das Rucksackproblem gefunden. Könnte man andersherum zeigen, dass es für das Rucksackproblem keinen effizienten Algorithmus gibt, so folgt aus der Reduktion, dass es auch keinen effizienten Algorithmus für das beschränkte Kürzeste-Wege-Problem geben kann.

Polynomielle Reduktionen setzen also die Komplexität von Problemen in einem gewissen Sinne zueinander in Beziehung. Es gibt mittlerweile in der Literatur Tausende Beispiele für polynomielle Reduktionen. Weitere polynomielle Reduktionen ergeben sich aus den bekannten, wenn man die Transitivität ausnutzt. Das bedeutet, wenn $A \leq_p B$ und $B \leq_p C$ für drei Sprachen A, B und C gilt, so gilt auch $A \leq_p C$. Die Leserinnen und Leser sollten dies als Übung beweisen.

Die große Bedeutung von polynomiellen Reduktionen ergibt sich daraus, dass es damit auch möglich ist, die Komplexität eines Problem zu der Komplexität aller Probleme aus NP in Beziehung zu setzen. Um dies zu erläutern, benötigen wir zunächst die folgende Definition.

Definition 4.14. *Eine Sprache L heißt* NP-schwer*, wenn $L' \leq_p L$ für jede Sprache $L' \in$ NP gilt. Ist eine Sprache L* NP*-schwer und gilt zusätzlich $L \in$ NP, so heißt L* NP-vollständig.

NP-vollständige Sprachen sind informell gesprochen, die schwierigsten Sprachen aus der Klasse NP. Das folgende Theorem zeigt, dass sie genau dann effizient gelöst werden können, wenn P = NP gilt.

Theorem 4.15. *Gibt es eine* NP*-schwere Sprache $L \in$ P, so gilt* P = NP.

Beweis. Sei $L' \in \mathrm{NP}$ beliebig. Aus Definition 4.14 folgt, dass $L' \leq_p L$ gilt. Wegen $L \in \mathrm{P}$ und Theorem 4.11 folgt daraus $L' \in \mathrm{P}$. Da dieses Argument für jede Sprache $L' \in \mathrm{NP}$ gilt, folgt $\mathrm{NP} \subseteq \mathrm{P}$. Wir haben bereits oben diskutiert, dass $\mathrm{P} \subseteq \mathrm{NP}$ gilt. Damit ist das Theorem bewiesen. □

Korollar 4.16. *Es sei L eine* NP*-vollständige Sprache. Dann gilt $L \in \mathrm{P}$ genau dann, wenn $\mathrm{P} = \mathrm{NP}$ gilt.*

Können wir für ein Problem beweisen, dass es NP-vollständig ist, so ist dies ein starkes Indiz dafür, dass das Problem nicht effizient gelöst werden kann, denn ein effizienter Algorithmus für dieses Problem würde auch für viele andere gut untersuchte Probleme wie zum Beispiel das Cliquenproblem und das Rucksackproblem effiziente Algorithmen implizieren. Auf den ersten Blick ist es allerdings vollkommen unklar, ob es überhaupt NP-vollständige Probleme gibt. Es wäre gut möglich, dass es gar kein Problem aus NP gibt, auf das alle anderen Probleme aus dieser Klasse reduziert werden können.

Unabhängig voneinander haben Stephen Cook und Leonid Levin 1971 bzw. 1973 auf den beiden Seiten des eisernen Vorhangs bewiesen, dass das Erfüllbarkeitsproblem der Aussagenlogik NP-vollständig ist. Dieses Problem ist uns bereits im ersten Semester begegnet und wir werden es im Folgenden mit SAT (engl. satisfiability) bezeichnen. Eine Eingabe für SAT besteht aus einer aussagenlogischen Formel und es soll entschieden werden, ob es eine Belegung der Variablen gibt, die diese Formel erfüllt.

Wir betrachten nur Formeln in konjunktiver Normalform als Eingaben und erinnern die Leserinnen und Leser an die folgende Definition aus der Vorlesung „Logik und diskrete Strukturen“.

Definition 4.17. *Eine Formel der Form x oder $\neg x$ für eine Variable x nennen wir ein* Literal*. Ein Literal der Form x nennen wir* positives Literal *und ein Literal der Form $\neg x$ nennen wir* negatives Literal*.*

Eine aussagenlogische Formel φ ist in konjunktiver Normalform (KNF)*, wenn sie eine Konjunktion von Disjunktionen von Literalen ist, d. h. wenn sie die Gestalt*

$$\varphi = \bigwedge_{i=1}^{n} \left(\bigvee_{j=1}^{m_i} \ell_{i,j} \right)$$

hat, wobei $n, m_1, \ldots, m_n \in \mathbb{N}$ gilt und $\ell_{i,j}$ für jedes i und j ein Literal ist.

Die Teilformeln $\bigvee_{j=1}^{m_i} \ell_{i,j}$ nennen wir die Klauseln *von φ.*

Wieder gehen wir auf die Codierung von Formeln als Zeichenketten nicht näher ein und gehen nur davon aus, dass eine sinnvolle binäre Codierung gewählt wurde, dass also $\mathrm{SAT} \subseteq \{0,1\}^*$ gilt.

Theorem 4.18 (Satz von Cook und Levin)**.** SAT *ist* NP*-vollständig.*

Beweis. Der Beweis besteht aus zwei Teilen. Wir müssen zeigen, dass SAT in NP liegt und dass sich jedes Problem aus NP polynomiell auf SAT reduzieren lässt. Eine

polynomielle nichtdeterministische Turingmaschine für SAT kann einfach konstruiert werden. Für eine gegebene aussagenlogische Formel erzeugt diese analog zum Beweis von Theorem 4.7 zunächst nichtdeterministisch eine beliebige Belegung der Variablen (d. h. jede Variable wird nichtdeterministisch auf 0 oder 1 gesetzt). Anschließend überprüft die Turingmaschine deterministisch, ob die erzeugte Belegung die Formel erfüllt, und akzeptiert genau dann, wenn dies der Fall ist. Diese Überprüfung kann für eine gegebene Belegung und eine gegebene Formel leicht in polynomieller Zeit durchgeführt werden. Genau dann, wenn die Formel erfüllbar ist, gibt es einen Rechenweg der nichtdeterministischen Turingmaschine, der zu einer akzeptierenden Endkonfiguration führt.

Der zweite Teil ist deutlich schwieriger zu beweisen. Wir müssen nachweisen, dass jede Sprache aus NP polynomiell auf SAT reduziert werden kann. Sei $L \in \mathrm{NP}$ beliebig. Über L wissen wir nur, dass es eine nichtdeterministische Turingmaschine $M = (Q, \Sigma, \Gamma, \square, q_0, \bar{q}, \delta)$ gibt, die L in polynomieller Zeit entscheidet. Sei p ein Polynom, für das $t_M(n) \leq p(n)$ für alle $n \in \mathbb{N}_0$ gilt.

Unser Ziel ist es, eine polynomiell berechenbare Funktion $f\colon \Sigma^* \to \{0,1\}^*$ zu finden, die jede Eingabe $x \in \Sigma^*$ für L in eine aussagenlogische Formel $f(x)$ in konjunktiver Normalform übersetzt, sodass gilt:

$$x \in L \iff f(x) \text{ ist erfüllbar.}$$

Sei im Folgenden $x \in \Sigma^*$ mit $n := |x|$ beliebig. Wir beschreiben nun, wie die Formel $\varphi := f(x)$ erzeugt wird. Da wir über L nichts weiter wissen, als dass es die nichtdeterministische Turingmaschine M gibt, die L entscheidet, müssen wir genau an dieser Stelle ansetzen. Die grobe Idee des Beweises besteht darin, dass wir mithilfe der Formel φ die Berechnung der Turingmaschine M simulieren wollen.

Um dies zu präzisieren, erinnern wir uns noch einmal daran, dass wir unter der *Konfiguration* einer Turingmaschine zu einem Zeitpunkt die aktuelle Kombination aus Bandinhalt, Kopfposition und Zustand verstehen. Sind K und K' Konfigurationen von M, so sagen wir, dass K' eine *direkte Nachfolgekonfiguration* von K ist, wenn es in Konfiguration K einen erlaubten Rechenschritt der Turingmaschine M gibt, der zu Konfiguration K' führt. Wir schreiben dann $K \vdash K'$. Bezeichnen wir mit K_0 die initiale Konfiguration der Turingmaschine M bei Eingabe x, so soll die Formel φ genau dann erfüllbar sein, wenn es eine Folge von Konfigurationen $K_0, K_1, \ldots, K_\ell$ mit $\ell \leq p(n)$ und $K_0 \vdash K_1 \vdash K_2 \vdash \ldots \vdash K_\ell$ gibt, wobei K_ℓ eine akzeptierende Endkonfiguration ist. Da die Turingmaschine M die Sprache L entscheidet und $t_M(n) \leq p(n)$ gilt, gibt es genau dann eine solche Folge von Konfigurationen, wenn $x \in L$ gilt.

Um die Konstruktion der Formel φ zu vereinfachen, bauen wir die Turingmaschine M ein wenig um. Eine akzeptierende Endkonfiguration ist dadurch gekennzeichnet, dass der Endzustand $\bar{q}$ erreicht ist und direkt unter dem Kopf das Zeichen 1 steht. Wir fügen der Turingmaschine M nun einen neuen Zustand q_{akz} hinzu und ändern die Zustandsüberführungsrelation δ so ab, dass in einer akzeptierenden Endkonfiguration der neue Zustand q_{akz} statt $\bar{q}$ erreicht wird. Jeder mögliche Übergang in den Zustand $\bar{q}$ wird also darauf überprüft, ob er zu einer akzeptierenden oder verwerfenden Endkonfiguration

führt. Führt er zu einer verwerfenden Endkonfiguration, so wird er nicht verändert. Führt er jedoch zu einer akzeptierenden Endkonfiguration, so wird statt in $\bar{q}$ in den Zustand q_{akz} gewechselt. Ist dieser Zustand einmal erreicht, so terminiert die Turingmaschine nicht, sondern sie gerät in eine Endlosschleife, in der sie den Zustand q_{akz} nicht mehr verlässt, den Kopf nicht mehr bewegt und keine Veränderungen am Band mehr vornimmt.

Durch die Einführung des Zustandes q_{akz} haben wir erreicht, dass wir uns nur noch Rechenwege der Länge genau $p(n)$ anschauen müssen. Unser Ziel ist es nun also, die Formel φ so zu generieren, dass die folgende Eigenschaft gilt.

> Die Formel φ ist genau dann erfüllbar, wenn es eine Folge von Konfigurationen $K_1, K_2, \ldots, K_{p(n)}$ der Turingmaschine M mit $K_0 \vdash K_1 \vdash K_2 \vdash \ldots \vdash K_{p(n)}$ gibt, wobei in $K_{p(n)}$ der Zustand q_{akz} angenommen wird und K_0 die initiale Konfiguration der Turingmaschine M bei Eingabe x ist.

Definition der Variablen: Die Variablen der Formel φ werden die Konfigurationen $K_0, \ldots, K_{p(n)}$ codieren. Sie enthalten also für jeden Zeitpunkt $t \in \{0, 1, \ldots, p(n)\}$ Informationen über den Bandinhalt, die Kopfposition und den Zustand. Nummerieren wir die Zellen des Bandes mit ganzen Zahlen und treffen die Konvention, dass die initiale Kopfposition den Index 0 hat, so kann die Turingmaschine in $p(n)$ Schritten nur die Zellen mit den Indizes $-p(n), -p(n)+1, \ldots, p(n)-1, p(n)$ erreichen. Alle anderen Zellen enthalten stets das Leerzeichen und sind deshalb nicht von Interesse. Wir können demnach die Konfigurationen $K_0, K_1, \ldots, K_{p(n)}$ mithilfe der folgenden (aussagenlogischen) Variablen vollständig beschreiben.

- $Q(t, q)$ für $t \in \{0, \ldots, p(n)\}$ und $q \in Q$ mit

$$Q(t,q) = \begin{cases} 1 & \text{falls in } K_t \text{ Zustand } q \text{ angenommen wird,} \\ 0 & \text{sonst.} \end{cases}$$

- $H(t, j)$ für $t \in \{0, \ldots, p(n)\}$ und $j \in \{-p(n), \ldots, p(n)\}$ mit

$$H(t,j) = \begin{cases} 1 & \text{falls Kopf in } K_t \text{ auf Zelle } j \text{ steht,} \\ 0 & \text{sonst.} \end{cases}$$

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

$$S(t,j,a) = \begin{cases} 1 & \text{falls Zelle } j \text{ in } K_t \text{ das Zeichen } a \text{ enthält,} \\ 0 & \text{sonst.} \end{cases}$$

Die Anzahl der Variablen ist polynomiell in n beschränkt, da p ein Polynom ist und Q und Γ endliche Mengen konstanter Größe sind.

Codierung einzelner Konfigurationen: Als erstes geben wir Klauseln an, die sicherstellen, dass die Variablen für jedes feste t eine Konfiguration von M codieren. Sei also $t \in \{0, \ldots, p(n)\}$ beliebig. Dann muss jede erfüllende Belegung von φ dergestalt sein,

- dass es genau ein $q \in Q$ mit $Q(t, q) = 1$ gibt,
- dass es genau ein $j \in \{-p(n), \ldots, p(n)\}$ mit $H(t, j) = 1$ gibt und
- dass es für jedes $j \in \{-p(n), \ldots, p(n)\}$ genau ein $a \in \Gamma$ mit $S(t, j, a) = 1$ gibt.

Diese drei Bedingungen haben gemeinsam, dass für eine bestimmte Variablenmenge codiert werden muss, dass genau eine der Variablen auf 1 und alle anderen auf 0 gesetzt sind. Für eine Variablenmenge $\{y_1, \ldots, y_m\}$ kann dies durch die Formel

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

erreicht werden. Diese Formel ist in konjunktiver Normalform und sie besitzt eine Länge von $O(m^2)$, wobei wir unter der Länge die Anzahl der auftretenden Literale verstehen. Wir können auf diese Weise also die obigen drei Bedingungen mit einer Formel φ_t der Länge $O(p(n)^2)$ in konjunktiver Normalform codieren.

Codierung von Konfigurationsübergängen: Mithilfe der Formeln φ_t können wir für jedes einzelne t erreichen, dass jede gültige Variablenbelegung eine Konfiguration K_t beschreibt. Diese Konfigurationen haben aber im Allgemeinen nichts miteinander zu tun. Deshalb müssen wir als nächstes codieren, dass K_t für jedes $t \in \{1, \ldots, p(n)\}$ eine direkte Nachfolgekonfiguration von K_{t-1} ist. Sei im Folgenden $t \in \{1, \ldots, p(n)\}$ beliebig.

Zunächst erzwingen wir, dass sich der Bandinhalt nur in der Zelle ändern kann, an der sich der Kopf befindet. Dies erreichen wir durch die Formel

$$\bigwedge_{j=-p(n)}^{p(n)} \bigwedge_{a \in \Gamma} \Big((S(t-1, j, a) \wedge \neg H(t-1, j)) \Rightarrow S(t, j, a) \Big).$$

Diese besagt, dass für jede Position $j \in \{-p(n), \ldots, p(n)\}$ und jedes Zeichen $a \in \Gamma$ gilt: Wenn zum Zeitpunkt $t-1$ an Position j das Zeichen a steht ($S(t-1, j, a)$) und der Kopf sich zum Zeitpunkt $t-1$ nicht an Position j befindet ($\neg H(t-1, j)$), so muss auch zum Zeitpunkt t an Position j das Zeichen a stehen ($S(t, j, a)$). Den Implikationspfeil haben wir der besseren Lesbarkeit halber verwendet. Man kann $A \implies B$ durch den äquivalenten Ausdruck $\neg A \vee B$ ersetzen. Wendet man dann noch das De Morgansche Gesetz an, dass $\neg(A \wedge B)$ äquivalent zu $\neg A \vee \neg B$ ist, so erhält man die folgende Formel in konjunktiver Normalform:

$$\bigwedge_{j=-p(n)}^{p(n)} \bigwedge_{a \in \Gamma} \Big(\neg S(t-1, j, a) \vee H(t-1, j) \vee S(t, j, a) \Big). \tag{4.1}$$

Diese Formel besitzt eine Länge von $O(p(n))$.

Nun müssen wir noch erreichen, dass im Schritt von K_{t-1} zu K_t ein durch δ beschriebener Rechenschritt ausgeführt wird. Dazu erzeugen wir für jedes $q \in Q$, jedes $j \in \{-p(n), \ldots, p(n)\}$ und jedes $a \in \Gamma$ die Formel

$$\begin{aligned} &\Big(Q(t-1,q) \wedge H(t-1,j) \wedge S(t-1,j,a)\Big) \\ \Rightarrow \quad &\bigvee_{((q,a),(q',a',D))\in\delta} \Big(Q(t,q') \wedge H(t,j+D) \wedge S(t,j,a')\Big), \end{aligned}$$

wobei wir der Einfachheit halber davon ausgehen, dass $D \in \{-1, 0, +1\}$ statt $D \in \{L, N, R\}$ die Bewegung des Kopfes angibt. Diese Formel besagt: Wenn sich die Turingmaschine zum Zeitpunkt $t-1$ in Zustand q befindet ($Q(t-1,q)$), der Kopf sich an Position j befindet ($H(t-1,j)$) und an Position j das Zeichen a steht ($S(t-1,j,a)$), so gilt für einen Übergang $((q,a),(q',a',D)) \in \delta$, dass sich die Turingmaschine zum Zeitpunkt t in Zustand q' befindet ($Q(t,q')$), der Kopf sich an Position $j+D$ befindet ($H(t,j+D)$) und an Position j das Zeichen a' steht ($S(t,j,a')$).

Wir haben bereits in der Vorlesung „Logik und diskrete Strukturen“ gesehen, dass es zu jeder Formel eine äquivalente Formel in konjunktiver Normalform gibt. Im Allgemeinen kann diese Formel um einen Faktor länger sein, der exponentiell in der Zahl der Variablen ist. Da die obige Formel für festes t sowie feste q, j und a konstante Länge besitzt (es gibt nur konstant viele Tripel (q',a',D) mit $((q,a),(q',a',D)) \in \delta$), gibt es eine äquivalente Formel in konjunktiver Normalform, die ebenfalls konstante Länge besitzt. Konjugieren wir die Formeln für jede Wahl von q, j und a, so erhalten wir insgesamt eine Formel der Länge $O(p(n))$.

Zusammen mit (4.1) erhalten wir somit für jedes $t \in \{1, \ldots, p(n)\}$ eine Formel $\varphi_{\to t}$ in konjunktiver Normalform, die codiert, dass K_t eine direkte Nachfolgekonfiguration von K_{t-1} ist.

Codierung der initialen Konfiguration: Um zu codieren, dass K_0 die initiale Konfiguration der Turingmaschine M bei Eingabe x ist, müssen wir erreichen, dass sich M in K_0 im Startzustand q_0 befindet, der Bandinhalt x ist und der Kopf auf dem ersten Zeichen von x steht. Sei $x = x_1 \ldots x_n$. Dann kann dies durch die Formel

$$\varphi_{\text{init}} = Q(0,q_0) \wedge H(0,0) \wedge \bigwedge_{i=1}^{n} S(0,i-1,x_i) \wedge \bigwedge_{j=-p(n)}^{-1} S(0,j,\square) \wedge \bigwedge_{j=n}^{p(n)} S(0,j,\square)$$

in konjunktiver Normalform erreicht werden.

Zusammensetzen der Formel: Nun können wir alle Teilformeln zusammensetzen und zusätzlich noch codieren, dass nach $p(n)$ Schritten der Zustand q_{akz} erreicht werden soll. Es ergibt sich die Formel

$$\varphi = \varphi_{\text{init}} \wedge \left(\bigwedge_{t=0}^{p(n)} \varphi_t\right) \wedge \left(\bigwedge_{t=1}^{p(n)} \varphi_{\to t}\right) \wedge Q(p(n), q_{\text{akz}}).$$

Diese Formel in konjunktiver Normalform besitzt eine Länge von $O(p(n)^3)$ und sie kann für ein gegebenes $x \in \Sigma^*$ in polynomieller Zeit konstruiert werden.

Aus unseren Vorüberlegungen ergibt sich direkt, dass es genau dann eine erfüllende Belegung für die Formel φ gibt, wenn es eine Folge von Konfigurationen $K_0, K_1, \ldots, K_{p(n)}$ der Turingmaschine M mit $K_0 \vdash K_1 \vdash K_2 \vdash \ldots \vdash K_{p(n)}$ gibt, wobei in $K_{p(n)}$ der Zustand q_{akz} angenommen wird und K_0 die initiale Konfiguration bei Eingabe x ist. Dies ist genau dann der Fall, wenn $x \in L$ gilt. Somit gilt $L \leq_p$ SAT. □

4.3 NP-vollständige Probleme

Der Beweis, dass SAT NP-vollständig ist, war relativ lang. Müsste man für jedes Problem einen ähnlichen Beweis führen, so wären sicherlich weit weniger NP-vollständige Probleme bekannt, als das heutzutage der Fall ist. Hat man erst einmal ein erstes NP-schweres Problem L gefunden, so kann man die NP-Schwere eines anderen Problems L' aber auch einfach dadurch nachweisen, dass man L polynomiell auf L' reduziert. Da sich jedes Problem aus NP auf L polynomiell reduzieren lässt, folgt dann aus der Transitivität des Reduktionskonzeptes, dass sich auch jedes Problem aus NP polynomiell auf L' reduzieren lässt. Dies halten wir noch einmal im folgenden Lemma fest.

Lemma 4.19. *Sei L' eine beliebige Sprache und sei L eine beliebige* NP*-schwere Sprache. Gilt $L \leq_p L'$, so ist auch L'* NP*-schwer.*

Genauso wie bei Many-One-Reduktionen, die wir in Abschnitt 3.3 kennengelernt haben, ist es auch bei Lemma 4.19 wichtig, die Reduktion in die richtige Richtung anzuwenden. Möchten wir zeigen, dass eine Sprache L' NP-schwer ist, so müssen wir eine andere NP-schwere Sprache L auf L' reduzieren und nicht umgekehrt.

Wir nutzen Lemma 4.19 zunächst, um nachzuweisen, dass das Problem Clique NP-schwer ist. Dazu führen wir als Hilfsproblem den Spezialfall von SAT ein, bei dem nur Formeln in KNF als Eingaben erlaubt sind, in denen jede Klausel aus genau drei Literalen besteht. Diesen Spezialfall nennen wir 3-SAT. Wir erlauben allerdings, dass eine Klausel dasselbe Literal mehrfach enthält. Dadurch entspricht 3-SAT de facto dem Spezialfall, in dem jede Klausel höchstens aus drei Literalen besteht.

Wir zeigen als erstes, dass 3-SAT NP-vollständig ist. Aus Theorem 4.18 folgt direkt, dass 3-SAT in NP liegt, denn es handelt sich bei 3-SAT um einen Spezialfall von SAT, bei dem nur noch eine Teilmenge der Eingaben erlaubt ist. Wir zeigen nun, dass dieser Spezialfall immer noch NP-schwer ist. Dies ist nicht klar, denn die Einschränkung auf bestimmte Eingaben könnte das Problem einfacher machen.[1]

Theorem 4.20. *Es gilt* SAT $\leq_p$ 3-SAT.

Beweis. Sei φ eine Eingabe für SAT, also eine Formel in KNF mit Klauseln beliebiger Länge. Wir konstruieren basierend auf φ eine Formel $f(\varphi) = \varphi'$ in KNF, in der jede

[1] Dies wäre zum Beispiel der Fall, wenn wir die Klauseln auf zwei statt drei Literale beschränken würden. Das daraus resultierende Problem 2-SAT ist polynomiell lösbar.

Klausel aus genau drei Literalen besteht. Die Formel φ' soll genau dann erfüllbar sein, wenn die Formel φ erfüllbar ist. Zur Konstruktion von φ' ersetzen wir Klauseln aus φ, deren Länge nicht drei ist, unabhängig von den anderen Klauseln durch eine oder mehrere Klauseln der Länge drei.

Sei $C = \ell_1 \vee \ldots \vee \ell_k$ eine beliebige Klausel in φ, deren Länge k ungleich drei ist. Gilt $k = 1$, so ersetzen wir $C = \ell_1$ in φ' durch die Klausel $\ell_1 \vee \ell_1 \vee \ell_1$. Gilt $k = 2$, so ersetzen wir $C = \ell_1 \vee \ell_2$ in φ' durch $\ell_1 \vee \ell_1 \vee \ell_2$. Durch diese lokalen Ersetzungen erhalten wir eine semantisch äquivalente Formel. Interessanter ist der Fall, dass $k \geq 4$ gilt. In diesem Fall ersetzen wir C in φ' durch eine Menge von $k - 2$ Klauseln. Wir führen dafür $k - 3$ neue Variablen $y_1^C, \ldots, y_{k-3}^C$ ein. Diese treten nur bei der lokalen Ersetzung von C auf. Bei der Ersetzung anderer Klauseln werden davon verschiedene Variablen erzeugt. Wir ersetzen C in φ' durch die folgenden Klauseln:

- $\ell_1 \vee \ell_2 \vee y_1^C$ (Typ 1),
- $\neg y_{i-2}^C \vee \ell_i \vee y_{i-1}^C$ für alle $i \in \{3, \ldots, k-2\}$ (Typ 2),
- $\neg y_{k-3}^C \vee \ell_{k-1} \vee \ell_k$ (Typ 3).

Wir ersetzen also beispielhaft eine Klausel $C = \ell_1 \vee \ell_2 \vee \ell_3 \vee \ell_4$ durch

$$(\ell_1 \vee \ell_2 \vee y_1^C) \wedge (\neg y_1^C \vee \ell_3 \vee \ell_4)$$

und eine Klausel $C = \ell_1 \vee \ell_2 \vee \ell_3 \vee \ell_4 \vee \ell_5$ durch

$$(\ell_1 \vee \ell_2 \vee y_1^C) \wedge (\neg y_1^C \vee \ell_3 \vee y_2^C) \wedge (\neg y_2^C \vee \ell_4 \vee \ell_5).$$

Die gerade beschriebenen lokalen Transformationen der Klauseln mit Länge ungleich drei können effizient durchgeführt werden, d. h. die Formel $f(\varphi) = \varphi'$ kann in polynomieller Zeit aus der Formel φ erzeugt werden. Es bleibt zu zeigen, dass φ' genau dann erfüllbar ist, wenn dies für φ der Fall ist.

Sei zunächst angenommen, dass φ erfüllbar ist, und sei x^* eine erfüllende Belegung für φ. Wir zeigen, dass es dann auch eine erfüllende Belegung für φ' gibt. Die Formel φ' enthält dieselben Variablen wie die Formel φ ergänzt um die zusätzlichen Variablen der Form y_i^C, die wir oben eingeführt haben. Wir übernehmen die Belegung x^* der Variablen aus φ und zeigen, dass wir die neuen Variablen der Form y_i^C so belegen können, dass jede Klausel von φ' erfüllt ist. Nichttrivial ist das nur für Klauseln C mit Länge mindestens vier. Sei $C = \ell_1 \vee \ldots \vee \ell_k$ mit $k \geq 4$ eine solche Klausel. Dann erfüllt die Belegung x^* mindestens ein Literal aus C. Sei dies ℓ_j für ein $j \in \{1, \ldots, k\}$.

- Gilt $j \in \{1, 2\}$, so setzen wir alle Variablen y_i^C auf 0. Dann sind alle Klauseln vom Typ 2 und Typ 3 erfüllt, da sie jeweils ein negatives Literal der Form $\neg y_i^C$ enthalten, und die Klausel vom Typ 1 ist erfüllt, da entweder ℓ_1 oder ℓ_2 erfüllt ist.
- Gilt $j \in \{k-1, k\}$, so setzen wir alle Variablen y_i^C auf 1. Dann sind alle Klauseln vom Typ 1 und Typ 2 erfüllt, da sie jeweils ein positives Literal der Form y_i^C enthalten, und die Klausel vom Typ 3 ist erfüllt, da entweder ℓ_{k-1} oder ℓ_k erfüllt ist.

- Gilt $j \notin \{1, 2, k-1, k\}$, so setzen wir die Variablen $y_1^C, \ldots, y_{j-2}^C$ auf 1 und die Variablen $y_{j-1}^C, \ldots, y_{k-3}^C$ auf 0. Dann ist die Klausel $\ell_1 \vee \ell_2 \vee y_1^C$ vom Typ 1 erfüllt, da $y_1^C = 1$ gilt. Außerdem ist für jedes $i \in \{3, \ldots, j-1\}$ die Klausel $\neg y_{i-2}^C \vee \ell_i \vee y_{i-1}^C$ vom Typ 2 erfüllt, da $y_{i-1}^C = 1$ gilt. Die Klausel $\neg y_{j-2}^C \vee \ell_j \vee y_{j-1}^C$ ist ebenfalls erfüllt, da sie das erfüllte Literal ℓ_j enthält. Schließlich ist auch für jedes $i \in \{j+1, \ldots, k-2\}$ die Klausel $\neg y_{i-2}^C \vee \ell_i \vee y_{i-1}^C$ vom Typ 2 erfüllt, da $y_{i-2}^C = 0$ gilt. Ebenso ist die Klausel $\neg y_{k-3}^C \vee \ell_{k-1} \vee \ell_k$ vom Typ 3 erfüllt, da $y_{k-3}^C = 0$ gilt.

Damit haben wir gezeigt, dass wir eine erfüllende Belegung der Variablen aus der Formel φ zu einer erfüllenden Belegung der Variablen aus der Formel φ' erweitern können. Ist φ erfüllbar, so ist also auch φ' erfüllbar.

Nun müssen wir noch die andere Richtung zeigen. Sei (x^*, y^*) eine erfüllende Belegung der Formel φ', wobei x^* die Belegung der Variablen bezeichne, die auch in φ vorkommen, und y^* die Belegung der Variablen, die ausschließlich in φ' enthalten sind. Wir zeigen, dass x^* eine erfüllende Belegung der Formel φ ist. Betrachten wir dazu eine Klausel C aus φ. Der einzige nichttriviale Fall ist wieder der, dass $C = \ell_1 \vee \ldots \vee \ell_k$ für $k \geq 4$ gilt. Wir müssen zeigen, dass die Belegung x^* mindestens eins der Literale $\ell_1, \ldots, \ell_k$ erfüllt. Nehmen wir an, dies sei nicht der Fall. Da es sich bei (x^*, y^*) um eine erfüllende Belegung für φ' handelt, sind die Klauseln vom Typ 1, Typ 2 und Typ 3, die aus C hervorgehen, alle erfüllt. Betrachten wir die Klausel $\ell_1 \vee \ell_2 \vee y_1^C$ vom Typ 1. Da laut unserer Annahme weder ℓ_1 noch ℓ_2 erfüllt ist, muss $(y^*)_1^C = 1$ gelten. Betrachten wir nun die erste Klausel $\neg y_1^C \vee \ell_3 \vee y_2^C$ vom Typ 2. Wegen $(y^*)_1^C = 1$ und der Annahme, dass ℓ_3 nicht erfüllt ist, muss $(y^*)_2^C = 1$ gelten. Analog kann man argumentieren, dass $(y^*)_1^C = (y^*)_2^C = \ldots = (y^*)_{k-3}^C = 1$ gelten muss. Dann folgt aber, dass die Klausel $\neg y_{k-3}^C \vee \ell_{k-1} \vee \ell_k$ vom Typ 3 nicht erfüllt ist, da $(y^*)_{k-3}^C = 1$ gilt und laut unserer Annahme weder ℓ_{k-1} noch ℓ_k erfüllt ist. Dies ist ein Widerspruch dazu, dass (x^*, y^*) eine erfüllende Belegung von φ' ist. Somit ist gezeigt, dass die Belegung x^* die Formel φ erfüllt. Ist φ' erfüllbar, so ist also auch φ erfüllbar. □

Wir wissen bereits aus Theorem 4.7, dass Clique in NP liegt. Mit einer polynomiellen Reduktion von 3-SAT zeigen wir nun noch, dass es sogar NP-vollständig ist.

Theorem 4.21. *Es gilt* 3-SAT $\leq_p$ Clique.

Beweis. Es sei $\varphi = \bigwedge_{i=1}^m (\bigvee_{j=1}^3 \ell_{i,j})$ eine Eingabe für 3-SAT mit m Klauseln, die jeweils drei Literale enthalten. Die Variablen, die in φ vorkommen, seien $x_1, \ldots, x_n$, d. h. jedes Literal $\ell_{i,j}$ stammt aus der Menge $\{x_1, \neg x_1, \ldots, x_n, \neg x_n\}$.

Wir erzeugen ausgehend von φ eine Eingabe $f(\varphi) = (G, k)$ für Clique. Der Graph $G = (V, E)$ enthält $3m$ viele Knoten und zwar einen Knoten (i, j) für jedes Literal $\ell_{i,j}$. Zwei Knoten (i, j) und (i', j') sind durch eine Kante verbunden, wenn $i \neq i'$ und $\ell_{i,j} \neq \neg \ell_{i',j'}$ gilt. Das bedeutet zwei Knoten werden genau dann miteinander verbunden, wenn sie Literale aus verschiedenen Klauseln darstellen, die beide gleichzeitig erfüllt werden können. Knoten aus derselben Klausel werden nicht durch Kanten verbunden. Ebenso werden Knoten nicht miteinander verbunden, wenn sie die Literale x_j und $\neg x_j$ für ein j darstellen. Alle anderen möglichen Kanten werden in G eingefügt. Wir setzen $k = m$.

Wir müssen zeigen, dass es in G genau dann eine k-Clique gibt, wenn die Formel φ erfüllbar ist. Sei dafür zunächst eine erfüllende Belegung x^* von φ gegeben. Diese erfüllt in jeder Klausel mindestens ein Literal. Wir wählen aus jeder Klausel ein beliebiges erfülltes Literal aus und fügen den entsprechenden Knoten der Menge V' hinzu. Dann enthält V' per Konstruktion $k = m$ viele Knoten. Ferner ist V' eine Clique, denn es gibt in V' keine Knoten, die zu derselben Klausel gehören, und es kann ebenfalls nicht sein, dass V' zwei Knoten enthält, die die Literale x_j und $\neg x_j$ für ein j darstellen, da diese Literale nicht beide gleichzeitig erfüllt sein können. Damit haben wir gezeigt, dass in G eine k-Clique existiert, wenn φ erfüllbar ist.

Sei nun umgekehrt eine k-Clique $V' \subseteq V$ in G gegeben. Da es sich um eine Clique handelt, kann V' keine zwei Knoten enthalten, die Literale aus derselben Klausel darstellen. Wegen $k = m$ folgt damit, dass V' für jede Klausel genau einen Knoten enthält, der ein Literal aus der Klausel darstellt. Ferner enthält V' für keine Variable x_j zwei Knoten, die die Literale x_j und $\neg x_j$ darstellen. Wir erhalten somit eine erfüllende Belegung für φ, indem wir alle Variablen x_j, für die das Literal x_j in V' enthalten ist, auf 1 setzen, und alle anderen auf 0. Damit ist gezeigt, dass φ erfüllbar ist, wenn G eine k-Clique enthält. □

Aus den polynomiellen Reduktionen, die wir bisher in der Vorlesung kennengelernt haben, folgt gemeinsam mit Theorem 4.18 bereits, dass einige Probleme NP-vollständig sind. Wir haben insbesondere die Reduktionen SAT $\leq_p$ 3-SAT, 3-SAT $\leq_p$ Clique und Clique $\leq_p$ VC gezeigt. Daraus folgt, dass die Probleme 3-SAT, Clique und VC NP-schwer sind. Man sieht leicht, dass diese Probleme alle zu NP gehören. Somit sind sie sogar NP-vollständig. In den Übungen lernen wir weitere Beispiele für NP-vollständige Probleme kennen.

Ein Problem, mit dem wir uns in dieser Vorlesung schon beschäftigt haben, das aber in der obigen Liste noch fehlt, ist das Rucksackproblem. Wir werden nun zeigen, dass dieses Problem und einige verwandte Probleme ebenfalls NP-schwer sind. Als erstes betrachten wir das Problem SubsetSum. Die Eingabe bei diesem Problem besteht aus Zahlen $a_1, \ldots, a_n \in \mathbb{N}$ sowie einer Zahl $b \in \mathbb{N}$ und es soll entschieden werden, ob es eine Teilmenge $I \subseteq \{1, \ldots, n\}$ gibt, für die $\sum_{i \in I} a_i = b$ gilt. Wieder geben wir keine konkrete Codierung von Eingaben für dieses Problem an. Wichtig ist lediglich, dass die Eingabelänge nur logarithmisch mit der Größe der Zahlen wächst. Dies erreicht man beispielsweise dadurch, dass man die Zahlen binär codiert.

Theorem 4.22. *Das Problem* SubsetSum *ist* NP*-vollständig.*

Beweis. Man sieht leicht ein, dass SubsetSum zu NP gehört, denn für eine gegebene Menge I kann effizient überprüft werden, ob $\sum_{i \in I} a_i = b$ gilt.

Wir zeigen nun, dass 3-SAT $\leq_p$ SubsetSum gilt. Sei dazu φ eine beliebige Eingabe für 3-SAT, also eine Formel in konjunktiver Normalform, in der jede Klausel maximal drei Literale enthält. Es seien $x_1, \ldots, x_N$ die Variablen, die in φ vorkommen, und es seien $C_1, \ldots, C_M$ die Klauseln von φ. Wir erzeugen aus φ eine Eingabe für SubsetSum mit $n = 2N{+}2M$ Zahlen (b nicht mitgezählt), in der jede Zahl mit $N{+}M$ Ziffern im Dezimalsystem codiert werden kann. Für eine Zahl z und einen Index $j \in \{1, \ldots, N{+}M\}$ bezeichnen wir im Folgenden mit $z(j)$ die j-te Ziffer der Zahl z in Dezimaldarstellung.

- Für jede Variable x_i mit $i \in \{1, \dots N\}$ erzeugen wir zwei Zahlen a_i und $\overline{a_i}$. Für $j \in \{1, \dots, N\}$ setzen wir

$$a_i(j) = \overline{a_i}(j) = \begin{cases} 1 & \text{falls } i = j, \\ 0 & \text{falls } i \neq j. \end{cases}$$

  Für $j \in \{1, \dots, M\}$ setzen wir

$$a_i(N+j) = \begin{cases} 1 & \text{falls } x_i \text{ in } C_j \text{ enthalten ist,} \\ 0 & \text{sonst} \end{cases}$$

  und

$$\overline{a_i}(N+j) = \begin{cases} 1 & \text{falls } \neg x_i \text{ in } C_j \text{ enthalten ist,} \\ 0 & \text{sonst.} \end{cases}$$

- Für jedes $i \in \{1, \dots, M\}$ erzeugen wir zwei Zahlen h_i und h'_i. Es sei $h_i(N+i) = h'_i(N+i) = 1$ und alle anderen Ziffern seien 0.

- Wir setzen

$$b(j) = \begin{cases} 1 & \text{falls } j \leq N, \\ 3 & \text{falls } j > N. \end{cases}$$

In dieser Reduktion spielt die Reihenfolge der Ziffern keine Rolle, da es bei der Addition der erzeugten Zahlen zu keinen Überläufen kommen kann. Dies liegt daran, dass es für jede Position maximal fünf Zahlen (b nicht mitgezählt) gibt, die an der Position eine 1 enthalten. Abbildung 4.2 stellt diese Reduktion noch einmal schematisch dar.

Die Zahlen, die wir konstruiert haben, sind sehr groß, da sie aus jeweils $N + M$ Ziffern im Dezimalsystem bestehen und somit Werte in der Größenordnung von 10^{N+M} annehmen können. Binär (oder dezimal) codiert können wir diese Zahlen dennoch in polynomieller Zeit aus der gegebenen Formel φ erzeugen.

Es bleibt zu zeigen, dass es genau dann eine Teilmenge I der Zahlen $a_i, \overline{a_i}, h_i, h'_i$ gibt, deren Summe genau b ist, wenn es eine erfüllende Belegung für φ gibt.

- Sei $x^\star$ eine erfüllende Belegung für φ. Wir konstruieren daraus eine Teilmenge I mit der gewünschten Summe. Gilt $x^\star_i = 1$, so nehmen wir a_i in die Teilmenge I auf. Gilt $x^\star_i = 0$, so nehmen wir $\overline{a_i}$ in die Teilmenge I auf.

  Bezeichnen wir mit A die Summe der bisher ausgewählten Zahlen, so gilt $A(j) = 1$ für alle $j \in \{1, \dots, N\}$, da wir entweder a_j oder $\overline{a_j}$ ausgewählt haben. Ferner folgt aus der Konstruktion, dass $A(N+j)$ für alle $j \in \{1, \dots, M\}$ angibt, wie viele Literale der Klausel C_j von der Belegung $x^\star$ erfüllt werden. Dementsprechend gilt $A(N+j) \in \{1, 2, 3\}$ für alle $j \in \{1, \dots, M\}$. Falls $A(N+j) = 2$ gilt, so fügen wir der Menge I noch die Zahl h_j hinzu. Gilt $A(N+j) = 1$, so fügen wir der Menge I noch die Zahlen h_j und h'_j hinzu. Danach hat die j-te Ziffer der Summe den Wert 3. Alle anderen Ziffern werden durch die Hinzunahme von h_j und h'_j nicht verändert.

  Insgesamt haben wir so eine Teilmenge der Zahlen konstruiert, deren Summe gleich b ist.

	1	2	3	$\cdots$	N	$N+1$	$N+2$	$\cdots$	$N+M$
a_1	**1**	0	0	$\cdots$	0	**1**	0	$\cdots$	$\cdots$
$\overline{a_1}$	**1**	0	0	$\cdots$	0	0	0	$\cdots$	$\cdots$
a_2	0	**1**	0	$\cdots$	0	0	**1**	$\cdots$	$\cdots$
$\overline{a_2}$	0	**1**	0	$\cdots$	0	**1**	0	$\cdots$	$\cdots$
a_3	0	0	**1**	$\cdots$	0	**1**	**1**	$\cdots$	$\cdots$
$\overline{a_3}$	0	0	**1**	$\cdots$	0	0	0	$\cdots$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
a_N	0	0	0	$\cdots$	**1**	0	0	$\cdots$	$\cdots$
$\overline{a_N}$	0	0	0	$\cdots$	**1**	0	**1**	$\cdots$	$\cdots$
h_1	0	0	0	$\cdots$	0	**1**	0	$\cdots$	0
h'_1	0	0	0	$\cdots$	0	**1**	0	$\cdots$	0
h_2	0	0	0	$\cdots$	0	0	**1**	$\cdots$	0
h'_2	0	0	0	$\cdots$	0	0	**1**	$\cdots$	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
h_M	0	0	0	$\cdots$	0	0	0	$\cdots$	**1**
h'_M	0	0	0	$\cdots$	0	0	0	$\cdots$	**1**
b	**1**	**1**	**1**	$\cdots$	**1**	**3**	**3**	$\cdots$	**3**

Abbildung 4.2: Schematische Darstellung der Reduktion 3-SAT $\leq_p$ SubsetSum: In diesem Beispiel gilt $C_1 = x_1 \vee \neg x_2 \vee x_3$ und $C_2 = x_2 \vee x_3 \vee \neg x_N$ (übernommen aus [6]).

- Sei nun andersherum I eine Teilmenge der Zahlen, deren Summe gleich b ist. Wir konstruieren daraus eine erfüllende Belegung $x^\star$ für φ. Zunächst beobachten wir, dass für jedes $j \in \{1, \ldots, N\}$ genau eine der Zahlen a_j und $\overline{a_j}$ in I enthalten ist, da sonst die j-te Ziffer der Summe der Zahlen aus I nicht gleich $b(j) = 1$ sein könnte. Wir setzen

$$x_j^\star = \begin{cases} 1 & \text{falls } a_j \text{ in } I \text{ enthalten ist,} \\ 0 & \text{falls } \overline{a_j} \text{ in } I \text{ enthalten ist.} \end{cases}$$

  Um zu sehen, dass $x^\star$ eine erfüllende Belegung von φ ist, betrachten wir die Summe A der Zahlen a_i und $\overline{a_i}$, die in I enthalten sind. Für jedes $j \in \{1, \ldots, M\}$ gilt $A(N+j) \geq 1$, da ansonsten die $(N+j)$-te Ziffer der Summe der Zahlen aus I nicht gleich $b(N+j) = 3$ sein könnte. Da $a_i(N+j) = 1$ oder $\overline{a_i}(N+j) = 1$ nur dann gilt, wenn C_j das Literal x_j bzw. $\neg x_j$ enthält, folgt daraus, dass die Belegung $x^\star$ mindestens ein Literal jeder Klausel erfüllt.

Damit ist der Beweis abgeschlossen. □

Wir betrachten nun das Problem Partition. Die Eingabe bei diesem Problem besteht aus Zahlen $a_1, \ldots, a_n \in \mathbb{N}$ und es soll entschieden werden, ob es eine Teilmenge $I \subseteq \{1, \ldots, n\}$ gibt, für die $\sum_{i \in I} a_i = \sum_{i \in \{1,\ldots,n\} \setminus I} a_i$ gilt. Bei diesem Problem handelt es sich also um den Spezialfall von SubsetSum, bei dem per Definition $b = \frac{1}{2} \sum_{i=1}^{n} a_i$ gilt.

Theorem 4.23. *Das Problem* Partition *ist* NP*-vollständig.*

Beweis. Da Partition ein Spezialfall von SubsetSum ist und wir bereits gezeigt haben, dass SubsetSum $\in$ NP gilt, folgt direkt Partition $\in$ NP.

Wir zeigen nun, dass SubsetSum $\leq_p$ Partition gilt, woraus mit Theorem 4.22 folgt, dass Partition NP-schwer ist. Sei $a_1, \ldots, a_n \in \mathbb{N}, b \in \mathbb{N}$ eine Instanz für SubsetSum. Wir definieren $A := \sum_{i=1}^{n} a_i$ und konstruieren eine Eingabe für Partition, die aus den Zahlen $a'_1, \ldots, a'_{n+2}$ besteht, die wie folgt definiert sind.

- Für $i \in \{1, \ldots, n\}$ sei $a'_i = a_i$.
- Es sei $a'_{n+1} = 2A - b$.
- Es sei $a'_{n+2} = A + b$.

Diese Reduktion kann in polynomieller Zeit berechnet werden. Es bleibt lediglich zu zeigen, dass es genau dann eine Aufteilung der Zahlen $a'_1, \ldots, a'_{n+2}$ in zwei Mengen mit derselben Summe gibt, wenn es eine Menge $I \subseteq \{1, \ldots, n\}$ mit $\sum_{i \in I} a_i = b$ gibt. Wir halten zunächst fest, dass $\sum_{i=1}^{n+2} a'_i = 4A$ gilt, und zeigen nun die beiden Richtungen der Äquivalenz.

- Sei $J \subseteq \{1, \ldots, n+2\}$ eine Menge mit $\sum_{i \in J} a'_i = \sum_{i \in \{1,\ldots,n+2\} \setminus J} a'_i = 2A$. Wegen $\sum_{i=1}^{n} a'_i = A < 2A$ gilt entweder $(n+1) \in J$ oder $(n+2) \in J$. Es können aber nicht sowohl $n+1$ als auch $n+2$ zu J gehören, da $a'_{n+1} + a'_{n+2} = 3A > 2A$ gilt. Wir betrachten zunächst den Fall $(n+1) \in J$ und setzen $I = J \setminus \{n+1\}$. Dann gilt
$$2A = \sum_{i \in J} a'_i = \sum_{i \in I} a'_i + a'_{n+1} = \sum_{i \in I} a'_i + 2A - b,$$
woraus $\sum_{i \in I} a'_i = \sum_{i \in I} a_i = b$ folgt. Die Menge I ist also eine Lösung für die gegebene Eingabe von SubsetSum. Wir betrachten nun den Fall $(n+2) \in J$ und setzen $I = J \setminus \{n+2\}$. Dann gilt
$$2A = \sum_{i \in J} a'_i = \sum_{i \in I} a'_i + a'_{n+2} = \sum_{i \in I} a'_i + A + b,$$
woraus $\sum_{i \in I} a'_i = \sum_{i \in I} a_i = A - b$ folgt. Demnach gilt
$$\sum_{i \in \{1,\ldots,n\} \setminus I} a_i = A - \sum_{i \in I} a_i = b$$
und somit ist die Menge $\{1, \ldots, n\} \setminus I$ eine Lösung für die gegebene Eingabe von SubsetSum.
- Sei nun umgekehrt eine Menge $I \subseteq \{1, \ldots, n\}$ mit $\sum_{i \in I} a_i = b$ gegeben. Für $J = I \cup \{n+1\}$ gilt
$$\sum_{i \in J} a'_i = \sum_{i \in I} a'_i + a'_{n+1} = \sum_{i \in I} a_i + a'_{n+1} = b + (2A - b) = 2A.$$
Dementsprechend gilt auch $\sum_{i \in \{1,\ldots,n+2\} \setminus J} a'_i = 4A - \sum_{i \in J} a'_i = 2A$ und damit ist J eine Lösung für die konstruierte Instanz von SubsetSum. □

Die NP-Vollständigkeit des Rucksackproblems folgt ebenfalls leicht aus Theorem 4.22.

Theorem 4.24. *Die Entscheidungsvariante des Rucksackproblems ist* NP*-vollständig.*

Beweis. Wir wissen bereits aus Theorem 4.7, dass die Entscheidungsvariante des Rucksackproblems zu NP gehört. Wir zeigen nun SubsetSum $\leq_p$ KP. Sei dazu $a_1, \ldots, a_n \in \mathbb{N}$, $b \in \mathbb{N}$ eine Eingabe für SubsetSum. Wir konstruieren daraus eine Eingabe für das Rucksackproblem mit n Objekten. Für jedes $i \in \{1, \ldots, n\}$ gelte $p_i = w_i = a_i$. Ferner setzen wir $t = z = b$. Gefragt ist dann nach einer Auswahl der Objekte mit Gewicht höchstens $t = b$ und Nutzen mindestens $z = b$.

Sei $I \subseteq \{1, \ldots, n\}$ eine solche Auswahl. Dann gilt $\sum_{i \in I} a_i = \sum_{i \in I} w_i \leq b$ und $\sum_{i \in I} a_i = \sum_{i \in I} p_i \geq b$ und damit $\sum_{i \in I} a_i = b$. Jede Lösung für die Eingabe des Rucksackproblems entspricht also einer Lösung für die Eingabe von SubsetSum. Sei andersherum $I \subseteq \{1, \ldots, n\}$ eine Auswahl mit $\sum_{i \in I} a_i = b$. Dann gilt $\sum_{i \in I} w_i = \sum_{i \in I} a_i \leq b$ und $\sum_{i \in I} p_i = \sum_{i \in I} a_i \geq b$. Demzufolge entspricht umgekehrt jede Lösung für die Eingabe von SubsetSum auch einer Lösung für die Eingabe des Rucksackproblems. □

Wir haben bereits am Anfang dieses Kapitels diskutiert, dass der bekannte Algorithmus, der das Rucksackproblem mithilfe dynamischer Programmierung in Zeit $O(N^2W)$ für $W = \max_i w_i$ löst, kein polynomieller Algorithmus ist, da die Laufzeit linear von W, die Eingabelänge aber nur logarithmisch von W abhängt. Algorithmen, deren Laufzeiten polynomiell von der Eingabelänge und den in der Eingabe vorkommenden Zahlen abhängen, nennt man auch *pseudopolynomiell*. Theorem 4.24 zeigt, dass es unter der Annahme P $\neq$ NP keinen polynomiellen Algorithmus für das Rucksackproblem gibt. Das dynamische Programm zeigt allerdings, dass das Rucksackproblem nur dann schwer sein kann, wenn die Gewichte sehr groß sind. Genau das ist in der Reduktion auch der Fall, denn die Zahlen, die in Theorem 4.22, auf das die NP-Vollständigkeit des Rucksackproblems zurückgeht, konstruiert werden, sind bezogen auf die Eingabelänge exponentiell groß.

NP-schwere Probleme, in deren Eingaben Zahlen vorkommen und die für Eingaben mit polynomiell großen Zahlen polynomiell lösbar sind, nennt man *schwach* NP*-schwere Probleme*. Sowohl das Rucksackproblem als auch SubsetSum und Partition sind schwach NP-schwer. Im Gegensatz dazu ist das TSP beispielsweise *stark* NP*-schwer*, da es bereits dann NP-schwer ist, wenn als Abstände nur die Zahlen 1 und 2 zugelassen sind.

Unter der Annahme P $\neq$ NP gibt es für NP-schwere Probleme keine Algorithmen mit polynomieller Laufzeit. Dies bedeutet jedoch nicht zwangsläufig, dass exponentielle Rechenzeit benötigt wird, um diese Probleme zu lösen. Denkbar wären auch Algorithmen mit einer Laufzeit, die weder polynomiell noch exponentiell ist, also beispielsweise ein Algorithmus für SAT mit einer Laufzeit von $O(n^{\log n})$, wobei n die Anzahl an aussagenlogischen Variablen in der Eingabe bezeichnet. Solche Algorithmen sind allerdings nicht bekannt und es gibt stärkere Annahmen als P $\neq$ NP, die die Existenz solcher Algorithmen ausschließen. Die (unbewiesene) *Exponentialzeithypothese* besagt beispielsweise, dass es für 3-SAT eine Konstante $\delta > 0$ gibt, sodass SAT nicht in Zeit $O(2^{\delta n})$ gelöst werden kann.

Kapitel 5

Approximationsalgorithmen

Wenn wir für ein Optimierungsproblem gezeigt haben, dass es NP-schwer ist, dann bedeutet das zunächst nur, dass es unter der Annahme P $\neq$ NP keinen effizienten Algorithmus gibt, der für jede Instanz eine optimale Lösung berechnet. Das schließt nicht aus, dass es einen effizienten Algorithmus gibt, der für jede Instanz eine Lösung berechnet, die fast optimal ist. Um das formal zu fassen, betrachten wir zunächst genauer, was ein *Optimierungsproblem* eigentlich ist, und definieren anschließend die Begriffe *Approximationsalgorithmus* und *Approximationsgüte*.

Ein *Optimierungsproblem* Π besteht aus einer Menge $\mathcal{I}_\Pi$ von *Instanzen* oder *Eingaben*. Zu jeder Instanz $I \in \mathcal{I}_\Pi$ gehört eine Menge $\mathcal{S}_I$ von *Lösungen* und eine *Zielfunktion* $f_I : \mathcal{S}_I \to \mathbb{R}_{\geq 0}$, die jeder Lösung einen reellen Wert zuweist. Zusätzlich ist bei einem Optimierungsproblem vorgegeben, ob wir eine Lösung x mit *minimalem* oder mit *maximalem Wert* $f_I(x)$ suchen. Wir bezeichnen in jedem Fall mit $\mathrm{OPT}(I)$ den Wert einer optimalen Lösung. Wenn die betrachtete Instanz I klar aus dem Kontext hervorgeht, so schreiben wir oft einfach OPT statt $\mathrm{OPT}(I)$.

Beispiel: Spannbaumproblem

Eine Instanz I des *Spannbaumproblems* wird durch einen ungerichteten Graphen $G = (V, E)$ und Kantengewichte $c : E \to \mathbb{N}$ beschrieben. Die Menge $\mathcal{S}_I$ der Lösungen für eine solche Instanz ist die Menge aller Spannbäume des Graphen G. Die Funktion f_I weist jedem Spannbaum $T \in \mathcal{S}_I$ sein Gewicht zu, also $f_I(T) = \sum_{e \in T} c(e)$, und wir möchten f_I minimieren. Es gilt $\mathrm{OPT}(I) = \min_{T \in \mathcal{S}_I} f_I(T)$.

Ein *Approximationsalgorithmus* A für ein Optimierungsproblem Π ist zunächst lediglich ein Polynomialzeitalgorithmus, der zu jeder Instanz I eine Lösung aus $\mathcal{S}_I$ ausgibt. Wir bezeichnen mit $A(I)$ die Lösung, die Algorithmus A bei Eingabe I ausgibt, und wir bezeichnen mit $w_A(I)$ ihren Wert, also $w_A(I) = f_I(A(I))$. Je näher der Wert $w_A(I)$ dem optimalen Wert $\mathrm{OPT}(I)$ ist, desto besser ist der Approximationsalgorithmus.

Definition 5.1 (Approximationsfaktor/Approximationsgüte)**.** *Ein Approximationsalgorithmus A für ein Minimierungs- bzw. Maximierungsproblem Π erreicht einen* Ap-

proximationsfaktor *oder eine* Approximationsgüte *von* $r \geq 1$ *bzw.* $r \leq 1$*, wenn*

$$w_A(I) \leq r \cdot \mathrm{OPT}(I) \quad \textit{bzw.} \quad w_A(I) \geq r \cdot \mathrm{OPT}(I)$$

für alle Instanzen $I \in \mathcal{I}_\Pi$ *gilt. Wir sagen dann, dass* A *ein* r-Approximationsalgorithmus *ist.*

Haben wir beispielsweise für ein Minimierungsproblem einen 2-Approximationsalgorithmus, so bedeutet das, dass der Algorithmus für jede Instanz eine Lösung berechnet, deren Wert höchstens doppelt so groß ist wie der optimale Wert. Haben wir einen $\frac{1}{2}$-Approximationsalgorithmus für ein Maximierungsproblem, so berechnet der Algorithmus für jede Instanz eine Lösung, deren Wert mindestens halb so groß ist wie der optimale Wert.

Ein Polynomialzeitalgorithmus für ein Problem, der stets eine optimale Lösung berechnet, ist ein 1-Approximationsalgorithmus. Ist ein Problem NP-schwer, so schließt dies unter der Voraussetzung P $\neq$ NP lediglich die Existenz eines solchen 1-Approximationsalgorithmus aus. Es gibt aber durchaus NP-schwere Probleme, für die es zum Beispiel 1,01-Approximationsalgorithmen gibt, also effiziente Algorithmen, die stets eine Lösung berechnen, deren Wert um höchstens ein Prozent vom optimalen Wert abweicht.

Wir lernen in diesem Kapitel Approximationsalgorithmen für ein Problem aus dem Bereich Scheduling, für das TSP und das Rucksackproblem kennen.

5.1 Scheduling auf identischen Maschinen

Scheduling-Probleme haben eine große Bedeutung in der Informatik. Es geht dabei darum, Prozessen Ressourcen zuzuteilen. In dem Modell, das wir betrachten, ist eine Menge $J = \{1, \ldots, n\}$ von *Jobs* oder *Prozessen* gegeben, die auf eine Menge $M = \{1, \ldots, m\}$ von *Maschinen* oder *Prozessoren* verteilt werden muss. Jeder Job $j \in J$ besitzt dabei eine *Größe* $p_j \in \mathbb{R}_{>0}$, die angibt, wie viele Zeiteinheiten benötigt werden, um ihn vollständig auszuführen. Ein *Schedule* $\pi : J \to M$ ist eine Abbildung, die jedem Job eine Maschine zuweist, auf der er ausgeführt wird. Wir bezeichnen mit $L_i(\pi)$ die *Ausführungszeit* (oder *Last*) von Maschine $i \in M$ in Schedule π, d. h.

$$L_i(\pi) = \sum_{j \in J : \pi(j) = i} p_j.$$

Der *Makespan* $C(\pi)$ eines Schedules entspricht der Ausführungszeit derjenigen Maschine, die am längsten benötigt, die ihr zugewiesenen Jobs abzuarbeiten, d. h.

$$C(\pi) = \max_{i \in M} L_i(\pi).$$

Wir betrachten den Makespan als Zielfunktion, die es zu minimieren gilt. Wir nennen die hier beschriebene Scheduling-Variante *Scheduling auf identischen Maschinen*, da alle Maschinen dieselbe Geschwindigkeit haben. Dieses Problem ist NP-schwer, was durch eine einfache Reduktion von Partition gezeigt werden kann.

Wir betrachten als erstes den Greedy-Algorithmus Least-Loaded, der die Jobs in der Reihenfolge $1, 2, \ldots, n$ durchgeht und jeden Job einer Maschine zuweist, die die kleinste Ausführungszeit bezogen auf die bereits zugewiesenen Jobs besitzt.

Theorem 5.2. *Der Algorithmus* Least-Loaded *ist ein* $(2 - 1/m)$*-Approximationsalgorithmus für das Problem Scheduling auf identischen Maschinen.*

Beweis. Sei eine beliebige Eingabe gegeben. Der Beweis beruht auf den folgenden beiden unteren Schranken für den Makespan eines optimalen Schedules π^*:

$$C(\pi^*) \geq \frac{1}{m} \sum_{j \in J} p_j \qquad \text{und} \qquad C(\pi^*) \geq \max_{j \in J} p_j.$$

Die erste Schranke folgt aus der Beobachtung, dass die durchschnittliche Ausführungszeit der Maschinen in jedem Schedule gleich $\frac{1}{m} \sum_{j \in J} p_j$ ist. Deshalb muss es in jedem Schedule (also auch in π^*) eine Maschine geben, deren Ausführungszeit mindestens diesem Durchschnitt entspricht. Die zweite Schranke basiert auf der Beobachtung, dass die Maschine, der der größte Job zugewiesen ist, eine Ausführungszeit von mindestens $\max_{j \in J} p_j$ besitzt.

Wir betrachten nun den Schedule π, den der Least-Loaded-Algorithmus berechnet. In diesem Schedule sei $i \in M$ eine Maschine mit größter Ausführungszeit. Es gilt also $C(\pi) = L_i(\pi)$. Es sei $j \in J$ der Job, der als letztes Maschine i hinzugefügt wurde. Zu dem Zeitpunkt, zu dem diese Zuweisung erfolgt ist, war Maschine i eine Maschine mit der kleinsten Ausführungszeit. Nur die Jobs $1, \ldots, j-1$ waren bereits verteilt und dementsprechend muss es eine Maschine gegeben haben, deren Ausführungszeit höchstens dem Durchschnitt $\frac{1}{m} \sum_{k=1}^{j-1} p_k$ entsprach. Wir können den Makespan von π nun wie folgt abschätzen:

$$\begin{aligned}
C(\pi) = L_i(\pi) &\leq \frac{1}{m} \left(\sum_{k=1}^{j-1} p_k \right) + p_j \leq \frac{1}{m} \left(\sum_{k \in J \setminus \{j\}} p_k \right) + p_j \\
&= \frac{1}{m} \left(\sum_{k \in J} p_k \right) + \left(1 - \frac{1}{m} \right) p_j \\
&\leq \frac{1}{m} \left(\sum_{k \in J} p_k \right) + \left(1 - \frac{1}{m} \right) \cdot \max_{k \in J} p_k \\
&\leq C(\pi^*) + \left(1 - \frac{1}{m} \right) \cdot C(\pi^*) = \left(2 - \frac{1}{m} \right) \cdot C(\pi^*),
\end{aligned}$$

wobei wir die beiden oben angegebenen unteren Schranken für $C(\pi^*)$ benutzt haben. □

Das folgende Beispiel zeigt, dass der Least-Loaded-Algorithmus im Worst Case keinen besseren Approximationsfaktor als $(2 - 1/m)$ erreicht.

Untere Schranke für den Least-Loaded-Algorithmus

Sei eine Anzahl m an Maschinen vorgegeben. Wir betrachten eine Instanz mit $n = m(m-1)+1$ vielen Jobs. Die ersten $m(m-1)$ Jobs haben jeweils eine Größe von 1 und der letzte Job hat eine Größe von m. In einer optimalen Lösung werden die ersten $m(m-1)$ Jobs gleichmäßig auf den Maschinen $1, \ldots, m-1$ verteilt und der letzte Job auf Maschine m. Dann haben alle Maschinen eine Ausführungszeit von genau m.

Der Least-Loaded-Algorithmus hingegen verteilt die ersten $m(m-1)$ Jobs gleichmäßig auf den Maschinen $1, \ldots, m$ und platziert den letzten Job auf einer beliebigen Maschine i. Diese Maschine hat dann eine Ausführungszeit von $(m-1)+m = 2m-1$. Es gilt

$$\frac{2m-1}{m} = 2 - \frac{1}{m}.$$

Damit ist gezeigt, dass der Least-Loaded-Algorithmus im Allgemeinen keinen besseren Approximationsfaktor als $2 - 1/m$ erreicht.

Wir zeigen nun, dass für das Problem Scheduling auf identischen Maschinen mit einem modifizierten Algorithmus ein Approximationsfaktor von $4/3$ erreicht werden kann.

Longest-Processing-Time (LPT)

1. Sortiere die Jobs so, dass $p_1 \geq p_2 \geq \ldots \geq p_n$ gilt.
2. Führe den Least-Loaded-Algorithmus auf den so sortierten Jobs aus.

Der einzige Unterschied zwischen dem LPT- und dem Least-Loaded-Algorithmus besteht darin, dass die Jobs nach ihrer Größe sortiert in absteigender Reihenfolge betrachtet werden. Diese Sortierung verbessert den Approximationsfaktor deutlich.

Theorem 5.3. *Der Algorithmus* Longest-Processing-Time *ist ein $\frac{4}{3}$-Approximationsalgorithmus für das Problem Scheduling auf identischen Maschinen.*

Beweis. Wir führen einen Widerspruchsbeweis und nehmen an, es gäbe eine Eingabe mit m Maschinen und n Jobs mit Größen $p_1 \geq \ldots \geq p_n$, auf der der LPT-Algorithmus einen Schedule π berechnet, dessen Makespan mehr als $\frac{4}{3}$-mal so groß ist wie der Makespan OPT des optimalen Schedules π^*. Außerdem gehen wir davon aus, dass wir eine Instanz mit der kleinstmöglichen Anzahl an Jobs gewählt haben, auf der dies der Fall ist.

Es sei nun $i \in M$ eine Maschine, die in Schedule π die größte Ausführungszeit besitzt, und es sei $j \in J$ der letzte Job, der vom LPT-Algorithmus Maschine i zugewiesen wird. Es gilt $j = n$, da ansonsten $p_1, \ldots, p_j$ eine Eingabe mit weniger Jobs ist, auf der der LPT-Algorithmus keine $\frac{4}{3}$-Approximation liefert. Genauso wie im Beweis von Theorem 5.2 können wir argumentieren, dass es zum Zeitpunkt der Zuweisung von

Job n eine Maschine mit Ausführungszeit höchstens $\frac{1}{m}\left(\sum_{k=1}^{n-1} p_k\right) \leq \mathrm{OPT}$ gibt. Da wir Job n der Maschine i mit der bisher kleinsten Ausführungszeit zuweisen, gilt

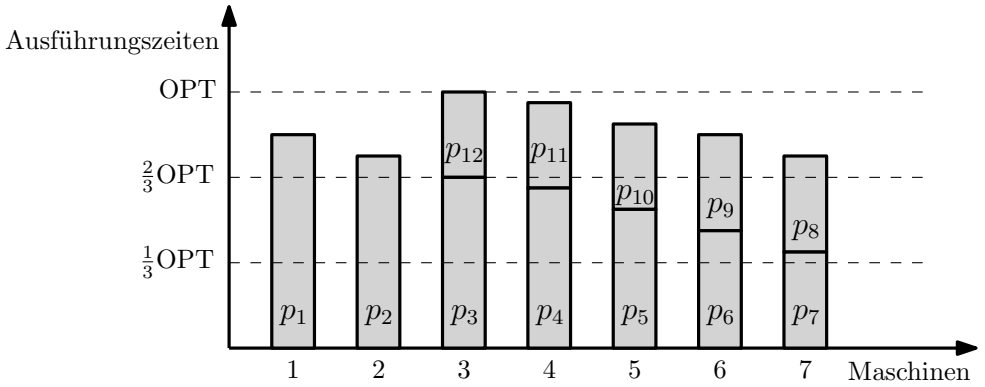
$$C(\pi) = L_i(\pi) \leq \frac{1}{m}\left(\sum_{k=1}^{n-1} p_k\right) + p_n \leq \mathrm{OPT} + p_n.$$

Der Makespan von π kann also nur dann größer als $\frac{4}{3} \cdot \mathrm{OPT}$ sein, wenn $p_n > \mathrm{OPT}/3$ gilt. Wegen der Sortierung der Jobs gilt $p_j > \mathrm{OPT}/3$ dann sogar für alle Jobs $j \in J$.

Das bedeutet, dass in jedem optimalen Schedule jeder Maschine höchstens zwei Jobs zugewiesen sein können. Für Eingaben, die diese Bedingung erfüllen, gilt insbesondere $n \leq 2m$ und man kann einen optimalen Schedule explizit angeben:

- Jeder Job $j \in \{1, \ldots, \min\{n, m\}\}$ wird Maschine j zugewiesen.
- Jeder Job $j \in \{m+1, \ldots, n\}$ wird Maschine $2m - j + 1$ zugewiesen.

Dieser Schedule ist in folgender Abbildung dargestellt. Den Beweis, dass dies wirklich



ein optimaler Schedule ist, falls alle Jobs echt größer als $\mathrm{OPT}/3$ sind, überlassen wir den Leserinnen und Lesern.

Wir zeigen nun, dass der optimale Schedule, den wir gerade beschrieben haben, dem Schedule entspricht, den der LPT-Algorithmus berechnet: Zunächst ist klar, dass die ersten $\min\{n, m\}$ Jobs jeweils einer eigenen Maschine zugewiesen werden, und wir deshalb ohne Beschränkung der Allgemeinheit davon ausgehen können, dass jeder Job $j \in \{1, \ldots, \min\{n, m\}\}$ Maschine j zugewiesen wird. Wir betrachten nun die Zuweisung eines Jobs $j \in \{m+1, \ldots, n\}$ und gehen induktiv davon aus, dass die Jobs aus $\{1, \ldots, j-1\}$ bereits so wie im oben beschriebenen optimalen Schedule zugewiesen wurden. Für eine Maschine $i \in M$ bezeichnen wir mit L_i die Ausführungszeit, die durch die bisher zugewiesenen Jobs verursacht wird. Im optimalen Schedule wird Job j Maschine $2m - j + 1$ zugewiesen. Deshalb gilt $L_{2m-j+1} + p_j \leq \mathrm{OPT}$. Ferner gilt $L_1 \geq L_2 \geq \ldots \geq L_{2m-j+1}$. Alle Maschinen $i \in \{2m - j + 2, \ldots, m\}$ haben bereits zwei Jobs zugewiesen. Da alle Jobs größer als $\frac{1}{3}\mathrm{OPT}$ sind, folgt für diese Maschinen $L_i + p_j > \mathrm{OPT}$ und damit $L_i > L_{2m-j+1}$. Damit ist gezeigt, dass Maschine $2m - j + 1$ bei der Zuweisung von Job j tatsächlich eine Maschine mit kleinster Ausführungszeit ist.

Damit haben wir einen Widerspruch zu der Annahme, dass der LPT-Algorithmus auf der gegebenen Eingabe keine $\frac{4}{3}$-Approximation erreicht. Wir haben gezeigt, dass der LPT-Algorithmus auf Eingaben, in denen alle Jobs echt größer als $\frac{1}{3}$OPT sind, den optimalen Schedule berechnet. Für alle anderen Eingaben haben wir bewiesen, dass der LPT-Algorithmus eine $\frac{4}{3}$-Approximation berechnet. □

5.2 Traveling Salesman Problem

Das Traveling Salesman Problem (TSP) haben wir bereits angesprochen. In diesem Abschnitt werden wir zunächst ein negatives Ergebnis über Approximationsalgorithmen zeigen. Wir zeigen, dass für das allgemeine TSP kein Approximationsalgorithmus existiert, der eine sinnvolle Approximationsgüte erreicht.

Traveling Salesman Problem (TSP)

Eingabe:	Menge $V = \{v_1, \ldots, v_n\}$ von Knoten symmetrische Distanzfunktion $d : V \times V \to \mathbb{R}_{\geq 0}$ (d. h. $\forall u, v \in V : d(u, v) = d(v, u) \geq 0$)
Lösungen:	alle Permutationen $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ eine solche Permutation nennen wir auch *Tour*
Zielfunktion:	minimiere $\sum_{i=1}^{n-1} d\Big(v_{\pi(i)}, v_{\pi(i+1)}\Big) + d\Big(v_{\pi(n)}, v_{\pi(1)}\Big)$

5.2.1 Nichtapproximierbarkeit

Was ist die beste Approximationsgüte, die für das TSP in polynomieller Zeit erreicht werden kann? Wir zeigen, dass es nicht mal einen 2^n-Approximationsalgorithmus für das TSP gibt. Das bedeutet insbesondere, dass es für keine Konstante a einen a-Approximationsalgorithmus gibt.

Theorem 5.4. *Falls* P $\neq$ NP*, so existiert kein* 2^n*-Approximationsalgorithmus für das TSP.*

Beweis. Wir betrachten das Hamiltonkreis-Problem HC. Dies ist das Problem, für einen ungerichteten Graphen zu entscheiden, ob es einen Kreis gibt, der jeden Knoten genau einmal enthält. Ein solcher Kreis wird auch Hamiltonkreis genannt. Man kann durch eine Reduktion von 3-SAT zeigen, dass dieses Problem NP-vollständig ist. Dies nutzen wir als Ausgangspunkt für das TSP. Dazu geben wir eine spezielle polynomielle Reduktion von HC auf TSP an, aus der wir den folgenden Schluss ziehen können: Falls ein 2^n-Approximationsalgorithmus für das TSP existiert, so kann HC in polynomieller Zeit gelöst werden.

Wir müssen uns also fragen, wie wir einen 2^n-Approximationsalgorithmus für das TSP nutzen können, um HC zu lösen. Sei $G = (V, E)$ der Graph, für den wir entscheiden

wollen, ob er einen Hamiltonkreis besitzt. Daraus erzeugen wir eine TSP-Instanz mit der Knotenmenge V und der folgenden Distanzfunktion $d : V \times V \to \mathbb{R}_{>0}$:

$$\forall u, v \in V, u \neq v : d(u, v) = d(v, u) = \begin{cases} 1 & \text{falls } \{u, v\} \in E, \\ n2^{n+1} & \text{falls } \{u, v\} \notin E. \end{cases}$$

Wir haben diese Distanzfunktion so gewählt, dass die konstruierte Instanz für das TSP genau dann eine Tour der Länge n besitzt, wenn der Graph G einen Hamiltonkreis besitzt. Besitzt der Graph G hingegen keinen Hamiltonkreis, so hat jede Tour der TSP-Instanz mindestens Länge $n - 1 + n2^{n+1}$. Die Distanzfunktion d kann in polynomieller Zeit berechnet werden, da die Codierungslänge von $n2^{n+1}$ polynomiell beschränkt ist, nämlich $\Theta(n)$.

Nehmen wir an, wir haben einen 2^n-Approximationsalgorithmus für das TSP. Wenn wir diesen Algorithmus auf einen Graphen G anwenden, der einen Hamiltonkreis enthält, so berechnet er eine Tour mit Länge höchstens $n2^n$, da die Länge der optimalen Tour n ist. Diese Tour kann offenbar keine Kante mit Gewicht $n2^{n+1}$ enthalten. Somit besteht die Tour nur aus Kanten aus der Menge E und ist somit ein Hamiltonkreis von G.

Damit ist gezeigt, dass ein 2^n-Approximationsalgorithmus für das TSP genutzt werden kann, um in polynomieller Zeit einen Hamiltonkreis in einem Graphen zu berechnen, falls ein solcher existiert. □

Die Wahl von 2^n in Theorem 5.4 ist willkürlich. Tatsächlich kann man mit obigem Beweis sogar für jede in Polynomialzeit berechenbare Funktion $r : \mathbb{N} \to \mathbb{R}_{\geq 1}$ zeigen, dass es keinen $r(n)$-Approximationsalgorithmus für das TSP gibt, falls P $\neq$ NP.

Die Technik, die wir im letzten Beweis genutzt haben, lässt sich auch für viele andere Probleme einsetzen. Wir fassen sie noch einmal zusammen: Um zu zeigen, dass es unter der Annahme P $\neq$ NP für ein Problem keinen r-Approximationsalgorithmus gibt, zeigt man, dass man mithilfe eines solchen Algorithmus ein NP-schweres Problem in polynomieller Zeit lösen kann.

5.2.2 Metrisches TSP

Es gibt viele Möglichkeiten, mit Optimierungsproblemen umzugehen, für die es nicht mal gute Approximationsalgorithmen gibt. Man sollte sich die Frage stellen, ob man das Problem zu allgemein formuliert hat. Oftmals erfüllen Eingaben, die in der Praxis auftreten, nämlich Zusatzeigenschaften, die das Problem einfacher machen. Eine Eigenschaft, die bei vielen Anwendungen gegeben ist, ist die Dreiecksungleichung. Das bedeutet, dass

$$d(u, w) \leq d(u, v) + d(v, w)$$

für alle Knoten $u, v, w \in V$ gilt. Möchte man von einem Knoten zu einem anderen, so ist also der direkte Weg nie länger als der zusammengesetzte Weg über einen Zwischenknoten. Sind die Knoten zum Beispiel Punkte im euklidischen Raum und ist die Distanz zweier Punkte durch ihren euklidischen Abstand gegeben, so ist diese Eigenschaft erfüllt.

Definition 5.5. *Sei X eine Menge und $d : X \times X \to \mathbb{R}_{\geq 0}$ eine Funktion. Die Funktion d heißt* Metrik auf X*, wenn die folgenden drei Eigenschaften erfüllt sind.*

- $\forall x, y \in X : d(x, y) = 0 \iff x = y$ (positive Definitheit)
- $\forall x, y \in X : d(x, y) = d(y, x)$ (Symmetrie)
- $\forall x, y, z \in X : d(x, z) \leq d(x, y) + d(y, z)$ (Dreiecksungleichung)

Das Paar (X, d) heißt metrischer Raum.

Wir interessieren uns nun für Instanzen des TSP, bei denen d eine Metrik auf V ist. Diesen Spezialfall des allgemeinen TSP nennen wir *metrisches* TSP. Wir zeigen, dass sich dieser Spezialfall deutlich besser approximieren lässt als das allgemeine Problem. Zunächst halten wir aber fest, dass das metrische TSP noch NP-schwer ist. Dies folgt, da das TSP bereits dann NP-schwer ist, wenn alle Distanzen zwischen verschiedenen Knoten entweder 1 oder 2 sind. Die Leserinnen und Leser mögen sich überlegen, dass die Dreiecksungleichung automatisch erfüllt ist, wenn nur 1 und 2 als Distanzen erlaubt sind. Somit ist gezeigt, dass auch das metrische TSP noch NP-schwer ist.

In folgendem Algorithmus benutzen wir den Begriff eines *Eulerkreises*. Dabei handelt es sich um einen Kreis in einem Graphen, der jede Kante des Graphen genau einmal enthält, der Knoten aber mehrfach besuchen darf. Ein Graph, der einen solchen Kreis enthält, wird auch *eulerscher Graph* genannt (siehe Abbildung 5.1).

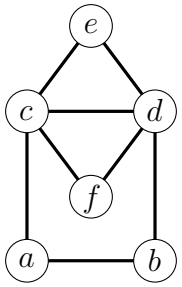


Abbildung 5.1: Dieser Graph enthält den Eulerkreis $(a, c, e, d, f, c, d, b, a)$.

Das Problem, einen Eulerkreis zu finden, sieht auf den ersten Blick so ähnlich aus wie das NP-schwere Problem, einen Hamiltonkreis zu finden, also einen Kreis bei dem jeder Knoten genau einmal besucht wird. Tatsächlich kann man aber effizient entscheiden, ob ein gegebener Graph einen Eulerkreis besitzt und, falls ja, einen solchen finden.

Wir erweitern unsere Betrachtungen auf Multigraphen. Das sind Graphen, in denen zwischen zwei Knoten eine beliebige Anzahl an Kanten verlaufen kann (statt maximal einer wie in einem normalen Graphen). Man kann zeigen, dass ein zusammenhängender Multigraph genau dann einen Eulerkreis enthält, wenn jeder Knoten geraden Grad hat. Außerdem kann man in einem solchen Graphen auch in polynomieller Zeit einen Eulerkreis berechnen.

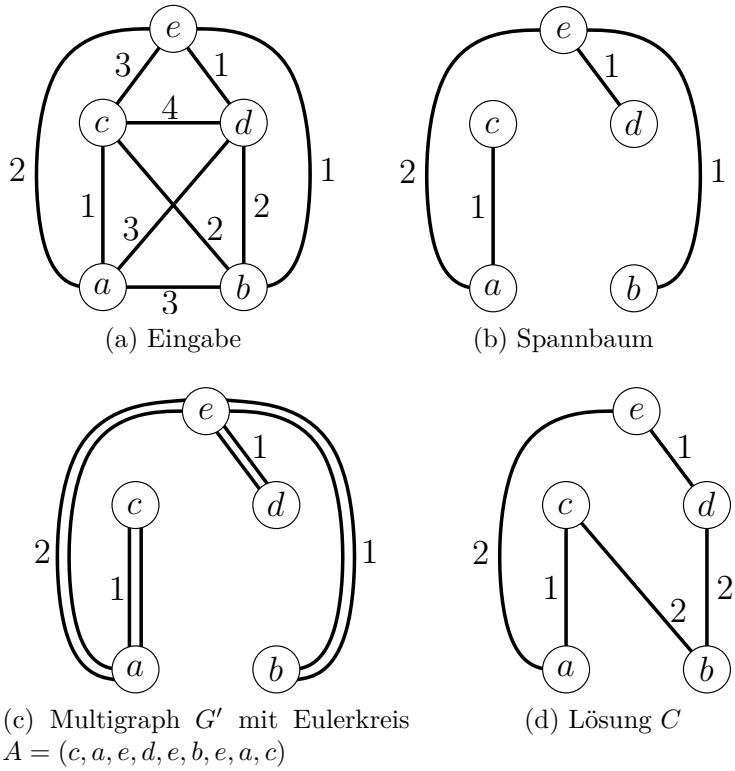
Nun können wir den ersten Approximationsalgorithmus für das metrische TSP vorstellen.

Doppelbaum-TSP

Die Eingabe sei eine Knotenmenge V und eine Metrik d auf V.

1. Sei $G = (V, E)$ ein vollständiger ungerichteter Graph mit Knotenmenge V. Berechne einen minimalen Spannbaum T von G bezüglich der Distanzen d.
2. Erzeuge einen Multigraphen G', der nur die Kanten aus T enthält und jede davon genau zweimal. In G' besitzt jeder Knoten geraden Grad.
3. Finde einen Eulerkreis A in G'.
4. Gib die Knoten in der Reihenfolge ihres ersten Auftretens in A aus. Das Ergebnis sei der Hamiltonkreis C.

In Schritt 4 wird also der gefundene Eulerkreis A in G' (der alle Knoten aus V mindestens einmal enthält) in einen Hamiltonkreis umgebaut. Dazu werden Knoten, die bereits besucht wurden, übersprungen. So wird aus einem Eulerkreis (a, b, c, b, a) zum Beispiel der Hamiltonkreis (a, b, c, a) (siehe Abbildung 5.2). Bilden die Distanzen d eine Metrik auf V, so garantiert die Dreiecksungleichung, dass der Kreis durch das Überspringen von bereits besuchten Knoten nicht länger wird.



(a) Eingabe (b) Spannbaum

(c) Multigraph G' mit Eulerkreis $A = (c, a, e, d, e, b, e, a, c)$ (d) Lösung C

Abbildung 5.2: Beispiel für den Algorithmus Doppelbaum-TSP

Theorem 5.6. *Der Algorithmus* Doppelbaum-TSP *ist ein 2-Approximationsalgorithmus für das metrische* TSP*.*

Beweis. Für eine Menge von Kanten $X \subseteq E$ bezeichnen wir im Folgenden mit $d(X)$ die Summe $\sum_{\{u,v\} \in X} d(u, v)$. Wir fassen in diesem Beweis T, A und C als ungeordnete

Teilmengen der Kanten auf und benutzen entsprechend auch die Bezeichnungen $d(T)$, $d(A)$ und $d(C)$.

Zunächst zeigen wir, dass $d(T)$ eine untere Schranke für die Länge OPT der optimalen Tour ist. Sei $C^\star \subseteq E$ ein kürzester Hamiltonkreis in G bezüglich der Distanzen d. Entfernen wir aus diesem Kreis eine beliebige Kante e, so erhalten wir einen Weg P, der jeden Knoten genau einmal enthält. Ein solcher Weg ist insbesondere ein Spannbaum des Graphen G, also gilt $d(P) \geq d(T)$, da T ein minimaler Spannbaum ist. Insgesamt erhalten wir damit

$$\mathrm{OPT} = d(C^\star) = d(P) + d(e) \geq d(P) \geq d(T).$$

Wir schließen den Beweis ab, indem wir zeigen, dass die Länge des ausgegebenen Hamiltonkreises C durch $2d(T)$ nach oben beschränkt ist. Der Eulerkreis A in G' benutzt jede Kante aus dem Spannbaum T genau zweimal. Damit gilt $d(A) = 2d(T)$. Um von dem Eulerkreis A zu dem Hamiltonkreis C zu gelangen, werden Knoten, die der Eulerkreis A mehrfach besucht, übersprungen. Da die Distanzen die Dreiecksungleichung erfüllen, wird durch das Überspringen von Knoten die Tour nicht verlängert. Es gilt also $d(C) \leq d(A)$. Insgesamt erhalten wir

$$d(C) \leq d(A) = 2d(T) \leq 2 \cdot \mathrm{OPT}.$$

Damit ist gezeigt, dass der Algorithmus stets eine 2-Approximation liefert. □

Es stellt sich die Frage, ob wir den Algorithmus gut analysiert haben, oder ob er in Wirklichkeit eine bessere Approximation liefert. Das folgende Beispiel zeigt, dass sich unsere Analyse nicht signifikant verbessern lässt.

Untere Schranke für Doppelbaum-TSP

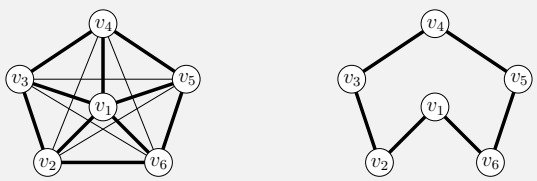
Wir betrachten eine Instanz mit Knotenmenge $V = \{v_1, \ldots, v_n\}$, in der jede Distanz entweder 1 oder 2 ist. Wie wir bereits oben erwähnt haben, ist die Dreiecksungleichung automatisch erfüllt ist, wenn nur 1 und 2 als Distanzen erlaubt sind. Wir gehen davon aus, dass n gerade ist. Es gelte

$$\forall i \in \{2, \ldots, n\} : d(v_1, v_i) = 1$$

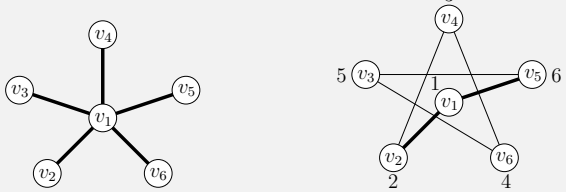
und

$$d(v_2, v_3) = d(v_3, v_4) = \ldots = d(v_{n-1}, v_n) = d(v_n, v_2) = 1.$$

Alle anderen Distanzen seien 2. Dann bilden die Kanten mit Distanz 1 die Vereinigung eines Sterns mit Mittelpunkt v_1 und des Kreises $(v_2, v_3, \ldots, v_n, v_2)$. Die folgende Abbildung zeigt links die Instanz für $n = 6$, wobei dicke Kanten Distanz 1 haben und dünne Kanten Distanz 2.



Die optimale Tour für diese Instanz hat Länge n. Eine mögliche Wahl, die auch in der obigen Abbildung rechts dargestellt ist, ist $(v_1, v_2, \ldots, v_n, v_1)$. Nun kann es sein, dass der Algorithmus Doppelbaum-TSP genau den Stern als minimalen Spannbaum wählt. Dies ist in der Abbildung unten links dargestellt. Basierend auf diesem Spannbaum könnte er den Hamiltonkreis $(v_1, v_2, v_4, v_6, \ldots, v_n, v_3, v_5, \ldots, v_{n-1}, v_1)$ berechnen. Dies ist in der Abbildung unten rechts dargestellt.



Der vom Algorithmus berechnete Hamiltonkreis besteht aus zwei Kanten mit Distanz 1 und $n-2$ Kanten mit Distanz 2. Somit sind seine Kosten insgesamt $2 + 2(n-2) = 2n-2$. Für große n kommt der Approximationsfaktor $(2n-2)/n$ der Zahl 2 beliebig nahe.

5.2.3 Christofides-Algorithmus

Wir können den Algorithmus, den wir im letzten Abschnitt präsentiert haben, deutlich verbessern. Im zweiten Schritt des Algorithmus waren wir nämlich verschwenderisch. Dort haben wir alle Kanten des Spannbaums verdoppelt, um einen Graphen zu erhalten, in dem jeder Knotengrad gerade ist. Es ist aber gut möglich, dass in dem Spannbaum ohnehin bereits viele Knoten geraden Grad haben. Für diese ist eine Verdoppelung der Kanten gar nicht notwendig. Diese Beobachtung führt zu folgendem Algorithmus, den Nicos Christofides 1976 vorgeschlagen hat. Man nennt den Algorithmus deshalb auch Christofides-Algorithmus.

Der Algorithmus benutzt den Begriff eines *perfekten Matchings*. Ein Matching M eines Graphen $G = (V, E)$ ist eine Teilmenge der Kanten, sodass kein Knoten zu mehr als einer Kante aus M inzident ist. Ein perfektes Matching M ist ein Matching mit $|M| = \frac{|V|}{2}$. Es muss also jeder Knoten zu genau einer Kante aus M inzident sein. Es ist bekannt, dass in einem vollständigen Graphen mit einer geraden Anzahl an Knoten und gewichteten Kanten ein perfektes Matching mit minimalem Gewicht in polynomieller Zeit berechnet werden kann. Hierbei ist das Gewicht eines Matchings definiert als die Summe der Gewichte der darin enthaltenen Kanten. Wie der Algorithmus dafür aussieht, wollen wir hier aber nicht diskutieren.

Christofides

Die Eingabe sei eine Knotenmenge V und eine Metrik d auf V.

1. Sei $G = (V, E)$ ein vollständiger ungerichteter Graph mit Knotenmenge V. Berechne einen minimalen Spannbaum T von G bezüglich der Distanzen d.
2. Sei $V' = \{v \in V \mid v \text{ hat ungeraden Grad in } T\}$. Berechne auf der Menge V' ein perfektes Matching M mit minimalem Gewicht bezüglich d.

3. Sei $\tilde{G} = (V, T \cup M)$ ein Multigraph, der jede Kante $e \in T \cap M$ zweimal enthält (und jede Kante $e \in (T \cup M) \setminus (T \cap M)$ einmal). Finde einen Eulerkreis A in dem Multigraphen $\tilde{G}$.
4. Gib die Knoten in der Reihenfolge ihres ersten Auftretens in A aus. Das Ergebnis sei der Hamiltonkreis C.

Theorem 5.7. *Der Christofides-Algorithmus ist ein $\frac{3}{2}$-Approximationsalgorithmus für das metrische* TSP*.*

Beweis. Zunächst beweisen wir, dass der Algorithmus, so wie er oben beschrieben ist, überhaupt ausgeführt werden kann. Dazu sind zwei Dinge zu klären:

1. Warum existiert auf der Menge V' ein perfektes Matching?
2. Warum existiert in dem Graphen $\tilde{G} = (V, T \cup M)$ ein Eulerkreis?

Da der Graph vollständig ist, müssen wir zur Beantwortung der ersten Frage nur zeigen, dass V' eine gerade Anzahl an Knoten enthält. Dazu betrachten wir den Graphen (V, T). Für $v \in V$ bezeichne $\delta(v)$ den Grad des Knotens v in diesem Graphen. Dann ist

$$\sum_{v \in V} \delta(v) = 2|T|$$

eine gerade Zahl, da jede der $|T|$ Kanten inzident zu genau zwei Knoten ist. Bezeichne q die Anzahl an Knoten mit ungeradem Grad. Ist q ungerade, so ist auch die Summe der Knotengrade ungerade, im Widerspruch zu der gerade gemachten Beobachtung.

Zur Beantwortung der zweiten Frage müssen wir lediglich zeigen, dass in dem Graphen $\tilde{G} = (V, T \cup M)$ jeder Knoten geraden Grad besitzt. Das folgt aber direkt aus der Konstruktion: ein Knoten hat entweder bereits geraden Grad im Spannbaum T, oder er hat ungeraden Grad im Spannbaum und erhält eine zusätzliche Kante aus dem Matching.

Nun wissen wir, dass der Algorithmus immer eine gültige Lösung berechnet. Um seinen Approximationsfaktor abzuschätzen, benötigen wir eine untere Schranke für den Wert der optimalen Lösung. Wir haben im vorangegangenen Abschnitt bereits eine solche untere Schranke gezeigt, nämlich das Gewicht des minimalen Spannbaumes. Wir zeigen nun noch eine weitere Schranke.

Lemma 5.8. *Es sei $V' \subseteq V$ beliebig, sodass $|V'|$ gerade ist. Außerdem sei M ein perfektes Matching auf V' mit minimalem Gewicht $d(M)$. Dann gilt $d(M) \leq \mathrm{OPT}/2$.*

Beweis. Es sei $C^\star$ eine optimale TSP-Tour auf der Knotenmenge V, und es sei C' eine Tour auf der Knotenmenge V', die entsteht, wenn wir die Knoten in V' in der Reihenfolge besuchen, in der sie auch in $C^\star$ besucht werden. Das heißt, wir nehmen in $C^\star$ Abkürzungen und überspringen die Knoten, die nicht zu V' gehören. Wegen der Dreiecksungleichung kann sich die Länge der Tour nicht vergrößern und es gilt $\mathrm{OPT} = d(C^\star) \geq d(C')$.

Es sei $V' = \{v_1, \ldots, v_k\}$ und ohne Beschränkung der Allgemeinheit besuche die Tour C' die Knoten in der Reihenfolge $(v_1, \ldots, v_k, v_1)$. Dann können wir die Tour C' in zwei disjunkte perfekte Matchings M_1 und M_2 wie folgt zerlegen:

$$M_1 = \{\{v_1, v_2\}, \{v_3, v_4\}, \ldots, \{v_{k-1}, v_k\}\}$$

und

$$M_2 = \{\{v_2, v_3\}, \{v_4, v_5\}, \ldots, \{v_{k-2}, v_{k-1}\}, \{v_k, v_1\}\}.$$

Wir haben bei der Definition von M_1 und M_2 ausgenutzt, dass $k = |V'|$ gerade ist. Wegen $C' = M_1 \cup M_2$ und $M_1 \cap M_2 = \emptyset$ gilt

$$d(M_1) + d(M_2) = d(C') \leq \text{OPT}.$$

Somit hat entweder M_1 oder M_2 ein Gewicht kleiner oder gleich OPT/2. Dies gilt dann natürlich insbesondere für das perfekte Matching M mit minimalem Gewicht $d(M)$. □

Mithilfe dieses Lemmas folgt nun direkt die Approximationsgüte: Mit den gleichen Argumenten wie beim Algorithmus Doppelbaum-TSP folgt, dass der Christofides-Algorithmus eine Tour C berechnet, deren Länge durch $d(T) + d(M)$ nach oben beschränkt ist. Wir wissen bereits, dass $d(T) \leq \text{OPT}$ gilt und das obige Lemma sagt nun noch, dass $d(M) \leq \text{OPT}/2$ gilt. Zusammen bedeutet das

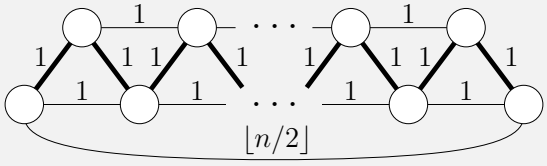
$$d(C) \leq d(T) + d(M) \leq \text{OPT} + \frac{1}{2} \cdot \text{OPT} \leq \frac{3}{2} \cdot \text{OPT}.$$

Damit ist das Theorem bewiesen. □

Auch für den Christofides-Algorithmus können wir wieder zeigen, dass sich unsere Analyse nicht signifikant verbessern lässt.

Untere Schranke für den Christofides-Algorithmus

Sei $n \in \mathbb{N}$ ungerade. Wir betrachten die folgende Instanz des metrischen TSP.



Alle Kanten, die nicht eingezeichnet sind, haben die größtmögliche Länge, die die Dreiecksungleichung zulässt. Die optimale Tour für diese Instanz hat Länge n. Berechnet der Christofides-Algorithmus jedoch im ersten Schritt den Spannbaum, der durch die dicken Kanten angedeutet ist, so ist der Spannbaum ein Pfad. Das heißt, nur die beiden Endknoten haben ungeraden Grad und werden im Matching M durch die Kante mit Gewicht $\lfloor n/2 \rfloor$ verbunden. Somit berechnet der Algorithmus eine Lösung mit Länge $(n-1) + \lfloor n/2 \rfloor$. Für große n entspricht dies in etwa $3n/2$ und der Approximationsfaktor kommt 3/2 beliebig nahe.

5.3 Rucksackproblem

Wir haben im vergangenen Semester für das Rucksackproblem bereits den $\frac{1}{2}$-Approximationsalgorithmus ApproxKP kennengelernt (ohne explizit die Terminologie von Approximationsalgorithmen zu verwenden). Wir werden in diesem Kapitel zeigen, dass für das Rucksackproblem sogar ein *Approximationsschema* existiert. Dabei handelt es sich um einen Algorithmus, dem zusätzlich zu der eigentlichen Eingabe des Problems noch eine Zahl $\varepsilon > 0$ übergeben wird und der eine $(1-\varepsilon)$-Approximation berechnet. Es handelt sich also um einen Approximationsalgorithmus, dem man als zusätzliche Eingabe die gewünschte Approximationsgüte übergeben kann.

Definition 5.9. *Ein* Approximationsschema *A für ein Optimierungsproblem Π ist ein Algorithmus, der zu jeder Eingabe der Form (I, ε) mit $I \in \mathcal{I}_\Pi$ und $\varepsilon > 0$ eine Lösung $A(I,\varepsilon) \in \mathcal{S}_I$ berechnet. Dabei muss für den Wert $w_A(I,\varepsilon) = f_I(A(I,\varepsilon))$ dieser Lösung für jede Eingabe (I,ε) bei einem Minimierungs- oder Maximierungsproblem Π die Ungleichung*

$$w_A(I,\varepsilon) \leq (1+\varepsilon)\cdot \mathrm{OPT}(I) \quad \textit{bzw.} \quad w_A(I,\varepsilon) \geq (1-\varepsilon)\cdot \mathrm{OPT}(I)$$

gelten.

Ein Approximationsschema A heißt polynomielles Approximationsschema (PTAS[1]), *wenn die Laufzeit von A für jede feste Wahl von $\varepsilon > 0$ durch ein Polynom in $|I|$ nach oben beschränkt ist.*

Wir nennen ein Approximationsschema A voll-polynomielles Approximationsschema (FPTAS[2]), *wenn die Laufzeit von A durch ein bivariates Polynom in $|I|$ und $1/\varepsilon$ nach oben beschränkt ist.*

Jedes FPTAS ist auch ein PTAS. Umgekehrt gilt dies jedoch nicht: Ein PTAS könnte z. B. eine Laufzeit von $\Theta(|I|^{1/\varepsilon})$ haben. Diese ist für jedes feste $\varepsilon > 0$ polynomiell in $|I|$ (wobei der Grad des Polynoms von der Wahl von ε abhängt). Diese Laufzeit ist bei einem FPTAS jedoch nicht erlaubt, weil sie nicht polynomiell in $1/\varepsilon$ beschränkt ist. Eine erlaubte Laufzeit eines FPTAS ist z. B. $\Theta(|I|^3/\varepsilon^2)$.

Ein PTAS für ein Problem Π impliziert, dass es für Π für jede Konstante $\varepsilon > 0$ einen $(1+\varepsilon)$-Approximationsalgorithmus bzw. $(1-\varepsilon)$-Approximationsalgorithmus gibt. Der Grad des Polynoms hängt aber im Allgemeinen vom gewählten ε ab. Ist die Laufzeit zum Beispiel $\Theta(|I|^{1/\varepsilon})$, so ergibt sich für $\varepsilon = 0.01$ eine Laufzeit von $\Theta(|I|^{100})$. Eine solche Laufzeit ist nur von theoretischem Interesse und für praktische Anwendungen vollkommen ungeeignet. Bei einem FPTAS ist es hingegen oft so, dass man auch für kleine ε eine passable Laufzeit erhält.

[1] Die Abkürzung geht auf die englische Bezeichnung *polynomial-time approximation scheme* zurück.

[2] Die Abkürzung geht auf die englische Bezeichnung *fully polynomial-time approximation scheme* zurück.

5.3.1 Dynamische Programmierung

Das Approximationsschema, das wir für das Rucksackproblem entwerfen werden, nutzt als wesentlichen Baustein den pseudopolynomiellen Algorithmus zur Lösung des Rucksackproblems, den wir in der Vorlesung „Algorithmen und Berechnungskomplexität I“ kennengelernt haben. Wir werden diesen Algorithmus noch einmal wiederholen. Dazu überlegen wir zunächst, wie eine gegebene Instanz in kleinere Teilinstanzen zerlegt werden kann. Zunächst ist es naheliegend, Teilinstanzen zu betrachten, die nur eine Teilmenge der Objekte enthalten. Das alleine reicht aber noch nicht aus.

Sei eine Instanz mit n Objekten gegeben. Wir bezeichnen wieder mit $p_1, \ldots, p_n \in \mathbb{N}$ die Nutzenwerte, mit $w_1, \ldots, w_n \in \mathbb{N}$ die Gewichte und mit $t \in \mathbb{N}$ die Kapazität des Rucksacks. Zunächst definieren wir $P = \max_{i \in \{1,\ldots,n\}} p_i$. Dann ist nP eine obere Schranke für den Nutzen einer optimalen Lösung. Wir definieren für jede Kombination aus $i \in \{1, \ldots, n\}$ und $p \in \{0, \ldots, nP\}$ folgendes Teilproblem: finde unter allen Teilmengen der Objekte $1, \ldots, i$ mit Gesamtnutzen mindestens p eine mit dem kleinsten Gewicht. Es sei $W(i, p)$ das Gewicht dieser Teilmenge. Existiert keine solche Teilmenge, so setzen wir $W(i, p) = \infty$.

Wir können die Bedeutung von $W(i, p)$ auch wie folgt zusammenfassen: Für jede Teilmenge $I \subseteq \{1, \ldots, i\}$ mit $\sum_{i \in I} p_i \geq p$ gilt $\sum_{i \in I} w_i \geq W(i, p)$. Ist $W(i, p) < \infty$, so gibt es außerdem eine Teilmenge $I \subseteq \{1, \ldots, i\}$ mit $\sum_{i \in I} p_i \geq p$ und $\sum_{i \in I} w_i = W(i, p)$.

Wir werden nun das Rucksackproblem lösen, indem wir eine Tabelle konstruieren, die für jede Kombination aus $i \in \{1, \ldots, n\}$ und $p \in \{0, \ldots, nP\}$ den Wert $W(i, p)$ enthält. Wir konstruieren die Tabelle Schritt für Schritt und fangen zunächst mit den einfachen Randfällen der Form $W(1, p)$ an. Wir stellen uns hier also die Frage, ob wir mit einer Teilmenge von $\{1\}$, also mit dem ersten Objekt, Nutzen mindestens p erreichen können, und wenn ja, mit welchem Gewicht. Ist $p > p_1$, so geht das nicht und wir setzen $W(1, p) = \infty$. Ist hingegen $1 \leq p \leq p_1$, so können wir mindestens Nutzen p erreichen, indem wir das erste Objekt wählen. Das Gewicht beträgt dann w_1. Dementsprechend setzen wir $W(1, p) = w_1$ für $1 \leq p \leq p_1$. Wir nutzen im Folgenden noch die Konvention $W(i, p) = 0$ für alle $i \in \{1, \ldots, n\}$ und $p \leq 0$.

Sei nun bereits für ein i und für alle $p \in \{0, \ldots, nP\}$ der Wert $W(i-1, p)$ bekannt. Wie können wir dann für $p \in \{0, \ldots, nP\}$ den Wert $W(i, p)$ aus den bekannten Werten berechnen? Gesucht ist eine Teilmenge $I \subseteq \{1, \ldots, i\}$ mit $\sum_{j \in I} p_j \geq p$ und kleinstmöglichem Gewicht. Wir unterscheiden zwei Fälle: entweder diese Teilmenge I enthält Objekt i oder nicht.

- Falls $i \notin I$, so ist $I \subseteq \{1, \ldots, i-1\}$ mit $\sum_{j \in I} p_j \geq p$. Unter allen Teilmengen, die diese Bedingungen erfüllen, besitzt I minimales Gewicht. Damit gilt $W(i, p) = W(i-1, p)$.

- Falls $i \in I$, so ist $I \setminus \{i\}$ eine Teilmenge von $\{1, \ldots, i-1\}$ mit Nutzen mindestens $p - p_i$. Unter allen Teilmengen, die diese Bedingungen erfüllen, besitzt $I \setminus \{i\}$ minimales Gewicht. Wäre das nicht so und gäbe es eine Teilmenge $I' \subseteq \{1, \ldots, i-1\}$ mit Nutzen mindestens $p - p_i$ und $w(I') < w(I \setminus \{i\})$, so wäre auch $p(I' \cup$

$\{i\}) \geq p$ und $w(I' \cup \{i\}) < w(I)$ im Widerspruch zur Wahl von I. Es gilt also $w(I \setminus \{i\}) = W(i-1, p-p_i)$ und somit insgesamt $W(i,p) = W(i-1,p-p_i)+w_i$.

Wenn wir vor der Aufgabe stehen, $W(i,p)$ zu berechnen, dann wissen wir a priori nicht, welcher der beiden Fälle eintritt. Wir testen deshalb einfach, welche der beiden Alternativen eine Menge I mit kleinerem Gewicht liefert, d. h. wir setzen $W(i,p) = \min\{W(i-1,p), W(i-1,p-p_i)+w_i\}$.

Ist $W(i,p)$ für alle Kombinationen von i und p bekannt, so können wir die Entscheidungsvariante des Rucksackproblems leicht lösen. Werden wir gefragt, ob es eine Teilmenge der Objekte mit Nutzen mindestens p gibt, deren Gewicht höchstens t ist, so testen wir einfach, ob $W(n,p) \leq t$ gilt. Um den maximal erreichbaren Nutzen mit einem Rucksack der Kapazität t zu finden, genügt es, in der Tabelle das größte p zu finden, für das $W(n,p) \leq t$ gilt.

Der folgende Algorithmus fasst zusammen, wie wir Schritt für Schritt die Tabelle füllen und damit die Wertvariante des Rucksackproblems lösen.

DynKP

1. // Sei $W(i,p) = 0$ für $i \in \{1,\ldots,n\}$ und $p \leq 0$.
2. $P := \max_{i\in\{1,\ldots,n\}} p_i$;
3. **for** $(p = 1;\ p \leq p_1;\ p{+}{+})$ $W(1,p) := w_1$;
4. **for** $(p = p_1 + 1;\ p \leq nP;\ p{+}{+})$ $W(1,p) := \infty$;
5. **for** $(i = 2;\ i \leq n;\ i{+}{+})$
6. **for** $(p = 1;\ p \leq nP;\ p{+}{+})$
7. $W(i,p) = \min\{W(i-1,p), W(i-1,p-p_i)+w_i\}$;
8. **return** maximales $p \in \{1,\ldots,nP\}$ mit $W(n,p) \leq t$

Theorem 5.10. *Der Algorithmus* DynKP *bestimmt in Zeit* $\Theta(n^2P)$ *den maximal erreichbaren Nutzen einer gegebenen Instanz des Rucksackproblems.*

Beweis. Die Korrektheit des Algorithmus folgt direkt aus unseren Vorüberlegungen. Formal kann man per Induktion über i zeigen, dass die Werte $W(i,p)$ genau die Bedeutung haben, die wir ihnen zugedacht haben, nämlich

$$W(i,p) = \min\left\{ w(I) \mid I \subseteq \{1,\ldots,i\}, \sum_{j\in I} p_j \geq p \right\}.$$

Die Laufzeit des Algorithmus ist nicht schwer zu analysieren. Die Tabelle, die berechnet wird, hat einen Eintrag für jede Kombination von $i \in \{1,\ldots,n\}$ und $p \in \{0,\ldots,nP\}$. Das sind insgesamt $\Theta(n^2P)$ Einträge und jeder davon kann in konstanter Zeit durch die Bildung eines Minimums berechnet werden. □

Beispiel für DynKP

Wir führen den Algorithmus DynKP auf der folgenden Instanz mit drei Objekten aus.

	i		
	1	2	3
p_i	2	1	3
w_i	3	2	4

Es gilt $P = 3$ und der Algorithmus DynKP berechnet die folgenden Werte $W(i, p)$.

	i		
p	1	2	3
1	3	2	2
2	3	3	3
3	∞	5	4
4	∞	∞	6
5	∞	∞	7
6	∞	∞	9
7	∞	∞	∞

Zur Beantwortung der Frage, ob es eine Teilmenge $I \subseteq \{1, \ldots, n\}$ mit $w(I) \leq 4$ und $p(I) \geq 5$ gibt, betrachten wir den Eintrag $W(3, 5)$. Dort steht eine 7 und somit benötigt man mindestens Gewicht 7, um Nutzen 5 zu erreichen. Wir können die Frage also mit „nein“ beantworten.

Möchten wir wissen, welchen Nutzen wir maximal mit Kapazität 5 erreichen können, so laufen wir die rechte Spalte von oben nach unten bis zum letzten Eintrag kleiner oder gleich 5 durch. Das ist der Eintrag $W(3, 3)$, also kann man mit Kapazität 5 maximal Nutzen 3 erreichen.

Wir halten noch fest, dass der Algorithmus leicht so modifiziert werden kann, dass er die Optimierungsvariante des Rucksackproblems löst, dass er also nicht nur den maximal erreichbaren Nutzen, sondern auch die dazugehörige Teilmenge der Objekte berechnet. Dazu müssen wir zusätzlich zu jedem Eintrag $W(i, p)$ speichern, welche Teilmenge $I \subseteq \{1, \ldots, i\}$ mit $\sum_{j \in I} p_j \geq p$ Gewicht genau $W(i, p)$ hat. Sei $I(i, p)$ diese Teilmenge. An der Stelle, an der wir $W(i, p) = \min\{W(i-1, p), W(i-1, p-p_i) + w_i\}$ berechnen, können wir auch $I(i, p)$ berechnen: wird das Minimum vom ersten Term angenommen, so setzen wir $I(i, p) = I(i-1, p)$, ansonsten setzen wir $I(i, p) = I(i-1, p-p_i) \cup \{i\}$.

5.3.2 Approximationsschema

Wir entwerfen nun ein FPTAS für das Rucksackproblem. Diesem liegt die folgende Idee zu Grunde: Mithilfe von DynKP können wir effizient Instanzen des Rucksackproblems lösen, in denen alle Nutzenwerte klein (d. h. polynomiell in der Eingabelänge

beschränkt) sind. Erhalten wir nun als Eingabe eine Instanz mit großen Nutzenwerten, so skalieren wir diese zunächst, d. h. wir teilen alle durch dieselbe Zahl K. Diese Skalierung führt dazu, dass der maximal erreichbare Nutzen ebenfalls um den Faktor K kleiner wird. Die optimale Lösung ändert sich jedoch nicht. In der skalierten Instanz ist die gleiche Teilmenge $I \subseteq \{1, \ldots, n\}$ optimal wie in der ursprünglichen Instanz.

Nach dem Skalieren sind die Nutzenwerte rationale Zahlen. Der Trick besteht nun darin, die Nachkommastellen der skalierten Nutzenwerte abzuschneiden. Wir können dann den Algorithmus DynKP anwenden, um die optimale Lösung für die skalierten und gerundeten Nutzenwerte zu bestimmen. Wählen wir den Skalierungsfaktor K richtig, so sind diese Zahlen so klein, dass DynKP polynomielle Laufzeit hat. Allerdings verlieren wir durch das Abschneiden der Nachkommastellen an Präzision. Das bedeutet, die optimale Lösung für die Instanz mit den skalierten und gerundeten Nutzenwerten ist nicht unbedingt optimal für die ursprüngliche Instanz. Ist jedoch K richtig gewählt, so stellt sie eine gute Approximation dar. Wir geben den Algorithmus nun formal an.

FPTAS-KP

Die Eingabe sei $(\mathcal{I}, \varepsilon)$ mit $\mathcal{I} = (p_1, \ldots, p_n, w_1, \ldots, w_n, t)$ und $w_i \leq t$ für alle i.

1. $P := \max_{i \in \{1,\ldots,n\}} p_i$;
2. $K := \frac{\varepsilon P}{n}$; // Skalierungsfaktor
3. **for** $i = 1$ **to** n **do** $p'_i = \lfloor p_i / K \rfloor$; // skaliere und runde die Nutzenwerte
4. Nutze Algorithmus DynKP, um die optimale Lösung für die Instanz $p'_1, \ldots, p'_n, w_1, \ldots, w_n, t$ des Rucksackproblems zu bestimmen und gib diese aus.

Theorem 5.11. *Der Algorithmus* FPTAS-KP *ist ein FPTAS für das Rucksackproblem mit einer Laufzeit von* $O(n^3/\varepsilon)$.

Beweis. Wir betrachten zunächst die Laufzeit des Algorithmus. Diese wird durch den Aufruf von DynKP dominiert, da die Skalierung der Nutzenwerte in den ersten drei Schritten in Linearzeit erfolgt. Sei $P' = \max_{i \in \{1,\ldots,n\}} p'_i$ der größte der skalierten Nutzenwerte. Dann beträgt die Laufzeit von DynKP $\Theta(n^2 P')$ gemäß Theorem 5.10. Es gilt

$$P' = \max_{i \in \{1,\ldots,n\}} \left\lfloor \frac{p_i}{K} \right\rfloor = \left\lfloor \frac{P}{K} \right\rfloor = \left\lfloor \frac{n}{\varepsilon} \right\rfloor \leq \frac{n}{\varepsilon}$$

und somit beträgt die Laufzeit von DynKP $\Theta(n^2 P') = \Theta(n^3/\varepsilon)$.

Nun müssen wir noch zeigen, dass der Algorithmus eine $(1-\varepsilon)$-Approximation liefert. Sei dazu $I' \subseteq \{1, \ldots, n\}$ eine optimale Lösung für die Instanz mit den Nutzenwerten $p'_1, \ldots, p'_n$ und sei $I \subseteq \{1, \ldots, n\}$ eine optimale Lösung für die eigentlich zu lösende Instanz mit den Nutzenwerten $p_1, \ldots, p_n$. Da das Skalieren der Nutzenwerte die optimale Lösung nicht ändert, ist I' auch eine optimale Lösung für die Nutzenwerte $p^*_1, \ldots, p^*_n$ mit $p^*_i = K \cdot p'_i$.

Beispiel

Sei $n = 4$, $P = 50$ und $\varepsilon = \frac{4}{5}$. Dann ist $K = \frac{\varepsilon P}{n} = 10$.

Die verschiedenen Nutzenwerte könnten zum Beispiel wie folgt aussehen:

$$\begin{array}{lll} p_1 = 33 & p'_1 = 3 & p^*_1 = 30 \\ p_2 = 25 & p'_2 = 2 & p^*_2 = 20 \\ p_3 = 50 & p'_3 = 5 & p^*_3 = 50 \\ p_4 = 27 & p'_4 = 2 & p^*_4 = 20 \end{array}$$

Man sieht an diesem Beispiel, dass p_i und p_i^* in der gleichen Größenordnung liegen und sich nicht stark unterscheiden. Die Nutzenwerte p_i^* kann man als eine Version der Nutzenwerte p_i mit geringerer Präzision auffassen.

Für eine Teilmenge $J \subseteq \{1, \ldots, n\}$ bezeichnen wir mit $p(J)$ bzw. $p^*(J)$ ihren Gesamtnutzen bezüglich der ursprünglichen Nutzenwerte und ihren Gesamtnutzen bezüglich der Nutzenwerte $p_1^*, \ldots, p_n^*$:

$$p(J) = \sum_{i \in J} p_i \qquad \text{und} \qquad p^*(J) = \sum_{i \in J} p_i^*.$$

Dann ist $p(I')$ der Wert der Lösung, die der Algorithmus FPTAS-KP ausgibt, und $p(I)$ ist der Wert OPT der optimalen Lösung.

Zunächst halten wir fest, dass die Nutzenwerte p_i und p_i^* nicht stark voneinander abweichen. Es gilt

$$p_i^* = K \cdot \left\lfloor \frac{p_i}{K} \right\rfloor \geq K \left(\frac{p_i}{K} - 1 \right) = p_i - K$$

und

$$p_i^* = K \cdot \left\lfloor \frac{p_i}{K} \right\rfloor \leq p_i.$$

Dementsprechend gilt für jede Teilmenge $J \subseteq \{1, \ldots, n\}$

$$p^*(J) \in [p(J) - nK, p(J)]\,.$$

Wir wissen, dass I' eine optimale Lösung für die Nutzenwerte $p_1^*, \ldots, p_n^*$ ist, da diese durch Skalieren aus den Nutzenwerten $p'_1, \ldots, p'_n$ hervorgehen. Es gilt also insbesondere $p^*(I') \geq p^*(I)$ und somit

$$p(I') \geq p^*(I') \geq p^*(I) \geq p(I) - nK.$$

Da jedes Objekt alleine in den Rucksack passt, gilt $p(I) \geq P$ und damit auch

$$\frac{p(I')}{\mathrm{OPT}} = \frac{p(I')}{p(I)} \geq \frac{p(I) - nK}{p(I)} = 1 - \frac{\varepsilon P}{p(I)} \geq 1 - \varepsilon,$$

womit der Beweis abgeschlossen ist. □

Das FPTAS für das Rucksackproblem rundet die Nutzenwerte so, dass der pseudopolynomielle Algorithmus die gerundete Instanz in polynomieller Zeit lösen kann. Wurde durch das Runden die Präzision nicht zu stark verringert, so ist die optimale Lösung bezüglich der gerundeten Nutzenwerte eine gute Approximation der optimalen Lösung bezüglich der eigentlichen Nutzenwerte. Diese Technik zum Entwurf eines FPTAS lässt

sich auch auf viele andere Probleme anwenden, für die es einen pseudopolynomiellen Algorithmus gibt. Ob sich die Technik anwenden lässt, hängt aber vom konkreten Problem und dem pseudopolynomiellen Algorithmus ab.

Für das Rucksackproblem können wir beispielsweise auch einen pseudopolynomiellen Algorithmus angeben, dessen Laufzeit $\Theta(n^2W)$ beträgt, wobei $W = \max_{i\in\{1,\dots,n\}} w_i$ das größte Gewicht ist. Bei einem solchen Algorithmus wäre es naheliegend, nicht die Nutzenwerte, sondern die Gewichte zu runden und dann mithilfe des pseudopolynomiellen Algorithmus die Instanz mit gerundeten Gewichten zu lösen. Das Ergebnis ist dann aber im Allgemeinen keine Approximation. Runden wir die Gewichte nämlich ab, so kann es passieren, dass dadurch eine Lösung gültig wird, die bezüglich der eigentlichen Gewichte zu schwer ist, die aber einen höheren Nutzen hat als die optimale Lösung. Der Algorithmus würde dann diese Lösung berechnen, die für die eigentliche Instanz gar nicht gültig ist. Rundet man andererseits alle Gewichte auf, so kann es passieren, dass die optimale Lösung ein zu hohes Gewicht bezüglich der gerundeten Gewichte hat und nicht mehr gültig ist. Es besteht dann die Möglichkeit, dass alle Lösungen, die bezüglich der aufgerundeten Gewichte gültig sind, einen viel kleineren Nutzen haben als die optimale Lösung. In diesem Fall würde der Algorithmus ebenfalls keine gute Approximation berechnen.

Als Faustregel kann man festhalten, dass die Chancen, einen pseudopolynomiellen Algorithmus in ein FPTAS zu transformieren, nur dann gut stehen, wenn der Algorithmus pseudopolynomiell bezogen auf die Koeffizienten in der Zielfunktion ist. Ist er pseudopolynomiell bezogen auf die Koeffizienten in den Nebenbedingungen, so ist hingegen Vorsicht geboten.

Kapitel 6

Lineare Programmierung

Wir haben im Laufe des Studiums bereits einige allgemeine Techniken zum Entwurf von Algorithmen wie zum Beispiel dynamische Programmierung und Greedy-Algorithmen kennengelernt. Dennoch mussten wir bei jedem neuen Problem separat prüfen, ob und wie genau sich diese Methoden anwenden lassen. Wir werden nun *lineare Programmierung* kennenlernen. Dabei handelt es sich um eine sehr allgemeine Technik zur Lösung von Optimierungsproblemen, die noch standardisierter ist als die Techniken, die wir bisher gesehen haben. Zwar handelt es sich bei linearer Programmierung um eine Technik zum Entwurf von polynomiellen Algorithmen, am Ende des Kapitels werden wir aber auch einen Zusammenhang zur Lösung von NP-schweren Problemen kennenlernen.

Ein *lineares Programm (LP)* ist ein Optimierungsproblem, bei dem es darum geht, die optimalen Werte für d reelle *Variablen* $x_1, \ldots, x_d \in \mathbb{R}$ zu bestimmen. Dabei gilt es, eine lineare *Zielfunktion*

$$c_1 x_1 + \ldots + c_d x_d \tag{6.1}$$

für gegebene Koeffizienten $c_1, \ldots, c_d \in \mathbb{R}$ zu minimieren oder zu maximieren. Es müssen dabei m lineare *Nebenbedingungen* eingehalten werden. Für jedes $i \in \{1, \ldots, m\}$ sind Koeffizienten $a_{i1}, \ldots, a_{id} \in \mathbb{R}$ und $b_i \in \mathbb{R}$ gegeben, und eine Belegung der Variablen ist nur dann gültig, wenn sie die Nebenbedingungen

$$\begin{aligned} a_{11} x_1 + \ldots + a_{1d} x_d &\leq b_1 \\ &\vdots \\ a_{m1} x_1 + \ldots + a_{md} x_d &\leq b_m \end{aligned} \tag{6.2}$$

einhält. Statt dem Relationszeichen $\leq$ erlauben wir prinzipiell auch Nebenbedingungen mit dem Relationszeichen $\geq$. Da man das Relationszeichen einer Nebenbedingung aber einfach dadurch umdrehen kann, dass man beide Seiten mit -1 multipliziert, gehen wir im Folgenden ohne Beschränkung der Allgemeinheit davon aus, dass alle Nebenbedingungen von der Form $\leq$ sind.

Zur Vereinfachung der Notation definieren wir $x = (x_1, \ldots, x_d)^{\mathrm{T}}$ und $c = (c_1, \ldots, c_d)^{\mathrm{T}}$. Damit sind $x \in \mathbb{R}^d$ und $c \in \mathbb{R}^d$ Spaltenvektoren und wir können die Zielfunktion (6.1)

als Skalarprodukt $c \cdot x$ schreiben. Außerdem definieren wir $A \in \mathbb{R}^{m \times d}$ als die Matrix mit den Einträgen a_{ij} und wir definieren $b = (b_1, \ldots, b_m)^{\mathrm{T}} \in \mathbb{R}^m$. Dann entspricht jede Zeile der Matrix einer Nebenbedingung und wir können die Nebenbedingungen (6.2) als $Ax \leq b$ schreiben. Zusammengefasst können wir ein lineares Programm also wie folgt beschreiben.

Lineares Programm

Eingabe:	$c \in \mathbb{R}^d$, $b \in \mathbb{R}^m$, $A \in \mathbb{R}^{m \times d}$
Lösungen:	alle $x \in \mathbb{R}^d$ mit $Ax \leq b$
Zielfunktion:	minimiere/maximiere $c \cdot x$

Wir präsentieren nun einige Beispiele, die die Leserinnen und Leser davon überzeugen sollen, dass man viele Probleme in Form eines linearen Programms ausdrücken kann.

Hobbygärtner

Ein Hobbygärtner besitzt 100 m^2 Land, auf dem er Blumen und Gemüse anbauen möchte.

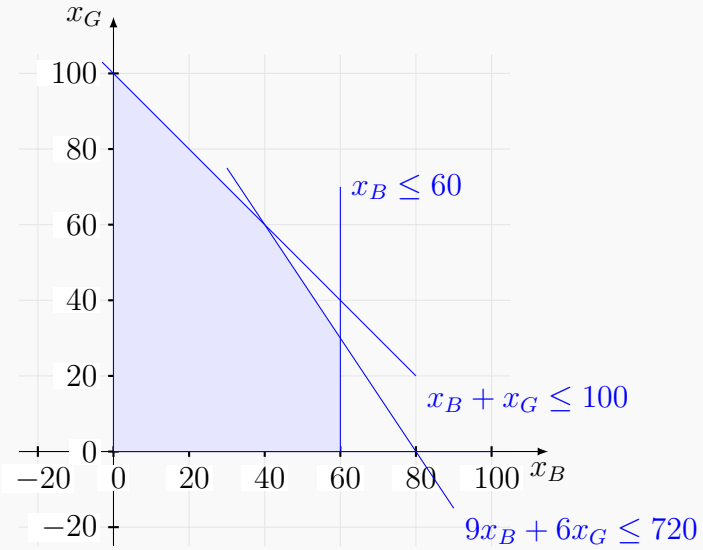
- Nur 60 m^2 des Landes sind für den Anbau von Blumen und Gemüse geeignet. Auf den restlichen 40 m^2 kann nur Gemüse angebaut werden.
- Er hat ein Budget in Höhe von 720 €, das er investieren kann. Das Pflanzen und Düngen der Blumen kostet ihn 9 € pro Quadratmeter; bei Gemüse sind es nur 6 € pro Quadratmeter.
- Der Erlös, den er pro Quadratmeter für Blumen erzielt, beträgt 20 €; bei Gemüse sind es nur 10 € pro Quadratmeter.

Auf wie vielen Quadratmetern sollte der Hobbygärtner Blumen und auf wie vielen sollte er Gemüse anbauen, wenn er seinen Gewinn maximieren möchte?

Um dieses Problem als LP zu formulieren, führen wir zwei Variablen $x_B \in \mathbb{R}$ und $x_G \in \mathbb{R}$ ein, die angeben, auf wie vielen Quadratmetern Blumen bzw. Gemüse angebaut werden. Dann können wir die Nebenbedingungen wie folgt formulieren:

$x_B \geq 0, x_G \geq 0$	(keine negative Anbaufläche)
$x_B + x_G \leq 100$	(maximal 100 m^2)
$x_B \leq 60$	(maximal 60 m^2 Blumen)
$9x_B + 6x_G \leq 720$	(Budget von 720 €)

Grafisch lassen sich die Nebenbedingungen wie folgt darstellen, wobei der blau gekennzeichnete Bereich die Menge der gültigen Lösungen darstellt:



Der Gewinn ist die Differenz von Erlös und Ausgaben. Somit soll die folgende Zielfunktion maximiert werden:

$$(20 - 9)x_B + (10 - 6)x_G = 11x_B + 4x_G.$$

Wir können auch einige Probleme, die wir bereits kennen, als lineares Programm formulieren.

Maximaler Fluss

Gegeben sei ein Flussnetzwerk $G = (V, E)$ mit Quelle $s \in V$ und Senke $t \in V$ und einer *Kapazitätsfunktion* $c : E \to \mathbb{N}_0$. Das Problem, einen maximalen Fluss von s nach t zu berechnen, können wir als LP darstellen, indem wir für jede Kante $e \in E$ eine Variable $x_e \in \mathbb{R}$ einführen, die dem Fluss auf Kante e entspricht. Wir wollen dann die Zielfunktion

$$\sum_{e=(s,v)} x_e - \sum_{e=(v,s)} x_e$$

unter den folgenden Bedingungen maximieren:

$$\forall e \in E : x_e \geq 0 \qquad \text{(Fluss nicht negativ)}$$

$$\forall e \in E : x_e \leq c(e) \qquad \text{(Fluss nicht größer als Kapazität)}$$

$$\forall v \in V \setminus \{s, t\} : \sum_{e=(u,v)} x_e - \sum_{e=(v,u)} x_e = 0 \qquad \text{(Flusserhaltung)}$$

Die letzten Nebenbedingungen sind als Gleichungen und nicht als Ungleichungen formuliert. Dies ist ohne Erweiterung des Modells möglich, da wir jede Gleichung durch zwei Ungleichungen mit den Relationszeichen $\leq$ und $\geq$ ersetzen können.

6.1 Grundlagen

6.1.1 Kanonische Form und Gleichungsform

Um Algorithmen zur Lösung von linearen Programmen zu beschreiben, ist es hilfreich, die folgenden beiden Formen zu betrachten. Dabei sei $c \in \mathbb{R}^d$, $b \in \mathbb{R}^m$ und $A \in \mathbb{R}^{m \times d}$, und $x \in \mathbb{R}^d$ sei ein Vektor aus d Variablen.

kanonische Form	*Gleichungsform*
$\min c \cdot x$	$\min c \cdot x$
$Ax \leq b$	$Ax = b$
$x \geq 0$	$x \geq 0$

Wir können uns ohne Beschränkung der Allgemeinheit auf eine dieser beiden Formen konzentrieren, denn jedes lineare Programm kann in ein äquivalentes lineares Programm in kanonischer Form und in ein äquivalentes lineares Programm in Gleichungsform transformiert werden. Dazu müssen nur die folgenden Operationen in der richtigen Reihenfolge angewendet werden. Wir bezeichnen bei der Beschreibung dieser Operationen mit $a_i = (a_{i1}, \ldots, a_{id})^{\mathrm{T}} \in \mathbb{R}^d$ den Vektor, der der i-ten Zeile der Matrix A entspricht. Dann können wir die i-te Nebenbedingung in kanonischer Form beispielsweise kurz als $a_i \cdot x \leq b_i$ schreiben.

- Haben wir ein lineares Programm, in dem eine Zielfunktion $c \cdot x$ maximiert werden soll, so können wir dies äquivalent dadurch ausdrücken, dass die Zielfunktion $-c \cdot x$, minimiert werden soll.
- Für jede Variable x_i, für die es keine Nebenbedingung $x_i \geq 0$ gibt, können wir zwei Variablen x_i' und x_i'' mit Nebenbedingungen $x_i' \geq 0$ und $x_i'' \geq 0$ einführen und jedes Vorkommen von x_i in Zielfunktion und Nebenbedingungen durch $x_i' - x_i''$ substituieren.
- Bei Nebenbedingungen der Form $a_i \cdot x \geq b_i$ können wir das Relationszeichen umdrehen, indem wir beide Seiten der Ungleichung mit -1 multiplizieren.
- Eine Gleichung $a_i \cdot x = b_i$ kann durch die beiden Ungleichungen $a_i \cdot x \leq b_i$ und $a_i \cdot x \geq b_i$ ersetzt werden.
- Eine Ungleichung $a_i \cdot x \leq b_i$ können wir in eine Gleichung umwandeln, indem wir eine *Schlupfvariable* s_i mit Nebenbedingung $s_i \geq 0$ einführen und die Ungleichung durch die Gleichung $s_i + a_i \cdot x = b_i$ ersetzen.

6.1.2 Geometrische Interpretation

Bevor wir einen Algorithmus beschreiben, der für ein gegebenes lineares Programm eine optimale Belegung der Variablen berechnet, ist es zunächst hilfreich, eine geometrische Interpretation von linearen Programmen zu erarbeiten. Dazu gehen wir davon aus, dass ein lineares Programm in kanonischer Form gegeben ist.

Nebenbedingungen

Jede Variablenbelegung $x \in \mathbb{R}^d$ kann als ein Punkt in dem d-dimensionalen Raum $\mathbb{R}^d$ aufgefasst werden. Die Nebenbedingungen bestimmen dann, welche Punkte zulässige Lösungen des linearen Programms sind. Eine Gleichung der Form $a_i \cdot x = b_i$ definiert eine *affine Hyperebene* $\{x \in \mathbb{R}^d \mid a_i \cdot x = b_i\}$. Dabei handelt es sich um die Verallgemeinerung einer normalen Ebene auf beliebige Dimensionen $d \in \mathbb{N}$. Für $d = 3$ entsprechen Hyperebenen normalen Ebenen; für $d = 2$ sind es beispielsweise Geraden. Jede solche affine Hyperebene definiert den *abgeschlossenen Halbraum*

$$\mathcal{H}_i = \{x \in \mathbb{R}^d \mid a_i \cdot x \leq b_i\}.$$

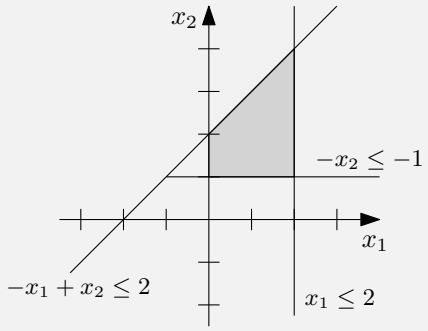
Eine Variablenbelegung $x \in \mathbb{R}^d$ erfüllt genau dann Nebenbedingung i, wenn $x \in \mathcal{H}_i$ gilt. Insgesamt ist eine Variablenbelegung $x \in \mathbb{R}^d$ in einem linearen Programm in kanonischer Form genau dann gültig, wenn $x \in \mathcal{P} := \mathcal{H}_1 \cap \ldots \cap \mathcal{H}_m \cap \mathbb{R}^d_{\geq 0}$ gilt, wobei durch den Schnitt mit $\mathbb{R}^d_{\geq 0}$ der Bedingung $x \geq 0$ Rechnung getragen wird.

Beispiel

In der folgenden Abbildung ist ein lineares Programm mit den Nebenbedingungen

$$\begin{aligned} x_1 &\geq 0, \qquad x_2 \geq 0, \\ x_1 &\leq 2, \quad -x_2 \leq -1, \\ -x_1 &+ x_2 \leq 2 \end{aligned}$$

dargestellt. Der gültige Bereich $\mathcal{P}$ ist in der Abbildung grau hervorgehoben.



Wir können die Menge $\mathbb{R}^d_{\geq 0}$ selbst als einen Schnitt von d Halbräumen darstellen:

$$\mathbb{R}^d_{\geq 0} = \{x \in \mathbb{R}^d \mid -x_1 \leq 0\} \cap \ldots \cap \{x \in \mathbb{R}^d \mid -x_d \leq 0\}.$$

Somit können wir insgesamt festhalten, dass $\mathcal{P}$ der Schnitt von endlich vielen Halbräumen ist. Einen solchen Schnitt nennt man *konvexes Polyeder*, und wir werden das zu einem linearen Programm gehörende konvexe Polyeder auch als sein *Lösungspolyeder* bezeichnen. Wir sagen, dass ein lineares Programm *zulässig* ist, wenn sein Lösungspolyeder nicht leer ist.

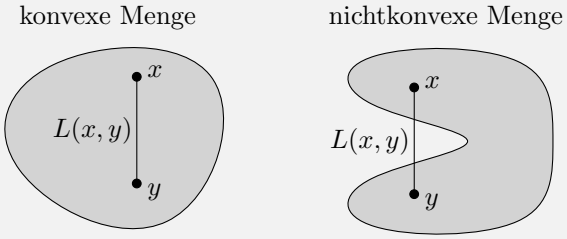
Wie der Name schon sagt, ist jedes konvexe Polyeder $\mathcal{P}$ *konvex*. Das bedeutet, dass für zwei beliebige Punkte $x, y \in \mathcal{P}$ auch alle Punkte auf der direkten Verbindungslinie

zwischen x und y zum konvexen Polyeder $\mathcal{P}$ gehören. Formal ausgedrückt ist eine Menge $X \subseteq \mathbb{R}^d$ konvex, wenn für jedes Paar $x, y \in X$ von Punkten

$$L(x, y) := \{\lambda x + (1 - \lambda)y \mid \lambda \in [0, 1]\} \subseteq X.$$

gilt. Hierbei ist $L(x, y)$ die Verbindungslinie zwischen x und y.

Beispiel



Lemma 6.1. *Jedes konvexe Polyeder $\mathcal{P}$ ist konvex.*

Beweis. Sei $\mathcal{P}$ der Durchschnitt der abgeschlossenen Halbräume $\mathcal{H}_1, \ldots, \mathcal{H}_n$. Zunächst beweisen wir, dass jeder abgeschlossene Halbraum konvex ist. Seien $u \in \mathbb{R}^d$ und $w \in \mathbb{R}$ beliebig und seien x und y beliebige Punkte aus dem abgeschlossenen Halbraum $\mathcal{H} = \{z \mid u \cdot z \leq w\}$. Für jedes $\lambda \in [0, 1]$ gehört dann auch der Punkt $\lambda x + (1 - \lambda)y$ zu $\mathcal{H}$, denn

$$u \cdot (\lambda x + (1 - \lambda)y) = \lambda(u \cdot x) + (1 - \lambda)(u \cdot y) \leq \lambda w + (1 - \lambda)w = w.$$

Somit gehören alle Punkte aus $L(x, y)$ zum abgeschlossenen Halbraum $\mathcal{H}$, und damit ist bewiesen, dass jeder beliebige abgeschlossene Halbraum konvex ist.

Nun müssen wir nur noch zeigen, dass der Schnitt zweier konvexer Mengen wieder konvex ist. Durch mehrfache Anwendung dieser Beobachtung folgt dann direkt das Lemma. Seien also $X \subseteq \mathbb{R}^d$ und $Y \subseteq \mathbb{R}^d$ konvexe Mengen, und seien $x, y \in X \cap Y$ beliebig. Da die Mengen X und Y konvex sind, gilt $L(x, y) \subseteq X$ und $L(x, y) \subseteq Y$. Dementsprechend gilt auch $L(x, y) \subseteq X \cap Y$ und somit ist $X \cap Y$ konvex. □

Sei ein lineares Programm in kanonischer Form mit Lösungspolyeder $\mathcal{P}$ gegeben. Die Tatsache, dass bei einem linearen Programm die erlaubten Variablenbelegungen eine konvexe Menge bilden, hat eine wichtige Konsequenz für die optimale Variablenbelegung. Wir sagen, eine Variablenbelegung $x \in \mathcal{P}$ ist *lokal optimal*, wenn es ein $\varepsilon > 0$ gibt, für das es kein $y \in \mathcal{P}$ mit $\|x - y\| \leq \varepsilon$ und $c \cdot y < c \cdot x$ gibt. Hierbei bezeichne $\|x - y\|$ den euklidischen Abstand zwischen x und y, also $\|x - y\| = \sqrt{(x_1 - y_1)^2 + \ldots + (x_d - y_d)^2}$. Eine Variablenbelegung ist also lokal optimal, wenn es eine Umgebung um sie herum gibt, in der es keine echt bessere zulässige Variablenbelegung gibt.

Theorem 6.2. *Sei ein lineares Programm in kanonischer Form mit Lösungspolyeder $\mathcal{P}$ gegeben und sei $x \in \mathcal{P}$ eine lokal optimale Variablenbelegung. Dann ist x auch global optimal, d. h. es gibt kein $y \in \mathcal{P}$ mit $c \cdot y < c \cdot x$.*

Beweis. Angenommen, es gäbe ein $y \in \mathcal{P}$ mit $c \cdot y < c \cdot x$. Da das Lösungspolyeder $\mathcal{P}$ konvex ist, liegt auch jeder Punkt auf der Verbindungslinie $L(x, y)$ in $\mathcal{P}$. Sei $z = \lambda x + (1 - \lambda)y$ mit $\lambda \in [0, 1)$ ein solcher Punkt. Dann können wir den Wert der Zielfunktion an der Stelle z schreiben als

$$c \cdot z = c \cdot (\lambda x + (1 - \lambda)y) = \lambda(c \cdot x) + (1 - \lambda)(c \cdot y) < c \cdot x,$$

wobei wir bei der letzten Ungleichung $\lambda < 1$ und $c \cdot y < c \cdot x$ ausgenutzt haben. Das bedeutet, jeder Punkt $z \in L(x, y)$ mit $z \neq x$ ist eine echt bessere Variablenbelegung als x. Da jeder solche Punkt zu $\mathcal{P}$ gehört und damit gültig ist, gibt es für jedes $\varepsilon > 0$ einen Punkt $z \in \mathcal{P}$ mit $c \cdot z < c \cdot x$ und $\|x - z\| \leq \varepsilon$. Damit ist x im Widerspruch zur Annahme nicht lokal optimal. □

Zielfunktion

Wir bezeichnen ein lineares Programm als *unbeschränkt*, wenn der zu minimierende Zielfunktionswert innerhalb des Lösungspolyeders $\mathcal{P}$ beliebig klein werden kann, d. h. wenn für alle $z \in \mathbb{R}$ eine Variablenbelegung $x \in \mathcal{P}$ mit $c \cdot x \leq z$ existiert. Ansonsten heißt das lineare Programm *beschränkt*.

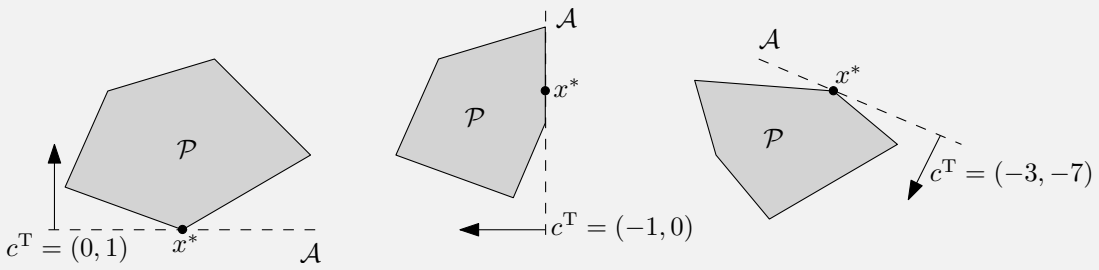
Bisher haben wir lediglich eine geometrische Interpretation der Nebenbedingungen erarbeitet. Wir können auch die Zielfunktion geometrisch interpretieren. Dazu überlegen wir uns zunächst, welche Teilmengen von $\mathbb{R}^d$ den gleichen Zielfunktionswert bezüglich einer linearen Zielfunktion haben. Sei $c \cdot x$ eine beliebige lineare Zielfunktion und sei $w \in \mathbb{R}$ beliebig. Die Menge

$$\{x \in \mathbb{R}^d \mid c \cdot x = w\}$$

bildet eine affine Hyperebene mit Normalenvektor c. Der Vektor c steht also senkrecht auf der Hyperebene, die die Punkte mit gleichem Zielfunktionswert enthält. Wir können nun wie folgt grafisch eine optimale Lösung eines linearen Programms bestimmen.

1. Finde ein $w \in \mathbb{R}$, für das der Schnitt der Hyperebene $\{x \in \mathbb{R}^d \mid c \cdot x = w\}$ mit dem Lösungspolyeder nichtleer ist.
2. Verschiebe die Hyperebene $\{x \in \mathbb{R}^d \mid c \cdot x = w\}$ solange parallel in Richtung $-c$ wie sie einen nichtleeren Schnitt mit dem Lösungspolyeder $\mathcal{P}$ besitzt. Das ist äquivalent dazu, den Wert w solange zu verringern wie $\{x \in \mathbb{R}^d \mid c \cdot x = w\} \cap \mathcal{P} \neq \emptyset$ gilt.
3. Terminiert der zweite Schritt nicht, da jede betrachtete Hyperebene einen nichtleeren Schnitt mit $\mathcal{P}$ besitzt, so ist das lineare Programm unbeschränkt. Ansonsten sei $\mathcal{A} = \{x \in \mathbb{R}^d \mid c \cdot x = w\}$ die letzte Hyperebene, für die $\mathcal{A} \cap \mathcal{P} \neq \emptyset$ gilt. Dann stellt jeder Punkt $x^* \in \mathcal{A} \cap \mathcal{P}$ eine optimale Variablenbelegung des linearen Programms dar, da $\mathcal{P} \subseteq \{x \in \mathbb{R}^d \mid c \cdot x \geq w\}$.

Beispiel



An dieser Stelle weisen wir darauf hin, dass man mit dem oben beschriebenen grafischen Verfahren zwar jedes lineare Programm optimal lösen kann, dass es sich jedoch um keinen Algorithmus handelt, der so ohne Weiteres implementiert werden kann. In höheren Dimensionen ist es beispielsweise bereits nicht trivial, im ersten Schritt ein geeignetes $w \in \mathbb{R}$ zu finden, für das der Schnitt von $\{x \in \mathbb{R}^d \mid c \cdot x = w\}$ und $\mathcal{P}$ nichtleer ist.

6.2 Simplex-Algorithmus

Der Simplex-Algorithmus, den wir in diesem Abschnitt beschreiben werden, wurde bereits 1947 von George Dantzig vorgeschlagen. Es handelt sich dabei auch heute noch um einen der erfolgreichsten Algorithmen zur Lösung von linearen Programmen in der Praxis.

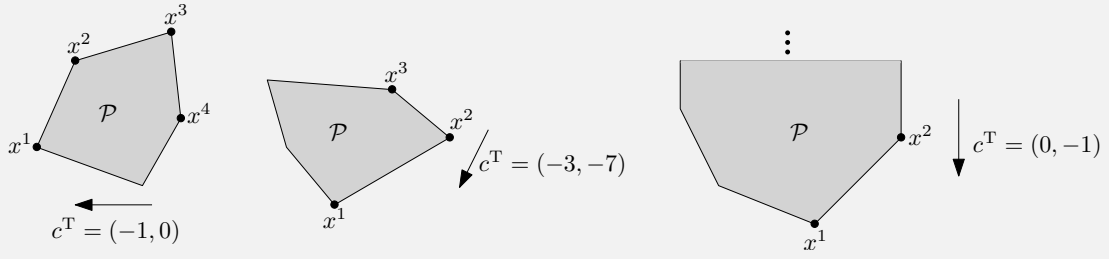
Anschaulich lässt sich das Vorgehen wie folgt erklären. Es sei ein lineares Programm mit Lösungspolyeder $\mathcal{P}$ gegeben. Der Simplex-Algorithmus führt eine lokale Suche auf den Ecken[1] des Lösungspolyeders durch. Das bedeutet, er startet mit einer beliebigen Ecke $x^1 \in \mathcal{P}$ und testet, ob es eine benachbarte Ecke mit besserem Zielfunktionswert gibt. Dabei sind zwei Ecken benachbart, wenn sie auf der gleichen Kante liegen. Gibt es eine solche benachbarte Ecke $x^2 \in \mathcal{P}$, so macht der Algorithmus mit x^2 analog weiter und testet wieder, ob es eine bessere benachbarte Ecke gibt. Den Übergang von einer Ecke zu einer besseren in der Nachbarschaft nennen wir auch *Pivotschritt*. Da der Algorithmus den Zielfunktionswert mit jedem Schritt verbessert, kann er keine Ecke mehrfach besuchen. Da es nur endliche viele Ecken gibt, muss also nach einer endlichen Anzahl an Schritten eine Ecke $x^k \in \mathcal{P}$ erreicht sein, in deren Nachbarschaft es keine bessere Ecke gibt. Es gibt dann zwei Fälle, die eintreten können. Entweder die Ecke x^k ist in einer unbeschränkten Kante enthalten, entlang derer der Zielfunktionswert beliebig gut werden kann, oder, wenn das nicht der Fall ist, dann ist die Ecke x^k optimal. Wir werden später beweisen, dass dies wirklich so ist.

Beispiel

Bei den ersten beiden Beispielen wird das Optimum an den Ecken x^4 bzw. x^3 angenommen. Im dritten Beispiel ist x^2 die letzte besuchte Ecke, von der aus es

[1] Auf eine formale Definition der Begriffe „Ecke“ und „Kante“ verzichten wir an dieser Stelle.

eine unendliche Kante gibt, entlang derer der Wert der Zielfunktion beliebig klein wird.



Genau wie bei dem geometrischen Algorithmus zur Lösung von linearen Programmen weisen wir auch hier darauf hin, dass sich aus der obigen anschaulichen Beschreibung des Simplex-Algorithmus noch keine direkte Implementierung ergibt. So ist zum Beispiel nicht klar, wie im ersten Schritt überhaupt eine beliebige Ecke von $\mathcal{P}$ gefunden werden kann oder wie man die Nachbarschaft einer Ecke durchsuchen kann.

6.2.1 Formale Beschreibung

Zur formalen Beschreibung betrachten wir ein lineares Programm in Gleichungsform, in dem die Zielfunktion $c \cdot x$ unter den Nebenbedingungen $Ax = b$ und $x \geq 0$ minimiert werden soll. Dabei seien $c \in \mathbb{R}^d$, $b \in \mathbb{R}^m$, $A \in \mathbb{R}^{m \times d}$ und $x \in \mathbb{R}^d$ sei ein Vektor, der aus d Variablen besteht. Es bezeichne $\mathcal{P}$ das zugehörige Lösungspolyeder. Wir gehen davon aus, dass alle Zeilen der Matrix A linear unabhängig sind.

Basislösungen

Wir beginnen mit der Definition von *Basislösungen*. Diese entsprechen genau den Ecken von $\mathcal{P}$.

Für $B \subseteq \{1, \ldots, d\}$ bezeichnen wir mit A_B die Teilmatrix von A, die aus genau den Spalten von A besteht, deren Index in B enthalten ist. Analog bezeichnen wir für $x \in \mathbb{R}^d$ mit x_B den $|B|$-dimensionalen Subvektor von x, der nur aus den Komponenten aus B besteht. Sei nun $x \in \mathbb{R}^d$ und $B(x) = \{i \in \{1, \ldots, d\} \mid x_i \neq 0\}$. Sind die Spalten von $A_{B(x)}$ linear unabhängig und gilt $Ax = b$, so heißt x *Basislösung*. Gilt zusätzlich $x \in \mathcal{P}$ (d. h. $x \geq 0$), so nennen wir x *zulässige Basislösung*. Für jede Basislösung x gilt $|B(x)| \leq m$, da der Rang der Matrix A durch m nach oben beschränkt ist.

Basislösungen kann man sich anschaulich wie folgt vorstellen. Sei ein lineares Programm in kanonischer Form mit d Variablen und m Nebenbedingungen gegeben, das wir in Gleichungsform transformieren, indem wir m Schlupfvariablen einfügen. Das entsprechende lineare Programm in Gleichungsform besteht dann aus $d+m$ Variablen und m Nebenbedingungen. In jeder Basislösung $x \in \mathbb{R}^{d+m}$ dieses linearen Programms sind höchstens m Variablen ungleich Null und damit mindestens $(d+m)-m = d$ Variablen gleich Null. Jede dieser Variablen ist entweder eine Variable aus dem ursprünglichen linearen Programm in kanonischer Form oder eine neu eingeführte Schlupfvariable. Ist eine Schlupfvariable gleich Null, so bedeutet das, dass die zugehörige

Ungleichung mit Gleichheit erfüllt ist. Ist eine Variable aus dem ursprünglichen linearen Programm gleich Null, so bedeutet das, dass die Nichtnegativitätsbedingung mit Gleichheit erfüllt ist. Auf jeden Fall sind d Nebenbedingungen mit Gleichheit erfüllt und die Basislösung entspricht dem Schnittpunkt der zugehörigen Hyperebenen.

Beispiel

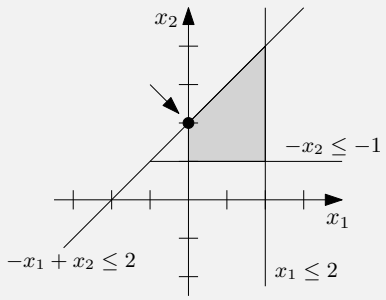
Wir betrachten die folgenden Nebenbedingungen in kanonischer Form:

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \geq 0 \quad \text{und} \quad \begin{pmatrix} 1 & 0 \\ 0 & -1 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \leq \begin{pmatrix} 2 \\ -1 \\ 2 \end{pmatrix}$$

In Gleichungsform können wir diese Nebenbedingungen wie folgt schreiben:

$$\begin{pmatrix} x_1 \\ x_2 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix} \geq 0 \quad \text{und} \quad \begin{pmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ -1 & 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix} = \begin{pmatrix} 2 \\ -1 \\ 2 \end{pmatrix}$$

Das Lösungspolyeder $\mathcal{P}$ ist in der Abbildung grau hervorgehoben. Die markierte Ecke $(0, 2)$ entspricht dem Schnittpunkt der Hyperebenen $x_1 = 0$ und $-x_1+x_2 = 2$. In dem linearen Programm in Gleichungsform entspricht diese Ecke der Basislösung $(0, 2, 2, 1, 0)$, wobei die letzten drei Komponenten s_1, s_2 und s_3 entsprechen.



Für $x = (0, 2, 2, 1, 0)$ gilt $B = B(x) = \{2, 3, 4\}$ und dementsprechend ist x_B die eindeutige Lösung des folgenden Gleichungssystems:

$$\begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_2 \\ s_1 \\ s_2 \end{pmatrix} = \begin{pmatrix} 2 \\ -1 \\ 2 \end{pmatrix}$$

Wir sagen, dass das lineare Programm *degeneriert* ist, wenn es eine zulässige Basislösung $x \in \mathcal{P}$ mit $|B(x)| < m$ gibt. Ist ein lineares Programm nicht degeneriert, so sind in jeder zulässigen Basislösung genau m Variablen ungleich Null. Wir werden in diesem Abschnitt davon ausgehen, dass das vorliegende lineare Programm nicht degeneriert ist. Diese Annahme erleichtert zwar die Beschreibung, sie ist aber nicht notwendig, denn an sich können mit dem Simplex-Algorithmus auch degenerierte lineare Programme gelöst werden. Wir haben gesehen, dass Basislösungen in linearen

Programmen in kanonischer Form Punkten entsprechen, in denen sich mindestens d der durch die Nebenbedingungen beschriebenen Hyperebenen schneiden. Ein solches lineares Programm ist genau dann degeneriert, wenn es einen Punkt in $\mathcal{P}$ gibt, in dem sich mehr als d Hyperebenen schneiden.

Pivotschritte

Wir gehen zunächst davon aus, dass wir eine beliebige zulässige Basislösung $x = x^1 \in \mathcal{P}$ kennen. Für $B = B(x)$ und $N = \{1, \ldots, d\} \setminus B$ ergibt sich diese Basislösung als eindeutige Lösung des Gleichungssystems $A_B x_B = b$ und $x_N = 0$. Aufgrund der Annahme, dass das lineare Programm nicht degeneriert ist, besteht die Matrix A_B aus genau m linear unabhängigen Spalten. Somit ist A_B eine invertierbare $m \times m$-Matrix. Es gilt also insgesamt

$$x = \begin{pmatrix} x_B \\ x_N \end{pmatrix} = \begin{pmatrix} A_B^{-1} b \\ 0 \end{pmatrix}.$$

Wir nennen die Variablen in x_B *Basisvariablen* und die Variablen in x_N *Nichtbasisvariablen*. Um zu einer benachbarten Ecke zu gelangen, wollen wir jetzt den Wert einer Nichtbasisvariablen erhöhen. Dazu ist es zunächst hilfreich, die Werte der Basisvariablen in Abhängigkeit von den Werten der Nichtbasisvariablen auszudrücken. Dazu bezeichne A_N die Teilmatrix von A, die aus genau den Spalten besteht, die nicht in A_B enthalten sind. Es gilt dann

$$A_B x_B + A_N x_N = b \iff x_B = A_B^{-1}(b - A_N x_N).$$

Wählen wir $x_N \in \mathbb{R}_{\geq 0}^{d-m}$ beliebig, so ergibt sich daraus eine eindeutige Wahl von x_B. Wir können deshalb die Zielfunktion in Abhängigkeit von x_N ausdrücken:

$$\begin{aligned} c \cdot x &= c_B \cdot x_B + c_N \cdot x_N = c_B \cdot (A_B^{-1}(b - A_N x_N)) + c_N \cdot x_N \\ &= c_B \cdot (A_B^{-1} b) + (c_N^{\mathrm{T}} - c_B^{\mathrm{T}} A_B^{-1} A_N) x_N. \end{aligned} \tag{6.3}$$

Eine Wahl $x_N \in \mathbb{R}_{\geq 0}^{d-m}$ ist nur dann zulässig, wenn sich daraus nichtnegative Werte für die Basisvariablen ergeben.

Wir nennen zwei Basislösungen *benachbart*, wenn sich die Mengen der Basisvariablen nur in einem Element unterscheiden. Wir können von einer Basislösung x zu einer benachbarten Basislösung gelangen, indem wir eine Nichtbasisvariable $j \in N$ auswählen und den Wert von x_j erhöhen und die Werte aller anderen Nichtbasisvariablen auf Null lassen. Die zu j gehörende Komponente des Vektors $\overline{c} := c_N^{\mathrm{T}} - c_B^{\mathrm{T}} A_B^{-1} A_N$ aus Gleichung (6.3) beschreibt dann, wie sich der Wert der Zielfunktion ändert. Wir nennen $\overline{c}$ auch den Vektor der *reduzierten Kosten*. Nur wenn die zu j gehörende Komponente des Vektors $\overline{c}$ negativ ist, verbessert sich die Lösung durch die Erhöhung von x_j.

Gilt $\overline{c} \geq 0$ für die aktuelle Basislösung x, so terminiert der Simplex-Algorithmus und gibt aus, dass x optimal ist. Ansonsten betrachten wir eine beliebige Nichtbasisvariable x_j mit $\overline{c}_j < 0$ und erhöhen diese kontinuierlich von 0. Wenn wir die Basisvariablen gemäß $x_B = A_B^{-1}(b - A_N x_N)$ entsprechend anpassen, dann verringert sich durch die Erhöhung von x_j gemäß Gleichung (6.3) der Zielfunktionswert. Es kann jedoch sein,

dass die Erhöhung von x_j dazu führt, dass sich der Wert einiger Basisvariablen verringert und sogar negativ wird. Dann erhalten wir keine zulässige Lösung mehr. Da wir davon ausgehen, dass das lineare Programm nicht degeneriert ist, haben in x aber alle Basisvariablen echt positive Werte. Das bedeutet, wir können x_j zumindest von Null auf einen eventuell kleinen, aber zumindest echt positiven Wert setzen, ohne dadurch eine unzulässige Lösung zu erhalten.

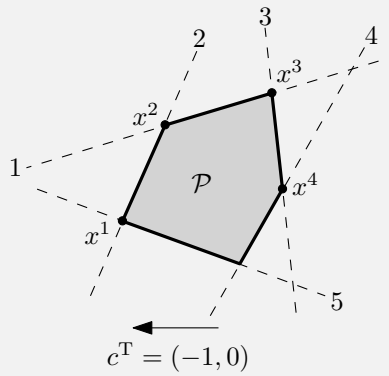
Die zu x_j gehörende Spalte der Matrix $A_B^{-1}A_N$ nennen wir im Folgenden $u \in \mathbb{R}^m$. Ist keine Komponente von u echt positiv, so können wir x_j beliebig groß setzen, ohne dadurch eine unzulässige Lösung zu erhalten. Das bedeutet, dass das lineare Programm unbeschränkt ist, da wir beliebig gute zulässige Lösungen finden können. In diesem Fall terminiert der Algorithmus. Ansonsten setzen wir x_j größtmöglich unter der Bedingung, dass alle Basisvariablen nichtnegativ bleiben. Das bedeutet

$$x_j := \min\left\{ \frac{x_\ell}{u_\ell} \;\middle|\; \ell \in B, u_\ell > 0 \right\}.$$

Sei x_k eine Basisvariable, für die das obige Minimum angenommen wird. Nachdem wir x_j erhöht haben, nimmt x_k den Wert Null an. Auf diese Weise erhalten wir eine neue Basislösung x' mit $B(x') = B(x) \cup \{j\} \setminus \{k\}$. Damit ist x' eine zu x benachbarte Basislösung mit besserem Zielfunktionswert. Hier haben wir ausgenutzt, dass das lineare Programm nicht degeneriert ist und dass deshalb $|B(x')| = m$ gilt.

Beispiel

In dem folgenden Beispiel ist eine Ausführung des Simplex-Algorithmus zu sehen, in der die Folge x^1, x^2, x^3, x^4 von Basislösungen durchlaufen wird. Jede dieser Basislösungen können wir als Schnittpunkt zweier Hyperebenen darstellen. Die Basislösung x^1 entspricht zum Beispiel der Menge $\{2, 5\}$ von Hyperebenen. Insgesamt ergibt sich die Folge $\{2, 5\}, \{1, 2\}, \{1, 3\}, \{3, 4\}$. Dort ist zu erkennen, dass jeweils eine Hyperebene ausgetauscht wird.



Berechnung der initialen Basislösung

Wir haben noch nicht diskutiert, wie eine initiale Basislösung gefunden werden kann, mit der der Simplex-Algorithmus gestartet werden kann. Sei ein lineares Programm mit den Nebenbedingungen $Ax = b$ gegeben und sei ohne Beschränkung der Allgemeinheit

$b \geq 0$. Ist das nicht der Fall, so können die entsprechenden Nebenbedingungen mit -1 multipliziert werden. Für die m Nebenbedingungen führen wir Hilfsvariablen $h_1 \geq 0, \ldots, h_m \geq 0$ ein. Die Nebenbedingung $a_i \cdot x = b_i$ ersetzen wir für jedes $i \in \{1, \ldots, m\}$ durch die Nebenbedingung $a_i \cdot x + h_i = b_i$. Wir ignorieren außerdem die gegebene Zielfunktion des linearen Programms und definieren als neue Zielfunktion $h_1 + \ldots + h_m$.

Das so entstandene lineare Programm hat die Eigenschaft, dass wir einfach eine zulässige Basislösung angeben können. Dazu setzen wir $h_i = b_i$ für jedes $i \in \{1, \ldots, m\}$ und $x_i = 0$ für alle $i \in \{1, \ldots, d\}$. Wir können somit den Simplex-Algorithmus benutzen, um für dieses lineare Programm eine optimale Basislösung (x^*, h^*) zu berechnen. Dieser Schritt wird als die *erste Phase* des Simplex-Algorithmus bezeichnet. Ist der Wert der Zielfunktion in dieser optimalen Lösung echt größer als Null, dann gibt es keine Lösung, die alle Gleichungen des ursprünglichen linearen Programms erfüllt, und somit ist dieses lineare Programm nicht zulässig. Ansonsten, wenn $h^* = 0$ gilt, so ist x^* eine zulässige Basislösung für das ursprüngliche lineare Programm. Mit dieser Lösung x^* kann dann der Simplex-Algorithmus für das ursprüngliche lineare Programm initialisiert werden. Die Optimierung der eigentlichen Zielfunktion wird dann als *zweite Phase* des Simplex-Algorithmus bezeichnet.

6.2.2 Korrektheit und Terminierung

Wir beweisen nun, dass der Simplex-Algorithmus tatsächlich immer die optimale Lösung liefert beziehungsweise erkennt, dass es keine optimale Lösung gibt.

Theorem 6.3. *In nicht-degenerierten linearen Programmen terminiert der Simplex-Algorithmus immer. Er findet eine optimale Lösung oder stellt fest, dass es keine oder keine optimale Lösung gibt.*

Beweis. Zunächst diskutieren wir, dass der Algorithmus auf jeden Fall terminiert. Hierzu stellen wir fest, dass sich sowohl in der ersten Phase als auch in der zweiten Phase jeweils der Zielfunktionswert durch einen Pivotschritt strikt verbessert. Somit wird jede Basislösung nur einmal betrachtet. Weil es außerdem nur endlich viele Basislösungen gibt, kann es nur endlich viele Pivotschritte geben.

Es bleibt die Korrektheit zu zeigen. Wir betrachten zunächst den Fall, dass er tatsächlich eine Lösung x liefert. Dies geschieht, wenn in der zweiten Phase der Vektor der reduzierten Kosten $\overline{c}$ zur Basislösung x keinen negativen Eintrag enthält. Wir behaupten nun, dass x eine optimale Lösung für das lineare Programm ist.

Es sei hier $\overline{c} := c_N^{\mathrm{T}} - c_B^{\mathrm{T}} A_B^{-1} A_N \geq 0$. Sei $x' \in \mathcal{P}$ eine beliebige zulässige Lösung. Dann ist $x'_N \geq 0$ und Gleichung (6.3) impliziert

$$c \cdot x' = c_B \cdot (A_B^{-1} b) + \overline{c}\, x'_N \geq c_B \cdot (A_B^{-1} b) = c \cdot x.$$

Also ist x' nicht besser als x, und somit muss x eine optimale Lösung sein.

Der Algorithmus kann auch terminieren, ohne eine Lösung zu liefern. Eine Möglichkeit hierzu ist, dass in der ersten Phase keine gültige Lösung für das ursprüngliche lineare

Programm gefunden wurde. Aus dem ersten Teil des Beweises wissen wir aber, dass die gefunden Lösung (x^*, h^*) in der ersten Phase optimal ist. Entsprechend kann es keine Lösung mit $h^* = 0$ geben. Dies bedeutet, dass es in der Tat keine gültige Lösung des ursprünglichen linearen Programms gibt.

Schließlich gibt es noch die Möglichkeit, dass der Algorithmus in der zweiten Phase abbricht, weil er feststellt, dass das lineare Programm unbeschränkt ist, weil $u_\ell \leq 0$ für ein ℓ gilt. Wie bereits beschrieben, kann in diesem Fall x_j beliebig groß gewählt werden. Somit kann es keine optimale Lösung geben, denn der Algorithmus wäre in der Lage, Lösungen mit beliebig kleinen Zielfunktionswerten zu berechnen. □

Der Simplex-Algorithmus berechnet ausschließlich Basislösungen. Aus seiner Korrektheit folgt, dass auch an einer Basislösung der optimale Zielfunktionswert angenommen wird.

Korollar 6.4. *Hat ein nicht-degeneriertes lineares Programm eine optimale Lösung, so gibt es auch eine optimale Lösung, die gleichzeitig Basislösung ist.*

6.2.3 Laufzeit

Die Laufzeit des Simplex-Algorithmus setzt sich aus zwei Komponenten zusammen. Zum einen muss geklärt werden, was die Komplexität eines einzelnen Pivotschrittes ist, und zum anderen, wie viele Pivotschritte es gibt. Die Antwort auf die erste Frage gibt folgendes Theorem, das wir nicht beweisen werden.

Theorem 6.5. *Die Laufzeit eines einzelnen Pivotschrittes ist polynomiell in der Eingabelänge des linearen Programms beschränkt.*

Aus der Beschreibung des Simplex-Algorithmus geht bereits hervor, dass zur Durchführung eines Pivotschrittes nur einige elementare Rechenoperationen durchgeführt werden müssen. Trotzdem ist das obige Theorem nicht trivial, denn die Laufzeit dieser Rechenoperationen hängt von der Darstellungslänge der Zahlen ab. Diese muss man in Beziehung zu der Eingabelänge setzen. Man muss also zeigen, dass die Zahlen, mit denen man beim Simplex-Algorithmus rechnet, nicht zu groß werden können.

Die Antwort auf die zweite Frage ist hingegen ernüchternd. Klee und Minty haben 1972 folgendes Theorem gezeigt.

Theorem 6.6. *Für jedes $n \in \mathbb{N}$ gibt es ein lineares Programm in Gleichungsform mit $3n$ Variablen und $2n$ Nebenbedingungen, in dem alle Koeffizienten ganzzahlig sind und Absolutwert höchstens 4 haben, und auf dem der Simplex-Algorithmus $2^n - 1$ Pivotschritte durchführen kann.*

In diesem Theorem haben wir davon gesprochen, dass der Simplex-Algorithmus $2^n - 1$ Pivotschritte „durchführen kann“. Diese Formulierung ist so gewählt, da es für eine Basislösung oft mehrere mögliche Pivotschritte gibt und wir bisher nicht spezifiziert

haben, welcher davon durchgeführt wird. Das Theorem ist so zu lesen, dass eine exponentiell lange Folge von Pivotschritten existiert, die der Simplex-Algorithmus durchführen kann. Es ist aber zunächst nicht ausgeschlossen, dass man durch eine geschickte Wahl der Pivotschritte garantieren kann, dass der Simplex-Algorithmus stets polynomiell viele Schritte benötigt. Eine Regel, die die Wahl des nächsten Pivotschrittes bestimmt, wenn mehrere zur Auswahl stehen, heißt *Pivotregel*. Zahlreiche solche Regeln wurden vorgeschlagen, für fast alle von ihnen kennt man aber mittlerweile Instanzen, auf denen der Simplex-Algorithmus exponentiell viele Pivotschritte benötigt. Ob es eine Pivotregel gibt, die garantiert, dass der Simplex-Algorithmus nach polynomiell vielen Schritten terminiert, ist bis heute unklar.

6.3 Komplexität von linearer Programmierung

Lineare Programmierung ist ein sehr gut untersuchtes Thema der Optimierung, und für lange Zeit war es eine große offene Frage, ob es Algorithmen gibt, die lineare Programme in polynomieller Zeit lösen. Diese Frage wurde 1979 von Leonid Khachiyan positiv beantwortet. Er stellte die *Ellipsoidmethode* vor, die in polynomieller Zeit testet, ob ein gegebenes Lösungspolyeder $\mathcal{P}$ leer ist oder nicht. Die Ellipsoidmethode erzeugt zunächst ein Ellipsoid (die mehrdimensionale Entsprechung einer Ellipse), das $\mathcal{P}$ komplett enthält. In jeder Iteration wird dann getestet, ob das Zentrum des Ellipsoides in $\mathcal{P}$ liegt. Falls ja, so ist $\mathcal{P}$ nicht leer. Ansonsten wird ein neues Ellipsoid berechnet, das $\mathcal{P}$ immer noch enthält, dessen Volumen, aber um einen gewissen Faktor kleiner ist als das Volumen des alten Ellipsoides. Man kann nachweisen, dass nach polynomiell vielen Iterationen entweder ein Punkt in $\mathcal{P}$ gefunden wird oder das aktuelle Ellipsoid so klein geworden ist, dass $\mathcal{P} = \emptyset$ gelten muss.

Eine beliebige Variablenbelegung $x \in \mathcal{P}$ zu berechnen oder zu entscheiden, dass keine solche existiert, ist ein scheinbar einfacheres Problem, als die optimale Lösung eines linearen Programms zu bestimmen. Tatsächlich gilt aber, dass sich ein polynomieller Algorithmus $\mathcal{A}$ für das erste Problem in einen polynomiellen Algorithmus für das zweite Problem transformieren lässt: Aus einem linearen Programm, in dem die Zielfunktion $c \cdot x$ unter den Nebenbedingungen $Ax \leq b$ minimiert werden soll, erzeugen wir ein neues lineares Programm mit den Nebenbedingungen $Ax \leq b$ und zusätzlich $c \cdot x \leq z$ für ein $z \in \mathbb{R}$. Ist das zu diesem linearen Programm gehörige Lösungspolyeder nicht leer, so besitzt das ursprüngliche lineare Programm eine Lösung mit Wert höchstens z. Wir suchen jetzt mithilfe von Algorithmus $\mathcal{A}$ und einer binären Suche das kleinste z, für das das oben genannte lineare Programm eine Lösung besitzt. Für dieses lineare Programm bestimmen wir dann mithilfe von $\mathcal{A}$ eine zulässige Lösung. Diese Lösung entspricht dann einer optimalen Lösung des ursprünglichen linearen Programms.

Obwohl die Ellipsoidmethode ein großer theoretischer Durchbruch war, ist ihr Einfluss auf die Praxis gering, denn in praktischen Anwendungen ist der Simplex-Algorithmus trotz seiner exponentiellen Worst-Case-Laufzeit der polynomiellen Ellipsoidmethode deutlich überlegen. Im Jahre 1984 gelang es Narendra Karmarkar mit dem *Innere-Punkte-Verfahren* einen polynomiellen Algorithmus zur Lösung von linearen Programmen vorzustellen, der auch in einigen praktischen Anwendungen ähnlich gute Laufzei-

ten wie der Simplex-Algorithmus erzielt. Dennoch ist der Simplex-Algorithmus auch heute noch der am meisten genutzte Algorithmus zur Lösung von linearen Programmen.

6.4 Ganzzahlige lineare Programme

Da man lineare Programme in polynomieller Zeit optimal lösen kann, ist nicht zu erwarten, dass wir ein NP-schweres Problem ohne Weiteres als lineares Programm formulieren können. Bei genauerer Betrachtung fällt jedoch auf, dass wir viele NP-schwere Problem in ähnlicher Form darstellen können. Nehmen wir das Rucksackproblem als Beispiel. Eine gegebene Instanz mit Nutzenwerten $p_1, \ldots, p_n$, Gewichten $w_1, \ldots, w_n$ und Kapazität t können wir wie folgt darstellen:

$$\begin{aligned} \text{maximiere} \quad & p_1 x_1 + \cdots + p_n x_n \\ \text{sodass} \quad & w_1 x_1 + \cdots + w_n x_n \leq t, \\ & \forall i : x_i \in \{0, 1\}. \end{aligned}$$

Die obige Darstellung entspricht fast einem linearen Programm: Die Variable x_i legt fest, ob das Objekt i in den Rucksack gepackt wird ($x_i = 1$) oder nicht ($x_i = 0$). Der Gesamtnutzen der eingepackten Objekte wird mit einer linearen Zielfunktion dargestellt, die es zu maximieren gilt, und es gibt eine lineare Nebenbedingung, die dafür sorgt, dass die ausgewählte Lösung, die Kapazität einhält. Im Unterschied zu den linearen Programmen, die wir bislang gesehen haben, können die Variablen x_i nicht mehr beliebige reelle Werte in einem Intervall (beispielsweise in $[0, 1]$) annehmen, sondern nur noch die beiden ganzzahligen Werte 0 und 1 erlaubt sind. Während dies auf den ersten Blick wie eine kleine Änderung aussehen mag, ist es ein ganz entscheidender Unterschied. Hätten wir reelle Werte erlaubt und $x_i \in [0, 1]$ anstatt $x_i \in \{0, 1\}$ gefordert, so hätten wir das resultierende Problem in polynomieller Zeit mit den in Abschnitt 6.3 beschriebenen Algorithmen lösen können. Dies entspricht dem Rucksackproblem mit teilbaren Objekten, das wir bereits in Algorithmen und Berechnungskomplexität I kennengelernt haben. Die Variable x_i gibt dann an, zu welchem Anteil Objekt i in den Rucksack gepackt wird. Wir haben bereits gesehen, dass diese Variante des Rucksackproblems effizient mithilfe eines Greedy-Algorithmus optimal gelöst werden kann.

Lineare Programme, in denen die Variablen nur ganzzahlige Werte annehmen dürfen, nennt man *ganzzahlige lineare Programme*. Das obige Beispiel zeigt bereits, dass es im Allgemeinen NP-schwer ist, die optimale Lösung eines ganzzahligen linearen Programms zu finden. Wird an einige der Variablen die Anforderung gestellt, dass sie nur ganzzahlige Werte annehmen dürfen, während die anderen Variablen weiterhin reelle Werte annehmen dürfen, so spricht man von einem *gemischt-ganzzahligen linearen Programm*. Im Folgenden werden wir diese Unterscheidung nicht explizit machen und nur von ganzzahligen linearen Programmen sprechen. Dies soll jedoch nicht ausschließen, dass einzelne Variablen auch beliebige reelle Werte annehmen dürfen. Bevor wir besprechen, warum es hilfreich sein kein, NP-schwere Probleme als ganzzahlige oder gemischt-ganzzahlige lineare Programme darzustellen, schauen wir uns noch einige weitere Beispiele an.

Vertex Cover

Beim Vertex-Cover-Problem ist als Eingabe ein ungerichteter Graph $G = (V, E)$ gegeben und gesucht ist eine möglichst kleine Auswahl $V' \subseteq V$ von Knoten, sodass jede Kante aus E zu mindestens einem Knoten aus V' inzident ist. Eine Instanz dieses Problems mit $V = \{1, \ldots, n\}$ können wir wie folgt als ganzzahliges lineares Programm darstellen:

$$\begin{aligned} \text{minimiere} \quad & x_1 + \cdots + x_n \\ \text{sodass} \quad & \forall e = (i, j) \in E : x_i + x_j \geq 1, \\ & \forall i : x_i \in \{0, 1\}. \end{aligned}$$

Dieses Programm enthält für jeden Knoten $i \in V$ eine Variable $x_i \in \{0, 1\}$, die angibt, ob er in der Auswahl V' enthalten ist ($x_i = 1$). Die zu minimierende Zielfunktion zählt die ausgewählten Knoten, während es für jede Kante eine Nebenbedingung gibt, dass mindestens einer der beiden Endknoten der Kante in der Menge V' enthalten sein muss.

Clique

Beim Cliquenproblem ist als Eingabe ein ungerichteter Graph $G = (V, E)$ gegeben und gesucht ist eine möglichst große Clique V'. Eine Instanz dieses Problems mit $V = \{1, \ldots, n\}$ können wir wie folgt als ganzzahliges lineares Programm darstellen:

$$\begin{aligned} \text{maximiere} \quad & x_1 + \cdots + x_n \\ \text{sodass} \quad & \forall (i, j) \notin E : x_i + x_j \leq 1, \\ & \forall i : x_i \in \{0, 1\}. \end{aligned}$$

Dieses Programm enthält für jeden Knoten $i \in V$ eine Variable $x_i \in \{0, 1\}$, die angibt, ob er in der Clique V' enthalten ist ($x_i = 1$). Die zu maximierende Zielfunktion zählt die ausgewählten Knoten. Für jedes Knotenpaar, das nicht durch eine Kante verbunden ist, gibt es eine Nebenbedingung, dass höchstens einer der beiden Knoten in der Menge V' enthalten sein darf.

Scheduling auf identischen Maschinen

Wir betrachten nun das Problem Scheduling auf identischen Maschinen. Dazu werden wir ein gemischt-ganzzahliges lineares Programm mit einer reellwertigen Variable und weiteren binären Variablen angeben. Sei eine Eingabe mit Jobgrößen $p_1, \ldots, p_n \in \mathbb{R}_{>0}$ und m Maschinen gegeben. Diese können wir wie folgt als

gemischt-ganzzahliges lineares Programm darstellen:

$$\begin{aligned} \text{minimiere} \quad & t \\ \text{sodass} \quad & \forall i \in \{1,\dots,m\} : \sum_{j=1}^{n} p_j x_{ij} \leq t, \\ & \forall j \in \{1,\dots,n\} : \sum_{i=1}^{m} x_{ij} = 1, \\ & \forall i,j : x_{ij} \in \{0,1\}. \end{aligned}$$

In diesem Programm ist t eine reellwertige Variable, die den Makespan codiert. Die Variable x_{ij} codiert binär, ob Job j Maschine i zugewiesen wird, und die Nebenbedingungen stellen sicher, dass die Ausführungszeit jeder Maschine höchstens dem Makespan entspricht und jeder Job einer Maschine zugewiesen wird. Dadurch, dass t minimiert wird, ist sicher gestellt, dass in einer optimalen Lösung t genau dem Makespan entspricht.

TSP

Als letztes Beispiel betrachten wir das TSP. Sei eine Instanz mit $V = \{1,\dots,n\}$ gegeben und bezeichne d_{ij} die Distanz von $i \in V$ zu $j \in V$. Hier ist es weniger offensichtlich, wie eine solche Instanz als ganzzahliges lineares Programm dargestellt werden kann. Dennoch ist es möglich. Dazu führen wir eine binäre Variable x_{ij} für jedes Paar $i,j \in V$ mit $i \neq j$ ein. Diese Variable codiert, ob auf der Tour die Kante von i nach j vorkommt. Dann können wir das folgende lineare Programm aufstellen, das allerdings noch nicht exakt das TSP beschreibt:

$$\begin{aligned} \text{minimiere} \quad & \sum_{i=1}^{n} \sum_{j \neq i, j=1}^{n} d_{ij} x_{ij} \\ \text{sodass} \quad & \forall j \in \{1,\dots,n\} : \sum_{i \neq j, i=1}^{n} x_{ij} = 1, \\ & \forall j \in \{1,\dots,n\} : \sum_{i \neq j, i=1}^{n} x_{ji} = 1, \\ & \forall i,j : x_{ij} \in \{0,1\}. \end{aligned}$$

Die Zielfunktion entspricht genau den Gesamtkosten der ausgewählten Kanten und die Nebenbedingungen stellen sicher, dass von jedem Knoten genau eine ausgehende und eine eingehende Kante ausgewählt wurde. Jede Tour erfüllt diese Nebenbedingungen, aber man stellt schnell fest, dass dieses lineare Programm auch Lösungen erlaubt, die gar keine Touren sind. Jede Menge von paarweise disjunkten Kreisen, die insgesamt jeden Knoten abdecken, ist eine gültige Lösung. Insofern müssen wir weitere Nebenbedingungen hinzufügen, um das TSP zu codieren. Dazu benötigen wir auch zusätzliche Variablen. Wir führen für jeden Knoten $i \in V$ mit $i \neq 1$ eine ganzzahlige Variable u_i ein. Diese kann Werte aus der Menge $\{1,\dots,n-1\}$ annehmen und soll codieren, an welcher Stelle der Knoten i in der

Tour vorkommt. Dabei gehen wir davon aus, dass Knoten 1 an Stelle 0 kommt. Wir möchten codieren, dass u_j größer als u_i sein muss, wenn die Kante (i, j) in der Tour enthalten ist. Dies erreichen wir durch die folgenden Nebenbedingungen:

$$\begin{aligned} &\forall i, j \in \{2, \ldots, n\}, i \neq j : u_i - u_j + n x_{ij} \leq n - 1, \\ &\forall i \in \{2, \ldots, n\} : u_i \in \{1, \ldots, n - 1\}. \end{aligned} \tag{6.4}$$

Betrachten wir die Nebenbedingung für ein Paar i und j genauer. Gilt $x_{ij} = 0$, so entspricht sie der Bedingung $u_i - u_j \leq n - 1$, die für jede Belegung der Variablen u_i und u_j aus $\{1, \ldots, n - 1\}$ automatisch erfüllt ist. Gilt $x_{ij} = 1$, so entspricht sie der gewünschten Bedingung $u_i \leq u_j - 1$.

Aus einer Tour für die Instanz des TSP erhalten wir eine Lösung für das ganzzahlige lineare Programm, indem wir mit den x_{ij}-Variablen die Kanten auf der Tour codieren und in den u_i-Variablen für jeden Knoten speichern, an welcher Stelle er in der Tour vorkommt. Hierfür ist es wichtig, dass die Bedingung (6.4) nur für $i \neq 1$ und $j \neq 1$ gelten muss. Das bedeutet insbesondere, dass die letzte Kante der Tour, die zu Knoten 1 zurückführt, durch diese Bedingungen nicht verboten ist.

Andersherum kann jede Lösung des ganzzahligen linearen Programms in eine Tour übersetzt werden. Wir haben oben bereits argumentiert, dass die Bedingungen sicherstellen, dass eine gültige Lösung des ganzzahligen linearen Programms eine Menge von disjunkten Kreisen codiert, die gemeinsam alle Knoten des Graphen abdecken. Die zusätzlichen Bedingungen (6.4), stellen dazu sicher, dass ein Kreis nur dann erlaubt ist, wenn er den Knoten 1 enthält. Für einen Kreis, der Knoten 1 nicht enthält, stellen diese Bedingungen nämlich sicher, dass die u_i-Werte entlang jeder Kante des Kreises streng monoton steigen müssen. Dies führt direkt zu einem Widerspruch. Da die Kreise paarweise disjunkt sind und es damit nur einen einzigen Kreis gibt, der Knoten 1 enthält, besteht die Lösung aus nur einem Kreis.

Die vorangegangenen Beispiele machen deutlich, dass ganzzahlige lineare Programme ein sehr mächtiges Werkzeug sind, um NP-schwere Probleme zu beschreiben. Warum ist das aber überhaupt interessant? Stellen wir eine Instanz eines NP-schweren Problems als ganzzahliges lineares Programm dar, so haben wir das Problem zunächst nur umformuliert, aber nicht gelöst. Dies ist aber deswegen interessant, da es zahlreiche Softwarepakete (sogenannte *Solver*) gibt, um ganzzahlige lineare Programme optimal zu lösen. Da dies im Allgemeinen NP-schwer ist, darf man nicht erwarten, dass diese Pakete immer effizient eine optimale Lösung finden. Natürlich gibt es ganzzahlige lineare Programme, auf denen sie exponentielle Rechenzeit benötigen. In die Optimierung dieser Softwarepakete sind jedoch teils jahrzehntelange Arbeit und Forschung geflossen, was zur Konsequenz hat, dass sie in vielen Anwendungen optimale Lösungen sehr viel schneller finden als beispielsweise naive Brute-Force-Algorithmen, die alle möglichen Lösungen betrachten. In vielen Fällen ist es mithilfe dieser Softwarepakete sogar möglich, große ganzzahlige Programme vergleichsweise schnell zu lösen.

Die gängigen Solver sind zwar stark optimiert, allerdings sind sie für allgemeine ganzzahlige lineare Programme ausgelegt und nicht für spezielle Probleme optimiert. In-

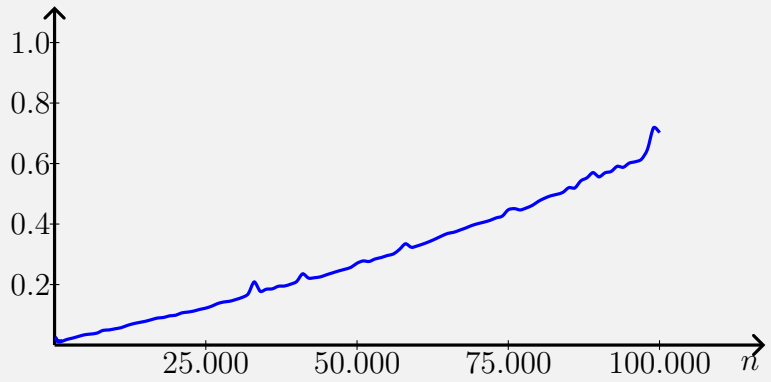
sofern sollte man sich bei jedem Problem fragen, ob es einen problemspezifischen Algorithmus gibt, der es schneller lösen kann als ein allgemeiner Solver für ganzzahlige Programme. Oft ist es jedoch aufwendig, solche problemspezifischen Algorithmen zu entwerfen. Noch dazu sind Formulierungen als ganzzahlige Programme sehr flexibel. Beispielsweise können einfach neue Nebenbedingungen hinzugenommen werden, was bei problemspezifischen Algorithmen in der Regel nicht der Fall ist. Deshalb spielt gerade bei komplexen Optimierungsproblemen ganzzahlige Programmierung oft eine wichtige Rolle.

Es gibt zahlreiche Solver für ganzzahlige lineare Programme. Zu den bekanntesten kommerziellen Paketen gehören Gurobi[2] und CPLEX[3]. Diese können im Rahmen von akademischen Lizenzen von Studierenden kostenfrei genutzt werden. Ein bekanntes Beispiel für einen nicht-kommerziellen Solver ist SCIP[4]. Diese Pakete sind mittlerweile recht benutzerfreundlich gestaltet und stellen einfach zu nutzende Schnittstellen zu einer Vielzahl bekannter Programmiersprachen zur Verfügung. Den Leserinnen und Lesern sei empfohlen sich mit der Nutzung eines dieser Pakete vertraut zu machen.

Exemplarische Rechenzeiten

Um ein Gefühl dafür zu vermitteln, wie mächtig gängige Solver für ganzzahlige lineare Programme sind, haben wir mit Gurobi einige Tests für das Rucksackproblem, das Cliquenproblem sowie das TSP durchgeführt. Für all diese Probleme haben wir zufällige Eingaben erzeugt und die Laufzeit betrachtet, die Gurobi benötigt, um optimale Lösungen auf diesen zufälligen Eingaben zu berechnen. Die Experimente wurden auf einem herkömmlichen Desktop-PC durchgeführt, dessen genaue Spezifikation keine Rolle spielt, da es uns nur um einen groben Eindruck geht, was mit Gurobi und vergleichbarer Software möglich ist.

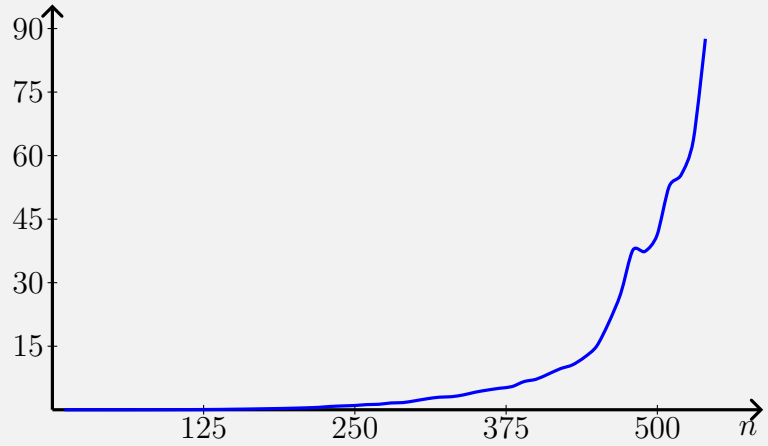
Betrachten wir als erstes das Rucksackproblem. Wir haben Instanzen mit n Objekten erzeugt, wobei jeder Nutzen p_i und jedes Gewicht w_i uniform zufällig aus dem Intervall $[0, 1]$ gewählt wurde. Die Kapazität des Rucksacks haben wir auf $n/4$ gesetzt. Wir haben $n = 50, 100, 150, \ldots, 100.000$ gewählt und 100 Durchläufe für jedes n durchgeführt. Im folgenden Diagramm ist für jedes n die durchschnittliche Laufzeit in Sekunden über die 100 Durchläufe aufgetragen.



[2]https://www.gurobi.com/
[3]https://www.ibm.com/de-de/analytics/cplex-optimizer
[4]https://www.scipopt.org/

Selbst für Eingaben mit 100.000 Objekten beträgt die durchschnittliche Laufzeit, die Gurobi zur Berechnung einer optimalen Lösung benötigt, weniger als eine Sekunde. Noch dazu scheint die Laufzeit nur in etwa linear mit n zu wachsen. Dies ist für ein NP-schweres Optimierungsproblem ein beachtliches Ergebnis, das auf den ersten Blick der NP-Schwere zu widersprechen scheint. Tatsächlich weiß man aber bereits aus theoretischen Analysen, dass Eingaben für das Rucksackproblem, in denen alle Nutzenwerte und Gewichte zufällig gewählt sind, Eingaben sind, die mit hoher Wahrscheinlichkeit effizient gelöst werden können. Es gibt auch Eingaben für das Rucksackproblem mit wenigen Objekten, für deren Lösung Gurobi eine sehr hohe Rechenzeit benötigt. Diese werden allerdings bei zufälliger Wahl der Nutzenwerte und Gewichte nur mit sehr geringer Wahrscheinlichkeit realisiert.

Als nächstes betrachten wir die Laufzeit zur Lösung von Clique. Hierfür haben wir Graphen mit n Knoten getestet und jede der $\binom{n}{2}$ vielen möglichen Kanten mit einer Wahrscheinlichkeit von $1/2$ in den Graphen eingefügt. Für $n = 10, 20, 30, \ldots, 540$ haben wir jeweils 100 Durchläufe gemacht. Im folgenden Diagramm ist wieder die durchschnittliche Laufzeit in Sekunden abgetragen.



Man beobachtet, dass die Laufzeiten deutlich größer sind als beim Rucksackproblem und dass sie auch deutlich schneller ansteigen. Die Entwicklung der Kurve deutet auf einen exponentiellen Anstieg hin. Positiv kann aber festgehalten werden, dass selbst für $n = 540$ optimale Lösungen im Durchschnitt noch in weniger als 90 Sekunden gefunden werden können.

Als letztes Beispiel haben wir auch das oben beschriebene ganzzahlige lineare Programm für das TSP in Gurobi gelöst. Dabei haben wir Instanzen mit n Knoten erzeugt und für jedes Knotenpaar einen zufälligen Abstand aus dem Intervall $[0, 1]$ gewählt. Die resultierenden Abstände waren also symmetrisch, erfüllten aber im Allgemeinen nicht die Dreiecksungleichung. Wir haben Experimente bis $n = 80$ durchgeführt und festgestellt, dass die durchschnittlichen Laufzeiten für das TSP nicht besonders aussagekräftig sind, da die Varianz sehr hoch ist. Für $n = 80$ konnten die meisten Instanzen in weniger als 10 Sekunden optimal gelöst werden. Bei anderen Instanzen haben wir die Berechnung dafür nach über 4 Stunden ohne Ergebnis abgebrochen. Anders als beim Rucksackproblem und dem Cliquenproblem trifft man beim TSP also bereits für kleine n auf schwierige zufällige Instanzen.

Auch wenn die vorangegangenen Beispiele optimistisch stimmen, sollte man nicht den Fehler machen zu glauben, dass jedes NP-schwere Problem mit den gängigen Solvern für ganzzahlige lineare Programme schnell gelöst werden kann. Für jedes der untersuchten Probleme gibt es auch Instanzen mit wenigen Objekten bzw. kleinen Graphen, auf denen die Solver exponentielle Rechenzeit zur Berechnung der optimalen Lösung benötigen. Die zufälligen Eingaben, die wir erzeugt haben, sind aber im Allgemeinen keine solchen Worst-Case-Eingaben. Oft haben zufällige Eingaben eine spezielle Struktur, die es Solvern vergleichsweise einfach macht, sie zu lösen. Die größte Clique in den zufälligen Graphen, die wir erzeugt haben, ist beispielsweise mit hoher Wahrscheinlichkeit sehr klein, was eine hilfreiche Eigenschaft ist, um sie zu finden. Insofern haben Experimente auf zufälligen Eingaben auch nur bedingt Aussagekraft für Eingaben in echten Anwendungen und schon gar nicht für das Worst-Case-Verhalten.

Auch wenn man die Optimierung eines ganzzahligen Programms aus Zeitgründen abbricht, erhält man oft bereits eine gute Lösung als Ausgabe und eine Schranke, wie gut diese Lösung verglichen zur optimalen Lösung ist. Bei den oben erwähnten Instanzen des TSP, bei denen wir die Berechnung nach 4 Stunden abgebrochen haben, hatte Gurobi beispielsweise bereits eine Lösung gefunden, die bewiesenermaßen nur wenige Prozent länger war als eine optimale Tour.

Zum Abschluss dieses Kapitels sei noch kurz erwähnt, dass es nicht nur Solver für ganzzahlige lineare Programme gibt, sondern auch für SAT. Ähnlich wie bei ganzzahligen Programmen sind auch diese Solver heutzutage heuristisch stark optimiert und können oft für große Formeln erfüllende Belegungen finden. Hat man also ein Problem vorliegen, das man auf SAT reduzieren kann, so ist es einen Versuch wert, praktisch relevante Eingaben mithilfe eines SAT Solvers zu lösen.

Kapitel 7

Exponentialzeitalgorithmen

Unter der Annahme P $\neq$ NP gibt es für NP-schwere Probleme keine polynomiellen Algorithmen. Unter der weitergehenden Exponentialzeithypothese gilt für viele NP-schwere Probleme sogar, dass jeder Algorithmus zur Lösung eines dieser Probleme im Worst Case exponentielle Rechenzeit benötigt. Diese erreicht man für die meisten NP-schweren Probleme bereits auf einfache Art und Weise durch eine vollständige Aufzählung aller möglichen Lösungen. In diesem Kapitel werden wir uns mit cleveren Exponentialzeitalgorithmen beschäftigen, die bessere Laufzeiten als naive Brute-Force-Methoden erreichen.

7.1 Held–Karp-Algorithmus

Als erstes betrachten wir das TSP. Ist eine Instanz mit n Knoten gegeben, so gibt es nur höchstens $n!$ viele verschiedene Touren. Diese kann ein Brute-Force-Algorithmus alle einmal betrachten, jeweils ihre Länge berechnen und am Ende die kürzeste Tour ausgeben. Der Wert von $n!$ ist bereits für sehr kleine Werte von n astronomisch groß, sodass dieser Algorithmus vollkommen unpraktikabel ist. Bereits für $n = 15$ ist $n!$ größer als eine Billion. Wir lernen nun einen Algorithmus kennen, dessen Laufzeit mit 2^n statt $n!$ wächst. Dieser kann zumindest noch für etwas größere Werte von n ausgeführt werden. Die Schwelle von einer Billion wird beispielsweise erst für $n = 40$ erreicht. Auch wenn dies sicherlich ebenfalls nicht die Laufzeit ist, die wir uns zur Lösung des TSP erhoffen, können wir wegen der NP-Schwere vermutlich keine deutlich bessere Worst-Case-Laufzeit erwarten.

Wir betrachten den Held-Karp-Algorithmus, der dynamische Programmierung zur Lösung des TSP einsetzt. Wie immer müssen wir uns bei dem Einsatz von dynamischer Programmierung zunächst Gedanken darüber machen, wie eine gegebene Instanz sinnvoll rekursiv in Teilprobleme zerlegt werden kann. Dazu überlegen wir uns, welche Struktur eine optimale Tour aufweisen muss. Sei $V = \{1, \ldots, n\}$ mit $n \geq 3$ die Menge der Knoten und seien Distanzen $d_{ij} \in \mathbb{R}_{\geq 0}$ gegeben. Dabei sei wie üblich $d_{ii} = 0$ für jeden Knoten i und $d_{ij} > 0$ für $i \neq j$.

Wir betrachten im Folgenden den Knoten 1 als den Startknoten, von dem die Tour ausgeht und der am Ende wieder erreicht werden soll. Sei C eine optimale Tour und sei j der letzte Knoten auf dieser Tour, der vor der Rückkehr zu Knoten 1 besucht wird. Entfernen wir aus der Tour C die letzte Kante $(j, 1)$, so erhalten wir einen Weg von Knoten 1 zu Knoten j, der jeden Knoten aus V genau einmal besucht. Insbesondere muss es sich dabei um einen kürzesten solchen Weg handeln, denn gäbe es einen kürzeren Weg von 1 nach j, der jeden Knoten genau einmal enthält, so könnte dieser mit der Kante $(j, 1)$ zu einer kürzeren Tour als C vervollständigt werden. Bezeichne im Folgenden $D_{V,j}$ die Länge eines kürzesten 1-j-Weges, der jeden Knoten aus V genau einmal enthält. Dann können wir die Länge der optimalen Tour als

$$\min_{j \in V \setminus \{1\}} (D_{V,j} + d_{j1})$$

schreiben.

Wie bestimmen wir nun $D_{V,j}$? Auch hier greifen wir uns wieder den vorletzten Knoten heraus. Sei also i ein Knoten, der auf einem kürzesten 1-j-Weg, der alle Knoten aus V genau einmal besucht, direkt vor dem Knoten j besucht wird. Dann ist der Teilweg von Knoten 1 zu Knoten i ein 1-i-Weg, der jeden Knoten aus $V \setminus \{j\}$ genau einmal besucht und Knoten j nicht besucht. Wir können genau wie oben argumentieren, dass es sogar ein kürzester solcher Weg ist: Gäbe es einen kürzeren 1-i-Weg, der jeden Knoten aus $V \setminus \{j\}$ genau einmal besucht und Knoten j nicht besucht, so könnte dieser mit der Kante (i, j) zu einem kürzeren 1-j-Weg erweitert werden, der jeden Knoten aus V genau einmal besucht.

Die obige Diskussion legt die folgende Definition von Teilproblemen nahe.

Definition 7.1. *Sei $S \subseteq \{1, \ldots, n\}$ mit $1 \in S$. Für jedes $j \in S \setminus \{1\}$ sei $D_{S,j}$ die Länge eines kürzesten Weges von Knoten 1 zu Knoten j, der jeden Knoten aus S genau einmal besucht und keine Knoten aus $V \setminus S$ enthält.*

Wir können die Werte $D_{S,j}$ rekursiv wie folgt berechnen.

Lemma 7.2. *Sei $S \subseteq \{1, \ldots, n\}$ mit $1 \in S$ und $|S| \geq 3$. Für $j \in S \setminus \{1\}$ gilt*

$$D_{S,j} = \min_{k \in S \setminus \{1,j\}} (D_{S \setminus \{j\},k} + d_{kj}).$$

Beweis. Das Lemma folgt mit dem gleichen Argument, das wir oben bereits zweimal benutzt haben. Sei P ein kürzester 1-j-Weg, der nur Knoten aus S besucht und jeden davon genau einmal. Dann beträgt die Länge von P per Definition genau $D_{S,j}$. Sei k der vorletzte Knoten auf dem Weg P, also der Knoten, von dem aus Knoten j erreicht wird. Sei P' der Teilweg von P, der von Knoten 1 zu Knoten k führt. Dann muss P' ein kürzester 1-k-Weg sein, der nur Knoten aus $S \setminus \{j\}$ enthält und jeden davon genau einmal. Gäbe es nämlich einen kürzeren solchen Weg, so könnte er mit der Kante (k, j) zu einem 1-j-Weg verlängert werden, der nur Knoten aus S und jeden davon genau einmal besucht. Dieser Weg wäre im Widerspruch zur Definition von P kürzer als P. Insofern beträgt die Länge von P' genau $D_{S \setminus \{j\},k}$. Das Lemma folgt, da die Länge von P der Summe der Länge von P' und der Länge der Kante (k, j) entspricht. Um den richtigen Knoten k zu finden, testen wir einfach alle Möglichkeiten aus $k \in S \setminus \{1, j\}$ und nehmen diejenige, die zum insgesamt kürzesten Weg führt. □

Den Basisfall der Rekursion bilden die Mengen S mit $|S| = 2$. Sei $S = \{1, j\}$ für einen Knoten $j \in \setminus\{1\}$. Dann gilt $D_{S,j} = d_{1j}$, da der einzige erlaubte Weg in diesem Teilproblem die direkte Kante von Knoten 1 zu Knoten j ist.

Basierend auf diesem Basisfall und Lemma 7.2 können wir nun die Werte $D_{S,j}$ für alle Teilmengen S und jeden Knoten j wie im folgenden Pseudocode berechnen. Dabei lösen wir die Teilprobleme für aufsteigende Größe der Menge S.

Held-Karp-Algorithmus

1. **for** $(j = 2;\ j \leq n;\ j{+}{+})$ $D_{\{1,j\},j} = d_{1j}$;
2. **for** $(s = 3;\ i \leq n;\ s{+}{+})$
3. **for each** $(S \subseteq V$ mit $|S| = s$ und $1 \in S)$
4. **for each** $(j \in S \setminus \{1\})$
5. $D_{S,j} = \min_{k \in S \setminus \{1,j\}}(D_{S\setminus\{j\},k} + d_{kj})$;
6. **return** $\min_{j \in V \setminus \{1\}}(D_{V,j} + d_{j1})$;

Theorem 7.3. *Für eine Eingabe für das TSP mit n Knoten berechnet der Held-Karp-Algorithmus in Laufzeit $O(n^2 2^n)$ die Länge einer kürzesten Tour.*

Beweis. Die Korrektheit folgt aus Lemma 7.2 und den weiteren Vorüberlegungen, die wir gemacht haben. Am Ende des Algorithmus entspricht der Wert $D_{V,j}$ der Länge eines kürzesten Weges von Knoten 1 zu Knoten j, der jeden Knoten genau einmal enthält. Dieser kann mit der Kante $(j, 1)$ zu einer Tour der Länge $D_{V,j} + d_{j1}$ erweitert werden. Für ein j muss dies der Länge der kürzesten Tour entsprechen. Da in Zeile 6 das j gesucht wird, das diesen Ausdruck minimiert, folgt die Korrektheit.

Zur Laufzeit beobachten wir, dass es $O(n2^n)$ viele Teilprobleme gibt, da es 2^n Teilmengen von S und $n-1$ Wahlen für j gibt. Jedes Teilproblem, das kein Basisproblem ist, wird durch die Bildung eines Minimums über weniger als n Terme in Zeile 5 gelöst. Damit ergibt sich eine Gesamtlaufzeit von $O(n^2 2^n)$. □

Wie üblich bei dynamischer Programmierung haben wir hier nur einen Algorithmus beschrieben, der die Länge einer optimalen Tour ausgibt, nicht aber die Tour selbst. Der Algorithmus kann aber leicht so abgeändert werden, dass er auch die Tour selbst berechnet. Dazu muss lediglich in den Zeilen 5 und 6 jeweils noch zusätzlich abgespeichert werden, für welches k bzw. j das Minimum angenommen wird. Mithilfe dieser Zusatzinformationen kann eine optimale Tour rekonstruiert werden.

7.2 Parametrisierte Algorithmen

Als nächstes betrachten wir noch einmal das Vertex-Cover-Problem. Wir betrachten der Einfachheit halber die Entscheidungsvariante, alle unsere Überlegungen können aber auch auf die Optimierungsvariante übertragen werden. Seien als Eingabe also ein

ungerichteter Graph $G = (V, E)$ und eine Zahl $k \in \mathbb{N}$ gegeben. Das Ziel ist es, zu entscheiden, ob es eine Teilmenge $V' \subseteq V$ der Knoten mit $|V'| \leq k$ gibt, sodass jede Kante aus E inzident zu mindestens einem Knoten aus V' ist.

Wir haben gesehen, dass das Vertex-Cover-Problem NP-schwer ist, deshalb können wir auch für dieses Problem nicht auf polynomielle Algorithmen hoffen. Es gibt 2^n viele Teilmengen von V, insofern besitzt ein naiver Brute-Force-Algorithmus für das Vertex-Cover-Problem eine Laufzeit von $O(2^n \cdot p(n))$ für ein geeignetes Polynom p. Um die Entscheidungsvariante für gegebenes k zu lösen, genügt es aber, sich nur Teilmengen der Größe k anzuschauen. Davon gibt es $\binom{n}{k} = O(n^k)$ viele. Ist k klein, so ist dies deutlich besser als sich alle möglichen Teilmengen von V anzuschauen. Allerdings ist auch eine Laufzeit in der Größenordnung n^k bereits für kleine k wie zum Beispiel $k = 10$ vollkommen unpraktikabel.

Wir werden nun Algorithmen kennenlernen, die für kleine k eine deutlich bessere Laufzeit erreichen. Dazu entwerfen wir einen rekursiven Algorithmus, den wir VC-Backtracking nennen. Dieser Algorithmus erhält als Eingabe einen Graphen G sowie eine Zahl k und er soll entscheiden, ob G ein Vertex Cover der Größe k besitzt. Dazu wählt er zunächst eine beliebige Kante $e = (x, y)$ des Graphen. Da in einem Vertex Cover insbesondere die Kante e abgedeckt ist, muss jedes Vertex Cover entweder den Knoten x oder den Knoten y enthalten (oder beide). Ist der Knoten x in einem Vertex Cover V' der Größe k enthalten, so müssen die anderen Knoten aus V' alle Kanten des Graphen G abdecken, die nicht zu x inzident sind. Wir entfernen deshalb den Knoten x und alle zu ihm inzidenten Kanten aus dem Graphen und testen, ob der so resultierende Graph (formal ist das der Graph, der von $V \setminus \{x\}$ induziert wird) ein Vertex Cover der Größe $k - 1$ besitzt. Ist dies der Fall, so besitzt der Graph G insgesamt ein Vertex Cover der Größe k. Analog gilt diese Überlegung für den Fall, dass der Knoten y in einem Vertex Cover der Größe k enthalten ist. Aus dieser Überlegung ergibt sich der folgende Algorithmus.

VC-Backtracking($\mathbf{G = (V, E)}, \mathbf{k}$)

1. **if** ($|E| = 0$) **return** true;
2. **if** ($k = 0$) **return** false;
3. Sei $e = (x, y) \in E$ beliebig.
4. Sei G_x der von $V \setminus \{x\}$ induzierte Teilgraph.
5. Sei G_y der von $V \setminus \{y\}$ induzierte Teilgraph.
6. **if** (VC-Backtracking($G_x, k - 1$)) **return** true;
7. **if** (VC-Backtracking($G_y, k - 1$)) **return** true;
8. **return** false;

Theorem 7.4. *Der Algorithmus VC-Backtracking entscheidet in Zeit $O(2^k(n + m))$, ob der Graph G ein Vertex Cover der Größe k besitzt.*

Beweis. Die Korrektheit folgt aus unseren Vorüberlegungen: Der Graph G besitzt genau dann ein Vertex Cover der Größe k, wenn entweder G_x oder G_y ein Vertex

Cover der Größe $k-1$ besitzt. Auch die Randfälle werden korrekt überprüft: Ist die Menge E leer, so besitzt der Graph G für jedes $k \geq 0$ ein Vertex Cover der Größe k (Zeile 1). Ist die Menge E nichtleer, so besitzt der Graph kein Vertex Cover der Größe 0 (Zeile 2).

Die Laufzeit des Algorithmus können wir wie folgt beschränken. Sei G ein Graph mit n Knoten und m Kanten. Der Rekursionsbaum des Algorithmus besitzt eine Tiefe von k und jeder Knoten besitzt höchstens zwei Kinder, da in jedem Aufruf maximal zwei rekursive Aufrufe erfolgen. Somit gibt es insgesamt höchstens $\sum_{i=0}^{k} 2^i = O(2^k)$ Aufrufe von VC-Backtracking. Jeder dieser Aufrufe benötigt eine Laufzeit von $O(n+m)$, da insbesondere die Graphen G_x und G_y in dieser Zeit erstellt werden können, wenn der Graph beispielsweise als Adjazenzliste codiert ist. □

Vergleicht man die Laufzeit $O(2^k(n+m))$ von VC-Backtracking mit der Laufzeit des einfachen Brute-Force-Algorithmus, die mit n^k wächst, so stellt man fest, dass zwar beide Laufzeiten exponentiell mit k wachsen, in der Laufzeit von VC-Backtracking das exponentielle Wachstum aber von der Größe des Graphen entkoppelt ist (2^k statt n^k). Dies führt dazu, dass VC-Backtracking noch für deutlich größere Werte von k praktikabel ist.

Wir können die Laufzeit noch weiter verbessern. Dazu beobachten wir zunächst, dass wir in Graphen, in denen jeder Knoten maximal Grad 2 besitzt, ein optimales Vertex Cover effizient finden können. Ein solcher Graph besteht nur aus paarweise disjunkten und nicht verbundenen Pfaden und Kreisen. Für jede dieser Zusammenhangskomponenten kann man getrennt ein minimales Vertex Cover berechnen. Dazu wählt man im Wesentlichen jeden zweiten Knoten auf dem Kreis bzw. Pfad aus. Ein Pfad der Länge n kann mit $\lfloor n/2 \rfloor$ Knoten überdeckt werden, ein Kreis der Länge n mit $\lceil n/2 \rceil$ Knoten.

Solange es noch einen Knoten mit Grad mindestens 3 in dem Graphen G gibt, wählen wir einen solchen Knoten x aus. Um alle zu x inzidenten Kanten abzudecken, müssen in jedem Vertex Cover des Graphen entweder der Knoten x selbst oder alle seine Nachbarn enthalten sein. Im ersten dieser beiden Fälle muss der von $V \setminus \{x\}$ induzierte Teilgraph ein Vertex Cover der Größe $k-1$ besitzen (analog zu der Argumentation bei VC-Backtracking). Im zweiten Fall sei $N = \{y \in V \mid (x,y) \in E\}$ die Menge der Nachbarn von x. Der von $V \setminus N$ induzierte Teilgraph muss dann ein Vertex Cover der Größe $k - |N|$ besitzen, das gemeinsam mit den Knoten aus N ein Vertex Cover der Größe k für den gesamten Graphen G bildet. Diese Beobachtung machen wir uns in dem folgenden rekursiven Algorithmus zunutze.

VC-Backtracking-Knoten($\mathbf{G = (V, E)}, \mathbf{k}$)

1. **if** $(k < 0)$ **return** false;
2. **if** $(|E| = 0)$ **return** true;
3. **if** (G besitzt nur Knoten mit Grad ≤ 2)
4. Berechne in polynomieller Zeit die Größe opt eines minimalen Vertex Covers.
5. **if** $(k \geq \text{opt})$ **return** true; **else return** false;

6. Sei $x \in V$ ein beliebiger Knoten mit Grad ≥ 3.
7. Sei $N = \{y \in V \mid (x, y) \in E\}$ die Nachbarschaft von x.
8. Sei G_x der von $V \setminus \{x\}$ induzierte Teilgraph.
9. Sei G_N der von $V \setminus N$ induzierte Teilgraph.
10. **if** (VC-Backtracking-Knoten$(G_x, k-1)$) **return** true;
11. **if** (VC-Backtracking-Knoten$(G_N, k-|N|)$) **return** true;
12. **return** false;

Theorem 7.5. *Der Algorithmus VC-Backtracking-Knoten entscheidet in Zeit $O(1{,}5^k(n+m))$, ob der Graph G ein Vertex Cover der Größe k besitzt.*

Beweis. Die Korrektheit folgt aus unseren Vorüberlegungen. Der Graph G besitzt genau dann ein Vertex Cover der Größe k, wenn entweder G_x ein Vertex Cover der Größe $k-1$ oder G_N ein Vertex Cover der Größe $k-|N|$ besitzt. Auch die Randfälle werden korrekt überprüft. Anders als bei VC-Backtracking kann es vorkommen, dass ein Aufruf mit einem $k < 0$ erfolgt. In diesem Fall gibt es kein Vertex Cover der Größe k. Dieser Fall wird in Zeile 1 abgefangen. Ist $k \geq 0$ und E leer, so existiert ein Vertex Cover der Größe k (Zeile 2).

Die Laufzeit des Algorithmus können wir wie folgt beschränken. Sei G ein Graph mit n Knoten und m Kanten und sei $N(k)$ die Anzahl an Knoten im Rekursionsbaum von VC-Backtracking-Knoten bei Parameter k, also die Anzahl an Aufrufen der Methode VC-Backtracking-Knoten. Für $k \leq 2$ ist $N(k)$ durch eine geeignete Konstante c nach oben beschränkt. Für $k \geq 3$ gilt $N(k) \leq N(k-1) + N(k-3) + 1$, da ein Aufruf mit Parameter k höchstens zwei rekursive Aufrufe mit den Parametern $k-1$ und $k-|N| \leq k-3$ generiert. Für eine geeignete Konstante a, die nur von c abhängt, gilt $N(k) \leq 1{,}5^{k+a} - 1$ für jedes $k \in \mathbb{N}_0$. Dies können wir induktiv beweisen. Der Induktionsanfang für $k \leq 2$ folgt durch eine hinreichend große Wahl der Konstante a. Für $k \geq 3$ folgt mit der Induktionsannahme

$$\begin{aligned} N(k) &\leq N(k-1) + N(k-3) + 1 \\ &\leq (1{,}5^{k-1+a} - 1) + (1{,}5^{k-3+a} - 1) + 1 \\ &= 1{,}5^{k+a}\left(\frac{1}{1{,}5} + \frac{1}{1{,}5^3}\right) - 1 \\ &\leq 0{,}97 \cdot 1{,}5^{k+a} - 1 \\ &\leq 1{,}5^{k+a} - 1. \end{aligned}$$

Das Theorem folgt nun aus der Beobachtung, dass jeder Aufruf von VC-Backtracking-Knoten (ohne die rekursiven Aufrufe) eine Laufzeit von $O(n+m)$ benötigt, da insbesondere die Graphen G_x und G_N in dieser Zeit erstellt werden können, wenn der Graph als Adjazenzliste codiert ist. □

Wir haben in diesem Abschnitt bislang ausschließlich über die Entscheidungsvariante von Vertex Cover gesprochen. Die Optimierungsvariante, in der k nicht Teil der Eingabe ist, kann auf die Entscheidungsvariante zurückgeführt werden. Zunächst kann die

Wertvariante gelöst werden, indem mit $k = 0$ beginnend getestet wird, ob der gegebene Graph ein Vertex Cover der Größe k besitzt. Ist dies nicht der Fall, so wird k um eins erhöht und der Test erneut durchgeführt. Die Laufzeit dieses Verfahrens ist nur um einen Faktor k^* größer als die Laufzeit der Entscheidungsvariante, wenn k^* die Größe eines optimalen Vertex Covers bezeichnet. Ähnlich kann dann die Optimierungsvariante auf die Wertvariante reduziert werden, wodurch ein weiterer polynomieller Faktor zur Laufzeit hinzukommt. Insgesamt ergibt sich das folgende Korollar.

Korollar 7.6. *Es gibt eine Algorithmus für die Optimierungsvariante des Vertex-Cover-Problems, dessen Worst-Case-Laufzeit für Graphen mit n Knoten und m Kanten $O(1{,}5^{k^*} \cdot p(n, m))$ für ein geeignetes Polynom p beträgt, wobei k^* die Größe eines optimalen Vertex Cover bezeichnet.*

Natürlich ist der Algorithmus aus dem vorangegangenen Korollar für Instanzen, bei denen das optimale Vertex Cover groß ist, nicht praktikabel. Jedoch kann er selbst bei Instanzen mit zum Beispiel $k^* = 50$ durchaus noch sinnvoll eingesetzt werden (abhängig der Größe des Graphen und der zur Verfügung stehenden Rechenzeit). Der Algorithmus ist damit einem einfachen Brute-Force-Ansatz deutlich überlegen.

Wir wollen nur kurz erwähnen, dass die Algorithmen, die wir in diesem Abschnitt kennengelernt haben, in den Bereich der *parametrisierten Algorithmen* fallen. Das sind Algorithmen für NP-schwere Probleme, bei denen man zusätzlich zu der eigentlichen Eingabe noch einen Parameter k definiert. Dieser Parameter kann wie im Fall von Korollar 7.6 der Wert einer optimalen Lösung sein oder auch ein struktureller Parameter der Eingabe wie zum Beispiel der Maximalgrad des Eingabegraphen oder Ähnliches. Das Ziel ist es dann, einen Algorithmus zu entwerfen, dessen Laufzeit sich für eine Funktion f und ein Polynom p durch $O(f(k) \cdot p(n))$ beschränken lässt, wobei n die Eingabelänge bezeichnet. Die Funktion f wächst in der Regel exponentiell in dem Parameter k (in Korollar 7.6 gilt beispielsweise $f(k) = 1{,}5^k$). Im Worst Case über alle Eingaben führen diese Algorithmen natürlich auch zu einer exponentiellen Laufzeit, da der Parameter k groß werden kann. Da das exponentielle Wachstum von der Länge der Eingabe entkoppelt ist, sind solche Algorithmen aber oft auch für große Eingaben praktikabel, solange der Parameter k nicht allzu groß wird.

Literaturverzeichnis

[1] M. R. Garey und D. S. Johnson: **Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)**. W. H. Freeman, First Edition Auflage, 1979.

[2] John E. Hopcroft, Rajeev Motwani und Jeffrey D. Ullman: **Einführung in Automatentheorie, Formale Sprachen und Berechenbarkeit**. Pearson Studium, 3. Auflage, 2007.

[3] Christos H. Papadimitriou: **Computational Complexity**. Addison Wesley, 1993.

[4] Tim Roughgarden: **Algorithms Illuminated (Part 4): Algorithms for NP-Hard Problems**. Soundlikeyourself Publishing, LLC, 2020.

[5] Alan M. Turing: **On Computable Numbers, with an Application to the Entscheidungsproblem**. Proceedings of the London Mathematical Society, 42:230–265, 1937.

[6] Berthold Vöcking: **Berechenbarkeit und Komplexität**, Vorlesungsskript, RWTH Aachen, Wintersemester 2013/14. `http://algo.rwth-aachen.de/Lehre/WS1314/VBuK/BuK.pdf`.

[7] Ingo Wegener: **Theoretische Informatik – eine algorithmenorientierte Einführung**. Teubner, 3. Auflage, 1993.