Einführung in die Lineare und Kombinatorische Optimierung

(Algorithmische Diskrete Mathematik I, kurz ADM I)

Skriptum zur Vorlesung im WS 2012/2013

Prof. Dr. Martin Grötschel
Institut für Mathematik
Technische Universität Berlin

Version vom 4. Dezember 2012

Vorwort

Bei dem vorliegenden Skript handelt es sich um die Ausarbeitung der vierstündigen Vorlesung „Einführung in die Lineare und Kombinatorische Optimierung“ (mit zugehörigen Übungen und Tutorien), die die grundlegende Vorlesung des dreisemestrigen Zyklus „Algorithmische Diskrete Mathematik“ bildet. Diese Vorlesung wurde von mir im Wintersemester 20012/13 zusammen mit Benjamin Hiller an der TU Berlin gehalten, der auch an der Ausarbeitung des vorliegenden Vorlesungsskripts beteiligt war.

Das erste Ziel dieser Vorlesung ist, das Verständnis für Fragestellung der mathematischen Optimierung und deren Anwendungen zu wecken und das Vorgehen bei der mathematischen Modellierung von Optimierungsproblemen aus der Praxis kennenzulernen. Das zweite und wichtigere Ziel ist die Einführung in die Methoden zur Lösung derartiger Probleme. Die erste Vorlesung des Zyklus vermittelt Grundlagen der Theorie der Graphen und Netzwerke sowie der linearen, kombinatorischen und ganzzahligen Optimierung. Hierbei wird auf algorithmische Aspekte besonderer Wert gelegt.

Grundkenntnisse der linearen Algebra werden vorausgesetzt, die aus der diskreten Mathematik (vornehmlich Graphentheorie) benötigten Begriffe und Grundlagen werden in der Vorlesung vorgestellt.

In der Vorlesung werden insbesondere kombinatorische Optimierungsprobleme behandelt, die sich graphentheoretisch formulieren lassen, wobei vornehmlich Probleme untersucht werden, die mit Hilfe polynomialer Algorithmen gelöst werden können. Hierzu gehört natürlich eine Einführung in die Komplexitätstheorie (die Klassen P und NP, NP-Vollständigkeit). Es werden gleichfalls einige Heuristiken und Approximationsverfahren für „schwere“ Probleme vorgestellt sowie Methoden zu deren Analyse. Mit der Darstellung des Simplexalgorithmus und einiger seiner theoretischen Konsequenzen (Dualitätssatz, Sätze vom komplementären Schlupf) beginnt die Einführung in die lineare Optimierung, wobei auch bereits einige Aspekte der ganzzahligen Optimierung behandelt werden.

Es gibt kein einzelnes Buch, das den gesamten, in dieser Vorlesung abgehandelten Themenkreis abdeckt. Daher sind in die einzelnen Kapitel Literaturhinweise eingearbeitet worden. Hinweise auf aktuelle Lehrbücher, die als Begleittexte zur Vorlesung geeignet sind finden sich auf der zur Vorlesung gehörigen Webseite:

`http://www.zib.de/groetschel/teaching/WS1213/Lecture-WS1213-deutsch.html`

Die vorliegende Ausarbeitung ist ein Vorlesungsskript und kein Buch. Obwohl mit der gebotenen Sorgfalt geschrieben, war nicht genügend Zeit für das bei Lehrbüchern notwendige intensive Korrekturlesen und das Einarbeiten umfassender Literaturhinweise. Die daher vermutlich vorhandenen Fehler bitte ich zu entschuldigen (und mir wenn möglich mitzuteilen). Das Thema wird nicht erschöpfend behandelt. Das Manuskript enthält nur die wesentlichen Teile der Vorlesung. Insbesondere sind die Schilderungen komplexer Anwendungsfälle, der Schwierigkeiten bei der Modellierung praktischer Probleme, der Probleme bei der praktischen Umsetzung und die Darstellung der Erfolge, die in den letzten Jahren beim Einsatz der hier vorgestellten Methodik in der Industrie erzielt wurden, nicht in das Skript aufgenommen worden.

Martin Grötschel

Inhaltsverzeichnis

1 Einführung **1**
1.1 Einführendes Beispiel . . . 1
1.2 Optimierungsprobleme . . . 6

2 Grundlagen und Notation **11**
2.1 Graphen und Digraphen: Wichtige Definitionen und Bezeichnungen . . . 11
2.1.1 Grundbegriffe der Graphentheorie . . . 11
2.1.2 Graphen . . . 11
2.1.3 Digraphen . . . 15
2.1.4 Ketten, Wege, Kreise, Bäume . . . 16
2.2 Lineare Algebra . . . 20
2.2.1 Grundmengen . . . 20
2.2.2 Vektoren und Matrizen . . . 21
2.2.3 Kombinationen von Vektoren, Hüllen, Unabhängigkeit . . . 25
2.3 Polyeder und lineare Programme . . . 26

3 Diskrete Optimierungsprobleme **37**
3.1 Kombinatorische Optimierungsprobleme . . . 37
3.2 Klassische Fragestellungen der Graphentheorie . . . 38
3.3 Graphentheoretische Optimierungsprobleme: Beispiele . . . 42

4 Komplexitätstheorie und Speicherung von Graphen **59**
4.1 Probleme, Komplexit"atsmaße, Laufzeiten . . . 59
4.2 Die Klassen $\mathcal{P}$ und $\mathcal{NP}$, $\mathcal{NP}$-Vollst"andigkeit . . . 62
4.3 Datenstrukturen zur Speicherung von Graphen . . . 69
4.3.1 Kanten- und Bogenlisten . . . 69
4.3.2 Adjazenzmatrizen . . . 70
4.3.3 Adjazenzlisten . . . 71

5 Bäume und Wege **77**
5.1 Graphentheoretische Charakterisierungen . . . 77
5.2 Optimale Bäume und Wälder . . . 81
5.3 Kürzeste Wege . . . 90
5.3.1 Ein Startknoten, nichtnegative Gewichte . . . 91
5.3.2 Ein Startknoten, beliebige Gewichte . . . 95
5.3.3 Kürzeste Wege zwischen allen Knotenpaaren . . . 100
5.3.4 Min-Max-Sätze und weitere Bemerkungen . . . 103

5.4 LP/IP-Aspekte von Bäumen und Wegen 106
5.5 Exkurs: Greedy-Heuristiken für das Rucksackproblem und deren Analyse . 109

6 Maximale Flüsse in Netzwerken 115
6.1 Das Max-Flow-Min-Cut-Theorem 116
6.2 Der Ford-Fulkerson-Algorithmus 119
6.3 Einige Anwendungen 125

7 Flüsse mit minimalen Kosten 129
7.1 Flüsse mit minimalen Kosten 129
7.2 Transshipment-, Transport- u. Zuordnungsprobleme 137
7.3 Der Netzwerk-Simplex-Algorithmus 139

1 Einführung

1.1 Einführendes Beispiel

Ein Unternehmen produziert am Standort A ein Gut, das es mit der Bahn zu den Städten B, C und D transportieren möchte. Genauer, es werden wöchentlich 6 Waggons benötigt, von denen 2 nach B, einer nach C sowie 3 nach D befördert werden müssen. Auf Anfrage des Unternehmens nennt die Bahn die maximalen wöchentlichen Beförderungskapazitäten (in Waggons) sowie die Kosten pro Waggon, siehe Tabelle 1.1.

Um sich die Situation zu veranschaulichen, macht der verantwortliche Planer eine Zeichnung der im Netz möglichen Verbindungen, ihrer Kapazitäten, sowie ihrer Kosten, siehe Abbildung 1.1. Diese Abbildung repräsentiert einen *gerichteten Graphen* (auch *Digraph* genannt) $D = (V, A)$, der aus *Knoten* V und *Bögen* (auch *gerichtete Kanten* genannt) $A \subseteq V \times V$ besteht, wobei die Bögen die Knoten miteinander verbinden. Die Knoten entsprechen dabei den Städten A, B, C und D, die Bögen den möglichen Verbindungen zwischen diesen Städten. Zusätzlich zu dieser Information enthält der Graph in Abbildung 1.1 weitere Daten, die abstrakt als *Bogengewichte* bezeichnet werden und jedem Bogen gewisse von der jeweiligen Fragestellung abhängige Werte zuordnen. Konkret haben wir die Bogengewichte $l\colon A \to \mathbb{R}_+$, $u\colon A \to \mathbb{R}_+$ und $c\colon A \to \mathbb{R}_+$, die jeweils die Mindestzahl der Waggons pro Woche (hier immer 0), maximale Anzahl der Waggons pro Woche sowie Kosten pro Waggon auf jeder Verbindung angeben. (Die „Benamung“der Variablen und Daten erfolgt entsprechend dem üblichen Vorgehen in der englischsprachigen Literatur; „V“ steht für „vertices“, „A“ für „arcs“, „l“ und „u“ für „lower bound“ und „upper bound“, „c“ für „cost“ usw.)

Der Planer möchte nun die kostengünstigste Möglichkeit bestimmen, die 6 Waggons zu ihren Zielorten zu transportieren. Dies ist ein typisches Problem der *kombinatorischen Optimierung*, welche sich (ganz allgemein formuliert) damit beschäftigt, aus einer endlichen Menge, deren Elemente „Lösungen“ genannt werden und die bewertet sind, eine

Verbindung	**maximale Anzahl Waggons pro Woche**	**Kosten pro Waggon**
A nach B	4	5
A nach C	3	1
B nach C	2	2
B nach D	3	2
C nach D	4	1

Tabelle 1.1: Kapazitäten und Kosten der Verbindungen im Bahnnetz.

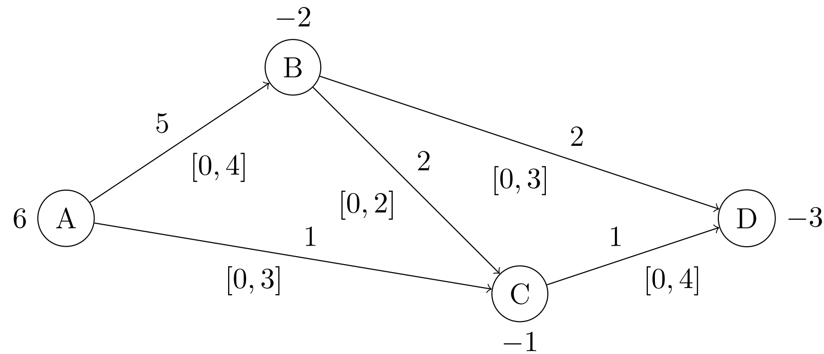


Abbildung 1.1: Beispielproblem als gerichteter Graph. Die Zahl an einem Knoten gibt an, wieviele Waggons dort bereitgestellt werden (positiv) bzw. wieviele Waggons angeliefert werden (negativ). Zahlen oberhalb eines Bogens geben die Transportkosten pro Waggon, Intervalle unterhalb eines Bogens die Kapazitäten der Verbindung an.

beste Lösung auszusuchen. Die in der kombinatorischen Optimierung auftretenden Probleme sind in der Regel „strukturiert“. In unserem Fall ist eine Grundstruktur durch einen Digraphen und die Funktionen l, u und c gegeben. Die Menge der Lösungen, über die optimiert werden soll, besteht aus der Menge aller möglichen Transporte von A zu den Zielen B, C und D. Kombinatorische Optimierungsprobleme können durch Enumeration aller Lösungen gelöst werden. Dies ist vielleicht in diesem Beispiel, aber im Allgemeinen keine gute Idee. In der Vorlesung geht es u. a. darum, mathematische Methoden zu entwickeln, um deartige Probleme (einigermaßen) effizient zu lösen.

In unserem Beispiel überlegt sich nun der Planer, dass er lediglich entscheiden muss, wieviele Waggons auf jeder der 5 Verbindungen befördert werden sollen. Dazu führt er die *Variablen* $f_{AB}, f_{AC}, f_{BC}, f_{BD}$ und f_{CD} ein. „f“ steht für „flow“ oder „Fluss“. Er überlegt sich, dass jede der Variablen mindestens den Wert 0 haben muss und nicht größer als die von der Bahn vorgegebene Kapazitätsgrenze sein darf. Außerdem notiert er sich, dass jede Variable nur ganzzahlige Werte annehmen kann. Insgesamt gelangt er so zu den Bedingungen

$$\begin{aligned}
f_{AB} &\in \mathbb{Z}, \quad 0 \leq f_{AB} \leq 4,\\
f_{AC} &\in \mathbb{Z}, \quad 0 \leq f_{AC} \leq 3,\\
f_{BC} &\in \mathbb{Z}, \quad 0 \leq f_{BC} \leq 2,\\
f_{BD} &\in \mathbb{Z}, \quad 0 \leq f_{BD} \leq 3,\\
f_{CD} &\in \mathbb{Z}, \quad 0 \leq f_{CD} \leq 4.
\end{aligned}$$

Mit diesen Variablen können die Gesamtkosten eines Transportes leicht als

$$5f_{AB} + f_{AC} + 2f_{BC} + 2f_{BD} + f_{CD}$$

dargestellt werden, und das Ziel ist, diese Gesamtkosten so gering wie möglich zu halten. Deswegen spricht man hier auch von der Zielfunktion (englisch: objective function). Nun

muss der Planer noch die Bedingungen festlegen, die den gewünschten Transport von A nach B, C und D beschreiben. Zunächst müssen 6 Waggons A verlassen, was sich als Gleichung

$$f_{AB} + f_{AC} = 6$$

schreiben lässt. Von den Waggons, die in B ankommen, sollen 2 dort verbleiben und der Rest weiter nach C und D fahren:

$$f_{AB} = 2 + f_{BC} + f_{BD}.$$

Analog können auch die Bedingungen in C und D formuliert werden.

Insgesamt hat der Planer nun folgende mathematische Formulierung vorliegen:

$$\min 5f_{AB} + f_{AC} + 2f_{BC} + 2f_{BD} + f_{CD} \tag{1.1a}$$
$$f_{AB} + f_{AC} = 6 \tag{1.1b}$$
$$-f_{AB} + f_{BC} + f_{BD} = -2 \tag{1.1c}$$
$$-f_{AC} - f_{BC} + f_{CD} = -1 \tag{1.1d}$$
$$-f_{BD} - f_{CD} = -3 \tag{1.1e}$$
$$0 \leq f_{AB} \leq 4 \tag{1.1f}$$
$$0 \leq f_{AC} \leq 3 \tag{1.1g}$$
$$0 \leq f_{BC} \leq 2 \tag{1.1h}$$
$$0 \leq f_{BD} \leq 3 \tag{1.1i}$$
$$0 \leq f_{CD} \leq 4 \tag{1.1j}$$
$$f_{AB}, \quad f_{AC}, \quad f_{BC}, \quad f_{BD}, \quad f_{CD} \in \mathbb{Z} \tag{1.1k}$$

Ein Optimierungsproblem dieser Art, in dem alle Variablen ganzzahlig alle Nebenbedingungen lineare Gleichungen oder Ungleichungen sind, und dessen Zielfunktion ebenfalls linear ist, heißt *Ganzzahliges Lineares Programm* oder als englische Abkürzung kurz *ILP* (oft auch nur *IP*, wenn aus dem Kontext klar ist, dass Nebenbedingungen und Zielfunktion linear sind). Sind alle Variablen kontinuierlich, so spricht man von einem *Linearen Programm* oder kurz *LP*. Zum Beispiel ist das Optimierungsproblem (1.1a)–(1.1j) ein LP.

Um nun eine optimale Transportvariante zu bestimmen, beschließt der Planer, die Ganzzahligkeitsbedingungen (1.1k) zunächst zu ignorieren, da dann nur ein lineares Gleichungssystem mit Variablenschranken übrig bleibt. Dem Planer fällt auf, dass die vier Gleichungen (1.1b) bis (1.1e) linear abhängig sind, weil sie sich zu 0 summieren. Man kann sich leicht überlegen, dass eine beliebige Gleichung gestrichen werden kann und die verbleibenden drei Gleichungen linear unabhängig sind. Wie wir aus der Linearen Algebra wissen, ist dann der Lösungsraum des Gleichungssystems ein 2-dimensionaler affiner Unterraum des $\mathbb{R}^5$. Mithilfe des Gauss-Algorithmus berechnet der Planer folgende Para-

metrisierung dieses Unterraums:

$$\begin{pmatrix} f_{AB} \\ f_{AC} \\ f_{BC} \\ f_{BD} \\ f_{CD} \end{pmatrix} = \begin{pmatrix} 6 \\ 1 \\ 3 \\ 0 \\ 0 \end{pmatrix} + s \begin{pmatrix} -1 \\ -1 \\ 0 \\ 1 \\ 0 \end{pmatrix} + t \begin{pmatrix} 0 \\ 1 \\ -1 \\ 0 \\ 1 \end{pmatrix}. \tag{1.2}$$

Aus der Parametrisierung (1.2) und den Schranken (1.1f) bis (1.1j) leitet der Planer das folgende LP her:

$$\min -6s + t \tag{1.3a}$$
$$-s + t \geq -1 \tag{1.3b}$$
$$-s + t \leq 1 \tag{1.3c}$$
$$2 \leq s \leq 3 \tag{1.3d}$$
$$0 \leq t \leq 3. \tag{1.3e}$$

Dieses LP ist „äquivalent" zu LP (1.1a)–(1.1j) in folgendem Sinn: Es gibt eine bijektive Abbildung zwischen den Lösungsräumen der beiden LPs, die mit den durch die Zielfunktionen gegebenen Ordnungen kompatibel ist. Mit anderen Worten: Jede Optimallösung von LP (1.3) entspricht genau einer Optimallösung von LP (1.1a)–(1.1j) und umgekehrt. Daher genügt es, das LP (1.3) zu lösen.

Da in LP (1.3) nur zwei Variablen vorkommen, kann man die zulässige Menge aller Paare (s, t), die alle Bedingungen erfüllen, graphisch darstellen (siehe Abbildung 1.2). Aus dieser Abbildung kann man direkt die optimale Lösung ablesen: Die optimale Lösung ist derjenige Punkt der grauen Fläche, der am weitesten in der dem Zielfunktionsvektor entgegengesetzten Richtung liegt (weil minimiert wird). Im Beispiel ist dies der Punkt $(3, 2)$, der der Lösung $f_{AB} = 3$, $f_{AC} = 3$, $f_{BC} = 0$, $f_{BD} = 1$, $f_{CD} = 2$ entspricht. Da alle Werte ganzzahlig sind, ist der Planer zufrieden und kommuniziert den enstprechenden Plan dem Bahnunternehmen.

Ist es immer so, dass bei ganzzahligen Problemdaten ein lineares Programm eine Optimallösung besitzt, die ganzzahlig ist? Natürlich nicht! Und deswegen geht unsere Story weiter: Nach einigen Tagen bekommt der Planer von der Bahn den Bescheid, dass die Waggons so leider nicht befördert werden können, weil die Kapazität des Rangierbahnhofs in C nicht ausreicht. Die Bahn beschreibt die Kapazitätsbeschränkung des Rangierbahnhofs genauer und der Planer bekommt die zusätzliche Bedingung

$$2f_{AC} + f_{BC} \leq 5,$$

die er via (1.2) in die äquivalente Bedingung

$$s + t \leq 4 \tag{1.4}$$

übersetzt. Das resultierende LP (1.3) mit der zusätzlichen Ungleichung (1.4) hat nun keine ganzzahlige Optimallösung mehr (siehe Abbildung 1.2); die Optimallösung ist $(2.5, 1.5)$ bzw. $f_{AB} = 3.5$, $f_{AC} = 2.5$, $f_{BC} = 0$, $f_{BD} = 1.5$, $f_{CD} = 1.5$ im Originalproblem.

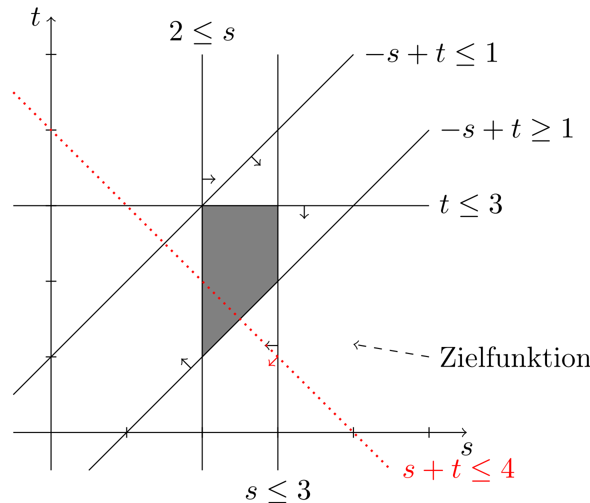


Abbildung 1.2: Graphische Darstellung des Optimierungsproblems (1.3). Die graue Menge ist die Menge der zulässigen Lösungen. Der gestrichelte Pfeil zeigt die Richtung, in der die Zielfunktion ansteigt. Die gepunktete Ungleichung entspricht der zusätzlichen Kapazitätsbedingung der Bahn.

Schluss, Variante 1: Der Planer ist nun mit seinem Schulwissen am Ende und fragt seinen Freund, einen Mathematiker, um Rat. Dieser empfiehlt ihm, sein Problem mit der Software SCIP (siehe `http://scip.zib.de`) zur Lösung ganzzahliger Optimierungsproblems zu lösen. Damit findet der Planer die neue Lösung $f_{AB} = 4$, $f_{AC} = 2$, $f_{BC} = 0$, $f_{BD} = 2$, $f_{CD} = 1$, mit der nun auch die Bahn einverstanden ist.

Schluss, Variante 2: Der Planer schaut sich die zulässige Menge in Abbildung 1.2 genau an und stellt fest, dass nur die ganzzahligen Punkte $(2, 1)$ und $(2, 2)$ zulässig sind. Er wählt den billigeren der beiden und gelangt zu der neuen Lösung $f_{AB} = 4$, $f_{AC} = 2$, $f_{BC} = 0$, $f_{BD} = 2$, $f_{CD} = 1$, mit der nun auch die Bahn einverstanden ist.

Die vorliegende Einführung ist natürlich „nur" ein didaktisches Beispiel. Wir haben gezeigt, wie man aus einer praktischen Fragestellung (die wir so vereinfacht haben, dass ihre Lösung graphisch gefunden werden kann), ein mathematisches Problem erstellt. Man nennt solch ein Vorgehen *mathematische Modellierung*. Es ist keineswegs so, dass jedem praktischen Problem ein eindeutiges mathematisches Modell entspricht. Es gibt viele unterschiedliche Modellierungsmethoden. Für welche man sich entscheidet, ist eine Frage des Geschmacks, der Vorbildung, oder der Algorithmen, die zur Lösung der mathematischen Modelle verfügbar sind. Ein einfaches Beispiel ist die Modellierung einer ja/nein-Entscheidung: Wir möchten ausdrücken, dass eine reelle Variable x nur die Werte 0 oder 1 annehmen darf ($x = 0$ entspricht „nein", $x = 1$ entspricht „ja"). Das können wir z. B. auf die folgenden Weisen erreichen:

- $x \in \{0, 1\}$,
- $0 \leq x \leq 1$, $x \in \mathbb{Z}$,

- $x = x^2$.

Welche Modellierung sinnvoll ist, kann man nicht grundsätzlich entscheiden. Die Wahl des Modells hängt vom gewählten Gesamtmodell und der geplanten algorithmischen Vorgehensweise ab.

1.2 Optimierungsprobleme

Wir wollen zunächst ganz informell, ohne auf technische Spitzfindigkeiten einzugehen, mathematische Optimierungsprobleme einführen. Sehr viele Probleme lassen sich wie folgt formulieren.

Gegeben seien eine Menge S und eine geordnete Menge $(T, \leq)$, d. h. zwischen je zwei Elementen $s, t \in T$ gilt genau eine der folgenden Beziehungen $s < t$, $s > t$ oder $s = t$. Ferner sei eine Abbildung $f : S \to T$ gegeben. Gesucht ist ein Element $x^* \in S$ mit der Eigenschaft $f(x^*) \geq f(x)$ für alle $x \in S$ (Maximierungsproblem) oder $f(x^*) \leq f(x)$ für alle $x \in S$ (Minimierungsproblem). Es ist üblich, hierfür eine der folgenden Schreibweisen zu benutzen:

$$\begin{array}{lll} \max_{x\in S} f(x) & \text{oder} & \max\{f(x) \mid x \in S\}, \\ \min_{x\in S} f(x) & \text{oder} & \min\{f(x) \mid x \in S\}. \end{array} \tag{1.5}$$

In der Praxis treten als geordnete Mengen $(T, \leq)$ meistens die reellen Zahlen $\mathbb{R}$, die rationalen Zahlen $\mathbb{Q}$ oder die ganzen Zahlen $\mathbb{Z}$ auf, alle mit der natürlichen Ordnung versehen (die wir deswegen auch gar nicht erst notieren). Die Aufgabe (1.5) ist viel zu allgemein, um darüber etwas Interessantes sagen zu können. Wenn S durch die Auflistung aller Elemente gegeben ist, ist das Problem entweder sinnlos oder trivial (man rechnet ganz einfach $f(x)$ für alle $x \in S$ aus). Das heißt, S muss irgendwie (explizit oder implizit) strukturiert sein, so dass vernünftige Aussagen über S möglich sind, ohne dass man alle Elemente in S einzeln kennt. Das gleiche gilt für die Funktion $f : S \to T$. Ist sie nur punktweise durch $x \mapsto f(x)$ gegeben, lohnt sich das Studium von (1.5) nicht. Erst wenn f durch hinreichend strukturierte "Formeln" bzw. "Eigenschaften" bestimmt ist, werden tieferliegende mathematische Einsichten möglich.

Die Optimierungsprobleme, die in der Praxis auftreten, haben fast alle irgendeine "vernünftige" Struktur. Das muss nicht unbedingt heißen, dass die Probleme dadurch auf einfache Weise lösbar sind, aber immerhin ist es meistens möglich, sie in das zur Zeit bekannte und untersuchte Universum der verschiedenen Typen von Optimierungsproblemen einzureihen und zu klassifizieren.

Im Laufe des Studiums werden Ihnen noch sehr unterschiedliche Optimierungsaufgaben begegnen. Viele werden von einem der folgenden Typen sein.

(1.6) Definition (Kontrollproblem). *Gegeben sei ein Steuerungsprozess (z. B. die Bewegungsgleichung eines Autos), etwa der Form*

$$\dot{x}(t) = f(t, x(t), u(t)),$$

wobei u eine Steuerung ist (Benzinzufuhr). Ferner seien eine Anfangsbedingung

$$x(0) = x_0,$$

(z. B.: das Auto steht) sowie eine Endbedingung

$$x(T) = x_1$$

(z. B. das Auto hat eine Geschwindigkeit von 50 km/h) gegeben. Gesucht ist eine Steuerung u für den Zeitraum $[0, T]$, so dass z. B.

$$\int_0^T |u|^2 \, \mathrm{d}t$$

minimal ist (etwa minimaler Benzinverbrauch). △

(1.7) Definition (Approximationsproblem). *Gegeben sei eine (numerisch schwierig auszuwertende) Funktion f, finde eine Polynom p vom Grad n, so dass*

$$\|f - p\| \quad \textit{oder} \quad \|f - p\|_\infty$$

minimal ist. △

(1.8) Definition (Nichtlineares Optimierungsproblem). *Es seien f, g_i $(i = 1, \ldots, m)$, h_j $(j = 1, \ldots, p)$ differenzierbare Funktionen von $\mathbb{R}^n \to \mathbb{R}$, dann heißt*

$$\begin{aligned} &\min f(x) \\ &g_i(x) \le 0 \qquad i = 1, \ldots, m \\ &h_i(x) = 0 \qquad j = 1, \ldots, p \\ &\quad x \in \mathbb{R}^n \end{aligned}$$

ein nichtlineares Optimierungsproblem. Ist eine der Funktionen nicht differenzierbar, so spricht man von einem nichtdifferenzierbaren Optimierungsproblem. Im Allgemeinen wird davon ausgegangen, daß alle betrachteten Funktionen zumindest stetig sind. △

(1.9) Definition (Konvexes Optimierungsproblem). *Eine Menge $S \subseteq \mathbb{R}^n$ heißt* **konvex**, *falls gilt: Sind $x, y \in S$ und ist $\lambda \in \mathbb{R}$, $0 \le \lambda \le 1$, dann gilt $\lambda x + (1 - \lambda) y \in S$. Eine Funktion $f : \mathbb{R}^n \to \mathbb{R}$ heißt* **konvex**, *falls für alle $\lambda \in \mathbb{R}$, $0 \le \lambda \le 1$ und alle $x, y \in \mathbb{R}^n$ gilt*

$$\lambda f(x) + (1 - \lambda) f(y) \ge f(\lambda x + (1 - \lambda) y).$$

Ist $S \subseteq \mathbb{R}^n$ konvex (z. B. kann S wie folgt gegeben sein $S = \{x \in \mathbb{R}^n \mid g_i(x) \le 0,\ i = 1, \ldots, m\}$ wobei die g_i konvexe Funktionen sind), und ist $f : \mathbb{R}^n \to \mathbb{R}$ eine konvexe Funktion, dann ist

$$\min_{x \in S} f(x)$$

ein konvexes Minimierungsproblem. △

(1.10) Definition (Lineares Optimierungsproblem (Lineares Programm)). *Gegeben seien $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{(m,n)}$, $b \in \mathbb{R}^m$, dann heißt*

$$\begin{aligned}&\max c^T x\\ &Ax \leq b\\ &x \in \mathbb{R}^n\end{aligned}$$

lineares Optimierungsproblem. △

(1.11) Definition (Lineares ganzzahliges Optimierungsproblem). *Gegeben seien $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{(m,n)}$, $b \in \mathbb{R}^m$, dann heißt*

$$\begin{aligned}&\max c^T x\\ &Ax \leq b\\ &x \in \mathbb{Z}^n\end{aligned}$$

lineares ganzzahliges (oder kurz: ganzzahliges) Optimierungsproblem. △

Selbst bei Optimierungsproblemen wie $(1.6), \ldots, (1.11)$, die nicht sonderlich allgemein erscheinen mögen, kann es sein, dass (bei spezieller Wahl der Zielfunktion f und der Nebenbedingungen) die Aufgabenstellung (finde ein $x^* \in S$, so dass $f(x^*)$ so groß (oder klein) wie möglich ist) keine vernünftige Antwort besitzt. Es mag sein, dass f über S unbeschränkt ist; f kann beschränkt sein über S, aber ein Maximum kann innerhalb S nicht erreicht werden, d. h., das „max“ müßte eigentlich durch „sup“ ersetzt werden. S kann leer sein, ohne dass dies a priori klar ist, etc. Der Leser möge sich Beispiele mit derartigen Eigenschaften überlegen! Bei der Formulierung von Problemen dieser Art muss man sich also Gedanken darüber machen, ob die betrachtete Fragestellung überhaupt eine sinnvolle Antwort erlaubt.

In unserer Vorlesung werden wir uns lediglich mit den Problemen (1.10) und (1.11) beschäftigen. Das lineare Optimierungsproblem (1.10) ist sicherlich das derzeit für die Praxis bedeutendste Problem, da sich außerordentlich viele und sehr unterschiedliche reale Probleme als lineare Programme formulieren lassen, bzw. durch die Lösung einer endlichen Anzahl von LPs gelöst werden können. Außerdem liegt eine sehr ausgefeilte Theorie vor. Mit den modernen Verfahren der linearen Optimierung können derartige Probleme mit Hunderttausenden (und manchmal sogar mehr) von Variablen und Ungleichungen fast „mühelos“ gelöst werden. Dagegen ist Problem (1.11) viel schwieriger. Die Einschränkung der Lösungsmenge auf die zulässigen ganzzahligen Lösungen führt direkt zu einem Sprung im Schwierigkeitsgrad des Problems. Verschiedene spezielle lineare ganzzahlige Programme können in beliebiger Größenordnung gelöst werden. Bei wenig strukturierten allgemeinen Problemen des Typs (1.11) versagen dagegen auch die besten Lösungsverfahren manchmal bereits bei weniger als 100 Variablen und Nebenbedingungen.

Über Kontrolltheorie (Probleme des Typs (1.6)), Approximationstheorie (Probleme des Typs (1.7)), Nichtlineare Optimierung (Probleme des Typs (1.8)) und (1.9) werden an der TU Berlin Spezialvorlesungen angeboten. Es ist anzumerken, dass sich sowohl die

Theorie als auch die Algorithmen zur Lösung von Problemen des Typs (1.6) bis (1.9) ganz erheblich von denen zur Lösung von Problemen des Typs (1.10) und (1.11) unterscheiden.

Ziel dieser Vorlesung ist zunächst, das Verständnis für Fragestellung der Optimierung und deren Anwendungen zu wecken. Die Studierenden sollen darüber hinaus natürlich in die Optimierungstheorie eingeführt werden und einige Werkzeuge theoretischer und algorithmischer Natur zur Lösung von linearen, kombinatorischen und ganzzahligen Optimierungsproblemen kennenlernen. Damit wird grundlegendes Rüstzeug zur Behandlung (Modellierung, numerischen Lösung) von brennenden Fragen der heutigen Zeit bereitgestellt. Die wirklich wichtigen Fragen benötigen zur ihrer Lösung jedoch erheblich mehr Mathematik sowie weitergehendes algorithmisches und informationstechnisches Know-how. Und ohne enge Zusammenarbeit mit Fachleuten aus Anwendungsdisziplinen wird man die meisten Fragen nicht anwendungsadäquat beantworten können. Einige Themen von derzeit großer Bedeutung, bei denen der Einsatz von mathematischer Optimierung (in Kombination mit vielfältigen Analysetechniken und praktischen Erfahrungen aus anderen Disziplinen) wichtig ist, hat der Präsident von acatech (Deutsche Akademie der Technikwissenschaften), Reinhard F. Hüttl, zu Beginn seiner Rede zur Eröffnung der Festveranstaltung 2012 am 16. Oktober 2012 im Konzerthaus am Gendarmenmarkt in Berlin beschrieben:

> *„Als wir im letzten Jahr an diesem Ort zur acatech-Festveranstaltung zusammenkamen, war die Energiewende bereits das, was man bürokratisch als „Beschlusslage“ bezeichnen würde. Gut ein Jahr nach diesem Beschluss bestimmen mehr Fragen als Antworten die Diskussion um unsere zukünftige Energieversorgung: Was wollen wir erreichen? - Den schnellstmöglichen Ausstieg aus der Kernenergie? Eine Verlangsamung des Klimawandels? Den raschen Ausbau erneuerbarer Energien? Möglichst hohe Energieeffizienz? Und wer sollte für welches Ziel und welche Maßnahme die Umsetzung verantworten? Und vor allem: Welche Ziele haben Priorität, und, um die dominierende Frage der letzten Tage und Wochen aufzugreifen, zu welchem Preis – im wörtlichen wie im übertragenen Sinne – wollen und können wir diese Ziele erreichen? Die Debatte der vergangenen Zeit hat gezeigt, dass es auf diese Fragen viele Antworten gibt. – In etwa so viele, wie es Interessensgruppen, Verbände, Unternehmen, Parteien und Parlamente gibt. Vielleicht lässt genau deshalb die Umsetzung der Energiewende auf sich warten. Auch die Wissenschaft hat keine endgültigen Antworten auf diese zentrale gesellschaftliche Frage. Aber: Wissenschaft kann helfen, Daten und Fakten zu sichten, ans Licht zu befördern und zu gewichten. Wissenschaft kann Handlungsoptionen darstellen und auch bewerten. Dies tun wir in unseren aktuellen Publikationen zu Fragen der Energie, der Mobilität, aber auch zu vielen anderen Themen. Das Entscheidungsprimat aber – und dies ist mir gerade in diesem Kontext sehr wichtig – lag und liegt bei der Politik.“*

Die mathematische Optimierung ist eine der wissenschaftlichen Disziplinen, die bei der Behandlung der skizzierten Problemfelder und Fragen eingesetzt werden muss. Aber sie

ist hier nur eine Hilfswissenschaft, denn die Formulierung von Gewichtungen und Zielen und die Bewertung von Ergebnissen unterliegen komplexen gesellschaftlichen Debatten. Wir sollten jedoch die Bedeutung der Mathematik hierbei nicht unterschätzen, sie hilft komplexe politische Entscheidungen (ein wenig) rationaler zu machen.

2 Grundlagen und Notation

2.1 Graphen und Digraphen: Wichtige Definitionen und Bezeichnungen

Bei der nachfolgenden Zusammenstellung von Begriffen und Bezeichnungen aus der Graphentheorie handelt es sich nicht um eine didaktische Einführung in das Gebiet der diskreten Mathematik. Dieses Kapitel ist lediglich als Nachschlagewerk gedacht, in dem die wichtigsten Begriffe und Bezeichnungen zusammengefasst und definiert sind.

2.1.1 Grundbegriffe der Graphentheorie

Die Terminologie und Notation in der Graphentheorie ist leider sehr uneinheitlich. Wir wollen daher hier einen kleinen Katalog wichtiger graphentheoretischer Begriffe und Bezeichnungen zusammenstellen und zwar in der Form, wie sie (in der Regel) in meinen Vorlesungen benutzt werden. Definitionen werden durch Kursivdruck hervorgehoben. Nach einer Definition folgen gelegentlich (in Klammern) weitere Bezeichnungen, um auf alternative Namensgebungen in der Literatur hinzuweisen.

Es gibt sehr viele Bücher über Graphentheorie. Wenn man zum Beispiel in der Datenbank MATH des Zentralblattes für Mathematik nach Büchern sucht, die den Begriff „graph theory“ im Titel enthalten, erhält man beinahe 400 Verweise. Bei über 50 Büchern taucht das Wort „Graphentheorie“ im Titel auf. Ich kenne natürlich nicht alle dieser Bücher. Zur Einführung in die mathematische Theorie empfehle ich u. a. Aigner (1984), Bollobás (1998), Diestel (2006)[1], Bondy and Murty (2008) und West (2005). Stärker algorithmisch orientiert und anwendungsbezogen sind z. B. Ebert (1981), Golombic (New York), Jungnickel (1994), Walther and Nägler (1987) sowie Krumke and Noltemeier (2005).

Übersichtsartikel zu verschiedenen Themen der Graphentheorie sind in den Handbüchern Graham et al. (1995) und Gross and Yellen (2004) zu finden.

2.1.2 Graphen

Ein *Graph* G ist ein Tripel (V, E, Ψ) bestehend aus einer nicht-leeren Menge V, einer Menge E und einer *Inzidenzfunktion* $\Psi : E \to V^{(2)}$. Hierbei bezeichnet $V^{(2)}$ die Menge der ungeordneten Paare von (nicht notwendigerweise verschiedenen) Elementen von V. Ein Element aus V heißt *Knoten* (oder *Ecke* oder *Punkt* oder *Knotenpunkt*; englisch: *vertex* oder *node* oder *point*), ein Element aus E heißt *Kante* (englisch: *edge* oder *line*).

[1] http://www.math.uni-hamburg.de/home/diestel/books/graphentheorie/GraphentheorieIII.pdf

Zu jeder Kante $e \in E$ gibt es also Knoten $u, v \in V$ mit $\Psi(e) = uv = vu$. (In der Literatur werden auch die Symbole $[u, v]$ oder $\{u, v\}$ zur Bezeichnung des ungeordneten Paares uv benutzt. Wir lassen zur Bezeichnungsvereinfachung die Klammern weg, es sei denn, dies führt zu unklarer Notation. Zum Beispiel bezeichnen wir die Kante zwischen Knoten 1 und Knoten 23 nicht mit 123, wir schreiben dann $\{1, 23\}$.)

Die Anzahl der Knoten eines Graphen heißt *Ordnung* des Graphen. Ein Graph heißt *endlich*, wenn V und E endliche Mengen sind, andernfalls heißt G *unendlich*. Wir werden uns nur mit endlichen Graphen beschäftigen und daher ab jetzt statt "endlicher Graph" einfach "Graph" schreiben. Wie werden versuchen, die natürliche Zahl n für die Knotenzahl und die natürliche Zahl m für die Kantenzahl eines Graphen zu reservieren. (Das gelingt wegen der geringen Anzahl der Buchstaben unseres Alphabets nicht immer.)

Gilt $\Psi(e) = uv$ für eine Kante $e \in E$, dann heißen die Knoten $u, v \in V$ *Endknoten* von e, und wir sagen, dass u und v mit e *inzidieren* oder *auf* e *liegen*, dass e die Knoten u und v *verbindet*, und dass u und v *Nachbarn* bzw. *adjazent* sind. Wir sagen auch, dass zwei Kanten *inzident* sind, wenn sie einen gemeinsamen Endknoten haben. Eine Kante e mit $\Psi(e) = uu$ heißt *Schlinge*; Kanten e, f mit $\Psi(e) = uv = \Psi(f)$ heißen *parallel*, man sagt in diesem Falle auch, dass die Knoten u und v durch eine *Mehrfachkante* verbunden sind. Graphen, die weder Mehrfachkanten noch Schlingen enthalten, heißen *einfach*. Der einfache Graph, der zu jedem in G adjazenten Knotenpaar u, v mit $u \neq v$ genau eine u und v verbindende Kante enthält, heißt der G *unterliegende einfache Graph*. Mit $\Gamma(v)$ bezeichnen wir die Menge der Nachbarn eines Knotens v. Falls v in einer Schlinge enthalten ist, ist v natürlich mit sich selbst benachbart. $\Gamma(W) := \bigcup_{v \in W} \Gamma(v)$ ist die Menge der Nachbarn von $W \subseteq V$. Ein Knoten ohne Nachbarn heißt *isoliert*.

Die Benutzung der Inzidenzfunktion Ψ führt zu einem relativ aufwendigen Formalimus. Wir wollen daher die Notation etwas vereinfachen. Dabei entstehen zwar im Falle von nicht-einfachen Graphen gelegentlich Mehrdeutigkeiten, die aber i. a. auf offensichtliche Weise interpretiert werden können. Statt $\Psi(e) = uv$ schreiben wir von nun an einfach $e = uv$ (oder äquivalent $e = vu$) und meinen damit die Kante e mit den Endknoten u und v. Das ist korrekt, solange es nur eine Kante zwischen u und v gibt. Gibt es mehrere Kanten mit den Endknoten u und v, und sprechen wir von der Kante uv, so soll das heißen, dass wir einfach eine der parallelen Kanten auswählen. Von jetzt an vergessen wir also die Inzidenzfunktion Ψ und benutzen die Abkürzung $G = (V, E)$, um einen Graphen zu bezeichnen. Manchmal schreiben wir auch, wenn erhöhte Präzision erforderlich ist, E_G oder $E(G)$ bzw. V_G oder $V(G)$ zur Bezeichnung der Kanten- bzw. Knotenmenge eines Graphen G.

Zwei Graphen $G = (V, E)$ und $H = (W, F)$ heißen *isomorph*, wenn es eine bijektive Abbildung $\varphi : V \to W$ gibt, so dass $uv \in E$ genau dann gilt, wenn $\varphi(u)\varphi(v) \in F$ gilt. Isomorphe Graphen sind also — bis auf die Benamung der Knoten und Kanten — identisch.

Eine Menge F von Kanten heißt *Schnitt*, wenn es eine Knotenmenge $W \subseteq V$ gibt, so dass $F = \delta(W) := \{uv \in E \mid u \in W, v \in V \setminus W\}$ gilt; manchmal wird $\delta(W)$ der *durch* W *induzierte Schnitt* genannt. Statt $\delta(\{v\})$ schreiben wir kurz $\delta(v)$. Ein Schnitt, der keinen anderen nicht-leeren Schnitt als echte Teilmenge enthält, heißt *Cokreis* (oder *minimaler Schnitt*). Wollen wir betonen, dass ein Schnitt $\delta(W)$ bezüglich zweier Knoten $s, t \in V$

die Eigenschaft $s \in W$ und $t \in V \setminus W$ hat, so sagen wir, $\delta(W)$ ist ein *s und t trennender Schnitt* oder kurz ein *$[s,t]$-Schnitt*.

Generell benutzen wir die eckigen Klammern [. , .], um anzudeuten, dass die Reihenfolge der Objekte in der Klammer ohne Bedeutung ist. Z. B. ist ein $[s,t]$-Schnitt natürlich auch ein $[t,s]$-Schnitt, da ja $\delta(W) = \delta(V \setminus W)$ gilt.

Wir haben oben Bezeichnungen wie $\Gamma(v)$ oder $\delta(W)$ eingeführt unter der stillschweigenden Voraussetzung, dass man weiß, in Bezug auf welchen Graphen diese Mengen definiert sind. Sollten mehrere Graphen involviert sein, so werden wir, wenn Zweideutigkeiten auftreten können, die Graphennamen als Indizes verwenden, also z. B. $\Gamma_G(v)$ oder $\delta_G(V)$ schreiben. Analog wird bei allen anderen Symbolen verfahren.

Der *Grad* (oder die *Valenz*) eines Knotens v (Bezeichnung: $\deg(v)$) ist die Anzahl der Kanten, mit denen er inzidiert, wobei Schlingen doppelt gezählt werden. Hat ein Graph keine Schlingen, so ist der Grad von v gleich $|\delta(v)|$. Ein Graph heißt *k-regulär*, wenn jeder Knoten den Grad k hat, oder kurz *regulär*, wenn der Grad k nicht hervorgehoben werden soll.

Sind W eine Knotenmenge und F eine Kantenmenge in $G = (V,E)$, dann bezeichnen wir mit $E(W)$ die Menge aller Kanten von G mit beiden Endknoten in W und mit $V(F)$ die Menge aller Knoten, die Endknoten mindestens einer Kante aus F sind.

Sind $G = (V,E)$ und $H = (W,F)$ zwei Graphen, so heißt der Graph $(V \cup W, E \cup F)$ die *Vereinigung* von G und H, und $(V \cap W, E \cap F)$ heißt der *Durchschnitt* von G und H. G und H heißen *disjunkt*, falls $V \cap W = \emptyset$, *kantendisjunkt*, falls $E \cap F = \emptyset$. Wir sprechen von einer *disjunkten* bzw. *kantendisjunkten Vereinigung* von zwei Graphen, wenn sie disjunkt bzw. kantendisjunkt sind.

Sind $G = (V,E)$ und $H = (W,F)$ Graphen, so dass $W \subseteq V$ und $F \subseteq E$ gilt, so heißt H *Untergraph* (oder *Teilgraph*) von G. Falls $W \subseteq V$, so bezeichnet $G - W$ den Graphen, den man durch *Entfernen* (oder *Subtrahieren*) aller Knoten in W und aller Kanten mit mindestens einem Endknoten in W gewinnt. $G[W] := G - (V \setminus W)$ heißt der *von W induzierte Untergraph* von G. Es gilt also $G[W] = (W, E(W))$. Für $F \subseteq E$ ist $G - F := (V, E \setminus F)$ der Graph, den man durch *Entfernen* (oder Subtrahieren) der Kantenmenge F enthält. Statt $G - \{f\}$ schreiben wir $G - f$, analog schreiben wir $G - w$ statt $G - \{w\}$ für $w \in V$. Ein Untergraph $H = (W,F)$ von $G = (V,E)$ heißt *aufspannend*, falls $V = W$ gilt.

Ist $G = (V,E)$ ein Graph und $W \subseteq V$ eine Knotenmenge, so bezeichnen wir mit $G \cdot W$ den Graphen, der durch *Kontraktion der Knotenmenge W* entsteht. Das heißt, die Knotenmenge von $G \cdot W$ besteht aus den Knoten $V \setminus W$ und einem neuen Knoten w, der die Knotenmenge W ersetzt. Die Kantenmenge von $G \cdot W$ enthält alle Kanten von G, die mit keinem Knoten aus W inzidieren, und alle Kanten, die genau einen Endknoten in W haben, aber dieser Endknoten wird durch w ersetzt (also können viele parallele Kanten entstehen). Keine der Kanten von G, die in $E(W)$ liegen, gehört zu $G \cdot W$. Falls $e = uv \in E$ und falls G keine zu e parallele Kante enthält, dann ist der Graph, der durch *Kontraktion der Kante e* entsteht (Bezeichnung $G \cdot e$), der Graph $G \cdot \{u,v\}$. Falls G zu e parallele Kanten enthält, so erhält man $G \cdot e$ aus $G \cdot \{u,v\}$ durch Addition von so vielen Schlingen, die den neuen Knoten w enthalten, wie es Kanten in G parallel zu e gibt. Der Graph $G \cdot F$, den man durch *Kontraktion einer Kantenmenge $F \subseteq E$* erhält, ist der

Graph, der durch sukzessive Kontraktion (in beliebiger Reihenfolge) der Kanten aus F gewonnen wird. Ist e eine Schlinge von G, so sind $G \cdot e$ und $G - e$ identisch.

Ein einfacher Graph heißt *vollständig*, wenn jedes Paar seiner Knoten durch eine Kante verbunden ist. Offenbar gibt es — bis auf Isomorphie — nur einen vollständigen Graphen mit n Knoten. Dieser wird mit K_n bezeichnet. Ein Graph G, dessen Knotenmenge V in zwei disjunkte nicht-leere Teilmengen V_1, V_2 mit $V_1 \cup V_2 = V$ zerlegt werden kann, so dass keine zwei Knoten in V_1 und keine zwei Knoten in V_2 benachbart sind, heißt *bipartit* (oder *paar*). Die Knotenmengen V_1, V_2 nennt man eine *Bipartition* (oder *2-Färbung*) von G. Falls G zu je zwei Knoten $u \in V_1$ und $v \in V_2$ genau eine Kante uv enthält, so nennt man G *vollständig bipartit*. Den — bis auf Isomorphie eindeutig bestimmten — vollständig bipartiten Graphen mit $|V_1| = m, |V_2| = n$ bezeichnen wir mit $K_{m,n}$.

Ist G ein Graph, dann ist das *Komplement* von G, bezeichnet mit $\overline{G}$, der einfache Graph, der dieselbe Knotenmenge wie G hat und bei dem zwei Knoten genau dann durch eine Kante verbunden sind, wenn sie in G nicht benachbart sind. Ist G einfach, so gilt $\overline{\overline{G}} = G$. Der *Kantengraph* (englisch: *line graph*) $L(G)$ eines Graphen G ist der einfache Graph, dessen Knotenmenge die Kantenmenge von G ist und bei dem zwei Knoten genau dann adjazent sind, wenn die zugehörigen Kanten in G einen gemeinsamen Endknoten haben.

Eine *Clique* in einem Graphen G ist eine Knotenmenge Q, so dass je zwei Knoten aus Q in G benachbart sind. Eine *stabile Menge* in einem Graphen G ist eine Knotenmenge S, so dass je zwei Knoten aus S in G nicht benachbart sind. Für stabile Mengen werden auch die Begriffe *unabhängige Knotenmenge* oder *Coclique* verwendet. Eine Knotenmenge K in G heißt *Knotenüberdeckung* (oder *Überdeckung von Kanten durch Knoten*), wenn jede Kante aus G mit mindestens einem Knoten in K inzidiert. Die größte Kardinalität (= Anzahl der Elemente) einer stabilen Menge (bzw. Clique) in einem Graphen bezeichnet man mit $\alpha(G)$ (bzw. $\omega(G)$); die kleinste Kardinalität einer Knotenüberdeckung mit $\tau(G)$.

Eine Kantenmenge M in G heißt *Matching* (oder *Paarung* oder *Korrespondenz* oder *unabhängige Kantenmenge*), wenn M keine Schlingen enthält und je zwei Kanten in M keinen gemeinsamen Endknoten besitzen. M heißt *perfekt*, wenn jeder Knoten von G Endknoten einer Kante des Matchings M ist. Ein perfektes Matching wird auch *1-Faktor* genannt. Eine Kantenmenge F in G heißt *k-Faktor* (oder *perfektes k-Matching*), wenn jeder Knoten von G in genau k Kanten aus F enthalten ist. Eine *Kantenüberdeckung* (oder *Überdeckung von Knoten durch Kanten*) ist eine Kantenmenge, so dass jeder Knoten aus G mit mindestens einer Kante dieser Menge inzidiert. Die größte Kardinalität eines Matchings in G bezeichnet man mit $\nu(G)$, die kleinste Kardinalität einer Kantenüberdeckung mit $\rho(G)$.

Eine Zerlegung der Knotenmenge eines Graphen in stabile Mengen, die so genannten *Farbklassen*, heißt *Knotenfärbung*; d. h. die Knoten werden so gefärbt, dass je zwei benachbarte Knoten eine unterschiedliche Farbe haben. Eine Zerlegung der Kantenmenge in Matchings heißt *Kantenfärbung*; die Kanten werden also so gefärbt, dass je zwei inzidente Kanten verschieden gefärbt sind. Eine Zerlegung der Knotenmenge von G in Cliquen heißt *Cliquenüberdeckung* von G. Die minimale Anzahl von stabilen Mengen (bzw. Cliquen) in einer Knotenfärbung (bzw. Cliquenüberdeckung) bezeichnet man mit

$\chi(G)$ (bzw. $\overline{\chi}(G)$), die minimale Anzahl von Matchings in einer Kantenfärbung mit $\gamma(G)$. Die Zahl $\gamma(G)$ heißt *chromatischer Index* (oder *Kantenfärbungszahl*), $\chi(G)$ *Färbungszahl* (oder *Knotenfärbungszahl* oder *chromatische Zahl*).

Ein Graph $G = (V, E)$ kann in die Ebene gezeichnet werden, indem man jeden Knoten durch einen Punkt repräsentiert und jede Kante durch eine Kurve (oder Linie oder Streckenstück), die die beiden Punkte verbindet, die die Endknoten der Kante repräsentieren. Ein Graph heißt *planar* (oder plättbar), falls er in die Ebene gezeichnet werden kann, so dass sich keine zwei Kanten (d. h. die sie repräsentierenden Kurven) schneiden — außer möglicherweise in ihren Endknoten. Eine solche Darstellung eines planaren Graphen G in der Ebene nennt man auch *Einbettung* von G in die Ebene.

2.1.3 Digraphen

Die Kanten eines Graphen haben keine Orientierung. In vielen Anwendungen spielen aber Richtungen eine Rolle. Zur Modellierung solcher Probleme führen wir gerichtete Graphen ein. Ein *Digraph* (oder *gerichteter Graph*) $D = (V, A)$ besteht aus einer (endlichen) nichtleeren *Knotenmenge* V und einer (endlichen) Menge A von *Bögen* (oder *gerichteten Kanten*; englisch: *arc*). Ein *Bogen* a ist ein geordnetes Paar von Knoten, also $a = (u, v)$, u ist der *Anfangs-* oder *Startknoten*, v der *End-* oder *Zielknoten* von a; u heißt *Vorgänger* von v, v *Nachfolger* von u, a *inzidiert mit* u und v. (Um exakt zu sein, müssten wir hier ebenfalls eine Inzidenzfunktion $\Psi = (t, h) : A \to V \times V$ einführen. Für einen Bogen $a \in A$ ist dann $t(a)$ der Anfangsknoten (englisch: tail) und $h(a)$ der Endknoten (englisch: head) von a. Aus den bereits oben genannten Gründen wollen wir jedoch die Inzidenzfunktion nur in Ausnahmefällen benutzen.) Wie bei Graphen gibt es auch hier *parallele Bögen* und *Schlingen*. Die Bögen (u, v) und (v, u) heißen *antiparallel*.

In manchen Anwendungsfällen treten auch "Graphen" auf, die sowohl gerichtete als auch ungerichtete Kanten enthalten. Wir nennen solche Objekte *gemischte Graphen* und bezeichnen einen gemischten Graphen mit $G = (V, E, A)$, wobei V die Knotenmenge, E die Kantenmenge und A die Bogenmenge von G bezeichnet.

Falls $D = (V, A)$ ein Digraph ist und $W \subseteq V, B \subseteq A$, dann bezeichnen wir mit $A(W)$ die Menge der Bögen, deren Anfangs- und Endknoten in W liegen, und mit $V(B)$ die Menge der Knoten, die als Anfangs- oder Endknoten mindestens eines Bogens in B auftreten. Unterdigraphen, induzierte Unterdigraphen, aufspannende Unterdigraphen, Vereinigung und Durchschnitt von Digraphen, das Entfernen von Bogen- oder Knotenmengen und die Kontraktion von Bogen- oder Knotenmengen sind genau wie bei Graphen definiert.

Ist $D = (V, A)$ ein Digraph, dann heißt der Graph $G = (V, E)$, der für jeden Bogen $(i, j) \in A$ eine Kante ij enthält, der *D unterliegende Graph*. Analog werden der *D unterliegende einfache Graph* und der *unterliegende einfache Digraph* definiert. Wir sagen, dass ein Digraph eine "ungerichtete" Eigenschaft hat, wenn der ihm unterliegende Graph diese Eigenschaft hat (z. B., D ist bipartit oder planar, wenn der D unterliegende Graph G bipartit oder planar ist). Geben wir jeder Kante ij eines Graphen G eine Orientierung, d. h., ersetzen wir ij durch einen der Bögen (i, j) oder (j, i), so nennen wir den so entstehenden Digraphen D *Orientierung* von G.

Ein einfacher Digraph heißt *vollständig*, wenn je zwei Knoten $u \neq v$ durch die beiden

Bögen $(u, v), (v, u)$ verbunden sind. Ein *Turnier* ist ein Digraph, der für je zwei Knoten $u \neq v$ genau einen der Bögen (u, v) oder (v, u) enthält. (Der einem Turnier unterliegende Graph ist also ein vollständiger Graph; jedes Turnier ist die Orientierung eines vollständigen Graphen.)

Für $W \subseteq V$ sei $\delta^+(W) := \{(i, j) \in A | \ i \in W, j \notin W\}, \delta^-(W) := \delta^+(V \setminus W)$ und $\delta(W) := \delta^+(W) \cup \delta^-(W)$. Die Bogenmenge $\delta^+(W)$ (bzw. $\delta^-(W)$) heißt *Schnitt*. Ist $s \in W$ und $t \notin W$, so heißt $\delta^+(W)$ auch (s, t)-Schnitt. (Achtung: in einem Digraphen ist ein (s, t)-Schnitt kein (t, s)-Schnitt!)

Statt $\delta^+(\{v\}), \delta^-(\{v\}), \delta(\{v\})$ schreiben wir $\delta^+(v), \delta^-(v), \delta(v)$. Der *Außengrad* (*Innengrad*) von v ist die Anzahl der Bögen mit Anfangsknoten (Endknoten) v. Die Summe von Außengrad und Innengrad ist der *Grad* von v. Ein Schnitt $\delta^+(W), \emptyset \neq W \neq V$, heißt *gerichteter Schnitt*, falls $\delta^-(W) = \emptyset$, d. h. falls $\delta(W) = \delta^+(W)$. Ist $r \in W$, so sagen wir auch, dass $\delta^+(W)$ ein *Schnitt mit Wurzel* r ist.

2.1.4 Ketten, Wege, Kreise, Bäume

Das größte Durcheinander in der graphentheoretischen Terminologie herrscht bei den Begriffen Kette, Weg, Kreis und bei den damit zusammenhängenden Namen. Wir haben uns für folgende Bezeichnungen entschieden.

In einem Graphen oder Digraphen heißt eine endliche Folge $W = (v_0, e_1, v_1, e_2, v_2, \cdots, e_k, v_k)$, $k \geq 0$, die mit einem Knoten beginnt und endet und in der Knoten und Kanten (Bögen) alternierend auftreten, so dass jede Kante (jeder Bogen) e_i mit den beiden Knoten v_{i-1} und v_i inzidiert, eine *Kette*. Der Knoten v_0 heißt *Anfangsknoten*, v_k *Endknoten* der Kette; die Knoten $v_1, \ldots, v_{k-1}$ heißen *innere Knoten*; W wird auch $[v_0, v_k]$-*Kette* genannt. Die Zahl k heißt *Länge* der Kette (= Anzahl der Kanten bzw. Bögen in W, wobei einige Kanten/Bögen mehrfach auftreten können und somit mehrfach gezählt werden). Abbildung 1 (b) zeigt eine Kette der Länge 13 im Graphen G aus Abbildung 1 (a). Aus einem solchen Bild kann man in der Regel nicht entnehmen, in welcher Reihenfolge die Kanten durchlaufen werden.

Falls (in einem Digraphen) alle Bögen e_i der Kette W der Form (v_{i-1}, v_i) (also gleichgerichtet) sind, so nennt man W *gerichtete Kette* bzw. (v_0, v_k)-*Kette*. Ist $W = (v_0, e_1, v_1, \ldots, e_k, v_k)$ eine Kette, und sind i, j Indizes mit $0 \leq i < j \leq k$, dann heißt die Kette $(v_i, e_{i+1}, v_{i+1}, \ldots, e_j, v_j)$ das $[v_i, v_j]$-*Segment* (bzw. (v_i, v_j)-*Segment*, wenn W gerichtet ist) von W. Jede (gerichtete) Kante, die zwei Knoten der Kette W miteinander verbindet, die aber nicht Element von W ist, heißt *Diagonale* (oder *Sehne*) von W.

Gibt es in einem Graphen keine parallelen Kanten, so ist eine Kette W bereits durch die Folge $(v_0, \ldots, v_k)$ ihrer Knoten eindeutig festgelegt. Desgleichen ist in einem Digraphen ohne parallele Bögen eine gerichtete Kette durch die Knotenfolge $(v_0, \ldots, v_k)$ bestimmt. Zur Bezeichnungsvereinfachung werden wir daher häufig von der Kette $(v_0, \ldots, v_k)$ in einem Graphen bzw. der gerichteten Kette $(v_0, \ldots, v_k)$ in einem Digraphen sprechen, obgleich bei parallelen Kanten (Bögen) die benutzten Kanten (Bögen) hiermit nicht eindeutig festgelegt sind. Diese geringfügige Ungenauigkeit sollte aber keine Schwierigkeiten bereiten. Gelegentlich interessiert man sich mehr für die Kanten (Bögen) einer Kette, insbesondere wenn diese ein Weg oder ein Kreis (siehe unten) ist. In solchen Fällen ist es

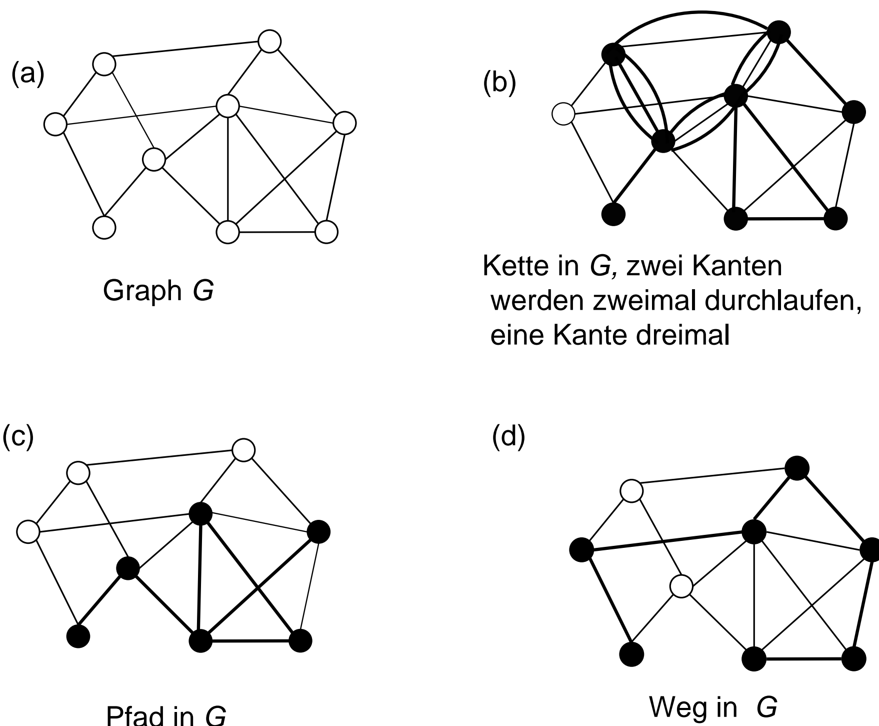


Abbildung 2.1: Kette, Pfad und Weg in einem Graph G.

zweckmäßiger, eine Kette als Kantenfolge $(e_1, e_2, \ldots, e_k)$ zu betrachten. Ist C die Menge der Kanten (Bögen) eines Kreises oder eines Weges, so spricht man dann einfach vom Kreis oder Weg C, während $V(C)$ die Menge der Knoten des Kreises oder Weges bezeichnet. Je nach behandeltem Themenkreis wird hier die am besten geeignete Notation benutzt.

Eine Kette, in der alle Knoten voneinander verschieden sind, heißt *Weg* (siehe Abbildung 1 (d)). Eine Kette, in der alle Kanten oder Bögen verschieden sind, heißt *Pfad*. Ein Beispiel ist in Abb. 1 (c) dargestellt. Ein Weg ist also ein Pfad, aber nicht jeder Pfad ist ein Weg. Ein Weg oder Pfad in einem Digraphen, der eine gerichtete Kette ist, heißt *gerichteter Weg* oder *gerichteter Pfad*. Wie bei Ketten sprechen wir von $[u, v]$-*Wegen*, (u, v)-*Wegen* etc.

Im Englischen heißt Kette *walk* oder *chain*. Im Deutschen benutzen z. B. Domschke (1982), Hässig (1979) und Berge and Ghouila-Houri (1969) ebenfalls das Wort Kette, dagegen schreiben Aigner (1984), Diestel (2006) und Wagner (1970) hierfür "Kantenzug", während König (1936), Halin (1989) und Sachs (1970) "Kantenfolge" benutzen; Ebert (1981) schließlich nennt unsere Ketten "ungerichtete Pfade". Dieses Wirrwarr setzt sich bezüglich der Begriffe Pfad und Weg auf ähnliche Weise fort.

Eine Kette heißt *geschlossen*, falls ihre Länge nicht Null ist und falls ihr Anfangsknoten mit ihrem Endknoten übereinstimmt. Ein geschlossener (gerichteter) Pfad, bei dem

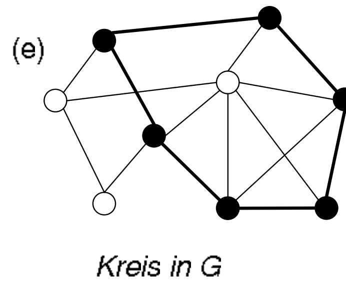


Abbildung 2.2: Ein Kreis in einem Graph G.

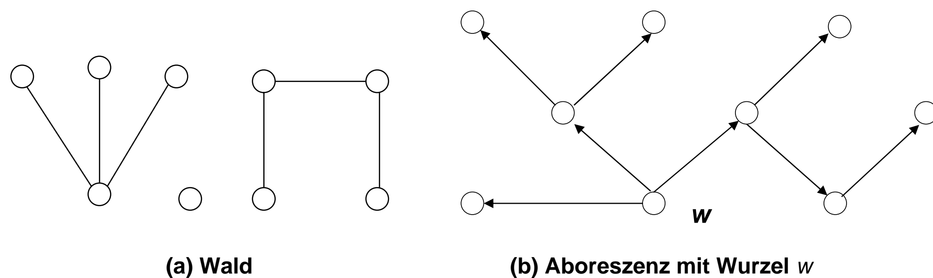


Abbildung 2.3: Ein Wald und eine Arboreszenz.

der Anfangsknoten und alle inneren Knoten voneinander verschieden sind, heißt *Kreis*, (*gerichteter Kreis*). Offensichtlich enthält jeder geschlossene Pfad einen Kreis, siehe Abb. 2.2.

Ein (gerichteter) Pfad, der jede Kante (jeden Bogen) eines Graphen (Digraphen) genau einmal enthält, heißt (gerichteter) *Eulerpfad*. Ein geschlossener Eulerpfad heißt *Eulertour*. Ein *Eulergraph* (*Eulerdigraph*) ist ein Graph (Digraph), der eine (gerichtete) Eulertour enthält.

Ein (gerichteter) Kreis (Weg) der Länge $|V|$ (bzw. $|V| - 1$) heißt (gerichteter) *Hamiltonkreis* (*Hamiltonweg*). Ein Graph (Digraph), der einen (gerichteten) Hamiltonkreis enthält, heißt *hamiltonsch*. Manchmal sagen wir statt Hamiltonkreis einfach *Tour*.

Ein *Wald* ist ein Graph, der keinen Kreis enthält, siehe Abb. 2.3(a). Ein zusammenhängender Wald heißt *Baum*. Ein Baum in einem Graphen heißt *aufspannend*, wenn er alle Knoten des Graphen enthält. Ein *Branching* B ist ein Digraph, der ein Wald ist, so dass jeder Knoten aus B Zielknoten von höchstens einem Bogen von B ist. Ein zusammenhängendes Branching heißt *Arboreszenz*, siehe Abb. 2.3(b). Eine *aufspannende Arboreszenz* ist eine Arboreszenz in einem Digraphen D, die alle Knoten von D enthält. Eine Arboreszenz enthält einen besonderen Knoten, genannt *Wurzel*, von dem aus jeder andere Knoten auf genau einem gerichteten Weg erreicht werden kann. Arboreszenzen werden auch *Wurzelbäume* genannt. Ein Digraph, der keinen gerichteten Kreis enthält, heißt *azyklisch*.

Ein Graph heißt *zusammenhängend*, falls es zu jedem Paar von Knoten s, t einen $[s, t]$-Weg in G gibt. Ein Digraph D heißt *stark zusammenhängend*, falls es zu je zwei Knoten s, t von D sowohl einen gerichteten (s, t)-Weg als auch einen gerichteten (t, s)-Weg in D gibt. Die *Komponenten* (*starken Komponenten*) eines Graphen (Digraphen) sind die bezüglich Kanteninklusion (Bogeninklusion) maximalen zusammenhängenden Untergraphen von G (maximalen stark zusammenhängenden Unterdigraphen von D). Eine Komponente heißt *ungerade Komponente*, falls ihre Knotenzahl ungerade ist, andernfalls heißt sie *gerade Komponente*.

Sei $G = (V, E)$ ein Graph. Eine Knotenmenge $W \subseteq V$ heißt *trennend*, falls $G - W$ unzusammenhängend ist. Für Graphen $G = (V, E)$, die keinen vollständigen Graphen der Ordnung $|V|$ enthalten, setzen wir $\kappa(G) := \min\{|W| \mid W \subseteq V \text{ ist trennend}\}$. Die Zahl $\kappa(G)$ heißt *Zusammenhangszahl* (oder *Knotenzusammenhangszahl*) von G. Für jeden Graphen $G = (V, E)$, der einen vollständigen Graphen der Ordnung $|V|$ enthält, setzen wir $\kappa(G) := |V| - 1$. Falls $\kappa(G) \geq k$, so nennen wir G *k-fach knotenzusammenhängend* (kurz: *k-zusammenhängend*). Ein wichtiger Satz der Graphentheorie (Satz von Menger) besagt, dass G k-fach zusammenhängend genau dann ist, wenn jedes Paar $s, t, s \neq t$, von Knoten durch mindestens k knotendisjunkte $[s, t]$-Wege miteinander verbunden ist. (Eine Menge von $[s, t]$-Wegen heißt *knotendisjunkt*, falls keine zwei Wege einen gemeinsamen inneren Knoten besitzen und die Menge der in den $[s, t]$-Wegen enthaltenen Kanten keine parallelen Kanten enthält.)

Eine Kantenmenge F eines Graphen $G = (V, E)$ heißt *trennend*, falls $G - F$ unzusammenhängend ist. Für Graphen G, die mehr als einen Knoten enthalten, setzen wir $\lambda(G) := \min\{|F| \mid F \subseteq E \text{ trennend}\}$. Die Zahl $\lambda(G)$ heißt *Kantenzusammenhangszahl*. Für Graphen G mit nur einem Knoten setzen wir $\lambda(G) = 0$. Falls $\lambda(G) \geq k$, so nennen wir G *k-fach kantenzusammenhängend* (kurz: *k-kantenzusammenhängend*). Eine Version des Menger'schen Satzes besagt, dass G k-kantenzusammenhängend genau dann ist, wenn jedes Paar $s, t, s \neq t$, von Knoten durch mindestens k kantendisjunkte $[s, t]$-Wege verbunden ist. Für Graphen G mit mindestens einem Knoten sind die Eigenschaften "G ist zusammenhängend", "G ist 1-kantenzusammenhängend" äquivalent.

Analoge Konzepte kann man in Digraphen definieren. Man benutzt hierbei den Zusatz "stark", um den "gerichteten Zusammenhang" zu kennzeichnen. Wir sagen, dass ein Digraph $D = (V, A)$ *stark k-zusammenhängend* (bzw. *stark k-bogenzusammenhängend*) ist, falls jedes Knotenpaar $s, t, s \neq t$ durch mindestens k knotendisjunkte (bzw. bogendisjunkte) (s, t)-Wege verbunden ist.

Wir setzen $\vec{\kappa}(D) := \max\{k \mid D \text{ stark } k\text{-zusammenhängend}\}$ und $\vec{\lambda}(D) := \max\{k \mid D \text{ stark } k\text{-bogenzusammenhängend}\}$; $\vec{\lambda}(D)$ heißt die *starke Zusammenhangszahl* von D, $\vec{\lambda}(D)$ die *starke Bogenzusammenhangszahl* von D.

Ein Kante e von G heißt *Brücke* (oder *Isthmus*), falls $G - e$ mehr Komponenten als G hat. Ein Knoten v von G heißt *Trennungsknoten* (oder *Artikulation*), falls die Kantenmenge E von G so in zwei nicht-leere Teilmengen E_1 und E_2 zerlegt werden kann, dass $V(E_1) \cap V(E_2) = \{v\}$ gilt. Ist G schlingenlos mit $|V| \geq 2$, dann ist v ein Trennungsknoten genau dann, wenn $\{v\}$ eine trennende Knotenmenge ist, d. h. wenn $G - v$ mehr Komponenten als G besitzt. Ein zusammenhängender Graph ohne Trennungsknoten wird *Block* genannt. Blöcke sind entweder isolierte Knoten, Schlingen oder Graphen mit 2 Knoten,

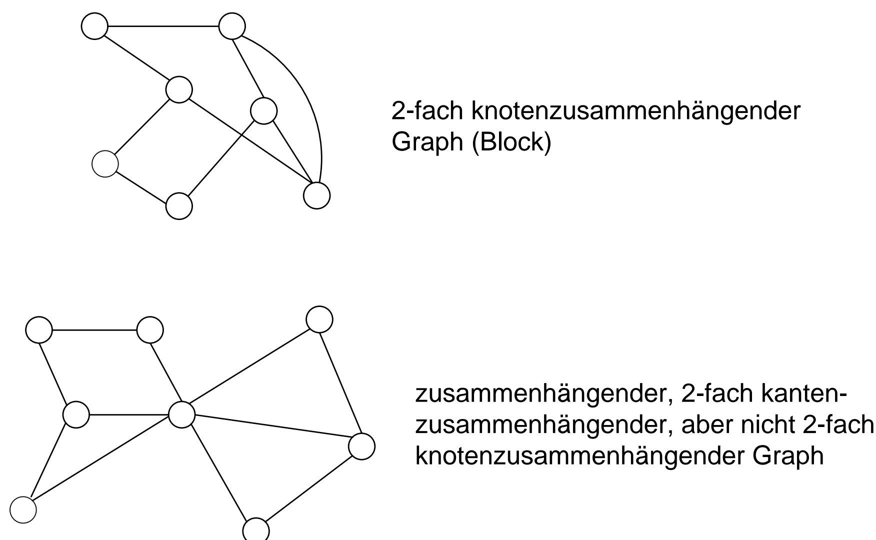


Abbildung 2.4: Ein Block und ein 2-fach kantenzusammenhängender Graph, der kein Block ist.

die durch eine Kante oder mehrere parallele Kanten verbunden sind oder, falls $|V| \geq 3$, 2-zusammenhängende Graphen. Ein *Block eines Graphen* ist ein Untergraph, der ein Block und maximal bezüglich dieser Eigenschaft ist. Jeder Graph ist offenbar die Vereinigung seiner Blöcke.

Abbildung 2.4 zeigt einen 2-fach knotenzusammenhängenden Graphen (Block) sowie einen zusammenhängenden, 2-fach kantenzusammenhängenden, aber nicht 2-fach knotenzusammenhängenden Graphen.

2.2 Lineare Algebra

2.2.1 Grundmengen

Wir benutzen folgende Bezeichnungen:

$$\begin{aligned} \mathbb{N} &= \{1,2,3,\dots\} = \text{Menge der } \textit{natürlichen Zahlen,} \\ \mathbb{Z} &= \text{Menge der } \textit{ganzen Zahlen,} \\ \mathbb{Q} &= \text{Menge der } \textit{rationalen Zahlen,} \\ \mathbb{R} &= \text{Menge der } \textit{reellen Zahlen,} \end{aligned}$$

Mit M_+ bezeichnen wir die Menge der nichtnegativen Zahlen in M für $M \in \{\mathbb{Z}, \mathbb{Q}, \mathbb{R}\}$. Wir betrachten $\mathbb{Q}$ und $\mathbb{R}$ als Körper mit der üblichen Addition und Multiplikation und der kanonischen Ordnung "$\leq$". Desgleichen betrachten wir $\mathbb{N}$ und $\mathbb{Z}$ als mit den üblichen Rechenarten versehen. Wir werden uns fast immer in diesen Zahlenuniversen bewegen, da diese die für die Praxis relevanten sind. Manche Sätze gelten jedoch nur, wenn wir

uns auf $\mathbb{Q}$ oder $\mathbb{R}$ beschränken. Um hier eine saubere Trennung zu haben, treffen wir die folgende Konvention. Wenn wir das Symbol

$$\mathbb{K}$$

benutzen, so heißt dies immer das $\mathbb{K}$ einer der angeordneten Körper $\mathbb{R}$ oder $\mathbb{Q}$ ist. Sollte ein Satz nur für $\mathbb{R}$ oder nur für $\mathbb{Q}$ gelten, so treffen wir die jeweils notwendige Einschränkung.

Für diejenigen, die sich für möglichst allgemeine Sätze interessieren, sei an dieser Stelle folgendes vermerkt. Jeder der nachfolgend angegebenen Sätze bleibt ein wahrer Satz, wenn wir als Grundkörper $\mathbb{K}$ einen archimedisch angeordneten Körper wählen. Ein bekannter Satz besagt, dass jeder archimedisch angeordnete Körper isomorph zu einem Unterkörper von $\mathbb{R}$ ist, der $\mathbb{Q}$ enthält. Unsere Sätze bleiben also richtig, wenn wir statt $\mathbb{K} \in \{\mathbb{Q}, \mathbb{R}\}$ irgendeinen archimedisch angeordneten Körper $\mathbb{K}$ mit $\mathbb{Q} \subseteq \mathbb{K} \subseteq \mathbb{R}$ wählen.

Wir können in fast allen Sätzen (insbesondere bei denen, die keine Ganzzahligkeitsbedingungen haben) auch die Voraussetzung "archimedisch" fallen lassen, d. h. fast alle Sätze gelten auch für angeordnete Körper. Vieles, was wir im $\mathbb{K}^n$ beweisen, ist auch in beliebigen metrischen Räumen oder Räumen mit anderen als euklidischen Skalarprodukten richtig. Diejenigen, die Spaß an derartigen Verallgemeinerungen haben, sind eingeladen, die entsprechenden Beweise in die allgemeinere Sprache zu übertragen.

In dieser Vorlesung interessieren wir uns für so allgemeine Strukturen nicht. Wir verbleiben in den (für die Praxis besonders wichtigen) Räumen, die über den reellen oder rationalen Zahlen errichtet werden. Also, nochmals, wenn immer wir das Symbol $\mathbb{K}$ im weiteren gebrauchen, gilt

$$\mathbb{K} \in \{\mathbb{R}, \mathbb{Q}\},$$

und $\mathbb{K}$ ist ein Körper mit den üblichen Rechenoperationen und Strukturen. Natürlich ist $\mathbb{K}_+ = \{x \in \mathbb{K} \mid x \geq 0\}$

Die Teilmengenbeziehung zwischen zwei Mengen M und N bezeichnen wir wie üblich mit $M \subseteq N$. Gilt $M \subseteq N$ und $M \neq N$, so schreiben wir $M \subset N$. $M \setminus N$ bezeichnet die mengentheoretische Differenz $\{x \in M \mid x \notin N\}$.

2.2.2 Vektoren und Matrizen

Ist R eine beliebige Menge, $n \in \mathbb{N}$, so bezeichnen wir mit

$$R^n$$

die Menge aller *n-Tupel* oder Vektoren der Länge n mit Komponenten aus R. (Aus technischen Gründen ist es gelegentlich nützlich, Vektoren $x \in R^0$, also Vektoren ohne Komponenten, zu benutzen. Wenn wir dies tun, werden wir es explizit erwähnen, andernfalls setzen wir immer $n \geq 1$ voraus.) Wir betrachten Vektoren $x = (x_i)_{i=1,\dots,n} \in R^n$ immer als *Spaltenvektoren* d. h.

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}.$$

Wollen wir mit Zeilenvektoren rechnen, so schreiben wir x^T (lies: x *transponiert*). Die Menge $\mathbb{K}^n$ ist bekanntlich ein n-dimensionaler Vektorraum über $\mathbb{K}$. Mit

$$y^T x := \sum_{i=1}^{n} x_i y_i$$

bezeichnen wir das *innere Produkt* zweier Vektoren $x, y \in \mathbb{K}^n$. Wir nennen x und y *senkrecht (orthogonal)*, falls $x^T y = 0$ gilt. Der $\mathbb{K}^n$ ist für uns immer (wenn nichts anderes gesagt wird) mit der *euklidischen Norm*

$$\|x\| := \sqrt{x^T x}$$

ausgestattet.

Für Mengen $S, T \subseteq \mathbb{K}^n$ und $\alpha \in \mathbb{K}$ benutzen wir die folgenden Standardbezeichnungen für Mengenoperationen

$$\begin{aligned} S + T &:= \{x + y \in \mathbb{K}^n \mid x \in S, y \in T\}, \\ S - T &:= \{x - y \in \mathbb{K}^n \mid x \in S, y \in T\}, \\ \alpha S &:= \{\alpha x \in \mathbb{K}^n \mid x \in S, \}. \end{aligned}$$

Einige Vektoren aus $\mathbb{K}^n$ werden häufig auftreten, weswegen wir sie mit besonderen Symbolen bezeichnen. Mit e_j bezeichnen wir den Vektor aus $\mathbb{K}^n$, dessen j-te Komponente 1 und dessen übrige Komponenten 0 sind. Mit 0 bezeichnen wir den Nullvektor, mit $\mathbb{1}$ den Vektor, dessen Komponenten alle 1 sind. Also

$$e_j = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \quad 0 = \begin{pmatrix} 0 \\ \vdots \\ \vdots \\ \vdots \\ 0 \end{pmatrix}, \quad \mathbb{1} = \begin{pmatrix} 1 \\ \vdots \\ \vdots \\ \vdots \\ 1 \end{pmatrix}.$$

Welche Dimension die Vektoren e_j, 0, $\mathbb{1}$ haben, ergibt sich jeweils aus dem Zusammenhang.

Für eine Menge R und $m, n \in \mathbb{N}$ bezeichnet

$$R^{(m,n)} \text{ oder } R^{m \times n}$$

die Menge der (m, n)-*Matrizen* (m Zeilen, n Spalten) mit Einträgen aus R. (Aus technischen Gründen werden wir gelegentlich auch $n = 0$ oder $m = 0$ zulassen, d. h. wir werden auch Matrizen mit m Zeilen und ohne Spalten bzw. n Spalten und ohne Zeilen betrachten. Dieser Fall wird jedoch immer explizit erwähnt, somit ist in der Regel $n \geq 1$ und $m \geq 1$ vorausgesetzt.) Ist $A \in R^{(m,n)}$, so schreiben wir

$$A = (a_{ij})_{\substack{i=1,\dots,m \\ j=1,\dots,n}}$$

und meinen damit, dass A die folgende Form hat

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix},$$

Wenn nichts anderes gesagt wird, hat A die Zeilenindexmenge $M = \{1, \dots, m\}$ und die Spaltenindexmenge $N = \{1, \dots, n\}$. Die *j-te Spalte* von A ist ein m-Vektor, den wir mit $A_{\cdot j}$ bezeichnen,

$$A_{\cdot j} = \begin{pmatrix} a_{1j} \\ \vdots \\ a_{mj} \end{pmatrix}.$$

Die *i-te Zeile* von A ist ein Zeilenvektor der Länge n, den wir mit $A_{i\cdot}$ bezeichnen, d. h.

$$A_{i\cdot} = (a_{i1}, a_{i2}, \dots, a_{in}).$$

Wir werden in dieser Vorlesung sehr häufig Untermatrizen von Matrizen konstruieren, sie umsortieren und mit ihnen rechnen müssen. Um dies ohne Zweideutigkeiten durchführen zu können, führen wir zusätzlich eine etwas exaktere als die oben eingeführte Standardbezeichnungsweise ein.

Wir werden — wann immer es aus technischen Gründen notwendig erscheint — die Zeilen- und Spaltenindexmengen einer (m, n)-Matrix A nicht als Mengen sondern als Vektoren auffassen. Wir sprechen dann vom

$$\begin{array}{rrcl} \text{vollen Zeilenindexvektor} & M & = & (1, 2, \dots, m) \\ \text{vollen Spaltenindexvektor} & N & = & (1, 2, \dots, n) \end{array}$$

von A. Ein *Zeilenindexvektor* von A ist ein Vektor mit höchstens m Komponenten, der aus M durch Weglassen einiger Komponenten von M und Permutation der übrigen Komponenten entsteht. Analog entsteht ein *Spaltenindexvektor* durch Weglassen von Komponenten von N und Permutation der übrigen Komponenten. Sind also

$$\begin{array}{rl} I = (i_1, i_2, \dots, i_p) & \text{ein Zeilenindexvektor von } A \text{ und} \\ J = (j_1, j_2, \dots, j_q) & \text{ein Spaltenindexvektor von } A, \end{array}$$

so gilt immer $i_s, i_t \in \{1, \dots, m\}$ und $i_s \neq i_t$ für $1 \leq s < t \leq m$, und analog gilt $j_s, j_t \in \{1, \dots, n\}$ und $j_s \neq j_t$ für $1 \leq s < t \leq n$. Wir setzen

$$A_{IJ} := \begin{pmatrix} a_{i_1 j_1} & a_{i_1 j_2} & \dots & a_{i_1 j_q} \\ a_{i_2 j_1} & a_{i_2 j_2} & \dots & \\ \vdots & \vdots & \ddots & \vdots \\ a_{i_p j_1} & a_{i_p j_2} & \dots & a_{i_p j_q} \end{pmatrix}$$

und nennen A_{IJ} *Untermatrix von A*. A_{IJ} ist also eine (p,q)-Matrix, die aus A dadurch entsteht, dass man die Zeilen, die zu Indizes gehören, die nicht in I enthalten sind, und die Spalten, die zu Indizes gehören, die nicht in J enthalten sind, streicht und dann die so entstehende Matrix umsortiert.

Ist $I = (i)$ und $J = (j)$, so erhalten wir zwei Darstellungsweisen für Zeilen bzw. Spalten von A:

$$\begin{aligned} A_{IN} &= A_{i\cdot}\ , \\ A_{MJ} &= A_{\cdot j}\ . \end{aligned}$$

Aus Gründen der Notationsvereinfachung werden wir auch die folgende (etwas unsaubere) Schreibweise benutzen. Ist M der volle Zeilenindexvektor von A und I ein Zeilenindexvektor, so schreiben wir auch

$$I \subseteq M,$$

obwohl I und M keine Mengen sind, und für $i \in \{1,\ldots,m\}$ benutzen wir

$$i \in I \quad \text{oder } i \notin I,$$

um festzustellen, dass i als Komponente von I auftritt oder nicht. Analog verfahren wir bezüglich der Spaltenindizes.

Gelegentlich spielt die tatsächliche Anordnung der Zeilen und Spalten keine Rolle. Wenn also z. B. $I \subseteq \{1,\ldots,n\}$ und $J \subseteq \{1,\ldots,m\}$ gilt, dann werden wir auch einfach schreiben

$$A_{IJ},$$

obwohl diese Matrix dann nur bis auf Zeilen- und Spaltenpermutationen definiert ist. Wir werden versuchen, diese Bezeichnungen immer so zu verwenden, dass keine Zweideutigkeiten auftreten. Deshalb treffen wir ab jetzt die Verabredung, dass wir — falls wir Mengen (und nicht Indexvektoren) I und J benutzen — die Elemente von $I = \{i_1,\ldots,i_p\}$ und von $J = \{j_1,\ldots,j_q\}$ kanonisch anordnen, d. h. die Indizierung der Elemente von I und J sei so gewählt, dass $i_1 < i_2 < \ldots < i_p$ und $j_1 < j_2 < \ldots < j_q$ gilt. Bei dieser Verabredung ist dann A_{IJ} die Matrix die aus A durch Streichen der Zeilen i, $i \notin I$, und der Spalten j, $j \notin J$, entsteht.

Ist $I \subseteq \{1,\ldots,m\}$ und $J \subseteq \{1,\ldots,n\}$ oder sind I und J Zeilen- bzw. Splaltenindexvektoren, dann schreiben wir auch

$$A_{I\cdot} \quad \text{statt} \quad A_{IN}$$

$$A_{\cdot J} \quad \text{statt} \quad A_{MJ}$$

$A_{I\cdot}$ entsteht also aus A durch Streichen der Zeilen i, $i \notin I$, $A_{\cdot J}$ durch Streichen der Spalten j, $j \notin J$.

Sind $A, B \in \mathbb{K}^{(m,n)}$, $C \in \mathbb{K}^{(n,s)}$, $\alpha \in \mathbb{K}$, so sind

$$\begin{aligned} \text{die } \textit{Summe} &\quad A + B\ , \\ \text{das } \textit{Produkt} &\quad \alpha A, \\ \text{das } \textit{Matrixprodukt} &\quad \text{AC} \end{aligned}$$

wie in der linearen Algebra üblich definiert.

Für einige häufig auftretende Matrizen haben wir spezielle Symbole reserviert. Mit 0 bezeichnen wir die Nullmatrix (alle Matrixelemente sind Null), wobei sich die Dimension der Nullmatrix jeweils aus dem Zusammenhang ergibt. (Das Symbol 0 kann also sowohl eine Zahl, einen Vektor als auch eine Matrix bezeichnen). Mit I bezeichnen wir die Einheitsmatrix. Diese Matrix ist quadratisch, die Hauptdiagonalelemente von I sind Eins, alle übrigen Null. Wollen wir die Dimension von I betonen, so schreiben wir auch I_n und meinen damit die (n,n)-Einheitsmatrix. Diejenige (m,n)-Matrix, bei der alle Elemente Eins sind, bezeichnen wir mit E. Wir schreiben auch $E_{m,n}$ bzw. E_n, um die Dimension zu spezifizieren (E_n ist eine (n,n)-Matrix). Ist x ein n-Vektor, so bezeichnet $\operatorname{diag}(x)$ diejenige (n,n)-Matrix $A = (a_{ij})$ mit $a_{ii} = x_i$ $(i = 1,\ldots,n)$ und $a_{ij} = 0$ $(i \neq j)$.

Wir halten an dieser Stelle noch einmal Folgendes fest: Wenn wir von einer Matrix A sprechen, ohne anzugeben, welche Dimension sie hat und aus welchem Bereich sie ist, dann nehmen wir implizit an, dass $A \in \mathbb{K}^{(m,n)}$ gilt. Analog gilt immer $x \in \mathbb{K}^n$, wenn sich nicht aus dem Zusammenhang anderes ergibt.

2.2.3 Kombinationen von Vektoren, Hüllen, Unabhängigkeit

Ein Vektor $x \in \mathbb{K}^n$ heißt *Linearkombination* der Vektoren $x_1,\ldots,x_k \in \mathbb{K}^n$, falls es einen Vektor $\lambda = (\lambda_1,\ldots,\lambda_k)^T \in \mathbb{K}^k$ gibt mit

$$x = \sum_{i=1}^{k} \lambda_i x_i \ .$$

Gilt zusätzlich:

$$\left.\begin{array}{l} \lambda \geq 0 \\ \lambda^T \mathbb{1} = 1 \\ \lambda \geq 0 \quad \text{und} \quad \lambda^T \mathbb{1} = 1 \end{array}\right\} \text{ so heißt } x \left\{\begin{array}{l} \textit{konische} \\ \textit{affine} \\ \textit{konvexe} \end{array}\right\} \textit{Kombination}$$

der Vektoren $x_1,\ldots,x_k$. Diese Kombinationen heißen *echt*, falls weder $\lambda = 0$ noch $\lambda = e_j$ für ein $j \in \{1,\ldots,k\}$ gilt.

Für eine nichtleere Teilmenge $S \subseteq \mathbb{K}^n$ heißt

$$\left.\begin{array}{l} \operatorname{lin}(S) \\ \operatorname{cone}(S) \\ \operatorname{aff}(S) \\ \operatorname{conv}(S) \end{array}\right\} \text{ die } \left\{\begin{array}{l} \textit{lineare} \\ \textit{konische} \\ \textit{affine} \\ \textit{konvexe} \end{array}\right\} \textit{Hülle} \text{ von } S, \text{ d. h.}$$

die Menge aller Vektoren, die als lineare (konische, affine oder konvexe) Kombination von endlich vielen Vektoren aus S dargestellt werden können. Wir setzen außerdem

$$\begin{array}{ll} \operatorname{lin}(\emptyset) & := \operatorname{cone}(\emptyset) := \{0\}, \\ \operatorname{aff}(\emptyset) & := \operatorname{conv}(\emptyset) := \emptyset. \end{array}$$

Ist A eine (m,n)-Matrix, so schreiben wir auch

$$\operatorname{lin}(A),\ \operatorname{cone}(A),\ \operatorname{aff}(A),\ \operatorname{conv}(A)$$

und meinen damit die lineare, konische, affine bzw. konvexe Hülle der Spaltenvektoren $A_{\cdot 1}, A_{\cdot 2}, \ldots, A_{\cdot n}$ von A. Eine Teilmenge $S \subseteq \mathbb{K}^n$ heißt

$$\left\{ \begin{array}{l} \textit{linearer Raum} \\ \textit{Kegel} \\ \textit{affiner Raum} \\ \textit{konvexe Menge} \end{array} \right\} \quad \text{falls} \quad \left\{ \begin{array}{l} S = \operatorname{lin}(S) \\ S = \operatorname{cone}(S) \\ S = \operatorname{aff}(S) \\ S = \operatorname{conv}(S) \end{array} \right\}.$$

Die hier benutzten Begriffe sind üblicher Standard, wobei für „linearen Raum“ in der lineraren Algebra in der Regel *Vektorraum* oder *Untervektorraum* benutzt wird. Der Begriff *Kegel* wird jedoch – u. a. in verschiedenen Zweigen der Geometrie – allgemeiner verwendet. „Unser Kegel“ ist in der Geometrie ein „abgeschlossener konvexer Kegel“.

Eine nichtleere endliche Teilmenge $S \subseteq \mathbb{K}^n$ heißt *linear* (bzw. *affin*) *unabhängig*, falls kein Element von S als echte Linearkombination (bzw. Affinkombination) von Elementen von S dargestellt werden kann. Die leere Menge ist affin, jedoch nicht linear unabhängig. Jede Menge $S \subseteq \mathbb{K}^n$, die nicht linear bzw. affin unabhängig ist, heißt *linear* bzw. *affin abhängig*. Aus der linearen Algebra wissen wir, dass eine linear (bzw. affin) unabhängige Teilmenge des $\mathbb{K}^n$ höchstens n (bzw. $n+1$) Elemente enthält. Für eine Teilmenge $S \subseteq \mathbb{K}^n$ heißt die Kardinalität einer größten linear (bzw. affin) unabhängigen Teilmenge von S der *Rang* (bzw. *affine Rang*) von S. Wir schreiben dafür $\operatorname{rang}(S)$ bzw. $\operatorname{arang}(S)$. Die *Dimension* einer Teilmenge $S \subseteq \mathbb{K}^n$, Bezeichnung: $\dim(S)$, ist die Kardinalität einer größten affin unabhängigen Teilmenge von S minus 1, d. h. $\dim(S) = \operatorname{arang}(S) - 1$.

Der *Rang einer Matrix* A, bezeichnet mit $\operatorname{rang}(A)$, ist der Rang ihrer Spaltenvektoren. Aus der linearen Algebra wissen wir, dass $\operatorname{rang}(A)$ mit dem Rang der Zeilenvektoren von A übereinstimmt. Gilt für eine (m, n)-Matrix A, $\operatorname{rang}(A) = \min\{m, n\}$, so sagen wir, dass A *vollen Rang* hat. Eine (n, n)-Matrix mit vollem Rang ist *regulär*, d. h. sie besitzt eine (eindeutig bestimmte) inverse Matrix (geschrieben A^{-1}) mit der Eigenschaft $AA^{-1} = I$.

2.3 Polyeder und lineare Programme

Ein wichtiger Aspekt der Vorlesung ist die Behandlung von Problemen der linearen Programmierung bzw. die Modellierung von kombinatorischen Optimierungsproblemen als (ganzzahlige) lineare Programme. In diesem Abschnitt stellen wir einige grundlegende Begriffe der zugehörigen Theorie bereit, stellen dar, in welchen Formen lineare Programme vorkommen, und wie man sie ineinander transformiert. Schließlich beweisen wir als Einführung den schwachen Dualitätssatz.

(2.1) Definition.

a) *Eine Teilmenge* $G \subseteq \mathbb{K}^n$ *heißt* Hyperebene, *falls es einen Vektor* $a \in \mathbb{K}^n \setminus \{0\}$ *und* $\alpha \in \mathbb{K}$ *gibt mit*

$$G = \{x \in \mathbb{K}^n \mid a^T x = \alpha\}.$$

Der Vektor a *heißt* Normalenvektor *zu* G.

b) *Eine Teilmenge* $H \subseteq \mathbb{K}^n$ *heißt* Halbraum, *falls es einen Vektor* $a \in \mathbb{K}^n \setminus \{0\}$ *und* $\alpha \in \mathbb{K}$ *gibt mit*

$$H = \{x \in \mathbb{K}^n \mid a^T x \leq \alpha\}.$$

Wir nennen a *den* Normalenvektor *zu* H. *Die Hyperebene* $G = \{x \in \mathbb{K}^n \mid a^T x = \alpha\}$ *heißt die zum Halbraum* H *gehörende Hyperebene oder die* H berandende Hyperebene, *und* H *heißt der zu* G *gehörende Halbraum.*

c) *Eine Teilmenge* $P \subseteq \mathbb{K}^n$ *heißt* Polyeder, *falls es ein* $m \in \mathbb{Z}_+$, *eine Matrix* $A \in \mathbb{K}^{(m,n)}$ *und einen Vektor* $b \in \mathbb{K}^m$ *gibt mit*

$$P = \{x \in \mathbb{K}^n \mid Ax \leq b\}.$$

Um zu betonen, dass P *durch* A *und* b *definiert ist, schreiben wir auch*

$$P = P(A, b) := \{x \in \mathbb{K}^n \mid Ax \leq b\}.$$

d) *Ein Polyeder* P *heißt* Polytop, *wenn es beschränkt ist, d. h. wenn es ein* $B \in \mathbb{K}$, $B > 0$ *gibt mit* $P \subseteq \{x \in \mathbb{K}^n \mid \|x\| \leq B\}$. △

Polyeder können wir natürlich auch in folgender Form schreiben

$$P(A, b) = \bigcap_{i=1}^{m} \{x \in \mathbb{K}^n \mid A_{i\cdot} x \leq b_i\}.$$

Halbräume sind offensichtlich Polyeder. Aber auch die leere Menge ist ein Polyeder, denn $\emptyset = \{x \mid 0^T x \leq -1\}$, und der gesamte Raum ist ein Polyeder, denn $\mathbb{K}^n = \{x \mid 0^T x \leq 0\}$. Sind alle Zeilenvektoren $A_{i\cdot}$ von A vom Nullvektor verschieden, so sind die bei der obigen Durchschnittsbildung beteiligten Mengen Halbräume. Ist ein Zeilenvektor von A der Nullvektor, sagen wir $A_{1\cdot} = 0^T$, so ist $\{x \in \mathbb{K}^n \mid A_{1\cdot} x \leq b_1\}$ entweder leer (falls $b_1 < 0$) oder der gesamte Raum $\mathbb{K}^n$ (falls $b_1 \geq 0$). Das heißt, entweder ist $P(A, b)$ leer oder die Mengen $\{x \mid A_{i\cdot} \leq b_i\}$ mit $A_{i\cdot} = 0$ können bei der obigen Durchschnittsbildung weggelassen werden. Daraus folgt

Jedes Polyeder $P \neq \mathbb{K}^n$ ist Durchschnitt von endlich vielen Halbräumen.

Gilt $P = P(A, b)$, so nennen wir das Ungleichungssystem $Ax \leq b$ ein P *definierendes System* (von linearen Ungleichungen). Sind $\alpha > 0$ und $1 \leq i < j \leq m$, so gilt offensichtlich

$$P(A, b) = P(A, b) \cap \{x \mid \alpha A_{i\cdot} x \leq \alpha b_i\} \cap \{x \mid (A_{i\cdot} + A_{j\cdot}) x \leq b_i + b_j\}.$$

Daraus folgt, dass A und b zwar $P = P(A, b)$ eindeutig bestimmen, dass aber P unendlich viele Darstellungen der Form $P(D, d)$ hat.

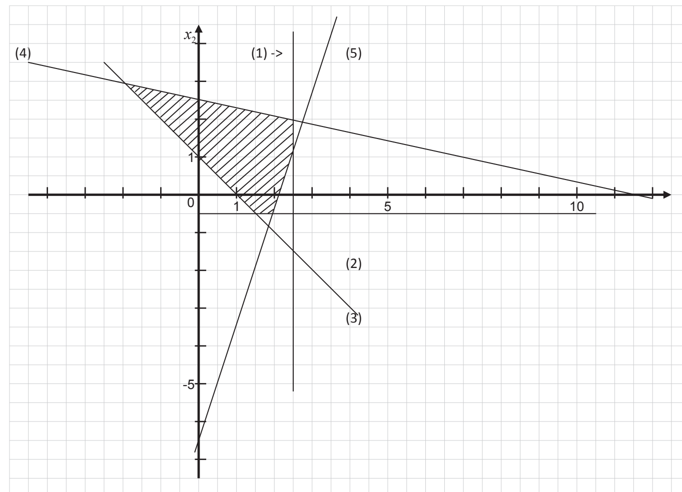


Abbildung 2.5: Darstellung des Polyeders aus Beispiel (2.2).

(2.2) Beispiel. Wir betrachten das Ungleichungssystem

$$
\begin{aligned}
2x_1 \qquad\quad &\leq 5 && (1)\\
-\,2x_2 &\leq 1 && (2)\\
-x_1 - \;\; x_2 &\leq -1 && (3)\\
2x_1 + 9x_2 &\leq 23 && (4)\\
6x_1 - 2x_2 &\leq 13 && (5)
\end{aligned}
$$

Hieraus erhalten wir die folgende Matrix A und den Vektor b:

$$
A = \begin{pmatrix} 2 & 0 \\ 0 & -2 \\ -1 & -1 \\ 2 & 9 \\ 6 & -2 \end{pmatrix}, \qquad b = \begin{pmatrix} 5 \\ 1 \\ -1 \\ 23 \\ 13 \end{pmatrix}
$$

Das Polyeder $P = P(A, b)$ ist die Lösungsmenge des obigen Ungleichungssystems (1)–(5) und ist in Abbildung 2.5 graphisch dargestellt. △

Die Mengen zulässiger Lösungen linearer Programme treten nicht immer in der Form $Ax \leq b$ auf. Häufig gibt es auch Gleichungen und vorzeichenbeschränkte Variable. Vorzeichenbeschränkungen sind natürlich auch lineare Ungleichungssysteme, und ein Gleichungssytem $Dx = d$ kann in der Form von zwei Ungleichungssystemen $Dx \leq d$ und $-Dx \leq -d$ geschrieben werden. Allgemeiner gilt

(2.4) Bemerkung. Die Lösungsmenge des Systems

$$\begin{array}{lll} Bx + Cy & & = c \\ Dx + Ey & & \leq d \\ x & & \geq 0 \\ x \in \mathbb{K}^p, & & y \in \mathbb{K}^q \end{array}$$

ist ein Polyeder. △

Beweis. Setze $n := p + q$ und

$$A := \begin{pmatrix} B & C \\ -B & -C \\ D & E \\ -I & 0 \end{pmatrix}, \quad b := \begin{pmatrix} c \\ -c \\ d \\ 0 \end{pmatrix}.$$

Dann ist $P(A, b)$ die Lösungsmenge des vorgegebenen Gleichungs- und Ungleichungssystems. □

Ein spezieller Polyedertyp wird uns häufig begegnen, weswegen wir für ihn eine besondere Bezeichnung wählen wollen. Für $A \in \mathbb{K}^{(m,n)}$, $b \in \mathbb{K}^m$ setzen wir

$$P^=(A, b) := \{x \in \mathbb{K}^n \mid Ax = b, x \geq 0\}.$$

Nicht alle Polyeder können in der Form $P^=(A, b)$ dargestellt werden, z. B. nicht $P = \{x \in \mathbb{K} \mid x \leq 1\}$.

Wir werden später viele Sätze über Polyeder P beweisen, deren Aussagen *darstellungsabhängig* sind, d. h. die Art und Weise, wie P gegeben ist, geht explizit in die Satzaussage ein. So werden sich z. B. die Charakterisierungen gewisser Polyedereigenschaften von $P(A, b)$ (zumindest formal) von den entsprechenden Charakterisierungen von $P^=(A, b)$ unterscheiden. Darstellungsabhängige Sätze wollen wir jedoch nur einmal beweisen (normalerweise für Darstellungen, bei denen die Resultate besonders einprägsam oder einfach sind), deshalb werden wir uns nun Transformationsregeln überlegen, die angeben, wie man von einer Darstellungsweise zu einer anderen und wieder zurück kommt.

(2.5) Transformationen.

Regel I: *Einführung von Schlupfvariablen*

Gegeben seien $a \in \mathbb{K}^n$, $\alpha \in \mathbb{K}$. Wir schreiben die Ungleichung

$$a^T x \leq \alpha \tag{2.6}$$

in der Form einer Gleichung und einer Vorzeichenbeschränkung

$$a^T x + y = \alpha, \quad y \geq 0. \tag{2.7}$$

y ist eine neue Variable, genannt *Schlupfvariable*. Es gilt:

$$\begin{aligned} x \text{ erfüllt (2.6)} \quad &\Longrightarrow \quad \binom{x}{y} \text{ erfüllt (2.7) für } y = \alpha - a^T x, \\ \binom{x}{y} \text{ erfüllt (2.7)} \quad &\Longrightarrow \quad x \text{ erfüllt (2.7).} \end{aligned}$$

Allgemein: Ein Ungleichungssystem $Ax \leq b$ kann durch Einführung eines Schlupfvariablenvektors y transformiert werden in ein Gleichungssystem mit Vorzeichenbedingung $Ax + y = b$, $y \geq 0$. Zwischen den Lösungsmengen dieser beiden Systeme bestehen die oben angegebenen Beziehungen. $P(A, b)$ und $\{\binom{x}{y} \in \mathbb{K}^{n+m} \mid Ax + Iy = b, y \geq 0\}$ sind jedoch zwei durchaus verschiedene Polyeder in verschiedenen Vektorräumen.

Regel II: *Einführung von vorzeichenbeschränkten Variablen*

Ist x eine (eindimensionale) nicht vorzeichenbeschränkte Variable, so können wir zwei vorzeichenbeschränkte Variablen x^+ und x^- einführen, um x darzustellen. Wir setzen

$$x := x^+ - x^- \quad \text{mit} \quad x^+ \geq 0,\ x^- \geq 0. \qquad \triangle$$

Mit den Regeln I und II aus (2.5) können wir z. B. jedem Polyeder $P(A, b) \subseteq \mathbb{K}^n$ ein Polyeder $P^=(D, d) \subseteq \mathbb{K}^{2n+m}$ wie folgt zuordnen. Wir setzen

$$D := (A, -A, I_m),\ d := b,$$

d. h. es gilt

$$P^=(D, d) = \left\{(x, y, z) \in \mathbb{K}^{2n+m} \mid Ax - Ay + z = b, x, y, z \geq 0\right\}.$$

Es ist üblich, die oben definierten Polyeder $P(A, b)$ und $P^=(D, d)$ *äquivalent* zu nennen. Hierbei hat "Äquivalenz" folgende Bedeutung. Für $x \in \mathbb{K}^n$ sei

$$\begin{aligned} x^+ &:= (x_1^+, \ldots, x_n^+)^T \quad \text{mit} \quad x_i^+ = \max\{0, x_i\}, \\ x^- &:= (x_1^-, \ldots, x_n^-)^T \quad \text{mit} \quad x_i^- = \max\{0, -x_i\}. \end{aligned}$$

Dann gilt:

$$\begin{aligned} x \in P(A, b) \quad &\Longrightarrow \quad \begin{pmatrix} x^+ \\ x^- \\ z \end{pmatrix} \in P^=(D, d) \text{ für } z = b - Ax \\ \begin{pmatrix} u \\ v \\ w \end{pmatrix} \in P^=(D, d) \quad &\Longrightarrow \quad x := u - v \in P(A, b). \end{aligned}$$

Besonders einfache Polyeder, auf die sich jedoch fast alle Aussagen der Polyedertheorie zurückführen lassen, sind polyedrische Kegel.

(2.8) Definition. *Ein Kegel $C \subseteq \mathbb{K}^n$ heißt* polyedrisch *genau dann, wenn C ein Polyeder ist.* △

(2.9) Bemerkung. Ein Kegel $C \subseteq \mathbb{K}^n$ ist genau dann polyedrisch, wenn es eine Matrix $A \in \mathbb{K}^{(m,n)}$ gibt mit
$$C = P(A, 0).$$
△

Beweis. Gilt $C = P(A, 0)$, so ist C ein Polyeder und offensichtlich ein Kegel.

Sei C ein polyedrischer Kegel, dann existieren nach Definition (2.1) c) eine Matrix A und ein Vektor b mit $C = P(A, b)$. Da jeder Kegel den Nullvektor enthält, gilt $0 = A0 \leq b$. Angenommen, es existiert ein $\overline{x} \in C$ mit $A\overline{x} \not\leq 0$, d. h. es existiert eine Zeile von A, sagen wir $A_{i.}$, mit $t := A_{i.}\overline{x} > 0$. Da C ein Kegel ist, gilt $\lambda\overline{x} \in C$ für alle $\lambda \in \mathbb{K}_+$. Jedoch für $\overline{\lambda} := \frac{b_i}{t} + 1$ gilt einerseits $\overline{\lambda}\overline{x} \in C$ und andererseits $A_{i.}(\overline{\lambda}\overline{x}) = \overline{\lambda}t > b_i$, ein Widerspruch. Daraus folgt, für alle $x \in C$ gilt $Ax \leq 0$. Hieraus ergibt sich $C = P(A, 0)$. □

In der folgenden Tabelle 2.1 haben wir alle möglichen Transformationen aufgelistet. Sie soll als "Nachschlagewerk" dienen.

Eine wichtige Eigenschaft von linearen Programmen ist die Tatsache, dass man effizient erkennen kann, ob ein gegebener Punkt tatsächlich der gesuchte Optimalpunkt ist oder nicht. Dahinter steckt die sogenannte *Dualitätstheorie*, die wir anhand unseres Einführungsbeispiels aus Kapitel 1.1 erläutern wollen. Durch Anwenden der Transformationsregeln können wir das LP (1.3) in der Form

$$\begin{aligned}
\min -6s + t & & \text{(2.10a)}\\
-s + t &\geq -1 & \text{(2.10b)}\\
s - t &\geq -1 & \text{(2.10c)}\\
s \qquad &\geq 2 & \text{(2.10d)}\\
-s \qquad &\geq -3 & \text{(2.10e)}\\
t &\geq 0 & \text{(2.10f)}\\
-t &\geq -3. & \text{(2.10g)}
\end{aligned}$$

schreiben. Wir wollen nun zeigen, dass der Punkt $x^* = (3, 2)$ mit Zielfunktionswert -16 minimal ist.

Ein Weg, dies zu beweisen, besteht darin, untere Schranken für das Minimum zu finden. Wenn man sogar in der Lage ist, eine untere Schranke zu bestimmen, die mit dem Zielfunktionswert einer Lösung des Ungleichungssystems übereinstimmt, ist man fertig: Der Zielfunktionswert irgendeiner Lösung ist immer eine obere Schranke, und wenn untere und obere Schranke gleich sind, ist der optimale Wert gefunden.

Um eine untere Schranke für die Zielfunktion $-6s + t$ zu finden, können wir uns zunächst die Schranken der Variablen anschauen. Wegen Ungleichung (2.10e) haben wir $-6s \geq -18$. Zusammen mit Ungleichung (2.10f) erhalten wir $-6s + t \geq -18$. Um zu dieser unteren Schranke zu gelangen, haben wir positive Vielfache der Variablenschranken

Tabelle 2.1: Transformationen zwischen Polyedern.

Transformation nach / von	$By \leq d$	$By = d$, $y \geq 0$	$By \leq d$, $y \geq 0$
$Ax = b$ hierbei ist $x_i^+ = \max\{0, x_i\}$ $x_i^- = \max\{0, -x_i\}$	$\begin{pmatrix} A \\ -A \end{pmatrix} y \leq \begin{pmatrix} b \\ -b \end{pmatrix}$ $y = x$	$(A, -A)\, y = b$ $y \geq 0$ $y = \begin{pmatrix} x^+ \\ x^- \end{pmatrix}$	$\begin{pmatrix} A & -A \\ -A & A \end{pmatrix} y \leq \begin{pmatrix} b \\ -b \end{pmatrix}$ $y \geq 0$ $y = \begin{pmatrix} x^+ \\ x^- \end{pmatrix}$
$Ax \leq b$	$Ay \leq b$ $y = x$	$(A, -A, I)\, y = b$ $y \geq 0$ $y = \begin{pmatrix} x^+ \\ x^- \\ b - Ax \end{pmatrix}$	$(A, -A)\, y \leq b$ $y \geq 0$ $y = \begin{pmatrix} x^+ \\ x^- \end{pmatrix}$
$Ax = b$ $x \geq 0$	$\begin{pmatrix} A \\ -A \\ -I \end{pmatrix} y \leq \begin{pmatrix} b \\ -b \\ 0 \end{pmatrix}$ $y = x$	$Ay = b$ $y \geq 0$ $y = x$	$\begin{pmatrix} A \\ -A \end{pmatrix} y \leq \begin{pmatrix} b \\ -b \end{pmatrix}$ $y \geq 0$ $y = x$
$Ax \leq b$ $x \geq 0$	$\begin{pmatrix} A \\ -I \end{pmatrix} y \leq \begin{pmatrix} b \\ 0 \end{pmatrix}$ $y = x$	$(A, I)\, y = b$ $y \geq 0$ $y = \begin{pmatrix} x \\ b - Ax \end{pmatrix}$	$Ay \leq b$ $y \geq 0$ $y = x$

so addiert, das wir auf der linken Seite die Zielfunktion erhalten. (Da unsere Ungleichungen alle in der Form "$\geq$" geschrieben sind, dürfen wir nur positiv skalieren, denn zwei Ungleichungen $a_1x_1 + a_2x_2 \leq \alpha$ und $b_1x_1 + b_2x_2 \geq \beta$ kann man nicht addieren.) Natürlich können wir dies mit allen Nebenbedingungsungleichungen machen, um so potenziell bessere untere Schranken zu bekommen. Gibt jede beliebige Skalierung und Addition eine untere Schranke? Machen wir noch einen Versuch. Addieren wir die Ungleichungen (2.10b) und (2.10e), so erhalten wir $-2s + t \geq -4$. Das hilft uns nicht weiter, denn die linke Seite der Ungleichung kann nicht zum Abschätzen der Zielfunktion benutzt werden: *Damit die rechte Seite der neuen Ungleichung eine untere Schranke für die Zielfunktion liefert, muss auf der linken Seite jeder Koeffizient höchstens so groß sein wie der entsprechende Koeffizient der Zielfunktion.*

Diese Erkenntnisse liefern uns ein neues mathematisches Problem. Wir suchen nichtnegative Multiplikatoren (Skalierungsfaktoren) der Ungleichungen (2.10b)–(2.10g) mit gewissen Eigenschaften. Multiplizieren wir die Ungleichungen mit $y_1, \dots, y_6$, so darf die Summe $-y_1 + y_2 + y_3 - y_4$ nicht den Wert des ersten Koeffizienten der Zielfunktion (also -6) überschreiten. Analog darf die Summe $y_1 - y_2 + y_5 - y_6$ nicht den Wert 1 überschreiten. Und die y_i sollen nicht negativ sein. Ferner soll die rechte Seite der Summenungleichung, also $-y_1 - y_2 + 2y_3 - 3y_4 - 3y_6$ so groß wie möglich werden. Daraus folgt, dass wir die folgende Aufgabe lösen müssen:

$$\max -y_1 - y_2 + 2y_3 - 3y_4 - 3y_6 \tag{2.11a}$$
$$-y_1 + y_2 + y_3 - y_4 \leq -6 \tag{2.11b}$$
$$y_1 - y_2 + y_5 - y_6 \leq 1 \tag{2.11c}$$
$$y_1,\quad y_2,\quad y_3,\quad y_4,\quad y_5,\quad y_6 \geq 0. \tag{2.11d}$$

Auch (2.11) ist ein lineares Programm. Aus unseren Überlegungen folgt, dass der Maximalwert von (2.11) höchstens so groß ist wie der Minimalwert von (1.3), da jede Lösung von (2.11) eine untere Schranke für (1.3) liefert. Betrachten wir z. B. den Punkt

$$y^* = (1, 0, 0, 5, 0, 0).$$

Durch Einsetzen in die Ungleichungen von (2.11) sieht man dass y^* alle Ungleichungen erfüllt. Addieren wir also zu Ungleichung (2.10b) das 5-fache der Ungleichung (2.10f), so erhalten wir

$$-6s + t \geq -16.$$

Damit wissen wir, dass der Minimalwert von (1.3) mindestens -16 ist. Der Punkt $x^* = (3, 2)$ liefert gerade diesen Wert, er ist also optimal. Und außerdem ist y^* eine Optimallösung von (2.11).

Die hier dargestellten Ideen bilden die Grundgedanken der Dualitätstheorie der linearen Programmierung, und sie sind außerordentlich nützlich bei der Entwicklung von Algorithmen und in Beweisen. In der Tat haben wir bereits ein kleines Resultat erzielt, das wir wie folgt formal zusammenfassen wollen.

(2.12) Satz. *Es seien $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, und A sei eine reelle (m, n)-Matrix. Betrachten wir die Aufgaben*

(P) $\min\{c^T x \mid Ax \geq b, x \geq 0\}$ *und*

(D) $\max\{y^T b \mid y^T A \leq c^T, y \geq 0\}$,

dann gilt Folgendes: Seien $x_0 \in \mathbb{R}^n$ *und* $y_0 \in \mathbb{R}^m$ *Punkte mit* $Ax_0 \geq b$, $x_0 \geq 0$ *und* $y_0^T A \leq c^T$, $y_0 \geq 0$, *dann gilt*

$$y_0^T b \leq c^T x_0.$$

△

Beweis. Durch einfaches Einsetzen:

$$y_0^T b \leq y_0^T (Ax_0) = (y_0^T A)x_0 \leq c^T x_0.$$

□

Satz (2.12) wird *schwacher Dualitätssatz* genannt, (D) heißt das *zu (P) duale LP* und (P) wird in diesem Zusammenhang als *primales LP* bezeichnet. Für Optimallösungen x^* und y^* von (P) bzw. (D) gilt nach (2.12) $y^{*T} b \leq c^T x^*$, das duale Problem liefert also stets eine untere Schranke für das primale Problem. Wir werden später zeigen, dass in diesem Falle sogar immer $y^{*T} b = c^T x^*$ gilt. Diese *starke Dualität*, also das Übereinstimmen der Optimalwerte von (P) und (D) ist allerdings nicht ganz so einfach zu beweisen wie Satz (2.12).

Die schwache Dualität überträgt sich auch auf ganzzahlige lineare Programme, für die jedoch im Allgemeinen kein starker Dualitätssatz gilt. Genauer gilt folgender Zusammenhang:

$$\begin{aligned}
&\min\{c^T x \mid Ax \geq b, x \geq 0, x \in \mathbb{Z}\} \\
&\geq \min\{c^T x \mid Ax \geq b, x \geq 0\} \\
&= \max\{y^T b \mid y^T A \leq c^T, y \geq 0\} \\
&\geq \max\{y^T b \mid y^T A \leq c^T, y \geq 0, y \in \mathbb{Z}\}.
\end{aligned}$$

Literaturverzeichnis

Zur linearen Algebra existieren unzählige Bücher. Deswegen geben wir hierzu keine Literaturliste an, sondern listen hier ausschließlich Literatur zur Graphentheorie und zur linearen Programmierung.

M. Aigner. *Graphentheorie: Eine Entwicklung aus dem 4-Farben-Problem.* Teubner Verlag, Studienbücher : Mathematik, Stuttgart, 1984. ISBN 3-519-02068-8.

Berge and Ghouila-Houri. *Programme, Spiele, Transportnetze.* Teubner Verlag, Leipzig, 1969.

B. Bollobás. *Modern Graph Theory.* Springer Verlag, New York, 1998. ISBN 0-387-98488-7.

J. A. Bondy and U. S. R. Murty. *Graph Theory.* Springer, Berlin, 2008.

V. Chvátal. *Linear Programming.* Freeman, 1983.

G. B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, 1998.

R. Diestel. *Graphentheorie.* Springer-Verlag, Heidelberg, 3. auflage edition, 2006. ISBN 3-540-21391-0.

W. Domschke. *Logistik: Rundreisen und Touren.* Oldenbourg-Verlag, München - Wien, (4., erweiterte Aufl. 1997, 1982.

J. Ebert. Effiziente Graphenalgorithmen. *Studientexte: Informatik*, 1981.

M. C. Golombic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, New York. ISBN 1980.

R. L. Graham, M. Grötschel, and L. Lovász, editors. *Handbook of Combinatorics, Volume I.* Elsevier (North-Holland); The MIT Press, Cambridge, Massachusetts, 1995. ISBN ISBN 0-444-82346-8/v.1 (Elsevier); ISBN 0-262-07170-3/v.1 (MIT).

J. L. Gross and J. Yellen. *Handbook of Graph Theory.* CRC Press, Boca Raton, 2004. ISBN 1-58488-090-2.

R. Halin. *Graphentheorie.* Akademie-Verlag Berlin, 2. edition, 1989.

K. Hässig. *Graphentheoretische Methoden des Operations Research.* Teubner-Verlag, Stuttgart, 1979.

D. Jungnickel. *Graphen, Netzwerke und Algorithmen.* BI Wissenschaftsverlag, Mannheim, 3. auflage edition, 1994.

D. König. *Theorie der endlichen und unendlichen Graphen.* Akademische Verlagsgesellschaft, Leipzig, 1936. mehrfach auf deutsch und in englischer Übersetzung nachgedruckt.

S. O. Krumke and H. Noltemeier. *Graphentheoretische Konzepte und Algorithmen.* Teubner, Wiesbaden, 2005. ISBN 3-519-00526-3.

J. Matousek and B. Gärtner. *Using and Understanding Linear Programming.* Springer, 2006.

M. Padberg. *Linear Optimization and Extensions.* Springer, 2001.

H. Sachs. *Einführung in die Theorie der endlichen Graphen.* Teubner, Leipzig, 1970, und Hanser, München, 1971, 1970.

A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, 1998.

R. J. Vanderbei. *Linear Programming – Foundations and Extensions.* Springer, 2008.

K. Wagner. *Graphentheorie.* BI Wissenschaftsverlag, Mannheim, 1970.

H. Walther and G. Nägler. *Graphen, Algorithmen, Programme.* VEB Fachbuchverlag, Leipzig, 1987.

D. B. West. *Introduction to Graph Theory.* Prentice Hall, Upper Saddle River, third edition, 2005. ISBN 0-13-014400-2.

3 Diskrete Optimierungsprobleme

Dieses Kapitel enthält eine Liste von algorithmischen Fragestellungen der Graphentheorie. Wir werden — neben historisch interessanten Aufgaben — insbesondere Optimierungsprobleme aufführen, die ein weites Anwendungsspektrum besitzen.

3.1 Kombinatorische Optimierungsprobleme

Bevor wir auf graphentheoretische Optimierungsprobleme eingehen, führen wir kombinatorische Optimierungsprobleme in allgemeiner Form ein.

(3.1) Definition (Allgemeines kombinatorisches Optimierungsproblem). *Gegeben seien eine endliche Menge $\mathcal{I}$ und eine Funktion $f : \mathcal{I} \to \mathbb{R}$, die jedem Element von $\mathcal{I}$ einen "Wert" zuordnet. Gesucht ist ein Element $I^* \in \mathcal{I}$, so daß $f(I^*)$ so groß (oder klein) wie möglich ist.* △

Eine Problemformulierung dieser Art ist relativ sinnlos, da über ein Problem, das wie oben gegeben ist, kaum vernünftige mathematische Aussagen gemacht werden können. Algorithmisch ist (3.1) auf triviale Weise lösbar: man durchlaufe alle Elemente I von $\mathcal{I}$, werte die Funktion $f(I)$ aus und wähle das Element I^* mit dem größten (oder kleinsten) Wert $f(I^*)$ aus. Falls die Elemente $I \in \mathcal{I}$ algorithmisch bestimmbar und $f(I)$ auswertbar ist, hat der eben beschriebene Enumerationsalgorithmus eine sogenannte lineare Laufzeit, da jedes Element von $\mathcal{I}$ nur einmal betrachtet wird.

Die üblicherweise auftretenden kombinatorischen Optimierungsprobleme sind jedoch auf andere, wesentlich strukturiertere Weise gegeben. Die Menge $\mathcal{I}$ ist nicht durch explizite Angabe aller Elemente spezifiziert sondern implizit durch die Angabe von Eigenschaften, die die Elemente von $\mathcal{I}$ haben sollen. Ebenso ist die Funktion f nicht punktweise sondern durch "Formeln" definiert.

In dieser Vorlesung wollen wir uns hauptsächlich auf den folgenden Problemtyp konzentrieren.

(3.2) Definition (Komb. Optimierungsproblem mit linearer Zielfunktion). *Gegeben seien eine endliche Menge E (genannt* Grundmenge*), eine Teilmenge $\mathcal{I}$ der Potenzmenge 2^E von E (die Elemente von $\mathcal{I}$ heißen* zulässige Mengen *oder* zulässige Lösungen*) und eine Funktion $c : E \to \mathbb{R}$. Für jede Menge $F \subseteq E$ definieren wir ihren "Wert" durch*

$$c(F) := \sum_{e \in F} c(e),$$

und wir suchen eine Menge $I^* \in \mathcal{I}$, so dass $c(I^*)$ so groß (oder klein) wie möglich ist. △

Zur Notationsvereinfachung werden wir in Zukunft einfach *kombinatorisches Optimierungsproblem* sagen, wenn wir ein Problem des Typs (3.2) meinen. Da ein derartiges Problem durch die Grundmenge E, die zulässigen Lösungen $\mathcal{I}$ und die Zielfunktion c definiert ist, werden wir kurz von einem kombinatorischen Optimierungsproblem $(E, \mathcal{I}, c)$ sprechen.

Die Zielfunktion haben wir durch Formulierung (3.2) bereits sehr speziell strukturiert. Aber Problem (3.2) ist algorithmisch immer noch irrelevant, falls wir eine explizite Angabe von $\mathcal{I}$ unterstellen. Wir werden nachfolgend (und im Verlaufe der Vorlesung noch sehr viel mehr) Beispiele des Typs (3.2) kennenlernen. Fast alle der dort auftretenden zulässigen Mengen lassen sich auf folgende Weise charakterisieren:

$$\mathcal{I} = \{I \subseteq E \mid I \text{ hat Eigenschaft } \Pi\}.$$

Wir werden uns damit beschäftigen, welche Charakteristika die Eigenschaft Π haben muss, damit die zugehörigen Probleme $(E, \mathcal{I}, c)$ auf einfache Weise gelöst werden können. Nehmen wir an, dass E insgesamt n Elemente enthält, dann führt natürlich jede Eigenschaft Π, die impliziert, dass $\mathcal{I}$ (relativ zu n) nur sehr wenige Elemente enthält, dazu, dass $(E, \mathcal{I}, c)$ einfach lösbar ist, falls man die Elemente von $\mathcal{I}$ explizit angeben kann. Typischerweise haben jedoch die interessanten kombinatorischen Optimierungsprobleme eine Anzahl von Lösungen, die exponentiell in n ist, etwa $n!$ oder 2^n. Eine vollständige Enumeration der Elemente solcher Mengen ist offenbar auch auf den größten Rechnern (für z. B. $n \geq 40$) nicht in „vernünftiger Zeit“ durchführbar. Das Ziel der kombinatorischen Optimierung besteht — kurz und vereinfachend gesagt — darin, Algorithmen zu entwerfen, die (erheblich) schneller als die Enumeration aller Lösungen sind.

3.2 Klassische Fragestellungen der Graphentheorie

Nachfolgend werden eine Reihe von graphentheoretischen Problemen skizziert, die die Entwicklung der Graphentheorie nachhaltig beeinflusst haben.

(3.3) Euler und das Königsberger Brückenproblem. Fast jedes Buch über Graphentheorie (Geben Sie einfach einmal “Königsberg bridges” in Google ein.) enthält einen Stadtplan von Königsberg und erläutert, wie Euler die Königsberger Karte zu dem Graphen aus Abbildung 3.1 “abstrahiert” hat.

Euler hat die Frage untersucht, ob es in diesem “Königsberger Brückengraphen” einen geschlossenen Pfad gibt, der alle Kanten genau einmal enthält. Heute nennen wir einen solchen Pfad *Eulertour*. Er hat das Problem nicht nur für den Graphen aus Abbildung 3.1 gelöst, sondern für alle Graphen: Ein Graph enthält eine Eulertour genau dann, wenn er zusammenhängend ist und jeder Knoten einen geraden Grad hat. Diesen Satz hat Euler 1736 bewiesen und damit die Graphentheorie begründet. Hinweise zur Geschichte des Königsberger Brückenproblems und zu verwandten Optimierungsproblemen finden Sie u. a. in Grötschel and Yuan (2012). △

(3.4) Das Haus vom Nikolaus. Jeder kennt die Aufgabe aus dem Kindergarten: Zeichne das Haus des Nikolaus, siehe Abbildung 3.2, in einem Zug!

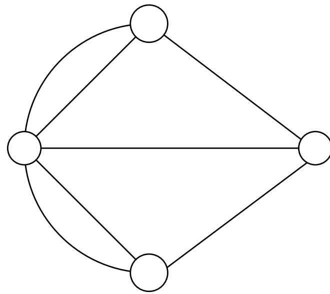


Abbildung 3.1: Das Königsberger Brückenproblem.

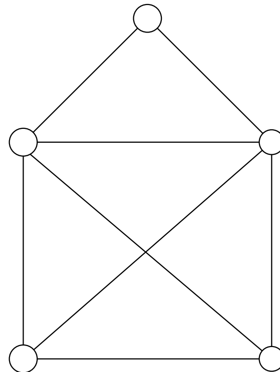


Abbildung 3.2: Das Haus vom Nikolaus.

Was hat diese Fragestellung mit dem Königsberger Brückenproblem zu tun? △

(3.5) Hamiltonsche Kreise. Der irische Mathematiker Sir William Hamilton (z.B. durch die "Erfindung" der Quaternionen bekannt) hat sich Ende der 50er Jahre des 19. Jahrhunderts mit Wege-Problemen beschäftigt und sich besonders dafür interessiert, wie man auf dem Dodekaedergraphen, siehe Abbildung 3.3, Kreise findet, die alle Knoten durchlaufen (heute *hamiltonsche Kreise* genannt) und die noch gewissen Zusatzanforderungen genügen. Er fand diese Aufgabe so spannend, dass er sie als Spiel vermarktet hat (offenbar nicht sonderlich erfolgreich). Ein Exemplar dieses Spiels mit dem Namen "The Icosian Game" befindet sich noch in der Bibliothek des Trinity College in Dublin, Irland, siehe Abbildung 3.4.

Die Aufgabe, in einem Graphen, einen Hamiltonkreis zu finden, sieht so ähnlich aus wie das Problem, eine Eulertour zu bestimmen. Sie ist aber viel schwieriger. Das hamiltonische Graphen-Problem hat sich später zum Travelling-Salesman-Problem "entwickelt". Historische Bemerkungen hierzu findet man zum Beispiel in Applegate et al. (2006) und Cook (2012). △

(3.6) Färbung von Landkarten. Nach Aigner (1984), der die Entwicklung der Graphentheorie anhand der vielfältigen Versuche, das 4-Farben-Problem zu lösen, darstellt,

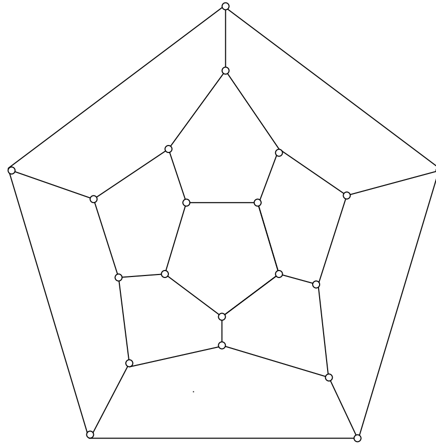


Abbildung 3.3: Graph mit einem Hamilton-Kreis.

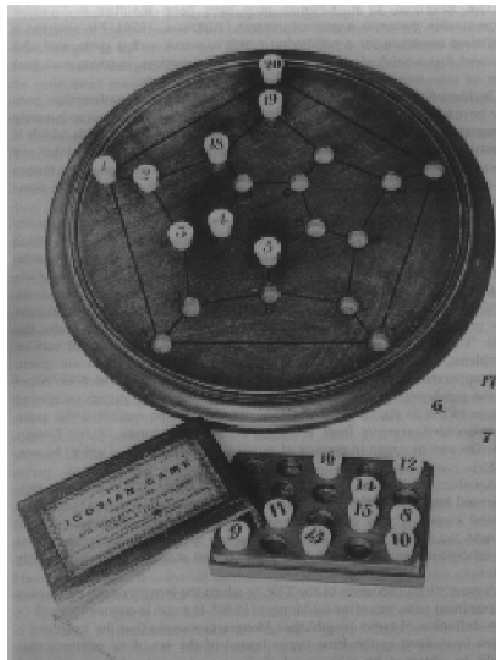


Abbildung 3.4: The Icosian Game.

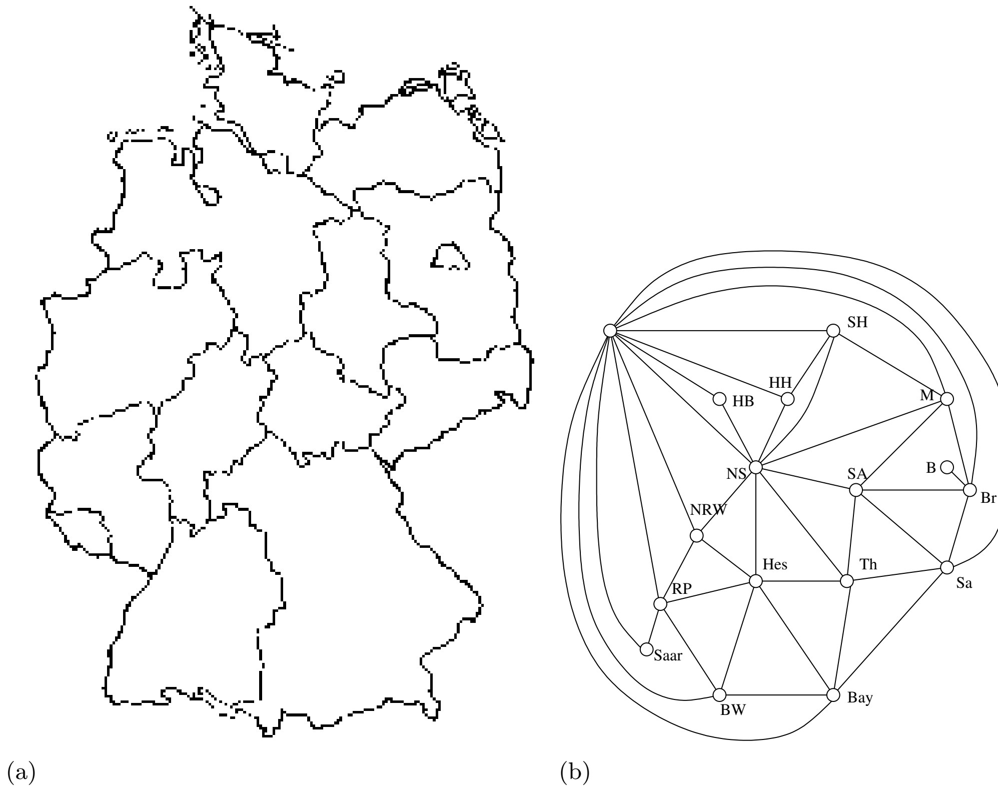


Abbildung 3.5: Abbildung (a) ist eine Karte von Deutschland mit seinen Bundesländern. In (b) repräsentiert jede Landeshauptstadt ihr zugehöriges Bundesland. Die Nachbarschaft eines jeden Knoten besteht aus den entsprechenden Nachbarländern.

begann die mathematische Beschäftigung mit dem Färbungsproblem im Jahre 1852 mit einem Brief von Augustus de Morgan an William Hamilton:

“Ein Student fragte mich heute, ob es stimmt, dass die Länder jeder Karte stets mit höchstens 4 Farben gefärbt werden können, unter der Maßgabe, dass angrenzende Länder verschiedene Farben erhalten.” Der Urheber der Frage war *Francis Guthrie*.

Aus einer Landkarte kann man einen Graphen machen, indem jedes Land durch einen Knoten repräsentiert wird und je zwei Knoten genau dann durch eine Kante verbunden werden, wenn die zugehörigen Länder benachbart sind. Abbildung 3.5 (a) zeigt die Karte der deutschen Bundesländer. Der “Bundesländergraph” in Abbildung 3.5 (b) hat daher je einen Knoten für die 16 Länder und einen weiteren Knoten für die “Außenwelt”. Dieser Knoten ist mit allen Bundesländerknoten verbunden, die an das Ausland oder das Meer (wie etwa Niedersachsen) grenzen.

“Landkartengraphen” kann man nach Konstruktion in die Ebene so zeichnen, dass sich je zwei Kanten (genauer: die Linien, die die Kanten in der Ebene repräsentieren) nicht schneiden (außer natürlich in ihren Endpunkten, wenn sie einen gemeinsamen Knoten

besitzen). Landkartengraphen sind also planar. Das 4-Farben-Problem (in etwas allgemeinerer Form) lautet dann: “Kann man die Knoten eines planaren Graphen so färben, dass je zwei benachbarte Knoten verschiedene Farben besitzen?”

Der Weg zur Lösung des 4-Farben-Problems war sehr lang, siehe hierzu Aigner (1984) . Die erste vollständige Lösung (unter Zuhilfenahme von Computerprogrammen) wurde 1976/1977 von K. Appel und W. Haken vorgelegt. Die Dokumentation eines transparenten Beweises von N. Robertson, D.P. Sanders, P. Seymour und R. Thomas, der weiterhin auf der Überprüfung vieler Einzelfälle durch Computerprogramme beruht, ist auf der Homepage von Robin Thomas zu finden:

`http://www.math.gatech.edu/~thomas/FC/fourcolor.html`. △

(3.7) Planarität. Durch das 4-Farben-Problem gelangte die Frage, wann kann man einen Graphen so in die Ebene einbetten, dass sich je zwei Kanten nicht überschneiden, in den Fokus der Forschung. Natürlich wurde sofort verallgemeinert: “Finde eine ‘gute’ Charakterisierung dafür, dass ein Graph in die Ebene, auf dem Torus, in die projektive Ebene, auf Henkelflächen etc. überschneidungsfrei einbettbar ist.”

Kuratowski gelang 1930 ein entscheidender Durchbruch. Es ist einfach zu sehen, dass weder der vollständige Graph K_5 noch der vollständige Graph $K_{3,3}$ planar sind. Kuratowski bewies, dass jeder nicht-planare Graph einen der Graphen K_5 oder $K_{3,3}$ “enthält”. Das heißt, ist G nicht planar, so kann man aus G durch Entfernen und durch Kontraktion von Kanten entweder den K_5 oder den $K_{3,3}$ erzeugen. Dies ist auch heute noch ein keineswegs triviales Ergebnis. △

3.3 Graphentheoretische Optimierungsprobleme: Einige Beispiele

In diesem Abschnitt wollen wir mehrere Beispiele von kombinatorischen Optimierungsproblemen, die sich mit Hilfe von Graphentheorie formulieren lassen, und einige ihrer Anwendungen auflisten. Diese Sammlung ist nicht im geringsten vollständig, sondern umfasst nur einige in der Literatur häufig diskutierte oder besonders anwendungsnahe Probleme. Wir benutzen dabei gelegentlich englische Namen, die mittlerweile auch im Deutschen zu Standardbezeichnungen geworden sind. Fast alle der nachfolgend aufgeführten “Probleme” bestehen aus mehreren eng miteinander verwandten Problemtypen. Wir gehen bei unserer Auflistung so vor, dass wir meistens zunächst die graphentheoretische Formulierung geben und dann einige Anwendungen skizzieren.

(3.8) Kürzeste Wege. Gegeben seien ein Digraph $D = (V, A)$ und zwei verschiedene Knoten $u, v \in V$, stelle fest, ob es einen gerichteten Weg von u nach v gibt. Falls das so ist, und falls “Entfernungen” $c_{ij} \geq 0$ für alle $(i, j) \in A$ bekannt sind, bestimme einen kürzesten gerichteten Weg von u nach v (d. h. einen (u, v)-Weg P, so dass $c(P)$ minimal ist). Dieses Problem wird üblicherweise *Problem des kürzesten Weges* (*shortest path problem*) genannt. Zwei interessante Varianten sind die folgenden: Finde einen kürzesten (u, v)-Weg gerader bzw. ungerader Länge (d. h. mit gerader bzw. ungerader Bogenzahl).

Das Problem des kürzesten Weges gibt es auch in einer ungerichteten Version. Hier sucht man in einem Graphen $G = (V, E)$ mit Entfernungen $c_e \geq 0$ für alle $e \in E$ bei gegebenen Knoten $u, v \in V$, $\neq v$, einen kürzesten $[u, v]$-Weg. Analog kann man nach einem kürzesten Weg gerader oder ungerader Länge fragen.

Natürlich kann man in allen bisher angesprochenen Problemen, das Wort "kürzester" durch "längster" ersetzen und erhält dadurch *Probleme der längsten Wege* verschiedener Arten. Hätten wir beim Problem des kürzesten Weges nicht die Beschränkung $c_{ij} \geq 0$ für die Zielfunktionskoeffizienten, wären die beiden Problemtypen offensichtlich äquivalent. Aber so sind sie es nicht! Ein Spezialfall (Zielfunktion $c_e = 1$ für alle $e \in E$) des Problems des längsten Weges ist das Problem zu entscheiden, ob ein Graph einen hamiltonschen Weg von u nach v enthält. △

Anwendungen dieses Problems und seiner Varianten sind offensichtlich. Alle Routenplaner, die im Internet zur Fahrstreckenplanung angeboten werden oder zur Unterstützung von Autofahrern in Navigationssysteme eingebaut sind, basieren auf Algorithmen zur Bestimmung kürzester Wege. Die Route jeder im Internet verschickten Nachricht wird ebenfalls durch (mehrfachen) Aufruf eines Kürzeste-Wege-Algorithmus ermittelt. Eine Anwendung aus der Wirtschafts- und Sozialgeographie, die nicht unbedingt im Gesichtsfeld von Mathematikern liegt, sei hier kurz erwähnt. Bei Fragen der Raumordnung und Landesplanung werden sehr umfangreiche Erreichbarkeitsanalysen angestellt, um Einzugsbereiche (bzgl. Straßen-, Nahverkehrs- und Bahnanbindung) festzustellen. Auf diese Weise werden Mittel- und Oberzentren des ländlichen Raumes ermittelt und Versorgungsgrade der Bevölkerung in Bezug auf Ärzte, Krankenhäuser, Schulen etc. bestimmt. Ebenso erfolgen Untersuchungen bezüglich des Arbeitsplatzangebots. Alle diese Analysen basieren auf einer genauen Ermittlung der Straßen-, Bus- und Bahnentfernungen (in Kilometern oder Zeiteinheiten) und Algorithmen zur Bestimmung kürzester Wege in den "Verbindungsnetzwerken".

(3.9) Das Zuordnungsproblem (assignment problem). Gegeben sei ein bipartiter Graph $G = (V, E)$ mit Kantengewichten $c_e \in \mathbb{R}$ für alle $e \in E$, gesucht ist ein Matching in G maximalen Gewichts. Man nennt dieses Problem das *Matchingproblem in bipartiten Graphen* oder kurz *bipartites Matchingproblem.* Haben die beiden Knotenmengen in der Bipartition von V gleiche Kardinalität und sucht man ein perfektes Matching minimalen Gewichts, so spricht man von einem *Zuordnungsproblem.* Es gibt noch eine weitere Formulierung des Zuordnungsproblems. Gegeben sei ein Digraph $D = (V, A)$, der auch Schlingen haben darf, mit Bogengewichten (meistens wird unterstellt, dass D vollständig ist und Schlingen hat), gesucht ist eine Bogenmenge minimalen Gewichts, so dass jeder Knoten von D genau einmal Anfangs- und genau einmal Endknoten eines Bogens aus B ist. (B ist also eine Menge knotendisjunkter gerichteter Kreise, so dass jeder Knoten auf genau einem Kreis liegt. Eine Schlinge wird hierbei als ein gerichteter Kreis aufgefasst.) Wir wollen dieses Problem *gerichtetes Zuordnungsproblem* nennen. △

Das Zuordnungsproblem hat folgende "Anwendung". Gegeben seien n Männer und n Frauen, für $1 \leq i, j \leq n$ sei c_{ij} ein "Antipathiekoeffizient". Gesucht ist eine Zuordnung

von Männern zu Frauen (Heirat), so dass die Summe der Antipathiekoeffizienten minimal ist. Dieses Problem wird häufig *Heiratsproblem* genannt.

Das Matchingproblem in bipartiten Graphen kann man folgendermaßen interpretieren. Ein Betrieb habe m offene Stellen und n Bewerber für diese Positionen. Durch Tests hat man herausgefunden, welche Eignung Bewerber i für die Stelle j hat. Diese "Kompetenz" sei mit c_{ij} bezeichnet. Gesucht wird eine Zuordnung von Bewerbern zu Positionen, so dass die "Gesamtkompetenz" maximal wird.

Das Zuordnungsproblem und das Matchingproblem in bipartiten Graphen sind offenbar sehr ähnlich, die Beziehungen zwischen dem Zuordnungsproblem und seiner gerichteten Version sind dagegen nicht ganz so offensichtlich. Dennoch sind diese drei Probleme in folgendem Sinne "äquivalent": man kann sie auf sehr einfache Weise ineinander transformieren, d. h. mit einem schnellen Algorithmus zur Lösung des einen Problems kann man die beiden anderen Probleme lösen, ohne komplizierte Transformationsalgorithmen einzuschalten.

Transformationstechniken, die einen Problemtyp in einen anderen überführen, sind außerordentlich wichtig und zwar sowohl aus theoretischer als auch aus praktischer Sicht. In der Theorie werden sie dazu benutzt, Probleme nach ihrem Schwierigkeitsgrad zu klassifizieren (siehe Kapitel 4), in der Praxis ermöglichen sie die Benutzung eines einzigen Algorithmus zur Lösung der verschiedensten Probleme und ersparen daher erhebliche Codierungs- und Testkosten. Anhand der drei vorgenannten Probleme wollen wir nun derartige Transformationstechniken demonstrieren.

Bipartites Matchingsproblem $\longrightarrow$ Zuordnungsproblem. Angenommen wir haben ein Matchingproblem in einem bipartiten Graphen und wollen es mit einem Algorithmus für Zuordnungsprobleme lösen. Das Matchingproblem ist gegeben durch einen bipartiten Graphen $G = (V, E)$ mit Bipartition V_1, V_2 und Kantengewichten $c_e \in \mathbb{R}$ für alle $e \in E$. O. B. d. A. können wir annehmen, dass $m = |V_1| \leq |V_2| = n$ gilt. Zur Menge V_1 fügen wir $n - m$ neue Knoten W (künstliche Knoten) hinzu. Wir setzen $V_1' := V_1 \cup W$. Für je zwei Knoten $i \in V_1'$ und $j \in V_2$, die nicht in G benachbart sind, fügen wir eine neue (künstliche) Kante ij hinzu. Die Menge der so hinzugefügten Kanten nennen wir E', und den Graphen $(V_1' \cup V_2, E \cup E')$ bezeichnen wir mit G'. G' ist der vollständige bipartite Graph $K_{n,n}$. Wir definieren neue Kantengewichte c_e' wie folgt:

$$c_e' := \begin{cases} 0 & \text{falls } e \in E' \\ 0 & \text{falls } e \in E \text{ und } c_e \leq 0 \\ -c_e & \text{falls } e \in E \text{ und } c_e > 0 \end{cases}$$

Lösen wir das Zuordnungsproblem bezüglich G' mit den Gewichten c_e', $e \in E \cup E'$, so erhalten wir ein perfektes Matching M' minimalen Gewichts bezüglich c'. Es ist nun einfach zu sehen, dass

$$M := \{e \in M' \mid c_e' < 0\}$$

ein Matching in G ist, das maximal bezüglich der Gewichtsfunktion c ist.

Zuordnungsproblem $\longrightarrow$ gerichtetes Zuordnungsproblem. Wir zeigen nun, dass man das Zuordnungsproblem mit einem Algorithmus für das gerichtete Zuordnungsproblem

lösen kann. Gegeben sei also ein bipartiter Graph $G = (V, E)$ mit Bipartition V_1, V_2 und Kantengewichten c_e. Es gelte $V_1 = \{u_1, u_2, \ldots, u_n\}$, $V_2 = \{v_1, v_2, \ldots, v_n\}$. Wir definieren einen Digraphen $D = (W, A)$ mit $W = \{w_1, \ldots, w_n\}$. Zwei Knoten $w_i, w_j \in W$ sind genau dann durch einen Bogen (w_i, w_j) verbunden, wenn $u_iv_j \in E$ gilt. Das Gewicht $c'((w_i, w_j))$ des Bogens (w_i, w_j) sei das Gewicht $c(u_iv_j)$ der Kante u_iv_j. Ist B eine minimale Lösung des gerichteten Zuordnungsproblems bezüglich D und c', so ist

$$M := \{u_iv_j \in E \mid (w_i, w_j) \in B\}$$

offenbar ein minimales perfektes Matching in G bezüglich der Gewichtsfunktion c. Es ist ebenfalls sofort klar, dass das gerichtete Zuordnungsproblem bezüglich D eine Lösung genau dann hat, wenn G ein perfektes Matching enthält.

Gerichtetes Zuordnungsproblem $\longrightarrow$ bipartites Matchingproblem. Schließlich wollen wir noch vorführen, dass man das gerichtete Zuordnungsproblem auf das Matchingproblem in bipartiten Graphen zurückführen kann. Gegeben sei also ein Digraph $D = (W, A)$ mit $W = \{w_1, \ldots, w_n\}$ und Bogengewichten $c((w_i, w_j))$ für alle $(w_i, w_j) \in A$. Wir definieren einen bipartiten Graphen $G = (V, E)$ mit Bipartition $V_1 = \{u_1, \ldots, u_n\}$, $V_2 = \{v_1, \ldots, v_n\}$ und Kantenmenge $E := \{u_iv_j \mid (w_i, w_j) \in A\}$. Es seien

$$z := n(\max\{|c((w_i, w_j))| : (w_i, w_j) \in A\}) + 1$$

und

$$c'(u_iv_j) := -c((w_i, w_j)) + z.$$

Nach Konstruktion gilt, dass jedes Matching in G mit k Kanten ein geringeres Gewicht hat als ein Matching mit $k+1$ Kanten, $k = 0, \ldots, n-1$. Daraus folgt, dass es eine Lösung des gerichteten Zuordnungproblems bezüglich D genau dann gibt, wenn jedes maximale Matching M bezüglich G und c' perfekt ist. Ist dies so, dann ist

$$B := \{(w_i, w_j) \in A \mid u_iv_j \in M\}$$

eine minimale Lösung des gerichteten Zuordnungsproblems mit Gewicht $c(B) = -c'(M) + nz$.

(3.10) Das Matchingproblem. Die Grundversion dieses Problems ist die folgende. Gegeben sei ein Graph $G = (V, E)$ mit Kantengewichten c_e für alle $e \in E$. Ist ein Matching M von G maximalen Gewichts $c(M)$ gesucht, so heißt dieses Problem *Matchingproblem*. Sucht man ein perfektes Matching minimalen Gewichts, so wird es *perfektes Matchingproblem* genannt.

Diese Probleme können wie folgt verallgemeinert werden. Gegeben seien zusätzlich nichtnegative ganze Zahlen b_v für alle $v \in V$ (genannt *Gradbeschränkungen*) und u_e für alle $e \in E$ (genannt *Kantenkapazitäten*). Ein *(perfektes) b-Matching*ist eine Zuordnung x_e von nichtnegativen ganzen Zahlen zu den Kanten $e \in E$, so dass für jeden Knoten $v \in V$ die Summe der Zahlen x_e über die Kanten $e \in E$, die mit v inzidieren, höchstens (exakt) b_v ist. Das *unkapazitierte (perfekte) b-Matchingproblem* ist die Aufgabe ein (perfektes) b-Matching $(x_e)_{e \in E}$ zu finden, so dass $\sum_{e \in E} c_e x_e$ maximal (minimal) ist.

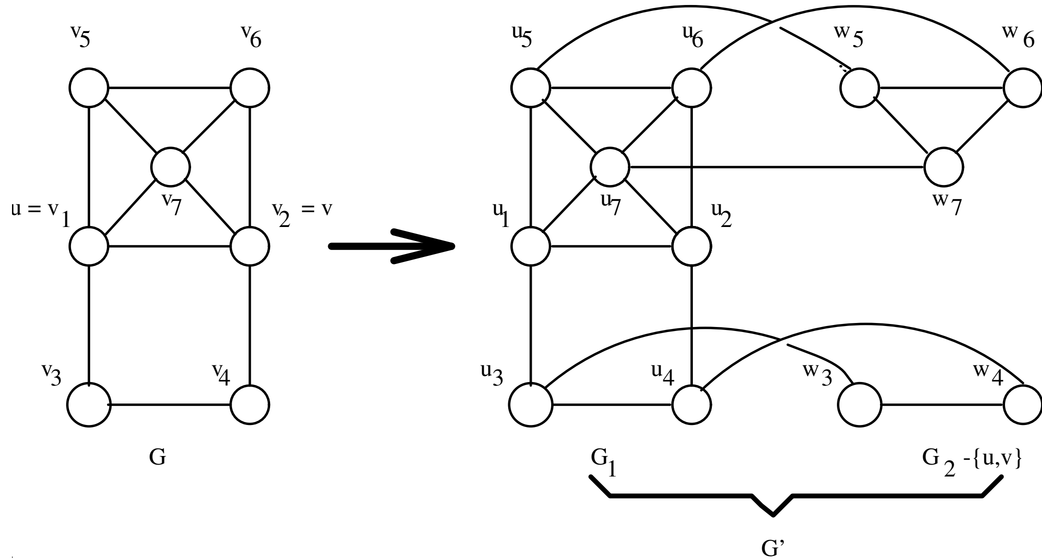


Abbildung 3.6: Die Abbildung entspricht z. B. dem ungeraden $[u, v]$-Weg (u, v_7, v_6, v) das perfekte Matching $M = \{u_1u_7, w_7w_6, u_6u_2, u_3w_3, u_4w_4, u_5w_5\}$ und umgekehrt.

Sollen die ganzzahligen Kantenwerte x_e für alle $e \in E$ zusätzlich noch die Kapazitätsschranken $0 \leq x_e \leq u_e$ erfüllen, so spricht man von einem *(perfekten) u-kapazitierten b-Matchingproblem.* $\triangle$

An dieser Stelle wollen wir noch eine – nicht offensichtliche – Problemtransformation vorführen. Und zwar wollen wir zeigen, dass die Aufgabe, in einem ungerichteten Graphen $G = (V, E)$ mit Kantengewichten $c_e \geq 0$ für alle $e \in E$ einen kürzesten Weg ungerader Länge zwischen zwei Knoten $u, v \in V$ zu bestimmen, mit einem Algorithmus für das perfekte Matchingproblem gelöst werden kann. Und zwar konstruieren wir aus G einen neuen Graphen G' wie folgt. Nehmen wir an, dass $V = \{v_1, \ldots, v_n\}$ gilt. Die Graphen $G_1 = (U, E_1)$ mit $U := \{u_1, \ldots, u_n\}$ und $G_2 = (W, E_2)$ mit $W := \{w_1, \ldots, w_n\}$ seien knotendisjunkte isomorphe Bilder (also Kopien) von G, so dass die Abbildungen $v_i \mapsto u_i$ und $v_i \mapsto w_i$, $i = 1, \ldots, n$ Isomorphismen sind. Aus G_2 entfernen wir die Bilder der Knoten u und v, dann verbinden wir die übrigen Knoten $w_i \in W$ mit ihren isomorphen Bildern $u_i \in U$ durch eine Kante u_iw_i. Diese neuen Kanten u_iw_i erhalten das Gewicht $c(u_iw_i) = 0$. Die Kanten aus G_1 und $G_2 - \{u, v\}$, die ja Bilder von Kanten aus G sind, erhalten das Gewicht ihrer Urbildkanten. Der Graph G' entsteht also aus der Vereinigung von G_1 mit $G_2 - \{u, v\}$ unter Hinzufügung der Kanten u_iw_i, siehe Abbildung 3.6. Man überlegt sich leicht, dass jedes perfekte Matching in G' einer Kantenmenge in G entspricht, die einen ungeraden $[u, v]$-Weg in G enthält und dass jedes minimale perfekte Matching in G' einen minimalen ungeraden $[u, v]$-Weg bestimmt.

Hausaufgabe. Finden Sie eine ähnliche Konstruktion, die das Problem, einen kürzesten $[u, v]$-Weg gerader Länge zu bestimmen, auf ein perfektes Matchingproblem zurückführt!

(3.11) Wälder, Bäume, Branchings, Arboreszenzen. Gegeben sei ein Graph $G = (V, E)$ mit Kantengewichten $c_e \in \mathbb{R}$ für alle $e \in E$. Die Aufgabe, einen Wald $W \subseteq E$ zu finden, so dass $c(W)$ maximal ist, heißt *Problem des maximalen Waldes.*

Die Aufgabe einen Baum $T \subseteq E$ zu finden, der G aufspannt und dessen Gewicht $c(T)$ minimal ist, heißt *Problem des minimalen aufspannenden Baumes* (*minimum spanning tree problem*). Diese beiden Probleme haben auch eine gerichtete Version.

Gegeben sei ein Digraph $D = (V, A)$ mit Bogengewichten $c_a \in \mathbb{R}$ für alle $a \in A$. Die Aufgabe, ein Branching $B \subseteq A$ maximalen Gewichts zu finden, heißt *maximales Branching-Problem*, die Aufgabe, eine Arboreszenz (mit vorgegebener Wurzel r) von D minimalen Gewichts zu finden, heißt *minimales Arboreszenz-Problem (r-Arboreszenz-Problem).* △

Die im folgenden Punkt zusammengefassten Probleme gehören zu den am meisten untersuchten und anwendungsreichsten Problemen.

(3.12) Routenplanung. Gegeben seien n Städte und Entfernungen c_{ij} zwischen diesen, gesucht ist eine Rundreise (*Tour*), die durch alle Städte genau einmal führt und minimale Länge hat. Haben die Entfernungen die Eigenschaft, dass $c_{ij} = c_{ji}$ gilt, $1 \leq i < j \leq n$, so nennt man dieses Problem *symmetrisches Travelling-Salesman-Problem (TSP)*, andernfalls heißt es *asymmetrisches TSP*. Graphentheoretisch läßt sich das TSP wie folgt formulieren. Gegeben sei ein vollständiger Graph (oder Digraph) G mit Kantengewichten (oder Bogengewichten), gesucht ist ein (gerichteter) hamiltonscher Kreis minimaler Länge. Beim TSP geht man durch jeden Knoten genau einmal, beim (gerichteten) *Chinesischen Postbotenproblem* (*Chinese postman problem*) durch jede Kante (jeden Bogen) mindestens einmal, d. h. in einem Graphen (Digraphen) mit Kantengewichten (Bogengewichten) wird eine Kette (gerichtete Kette) gesucht, die jede Kante (jeden Bogen) mindestens einmal enthält und minimale Länge hat.

Zu diesen beiden Standardproblemen gibt es hunderte von Mischungen und Varianten. Z. B., man sucht eine Kette, die durch einige vorgegebene Knoten und Kanten mindestens einmal geht und minimale Länge hat; man legt verschiedene Ausgangspunkte (oder Depots) fest, zu denen man nach einer gewissen Streckenlänge wieder zurückkehren muss, etc. Eine relativ allgemeine Formulierung ist die folgende. Gegeben ist ein gemischter Graph mit Knotenmenge V, Kantenmenge E und Bogenmenge A. Ferner sind eine Menge von Depots $W \subseteq V$, von denen aus Reisen gestartet werden müssen, eine Menge $U \subseteq V$ von Knoten, die mindestens einmal besucht werden müssen, und eine Menge $B \subseteq E \cup A$ von Kanten und Bögen, die mindestens einmal durchlaufen werden müssen. Gesucht sind geschlossene Ketten von Kanten und gleichgerichteten Bögen, so dass jede dieser Folgen mindestens (oder genau) einen der Knoten aus W enthält und die Vereinigung dieser Ketten jeden Knoten aus U und jede Kante (Bogen) aus B mindestens einmal enthält und minimale Länge hat. △

Anwendungen dieser Probleme in der Routenplanung von Lieferwagen, von Straßenkehrmaschinen, der Müllabfuhr, von Speditionen etc. sind offensichtlich. Aber auch bei der Steuerung von NC-Maschinen (zum automatischen Bohren, Löten oder Schweißen) oder der Verdrahtung von Leiterplatten (z. B. von Testbussen) tritt das TSP (oder eine

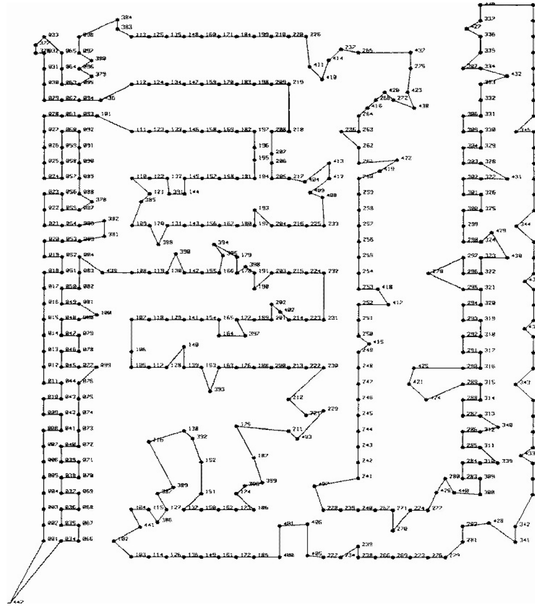


Abbildung 3.7: Optimaler Weg auf einer Leiterplatte um 441 Löcher zu bohren.

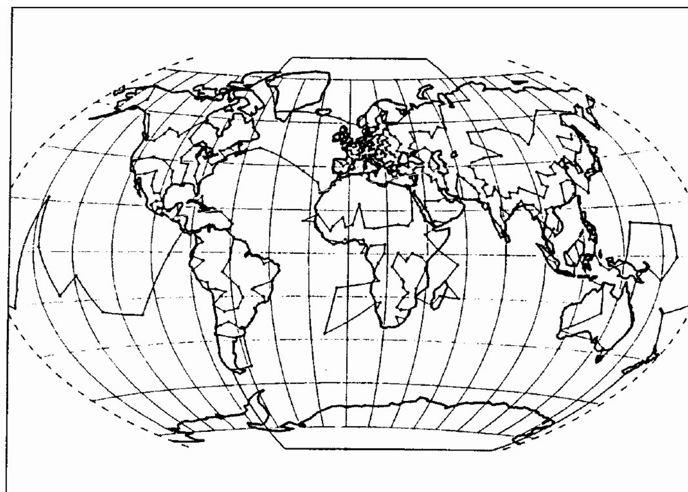


Abbildung 3.8: Optimale Tour für 666 Städte weltweit.

seiner Varianten) auf. Abbildung 3.7 zeigt eine Leiterplatte, durch die 441 Löcher gebohrt werden müssen. Links unten ist der Startpunkt, an den der Bohrkopf nach Beendigung des Arbeitsvorganges zurückkehrt, damit eine neue Platte in die Maschine eingelegt werden kann. Abbildung 3.7 zeigt eine optimale Lösung dieses 442-Städte-TSP. Die Bohrmaschine muss eine Weglänge von 50.069 Einheiten zurückzulegen.

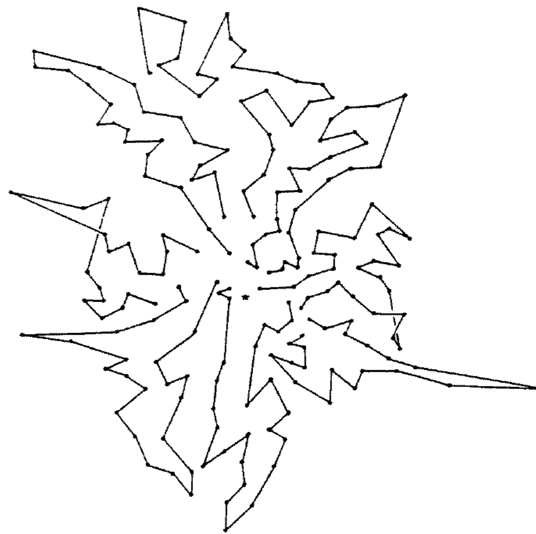


Abbildung 3.9: Optimale Routen für die Wartung der Telefonzellen in der holländischen Stadt Haarlem.

Abbildung 3.8 zeigt 666 Städte auf der Weltkugel. Wählt man die "Luftliniendistanz" (bezüglich eines Großkreises auf der Kugel) als Entfernung zwischen zwei Städten, so zeigt Abbildung 3.8 eine kürzeste Rundreise durch die 666 Orte dieser Welt. Die Länge dieser Reise ist 294 358 km lang. Abbildungen 3.7 und 3.8 sind Grötschel and Holland (1991) entnommen. Im Internet finden Sie unter der URL:
`http://www.math.princeton.edu/tsp/` interessante Informationen zum TSP sowie weitere Bilder von TSP-Beispielen. Daten zu vielen TSP-Beispielen wurden von G. Reinelt gesammelt und sind unter der folgenden URL zu finden:

`http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/`

Der beste zur Zeit verfügbare Code zur Lösung von TSPs ist zu finden unter

`http://www.tsp.gatech.edu/concorde/`

In Abbildung 3.9 sind die eingezeichneten Punkte Standorte von Telefonzellen in der holländischen Stadt Haarlem. Der Stern in der Mitte ist das Postamt. Die Aufgabe ist hier, eine Routenplanung für den sich wöchentlich wiederholenden Telefonzellenwartungsdienst zu machen. Die einzelnen Touren starten und enden am Postamt (diese Verbindungen sind nicht eingezeichnet) und führen dann so zu einer Anzahl von Telefonzellen, dass die Wartung aller Telefonzellen auf der Tour innerhalb einer Schicht durchgeführt werden kann.

Als Travelling-Salesman-Problem lassen sich auch die folgenden Anwendungsprobleme formulieren:

- Bestimmung einer optimalen Durchlaufreihenfolge der Flüssigkeiten (Chargen) in einer Mehrproduktenpipeline (Minimierung Reinigungszeiten),
- Bestimmung der optimalen Verarbeitungsfolge von Lacken in einer Großlackiererei (Minimierung Reinigungszeiten),
- Bestimmung einer Reihenfolge des Walzens von Profilen in einem Walzwerk, so dass die Umrüstzeiten der Walzstraße minimiert werden,
- Bestimmung der zeitlichen Reihenfolge von archäologischen Fundstätten (Grablegungsreihenfolge von Gräbern in einem Gräberfeld, Besiedlungsreihenfolge von Orten) aufgrund von Ähnlichkeitsmaßen (Distanzen), die durch die aufgefundenen Fundstücke definiert werden, siehe hierzu Gertzen and Grötschel (2012).

Umfangreiche Information über das TSP, seine Varianten und Anwendungen kann man in den Sammelbänden Lawler et al. (1985) und Gutin and Punnen (2002) finden.

(3.13) Stabile Mengen, Cliquen, Knotenüberdeckungen. Gegeben sei ein Graph $G = (V, E)$ mit Knotengewichten $c_v \in \mathbb{R}$ für alle $v \in V$. Das *Stabile-Mengen-Problem* ist die Aufgabe, eine stabile Menge $S \subseteq V$ zu suchen, so dass $c(S)$ maximal ist, das *Cliquenproblem* die Aufgabe, eine Clique $Q \subseteq V$ zu suchen, so dass $c(Q)$ maximal ist, und das *Knotenüberdeckungsproblem* die Aufgabe, eine Knotenüberdeckung $K \subseteq V$ zu suchen, so dass $c(K)$ minimal ist. △

Die drei oben aufgeführten Probleme sind auf triviale Weise ineinander überführbar. Ist nämlich $S \subseteq V$ eine stabile Menge in G, so ist S eine Clique im komplementären Graphen $\overline{G}$ von G und umgekehrt. Also ist das Stabile-Menge-Problem für G mit Gewichtsfunktion c nicht anders als das Cliquenproblem für $\overline{G}$ mit derselben Gewichtsfunktion und umgekehrt. Ist ferner $S \subseteq V$ eine stabile Menge in G, so ist $V \setminus S$ eine Knotenüberdeckung von G. Daraus folgt, dass zu jeder gewichtsmaximalen stabilen Menge S die zugehörige Knotenüberdeckung $V \setminus S$ gewichtsminimal ist und umgekehrt. Das Stabile-Menge-Problem, das Cliquenproblem und das Knotenüberdeckungsproblem sind also drei verschiedene Formulierungen einer Aufgabe. Anwendungen dieser Probleme finden sich z. B. in folgenden Bereichen:

- Einsatzplanung von Flugzeugbesatzungen
- Busfahrereinsatzplanung
- Tourenplanung im Behindertentransport
- Auslegung von Fließbändern
- Investitionsplanung
- Zuordnung von Wirtschaftsprüfern zu Prüffeldern
- Entwurf von optimalen fehlerkorrigierenden Codes

- Schaltkreisentwurf
- Standortplanung
- Wiedergewinnung von Information aus Datenbanken
- Versuchsplanung
- Signalübertragung.

Aber auch das folgende Schachproblem kann als Stabile-Menge-Problem formuliert werden: Bestimme die maximale Anzahl von Damen (oder Türmen, oder Pferden etc.), die auf einem $n \times n$ Schachbrett so plaziert werden können, dass keine eine andere schlägt.

(3.14) Färbungsprobleme. Gegeben sei ein Graph $G = (V, E)$. Zusätzlich seien Knotengewichte b_v für alle $v \in V$ gegeben. Die Aufgabe, eine Folge von (nicht notwendigerweise verschiedenen) stabilen Mengen $S_1, \ldots, S_t$ von G zu suchen, so dass jeder Knoten in mindestens b_v dieser stabilen Mengen enthalten und t minimal ist, heißt (gewichtetes) *Knotenfärbungsproblem* oder kurz *Färbungsproblem*. Beim (gewichteten) *Kantenfärbungsproblem* sind statt Knotengewichten Kantengewichte c_e, $e \in E$, gegeben und gesucht ist eine Folge von (nicht notwendigerweise verschiedenen) Matchings $M_1, \ldots, M_s$, so dass jede Kante in mindestens c_e dieser Matchings enthalten und s so klein wie möglich ist. △

Das geographische Färbungsproblem ist uns schon in (3.6) begegnet. Hat man eine Färbung der Länder, so dass je zwei benachbarte Länder verschieden gefärbt sind, so entspricht jede Gruppe von Ländern gleicher Farbe einer stabilen Menge in G. Hat man umgekehrt eine Zerlegung der Knotenmenge von G in stabile Mengen, so kann man jeweils die Länder, die zu den Knoten einer stabilen Menge gehören mit derselben Farbe belegen und erhält dadurch eine zulässige Landkartenfärbung. Das Landkartenfärbungsproblem ist also das Knotenfärbungsproblem des zugehörigen Graphen mit $b_v = 1$ für alle $v \in V$.

Die Aufgabe, in einer geographischen Region die Sendefrequenzen von Rundfunksendern (oder Mobilfunkantennen) so zu verteilen, dass sich die Sender gegenseitig nicht stören und alle Rundfunkteilnehmer, die für sie gedachten Programme auch empfangen können, kann man als Färbungsproblem (mit weiteren Nebenbedingungen) formulieren.

(3.15) Schnittprobleme. Gegeben sei ein Graph $G = (V, E)$ mit Kantengewichten $c_e \in \mathbb{R}$ für alle $e \in E$. Das Problem, einen Schnitt $\delta(W)$ in G zu finden mit maximalem Gewicht $c(\delta(W))$, heißt *Max-Cut-Problem*. Sind alle Kantengewichte c_e nicht-negativ, so nennt man das Problem, einen Schnitt minimalen Gewichts in G zu finden, *Min-Cut-Problem*. △

Das Min-Cut-Problem ist in der Theorie der Netzwerkflüsse sehr wichtig (siehe Kapitel 6).

Das Max-Cut-Problem hat z. B. eine interessante Anwendung in der Physik, und zwar kann man beim Studium magnetischer Eigenschaften von Spingläsern im Rahmen des Ising Modells die Aufgabe, einen Grundzustand (energieminimale Konfiguration bei 0°

K) zu bestimmen, als Max-Cut-Problem formulieren. Ich will diese Anwendung kurz skizzieren.

Ein *Spinglas* besteht aus nichtmagnetischem Material, das an einigen Stellen durch magnetische Atome "verunreinigt" ist. Man interessiert sich für die Energie des Systems und die Orientierung der magnetischen Atome (Verunreinigungen) bei 0° K, also für den so genannten (gefrorenen) Grundzustand des Spinglases. Dieser Grundzustand ist experimentell nicht herstellbar, und die Physiker haben unterschiedliche, sich z. T. widersprechende Theorien über einige Eigenschaften dieses Grundzustandes.

Mathematisch wird dieses Problem wie folgt modelliert. Jeder Verunreinigung i wird ein Vektor $S_i \in \mathbb{R}^3$ zugeordnet, der (bei einem gegebenen Bezugssytem) die Orientierung des Atomes im Raum, d. h. den magnetischen Spin, beschreibt. Zwischen zwei Verunreinigungen i, j besteht eine magnetische Interaktion, die durch

$$H_{ij} = J(r_{ij}) S_i \cdot S_j$$

beschrieben wird, wobei $J(r_{ij})$ eine Funktion ist, die vom Abstand r_{ij} der Verunreinigungen abhängt, und $S_i \cdot S_j$ das innere Produkt der Vektoren S_i, S_j ist. In der Praxis wird J (bei gewissen physikalischen Modellen) wie folgt bestimmt:

$$J(r_{ij}) := \cos(K r_{ij}) / r_{ij}^3 ,$$

wobei K eine materialabhängige Konstante ist (z. B. $K = 2.4 \times 10^8$). Die gesamte Energie einer Spinkonfiguration ist gegeben durch

$$H = -\sum J(r_{ij}) S_i \cdot S_j + \sum F \cdot S_i ,$$

wobei F ein äußeres magnetisches Feld ist. (Der Einfachheit halber nehmen wir im folgenden an $F = 0$.) Ein Zustand minimaler Energie ist also dadurch charakterisiert, dass $\sum J(r_{ij}) S_i \cdot S_j$ maximal ist.

Das hierdurch gegebene Maximierungsproblem ist mathematisch kaum behandelbar. Von Ising wurde folgende Vereinfachung vorgeschlagen. Statt jeder beliebigen räumlichen Orientierung werden jeder Verunreinigung nur zwei Orientierungen erlaubt: "Nordpol oben" oder "Nordpol unten". Die dreidimensionalen Vektoren S_i werden dann in diesem Modell durch Variable s_i mit Werten in der zweielementigen Menge $\{1, -1\}$ ersetzt. Unter Physikern besteht Übereinstimmung darüber, dass dieses Ising-Modell das wahre Verhalten gewisser Spingläsern gut widerspiegelt. Das obige Maximierungsproblem lautet dann bezüglich des Ising Modells:

$$\max\{\sum J(r_{ij}) s_i s_j \mid s_i \in \{-1, 1\}\} .$$

Nach dieser durch die Fachwissenschaftler vorgenommenen Vereinfachung ist der Schritt zum Max-Cut-Problem leicht. Wir definieren einen Graphen $G = (V, E)$, wobei jeder Knoten aus V eine Verunreinigung repräsentiert, je zwei Knoten i, j sind durch eine Kante verbunden, die das Gewicht $c_{ij} = -J(r_{ij})$ trägt. (Ist r_{ij} groß, so ist nach Definition c_{ij} sehr klein, und üblicherweise werden Kanten mit kleinen Gewichten c_{ij} gar

nicht berücksichtigt). Eine Partition von V in V_1 und V_2 entspricht einer Orientierungsfestlegung der Variablen, z. B. $V_1 := \{i \in V \mid i$ repräsentiert eine Verunreinigung mit Nordpol oben$\}$, $V_2 := \{i \in V \mid$ der Nordpol von i ist unten$\}$. Bei gegebenen Orientierungen der Atome (Partition V_1, V_2 von V) ist die Energie des Spinglaszustandes also wie folgt definiert:

$$\sum_{i\in V_1, j\in V_2} c_{ij} - \sum_{i,j\in V_1} c_{ij} - \sum_{i,j\in V_2} c_{ij}\,.$$

Der Zustand minimaler Energie kann also durch Maximierung des obigen Ausdrucks bestimmt werden. Addieren wir zu diesem Ausdruck die Konstante $C := \sum_{i,j\in V} c_{ij}$, so folgt daraus, dass der Grundzustand eines Spinglases durch die Lösung des Max-Cut Problems

$$\max\{\sum_{i\in V_1}\sum_{j\in V_2} c_{ij} \mid V_1, V_2 \text{ Partition von } V\}$$

bestimmt werden kann. Eine genauere Darstellung und die konkrete Berechnung von Grundzuständen von Spingläsern (und weitere Literaturhinweise) kann man in Barahona et al. (1988) finden. Dieses Paper beschreibt auch eine Anwendung des Max-Cut-Problems im VLSI-Design und bei der Leiterplattenherstellung: Die Lagenzuweisung von Leiterbahnen, so dass die Anzahl der Kontaktlöcher minimal ist.

(3.16) Standortprobleme. Probleme dieses Typs tauchen in der englischsprachigen Literatur z. B. unter den Namen Location oder Allocation Problems, Layout Planning, Facilities Allocation, Plant Layout Problems oder Facilities Design auf. Ihre Vielfalt ist (ähnlich wie bei (3.12)) kaum in wenigen Zeilen darstellbar. Ein (relativ allgemeiner) Standardtyp ist der folgende. Gegeben sei ein Graph (oder Digraph), dessen Knoten Städte, Wohnbezirke, Bauplätze, mögliche Fabrikationsstätten etc. repräsentieren, und dessen Kanten Verkehrsverbindungen, Straßen, Kommunikations- oder Transportmöglichkeiten etc. darstellen. Die Kanten besitzen "Gewichte", die z. B. Entfernungen etc. ausdrücken. Wo sollen ein Krankenhaus, ein Flughafen, mehrere Polizei- oder Feuerwehrstationen, Warenhäuser, Anlieferungslager, Fabrikationshallen ... errichtet werden, so dass ein "Optimalitätskriterium" erfüllt ist? Hierbei tauchen häufig Zielfunktionen auf, die nicht linear sind. Z. B. soll ein Feuerwehrdepot so stationiert werden, dass die maximale Entfernung vom Depot zu allen Wohnbezirken minimal ist; drei Auslieferungslager sollen so errichtet werden, dass jedes Lager ein Drittel der Kunden bedienen kann und die Summe der Entfernungen der Lager zu ihren Kunden minimal ist bzw. die maximale Entfernung minimal ist. △

(3.17) Lineare Anordnungen und azyklische Subdigraphen. Gegeben sei ein vollständiger Digraph $D_n = (V, A)$ mit Bogengewichten $c((i,j))$ für alle $(i,j) \in A$. Das Problem, eine lineare Reihenfolge der Knoten, sagen wir $i_1, \ldots, i_n$, zu bestimmen, so dass die Summe der Gewichte der Bögen, die konsistent mit der linearen Ordnung sind (also $\sum_{p=1}^{n-1}\sum_{q=p+1}^{n} c((i_p, i_q))$), maximal ist, heißt *Linear-Ordering-Problem.* Das *Azyklische-Subdigraphen-Problem* ist die Aufgabe, in einem Digraphen $D = (V, A)$ mit Bogengewichten eine Bogenmenge $B \subseteq A$ zu finden, die keinen gerichteten Kreis enthält

und deren Gewicht maximal ist. Beim *Feedback-Arc-Set-Problem* sucht man eine Bogenmenge minimalen Gewichts, deren Entfernung aus dem Digraphen alle gerichteten Kreise zerstört. △

Die drei in (3.17) genannten Probleme sind auf einfache Weise ineinander transformierbar. Diese Probleme haben interessante Anwendungen z. B. bei der Triangulation von Input-Output-Matrizen, der Rangbestimmung in Turniersportarten, im Marketing und der Psychologie. Weitergehende Informationen finden sich in Grötschel et al. (1984) und in Reinelt (1985). Einige konkrete Anwendungsbeispiele werden in den Übungen behandelt.

(3.18) Entwurf kostengünstiger und ausfallsicherer Telekommunikationsnetzwerke. Weltweit wurden in den letzten Jahren (und das geschieht weiterhin) die Kupferkabel, die Telefongespräche etc. übertragen, durch Glasfaserkabel ersetzt. Da Glasfaserkabel enorm hohe Übertragungskapazitäten haben, wurden anfangs die stark "vermaschten" Kupferkabelnetzwerke durch Glasfasernetzwerke mit Baumstruktur ersetzt. Diese Netzwerkstrukturen haben jedoch den Nachteil, dass beim Ausfall eines Verbindungskabels (z. B. bei Baggerarbeiten) oder eines Netzknotens (z. B. durch einen Brand) große Netzteile nicht mehr miteinander kommunizieren können. Man ist daher dazu übergegangen, Telekommunikationsnetzwerke mit höherer Ausfallsicherheit wie folgt auszulegen. Zunächst wird ein Graph $G = (V, E)$ bestimmt; hierbei repräsentiert V die Knotenpunkte, die in einem Telekommunikationsnetz verknüpft werden sollen, und E stellt die Verbindungen zwischen Knoten dar, die durch das Ziehen eines direkten (Glasfaser-) Verbindungskabels realisiert werden können. Gleichzeitig wird geschätzt, was das Legen einer direkten Kabelverbindung kostet. Anschließend wird festgelegt, welche Sicherheitsanforderungen das Netz erfüllen soll. Dies wird so gemacht, dass man für je zwei Knoten bestimmt, ob das Netz noch eine Verbindung zwischen diesen beiden Knoten besitzen soll, wenn ein, zwei, drei ... Kanten oder einige andere Knoten ausfallen. Dann wird ein Netzwerk bestimmt, also eine Teilmenge F von E, so dass alle Knoten miteinander kommunizieren können, alle Sicherheitsanforderungen erfüllt werden und die Baukosten minimal sind. Mit Hilfe dieses Modells (und zu seiner Lösung entwickelter Algorithmen) werden z. B. in den USA Glasfasernetzwerke für so genannte LATA-Netze entworfen und ausgelegt, siehe Grötschel et al. (1992) und Grötschel et al. (1995). Abbildung 3.10(a) zeigt das Netzwerk der möglichen direkten Kabelverbindungen in einer großen Stadt in den USA, Abbildung 3.10(b) zeigt eine optimale Lösung. Hierbei sind je zwei durch ein Quadrat gekennzeichnete Knoten gegen den Ausfall eines beliebigen Kabels geschützt (d. h. falls ein Kabel durchschnitten wird, gibt es noch eine (nicht notwendig direkte, sondern auch über Zwischenknoten verlaufende) Verbindung zwischen je zwei dieser Knoten, alle übrigen Knotenpaare wurden als relativ unwichtig erachtet und mussten nicht gegen Kabelausfälle geschützt werden. △

(3.19) Betrieb von Gasnetzen. Gasnetze sind ein wichtiger Teil der europäischen Gasversorgung. Durch die physikalischen Eigenschaften von Gasen sowie die zur Gasförderung eingesetzte Technik unterscheiden sich adäquate mathematische Modelle deutlich

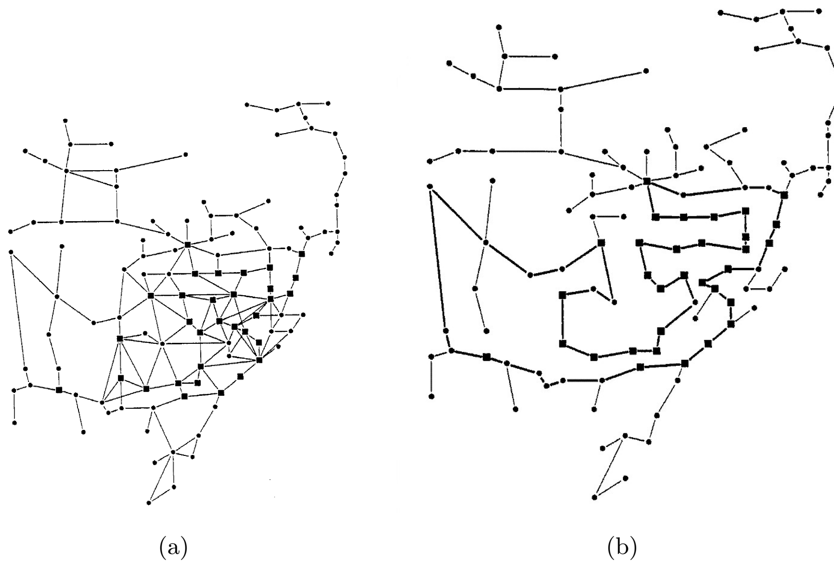


Abbildung 3.10: Abbildung (a) stellt ein Telekommunikationsnetz dar. In Abbildung (b) ist eine Optimallösung des kostengünstigsten Telekommunikationsnetzes in diesem Graphen zu sehen.

von denen für z. B. Telekommunikationsnetze. Insbesondere sind nicht nur (Gas)Flüsse relevant, sondern auch die Drücke des Gases an verschiedenen Stellen des Netzes. In Rohren verringert sich der Druck des Gases wegen der Reibung am Rohr. Daher muss in bestimmten Abständen der Druck durch sogenannte Kompressoren wieder erhöht werden. Aus technischen Gründen gibt es außerdem Druckregler, die den Druck reduzieren, sowie Ventile, um den Gasfluss zu steuern.

Ein Gasnetz kann (stark vereinfacht) modelliert werden als ein Digraph $G = (V, A)$, wobei jeder Bogen $a \in A$ ein Element (Rohr, Ventil, Druckregler, Kompressor) des Gasnetzes repräsentiert. Der Gasfluss über einen Bogen $a \in A$ sei $x_a \in \mathbb{R}$, der Druck an einem Knoten $v \in V$ sei $p_v \in \mathbb{R}_+$, und die Ein- bzw. Ausspeisung in v sei $d_v \in \mathbb{R}$. An jedem Knoten $v \in V$ gilt die *Flusserhaltungsgleichung*

$$d_v + \sum_{a \in \delta^-(v)} x_a - \sum_{a \in \delta^+(v)} x_a = 0,$$

d. h. Gas, dass in einen Knoten hineinströmt, verlässt ihn auch wieder.

Für jedes Element $a = (u, v) \in A$ gilt, je nach Elementtyp, eine Beziehung zwischen Fluss und Druck. Für ein Rohr $a = (u, v)$ gilt approximativ der Zusammenhang

$$p_v^2 - p_u^2 = \alpha_a x_a |x_a|,$$

wobei die Konstante α_a die physikalischen Eigenschaften des Rohres abbildet. Ein Regler $a = (u, v)$ kann in seiner Betriebsrichtung den Druck in gewissen Grenzen unabhängig

vom Fluss verringern, d. h. es gilt

$$p_v \geq p_u - \Delta_a$$

mit maximaler Druckverringerung Δ_a, falls $x_a > 0$ und $p_v = p_u$ sonst. Analog kann ein Verdichter den Druck in Betriebsrichtung erhöhen. Ein Ventil $a = (u, v)$ erlaubt es, einzelne Komponenten zu sperren und damit Druck und Fluss vor und hinter dem Ventil zu entkoppeln, d. h. es gilt

$$\begin{aligned} \text{Ventil } a \text{ geschlossen:} \quad & x_a = 0, \qquad p_u, p_v \text{ beliebig,} \\ \text{Ventil } a \text{ geöffnet:} \quad & x_a \text{ beliebig, } p_u = p_v. \end{aligned}$$

Eine interessante Frage ist zum Beispiel, ob der Ein- und Ausspeisevektor $d = (d_v)_{v \in V}$ im Gasnetz realisiert werden kann. Durch die (nichtlineare) Abhängigkeit des Gasflusses von den Drücken und der Möglichkeit, das Verhalten von Elementen des Netzes zwischen sehr verschiedenen Zuständen umzuschalten, ist diese Frage viel schwerer zu beantworten als für Netzwerkflussprobleme ohne Druck-Fluss-Koppelung, die wir später betrachten werden. △

Literaturverzeichnis

M. Aigner. *Graphentheorie: Eine Entwicklung aus dem 4-Farben-Problem.* Teubner Verlag, Studienbücher : Mathematik, Stuttgart, 1984. ISBN 3-519-02068-8.

D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook. *The Traveling Salesman Problem: A Computational Study.* Princeton University Press, 2006.

F. Barahona, M. Grötschel, M. Jünger, and G. Reinelt. An application of combinatorial optimization to statistical physics and circuit layout design. *Operations Research*, 36 (3):493–513, 1988.

Berge and Ghouila-Houri. *Programme, Spiele, Transportnetze.* Teubner Verlag, Leipzig, 1969.

C. Berge. *Hypergraphs, Combinatorics of finite sets*, volume 45. North-Holland Mathematical Library, Amsterdam, 1989.

B. Bollobás. *Graph Theory: An Introductory Course.* Springer Verlag, New York, 1979.

J. A. Bondy and U. S. R. Murty. *Graph Theory.* Springer, Berlin, 2008.

W. J. Cook. *In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation.* Princeton University Press, 2012.

W. Domschke. *Logistik: Rundreisen und Touren.* Oldenbourg-Verlag, München - Wien, (4., erweiterte Aufl. 1997, 1982.

J. Ebert. Effiziente Graphenalgorithmen. *Studientexte: Informatik*, 1981.

T. L. Gertzen and M. Grötschel. Flinders Petrie, the Travelling Salesman Problem, and the beginning of mathematical modeling in archaeology. *Documenta Mathematica*, pages 199–210, 2012. Elektronisch verfügbar unter `http://www.zib.de/groetschel/pubnew/paper/gertzgroe2012.pdf`.

M. C. Golombic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, New York. ISBN 1980.

R. L. Graham, M. Grötschel, and L. Lovász, editors. *Handbook of Combinatorics, Volume I.* Elsevier (North-Holland); The MIT Press, Cambridge, Massachusetts, 1995. ISBN ISBN 0-444-82346-8/v.1 (Elsevier); ISBN 0-262-07170-3/v.1 (MIT).

M. Grötschel and O. Holland. Solution of large-scale symmetric travelling salesman problems. *Mathematical Programming, Series A*, 51(2):141–202, 1991.

M. Grötschel, M. Jünger, and G. Reinelt. A Cutting Plane Algorithm for the Linear Ordering Problem. *Operations Research*, 32(6):1195–1220, 1984.

M. Grötschel, C. L. Monma, and M. Stoer. Computational Results with a Cutting Plane Algorithm for Designing Communication Networks with Low-Connectivity Constraints. *Operations Research*, 40(2):309–330, 1992.

M. Grötschel, C. L. Monma, and M. Stoer. Design of Survivable Networks. In M. O. Ball, T. L. Magnanti, C. L. Monma, and G. L. Nemhauser, editors, *Network Models*, volume 7 of *Handbooks in Operations Research and Management Science*, pages 617–672. North-Holland, 1995.

M. Grötschel and Y. Yuan. Euler, Mei-Ko Kwan, Königsberg, and a Chinese Postman. *Documenta Mathematica*, pages 43–50, 2012. Elektronisch verfügbar unter `http://www.zib.de/groetschel/pubnew/paper/groeyuan2012.pdf`.

G. Gutin and A. P. Punnen, editors. *The Traveling Salesman Problem and Its Variations.* Kluwer Academic Publishers, 2002.

R. Halin. *Graphentheorie.* Akademie-Verlag Berlin, 2. edition, 1989.

K. Hässig. *Graphentheoretische Methoden des Operations Research.* Teubner-Verlag, Stuttgart, 1979.

D. König. *Theorie der endlichen und unendlichen Graphen.* Akademische Verlagsgesellschaft, Leipzig, 1936. mehrfach auf deutsch und in englischer Übersetzung nachgedruckt.

E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization.* Wiley, Chichester, 1985.

T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout.* Teubner, Stuttgart und Wiley, Chichester, 1990.

J. K. Lenstra. *Sequencing by Enumerative Methods*. PhD thesis, Mathematisch Centrum, Amsterdam, 1976.

J. G. Oxley. *Matroid Theory*. Oxford University Press, Oxford, 1992.

G. Reinelt. *The Linear Ordering Problem: Algorithms and Applications*. Heldermann Verlag, Berlin, 1985.

H. Sachs. *Einführung in die Theorie der endlichen Graphen*. Teubner, Leipzig, 1970, und Hanser, München, 1971, 1970.

M. Stoer. Design of survivable networks. *Lecture Notes for Mathematics*, 1992.

K. Wagner. *Graphentheorie*. BI Wissenschaftsverlag, Mannheim, 1970.

H. Walther and G. Nägler. *Graphen, Algorithmen, Programme*. VEB Fachbuchverlag, Leipzig, 1987.

4 Komplexitätstheorie und Speicherung von Graphen

In diesem Kapitel führen wir einige der Grundbegriffe der Komplexitätstheorie ein, die für die algorithmische Graphentheorie und die kombinatorische Optimierung von Bedeutung sind. Wir behandeln insbesondere die Klassen $\mathcal{P}$ und $\mathcal{NP}$ und das Konzept der $\mathcal{NP}$-Vollständigkeit. Die Darstellung erfolgt auf informellem Niveau. Gute Bücher zur Einführung in die Komplexitätstheorie sind Garey and Johnson (1979), Papadimitriou (1994) und Wegener (2003), weiterführende Aspekte werden unter anderem in Wagner and Wechsung (1986) und van Leeuwen (1990) behandelt. Shmoys and Tardos (1995) ist ein guter Übersichtsartikel. Einen detaillierten Überblick über Komplexitätsklassen gibt Johnson (1990), noch mehr Komplexitätsklassen findet man unter der URL:

`http://qwiki.caltech.edu/wiki/Complexity_Zoo`.

Komplexitätstheorie kann man nicht ohne Grundkenntnisse in Kodierungstechniken betreiben. Noch wichtiger aber sind Datenstrukturen beim Entwurf effizienter Algorithmen. Diesen Themenkreis streifen wir kurz in Abschnitt 4.3. Zur Vertiefung dieses Gebietes empfehlen wir Aho et al. (1974), Cormen et al. (2001), Mehlhorn (1984), Meinel (1991), Ottmann and Widmayer (2012) und Tarjan (1983). Ein Handbuch zu Datenstrukturen mit breiten Übersichtsartikeln ist Mehta and Sahni (2004).

4.1 Probleme, Komplexitätsmaße, Laufzeiten

In der Mathematik (und nicht nur hier) kann das Wort „Problem“ sehr verschiedene Bedeutungen haben. Für unsere Zwecke benötigen wir eine (einigermaßen) präzise Definition. Ein *Problem* ist eine allgemeine Fragestellung, bei der mehrere Parameter offen gelassen sind und für die eine Lösung oder Antwort gesucht wird.

Ein Problem ist dadurch definiert, dass alle seine Parameter beschrieben werden und dass genau angegeben wird, welche Eigenschaften eine Antwort (Lösung) haben soll. Sind alle Parameter eines Problems mit expliziten Daten belegt, dann spricht man im Englischen von einem „problem instance“. Im Deutschen hat sich hierfür bisher kein Standardbegriff ausgeprägt. Es sind u. a. die Wörter Einzelfall, Fallbeispiel, Problembeispiel, Probleminstanz oder Problemausprägung gebräuchlich. Ebenso wird auch das (allgemeine) Problem manchmal als Problemtyp oder Problemklasse bezeichnet. In diesem Skript werden wir keiner starren Regel folgen. Aus dem Kontext heraus dürfte i. a. klar sein, was gemeint ist.

Das Travelling-Salesman-Problem ist in diesem Sinne ein Problem. Seine offenen Parameter sind die Anzahl der Städte und die Entfernungen zwischen diesen Städten. Eine

Entfernungstabelle in einem Autoatlas definiert ein konkretes Beispiel für das Travelling-Salesman-Problem.

Aus mathematischer Sicht kann man es sich einfach machen: Ein Problem ist die Menge aller Problembeispiele. Das Travelling-Salesman-Problem ist also die Menge aller TSP-Instanzen. Das ist natürlich nicht sonderlich tiefsinnig, vereinfacht aber die mathematische Notation.

Wir sagen, dass ein Algorithmus Problem Π *löst*, wenn er für jedes Problembeispiel $\mathcal{I} \in \Pi$ eine Lösung findet. Das Ziel des Entwurfs von Algorithmen ist natürlich, möglichst „effiziente" Verfahren zur Lösung von Problemen zu finden.

Um dieses Ziel mit Inhalt zu füllen, müssen wir den Begriff „Effizienz" meßbar machen. Mathematiker und Informatiker haben hierzu verschiedene *Komplexitätsmaße* definiert. Wir werden uns hier nur mit den beiden für uns wichtigsten Begriffen *Zeit-* und *Speicherkomplexität* beschäftigen. Hauptsächlich werden wir über Zeitkomplexität reden.

Es ist trivial, dass die Laufzeit eines Algorithmus abhängt von der „Größe" eines Problembeispiels, d. h. vom Umfang der Eingabedaten. Bevor wir also Laufzeitanalysen anstellen können, müssen wir beschreiben, wie wir unsere Problembeispiele darstellen, bzw. kodieren wollen. Allgemein kann man das durch die Angabe von *Kodierungsschemata* bewerkstelligen. Da wir uns jedoch ausschließlich mit Problemen beschäftigen, die mathematisch darstellbare Strukturen haben, reicht es für unsere Zwecke aus, Kodierungsvorschriften für die uns interessierenden Strukturen anzugeben. Natürlich gibt es für jede Struktur beliebig viele Kodierungsmöglichkeiten. Wir werden die geläufigsten benutzen und merken an, dass auf diesen oder dazu (in einem spezifizierbaren Sinn) äquivalenten Kodierungsvorschriften die gesamte derzeitige Komplexitätstheorie aufbaut.

Ganze Zahlen kodieren wir *binär*. Die binäre Darstellung einer nicht-negativen ganzen Zahl n benötigt $\lceil \log_2(|n|+1) \rceil$ Bits (oder Zellen). Hinzu kommt ein Bit für das Vorzeichen. Die *Kodierungslänge* $\langle n \rangle$ einer ganzen Zahl n ist die Anzahl der zu ihrer Darstellung notwendigen Bits, d. h.

$$\langle n \rangle := \lceil \log_2(|n| + 1) \rceil + 1. \tag{4.1}$$

Jede rationale Zahl r hat eine Darstellung $r = \frac{p}{q}$ mit $p, q \in \mathbb{Z}$, p und q teilerfremd und $q > 0$. Wir nehmen an, dass jede rationale Zahl so dargestellt ist, und können daher sagen, dass die Kodierungslänge von $r = \frac{p}{q}$ gegeben ist durch

$$\langle r \rangle := \langle p \rangle + \langle q \rangle.$$

Wir werden im Weiteren auch sagen, dass wir eine ganze oder rationale Zahl r in einem *Speicherplatz* (oder *Register*) speichern, und wir gehen davon aus, dass der Speicherplatz für r die benötigten $\langle r \rangle$ Zellen besitzt.

Die Kodierungslänge eines Vektors $x = (x_1, \ldots, x_n)^T \in \mathbb{Q}^n$ ist

$$\langle x \rangle := \sum_{i=1}^{n} \langle x_i \rangle,$$

und die Kodierungslänge einer Matrix $A \in \mathbb{Q}^{(m,n)}$ ist

$$\langle A \rangle := \sum_{i=1}^{m} \sum_{j=1}^{n} \langle a_{ij} \rangle.$$

Für diese Vorlesung besonders wichtig sind die Datenstrukturen zur Kodierung von Graphen und Digraphen. Auf diese werden wir in Abschnitt 4.3 genauer eingehen.

Sind alle Kodierungsvorschriften festgelegt, so müssen wir ein *Rechnermodell* entwerfen, auf dem unsere Speicher- und Laufzeitberechnungen durchgeführt werden sollen. In der Komplexitätstheorie benutzt man hierzu i. a. *Turing-Maschinen* oder *RAM-Maschinen*. Wir wollen auf diese Rechnermodelle nicht genauer eingehen. Wir nehmen an, dass der Leser weiß, was Computer sind und wie sie funktionieren, und unterstellen einfach, dass jeder eine naive Vorstellung von einer „vernünftigen“ Rechenmaschine hat. Dies reicht für unsere Zwecke aus.

Wir stellen uns den Ablauf eines Algorithmus A (der in Form eines Rechnerprogramms vorliegt) auf einer Rechenanlage wie folgt vor: Der Algorithmus soll Problembeispiele $\mathcal{I}$ des Problems Π lösen. Alle Problembeispiele liegen in kodierter Form vor. Die Anzahl der Zellen, die notwendig sind, um $\mathcal{I}$ vollständig anzugeben, nennen wir die *Kodierungslänge* oder *Inputlänge* $\langle\mathcal{I}\rangle$ von $\mathcal{I}$. Der Algorithmus liest diese Daten und beginnt dann Operationen auszuführen, d. h. Zahlen zu berechnen, zu speichern, zu löschen, usw. Die Anzahl der Zellen, die während der Ausführung des Algorithmus A mindestens einmal benutzt wurden, nennen wir den *Speicherbedarf von A zur Lösung von $\mathcal{I}$* . Üblicherweise schätzt man den Speicherbedarf eines Algorithmus A dadurch nach oben ab, daß man die Anzahl der von A benutzten Speicherplätze bestimmt und diesen Wert mit der größten Anzahl von Zellen multipliziert, die einer der Speicherplätze beim Ablauf des Algorithmus benötigte.

Die *Laufzeit von A zur Lösung von $\mathcal{I}$* ist (etwas salopp gesagt) die Anzahl der elementaren Operationen, die A bis zur Beendigung des Verfahrens ausgeführt hat. Dabei wollen wir als *elementare Operationen* zählen:

Lesen, Schreiben und Löschen,

Addieren, Subtrahieren, Multiplizieren, Dividieren und Vergleichen

von rationalen (oder ganzen) Zahlen. Da ja zur Darstellung derartiger Zahlen mehrere Zellen benötigt werden, muss zur genauen Berechnung der Laufzeit jede elementare Operation mit den Kodierungslängen der involvierten Zahlen multipliziert werden. Die *Laufzeit von A zur Lösung von $\mathcal{I}$* ist die Anzahl der elementaren Rechenoperationen, die A ausgeführt hat, um eine Lösung von $\mathcal{I}$ zu finden, multipliziert mit der Kodierungslänge der bezüglich der Kodierungslänge größten ganzen oder rationalen Zahl, die während der Ausführung des Algorithmus aufgetreten ist. Wir tun also so, als hätten wir alle elementaren Operationen mit der in diesem Sinne größten Zahl ausgeführt und erhalten somit sicherlich eine Abschätzung der „echten“ Laufzeit nach oben.

(4.2) Definition. *Sei A ein Algorithmus zur Lösung eines Problems Π.*

(a) *Die Funktion $f_A : \mathbb{N} \longrightarrow \mathbb{N}$, definiert durch*

$$f_A(n) := \max\{\textit{Laufzeit von } A \textit{ zur Lösung von } \mathcal{I} \mid \mathcal{I} \in \Pi \textit{ und } \langle\mathcal{I}\rangle \leq n\},$$

heißt Laufzeitfunktion *von A.*

(b) *Die Funktion* $s_A : \mathbb{N} \longrightarrow \mathbb{N}$, *definiert durch*

$$s_A(n) := \max\{\textit{Speicherbedarf von } A \textit{ zur Lösung von } \mathcal{I} \mid \mathcal{I} \in \Pi \textit{und } \langle \mathcal{I} \rangle \leq n\},$$

heißt Speicherplatzfunktion *von A.*

(c) *Der Algorithmus A hat eine* polynomiale Laufzeit *(kurz: A ist ein* polynomialer Algorithmus*), wenn es ein Polynom* $p : \mathbb{N} \longrightarrow \mathbb{N}$ *gibt mit*

$$f_A(n) \leq p(n) \quad \textit{für alle } n \in \mathbb{N}.$$

Wir sagen f_A *ist von der Ordnung höchstens* n^k *(geschrieben* $f_A = O(n^k)$*), falls das Polynom p den Grad k hat.*

(d) *Der Algorithmus A hat* polynomialen Speicherplatzbedarf*, falls es ein Polynom* $q : \mathbb{N} \longrightarrow \mathbb{N}$ *gibt mit* $s_A(n) \leq q(n)$ *für alle* $n \in \mathbb{N}$. △

Wir werden uns in der Vorlesung hauptsächlich mit Problemen beschäftigen, für die polynomiale Algorithmen existieren. Wir werden aber auch Probleme behandeln, die in einem noch zu präzisierenden Sinne „schwieriger“ sind und für die (bisher) noch keine polynomialen Verfahren gefunden worden sind.

Eine triviale Bemerkung sei hier gemacht. Ein Algorithmus, dessen Speicherplatzfunktion nicht durch ein Polynom beschränkt werden kann, kann keine polynomiale Laufzeit haben, da nach Definition jede Benutzung eines Speicherplatzes in die Berechnung der Laufzeitfunktion eingeht.

Hausaufgabe. Bestimmen Sie die Laufzeitfunktion und die Speicherplatzfunktion des folgenden Algorithmus:

Eingabe: ganze Zahl n
1: $k := \langle n \rangle$
2: **for** $i = 1, \ldots, k$ **do**
3: $\quad n := n \cdot n \cdot n$
4: **end for**
5: Gib n aus.

4.2 Die Klassen $\mathcal{P}$ und $\mathcal{NP}$, $\mathcal{NP}$-Vollständigkeit

Wir wollen nun einige weitere Begriffe einführen, um zwischen „einfachen“ und „schwierigen“ Problemen unterscheiden zu können. Wir werden dabei zunächst – aus technischen Gründen – nur Entscheidungsprobleme behandeln und später die Konzepte auf (die uns eigentlich interessierenden) Optimierungsprobleme erweitern. Ein *Entscheidungsproblem* ist ein Problem, das nur zwei mögliche Antworten besitzt, nämlich „ja“ oder „nein“. Die Fragen „Enthält ein Graph einen Kreis?“, „Enthält ein Graph einen hamiltonschen Kreis?“, „Ist die Zahl n eine Primzahl?“ sind z. B. Entscheidungsprobleme. Da wir uns nicht mit (für uns unwichtigen) Feinheiten der Komplexitätstheorie beschäftigen wollen, werden wir

im weiteren nur solche Entscheidungsprobleme betrachten, für die Lösungsalgorithmen mit endlicher Laufzeitfunktion existieren.

Die Klasse aller derjenigen Entscheidungsprobleme, für die ein polynomialer Lösungsalgorithmus existiert, wird mit $\mathcal{P}$ bezeichnet. Diese Definition ist recht informell. Wenn wir genauer wären, müßten wir $\mathcal{P}$ relativ zu einem Kodierungsschema und zu einem Rechnermodell definieren. Die Definition würde dann etwa wie folgt lauten. Gegeben sei ein Kodierungsschema E und ein Rechnermodell M, Π sei ein Entscheidungsproblem, wobei jedes Problembeispiel aus Π durch das Kodierungsschema E kodiert werden kann. Π gehört zur Klasse $\mathcal{P}$ (bezüglich E und M), wenn es einen auf M implementierbaren Algorithmus zur Lösung der Problembeispiele aus Π gibt, dessen Laufzeitfunktion auf M polynomial ist. Wir wollen im weiteren derartig komplizierte und unübersichtliche Definitionen vermeiden und werden auf dem (bisherigen) informellen Niveau bleiben in der Annahme, die wesentlichen Anliegen ausreichend klar machen zu können.

Wir werden im Abschnitt 4.3 sehen, dass das Problem „Enthält ein Graph einen Kreis?" zur Klasse $\mathcal{P}$ gehört. Aber trotz enormer Anstrengungen sehr vieler Forscher ist es noch nicht gelungen, das Problem „Enthält ein Graph einen hamiltonschen Kreis" in polynomialer Zeit zu lösen.

Diese Frage ist „offenbar" schwieriger. Um zwischen den Problemen in $\mathcal{P}$ und den „schwierigeren" formal unterscheiden zu können, sind die folgenden Begriffe geprägt worden.

Wir sagen – zunächst einmal informell –, dass ein Entscheidungsproblem Π zur Klasse $\mathcal{NP}$ gehört, wenn es die folgende Eigenschaft hat: Ist die Antwort für ein Problembeispiel $\mathcal{I} \in \Pi$ „ja", dann kann die Korrektheit der Antwort in polynomialer Zeit überprüft werden.

Bevor wir etwas genauer werden, betrachten wir ein Beispiel. Wir wollen herausfinden, ob ein Graph einen hamiltonschen Kreis enthält. Jeden Graphen können wir (im Prinzip) auf ein Blatt Papier zeichnen. Hat der gegebene Graph einen hamiltonschen Kreis, und hat jemand (irgendwie) einen solchen gefunden und alle Kanten des Kreises rot angestrichen, dann können wir auf einfache Weise überprüfen, ob die rot angemalten Kanten tatsächlich einen hamiltonschen Kreis darstellen. Bilden sie einen solchen Kreis, so haben wir die Korrektheit der „ja"-Antwort in polynomialer Zeit verifiziert.

Nun die ausführliche Definition:

(4.3) Definition. *Ein Entscheidungsproblem* Π *gehört zur* Klasse $\mathcal{NP}$, *wenn es die folgenden Eigenschaften hat:*

1. *Für jedes Problembeispiel $\mathcal{I} \in \Pi$, für das die Antwort „ja" lautet, gibt es mindestens ein Objekt Q, mit dessen Hilfe die Korrektheit der „ja"-Antwort überprüft werden kann.*

2. *Es gibt einen Algorithmus, der Problembeispiele $\mathcal{I} \in \Pi$ und Zusatzobjekte Q als Input akzeptiert und der in einer Laufzeit, die polynomial in $\langle \mathcal{I} \rangle$ ist, überprüft, ob Q ein Objekt ist, aufgrund dessen Existenz eine „ja"-Antwort für $\mathcal{I}$ gegeben werden muss.* $\triangle$

Die Probleme „Hat ein Graph G einen Kreis?", „Hat ein Graph G einen hamiltonschen Kreis?" sind somit in $\mathcal{NP}$. Hat nämlich G einen Kreis oder hamiltonschen Kreis, so wählen wir diesen als Objekt Q. Dann entwerfen wir einen polynomialen Algorithmus, der für einen Graphen G und eine zusätzliche Kantenmenge Q entscheidet, ob Q ein Kreis oder hamiltonscher Kreis von G ist. Auch die Frage „Ist $n \in \mathbb{N}$ eine zusammengesetzte Zahl?" ist in $\mathcal{NP}$, denn liefern wir als „Objekt" zwei Zahlen $\neq 1$, deren Produkt n ist, so ist n keine Primzahl. Die Überprüfung der Korrektheit besteht somit in diesem Fall aus einer einzigen Multiplikation.

Die obige Definition der Klasse $\mathcal{NP}$ enthält einige Feinheiten, auf die ich ausdrücklich hinweisen möchte.

- Es wird nichts darüber gesagt, wie das Zusatzobjekt Q zu finden ist. Es wird lediglich postuliert, dass es existiert, aber nicht, dass man es z. B. mit einem polynomialen Algorithmus finden kann.

- Die Laufzeit des Algorithmus 2 in 4.3 ist nach Definition polynomial in $\langle \mathcal{I} \rangle$. Da der Algorithmus Q lesen muss, folgt daraus, dass die Kodierungslänge von Q durch ein Polynom in der Kodierungslänge von $\mathcal{I}$ beschränkt sein muss. Auf die Frage „Hat die Gleichung $x^2 + y^2 = n^2$ eine Lösung x, y?" ist „$x = n$ und $y = 0$" ein geeignetes Zusatzobjekt Q, aber weder „$x = y = n\sqrt{0.5}$" ($\sqrt{0.5}$ kann nicht endlich binär kodiert werden) noch „$x = \frac{n2^{2^n}}{2^{2^n}}, y = 0$" (die Kodierungslänge von x ist exponentiell in der Inputlänge des Problems) wären als Zusatzobjekt Q geeignet, um die Korrektheit der „ja"-Antwort in polynomialer Zeit verifizieren zu können.

- Ferner ist die Definition von $\mathcal{NP}$ unsymmetrisch in „ja" und „nein". Die Definition impliziert nicht, dass wir auch für die Problembeispiele mit „nein"-Antworten Objekte Q und polynomiale Algorithmen mit den in 4.3 spezifizierten Eigenschaften finden können.

Wir sagen, dass die Entscheidungsprobleme, die Negationen von Problemen aus der Klasse $\mathcal{NP}$ sind, zur *Klasse* co$-\mathcal{NP}$ gehören. Zu co$-\mathcal{NP}$ gehören folglich die Probleme „Hat G keinen Kreis?", „Hat G keinen hamiltonschen Kreis?", „Ist $n \in \mathbb{N}$ eine Primzahl?". Es ist bekannt, dass das erste und das letzte dieser drei Probleme ebenfalls zu $\mathcal{NP}$ gehören. Diese beiden Probleme gehören also zu $\mathcal{NP} \cap$ co$-\mathcal{NP}$. Vom Problem „Hat G keinen hamiltonschen Kreis?" weiß man nicht, ob es zu $\mathcal{NP}$ gehört. Niemand hat bisher Objekte Q und einen Algorithmus finden können, die den Forderungen 1 und 2 aus 4.3 genügen.

Das Symbol $\mathcal{NP}$ ist abgeleitet vom Begriff „nichtdeterministischer polynomialer Algorithmus". Dies sind – grob gesagt – Algorithmen, die am Anfang „raten" können, also einen nichtdeterministischen Schritt ausführen können und dann wie übliche Algorithmen ablaufen. Ein nichtdeterministischer Algorithmus „löst" z. B. das hamiltonsche Graphenproblem wie folgt: Am Anfang rät er einen hamiltonschen Kreis. Gibt es keinen, so hört das Verfahren auf. Gibt es einen, so überprüft er, ob das geratene Objekt tatsächlich ein hamiltonscher Kreis ist. Ist das so, so antwortet er mit „ja".

Trivialerweise gilt $\mathcal{P} \subseteq \mathcal{NP}$, da für Probleme in $\mathcal{P}$ Algorithmen existieren, die ohne Zusatzobjekte Q in polynomialer Zeit eine „ja"- oder „nein"-Antwort liefern. Also gilt

auch $\mathcal{P} \subseteq$ co$-\mathcal{NP}$. Eigentlich sollte man meinen, dass Algorithmen, die raten können, mächtiger sind als übliche Algorithmen. Trotz gewaltiger Forschungsanstrengungen seit den 70er Jahren ist die Frage, ob $\mathcal{P} = \mathcal{NP}$ gilt oder nicht, immer noch ungelöst. Meiner Meinung nach ist dieses Problem eines der wichtigsten offenen Probleme der heutigen Mathematik und Informatik. Das Clay Mathematics Institute hat im Jahr 2000 einen Preis von 1 Mio US$ für die Lösung des $\mathcal{P} = \mathcal{NP}$-Problems ausgesetzt, siehe: `http://www.claymath.org/millenium`. Jeder, der sich mit diesem Problem beschäftigt hat, glaubt, dass $\mathcal{P} \neq \mathcal{NP}$ gilt. (Eine für die allgemeine Leserschaft geschriebene Diskussion dieser Frage ist in Grötschel (2002) zu finden.) Könnte diese Vermutung bestätigt werden, so würde das – wie wir gleich sehen werden – bedeuten, dass für eine sehr große Zahl praxisrelevanter Probleme niemals wirklich effiziente Lösungsalgorithmen gefunden werden können. Wir werden uns also mit der effizienten Auffindung suboptimaler Lösungen zufrieden geben und daher auf den Entwurf von Heuristiken konzentrieren müssen. Deswegen wird auch im weiteren Verlauf des Vorlesungszyklus viel Wert auf die Untersuchung und Analyse von Heuristiken gelegt.

Wir haben gesehen, dass $\mathcal{P} \subseteq \mathcal{NP} \cap$ co$-\mathcal{NP}$ gilt. Auch bezüglich der Verhältnisse dieser drei Klassen zueinander gibt es einige offene Fragen.

$$\begin{aligned} &\text{Gilt} \quad \mathcal{P} = \mathcal{NP} \cap \text{co}-\mathcal{NP}? \\ &\text{Gilt} \quad \mathcal{NP} = \text{co}-\mathcal{NP}? \end{aligned}$$

Aus $\mathcal{NP} \neq$ co$-\mathcal{NP}$ würde $\mathcal{P} \neq \mathcal{NP}$ folgen, da offenbar $\mathcal{P} =$ co$-\mathcal{P}$ gilt.

Die Klassenzugehörigkeit des oben erwähnten und bereits von den Griechen untersuchten Primzahlproblems war lange Zeit offen. Dass das Primzahlproblem in co$-\mathcal{P}$ ist, haben wir oben gezeigt. Rivest gelang es 1977 zu zeigen, dass das Primzahlproblem auch in $\mathcal{NP}$ ist. Beginnend mit dem Sieb des Erathostenes sind sehr viele Testprogramme entwickelt worden. Erst 2002 gelang es drei Indern, einen polynomialen Algorithmus zu entwickeln, der in polynomialer Zeit herausfindet, ob eine ganze Zahl eine Primzahl ist oder nicht, siehe Agrawal et al. (2004) oder URL:

`http://www.cse.iitk.ac.in/users/manindra/primality.ps`

Wir wollen nun innnerhalb der Klasse $\mathcal{NP}$ eine Klasse von besonders schwierigen Problemen auszeichnen.

(4.4) Definition. *Gegeben seien zwei Entscheidungsprobleme* Π *und* Π'. *Eine* polynomiale Transformation *von* Π *in* Π' *ist ein polynomialer Algorithmus, der, gegeben ein (kodiertes) Problembeispiel* $\mathcal{I} \in \Pi$, *ein (kodiertes) Problembeispiel* $\mathcal{I}' \in \Pi'$ *produziert, so dass folgendes gilt:*

Die Anwort auf $\mathcal{I}$ *ist genau dann „ja", wenn die Antwort auf* $\mathcal{I}'$ *„ja" ist.* $\triangle$

Offenbar gilt Folgendes: Ist Π in Π' polynomial transformierbar und gibt es einen polynomialen Algorithmus zur Lösung von Π', dann kann man auch Π in polynomialer Zeit lösen. Man transformiert einfach jedes Problembeispiel aus Π in eine Problembeispiel

aus Π' und wendet den Algorithmus für Π' an. Da sowohl der Transformationsalgorithmus als auch der Lösungsalgorithmus polynomial sind, hat auch die Kombination beider Algorithmen eine polynomiale Laufzeit.

Nun kommen wir zu einem der wichtigsten Begriffe dieser Theorie, der spezifiziert, welches die schwierigsten Probleme in der Klasse $\mathcal{NP}$ sind.

(4.5) Definition. *Ein Entscheidungsproblem* Π *heißt* $\mathcal{NP}$-vollständig*, falls* $\Pi \in \mathcal{NP}$ *und falls jedes andere Problem aus* $\mathcal{NP}$ *polynomial in* Π *transformiert werden kann.* △

Jedes $\mathcal{NP}$-vollständige Entscheidungsproblem Π hat also die folgende Eigenschaft. Falls Π in polynomialer Zeit gelöst werden kann, dann kann auch jedes andere Problem aus $\mathcal{NP}$ in polynomialer Zeit gelöst werden; in Formeln:

$$\Pi\ \mathcal{NP}\text{-vollständig und } \Pi \in \mathcal{P} \Longrightarrow \mathcal{P} = \mathcal{NP}.$$

Diese Eigenschaft zeigt, dass – bezüglich polynomialer Lösbarkeit – kein Problem in $\mathcal{NP}$ schwieriger ist als ein $\mathcal{NP}$-vollständiges. Natürlich stellt sich sofort die Frage, ob es überhaupt $\mathcal{NP}$-vollständige Probleme gibt. Dies hat Cook (1971) in einer für die Komplextiätstheorie fundamentalen Arbeit bewiesen. In der Tat sind (leider) fast alle praxisrelevanten Probleme $\mathcal{NP}$-vollständig. Ein Beispiel: Das hamiltonsche Graphenproblem „Enthält G einen hamiltonschen Kreis?“ ist $\mathcal{NP}$-vollständig.

Wir wollen nun Optimierungsprobleme in unsere Betrachtungen einbeziehen. Aus jedem Optimierungsproblem kann man wie folgt ein Entscheidungsproblem machen. Ist Π ein Maximierungsproblem (Minimierungsproblem), so legt man zusätzlich zu jedem Problembeispiel $\mathcal{I}$ noch eine Schranke, sagen wir B, fest und fragt:

Gibt es für $\mathcal{I}$ eine Lösung, deren Wert nicht kleiner (nicht größer) als B ist?

Aus dem Travelling-Salesman-Problem wird auf diese Weise ein Entscheidungsproblem, man fragt einfach: „Enthält das Problembeispiel eine Rundreise, deren Länge nicht größer als B ist?“.

Wir nennen ein Optimierungsproblem $\mathcal{NP}$*-schwer*, wenn das (wie oben angegebene) zugeordnete Entscheidungsproblem $\mathcal{NP}$-vollständig ist. Diese Bezeichnung beinhaltet die Aussage, dass alle $\mathcal{NP}$-schweren Optimierungsprobleme mindestens so schwierig sind wie die $\mathcal{NP}$-vollständigen Probleme. Könnten wir nämlich ein $\mathcal{NP}$-schweres Problem (wie das Travelling-Salesman-Problem) in polynomialer Zeit lösen, dann könnten wir auch das zugehörige Entscheidungsproblem in polynomialer Zeit lösen. Wir berechnen den Wert w einer Optimallösung und vergleichen ihn mit B. Ist bei einem Maximerungsproblem (Minimierungsproblem) $w \geq B$ ($w \leq B$), so antworten wir „ja“, andernfalls „nein“.

Häufig kann man Entscheidungsprobleme dazu benutzen, um Optimierungsprobleme zu lösen. Betrachten wir als Beispiel das TSP-Entscheidungsproblem und nehmen wir an, dass alle Problembeispiele durch ganzzahlige Entfernungen zwischen den Städten gegeben sind. Ist s die kleinste vorkommende Zahl, so ziehen wir von allen Entfernungen den Wert s ab. Damit hat dann die kürzeste Entfernung den Wert 0. Ist nun t die längste aller (so modifizierten) Entfernungen, so ist offensichtlich, dass jede Tour eine nichtnegative Länge hat und, da jede Tour n Kanten enthält, ist keine Tourlänge größer als $n \cdot t$.

Wir fragen nun den Algorithmus zur Lösung des TSP-Entscheidungsproblems, ob es eine Rundreise gibt, deren Länge nicht größer als $\frac{nt}{2}$ ist. Ist das so, fragen wir, ob es eine Rundreise gibt, deren Länge höchstens $\frac{nt}{4}$ ist, andernfalls fragen wir, ob es eine Rundreise gibt mit Länge höchstens $\frac{3nt}{4}$. Wir fahren auf diese Weise fort, bis wir das Intervall ganzer Zahlen, die als mögliche Länge einer kürzesten Rundreise in Frage kommen, auf eine einzige Zahl reduziert haben. Diese Zahl muss dann die Länge der kürzesten Rundreise sein. Insgesamt haben wir zur Lösung des Optimierungsproblems das zugehörige TSP-Entscheidungsproblem ($\lceil \log_2(nt) \rceil + 1$)-mal aufgerufen, also eine polynomiale Anzahl von Aufrufen eines Algorithmus vorgenommen. Dieser Algorithmus findet also in polynomialer Zeit die Länge einer kürzesten Rundreise – bei einem gegebenen polynomialen Algorithmus für das zugehörige Entscheidungsproblem. (Überlegen Sie sich, ob — und gegebenenfalls wie — man eine kürzeste Rundreise finden kann, wenn man ihre Länge kennt!)

Dem aufmerksamen Leser wird nicht entgangen sein, dass die oben beschriebene Methode zur Reduktion des Travelling-Salesman-Problems auf das zugehörige Entscheidungsproblem nichts anderes ist als das bekannte Verfahren der *binären Suche*.

Mit diesem oder ähnlichen „Tricks“ lassen sich viele Optimierungsprobleme durch mehrfachen Aufruf von Algorithmen für Entscheidungsprobleme lösen. Wir nennen ein Optimerungsproblem Π *$\mathcal{NP}$-leicht*, falls es ein Entscheidungsproblem Π' in $\mathcal{NP}$ gibt, so dass Π durch polynomial viele Aufrufe eines Algorithmus zur Lösung von Π' gelöst werden kann. $\mathcal{NP}$-leichte Probleme sind also nicht schwerer als die Probleme in $\mathcal{NP}$. Unser Beispiel oben zeigt, dass das TSP auch $\mathcal{NP}$-leicht ist.

Wir nennen ein Optimierungsproblem *$\mathcal{NP}$-äquivalent*, wenn es sowohl $\mathcal{NP}$-leicht als auch $\mathcal{NP}$-schwer ist. Diese Bezeichnung ist im folgenden Sinne gerechtfertigt. Ein $\mathcal{NP}$-äquivalentes Problem ist genau dann in polynomialer Zeit lösbar, wenn $\mathcal{P} = \mathcal{NP}$ gilt. Wenn jemand einen polynomialen Algorithmus für das TSP findet, hat er damit gleichzeitig $\mathcal{P} = \mathcal{NP}$ bewiesen.

Wir wollen im Weiteren dem allgemein üblichen Brauch folgen und die feinen Unterschiede zwischen den oben eingeführten Bezeichnungen für Entscheidungs- und Optimierungsprobleme nicht so genau nehmen. Wir werden häufig einfach von *$\mathcal{NP}$-vollständigen Optimierungsproblemen* sprechen, wenn diese $\mathcal{NP}$-schwer sind. Die Begriffe $\mathcal{NP}$-leicht und $\mathcal{NP}$-äquivalent werden wir kaum gebrauchen, da sie für unsere Belange nicht so wichtig sind.

Die folgenden Beispiele von kombinatorischen Optimierungsproblemen, die wir in früheren Abschnitten eingeführt haben, sind $\mathcal{NP}$-schwer:

- das symmetrische Travelling Salesman Problem
- das asymmetrische Travelling Salesman Problem
- das Chinesische Postbotenproblem für gemischte Graphen
- fast alle Routenplanungsprobleme
- das Stabile-Mengen-Problem

- das Cliquenproblem
- das Knotenüberdeckungsproblem
- das Knotenfärbungsproblem
- das Kantenfärbungsproblem
- das Max-Cut-Problem
- die meisten Standortprobleme
- das Linear-Ordering-Problem
- das azyklische Subdigraphenproblem
- das Feedback-Arc-Set-Problem.

Einige hundert weitere $\mathcal{NP}$-vollständige bzw. $\mathcal{NP}$-schwere Probleme und einige tausend Varianten von diesen sind in dem bereits zitierten Buch von Garey and Johnson (1979) aufgeführt. Probleme, die mit diesem Themenkreis zusammenhängen, wurden auch in einer von 1981 – 1992 laufenden Serie von Aufsätzen von D. S. Johnson mit dem Titel „The $\mathcal{NP}$-completeness column: an ongoing guide“ im *Journal of Algorithms* behandelt. Seit 2005 sind drei weitere Artikel der Serie in *ACM Trans. Algorithms* publiziert worden. Der derzeit letzte Artikel ist 2007 erschienen. Alle (bisher) 26 Artikel sind unter der folgenden URL zu finden:

`http://www.research.att.com/~dsj/columns/`

Im Internet finden Sie unter der URL

`http://www.nada.kth.se/~viggo/problemlist/compendium.html`

ein „Compendium of $\mathcal{NP}$ Optimization Problems“. Eine aktuelle Zusammenfassung der Geschichte des Konzepts der $\mathcal{NP}$-Vollständigkeit findet sich in Johnson (2012).

Der wichtigste Aspekt der hier skizzierten Theorie ist, dass man zwischen „einfachen“ und „schwierigen“ Problemen zu unterscheiden lernt, und dass man – sobald man weiß, dass ein Problem schwierig ist – andere Wege (Heuristiken, etc.) als bei Problemen in $\mathcal{P}$ suchen muss, um das Problem optimal oder approximativ zu lösen. In dieser Vorlesung soll versucht werden, einige der Methoden zu beschreiben, mit denen man derartige Probleme angreifen kann.

Zum Schluss dieses Abschnitts sei angemerkt, dass es noch viel schwierigere Probleme als die $\mathcal{NP}$-schweren Probleme gibt. Ein „Klassiker“ unter den wirklich schwierigen Problemen ist das Halteproblem von Turing-Maschinen, die wir in Abschnitt 4.1 erwähnt haben. Wir skizzieren die Fragestellung kurz.

Wir nennen ein Entscheidungsproblem *entscheidbar*, wenn es eine Turing-Maschine (genauer einen Algorithmus, der auf einer Turing-Maschine läuft) gibt, die für jede Eingabe

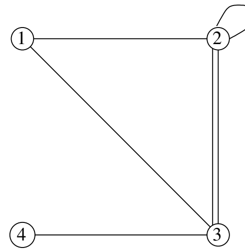


Abbildung 4.1: Ungerichteter Graph zur Kantenliste 4, 6, 1, 2, 2, 3, 4, 3, 3, 2, 2, 2, 1, 3.

terminiert und „ja" oder „nein" antwortet. Das *Halteproblem* ist nun das folgende: Gegeben seien eine Turing-Maschine M und eine Eingabe w, hält M auf w? Dieses Halteproblem ist beweisbar unentscheidbar. Der *Satz von Rice* zeigt noch etwas viel Stärkeres. Alle „interessanten" (das kann man präzise definieren) Eigenschaften von Turing-Maschinen sind unentscheidbar.

David Hilbert hat im Jahr 1900 eine Liste von 23 wichtigen, seinerzeit ungelösten, mathematischen Problemen vorgelegt. Sein zehntes Problem lautete: Gibt es einen Algorithmus, der für eine beliebige diophantische Gleichung entscheidet, ob sie lösbar ist? Diophantische Gleichungen sind von der Form $p(x_1, \ldots, x_n) = 0$, wobei p ein Polynom mit ganzzahligen Koeffizienten ist. J. Matijassewitsch hat 1970 in seiner Dissertation bewiesen, dass es keinen solchen Algorithmus gibt. Um Lösungen von allgemeinen diophantischen Gleichungen zu finden, muss man also spezielle Fälle betrachten, gute Einfälle und Glück haben.

4.3 Datenstrukturen zur Speicherung von Graphen

Wir wollen hier einige Methoden skizzieren, mit deren Hilfe man Graphen und Digraphen speichern kann, und ihre Vor- und Nachteile besprechen. Kenntnisse dieser Datenstrukturen sind insbesondere dann wichtig, wenn man laufzeit- und speicherplatzeffiziente (oder gar -optimale) Codes von Algorithmen entwickeln will.

4.3.1 Kanten- und Bogenlisten

Die einfachste Art, einen Graphen oder Digraphen zu speichern, ist die *Kantenliste* für Graphen bzw. die *Bogenliste* für Digraphen. Ist $G = (V, E)$ ein Graph mit $n = |V|$ Knoten und $m = |E|$ Kanten, so sieht eine Kantenliste wie folgt aus:

$n, m, a_1, e_1, a_2, e_2, a_3, e_3, \ldots, a_m, e_m,$

wobei a_i, e_i die beiden Endknoten der Kante i sind. Die Reihenfolge des Aufführens der Endknoten von i bzw. den Kanten selbst ist beliebig. Bei Schleifen wird der Endknoten zweimal aufgelistet.

Eine mögliche Kantenliste für den Graphen aus Abbildung 4.1 ist die folgende:

$$4, 6, 1, 2, 2, 3, 4, 3, 3, 2, 2, 2, 1, 3.$$

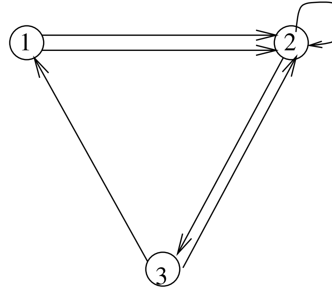


Abbildung 4.2: Gerichteter Graph zur Kantenliste $3, 6, 1, 2, 2, 3, 3, 2, 2, 2, 1, 2, 3, 1$.

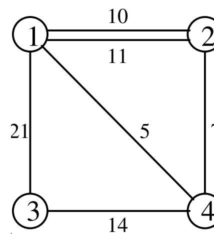


Abbildung 4.3: Gewichteter Graph mit den Kantenlisten $4, 6, 1, 2, 10, 2, 1, 11, 2, 4, 7, 4, 3, 14, 3, 1, 21, 1, 4, 5$ und $4, 6, 1, 2, 1, 2, 2, 4, 3, 4, 1, 4, 1, 3, 11, 10, 7, 14, 5, 21$.

Bei der Bogenliste eines Digraphen verfahren wir genauso; wir müssen jedoch darauf achten, dass ein Bogen immer zunächst durch seinen Anfangs- und dann durch seinen Endknoten repräsentiert wird.

Eine Bogenliste des Digraphen aus Abbildung 4.2 ist

$$3, 6, 1, 2, 2, 3, 3, 2, 2, 2, 1, 2, 3, 1.$$

Haben die Kanten oder Bögen Gewichte, so repräsentiert man eine Kante (einen Bogen) entweder durch Anfangsknoten, Endknoten, Gewicht oder macht eine Kanten- bzw. Bogenliste wie oben und hängt an diese noch eine Liste mit den m Gewichten der Kanten $1, 2, \ldots, m$ an.

Der gewichtete Graph aus Abbildung 4.3 ist in den beiden folgenden Kantenlisten mit Gewichten gespeichert:

$$4, 6, 1, 2, 10, 2, 1, 11, 2, 4, 7, 4, 3, 14, 3, 1, 21, 1, 4, 5$$

$$4, 6, 1, 2, 1, 2, 2, 4, 3, 4, 1, 4, 1, 3, 11, 10, 7, 14, 5, 21.$$

Der Speicheraufwand einer Kanten- bzw. Bogenliste beträgt $2(m+1)$ Speicherplätze, eine Liste mit Gewichten erfordert $3m+2$ Speicherplätze.

4.3.2 Adjazenzmatrizen

Ist $G = (V, E)$ ein ungerichteter Graph mit $V = \{1, 2, \ldots, n\}$, so ist die symmetrische (n, n)-Matrix $A = (a_{ij})$ mit

$$\begin{aligned} a_{ji} = a_{ij} &= \text{ Anzahl der Kanten, die } i \text{ und } j \text{ verbinden, falls } i \neq j \\ a_{ii} &= 2 \cdot (\text{Anzahl der Schleifen, die } i \text{ enthalten}),\ i = 1, \ldots, n \end{aligned}$$

die *Adjazenzmatrix* von G. Aufgrund der Symmetrie kann man etwa die Hälfte der Speicherplätze sparen. Hat ein Graph keine Schleifen (unser Normalfall), dann genügt es, die obere (oder untere) Dreiecksmatrix von A zu speichern. Man macht das z. B. in der Form

$$a_{12}, a_{13}, \ldots, a_{1n}, a_{23}, a_{24}, \ldots, a_{2n}, a_{34}, \ldots, a_{n-1,n}.$$

Die Adjazenzmatrix des Graphen in Abbildung 4.1 sieht wie folgt aus:

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 2 & 2 & 0 \\ 1 & 2 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Hat ein Graph Kantengewichte, und ist er einfach, so setzt man

$$a_{ij} = \text{ Gewicht der Kante } ij,\ \ i, j = 1, \ldots, n.$$

Ist eine Kante ij nicht im Graphen G enthalten, so setzt man

$$a_{ij} = 0, \quad a_{ij} = -\infty \quad \text{oder} \quad a_{ij} = +\infty,$$

je nachdem, welche Gewichtszuordnung für das Problem sinnvoll ist und die Benutzung dieser Kante ausschließt. Wiederum braucht man von der so definierten Matrix A nur die obere Dreiecksmatrix zu speichern.

Die Adjazenzmatrix $A = (a_{ij})$ eines Digraphen $D = (V, E)$ mit $V = \{1, 2, \ldots, n\}$ ohne Schleifen ist definiert durch

$$\begin{aligned} a_{ii} &= 0, \quad i = 1, 2, \ldots, n \\ a_{ij} &= \text{ Anzahl der Bögen in } E \text{ mit Anfangsknoten } i \text{ und Endknoten } j,\ i \neq j. \end{aligned}$$

Bogengewichte werden wie bei Adjazenzmatrizen von Graphen behandelt. Der Speicheraufwand von Adjazenzmatrizen beträgt n^2 bzw. $\binom{n}{2}$ Speicherplätze, falls bei einem ungerichteten Graphen nur die obere Dreiecksmatrix gespeichert wird.

4.3.3 Adjazenzlisten

Speichert man für einen Graphen $G = (V, E)$ die Anzahl der Knoten und für jeden Knoten $v \in V$ seinen Grad und die Namen der Nachbarknoten, so nennt man eine solche Datenstruktur *Adjazenzliste* von G. Für den Graphen aus Abbildung 4.1 sieht die

Adjazenzliste wie folgt aus

$$
\begin{array}{cccc}
 & \text{Knotennummer} & & \\
 & \downarrow & & \\
4 & 1 & 2 & 2,\ 3 \\
 & 2 & 5 & 1,\ 3,\ 3,\ 2,\ 2 \\
 & 3 & 4 & 1,\ 2,\ 2,\ 4 \\
 & 4 & 1 & 3 \\
 & & \uparrow & \\
 & & \text{Grad} &
\end{array}
$$

Die Liste der Nachbarknoten eines Knoten v heißt *Nachbarliste* von v. Jede Kante ij, $i \neq j$, ist zweimal repräsentiert, einmal auf der Nachbarliste von i, einmal auf der von j. Bei Gewichten hat man wieder zwei Möglichkeiten. Entweder man schreibt direkt hinter jeden Knoten j auf der Nachbarliste von i das Gewicht der Kante ij, oder man legt eine kompatible Gewichtsliste an.

Bei Digraphen geht man analog vor, nur speichert man auf der Nachbarliste eines Knoten i nur die Nachfolger von i. Man nennt diese Liste daher auch *Nachfolgerliste*. Ein Bogen wird also nur einmal in der Adjazenzliste eines Digraphen repräsentiert. (Wenn es für das Problem günstiger ist, kann man natürlich auch Vorgängerlisten statt Nachfolgerlisten oder beides anlegen.)

Zur Speicherung einer Adjazenzliste eines Graphen braucht man $2n+1+2m$ Speicherplätze, für die Adjazenzliste eines Digraphen werden $2n+1+m$ Speicherplätze benötigt.

Kanten- bzw. Bogenlisten sind die kompaktesten aber unstrukturiertesten Speicherformen. Will man wissen, ob G die Kante ij enthält, muss man im schlechtesten Fall die gesamte Liste durchlaufen und somit $2m$ Abfragen durchführen. Eine derartige Frage benötigt bei der Speicherung von G in einer Adjazenzmatrix nur eine Abfrage, während man bei einer Adjazenzliste die Nachbarliste von i (oder j) durchlaufen und somit (Grad von i), im schlechtesten Fall also $n-1$ Abfragen durchführen muss.

Für dünn besetzte Graphen ist die Speicherung in einer Adjazenzmatrix i. a. zu aufwendig. Es wird zu viel Platz vergeudet. Außerdem braucht fast jeder Algorithmus, der für Adjazenzmatrizen konzipiert ist, mindestens $O(n^2)$ Schritte, da er ja jedes Element von A mindestens einmal anschauen muss, um zu wissen, ob die zugehörige Kante im Graphen ist oder nicht. Mit Hilfe von Adjazenzlisten kann man dagegen dünn besetzte Graphen in der Regel sehr viel effizienter bearbeiten. Das Buch Aho, Hopcroft & Ullman (1974), Reading, MA, Addison-Wesley, informiert sehr ausführlich über dieses Thema.

Wir wollen hier als Beispiel nur einen einzigen einfachen, aber vielseitig und häufig anwendbaren Algorithmus zur Untersuchung von Graphen erwähnen: das Depth-First-Search-Verfahren (kurz DFS-Verfahren bzw. auf deutsch: Tiefensuche).

Wir nehmen an, dass ein Graph $G = (V, E)$ gegeben ist und alle Knoten *unmarkiert* sind. Alle Kanten seien *unbenutzt*. Wir wählen einen Startknoten, sagen wir v, und *markieren* ihn. Dann wählen wir eine Kante, die mit v inzidiert, sagen wir vw, gehen zu w und *markieren* w. Die Kante vw ist nun *benutzt* worden. Allgemein verfahren wir wie folgt. Ist x der letzte von uns markierte Knoten, dann versuchen wir eine mit x inzidente Kante xy zu finden, die noch nicht benutzt wurde. Ist y markiert, so suchen wir eine weitere

mit x inzidente Kante, die noch nicht benutzt wurde. Ist y nicht markiert, dann gehen wir zu y, markieren y und beginnen von neuem (y ist nun der letzte markierte Knoten). Wenn die Suche nach Kanten, die mit y inzidieren und die noch nicht benutzt wurden, beendet ist (d. h. alle Kanten, auf denen y liegt, wurden einmal berührt), kehren wir zu x zurück und fahren mit der Suche nach unbenutzten Kanten, die mit x inzidieren fort, bis alle Kanten, die x enthalten, abgearbeitet sind. Diese Methode nennt man Tiefensuche, da man versucht, einen Knoten so schnell wie möglich zu verlassen und „tiefer" in den Graphen einzudringen.

Eine derartige Tiefensuche teilt die Kanten des Graphen in zwei Teilmengen auf. Eine Kante xy heißt *Vorwärtskante*, falls wir bei der Ausführung des Algorithmus von einem markierten Knoten x entlang xy zum Knoten y gegangen sind und dabei y markiert haben. Andernfalls heißt xy *Rückwärtskante*. Man überlegt sich leicht, dass die Menge der Vorwärtskanten ein Wald von G ist, der in jeder Komponente von G einen aufspannenden Baum bildet. Ist der Graph zusammenhängend, so nennt man die Menge der Vorwärtskanten *DFS-Baum* von G. Mit Hilfe von Adjazenzlisten kann die Tiefensuche sehr effizient rekursiv implementiert werden.

(4.6) Algorithmus Depth-First-Search.

Eingabe: Graph $G = (V, E)$ in Form einer Adjazenzliste, d. h. für jeden Knoten $v \in V$ ist eine Nachbarliste $N(v)$ gegeben.

Ausgabe: Kantenmenge T (= DFS-Baum, falls G zusammenhängend ist).

1: Alle Knoten $v \in V$ seien unmarkiert.
2: Setze $T := \emptyset$.
3: **for all** $v \in V$ **do**
4:     Ist v unmarkiert, dann rufe SEARCH(v) auf.
5: **end for**
6: Gib T aus.
7: **procedure** SEARCH(v) ▷ Rekursives Unterprogramm
8:     Markiere v.
9:     **for all** $w \in N(v)$ **do**
10:         Ist w unmarkiert, setze $T := T \cup \{vw\}$ und rufe SEARCH(w) auf.
11:     **end for**
12: **end procedure**

In Algorithmus (4.6) wird im Hauptprogramm jeder Knoten einmal berührt, und im Unterprogramm jede Kante genau zweimal. Hinzu kommt die Ausgabe von T. Die Laufzeit des Verfahrens ist also $O(|V| + |E|)$. Diese Laufzeit könnte man niemals bei der Speicherung von G in einer Adjazenzmatrix erreichen.

Mit Hilfe des obigen Verfahrens können wir unser mehrmals zitiertes Problem „Enthält G einen Kreis?" lösen. Offensichtlich gilt: G enthält genau dann einen Kreis, wenn $E \setminus T$ nicht leer ist. Wir haben somit einen polynomialen Algorithmus zur Lösung des Kreisproblems gefunden. Der DFS-Baum, von Algorithmus (4.6) produziert, hat einige interessante Eigenschaften, die man dazu benutzen kann, eine ganze Reihe von weiteren Graphenproblemen sehr effizient zu lösen. Der hieran interessierte Leser sei z. B. auf das

Buch Aho et al. (1974) verwiesen.

Literaturverzeichnis

M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in $\mathcal{P}$. *Annals of Mathematics*, 160: 781–793, 2004.

A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, 1974.

W. J. Cook. The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, Ohio, 1971. Shaker Heights.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, Mass., 2001.

M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of $\mathcal{NP}$-completeness*. W. H. Freeman and Company, New York, 1979. ISBN 0716710455.

M. Grötschel. $\mathcal{P} = \mathcal{NP}$?. *Elemente der Mathematik, Eine Zeitschrift der Schweizerischen Mathematischen Gesellschaft*, 57(3):96–102, 2002. (siehe URL: http://www.zib.de/groetschel/pubnew/biblio.html).

D. S. Johnson. A catalog of complexity classes. In J. van Leeuwen (1990), editor, *Algorithms and Complexity, Handbook of Theoretical Computer Science*, volume Vol. A, pages 67–161. Elsevier, Amsterdam, 1990.

D. S. Johnson. A brief history of $\mathcal{NP}$-completeness, 1954–2012. *Documenta Mathematica*, pages 359–376, 2012.

K. Mehlhorn. *Data Structures and Algorithms*, volume 1–3. Springer-Verlag, EATCS Monographie edition, 1984. (dreibändige Monographie, Band I liegt auch auf deutsch im Teubner-Verlag (1986) vor).

D. P. Mehta and S. Sahni, editors. *Handbook of Data Structures and Applications*. Chapman & Hall, 2004.

C. Meinel. *Effiziente Algorithmen. Entwurf und Analyse*. Fachbuchverlag, Leipzig, 1991.

T. Ottmann and P. Widmayer. *Algorithmen und Datenstrukturen*. Spektrum Akademischer Verlag, 2012.

C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Amsterdam, 1994.

D. B. Shmoys and E. Tardos. Computational complexity. In *Handbook of Combinatorics*, chapter 29, pages 1599–1645. North-Holland, Amsterdam, 1995.

R. E. Tarjan. Data structures and network algorithms. *SIAM*, 1983.

J. van Leeuwen. Algorithms and complexity. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, pages 525–631. Elsevier, Amsterdam, 1990.

K. Wagner and G. Wechsung. *Computational Complexity.* VEB Deutscher Verlag der Wissenschaften, Berlin, 1986.

I. Wegener. *Komplexitätstheorie: Grenzen der Effizienz von Algorithmen.* Springer, 2003.

5 Bäume und Wege

Dieses Kapitel ist einem Thema gewidmet, das algorithmisch sehr einfach zu lösen ist: Bestimme einen kostenminimalen aufspannenden Baum in einem Graphen. Wir werden Varianten dieser Aufgabe und auch „gerichtete Versionen" betrachten.

Bevor wir jedoch algorithmischen Fragen nachgehen, sollen Wälder aus graphentheoretischer Sicht analysiert werden. Das Ziel des ersten Abschnitts dieses Kapitels ist nicht eine umfassende Behandlung des Themenkreises, sondern das Einüben typischer graphentheoretischer Beweisargumente. Bäume sind sehr einfache Objekte. Die meisten Eigenschaften von Bäumen können mit minimalem Aufwand nachgewiesen werden. Die dabei benutzten Argumente tauchen jedoch in der Graphentheorie – meistens in etwas komplizierterer Form – immer wieder auf. Wir hoffen, diese Beweistechniken hier sichtbar machen zu können.

5.1 Graphentheoretische Charakterisierungen von Bäumen und Arboreszenzen

Wir erinnern daran, dass ein Graph, der keinen Kreis enthält, *Wald* genannt wird, dass ein Graph G *zusammenhängend* heißt, wenn es in G zwischen je zwei Knoten eine sie verbindende Kette (oder äquivalent dazu, einen sie verbindenden Weg) gibt, und dass ein *Baum* ein zusammenhängender Wald ist. Ein Untergraph eines Graphen $G = (V, E)$, der ein Baum ist und alle Knoten V enthält, heißt *aufspannender Baum* (von G). Eine *Zusammenhangskomponente* (kurz: *Komponente*) eines Graphen G ist ein maximaler zusammenhängender Untergraph von G. Wir werden nun einige Eigenschaften von Bäumen und Wäldern beweisen. Wir beginnen mit trivialen Beobachtungen.

(5.1) Lemma. *Ein Baum $G = (V, E)$ mit mindestens zwei Knoten hat mindestens zwei Knoten mit Grad 1.* $\triangle$

Beweis. Da ein Baum zusammenhängend ist, liegt jeder Knoten auf mindestens einer Kante. Wir wählen einen beliebigen Knoten, sagen wir v. Wir starten in v einen (vereinfachten) DFS-Algorithmus. Wir markieren v und gehen zu einem Nachbarn, sagen wir w, von v. Wir markieren w. Hat w den Grad 1, stoppen wir die Suche. Andernfalls gehen wir zu einem von v verschiedenen Nachbarn von w und fahren so fort. Da ein Baum keinen Kreis enthält, kehrt dieses Verfahren niemals zu einem bereits markierten Knoten zurück. Da der Graph endlich ist, muss das Verfahren irgendwann mit einem Knoten mit Grad 1 aufhören. Hat der Anfangsknoten v auch Grad 1, können wir aufhören. Falls nicht, gehen wir zu einem von w verschiedenen Nachbarn von v und wiederholen das obige Verfahren. Auf diese Weise finden wir einen zweiten Knoten mit Grad 1. $\square$

Der Beweis ist etwas länglich geraten. Der Grund dafür ist, einmal zu zeigen, wie durch einfache Analyse eines sehr einfachen Algorithmus Eigenschaften von Graphen nachgewiesen werden können. Was können Sie aus diesem Beweisverfahren „herausholen“, wenn Sie den Algorithmus statt mit einem beliebigen Knoten v mit einem Knoten v mit maximalem Grad beginnen?

(5.2) Lemma. *Sei $G = (V, E)$ ein Graph.*

(a) Es gilt: $2|E| = \sum_{v \in V} \deg(v)$.

(b) Ist G ein Baum, so gilt: $|E| = |V| - 1$.

(c) Gilt $\deg(v) \geq 2$ *für alle $v \in V$, so enthält G einen Kreis.* △

Beweis. (a) Da jede Kante genau zwei (nicht notwendig verschiedene) Knoten enthält, wird bei der Summe der Knotengerade jede Kante genau zweimal gezählt.

(b) Beweis durch Induktion! Die Behauptung ist offensichtlich richtig für $|V| = 1$ und $|V| = 2$. Wir nehmen an, dass die Behauptung korrekt ist für alle Bäume mit höchstens $n \geq 2$ Knoten. Sei $G = (V, E)$ ein Baum mit $n + 1$ Knoten. Nach Lemma (5.1) enthält G einen Knoten v mit Grad 1. $G - v$ ist dann ein Baum mit n Knoten. Nach Induktionsvoraussetzung hat $G - v$ genau $n - 1$ Kanten, also enthält G genau n Kanten.

(c) Enthält G Schlingen, so ist nichts zu zeigen, da Schlingen Kreise sind. Andernfalls führen wir folgenden Markierungsalgorithmus aus. Wir wählen einen beliebigen Knoten v_1 und markieren ihn. Wir wählen einen beliebigen Nachbarn v_2 von v_1 und markieren sowohl v_2 als auch die Kante v_1v_2. Haben wir die Knoten $v_1, \ldots, v_k$, $k \geq 2$, markiert, so enthält G, da $\deg(v) \geq 2$, eine von der markierten Kante $v_{k-1}v_k$ verschiedene Kante v_kv_{k+1} zu einem v_k-Nachbarn v_{k+1}. Ist v_{k+1} bereits markiert, so ist v_{k+1} einer der bereits markierten Knoten, sagen wir v_i, $1 \leq i \leq k$. Daher bildet die Knotenfolge $(v_i, v_{i+1}, \ldots, v_k)$ einen Kreis. Ist v_{k+1} nicht markiert, so markieren wir v_{k+1} und die Kante v_kv_{k+1} und fahren mit unserem Markierungsalgorithmus fort. Da $|V|$ endlich ist, wird nach endlich vielen Schritten die Situation eintreten, dass jeder Nachbar des gerade markierten Knotens v_k bereits markiert ist. Und damit ist ein Kreis gefunden. □

(5.3) Lemma. *Ein Graph $G = (V, E)$ mit mindestens 2 Knoten und mit weniger als $|V| - 1$ Kanten ist unzusammenhängend.* △

Beweis. Sei $m := |E|$. Wäre G zusammenhängend, müsste es in G von jedem Knoten zu jedem anderen einen Weg geben. Wir führen einen Markierungsalgorithmus aus. Wir wählen einen beliebigen Knoten $v \in V$ und markieren v. Wir markieren alle Nachbarn von v und entfernen die Kanten, die von v zu seinen Nachbarn führen. Wir gehen nun zu einem markierten Knoten, markieren dessen Nachbarn und entfernen die Kanten, die

noch zu diesen Nachbarn führen. Wir setzen dieses Verfahren fort, bis wir keinen Knoten mehr markieren können. Am Ende haben wir höchstens m Kanten entfernt sowie v und maximal m weitere Knoten markiert. Da $m < |V| - 1$ gilt, ist mindestens ein Knoten unmarkiert, also nicht von v aus auf einem Weg erreichbar. Daher ist G unzusammenhängend. □

Der nächste Satz zeigt, dass die Eigenschaft, ein Baum zu sein, auf viele äquivalente Weisen charakterisiert werden kann.

(5.4) Satz. *Sei $G = (V, E)$, $|V| = n \geq 2$ ein Graph. Dann sind äquivalent:*

(1) G ist ein Baum.

(2) G enthält keinen Kreis und $n - 1$ Kanten.

(3) G ist zusammenhängend und enthält $n - 1$ Kanten.

(4) G ist zusammenhängend und enthält keinen Kreis.

(5) Jedes Knotenpaar aus V ist durch genau einen Weg miteinander verbunden.

(6) G enthält keinen Kreis; wird irgendeine Kante uv mit $u, v \in V$ und $uv \notin E$ zu G hinzugefügt, so entsteht genau ein Kreis.

(7) G ist zusammenhängend, und für alle $e \in E$ ist $G - e$ unzusammenhängend. △

Beweis. (1) $\iff$ (4) Definition.

(4) $\Longrightarrow$ (5) Da G zusammenhängend ist, ist jedes Knotenpaar durch einen Weg miteinander verbunden. Gibt es zwischen einem Knotenpaar zwei verschiedene Wege, so ist die Verknüpfung dieser beiden Wege eine geschlossene Kette, die offensichtlich einen Kreis enthält. Widerspruch!

(5) $\Longrightarrow$ (6) Enthielte G einen Kreis, so gäbe es Knotenpaare, die durch zwei verschiedene Wege miteinander verbunden sind. Also enthält G keinen Kreis. Sei $uv \notin E$. Da G einen $[u, v]$-Weg P enthält, ist $P \cup uv$ ein Kreis. Gäbe es in $G + uv$ einen weiteren Kreis, so gäbe es in G zwei verschiedene $[u, v]$-Wege, ein Widerspruch!

(6) $\Longrightarrow$ (7) Gibt es für $uv \notin E$ in $G + uv$ einen Kreis, so gibt es in G einen $[u, v]$-Weg. Daraus folgt, dass G zusammenhängend ist. Gibt es eine Kante $uv \in E$ mit $G - uv$ zusammenhängend, so gibt es in $G - uv$ einen $[u, v]$-Weg P. Dann aber ist $P \cup uv$ ein Kreis in G, Widerspruch!

(7) $\Longrightarrow$ (4) Gäbe es in G einen Kreis, so wäre $G - e$ für jede Kante e dieses Kreises zusammenhängend. Also enthält G keinen Kreis.

(4) $\Longrightarrow$ (2) folgt aus Lemma (5.2).

(2) $\Longrightarrow$ (3) Ist G ein Graph, der (2) erfüllt, so gilt nach Voraussetzung $|E| = |V| - 1$ und nach Lemma (5.2)(a) $2|E| = \sum_{v \in V} \deg(v)$. Der Durchschnittsgrad der Knoten von G ist daher gleich $2(|V| - 1)/|V|$ und somit kleiner als 2.

Hat G keinen Knoten mit Grad 1, so gibt es Knoten mit Grad 0. Wir entfernen diese aus G. Es verbleibt ein Graph, in dem alle Knoten mindestens Grad 2 haben. Dieser enthält nach Lemma (5.2)(c) einen Kreis und somit auch G, ein Widerspruch.

In G gibt es folglich einen Knoten mit Grad 1. Wir beweisen die Aussage durch Induktion. Sie ist offensichtlich korrekt für $|V| = 2$. Wir nehmen an, dass sie für alle Graphen mit höchstens $n \geq 2$ Knoten gilt und wählen einen Graphen G mit $n + 1$ Knoten. G enthält dann, wie gerade gezeigt, einen Knoten v mit $\deg(v) = 1$. $G - v$ hat n Knoten und $n - 1$ Kanten und enthält keinen Kreis. Nach Induktionsannahme ist dann $G - v$ zusammenhängend. Dann aber ist auch G zusammenhängend.

(3) $\Longrightarrow$ (4) Angenommen G enthält einen Kreis. Sei e eine Kante des Kreises, dann ist $G - e$ zusammenhängend und hat $n - 2$ Kanten. Dies widerspricht Lemma (5.3). □

Eine Kante e eines Graphen G, die die Eigenschaft besitzt, dass $G - e$ unzusammenhängend ist, heißt *Brücke*. Satz (4.4) (7) zeigt insbesondere, dass G ein Baum genau dann ist, wenn jede Kante von G eine Brücke ist.

(5.5) Korollar.

(a) Ein Graph ist zusammenhängend genau dann, wenn er einen aufspannenden Baum enthält.

(b) Sei $G = (V, E)$ ein Wald und sei p die Anzahl der Zusammenhangskomponenten von G, dann gilt $|E| = |V| - p$. △

Die hier bewiesenen Eigenschaften kann man auf analoge Weise auch auf gerichtete Bäume und Wälder übertragen. Wir geben hier nur die Resultate an und laden den Leser ein, die Beweise selbst auszuführen.

Ein Digraph, der keinen Kreis enthält und bei dem jeder Knoten den Innengrad höchstens 1 hat (also $\deg^-(v) \leq 1$), heißt *Branching*. Ein zusammenhängendes Branching heißt *Arboreszenz*. Jedes Branching ist also ein Wald, jede Arboreszenz ein Baum. Ein Knoten v in einem Digraphen heißt *Wurzel*, wenn jeder Knoten des Digraphen von v aus auf einem gerichteten Weg erreicht werden kann.

Ein Digraph D heißt *quasi-stark zusammenhängend*, falls es zu jedem Paar u, v von Knoten einen Knoten w in D (abhängig von u und v) gibt, so dass es von w aus einen gerichteten Weg zu u und einen gerichteten Weg zu v gibt.

Es ist einfach zu sehen, dass jede Arboreszenz genau eine Wurzel hat, und dass ein Digraph genau dann quasi-stark zusammenhängend ist, wenn er eine Wurzel besitzt.

(5.6) Satz. *Sei $D = (V, A)$ ein Digraph mit $n \geq 2$ Knoten. Dann sind die folgenden Aussagen äquivalent:*

(1) D ist eine Arboreszenz.

(2) D ist ein Baum mit Wurzel.

(3) D hat $n-1$ *Bögen und ist quasi-stark zusammenhängend.*

(4) D enthält keinen Kreis und ist quasi-stark zusammenhängend.

(5) D enthält einen Knoten r*, so dass es in D für jeden anderen Knoten* v *genau einen gerichteten* (r,v)*-Weg gibt.*

(6) D ist quasi-stark zusammenhängend, und für alle $a \in A$ *ist* $D-a$ *nicht quasi-stark zusammenhängend.*

(7) D ist quasi-stark zusammenhängend, besitzt einen Knoten r *mit* $\deg^-(r)=0$ *und erfüllt* $\deg^-(v)=1$ *für alle* $v \in V\setminus\{r\}$*.*

(8) D ist ein Baum, besitzt einen Knoten r *mit* $\deg^-(r)=0$ *und erfüllt* $\deg^-(v)=1$ *für alle* $v \in V\setminus\{r\}$*.*

(9) D enthält keinen Kreis, einen Knoten r *mit* $\deg^-(r)=0$ *und erfüllt* $\deg^-(v)=1$ *für alle* $v \in V\setminus\{r\}$*.* △

5.2 Optimale Bäume und Wälder

Das Problem, in einem Graphen mit Kantengewichten einen aufspannenden Baum minimalen Gewichts oder einen Wald maximalen Gewichts zu finden, haben wir bereits in (3.11) eingeführt. Beide Probleme sind sehr effizient lösbar und haben vielfältige Anwendungen. Umfassende Überblicke über die Geschichte dieser Probleme, ihre Anwendungen und die bekannten Lösungsverfahren geben die Aufsätze Graham and Hell (1982) und Nešetřil and Nešetřilová (2012) sowie Kapitel 50 des Buches von Schrijver (2003), volume B.

Wir wollen hier jedoch nur einige dieser Lösungsmethoden besprechen. Zunächst wollen wir uns überlegen, dass die beiden Probleme auf sehr direkte Weise äquivalent sind. Angenommen wir haben einen Algorithmus zur Lösung eines Maximalwald-Problems, und wir wollen in einem Graphen $G=(V,E)$ mit Kantengewichten c_e, $e \in E$, einen minimalen aufspannenden Baum finden, dann gehen wir wie folgt vor. Wir setzen

$$
\begin{aligned}
M &:= \max\{|c_e| \mid e \in E\} + 1,\\
c'_e &:= M - c_e
\end{aligned}
$$

und bestimmen einen maximalen Wald W in G bezüglich der Gewichtsfunktion c'. Falls G zusammenhängend ist, ist W ein aufspannender Baum, denn andernfalls gäbe es eine Kante $e \in E$, so dass $W' := W \cup \{e\}$ ein Wald ist, und wegen $c'_e > 0$, wäre $c'(W') > c'(W)$. Aus der Definition von c' folgt direkt, dass W ein minimaler aufspannender Baum von G bezüglich c ist. Ist W nicht zusammenhängend, so ist auch G nicht zusammenhängend, also existiert kein aufspannender Baum.

Haben wir einen Algorithmus, der einen minimalen aufspannenden Baum in einem Graphen findet, und wollen wir einen maximalen Wald in einem Graphen $G = (V, E)$ mit Kantengewichten c_e, $e \in E$, bestimmen, so sei $K_n = (V, E_n)$ der vollständige Graph mit $n = |V|$ Knoten und folgenden Kantengewichten

$$\begin{aligned} c'_e &:= -c_e \quad \text{für alle } e \in E \text{ mit } c_e > 0, \\ c'_e &:= M \quad \text{für alle } e \in E_n \setminus \{e \in E \mid c_e > 0\}, \end{aligned}$$

wobei wir z. B. setzen

$$M := n \cdot (\max\{|c_e| \mid e \in E\} + 1).$$

Ist B ein minimaler aufspannender Baum von K_n bezüglich der Kantengewichte c', dann ist offenbar aufgrund unserer Konstruktion $W := B \setminus \{e \in B \mid c'_e = M\}$ ein Wald in G maximalen Gewichts.

Der folgende sehr einfache Algorithmus findet einen maximalen Wald.

(5.7) Algorithmus GREEDY-MAX.
Eingabe: Graph $G = (V, E)$ mit Kantengewichten $c(e)$ für alle $e \in E$.
Ausgabe: Wald $W \subseteq E$ mit maximalem Gewicht $c(W)$.

1. (Sortieren): Ist k die Anzahl der Kanten von G mit positivem Gewicht, so numeriere diese k Kanten, so dass gilt $c(e_1) \geq c(e_2) \geq \ldots \geq c(e_k) > 0$.
2. Setze $W := \emptyset$.
3. FOR $i = 1$ TO k DO:

   Falls $W \cup \{e_i\}$ keinen Kreis enthält, setze $W := W \cup \{e_i\}$.
4. Gib W aus. △

(5.8) Satz. *Der Algorithmus GREEDY-MAX arbeitet korrekt.* △

Beweis. Hausaufgabe! □

Versuchen Sie, einen direkten Beweis für die Korrektheit von Algorithmus (5.7) zu finden. Im nachfolgenden Teil dieses Abschnitts und in Kapitel 5 werden wir Sätze angeben, aus denen Satz (5.8) folgt.

Der obige Algorithmus heißt "Greedy-Max" ("greedy" bedeutet "gierig" oder "gefräßig"), weil er versucht, das bezüglich der Zielfunktionskoeffizienten jeweils "Beste" zu nehmen, das im Augenblick noch verfügbar ist. Wir werden später noch andere Algorithmen vom Greedy-Typ kennenlernen, die bezüglich anderer Kriterien das "Beste" wählen. Der Greedy-Algorithmus funktioniert auf analoge Weise für das Minimum Spanning Tree Problem.

(5.9) Algorithmus GREEDY-MIN.

Eingabe: Graph $G = (V, E)$ mit Kantengewichten $c(e)$ für alle $e \in E$.
Ausgabe: Maximaler Wald $T \subseteq E$ mit minimalem Gewicht $c(T)$.

1. (Sortieren): Numeriere die m Kanten des Graphen G, so dass gilt
   $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$.
2. Setze $T := \emptyset$.
3. FOR $i = 1$ TO m DO:

   Falls $T \cup \{e_i\}$ keinen Kreis enthält, setze $T := T \cup \{e_i\}$.
4. Gib T aus. △

Aus Satz (5.8) und unseren Überlegungen zur Reduktion des Waldproblems auf das Baumproblem und umgekehrt folgt:

(5.10) Satz. *Algorithmus* (5.9) *liefert einen maximalen Wald T (d. h. für jede Zusammenhangskomponente $G' = (V', E')$ von G ist $T \cap E'$ ein aufspannender Baum), dessen Gewicht $c(T)$ minimal ist. Ist G zusammenhängend, so ist T ein aufspannender Baum von G minimalen Gewichts $c(T)$.* △

Die Laufzeit von Algorithmus (5.7) bzw. (5.9) kann man wie folgt abschätzen. Mit den gängigen Sortierverfahren der Informatik (z. B. HEAP-SORT) kann man die Kanten von E in $O(k \log_2 k)$ bzw. $O(m \log_2 m)$ Schritten in der geforderten Weise ordnen. In Schritt 3 ruft man k- bzw. m-mal ein Unterprogramm auf, das überprüft, ob eine Kantenmenge einen Kreis besitzt oder nicht. Durch Benutzung geeigneter Datenstrukturen kann man einen derartigen Aufruf in höchstens $O(n)$ Schritten abarbeiten. Daraus folgt, dass Schritt 3 in höchstens $O(mn)$ Schritten ausgeführt werden kann. Dies ist auch die Gesamtlaufzeit des Verfahrens. Mit speziellen "Implementierungstricks" kann die Laufzeit von Schritt 3 auf $O(m + n \log n)$ gesenkt und damit die Gesamtlaufzeit sogar auf $O(m \log m)$ Schritte reduziert werden. In der Literatur wird Algorithmus (5.9) häufig *Kruskal-Algorithmus* genannt.

Einen gewichtsminimalen aufspannenden Baum kann man übrigens auch mit folgendem Verfahren finden.

(5.11) "Dualer" Greedy-Algorithmus.
Eingabe: Zusammenhängender Graph $G = (V, E)$ mit Kantengewichten $c(e)$ für alle $e \in E$.
Ausgabe: Aufspannender Baum $T \subseteq E$ minimalen Gewichts $c(T)$.

1. (Sortieren): Numeriere die m Kanten des Graphen G, so dass gilt
   $c(e_1) \geq c(e_2) \geq \ldots \geq c(e_m)$.
2. Setze $T := E$.
3. FOR $i = 1$ TO m DO:

Falls $T \setminus \{e_i\}$ zusammenhängend ist, setze $T := T \setminus \{e_i\}$.

4. Gib T aus. △

Der Korrektheitsbeweis für Algorithmus (5.11) bleibt einer Übungsaufgabe überlassen. Wie bereits erwähnt, gibt es eine Vielzahl weiterer Verfahren zur Bestimmung minimaler aufspannender Bäume. Ein gemeinsames Skelett für mehrere dieser Algorithmen kann wie folgt skizziert werden.

(5.12) Algorithmus META-MST.

Eingabe: Zusammenhängender Graph $G = (V, E)$ mit Kantengewichten $c(e)$ für alle $e \in E$.

Ausgabe: Aufspannender Baum T von G minimalen Gewichts. △

1. (Initialisierung):

   FOR ALL $i \in V$ DO:

   Setze $V_i := \{i\}$ und $T_i := \emptyset$.

2. DO $|V| - 1$ TIMES:
   (a) Wähle eine nicht-leere Menge V_i.
   (b) Wähle eine Kante $uv \in E$ mit $u \in V_i$, $v \in V \setminus V_i$ und $c(uv) \leq c(pq)$ für alle $pq \in E$ mit $p \in V_i$, $q \in V \setminus V_i$.
   (c) Bestimme j, so dass $v \in V_j$.
   (d) Setze $V_i := V_i \cup V_j$; $V_j := \emptyset$.
   (e) Setze $T_i := T_i \cup T_j \cup \{uv\}$; $T_j := \emptyset$.

3. Gib diejenige Kantenmenge T_i mit $T_i \neq \emptyset$ aus.

Algorithmus (5.9) ist ein Spezialfall von Algorithmus (5.12). Überlegen Sie sich wieso!

(5.13) Satz. *Algorithmus* (5.12) *bestimmt einen minimalen aufspannenden Baum.* △

Beweis. Wir zeigen durch Induktion über $p = |T_1| + \ldots + |T_n|$, dass G einen minimalen aufspannenden Baum T enthält mit $T_i \subseteq T$ für alle i. Ist $p = 0$, so ist nichts zu zeigen. Sei uv eine Kante, die bei einem Durchlauf von Schritt 2 in (b) gewählt wurde. Nach Induktionsvoraussetzung sind alle bisher bestimmten Mengen T_i in einem minimalen aufspannenden Baum T enthalten. Gilt $uv \in T$, so sind wir fertig. Ist $uv \notin T$, so enthält $T \cup \{uv\}$ einen Kreis. Folglich muss es eine Kante $rs \in T$ geben mit $r \in V_i$, $s \in V \setminus V_i$. Aufgrund unserer Wahl in (b) gilt $c(uv) \leq c(rs)$. Also ist $T := (T \setminus \{rs\}) \cup \{uv\}$ ebenfalls ein minimaler aufspannender Baum und der neue Baum T erfüllt unsere Bedingungen. Die Korrektheit des Algorithmus folgt aus dem Fall $p = n - 1$. □

Die Laufzeit des Algorithmus (5.12) hängt natürlich sehr stark von den Datenstrukturen ab, die man zur Ausführung des Schrittes 2 implementiert. Wir können an dieser Stelle nicht ausführlich auf Implementierungstechniken eingehen und verweisen hierzu auf Mehlhorn (1984), Vol 2, Kapitel IV, Abschnitt 8. Hier wird gezeigt, dass bei geeigneten Datenstrukturen eine Laufzeit von $O(n \log\log m)$ Schritten erreicht werden kann. Für planare Graphen ergibt sich sogar eine $O(n)$-Laufzeit.

Spanning-Tree-Algorithmen werden häufig als Unterprogramme zur Lösung von Travelling-Salesman- und anderen Problemen benötigt. Speziell ist hier eine Implementation dieser Algorithmen für vollständige Graphen erforderlich. Der nachfolgende Algorithmus lässt sich gerade für diesen Fall vollständiger Graphen einfach implementieren und hat sowohl empirisch wie theoretisch günstige Rechenzeiten aufzuweisen. Dieses Verfahren, das offensichtlich ebenfalls eine Spezialisierung von (5.12) ist, wird häufig PRIM-Algorithmus genannt.

(5.14) PRIM-Algorithmus.

Eingabe: Zusammenhängender Graph $G = (V, E)$ mit Kantengewichten $c(e)$ für alle $e \in E$.

Ausgabe: Aufspannender Baum T minimalen Gewichts $c(T)$.

1. Wähle $w \in V$ beliebig, setze $T := \emptyset$, $W := \{w\}$, $V := V \setminus \{w\}$.
2. Ist $V = \emptyset$, dann gib T aus und STOP.
3. Wähle eine Kante uv mit $u \in W$, $v \in V$, so dass $c(uv) = \min\{c(e) \mid e \in \delta(W)\}$.
4. Setze

$$\begin{aligned} T &:= T \cup \{uv\} \\ W &:= W \cup \{v\} \\ V &:= V \setminus \{v\} \end{aligned}$$

und gehe zu 2. △

Das PRIM-Verfahren hat, bei geeigneten Datenstrukturen, eine Laufzeit von $O(m + n \log n)$ und kann für den vollständigen Graphen K_n so implementiert werden, dass seine Laufzeit $O(n^2)$ beträgt, was offenbar bezüglich der Ordnung (also bis auf Multiplikation mit Konstanten und bis auf lineare Terme) bestmöglich ist, da ja jede der $\frac{n(n-1)}{2}$ Kanten mindestens einmal überprüft werden muss. Bereits bei dieser Überprüfung sind $O(n^2)$ Schritte notwendig. Nachfolgend finden Sie eine Liste eines PASCAL-Programms für Algorithmus (5.14), das für die Bestimmung minimaler aufspannender Bäume im K_n konzipiert ist.

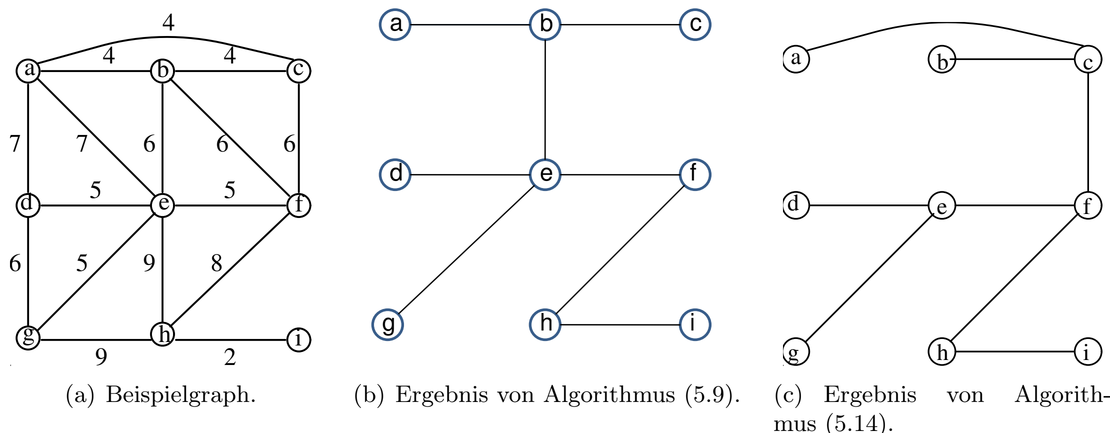
(5.15) Algorithmus PASCAL-Implementierung von Algorithmus (5.14)**.**

```
PROGRAM prim(inp, outp);
```

```
/**********************************************************************
 *                                                                    *
 *       Prim's Algorithm to Determine a Minimum Spanning Tree        *
 *       in a Complete Graph With n Nodes                             *
 *                                                                    *
 *       (G. Reinelt)                                   *
 *                                                                    *
 *--------------------------------------------------------------------*
 *                                                                    *
 *  Input:                                                            *
 *                                                                    *
 *  There are four ways to input the edge weights of the complete     *
 *  graph K_n. In any case we assume that first two numbers are given:   *
 *                                                                    *
 *      n = number of nodes                                           *
 *   mode = input mode                                                *
 *                                                                    *
 *  Mode specifies the input mode for the edge weights. All edge weights  *
 *  have to be integers.                                              *
 *                                                                    *
 *  Mode=0 : The full matrix of edge weights is given. The entries are   *
 *           stored row by row. The lower diagonal and the diagonal   *
 *           entries are ignored.                                     *
 *                                                                    *
 *  Mode=1 : The matrix of edge weights is given as upper triangular  *
 *           matrix. The entries are stored row by row.               *
 *                                                                    *
 *  Mode=2 : The matrix of edge weights is given as lower triangular  *
 *           matrix. The entries are stored row by row.               *
 *                                                                    *
 *  Mode=3 : The edge weights are given in an edge list of the        *
 *           form: 1st endnode, 2nd endnode, edge weight. Edges which *
 *           are not present are assumed to have 'infinite' weight.   *
 *           The input is ended if the first endnode is less than 1.  *
 *                                                                    *
 **********************************************************************/

CONST  max_n  = 100;             { maximum number of nodes }
       max_n2 = 4950;            { max_n choose 2 = number of edges of K_n}
                                 { to process larger graphs only max_n and
                                   max-n2 have to be changed }
       inf    = maxint;          { infinity }

TYPE   arrn2 = ARRAY[1..max_n2] OF integer;
       arrn  = ARRAY[1..max_n]  OF integer;

VAR    i, j,
       mode,                     { input mode of weights:
                                     0  : full matrix
                                     1  : upper triangular matrix
                                     2  : lower triangular matrix
                                     3  : edge list               }
       min,                      { minimum distance }
       ind,                      { index of entering edge }
       newnode,                  { entering tree node }
       t1, t2,                   { entering tree edge }
       outnodes,                 { number of nodes not in tree }
       c,
       weight,                   { weight of tree }
       nchoose2,
       n          : integer;     { number of nodes }
       w          : arrn2;       { vector of weights }
       dope,                     { dope vector for index calculations }
       dist,                     { shortest distances to non-tree nodes }
       in_t,                     { in-tree node of shortest edge }
       out_t      : arrn;        { out-tree node of shortest edge }
                                 { minimum tree is also stored in in_t & out_t}
       connected : boolean;      { true <=> input graph is connected? }
```

```
       inp,                     { input file }
       outp      : text;        { output file }

BEGIN {MAIN PROGRAM}

{====== Input of complete graph =====}

reset(inp);
rewrite(outp);

{- number of nodes -}
writeln(outp,'Enter number of nodes:');
read(inp,n);

IF (n<1) OR (n>max_n) THEN
 BEGIN
   writeln(outp,'Number of nodes too large or not positive!');
   HALT;
 END;

{- initialize dope vector -}
nchoose2 := (n * (n-1)) DIV 2;
FOR i:=1 TO nchoose2 DO
  w[i] := inf;
dope[1] := -1;
FOR i:=2 TO n DO
  dope[i] := dope[i-1] + n - i;

{- input mode -}
writeln(outp,'Enter input mode:');
read(inp,mode);

{- edge weights -}
CASE mode OF
   0 : {* full matrix *}
       BEGIN
         FOR i:=1 TO n DO
           FOR j:=1 TO n DO
             BEGIN
               read(inp, c);
               IF i<j THEN w[dope[i]+j] := c;
             END
       END;
   1 : {* upper triangular matrix *}
       BEGIN
         FOR i:=1 TO nchoose2 DO
           read(inp, w[i]);
       END;
   2 : {* lower triangular matrix *}
       BEGIN
         FOR i:=2 TO n DO
           FOR j:=1 TO i-1 DO
             BEGIN
               read(inp, c);
               w[dope[j]+i] := c;
             END;
       END;
   3 : {* edge list *}
       BEGIN
         read(inp, i, j, c);
         WHILE (i>0) DO
          BEGIN
            IF (i<1) OR (i>n) OR
               (j<1) OR (j>n)
             THEN BEGIN
                    writeln(outp,'Input error, node out of range!');
                    HALT;
                  END;
            IF i<j
```

```
              THEN w[dope[i]+j] := c
              ELSE w[dope[j]+i] := c;
            read(inp, i, j, c);
          END;
       END;
 ELSE: {* invalid mode *}
       BEGIN
         writeln(outp,'Invalid input mode!');
         HALT;
       END;
 END;{OF CASE}

{====== Initialization ======}

connected := true;
outnodes  := n-1;
weight    := 0;
FOR i:=1 TO outnodes DO
 BEGIN
   in_t[i]  := 1;
   out_t[i] := i+1;
   dist[i]  := w[i];
 END;

{====== Prim's Algorithm ======}

WHILE (outnodes > 1) AND connected DO
 BEGIN
   {- determine entering node -}
   min := inf;
   ind := 0;
   FOR i:=1 TO outnodes DO
     IF dist[i] < min
      THEN BEGIN
             min := dist[i];
             ind := i;
           END;

   IF ind = 0
    THEN connected := false
    ELSE BEGIN
           {- augment tree -}
           weight          := weight + min;
           newnode         := out_t[ind];
           t1              := in_t[ind];
           t2              := out_t[ind];
           c               := dist[ind];
           in_t[ind]       := in_t[outnodes];
           out_t[ind]      := out_t[outnodes];
           dist[ind]       := dist[outnodes];
           in_t[outnodes]  := t1;
           out_t[outnodes] := t2;
           dist[outnodes]  := c;
           outnodes        := outnodes - 1;
           {- update dist[] and in_t[] -}
           FOR i:=1 TO outnodes DO
            BEGIN
              IF newnode < out_t[i]
                THEN c := w[dope[newnode]+out_t[i]]
                ELSE c := w[dope[out_t[i]]+newnode];
              IF c < dist[i]
              THEN BEGIN
                     in_t[i] := newnode;
                     dist[i] := c;
                   END;
            END;
         END;
 END;
{- insert the last edge -}
```



(a) Beispielgraph. (b) Ergebnis von Algorithmus (5.9). (c) Ergebnis von Algorithmus (5.14).

Abbildung 5.1: Ein Beispielgraph und die MSTs, die sich als Ergebnis der Algorithmen (5.9) und (5.14) ergeben.

```
IF connected
 THEN IF dist[1]>=inf
       THEN connected := false
       ELSE weight    := weight + dist[1];

{====== Output of minimum spanning tree ======}

writeln(outp);
IF NOT connected
 THEN writeln(outp,'The graph is disconnected.')
 ELSE BEGIN
        writeln(outp,'Minimum spanning tree:');
        writeln(outp,'======================');
        writeln(outp);
        FOR i:=n-1 DOWNTO 1 DO
          writeln(outp, in_t[i]:5, ' - ', out_t[i]:3,
                  '  (', dist[i]:1,')');
        writeln(outp);
        writeln(outp,'Weight: ', weight:6);
        writeln(outp);
      END;
 END.
```

Wir wollen nun noch ein Beispiel angeben, das die Vorgehensweise der Algorithmen (5.9), (5.11) und (5.14) verdeutlicht.

(5.16) Beispiel. Wir betrachten den in Abbildung 5.1(a) dargestellten Graphen. Wir wenden Algorithmus (5.9) an. Zunächst sortieren wir die Kanten in nicht absteigender Reihenfolge hi, bc, ab, ac, de, ef, eg, be, bf, cf, dg, ad, ae, hf, he, hg. In Schritt 3 von (5.9) werden die in der Abbildung 5.1(b) gezeichneten Kanten ausgewählt. Den Prim-Algorithmus (5.14) starten wir mit dem Knoten $w = a$. Es ergibt sich der in Abbildung 5.1(c) gezeichnete minimale aufspannende Baum. △

Wie Beispiel (5.16) zeigt, muss ein minimaler Baum nicht eindeutig bestimmt sein. Überlegen Sie sich bitte, wie man feststellen kann, ob ein minimaler aufspannender Baum eindeutig ist.

Weitergehende Informationen über Branchings und Aboreszenzen (sowie Wälder und Bäume) finden sich im Buch Schrijver (2003) in Part V.

Im Internet finden sich viele „Graph Libraries“ oder „Algorithm Repositories“, in denen fertig implementierte Algorithmen angeboten werden, die das „Minimum Spanning Tree“- oder „Maximum Weighted Branching“-Problem lösen. Einige der Algorithmensammlungen sind kommerziell (und kosten Geld), einige sind frei verfügbar, einige interaktiv abrufbar und viele haben Visualisierungskomponenten. Die Halbwertzeit der Webseiten ist häufig nicht besonders hoch. Es folgen einige Webseiten, die Baum-, Branching- und viele andere Graphenalgorithmen anbieten:

- COIN-OR::LEMON 1.1: `http://lemon.cs.elte.hu`
- QuickGraph: `http://quickgraph.codeplex.com`
- The Stony Brook Algorithm Repository: `http://www.cs.sunysb.edu/~algorith/`
- LEDA: `http://www.algorithmic-solutions.com/leda/index.htm`

5.3 Kürzeste Wege

Wir wollen uns nun mit der Aufgabe beschäftigen, in einem Digraphen mit Bogengewichten kürzeste gerichtete Wege zu finden. Wir werden Algorithmen vorstellen, die kürzeste Wege von einem Knoten zu einem anderen oder zu allen anderen oder kürzeste Wege zwischen zwei Knoten finden. Wir beschränken uns auf Digraphen, da derartige Probleme in ungerichteten Graphen auf einfache Weise auf gerichtete Probleme reduziert werden können. Denn ist ein Graph $G = (V, E)$ mit Kantenlängen $c(e) \geq 0$ für alle $e \in E$ gegeben, so ordnen wir diesem Graphen den Digraphen $D = (V, A)$ mit $A = \{(i, j), (j, i) \mid ij \in E\}$ und $c((i, j)) := c((j, i)) := c(ij)$ zu. Den (ungerichteten) $[u, v]$-Wegen in G entsprechen dann die gerichteten (u, v)-Wege bzw. (v, u)-Wege in D und umgekehrt. Einander entsprechende Wege in G und D haben nach Definition gleiche Längen. Also liefert uns ein kürzester (u, v)-Weg (oder ein kürzester (v, u)-Weg) in D einen kürzesten $[u, v]$-Weg in G.

Kürzeste-Wege-Probleme spielen in der kombinatorischen Optimierung eine große Rolle. Es ist daher nicht überraschend, dass es zu diesem Problemkreis eine außerordentlich umfangreiche Literatur und sehr viele Lösungsvorschläge gibt. Wenn man dann noch Variationen hinzunimmt wie: Berechnung längster Wege oder zuverlässiger Wege, von Wegen maximaler Kapazität, der k kürzesten Wege, von Wegen mit gerader oder ungerader Bogenzahl etc., so liefert das den Stoff einer gesamten Vorlesung. Wir wollen in dieser Vorlesung lediglich drei Algorithmen (für unterschiedliche Spezialfälle) behandeln. Der Leser, der sich für umfassendere Darstellungen interessiert, sei auf die Bücher Ahuja et al. (1993), Krumke and Noltemeier (2005), Lawler (1976), Mehlhorn (1984), Domschke (1972), Schrijver (2003), Syslo et al. (1983) verwiesen. Es werden derzeit immer noch

neue Algorithmen oder Modifikationen bekannter Algorithmen entdeckt, die aus theoretischer oder praktischer Sicht schneller als die bekannten Verfahren sind oder sonstige Vorzüge haben.

Es gibt keinen Algorithmus zur Bestimmung eines kürzesten (s,t)-Weges, der nicht (zumindest implizit) auch alle übrigen kürzesten Wege von s nach v, $s \neq v \neq t$, berechnet. Die Algorithmen für Kürzeste-Wege-Probleme kann man in zwei Kategorien einteilen, und zwar solche, die negative Bogenlängen zulassen, und solche, die nur nichtnegative Bogenlängen behandeln können. Von jedem der beiden Typen stellen wir einen Vertreter vor. Ferner wollen wir noch einen Algorithmus behandeln, der kürzeste Wege zwischen allen Knoten berechnet.

Vermutlich haben sich die Menschen schon in grauer Vorzeit mit der Bestimmung kürzester Wege beschäftigt, um z.B. Transporte zu vereinfachen, den Handel zu erleichtern etc. Mathematik – im heutigen Sinne – wurde dabei sicherlich nicht verwendet. Eines der ältesten (uns bekannten) Wegeprobleme der (belletristischen) Literatur kommt aus einer klassischen Quelle: Friedrich Schillers (1759–1805) Schauspiel "Wilhelm Tell". Dieser konnte bereits 1291 nicht nur gut schießen, sondern auch optimieren. Und nur mit dieser Kombination konnte er die Schweiz befreien! Tell befindet sich nach dem Apfelschuss am Ufer des Vierwaldstätter Sees unweit des Ortes Altdorf. Er muss unbedingt vor dem Reichsvogt Hermann Geßler die Hohle Gasse in Küßnacht erreichen, siehe Abbildung 5.2(b).

Schiller berichtet:

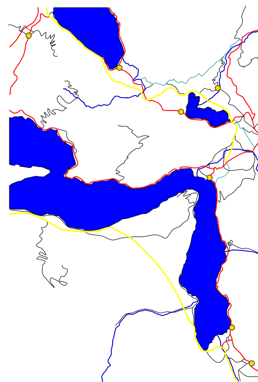
Tell. Nennt mir den nächsten Weg nach Arth und Küßnacht
Fischer. Die offne Straße zieht sich über Steinen
Den kürzern Weg und heimlichern
Kann Euch mein Knabe über Lowerz führen.
Tell (gibt ihm die Hand). Gott lohn Euch Eure Guttat. Lebet wohl.

Der Fischer löst für Tell in dieser Szene offensichtlich ein graphentheoretisches Optimierungsproblem. In einem Graphen (Wegenetz am Vierwaldstätter See) mit Kantenlängen (Reisezeit) soll der kürzeste Weg zwischen zwei vorgegebenen Punkten (Altdorf und Küßnacht) bestimmt werden. Tell behandelt sogar eine kompliziertere Variante mit einer zusätzlichen Nebenbedingung: Die Summe von „Verhaftungskoeffizienten" muss unterhalb eines sicheren Grenzwertes bleiben. Man kann dies auch als multikriterielles Optimierungsproblem auffassen (Weglänge und Sicherheit gleichzeitig optimieren). Dies ist ein Aufgabentyp, den wir auch heute noch nicht gut beherrschen. (In der Vorlesung wird mehr dazu berichtet).

5.3.1 Ein Startknoten, nichtnegative Gewichte

Das Verfahren, das wir nun darstellen wollen, ist mehrfach entdeckt worden. Es wird allgemein nach Dijkstra (1959) benannt. Wir gehen davon aus, dass ein Digraph $D = (V, A)$ mit "Gewichten" bzw. "Längen" oder "Entfernungen" $c(a) \geq 0$ für alle $a \in A$

(a) F. Schiller.


(b) Vierwaldstätter See.

(c) W. Tell.

Abbildung 5.2: Mathematik in der Belletristik: Schillers Schauspiel „Wilhelm Tell“.

gegeben ist. Ferner seien ein Startknoten s und möglicherweise ein Endknoten t gegeben. Das Verfahren findet einen kürzesten gerichteten Weg von s zu allen anderen Knoten bzw. einen kürzesten (s, t)-Weg.

Der Algorithmus wird häufig als *Markierungsmethode* bezeichnet. (Warum, wird aus dem Weiteren klar.) Seine Idee kann man wie folgt beschreiben.

Wir beginnen im Startknoten s, markieren s und ordnen s die *permanente Distanz* Null (= Länge des kürzesten Weges von s zu sich selbst) zu. Alle übrigen Knoten v seien unmarkiert, und wir ordnen ihnen als *temporäre Distanz* (= Länge des kürzesten bisher gefundenen (s, v)-Weges) entweder $+\infty$ oder die Länge des Bogens (s, v), falls dieser in D existiert, zu. Der unmarkierte Knoten mit der kleinsten temporären Distanz ist dann der Knoten, der am nächsten zu s liegt. Nennen wir den Knoten u. Da alle Bogenlängen nicht-negativ sind, ist der Bogen (s, u) der küzeste Weg von s nach u. Wir markieren daher u und erklären die temporäre Distanz von u als *permanent*, weil wir den (global) kürzesten (s, u)-Weg gefunden haben. Nun bestimmen wir alle Nachfolger v von u und vergleichen die temporäre Distanz von v mit der permanenten Distanz von u plus der Länge des Bogens (u, v). Ist diese Summe kleiner als die bisherige temporäre Distanz, wird sie die neue temporäre Distanz, weil der bisher bekannte Weg von s nach v länger ist als der Weg von s über u nach v. Wir wählen nun wieder eine kleinste der temporären Distanzen, erklären sie als permanent, da der bisher gefundene Weg durch Umwege über andere Knoten nicht verkürzt werden kann, markieren den zugehörigen Knoten und fahren so fort bis entweder alle Knoten oder der gesuchte Endknoten t markiert sind. Etwas formaler kann man diesen Algorithmus wie folgt aufschreiben.

(5.17) DIJKSTRA-Algorithmus.

Eingabe: Digraph $D = (V, A)$, Gewichte $c(a) \geq 0$ für alle $a \in A$, ein Knoten $s \in V$ (und möglicherweise ein Knoten $t \in V \setminus \{s\}$).

Ausgabe: Kürzeste gerichtete Wege von s nach v für alle $v \in V$ und ihre Länge (bzw. ein kürzester (s, t)-Weg).

Datenstrukturen: DIST(v) Länge des kürzesten (s,v)-Weges
VOR(v) Vorgänger von v im kürzesten (s,v)-Weg

1. Setze:

$$\begin{aligned} \text{DIST}(s) &:= 0 \\ \text{DIST}(v) &:= c((s,v)) && \forall\, v \in V \text{ mit } (s,v) \in A \\ \text{DIST}(v) &:= +\infty && \forall\, v \in V \text{ mit } (s,v) \notin A \\ \text{VOR}(v) &:= s && \forall\, v \in V \setminus \{s\} \end{aligned}$$

   Markiere s, alle übrigen Knoten seien unmarkiert.

2. Bestimme einen unmarkierten Knoten u, so dass
   $\text{DIST}(u) = \min\{\text{DIST}(v) \mid v \text{ unmarkiert}\}$. Markiere u.
   (Falls $u = t$, gehe zu 5.)

3. Für alle unmarkierten Knoten v mit $(u,v) \in A$ führe aus:

   Falls $\text{DIST}(v) > \text{DIST}(u) + c((u,v))$ setze:
   $\text{DIST}(v) := \text{DIST}(u) + c((u,v))$ und $\text{VOR}(v) := u$.

4. Sind noch nicht alle Knoten markiert, gehe zu 2.

5. Für alle markierten Knoten v ist DIST(v) die Länge eines kürzesten (s,v)-Weges. Falls v markiert ist und $\text{DIST}(v) < +\infty$, so ist VOR(v) der Vorgänger von v in einem kürzesten (s,v)-Weg, d. h. durch Rückwärtsgehen bis s kann ein kürzester (s,v)-Weg bestimmt werden. (Brechen wir das Verfahren nicht in Schritt 2 ab und gilt am Ende $\text{DIST}(v) = +\infty$, so heißt das, dass es in D keinen (s,v)-Weg gibt.)△

Zur Notationsvereinfachung für den nachfolgenden Beweis bezeichnen wir mit $\text{DIST}_k(v)$ den Wert der in (5.17) berechneten Distanzfunktion nach dem k-ten Durchlaufen der Schritte 2, 3 und 4. Für den DIJKSTRA-Algorithmus gilt aufgrund der Auswahlvorschrift nach der k-ten Markierungsphase Folgendes: Sind die Knoten in der Reihenfolge $v_1, v_2, \ldots, v_k$ markiert worden, so gilt $\text{DIST}_k(v_1) \leq \ldots \leq \text{DIST}_k(v_k) \leq \text{DIST}_k(v)$ für alle bisher unmarkierten Knoten v.

(5.18) Satz. *Der Dijkstra-Algorithmus arbeitet korrekt.* △

Beweis. Wir zeigen durch Induktion über die Anzahl k markierter Knoten Folgendes: Ist v markiert, so enthält $\text{DIST}_k(v)$ die Länge eines kürzesten (s,v)-Weges; ist v unmarkiert, so enthält $\text{DIST}_k(v)$ die Länge eines kürzesten (s,v)-Weges, wobei als innere Knoten nur markierte Knoten zugelassen sind. (Falls $\text{DIST}_k(v) = +\infty$, so wird dies als Nichtexistenz eines (s,v)-Weges bzw. eines (s,v)-Weges über markierte innere Knoten interpretiert). Hieraus folgt offenbar die Behauptung.

Ist nur ein Knoten (also s) markiert, so ist unsere Behauptung aufgrund der Definition in Schritt 1 korrekt. Wir nehmen nun an, dass die Behauptung richtig ist für k markierte Knoten und dass das Verfahren in Schritt 2 einen $(k+1)$-sten Knoten, sagen wir u, markiert und Schritt 3 durchlaufen hat. Nach Induktionsvoraussetzung ist

$\text{DIST}_k(u)$ die Länge eines kürzesten (s,u)-Weges, der als innere Knoten nur die ersten k markierten Knoten benutzen darf. Gäbe es einen kürzeren gerichteten Weg, sagen wir P, von s nach u, so müsste dieser einen Bogen von einem markierten Knoten zu einem bisher nicht markierten Knoten enthalten. Sei (v,w) der erste derartige Bogen auf dem Weg P. Der Teilweg $\overline{P}$ des Weges P von s nach w ist also ein (s,w)-Weg, dessen innere Knoten markiert sind. Folglich gilt nach Induktionsvoraussetzung $\text{DIST}_{k+1}(w) \leq c(\overline{P})$. Aus $\text{DIST}_{k+1}(u) \leq \text{DIST}_{k+1}(w)$ und der Nichtnegativität der Bogenlängen folgt $\text{DIST}_{k+1}(u) \leq c(\overline{P}) \leq c(P)$, ein Widerspruch.

Es bleibt noch zu zeigen, dass für die derzeit unmarkierten Knoten v der Wert $\text{DIST}_{k+1}(v)$ die Länge eines kürzesten (s,v)-Weges ist, der nur markierte innere Knoten enthalten darf. Im Update-Schritt 3 wird offenbar die Länge eines (s,v)-Weges über markierte Knoten verschieden von u verglichen mit der Länge eines (s,v)-Weges über markierte Knoten, der als vorletzten Knoten den Knoten u enthält. Angenommen es gibt einen (s,v)-Weg P über markierte Knoten (inclusive u), dessen vorletzter Knoten w verschieden von u ist und dessen Länge geringer ist als die kürzeste Länge der oben betrachteten Wege. Da $\text{DIST}_{k+1}(w)$ die Länge eines kürzesten (s,w)-Weges ist und es einen solchen, sagen wir P', gibt, der nur markierte Knoten enthält, die verschieden von u sind (w wurde vor u markiert), kann der (s,w)-Weg auf P nicht kürzer als P' sein, also ist P nicht kürzer als die Länge von $P' \cup \{(w,v)\}$. Widerspruch. □

In der Datenstruktur VOR merken wir uns zu jedem Knoten v seinen Vorgänger in einem kürzesten (s,v)-Weg. Einen kürzesten (s,v)-Weg erhält man also in umgekehrter Reihenfolge durch die Knotenfolge

$$v, \text{VOR}(v), \text{VOR}(\text{VOR}(v)), \ldots, \text{VOR}(\text{VOR}(\ldots \text{VOR}(v) \ldots)).$$

Durch VOR ist offenbar eine Arboreszenz mit Wurzel s in D definiert. Daraus folgt sofort:

(5.19) Satz. *Sei $D = (V,A)$ ein Digraph mit nichtnegativen Bogengewichten und $s \in V$, dann gibt es eine Arboreszenz B mit Wurzel s, so dass für jeden Knoten $v \in V$, für den es einen (s,v)-Weg in D gibt, der (eindeutig bestimmte) gerichtete Weg in B von s nach v ein kürzester (s,v)-Weg ist.* △

An dieser Stelle sei darauf hingewiesen, dass der PRIM-Algorithmus (5.14) und der DIJKSTRA-Algorithmus (5.17) (im Wesentlichen) identische Algorithmen sind. Sie unterscheiden sich lediglich bezüglich einer Gewichtstransformation. In Schritt 3 von (5.14) wird $\min\{c(e) \mid e \in \delta(W)\}$ gesucht, in Schritt 2 von (5.17) wird auch ein derartiges Minimum gesucht, jedoch sind vorher in Schritt 3 die Gewichte der Bögen des Schnittes modifiziert worden.

Den DIJKSTRA-Algorithmus kann man ohne Schwierigkeiten so implementieren, dass seine Laufzeit $O(|V|^2)$ beträgt. Bei Digraphen mit geringer Bogenzahl kann die Laufzeit durch Benutzung spezieller Datenstrukturen beschleunigt werden, siehe hierzu z.B. Ahuja et al. (1993) oder Schrijver (2003).

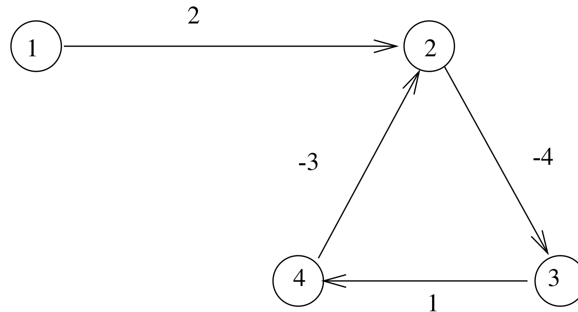


Abbildung 5.3: Digraph mit negativen Kantengewichten

5.3.2 Ein Startknoten, beliebige Gewichte

Das Problem, einen kürzesten Weg in einem Digraphen mit beliebigen Bogengewichten zu bestimmen, ist trivialerweise äquivalent zum Problem, einen längsten Weg in einem Digraphen mit beliebigen Bogengewichten zu finden. Gäbe es für das letztere Problem einen polynomialen Algorithmus, so könnte man in polynomialer Zeit entscheiden, ob ein Digraph einen gerichteten hamiltonschen Weg enthält. Dieses Problem ist aber $\mathcal{NP}$-vollständig, also ist das Kürzester-Weg-Problem für beliebige Gewichte $\mathcal{NP}$-schwer.

Andererseits kann man dennoch in beliebig gewichteten Digraphen kürzeste Wege finden, wenn die negativen Gewichte "gut verteilt" sind oder der Digraph bestimmte Eigenschaften hat. Der DIJKSTRA-Algorithmus funktioniert bei negativen Gewichten nicht (im Induktionsschritt des Beweises von (5.18) wurde von der Nichtnegativität explizit Gebrauch gemacht). Wir wollen nun auf ein Verfahren eingehen, das unabhängig voneinander von Moore (1959) und Bellman (1958) vorgeschlagen wurde. Zu diesem Verfahren gibt es eine Vielzahl von Verbesserungsvorschlägen (siehe hierzu z. B. Lawler (1976), Syslo et al. (1983), Glover et al. (1985)).
Die Idee hinter diesem Verfahren lässt sich wie folgt beschreiben. Wir wollen vom Startknoten s aus zu allen anderen Knoten v einen kürzesten (s, v)-Weg bestimmen. Wir initialisieren DIST(v) wieder mit $+\infty$ oder mit $c((s, v))$ (DIST(v) enthält also die Länge des kürzesten zur Zeit bekannten (s, v)-Weges mit einer bestimmten Eigenschaft) und setzen wie in (5.17) VOR$(v) = s$. Nun versuchen wir, die Distanzen DIST(v) sukzessive zu reduzieren. Finden wir einen Bogen (u, v) mit DIST$(u) + c((u, v)) <$ DIST(v), so setzen wir DIST$(v) :=$ DIST$(u) + c((u, v))$ und VOR$(v) := u$. Wir führen diese Iteration so lange fort, bis kein Wert DIST(u) mehr reduziert werden kann. Die verschiedenen Versionen des Moore-Bellman-Algorithmus unterscheiden sich durch die Art, wie diese Basisiteration ausgeführt wird (d. h. in welcher Reihenfolge die Knoten und Bögen (u. U. mehrfach) abgearbeitet werden).

Wir wollen uns zunächst überlegen, unter welchen Umständen der MOORE-BELLMAN-Algorithmus bei allgemeinen Gewichten nicht funktioniert. Wir betrachten den Digraphen aus Abbildung 5.3 mit den dort eingetragenen Gewichten. Wir initialisieren mit DIST$(1) = 0$, DIST$(2) = 2$, DIST$(3) =$ DIST$(4) = +\infty$, VOR$(i) = 1$, $i = 1, 2, 3, 4$. Wir stellen fest, dass DIST$(3) >$ DIST$(2) + c((2, 3)) = -2$, und setzen DIST$(3) = -2$,

VOR(3) = 2. Wir setzen analog $\text{DIST}(4) = \text{DIST}(3) + c((3,4)) = -1$, $\text{VOR}(4) = 3$. Nun gilt $\text{DIST}(2) = 2 > \text{DIST}(4) + c((4,2)) = -4$, also setzen wir $\text{DIST}(2) = -4$. Was ist passiert? Der kürzeste Weg von 1 nach 2 besteht offensichtlich aus dem Bogen (1,2) und hat die Länge 2. Wenden wir unser Verfahren an, so stellen wir fest, dass wir von 1 nach 4 mit der Weglänge -1 gelangen können. Dieser Weg enthält den Knoten 2. Aber nun können wir von 4 nach 2 zurückgehen, und unsere gerichtete Kette von 1 nach 2, nach 3, nach 4 und wieder zu 2 hat eine geringere Länge als der direkte Weg von 1 nach 2. Der Grund für diese Wegverkürzung liegt darin, dass wir einen Kreis, hier den Kreis (2,3,4), entdeckt haben, dessen Gesamtlänge negativ ist. Laufen wir nun noch einmal durch diesen Kreis, so können wir die "Weglänge" noch weiter verkürzen, d. h. unser Verfahren wird eine immer kleinere "Weglänge" produzieren und nicht enden. Nennen wir einen gerichteten Kreis C *negativ*, wenn sein Gewicht $c(C)$ negativ ist, so zeigt die obige Überlegung, dass negative Kreise in einem Digraphen zum Scheitern des Verfahrens führen. Hat ein Digraph überhaupt keinen gerichteten Kreis, ist er also azyklisch, so gibt es insbesondere keine negativen Kreise, und das MOORE-BELLMAN-Verfahren funktioniert.

(5.20) MOORE-BELLMAN-Algorithmus für azyklische Digraphen.

Eingabe: Azyklischer Digraph $D = (V, A)$, Gewichte $c(a)$ für alle $a \in A$ (beliebige negative Gewichte sind zugelassen), ein Knoten $s \in V$.

Ausgabe: Für jeden Knoten $v \in V$ ein kürzester (s, v)-Weg und seine Länge.

Datenstrukturen: DIST(v), VOR(v) für alle $v \in V$. O. B. d. A. nehmen wir an, dass $V = \{1, 2, \dots, n\}$ gilt und alle Bögen die Form (u, v) mit $u < v$ haben.

1. Setze:

$$\begin{aligned} \text{DIST}(v) &:= \begin{cases} 0 & \text{falls } s = v \\ +\infty & \text{falls } s \neq v \text{ und } (s,v) \notin A \\ c((s,v)) & \text{andernfalls} \end{cases} \\ \text{VOR}(v) &:= s. \end{aligned}$$

2. DO $v = s + 2$ TO n:

3. DO $u = s + 1$ TO $v - 1$:

   Falls $(u, v) \in A$ und $\text{DIST}(u) + c((u,v)) < \text{DIST}(v)$ setze

   $\text{DIST}(v) := \text{DIST}(u) + c((u,v))$ und $\text{VOR}(v) := u$.

   END 3.

   END 2.

4. Falls $\text{DIST}(v) < +\infty$, so enthält DIST(v) die Länge des kürzesten gerichteten Weges von s nach v, und aus VOR kann ein kürzester (s, v)-Weg entnommen werden. Falls $\text{DIST}(v) = +\infty$, so existiert in D kein (s, v)-Weg. △

(5.21) Satz. *Algorithmus* (5.20) *funktioniert für beliebige azyklische Digraphen D und beliebige Bogengewichte.* △

Beweis. Nach Voraussetzung haben alle Bögen in D die Form (u, v) mit $u < v$. Folglich gibt es in D keinen (s, v)-Weg für $v < s$. Nach Definition ist die Länge eines (s, s)-Weges gleich Null. Ferner enthält jeder (s, v)-Weg mit $v > s$ nur innere Knoten u mit $s < u < v$. Es gibt höchstens einen $(s, s+1)$-Weg, nämlich den Bogen $(s, s+1)$, falls er in D existiert, also enthält $\text{DIST}(v)$ für $1 \le v \le s + 1$ die Länge eines kürzesten (s, v)-Weges in D.

Ist $v > s + 1$, so folgt durch Induktion über die Schleifenindizes der Schleife 2, dass $\text{DIST}(u)$ die Länge eines kürzesten (s, u)-Weges für $1 \le u \le v$ enthält. Aus formalen Gründen lassen wir Schleife 2 mit $v = s + 1$ beginnen. Dadurch wird kein Wert $\text{DIST}(u)$ in Schritt 3 geändert. Für $v = s + 1$ ist somit nach obiger Bemerkung die Behauptung korrekt. Sei also die Behauptung für v richtig und betrachten wir den Knoten $v+1$. Nach Induktionsvoraussetzung enthält $\text{DIST}(u)$, $1 \le u \le v$, die Länge eines kürzesten (s, u)-Weges. Da ein $(s, v + 1)$-Weg entweder von s direkt nach $v + 1$ führt (das Gewicht dieses Bogens ist gegenwärtig in $\text{DIST}(v + 1)$ gespeichert) oder zunächst zu Zwischenknoten u im Intervall $s < u \le v$ und dann auf einen Bogen nach $v + 1$ führt, ist also die Länge des kürzesten $(s, v + 1)$-Weges gegeben durch das Minimum der folgenden beiden Werte:

$$c((s, v + 1)) = \text{DIST}(v + 1),$$

$$\text{Länge des kürzesten } (s, u)\text{-Weges } + c((u, v + 1)) = \text{DIST}(u) + c((u, v + 1)).$$

Dieses Minimum wird offenbar bei Ausführung der Schleife 3 für $v+1$ berechnet. Daraus folgt die Behauptung. □

Da das Verfahren (5.20) im wesentlichen aus zwei Schleifen besteht, die beide über maximal $n - 2$ Indizes laufen, ist die Laufzeit des Verfahrens $O(n^2)$.

Wir geben nun den MOORE-BELLMAN-Algorithmus für beliebige Digraphen in zwei verschiedenen Varianten an:

(5.22) MOORE-BELLMAN-Algorithmus.

Eingabe: Digraph $D = (V, A)$, Gewichte $c(a)$ für alle $a \in A$ (können auch negativ sein), ein Knoten $s \in V$.

Ausgabe: Für jeden Knoten $v \in V$ ein kürzester (s, v)-Weg und seine Länge. Korrektheit des Output ist nur dann garantiert, wenn D keinen negativen Kreis enthält.

Datenstrukturen: $\text{DIST}(v)$, $\text{VOR}(v)$ für alle $v \in V$ (wie in Algorithmus (5.17))

1. Setze:

$$\begin{aligned} \text{DIST}(s) &:= 0 \\ \text{DIST}(v) &:= c((s, v)) && \text{falls } (s, v) \in A \\ \text{DIST}(v) &:= \infty && \text{sonst} \\ \text{VOR}(v) &:= s && \forall\, v \in V. \end{aligned}$$

YEN-VARIANTE Wir nehmen hier zur Vereinfachung der Darstellung o. B. d. A. an, dass $V = \{1, \ldots, n\}$ und $s = 1$ gilt.

2. DO $m = 0$ TO $n - 2$:
3. Falls m gerade: DO $v = 2$ TO n:

4. DO $u = 1$ TO $v - 1$:
   Falls $(u, v) \in A$ und $\text{DIST}(u) + c((u, v)) < \text{DIST}(v)$,
   setze $\text{DIST}(v) := \text{DIST}(u) + c((u, v))$ und $\text{VOR}(v) := u$.
   END 4.
   END 3.

5. Falls m ungerade: DO $v = n - 1$ TO 1 BY -1:

6. DO $u = n$ TO $v + 1$ BY -1:
   Falls $(u, v) \in A$ und $\text{DIST}(u) + c((u, v)) < \text{DIST}(v)$,
   setze $\text{DIST}(v) := \text{DIST}(u) + c((u, v))$
   und $\text{VOR}(v) := u$.
   END 6.
   END 5.
   END 2.
   Gehe zu 7.

D'ESOPO-PAPE-VARIANTE

$2'$. Initialisiere eine Schlange Q und setze s in Q.

$3'$. Hole das erste Element aus der Schlange, sagen wir u.

$4'$. Für alle Bögen (u, v), die in u beginnen, führe aus:

$5'$. Falls $\text{DIST}(u) + c((u, v)) < \text{DIST}(v)$

a) setze $\text{DIST}(v) := \text{DIST}(u) + c((u, v))$ und $\text{VOR}(v) := u$,

b) Falls v noch nicht in Q war, setze v an das Ende von Q,

c) Falls v schon in Q war, aber gegenwärtig nicht in Q ist, setze v an den Anfang von Q.

END $4'$.

$6'$. Ist die Schlange nicht leer, gehe zu $3'$, andernfalls zu 7.

7. Falls $\text{DIST}(v) < +\infty$, so enthält $\text{DIST}(v)$ die Länge eines kürzesten (s, v)-Weges, und aus $\text{VOR}(v)$ kann wie üblich ein kürzester (s, v)-Weg rekonstruiert werden. Ist $\text{DIST}(v) = +\infty$, so gibt es in D keinen (s, v)-Weg. △

Es ist intuitiv einsichtig, dass das MOORE-BELLMAN-Verfahren ein korrektes Ergebnis liefert, falls keine negativen Kreise vorliegen. Ebenso leuchtet ein, dass die D'ESOPO-PAPE-Variante eine Spezialisierung dieses Verfahrens ist mit einer konkreten Angabe der Bearbeitungsreihenfolge. Wir wollen nun noch die Korrektheit der YEN-Variante vorführen.

(5.23) Satz. *Die YEN-Variante des MOORE-BELLMAN-Verfahrens arbeitet korrekt, falls D keinen negativen gerichteten Kreis enthält.* △

Beweis. Wir geben dem Vektor DIST eine Interpretation, aus der die Korrektheit einfach folgt. Wir haben in (5.22) angenommen, dass $s = 1$ gilt und die Knoten mit

$1, 2, \ldots, n$ bezeichnet sind. Wir nennen einen Bogen (u, v) einen *Aufwärtsbogen*, falls $u < v$ gilt, andernfalls heißt (u, v) *Abwärtsbogen*. Wir sprechen von einem *Richtungswechsel*, wenn in einem (s, v)-Weg ein Abwärtsbogen auf einen Aufwärtsbogen folgt oder umgekehrt. Da $s = 1$, ist der erste Bogen immer ein Aufwärtsbogen, also ist der erste Richtungswechsel immer aufwärts nach abwärts. Um einfacher argumentieren zu können, bezeichnen wir mit $\text{DIST}(v, m)$ den Inhalt des Vektors $\text{DIST}(v)$ nach Beendigung der m-ten Iteration der äußeren Schleife.

Wir behaupten nun:

$$\text{DIST}(v, m) = min\{c(W) \mid W \text{ ist ein gerichteter } (1, v)\text{-Weg mit höchstens } m \text{ Richtungswechseln}\},\ 0 \leq m \leq n - 2.$$

Da ein $(1, v)$-Weg höchstens $n - 1$ Bögen und somit höchstens $n - 2$ Richtungswechsel besitzt, folgt der Satz aus dem Beweis unserer Behauptung.

Wir beweisen unsere Behauptung durch Induktion über m. Für $m = 0$ ist der Durchlauf der Schritte 3 und 4 nichts anderes als Algorithmus (angewendet auf $s = 1$ und den azyklischen Digraphen der Aufwärtsbögen, der keinen gerichteten und somit auch keinen gerichteten negativen Kreis enthält), dessen Korrektheit wir in Satz (5.21) bewiesen haben. $\text{DIST}(v, 0)$ enthält somit die Länge des kürzesten $(1, v)$-Weges ohne Richtungswechsel, die Behauptung für $m = 0$ ist also richtig.

Nehmen wir nun an, dass unsere Behauptung für $m \geq 0$ richtig ist und dass wir Schleife 2 zum $(m+1)$-sten Male durchlaufen haben. Wir müssen zwei Fälle unterscheiden: $m+1$ gerade oder ungerade. Wir führen den Fall $m + 1$ ungerade vor, der andere Fall folgt analog. Die Menge der $(1, v)$-Wege mit höchstens $m + 1$ Richtungswechseln besteht aus folgenden Wegen:

(a) $(1, v)$-Wege mit höchstens m Richtungswechseln,

(b) $(1, v)$-Wege mit genau $m + 1$ Richtungswechseln.

Die Minimallänge der Wege in (a) kennen wir nach Induktionsvoraussetzung bereits, sie ist $\text{DIST}(v, m)$.

Wir haben angenommen, dass $s = 1$ gilt, also ist der erste Bogen eines jeden Weges ein Aufwärtsbogen. Für einen $(1, v)$-Weg mit $m + 1$ Richtungswechseln und $m + 1$ ungerade ist daher der letzte Bogen ein Abwärtsbogen.

Zur Bestimmung des Minimums in (b) führen wir eine weitere Induktion über $u = n, n-1, \ldots, v+1$ durch. Da jeder Weg, der in n endet mit einem Aufwärtsbogen aufhört, gibt es keinen $(1, n)$-Weg mit genau $m + 1$ Richtungswechseln, also gilt $\text{DIST}(n, m) = \text{DIST}(n, m + 1)$.

Nehmen wir nun an, dass wir wissen, dass $\text{DIST}(w, m + 1)$ für $n \geq w \geq u > v + 1$ die Länge eines kürzesten $(1, w)$-Weges mit höchstens $m + 1$ Richtungswechseln ist. Zur Bestimmung der Länge eines kürzesten $(1, u - 1)$-Weges mit höchstens $m + 1$ Richtungswechseln müssen wir die Länge eines kürzesten $(1, u - 1)$-Weges mit höchsten m Richtungswechseln (diese ist in $\text{DIST}(u - 1, m)$ gespeichert) vergleichen mit der Länge eines kürzesten $(1, u - 1)$-Weges mit genau $m + 1$ Richtungswechseln.

Sei nun P ein kürzester $(1, u-1)$-Weg mit genau $m+1$ Richtungswechseln. Sei r der Knoten auf P, bei dem der letzte Richtungswechsel erfolgt. Da der letzte Bogen auf P, weil $m+1$ ungerade ist, ein Abwärtsbogen ist, gilt $u \le r \le n$. Der Weg P_r auf P von 1 bis r ist ein gerichteter Weg mit m Richtungswechseln. Also gilt nach Induktionsvoraussetzung $c(P_r) \ge \text{DIST}(r, m)$. Für alle Knoten s, die auf P zwischen r und $u-1$ liegen (also $u-1 < s < r$), ist der Weg P_s auf P von 1 bis s ein gerichteter $(1, s)$-Weg mit genau $m+1$ Richtungswechseln. Somit ist nach Induktionsvoraussetzung $c(P_s) \ge \text{DIST}(s, m+1)$. Ist t der vorletzte Knoten auf P, also $(t, u-1) \in P$, so ist $c(P) = c(P_t) + c(t, u-1) \ge \text{DIST}(t, m+1) + c(t, u-1)$. Der letzte Wert geht in die Minimumsbildung in Schritt 6 ein. Also wird in Schritt 6 der kürzeste aller $(1, u-1)$-Wege mit höchstens $m+1$ Richtungswechseln berechnet. □

Wir haben festgestellt, dass die beiden Varianten des MOORE-BELLMAN-Verfahrens korrekt arbeiten, wenn der gegebene Digraph keine negativen Kreise enthält, aber haben bisher verschwiegen, wie man das effektiv entdeckt. Wie man das bei der D'ESOPO-PAPE-Variante auf einfache Weise machen kann — ohne andere Algorithmen einzuschalten — ist mir nicht bekannt. Bei der YEN-Variante gibt es eine simple Modifikation, die das Gewünschte leistet.

(5.24) Bemerkung. Nehmen wir an, dass jeder Knoten des Digraphen D von $s = 1$ auf einem gerichteten Weg erreicht werden kann. D enthält einen negativen Kreis genau dann, wenn bei einer zusätzlichen Ausführung der Schleife 2 der YEN-Variante (also für $m = n-1$) der Wert $\text{DIST}(v)$ für mindestens einen Knoten $v \in V$ geändert wird. △

Der Beweis dieser Bemerkung sei dem Leser überlassen. Auf das Thema „negative Kreise“ werden wir später noch einmal zurückkommen.

Die YEN-Variante des MOORE-BELLMAN-Algorithmus hat, da drei Schleifen über maximal n Indizes ineinander geschaltet sind, eine Laufzeit von $O(n^3)$. Für die D'ESOPO-PAPE-Variante gibt es (konstruierte) Beispiele mit exponentieller Laufzeit. Dennoch hat sie sich in der Praxis als sehr schnell erwiesen und ist fast immer der YEN-Variante überlegen. Sind alle Gewichte positiv und sollen kürzeste (s, v)-Wege für alle $v \in V$ bestimmt werden, so ist die DIJKSTRA-Methode für Digraphen mit vielen Bögen (d. h. $O(n^2)$ Bögen) die bessere Methode; bei Digraphen mit wenigen Bögen haben extensive Testläufe gezeigt, dass die D'ESOPO-PAPE-Variante in der Praxis günstigere Laufzeiten erbringt.

5.3.3 Kürzeste Wege zwischen allen Knotenpaaren

Natürlich kann man kürzeste Wege zwischen je zwei Knotenpaaren eines Digraphen D dadurch bestimmen, dass man das DIJKSTRA- oder das MOORE-BELLMAN-Verfahren n-mal anwendet, d. h. jeder Knoten wird einmal als Startknoten gewählt. Bei Benutzung der DIJKSTRA-Methode (nicht-negative Gewichte vorausgesetzt) hätte dieses Verfahren eine Laufzeit von $O(n^3)$. Falls negative Gewichte vorkommen, müsste die YEN-Variante verwendet werden, was zu einer Laufzeit von $O(n^4)$ führt. Es gibt jedoch einen extrem einfachen $O(n^3)$-Algorithmus, der das Gleiche leistet. Dieses Verfahren geht auf Floyd (1962) zurück.

(5.25) FLOYD-Algorithmus.

Eingabe: Digraph $D = (V, A)$, $V = \{1, \dots, n\}$ mit Gewichten $c(a)$ (können auch negativ sein), für alle $a \in A$.

Ausgabe: Eine (n, n)-Matrix $W = (w_{ij})$, so dass für $i \neq j$ w_{ij} die Länge des kürzesten (i, j)-Weges und w_{ii} die Länge eines kürzesten gerichteten Kreises, der i enthält, ist (eine Matrix mit diesen Eigenschaften nennt man *Kürzeste-Weglängen-Matrix*) und eine (n, n)-Matrix $P = (p_{ij})$, so dass p_{ij} der vorletzte Knoten eines kürzesten (i, j)-Weges (bzw. (i, i)-Kreises) ist.

1. DO $i = 1$ TO n:
   DO $j = 1$ TO n:

$$w_{ij} := \begin{cases} c((i,j)) & \text{falls } (i,j) \in A \\ +\infty & \text{andernfalls} \end{cases}$$

$$p_{ij} := \begin{cases} i & \text{falls } (i,j) \in A \\ 0 & \text{andernfalls (bedeutet, zur Zeit kein Weg bekannt)} \end{cases}$$

   END
   END.

2. DO $l = 1$ TO n:
   DO $i = 1$ TO n:
   DO $j = 1$ TO n:
   Falls $w_{ij} > w_{il} + w_{lj}$,
   setze $w_{ij} := w_{il} + w_{lj}$ und $p_{ij} := p_{lj}$.
   (Falls $i = j$ und $w_{ii} < 0$, kann abgebrochen werden.)
   END
   END
   END.

3. Gib W und P aus. △

Für zwei Knoten i, j kann der in P gespeicherte kürzeste (i, j)-Weg wie folgt bestimmt werden. Setze $k := 1$ und $v_k := p_{ij}$.

Ist $v_k = i$ dann STOP, andernfalls setze $v_{k+1} := p_{iv_k}, k := k + 1$ und wiederhole, d. h. wir iterieren so lange bis ein Knoten, sagen wir v_s, der Knoten i ist, dann ist

$$(i = v_s, v_{s-1}, v_{s-2}, \dots, v_1, j)$$

ein kürzester (i, j)-Weg. Überzeugen Sie sich, dass dies stimmt!

(5.26) Satz. *Sei $D = (V, A)$ ein Digraph mit beliebigen Bogengewichten $c(a)$ für alle $a \in A$. Sei W die (n, n)-Matrix, die vom FLOYD-Algorithmus produziert wird, dann gilt:*

(a) Der FLOYD-Algorithmus liefert genau dann eine Kürzeste-Weglängen-Matrix W, wenn D keinen negativen gerichteten Kreis enthält.

(b) D enthält genau dann einen negativen gerichteten Kreis, wenn ein Hauptdiagonalelement von W negativ ist. △

Beweis. Zur Notationsvereinfachung bezeichnen wir die Anfangsmatrix W aus Schritt 1. mit W^0, die Matrix W nach Beendigung des l-ten Durchlaufs der äußeren Schleife von Schritt 2. mit W^l. Durch Induktion über $l = 0, 1, \ldots, n$ zeigen wir, dass W^l genau dann die Matrix der kürzesten Längen von (i, j)-Wegen (bzw. (i, i)-Kreisen) ist, bei denen die Knoten $1, \ldots, l$ als innere Knoten auftreten können, wenn D keinen negativen Kreis in der Knotenmenge $1, \ldots, l$ besitzt. Ist letzteres der Fall, so gilt $w^l_{ii} < 0$ für ein $i \in \{1, \ldots, l\}$.

Für $l = 0$ ist die Behauptung offenbar richtig. Angenommen, sie ist für $l \geq 0$ richtig, und wir haben die äußere Schleife von Schritt 2. zum $(l + 1)$-sten Male durchlaufen. Bei diesem Durchlauf haben wir folgenden Schritt ausgeführt.

$$\text{Falls } w^l_{ij} > w^l_{i,l+1} + w^l_{l+1,j}, \text{ dann setze } w^{l+1}_{ij} := w^l_{i,l+1} + w^l_{l+1,j},$$

d. h. wir haben die (nach Induktionsvoraussetzung) kürzeste Länge eines (i, j)-Weges über die Knoten $1, \ldots, l$ verglichen mit der Summe der kürzesten Längen eines $(i, l+1)$-Weges und eines $(l + 1, j)$-Weges jeweils über die Knoten $1, \ldots, l$. Die letztere Summe repräsentiert also die Länge eines kürzesten (i, j)-Weges über $1, \ldots, l+1$, der den Knoten $l + 1$ enthält. Falls diese Summe kleiner als w^l_{ij} ist, setzen wir $w^{l+1}_{ij} := w^l_{i,l+1} + w^l_{l+1,j}$, andernfalls $w^{l+1}_{ij} = w^l_{ij}$. Daraus folgt die Behauptung, es sei denn, $w^l_{ij} > w^l_{i,l+1} + w^l_{l+1,j}$ und die Verkettung, sagen wir K des $(i, l + 1)$-Weges mit dem $(l + 1, j)$-Weg ist gar kein Weg, d. h. K ist eine gerichtete (i, j)-Kette, die einen Knoten mindestens zweimal enthält. Die Kette K enthält natürlich einen (i, j)-Weg, sagen wir $\overline{K}$, und $\overline{K}$ geht aus K dadurch hervor, dass wir die in K vorkommenden gerichteten Kreise entfernen. Der Knoten $l + 1$ ist nach Konstruktion in einem der Kreise enthalten, also ist $\overline{K}$ ein (i, j)-Weg, der nur Knoten aus $\{1, \ldots, l\}$ enthält, d. h. $w^l_{ij} \leq c(\overline{K})$. Aus $c(K) = w^l_{i,l+1} + w^l_{l+1,j} < w^l_{ij}$ folgt, dass mindestens einer der aus K entfernten gerichteten Kreise eine negative Länge hat. Für jeden Knoten i dieses negativen Kreises muss folglich $w^{l+1}_{ii} < 0$ gelten. Daraus folgt die Behauptung. □

Der FLOYD-Algorithmus liefert also explizit einen Kreis negativer Länge, falls ein solcher existiert.

(5.27) Korollar. *Für einen Digraphen D mit Bogengewichten, der keine negativen gerichteten Kreise enthält, kann ein kürzester gerichteter Kreis in $O(n^3)$ Schritten bestimmt werden.* △

Beweis. Wir führen den FLOYD-Algorithmus aus. Nach Beendigung des Verfahrens ist in w_{ii}, $i = 1, \ldots, n$ die Länge eines kürzesten gerichteten Kreises, der den Knoten i enthält, verzeichnet. Wir wählen einen Wert w_{ii}, der so klein wie möglich ist, und rekonstruieren aus der Matrix P, wie oben angegeben, den gerichteten Kreis, der i enthält. Dieser ist ein kürzester gerichteter Kreis in D. Diese "Nachbearbeitung" erfordert lediglich $O(n)$ Operationen, also ist die worst-case-Komplexität des FLOYD-Algorithmus auch die Laufzeitschranke für das Gesamtverfahren. □

Wendet man Algorithmus (5.25) auf Entfernungstabellen in Straßenatlanten an, so wird man feststellen, dass es häufig Städte i, j, k gibt mit $c_{ij} + c_{jk} < c_{ik}$. Die Entfernungen genügen also nicht der Dreiecksungleichung. Warum ist das so?

5.3.4 Min-Max-Sätze und weitere Bemerkungen

Es folgen in einem kurzen Überblick ein paar Zusatzbemerkungen zum Problemkreis „Kürzeste Wege“.

Zwei Min-Max-Sätze In der Optimierungstheorie sind sogenannte Dualitäts- oder Min-Max-Sätze von besonderer Bedeutung. Diese Sätze sind von folgendem Typ: Man hat eine Menge P und eine Zielfunktion c, die jedem Element x von P einen Wert $c(x)$ zuordnet. Gesucht wird

$$\min\{c(x) \mid x \in P\}.$$

Dann gelingt es manchmal auf natürliche Weise und unter gewissen technischen Voraussetzungen eine Menge D und eine Zielfunktion b zu finden, die jedem $y \in D$ einen Wert $b(y)$ zuweist, mit der Eigenschaft

$$\min\{c(x) \mid x \in P\} = \max\{b(y) \mid y \in D\}.$$

Wie wir später sehen werden, ist die Existenz eines Satzes dieser Art häufig ein Indikator dafür, dass das Minimierungs- und das Maximierungsproblem „gut“ gelöst werden können. Für das Kürzeste-Wege-Problem gibt es verschiedene Min-Max-Sätze. Wir geben zwei Beispiele an und erinnern daran, dass ein (s,t)-Schnitt in einem Digraphen $D = (V, A)$ eine Bogenmenge der Form $\delta^+(W) = \{(i, j) \in A \,|\, i \in W, j \in (V \setminus W)\}$ ist mit der Eigenschaft $s \in W$, $t \in W \setminus V$.

(5.28) Satz. *Sei $D = (V, A)$ ein Digraph, und seien $s, t \in V$, $s \neq t$. Dann ist die minimale Länge (= Anzahl der Bögen) eines (s,t)-Weges gleich der maximalen Anzahl bogendisjunkter (s,t)-Schnitte.* △

Beweis. Jeder (s,t)-Weg enthält aus jedem (s,t)-Schnitt mindestens einen Bogen. Gibt es also d bogendisjunkte (s,t)-Schnitte, so hat jeder (s,t)-Weg mindestens die Länge d. Daher ist das Minimum (d. h. die kürzeste Länge eines (s,t)-Weges) mindestens so groß wie das Maximum (gebildet über die Anzahl bogendisjunkter (s,t)-Schnitte).

Sei nun d die Länge eines kürzesten Weges, und sei $V_i, i = 1, ..., d$, die Menge der Knoten $v \in V$, die von s aus auf einem Weg der Länge kleiner als i erreicht werden können. Dann sind die Schnitte $\delta^+(V_i)$ genau d bogendisjunkte (s,t)-Schnitte. □

Eine Verallgemeinerung dieses Sachverhaltes auf gewichtete Digraphen ist das folgende Resultat.

(5.29) Satz. *Seien $D = (V, A)$ ein Digraph, $s, t \in V$, $s \neq t$, und $c(a) \in \mathbb{Z}_+$ für alle $a \in A$. Dann ist die kürzeste Länge eines (s,t)-Weges gleich der maximalen Anzahl d von (nicht notwendig verschiedenen) (s,t)-Schnitten $C_1, ..., C_d$, so dass jeder Bogen $a \in A$ in höchstens $c(a)$ Schnitten C_i liegt.* △

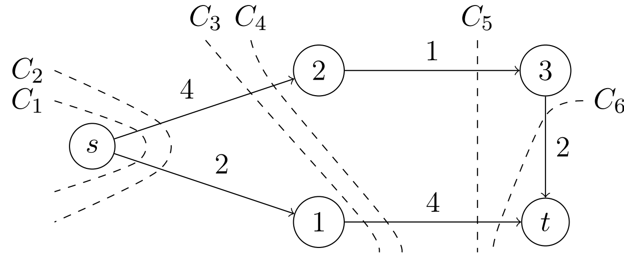


Abbildung 5.4: Beispiel für Satz (5.29).

Beweis. Sei P ein (s,t)-Weg und seien $C_1, ..., C_d$ (s,t)-Schnitte wie im Satz gefordert, dann gilt

$$c(P) = \sum_{a \in P} c(a) \geq \sum_{a \in P} |\{i : a \in C_i\}| = \sum_{i=1}^{d} |C_i \cap P| \geq \sum_{i=1}^{d} 1 = d$$

Also ist das Minimum nicht kleiner als das Maximum.

Wählen wir die (s,t)-Schnitte $C_i := \delta^+(V_i)$, mit $V_i := \{v \in V \mid v$ kann von s aus auf einem gerichteten Weg P mit $c(P) \leq i-1$ erreicht werden$\}$, $i = 1, ..., d$, dann sehen wir, dass Gleichheit gilt. □

Ein ganz einfaches Beispiel soll zur Veranschaulichung von Satz (5.29) dienen. Wir betrachten Abbildung 5.4. Der kürzeste Weg von s nach t hat offensichtlich Länge 6. Konstruieren wir die Knotenmengen V_i und die realisierenden (s,t)-Schnitte $\delta^+(V_i)$, so ergeben sich folgende Knotenmengen: $V_1 = V_2 = \{s\}$, $V_3 = V_4 = \{s, 1\}$, $V_5 = \{s, 1, 2\}$, $V_6 = \{s, 1, 2, 3\}$. Die zugehörigen Schnitte $C_i := \delta^+(V_i)$ sind in Abbildung 5.4 als gestrichelte Linien angedeutet.

Kommen wir zur Einführung von min-max-Sätzen zurück. In abstrakter kombinatorischer Notation kann man Satz (5.29) wie folgt notieren. Mit $\mathcal{P}(D,c)$ bezeichnen wir die Menge aller (s,t)-Wege P in D und $c(P) := \sum_{a \in P} c(a)$ sei die Länge eines Weges $P \in \mathcal{P}(D,c)$. Mit $\mathcal{C}$ bezeichnen wir die Menge aller endlichen Folgen $(C_1, C_2, \ldots, C_k)$, $k \geq 1$, von (s,t)-Schnitten, d. h. $C_i = \delta^+(W_i)$ mit $s \in W_i$, $t \notin W_i$, wobei C_i und C_j, $i \neq j$, nicht voneinander verschieden sein müssen. Sei

$$\mathcal{D}(D,c) := \{(C_1, C_2, \ldots, C_k) \in \mathcal{C} \mid |\{i \in \{1, \ldots, k\} \colon a \in C_i\}| \leq c(a) \ \forall a \in A\}$$

und der Wert $b((C_1, C_2, \ldots, C_k))$ einer Folge aus $\mathcal{D}(D,c)$ sei die Anzahl der nicht notwendig verschiedenen Elemente der Folge, also $b((C_1, C_2, \ldots, C_k)) = k$. Dann gilt:

$$\min\{c(P) \mid P \in \mathcal{P}(D,c)\} = \max\{b((C_1, C_2, \ldots, C_k)) \mid (C_1, C_2, \ldots, C_k) \in \mathcal{D}(D,c)\}.$$

Kürzeste Wege in ungerichteten Graphen Transformieren wir einen ungerichteten Graphen G in einen gerichteten Graphen D, indem wir jeder Kante ij die beiden Bögen

(i, j) und (j, i) mit demGewicht von ij zuordnen, so können wir natürlich durch Anwendung unserer Verfahren auf D auch kürzeste Wege bzw. Kreise in G bestimmen. Man beachte jedoch, dass ein negatives Kantengewicht $c(ij)$ in G automatisch zu einem negativen gerichteten Kreis $(i, j)(j, i)$ in D führt. Mit unseren Methoden können wir also nur kürzeste Wege und Kreise in Graphen mit nichtnegativen Kantengewichten bestimmen.

Es sei an dieser Stelle jedoch darauf hingewiesen, dass auch in Graphen mit negativen Kantengewichten kürzeste Wege und Kreise bestimmt werden können, falls kein Kreis negativ ist. Dies geschieht mit Methoden der Matching-Theorie, auf die wir hier aus Zeitgründen nicht eingehen können.

Laufzeiten Genaue Laufzeitanalysen von verschiedenen Varianten der hier vorgestellten Algorithmen zur Berechnung von kürzesten Wegen findet man z. B. in Ahuja et al. (1993) auf den Seiten 154–157 ebenso, wie einen kurzen geschichtlichen Überblick.

Umfangreiche historische Bemerkungen zur Theorie und Algorithmik von kürzesten Wegen bietet das Buch von Schrijver (2003). In den Abschnitten 7.5 und 8.6 sind z. B. Tabellen zu finden, die die Verbesserungen der Worst-Case-Laufzeiten von Kürzeste-Wege-Algorithmen dokumentieren.

Ein Algorithmus zur Bestimmung kürzester Wege muss jeden Bogen des gegebenen Digraphen $D = (V, A)$ mindestens einmal „anfassen“. Eine untere Schranke für die Laufzeit eines jeden Algorithmus dieser Art ist somit $O(m)$, $m = |A|$. Thorup (1997) hat gezeigt, dass man diese Laufzeit für ungerichtete Graphen mit nichtnegativen Kantengewichten tatsächlich erreichen kann. Er benutzt dazu sogenannte “Atomic Heaps”, deren Verwendung $n = |V| \geq 2^{12^{20}}$ voraussetzt. Das bedeutet, dass diese Methode zwar theoretisch “gut”, aber für die Praxis ungeeignet ist. (Thorup diskutiert in seinem Aufsatz auch implementierbare Varianten, allerdings haben diese eine schlechtere Laufzeit, z.B. $O(\log C_{max} + m + n \log\log\log n)$, wobei C_{max} das größte Kantengewicht bezeichnet.)

Bei Routenplanern, wie sie z.B. im Internet oder in den Bordcomputern von Autos angeboten werden, treten Digraphen mit mehreren Millionen Knoten auf (die in der Regel nur einen kleinen Grad haben). Die Anbieter solcher Programme haben für derartige Probleme, bei denen ja der Grundgraph, der auf der CD gespeichert ist, fest bleibt, Spezialverfahren entwickelt (z.B. durch intensives Preprocessing und die Abspeicherung wichtiger kürzester Verbindungen), die Kürzeste-Wege-Probleme dieser Art sehr schnell lösen. Um (selbst gesetzte) Zeitschranken für den Nutzer einzuhalten, benutzen diese Algorithmen z.T. Heuristiken, und derartige Algorithmen liefern nicht notwendig immer einen beweisbaren kürzesten Weg. Einen Überblick über diesen Aspekt findet man in Goldberg (2007).

Fast alle Navigationssysteme bieten mehrere Optimierungsmöglichkeiten zur Bestimmung eines „besten“ Weges an. Man kann z. B. den schnellsten oder den kürzesten Weg bestimmen lassen. Manche Navigationssysteme offerieren eine „Kombinationsoptimierung“, man kann dann etwa eingeben, dass Schnelligkeit mit 70% und Streckenkürze mit 30% berücksichtigt werden. Dies ist eine spezielle Version der Mehrzieloptimierung. Die zwei Zielfunktionen werden hierbei mit Parametern multipliziert und dann aufaddiert, so dass nur eine einzige Zielfunktion entsteht. Das nennt man *Skalierung*. Man könnte auch an-

ders vorgehen: z. B. könnte man nach der schnellsten Route suchen, die eine vorgegebene km-Zahl nicht überschreitet. Der Grund dafür, dass Navigationssysteme solche Optionen nicht anbieten, liegt darin, dass Kürzeste-Wege-Probleme mit Nebenbedingungen in der Regel $\mathcal{NP}$-schwer sind. Das ist z. B. so bei „schnellster Weg mit km-Beschränkung" oder „kürzester Weg mit Zeitbeschränkung".

Noch besser für den Autofahrer wäre die Angabe der *Pareto-Menge*. Im Falle der beiden Zielfunktionen „kürzester Weg" und „schnellster Weg" müsste das Navigationssystem alle Wege angeben, die „nicht dominiert" sind. Solche Wege haben die Eigenschaft, dass beim Versuch, die Weglänge kürzer zu machen, die Fahrzeit erhöht wird oder umgekehrt. Das hierbei unüberwindliche Problem ist, dass die Kardinalität der Pareto-Menge exponentiell in der Kodierungslänge der Daten wachsen kann. Die Navigationssysteme würden „unendlich lange" rechnen und der Autofahrer in der Informationsflut ertrinken. Aus diesem theoretischen Grund wird nur mit einer Skalierung gearbeitet, die durch den Nutzer (falls er das will) vorgegeben wird.

Man könnte glauben, dass Fahrzeugnavigation das größte Anwendungsfeld von Methoden zur Berechnung kürzester Wege sei, aber der Umfang der Verwendung dieser Methoden ist im Internet noch viel größer. Das derzeit am häufigsten verwendete Routing-Protokoll ist das „Open Shortest Path First-Protokoll" (kurz: OSPF). Bei Verwendung dieses Protokolls wird für jedes Datenpaket ein kürzester Weg (u. U. mehrfach) bestimmt, und wenn man allein die Anzahl der E-Mails abschätzt, die weltweit täglich versandt werden, so sieht man sehr schnell, wie häufig die Bestimmung kürzester Wege zum Einsatz kommt. Ich kann hier das OSPF-Protokoll nicht im Detail erklären und verweise dazu auf Internetquellen, z. B. Wikipedia.

Die am Ende von Abschnitt 5.2 genannten Webseiten bieten auch verschiedene Algorithmen zur Berechnung kürzester Wege an.

5.4 LP/IP-Aspekte von Bäumen und Wegen

Wir haben nun gelernt, wie man minimale Bäume, maximale Wälder und kürzeste Wege bestimmen kann. Dies ist mit theoretisch und praktisch hocheffizienten Algorithmen möglich. Häufig ist das Auffinden von Bäumen und Wegen jedoch nur ein Teilproblem eines komplizierten mathematischen Modells, welches mit Methoden der gemischt-ganzzahligen Optimierung behandelt werden muss. Und ebenso häufig treten Zusatzanforderungen an die Bäume und Wege auf, die aus diesen einfachen Problemen $\mathcal{NP}$-schwere Aufgaben machen. In solchen Fällen (und nicht nur dabei) ist es erforderlich, Bäume und Wege-Probleme als ganzzahlige (oder wenn möglich lineare) Programme zu formulieren. Wir deuten nachfolgend an, wie das gemacht werden kann.

Wir beginnen mit Wäldern in einem gegebenen Graphen $G = (V, E)$ mit Kantengewichten c_e, $e \in E$. Wir wollen ein lineares Programm aufstellen, dessen ganzzahlige Lösungen den Wäldern in G entsprechen. Dazu müssen wir lernen, wie man eine Kantenmenge $F \subseteq E$ als Vektor darstellt. Wir betrachten hierzu den Vektorraum $\mathbb{R}^E$. Man kann diesen als die Menge der Abbildungen von E in $\mathbb{R}$ auffassen. Die (für uns) angenehmste Sichtweise ist die folgende: Ein Vektor $x \in \mathbb{R}^E$ hat für jedes Element $e \in E$

eine Komponente x_e. Wir schreiben den Vektor dann wie folgt: $x = (x_e)_{e \in E}$. Für eine beliebige Teilmenge $F \subseteq E$ definieren wir ihren *Inzidenzvektor* $\chi^F = (\chi_e^F)_{e \in E}$ dadurch, dass wir setzen: $\chi_e^F = 1$, falls $e \in F$, und $\chi_e^F = 0$, falls $e \notin F$. Definieren wir: $\mathfrak{W} := \{F \subseteq E \mid (V, F) \text{ ist ein Wald in } G\}$, so können wir die Aufgabe, einen gewichtsmaximalen Wald in $\mathfrak{W}$ zu finden, nun auffassen als die Aufgabe, in der Menge aller Inzidenzvektoren

$$\mathfrak{F} := \{\chi^F \in \mathbb{R}^E \mid F \in \mathfrak{W}\}$$

von Wäldern einen Vektor mit höchstem Gewicht zu bestimmen. Die Kantengewichte c_e können wir analog zu einem Vektor $c = (c_e)_{e \in E} \in \mathbb{R}^E$ zusammenfassen, und damit haben wir die Aufgabe, einen gewichtsmaximalen Wald zu finden, als Optimierungsproblem mit linearer Zielfunktion

$$\max \Big\{ c^T x = \sum_{e \in E} c_e x_e \;\Big|\; x \in \mathfrak{F} \Big\}$$

über einer Menge von 0/1-Vektoren formuliert.

Um diese Aufgabe als lineares ganzzahliges Programm auffassen zu können, müssen wir nun noch lineare Ungleichungen finden, so dass die ganzzahligen Lösungen des Ungleichungssystems genau die Vektoren in $\mathfrak{F}$ sind. Wir beginnen mit einer Trivialüberlegung. Die Komponenten der Vektoren in $\mathfrak{F}$ nehmen nur die Werte 0 oder 1 an. Damit erfüllt jedes Element von $\mathfrak{F}$ die Ungleichungen

$$0 \le x_e \le 1 \quad \forall e \in E.$$

Nun gilt es, ein System von Ungleichungen zu „erraten“, das von allen Elementen aus $\mathfrak{F}$ erfüllt wird, und das die Eigenschaft hat, dass jeder 0/1-Vektor, der nicht in $\mathfrak{F}$ ist, mindestens eine der Ungleichungen des Systems verletzt. Die Idee dazu liefert die Definition. Ein Wald (V, F), $F \subseteq E$ ist dadurch gekennzeichnet, dass (V, F) keinen Kreis enthält. Fordern wir also, dass für jeden Kreis $C \subseteq E$ die Ungleichung

$$x(C) := \sum_{e \in C} x_e \le |C| - 1$$

(genannt *Kreisungleichung*) erfüllt sein muss, so erfüllt nach Definition jeder Inzidenzvektor eines Waldes jede dieser Ungleichungen. Ist χ^H der Inzidenzvektor einer Kantenmenge $H \subseteq E$, die einen Kreis C enthält, so gilt:

$$\sum_{e \in C} \chi_e^H = |C|$$

und damit verletzt χ^H die zu C gehörige Kreisungleichung. Wir fassen zusammen: Das System von Ungleichungen

$$\begin{aligned} 0 \le x_e &\le 1 && \forall e \in E, \\ x(C) &\le |C| - 1 && \forall C \subseteq E,\ C \text{ Kreis in } G \end{aligned}$$

ist ein System von Ungleichungen, dessen 0/1-Lösungen genau der Menge der Inzidenzvektoren der Wälder in G entspricht. Daraus folgt, dass

$$\begin{aligned} \max \quad & c^T x \\ & 0 \leq x_e \leq 1 && \forall e \in E, \\ & x(C) \leq |C| - 1 && \forall C \subseteq E,\ C \text{ Kreis in } G, \\ & x_e \in \mathbb{Z} && \forall e \in E \end{aligned} \tag{5.30}$$

ein ganzzahliges lineares Programm ist, dessen ganzzahlige Lösungen die Inzidenzvektoren von Wäldern sind.

Es wird sich zeigen (und an dieser Stelle beweisen wir das noch nicht), dass das obige ganzzahlige Programm keine „gute Formulierung" des Wald-Problems ist. Es gibt (mindestens) ein besseres Modell. Man kann beweisen, dass das folgende lineare Programm

$$\begin{aligned} \max \quad & c^T x \\ & 0 \leq x_e \leq 1 && \forall e \in E \\ & x(E(W)) \leq |W| - 1 && \forall W \subseteq V, (W, E(W)) \text{ ist 2-fach zusammenhängend} \end{aligned} \tag{5.31}$$

die Eigenschaft hat, dass die „Ecklösungen" dieses LPs genau die Inzidenzvektoren der Wälder in G sind. Man kann also das Wald-Problem durch ein lineares Programm lösen (ohne Ganzzahligkeitsforderung). Man zahlt allerdings einen hohen Preis dafür. Die Anzahl der Nebenbedingungen ist exponentiell in $|V|$. Man kann sie z. B. für einen vollständigen Graphen mit 100.000 Knoten überhaupt nicht aufschreiben (es gibt 2^{100000} Ungleichungen). Dennoch kann man derartige LPs effizient lösen, wie wir später sehen werden.

Wir wissen nun, wie man Wälder mit linearer Optimierung behandelt. Kann man auf ähnliche Weise auch minimale aufspannende Bäume bestimmen? Überlegen Sie sich das bitte!

Und nun zu (s,t)-Wegen in einem Digraphen $D = (V, A)$ mit $s, t \in V, s \neq t$. Wir gehen hier analog vor. Jedem Bogen $(u, v) \in A$ ordnen wir eine Variable x_{uv} zu. Diese Variablen fassen wir zu einem Vektor $x = (x_{uv})_{(u,v) \in A} \in \mathbb{R}^A$ zusammen. Für jeden (s,t)-Weg $P \subseteq A$ definieren wir den zugehörigen Inzidenzvektor $\chi^P = (\chi^P_{uv})_{(u,v) \in A} \in \mathbb{R}^A$ durch $\chi^P_{uv} = 1$, falls $(u, v) \in P$, und $\chi^P_{uv} = 0$, falls $(u, v) \notin P$. Kann man nun „kanonische" Gleichungen und Ungleichungen finden, so dass das Kürzeste-Wege-Problem als ganzzahliges Programm oder gar als lineares Programm ohne Ganzzahligkeitsbedingung gelöst werden kann? Es folgen ein paar einfache Überlegungen hierzu.

Da alle Inzidenzvektoren 0/1-Vektoren sind, müssen die Ungleichungen

$$0 \leq x_{uv} \leq 1 \quad \forall (u, v) \in A \tag{5.32}$$

erfüllt sein. Aus dem Anfangsknoten $s \in V$ führt ein Bogen hinaus, in den Endknoten $t \in V$ des (s,t)-Weges führt ein Bogen hinein. Daraus folgt, dass die beiden folgenden

Gleichungen erfüllt sein müssen:

$$\begin{aligned} x(\delta^+(s)) &= \sum_{(s,v)\in A} x_{sv} = 1 \\ x(\delta^-(t)) &= \sum_{(u,t)\in A} x_{ut} = 1. \end{aligned} \tag{5.33}$$

Jeder andere Knoten $v \in V \setminus \{s,t\}$ ist nur „Durchgangsknoten“, d. h. wenn ein Bogen des (s,t)-Weges in v hineinführt, dann geht auch ein Bogen dieses Weges hinaus. Geht kein Bogen in v hinein, dann geht auch keiner hinaus. Also müssen die folgenden Ungleichungen erfüllt sein:

$$x(\delta^-(v)) - x(\delta^+(v)) = \sum_{(u,v)\in A} x_{uv} - \sum_{(v,w)\in A} x_{vw} = 0 \quad \forall v \in V \setminus \{s,t\}. \tag{5.34}$$

Reicht dieses System von Gleichungen und Ungleichungen (5.32), (5.33), (5.34) zur Lösung des Kürzeste-Wege-Problems aus? Benötigt man zusätzlich Ganzzahligkeitsbedingungen? Braucht man noch mehr Ungleichungen oder Gleichungen? Denken Sie darüber nach!

5.5 Exkurs: Greedy-Heuristiken für das Rucksackproblem und deren Analyse

Betrachtet man das Kürzeste-Wege-Problem und die Berechnung minimaler aufspannender Bäume, dann könnte man meinen, dass der Greedy-Algorithmus immer eine gute Methode zur Lösung kombinatorischer Optimierungsprobleme ist. Das ist natürlich nicht so. Wir wollen deshalb Varianten dieses Algorithmen-Typs für das *Rucksackproblem* (engl.: *knapsack problem*) untersuchen. Das Rucksackproblem ist in einem gewissen Sinne eines der einfachsten (aber dennoch $\mathcal{NP}$-schweren) ganzzahligen Optimierungsprobleme.

Das Rucksackproblem kann folgendermaßen beschrieben werden. Gegeben seien n verschiedene Arten von Gegenständen, die in einen Rucksack gefüllt werden sollen. Der Rucksack hat ein beschränktes Fassungsvermögen b. Jeder Gegenstand einer Art j hat ein „Gewicht“ a_j und einen „Wert“ c_j. Der Rucksack soll so gefüllt werden, dass seine Kapazität nicht überschritten wird und der Gesamtinhalt so wertvoll wie möglich ist. Von diesem Problem gibt es viele Varianten. Hier betrachten wir zwei von diesen.

(5.35) Definition. *Seien a_j, $c_j \in \mathbb{Z}_+$, $j = 1, 2, \ldots, n$ und $b \in \mathbb{N}$.*

(a) Das Problem

$$\begin{aligned} \max \sum_{j=1}^{n} c_j x_j & \\ \sum_{j=1}^{n} a_j x_j \le b & \\ x_j \in \mathbb{Z}_+ & \end{aligned} \tag{KP}$$

heißt (allgemeines) Rucksack-Problem *oder* Rucksackproblem.

(b) *Fordern wir zusätzlich, dass* $x_j \in \{0,1\}$, $j = 1, \ldots, n$ *gilt, so heißt* (KP) binäres Rucksack-Problem *oder* 0/1-Rucksack-Problem. △

Zu dem (so simpel erscheinenden) Rucksackproblem gibt es eine außerordentlich umfangreiche Literatur. Wir erwähnen hier nur zwei Bücher. Der „Klassiker" zum Thema ist Martello and Toth (1990), seine „Fortsetzung" Kellerer et al. (2004). Dieses Buch mit 546 Seiten enthält so gut wie alles, was man über das Rucksack-Problem wissen will.

Wir untersuchen zwei Versionen des Greedy-Algorithmus für das Rucksack-Problem.

(5.36) Zwei Greedy-Algorithmen für das allgemeine bzw. binäre Rucksack-Problem.

Eingabe: $c_j, a_j \in \mathbb{N}$, $j = 1, \ldots, n$ und $b \in \mathbb{N}$.

Ausgabe: Eine zulässige (approximative) Lösung für (KP) bzw. das binäre Rucksackproblem.

1. *Zielfunktionsgreedy*: Ordne die Indizes, so dass $c_1 \geq c_2 \geq \ldots \geq c_n$ gilt.

1'. *Gewichtsdichtengreedy*: Berechne die *Gewichtsdichten* $\rho_j := c_j/a_j$, $j = 1, 2, \ldots, n$, und ordne die Indizes so dass $\rho_1 \geq \rho_2 \geq \ldots \geq \rho_n$ gilt.

2. DO $j = 1$ TO n:

   allgemeiner Rucksack:

   Setze $x_j := \lfloor b/a_j \rfloor$ und $b := b - \lfloor b/a_j \rfloor$.

   0/1-*Rucksack*:

   Falls $a_j > b$, setze $x_j := 0$.

   Falls $a_j \leq b$, setze $x_j := 1$ und $b := b - a_j$.

   END. △

Die Laufzeit der in (5.36) beschriebenen Algorithmen ist offensichtlich $O(n \log(n))$, wobei das Sortieren die Hauptarbeit erfordert.

Wir nennen einen Algorithmus A *ϵ-approximativ*, $0 < \epsilon \leq 1$, falls er für *jedes* Problembeispiel $\mathcal{I}$ eines Maximierungsproblems einen Wert $c_A(\mathcal{I})$ liefert, der mindestens einen ϵ-Anteil des Optimalwerts $c_{\text{opt}}(\mathcal{I})$ von $\mathcal{I}$ beträgt, in Formeln, falls

$$c_A(\mathcal{I}) \geq \epsilon c_{\text{opt}}(\mathcal{I}). \tag{5.37}$$

(5.38) Bemerkung. Der Zielfunktionsgreedy kann für das allgemeine und auch für das binäre Rucksack-Problem beliebig schlechte Lösungen liefern, d. h. es gibt kein ϵ, $0 < \epsilon \leq 1$, sodass dieser Greedy-Algorithmus ϵ-approximativ ist. △

Beweis. (a) Allgemeines Rucksack-Problem. Wir betrachten das folgende Beispiel mit $\alpha \in \mathbb{Z}$, $\alpha \geq 2$:

$$\begin{aligned} \max\ & \alpha x_1 + (\alpha - 1)x_2 \\ & \alpha x_1 + \quad x_2 \leq \alpha \\ & x_1, \quad x_2 \in \mathbb{Z}_+ \end{aligned}$$

Offenbar gilt $c_{\text{greedy}} = \alpha$ und der Wert der Optimallösung ist $c_{\text{opt}} = \alpha(\alpha - 1)$. Aus der Forderung $c_{\text{greedy}} \geq \epsilon c_{\text{opt}}$ folgt dann $\epsilon \leq 1/(\alpha - 1)$; da wir α beliebig wählen können, kann es kein festes $\epsilon > 0$ mit $c_{\text{greedy}} \geq \epsilon c_{\text{opt}}$ geben.

(b) 0/1-Rucksack-Problem. Wir betrachten für $n \in \mathbb{Z}$, $n \geq 2$:

$$\begin{aligned} \max\ & nx_1 + (n-1)x_2 + \cdots + (n-1)x_n \\ & nx_1 + \quad x_2 + \cdots + \quad x_n \leq n \\ & x_j \in \{0, 1\}. \end{aligned}$$

Trivialerweise gilt $c_{\text{greedy}} = n$ und $c_{\text{opt}} = n(n - 1)$, und wir können mit demselben Argument wie oben sehen, dass der Zielfunktionsgreedy nicht ϵ-approximativ ist. □

(5.39) Satz. *Der Gewichtsdichten-Greedyalgorithmus ist für das allgemeine Rucksackproblem ein 1/2-approximativer Algorithmus.* △

Beweis. O. B. d. A. können wir annehmen, dass $\rho_1 \geq \rho_2 \geq \ldots \geq \rho_n$ und $a_1 \leq b$ gilt. Es gilt offensichtlich für den Zielfunktionswert c_{Ggreedy} des Gewichtsdichten-Greedyalgorithmus

$$c_{\text{Ggreedy}} \geq c_1 x_1 = c_1 \lfloor b/a_1 \rfloor,$$

und ebenso

$$c_{\text{opt}} \leq c_1 b/a_1 \leq c_1(\lfloor b/a_1 \rfloor + 1) \leq 2c_1 \lfloor b/a_1 \rfloor \leq 2c_{\text{Ggreedy}}$$

und somit

$$c_{\text{Ggreedy}} \geq \tfrac{1}{2} c_{\text{opt}}.$$

Wir zeigen nun, dass diese Schranke tatsächlich asymptotisch angenommen wird. Dazu betrachten wir das folgende Rucksackproblem:

$$\begin{aligned} \max\ & 2\alpha x_1 + 2(\alpha - 1)x_2 \\ & \alpha x_1 + (\alpha - 1)x_2 \leq 2(\alpha - 1) \\ & x_1, \quad x_2 \in \mathbb{Z}_+ \end{aligned}$$

Offensichtlich gilt $\rho_1 \geq \rho_2$, $c_{\text{Ggreedy}} = 2\alpha$, $c_{\text{opt}} = 4(\alpha - 1)$ und somit

$$\frac{c_{\text{Ggreedy}}}{c_{\text{opt}}} = \frac{2\alpha}{4(\alpha - 1)} \to \frac{1}{2}.$$

□

Um lästige Trivialbemerkungen zu vermeiden, nehmen wir im Weiteren an, dass die Indizes so geordnet sind, dass für die Gewichtsdichten $\rho_1 \geq \rho_2 \geq \ldots \geq \rho_n$ und dass $a_j \leq b$ gilt für $j = 1, \ldots, n$. Leider gilt die schöne Schranke aus (5.39) nicht für das 0/1-Rucksack-Problem.

(5.40) Bemerkung. Gegeben sei ein 0/1-Rucksack-Problem. Dann gilt:

(a) Für $k = 0, 1, \ldots, n-1$ gilt

$$c_{\text{opt}} \leq \sum_{j=1}^{k} c_j + \frac{c_{k+1}}{a_{k+1}} \cdot \Big(b - \sum_{j=1}^{k} a_j\Big).$$

(b) $c_{\text{Ggreedy}} > c_{\text{opt}} - \max\{c_j \mid j = 1, \ldots, n\}$. △

Beweis. (a) Sei $x_1^*, \ldots, x_n^*$ eine optimale Lösung des 0/1-Rucksack-Problems und $k \in \{0, 1, \ldots, n-1\}$. Dann gilt:

$$\begin{aligned} c_{\text{opt}} = \sum_{j=1}^{n} c_j x_j^* &\leq \sum_{j=1}^{k} c_j x_j^* + \sum_{j=k+1}^{n} \frac{a_j c_{k+1}}{a_{k+1}} x_j^* \\ &= \frac{c_{k+1}}{a_{k+1}} \sum_{j=1}^{n} a_j x_j^* + \sum_{j=1}^{k} \Big(c_j - \frac{c_{k+1}}{a_{k+1}} a_j\Big) x_j^* \\ &\leq \frac{c_{k+1}}{a_{k+1}} b + \sum_{j=1}^{k} \Big(c_j - \frac{c_{k+1}}{a_{k+1}} a_j\Big) \\ &= \sum_{j=1}^{k} c_j + \frac{c_{k+1}}{a_{k+1}} \Big(b - \sum_{j=1}^{k} a_j\Big). \end{aligned}$$

(b) Ist $\sum_{j=1}^{n} a_j \leq b$, so liefert der Greedy-Algorithmus offenbar die optimale Lösung und die Behauptung ist korrekt. Sei also $k \leq n$ der größte Index, so dass $\sum_{j=1}^{k} a_j \leq b$. Dann gilt

$$0 \leq b - \sum_{j=1}^{k} a_j < a_{k+1},$$

und aus (a) folgt

$$c_{\text{opt}} \leq \sum_{j=1}^{k} c_j + \frac{c_{k+1}}{a_{k+1}} a_{k+1} \leq c_{\text{Ggreedy}} + c_{k+1},$$

woraus (b) folgt. □

(5.41) Bemerkung.

(a) Der Gewichtsdichten-Greedyalgorithmus kann im Falle des 0/1-Rucksack-Problems beliebig schlechte Lösungen liefern.

(b) Führen wir sowohl den Gewichtsdichten- als auch den Zielfunktions-Greedyalgorithmus für ein 0/1-Rucksack-Problem aus, so ist dieses kombinierte Verfahren ein $\frac{1}{2}$-approximativer Algorithmus. △

Beweis. (a) Wir betrachten das folgende Beispiel mit $\rho_1 \geq \rho_2$:

$$\begin{aligned} \max\ & x_1 + \alpha x_2 \\ & x_1 + \alpha x_2 \leq \alpha \\ & x_1, \quad x_2 \in \{0,1\}. \end{aligned}$$

Es gilt $c_{\text{opt}} = \alpha$ und $c_{\text{Ggreedy}} = 1$, also ist der Algorithmus für kein ϵ, $0 < \epsilon \leq 1$, ϵ-approximativ.

(b) Gilt nach Ausführung des Gewichtsdichten-Greedyalgorithmus $c_{\text{Ggreedy}} \geq \frac{1}{2} c_{\text{opt}}$, so sind wir fertig. Andernfalls sei

$$c_k = \max\{c_j \mid j = 1, \ldots, n\}$$

und mit (5.40)(b) gilt dann $\frac{1}{2} c_{\text{opt}} > c_{\text{Ggreedy}} > c_{\text{opt}} - c_k$, woraus $c_k > \frac{1}{2} c_{\text{opt}}$ folgt. Für den Wert c_{Ggreedy} des Zielfunktions-Greedyalgorithmus gilt trivialerweise $c_{\text{Ggreedy}} \geq c_k$. Daraus ergibt sich die Behauptung. □

(5.41)(b) ist ein bemerkenswertes Ergebnis: Die Kombination zweier „schlechter“ Heuristiken ergibt eine „gute“ Heuristik (wenn man die theoretische Approximationsgüte als Qualitätsmaßstab wählt). Dies ist ein durchaus seltenes Ergebnis.

Literaturverzeichnis

R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows, Theory, Algorithms and Applications.* Paramount Publishing International, Prentice Hall, New York, 1993.

R. E. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1: 269–271, 1959.

W. Domschke. *Kürzeste Wege in Graphen.* Verlag A. Hain, Meisenheim am Glan, 1972.

R. W. Floyd. Algorithm 97, shortest path. *Communications of the ACM*, 5(6):345, 1962.

F. Glover, D. D. Klingman, and N. V. Phillips. A new polynomially bounded shortest path algorithm. *Operations Research*, 33(1):65–73, 1985.

A. Goldberg. Point-to-point shortest path algorithms with preprocessing. In *SOFSE 2007: Theory and Practice of Computer Science*, pages 88–102, 2007.

R. L. Graham and P. Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7:43–57, 1982.

H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems.* Springer, 2004.

S. O. Krumke and H. Noltemeier. *Graphentheoretische Konzepte und Algorithmen.* Teubner, Wiesbaden, 2005. ISBN 3-519-00526-3.

E. L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart & Winston, New York, 1976.

S. Martello and P. Toth. *Knapsack problems: Algorithms and computer implementations.* J. Wiley & Sons, 1990.

K. Mehlhorn. *Data Structures and Algorithms*, volume 1–3. Springer-Verlag, EATCS Monographie edition, 1984. (dreibändige Monographie, Band I liegt auch auf deutsch im Teubner-Verlag (1986) vor).

E. F. Moore. The shortest path through a maze. In *Proc. Int. Symp. on Theory of Switching Part II*, pages 285–292. Harvard University Press, 1959.

J. Nešetřil and H. Nešetřilová. The origins of minimal spanning tree algorithms – Borůvka and Jarník. *Documenta Mathematica*, pages 127–141, 2012.

A. Schrijver. *Combinatorial Optimization – Polyhedra and Efficiency.* Springer-Verlag, Berlin, 2003.

M. M. Syslo, N. Deo, and J. S. Kowalik. *Discrete optimization algorithms.* Prentice Hall, Englewood Cliffs, N.J., 1983.

M. Thorup. Undirected single shortest paths in linear time. In *Proceedings of the 38th IEE Symposium on Foundations of Comp. Sci. (FOCS)*, pages 12–21, 1997.

6 Maximale Flüsse in Netzwerken

In diesem Kapitel behandeln wir ein in sowohl theoretischer als auch praktischer Hinsicht auß erordentlich interessantes Gebiet: Flüsse in Netzwerken. Es war früher üblich und wird aus Traditionsgründen häufig weiter so gehandhabt, einen Digraphen mit Bogengewichten bzw. -kapazitäten ein *Netzwerk* zu nennen. Sind zusätzlich zwei Knoten s und t ausgezeichnet, so spricht man von einem (s, t)-*Netzwerk*. Wir wollen diese Bezeichnung hier nur gelegentlich übernehmen, mit dem Kapiteltitel aber den historischen Bezug herstellen. Es wird sich (später) zeigen, dass die Netzwerkflusstheorie als ein Bindeglied zwischen der linearen und der ganzzahligen Optimierung aufgefasst werden kann. Netzwerkflussprobleme sind (ganzzahlige) lineare Programme, für die sehr schnelle kombinatorische Lösungsmethoden existieren. Der Dualitätssatz der linearen Programmierung hat hier eine besonders schöne Form.

Netzwerkflüsse haben folgenden Anwendungshintergrund. Wir betrachten ein Rohrleitungssystem (Abwasserkanäle, Frischwasserversorgung), bei dem die Rohre gewisse Kapazitäten (z. B. maximale Durchflussmenge pro Minute) haben. Einige typische Fragen lauten: Was ist die maximale Durchflussmenge durch das Netzwerk? Welche Wassermenge kann man maximal pro Minute vom Speicher zu einem bestimmten Abnehmer pumpen? Wieviel Regenwasser kann das System maximal aufnehmen? Ähnliches gilt für Telefonnetzwerke. Hier sind die „Rohre“ die Verbindungen zwischen zwei Knotenpunkten, die Kapazitäten die maximalen Anzahlen von Gesprächen bzw. die maximalen Datenmengen pro Zeiteinheit, die über eine Verbindung geführt werden können. Man interessiert sich z. B. für die maximale Zahl von Gesprächen, die parallel zwischen zwei Orten (Ländern) geführt werden können, oder den größtmöglichen Datentransfer pro Zeiteinheit zwischen zwei Teilnehmern.

Darüber hinaus treten Netzwerkflussprobleme in vielfältiger Weise als Unter- oder Hilfsprobleme bei der Lösung komplizierter Anwendungsprobleme auf, z. B. bei vielen Routenplanungs- und verschiedenen Logistikproblemen. Insbesondere werden Netzwerkflussalgorithmen sehr häufig als Separierungsroutinen bei Schnittebenenverfahren eingesetzt, ein Thema von ADM II und ADM III.

Das klassische Werk der Netzwerkflusstheorie ist das Buch L. R. Ford and Fulkerson (1962). Es ist auch heute noch lesenswert. Es gibt unzählige Veröffentlichungen zur Theorie, Algorithmik und den Anwendungen der Netzwerkflusstheorie. Durch neue algorithmische Ansätze (Präfluss-Techniken, Skalierungsmethoden) und effiziente Datenstrukturen sind Ende der 80er und zu Beginn der 90er Jahre sehr viele Artikel zu diesem Thema erschienen. Gute Darstellungen hierzu sind in den umfangreichen und sehr informativen Übersichtsartikeln Ahuja et al. (1989), V. Goldberg and Tarjan (1990) und Frank (1995) zu finden. Ein sehr empfehlenswertes Buch ist Ahuja et al. (1993). Die beiden Handbücher Ball et al. (1995a), Ball et al. (1995b) enthalten umfassende Informationen zur

Theorie, Algorithmik und zu den Anwendungen von Netzwerken. Und natürlich gibt es eine hoch kondensierte Zusammenfassung der Netzwerkflusstheorie und -algorithmik in Schrijver (2003).

6.1 Das Max-Flow-Min-Cut-Theorem

Im Folgenden sei $D = (V, A)$ ein Digraph mit Bogenkapazitäten $c(a) \in \mathbb{R}, c(a) \geq 0$ für alle $a \in A$. Ferner seien s und t zwei voneinander verschiedene Knoten aus V. Der Knoten s heißt *Quelle* (englisch: source), und t heißt *Senke* (englisch: sink). Eine Funktion $x\colon A \to \mathbb{R}$ (bzw. ein Vektor $x \in \mathbb{R}^A$) heißt (s,t)*-Fluss*, wenn die folgenden *Flusserhaltungsbedingungen* erfüllt sind:

$$\sum_{a \in \delta^-(v)} x_a = \sum_{a \in \delta^+(v)} x_a \qquad \forall v \in V \setminus \{s,t\}. \tag{6.1}$$

Erfüllt x zusätzlich die *Kapazitätsbedingungen*

$$0 \leq x_a \leq c_a \qquad \forall a \in A, \tag{6.2}$$

so ist x ein *zulässiger* (s,t)*-Fluss*. Für einen (s,t)-Fluss $x \in \mathbb{R}^A$ heißt

$$\operatorname{val}(x) := \sum_{a \in \delta^+(s)} x_a - \sum_{a \in \delta^-(s)} x_a \tag{6.3}$$

der *Wert* des (s,t)-Flusses x. Wenn nicht extra darauf hingewiesen wird, ist mit „Fluss" stets ein zulässiger Fluss gemeint.

Wir werden uns in diesem Abschnitt damit beschäftigen, eine Charakterisierung des maximalen Wertes eines (s,t)-Flusses zu finden. In den nächsten beiden Abschnitten werden wir Algorithmen zur Bestimmung eines maximalen zulässigen Flusses angeben.

Wir erinnern daran, dass ein (s,t)*-Schnitt* in D eine Bogenmenge der Form $\delta^+(W) = \delta^-(V \setminus W) = \{(i,j) \in A \mid i \in W, j \in V \setminus W\}$ mit $s \in W \subseteq V$ und $t \in V \setminus W$ ist. Die *Kapazität eines Schnittes* $\delta^+(W)$ ist wie üblich mit $c(\delta^+(W)) = \sum_{a \in \delta^+(W)} c_a$ definiert. Aus der „Rohrleitungsanwendung" wird der Name „Schnitt" klar. Durchschneiden wir alle Rohre irgendeines Schnittes, d. h. entfernen wir alle Bögen eines (s,t)-Schnittes aus dem Digraphen, dann kann kein Fluss mehr von s nach t „fließen". Diese triviale Beobachtung liefert:

(6.4) Lemma.

(a) *Seien $s \in W$, $t \notin W$, dann gilt für jeden zulässigen (s,t)-Fluss x:*

$$\operatorname{val}(x) = \sum_{a \in \delta^+(W)} x_a - \sum_{a \in \delta^-(W)} x_a.$$

(b) *Der maximale Wert eines zulässigen (s,t)-Flusses ist höchstens so groß wie die minimale Kapazität eines (s,t)-Schnittes.* △

Beweis. (a) Aus der Flusserhaltungsbedingung (6.1) folgt

$$\begin{aligned}\operatorname{val}(x) &= \sum_{a\in\delta^+(s)} x_a - \sum_{a\in\delta^-(s)} x_a = \sum_{v\in W}\Big(\sum_{a\in\delta^+(v)} x_a - \sum_{a\in\delta^-(v)} x_a\Big)\\ &= \sum_{a\in\delta^+(W)} x_a - \sum_{a\in\delta^-(W)} x_a.\end{aligned}$$

(b) Seien $\delta^+(W)$ ein beliebiger (s,t)-Schnitt und x ein zulässiger (s,t)-Fluss, dann gilt wegen (a) und (6.2):

$$\operatorname{val}(x) = \sum_{a\in\delta^+(W)} x_a - \sum_{a\in\delta^-(W)} x_a \le \sum_{a\in\delta^+(W)} c_a = c(\delta^+(W)).$$

□

Wir werden später einen kombinatorischen Beweis dafür angeben, dass der maximale Wert eines (s,t)-Flusses gleich der minimalen Kapazität eines (s,t)-Schnittes ist. Hier wollen wir jedoch bereits eine Vorschau auf die lineare Programmierung machen, die das Max-Flow-Min-Cut-Theorem in einen allgemeineren Kontext einbettet. Wir präsentieren dieses Resultat daher als Anwendung von Resultaten, die erst später in der Vorlesung im Kapitel über lineare Optimierung behandelt werden.

Wir schreiben zunächst die Aufgabe, einen maximalen (s,t)-Flusswert in D zu finden, als lineares Programm. Dieses lautet wie folgt:

$$\begin{array}{llll} \max \displaystyle\sum_{a\in\delta^+(s)} x_a - \sum_{a\in\delta^-(s)} x_a & \big[\, = x(\delta^+(s)) - x(\delta^-(s))\,\big] & & \\ \displaystyle\sum_{a\in\delta^-(v)} x_a - \sum_{a\in\delta^+(v)} x_a & = x(\delta^-(v)) - x(\delta^+(v)) & = 0 & \forall v \in V\setminus\{s,t\}, \\ \qquad\qquad x_a & & \le c_a & \forall a\in A, \\ \qquad\qquad x_a & & \ge 0 & \forall a \in A. \end{array} \tag{6.5}$$

Jede zulässige Lösung von (6.5) ist also ein zulässiger (s,t)-Fluss, und jede optimale Lösung ein maximaler (s,t)-Fluss. Um das zu (6.5) duale lineare Programm aufschreiben zu können, führen wir für jeden Knoten $v\in V\setminus\{s,t\}$ eine Dualvariable z_v und für jeden Bogen $a\in A$ eine Dualvariable y_a ein. Das folgende lineare Programm ist dann (im Sinne

der Theorie der linearen Optimierung) zu (6.5) dual.

$$
\begin{aligned}
\min \sum_{a \in A} c_a y_a & \\
y_a + z_v - z_u &\geq 0 && \text{falls } a = (u,v) \in A(V \setminus \{s,t\}),\\
y_a \qquad - z_u &\geq -1 && \text{falls } a = (u,s), u \neq t,\\
y_a + z_v \qquad &\geq 1 && \text{falls } a = (s,v), v \neq t,\\
y_a \qquad - z_u &\geq 0 && \text{falls } a = (u,t), u \neq s,\\
y_a + z_v \qquad &\geq 0 && \text{falls } a = (t,v), v \neq s,\\
y_a &\geq 1 && \text{falls } a = (s,t),\\
y_a &\geq -1 && \text{falls } a = (t,s),\\
y_a &\geq 0 && \text{für alle } a \in A.
\end{aligned}
$$

Führen wir zusätzlich (zur notationstechnischen Vereinfachung) die Variablen z_s und z_t ein und setzen sie mit 1 bzw. 0 fest, so kann man dieses LP äquivalent, aber etwas kompakter wie folgt schreiben:

$$
\begin{aligned}
\min \sum_{a \in A} c_a y_a & \\
y_a + z_v - z_u &\geq 0 && \text{für alle } a = (u,v) \in A,\\
z_s &= 1 \\
z_t &= 0 \\
y_a &\geq 0 && \text{für alle } a \in A.
\end{aligned}
\tag{6.6}
$$

Wir benutzen nun (6.5) und (6.6), um folgenden berühmten Satz, der auf Ford, Jr. and Fulkerson (1956) und P. Elias and Shannon (1956) zurückgeht, zu beweisen.

(6.7) Satz (Max-Flow-Min-Cut-Theorem). *Gegeben seien ein Digraph $D = (V, A)$ mit Bogenkapazitäten $c_a \in \mathbb{R}$, $c_a \geq 0$ für alle $a \in A$, und zwei verschiedene Knoten $s, t \in V$. Dann ist der maximale Wert eines zulässigen (s,t)-Flusses gleich der minimalen Kapazität eines (s,t)-Schnittes.* △

Beweis. Aufgrund von Lemma (6.4) genügt es zu zeigen, dass es einen (s,t)-Schnitt gibt, dessen Kapazität gleich dem maximalen Flusswert ist. Da alle Variablen beschränkt sind und der Nullfluss zulässig ist, hat (6.5) eine optimale Lösung. Sei also x^* ein optimaler zulässiger (s,t)-Fluss mit Wert $\text{val}(x^*)$. Aufgrund des Dualitätssatzes der linearen Programmierung gibt es eine Lösung, sagen wir y^*_a, $a \in A$ und z^*_v, $v \in V$, des zu (6.5) dualen Programms (6.6) mit $\text{val}(x^*) = \sum_{a \in A} c_a y^*_a$. Wir setzen: $W := \{u \in V \mid z^*_u > 0\}$ und zeigen, dass $\delta^+(W)$ ein (s,t)-Schnitt mit $c(\delta^+(W)) = \text{val}(x^*)$ ist.

Offenbar gilt $s \in W$, $t \notin W$, also ist $\delta^+(W)$ ein (s,t)-Schnitt. Ist $a = (u,v) \in \delta^+(W)$, dann gilt $z^*_u > 0$, $z^*_v \leq 0$ und folglich $y^*_a \geq z^*_u - z^*_v > 0$. Aufgrund des Satzes vom schwachen komplementären Schlupf muss dann in der zu y^*_a gehörigen Ungleichung $x_a \leq c_a$ des primalen Programms (6.5) Gleichheit gelten. Also erfüllt der optimale (s,t)-Fluss

x^* die Gleichung $x_a^* = c_a$. Ist $a = (u, v) \in \delta^-(W)$, so gilt $z_v^* > 0$, $z_u^* \leq 0$ und somit (da $y_a^* \geq 0$) $y_a^* - z_u^* + z_v^* \geq z_v^* - z_u^* > 0$. Die Ungleichung ist also „locker“. Der Satz vom komplementären Schlupf impliziert nun, dass die zugehörige Primalvariable $x_a \geq 0$ nicht positiv sein kann. Also gilt $x_a^* = 0$. Daraus folgt

$$c(\delta^+(W)) = x^*(\delta^+(W)) - x^*(\delta^-(W)) = x^*(\delta^+(s)) - x^*(\delta^-(s)) = \text{val}(x^*). \qquad \square$$

Vielen Lesern des Manuskripts mag der obige Beweis noch unverständlich sein. Er wurde jedoch aufgenommen, um hier schon Beispielmaterial für die Theorie der linearen Programmierung vorzubereiten.

Man beachte, dass der obige Beweis des Max-Flow Min-Cut Theorems konstruktiv ist. Aus jeder optimalen Lösung des dualen linearen Programms können wir in polynomialer Zeit einen (s, t)-Schnitt konstruieren, der den gleichen Wert wie die Optimallösung von (6.6) hat und somit ein (s, t)-Schnitt in D minimaler Kapazität ist. Aus jedem (s, t)-Schnitt $\delta^+(W)$ können wir durch

$$\begin{aligned} y_a &:= 1 \quad \text{für alle } a \in \delta^+(W) \\ y_a &:= 0 \quad \text{für alle } a \in A \setminus \delta^+(W) \\ z_v &:= 1 \quad \text{für alle } v \in W \\ z_v &:= 0 \quad \text{für alle } v \in V \setminus W \end{aligned}$$

auch eine Lösung von (6.6) konstruieren, und daraus folgt, dass das lineare Programm (6.6) immer auch ganzzahlige optimale Lösungen hat. Wir können somit also das ganzzahlige Programm bestehend aus (6.6) plus Ganzzahligkeitsbedingungen für y_a und z_v durch das im Beweis von (6.7) angegebene Verfahren lösen. Wir werden im nächsten Abschnitt zeigen, dass auch das LP (6.5) immer ganzzahlige Optimallösungen hat, wenn alle Kapazitäten ganzzahlig sind. Die diesem Beweis unterliegende Konstruktion ist der Startpunkt für effiziente Algorithmen zur Lösung des Maximalflussproblems.

Das Max-Flow-Min-Cut-Theorem hat vielfältige Anwendungen in der Graphentheorie und kombinatorischen Optimierung. Aus Zeitgründen können wir an dieser Stelle nicht darauf eingehen. Wir verweisen u.a. auf Ahuja et al. (1993) und Schrijver (2003).

6.2 Der Ford-Fulkerson-Algorithmus

Wir haben gesehen, dass das Maximalflussproblem und das Minimalschnittproblem lineare Programme sind, folglich können wir sie effizient lösen. Das heißt, in der Praxis dürfte der Simplexalgorithmus für (6.5) und (6.6) in kurzer Zeit gute Lösungen liefern, während theoretisch die Ellipsoidmethode eine Laufzeit garantiert, die polynomial in der Inputlänge $|V| + |A| + \sum_{a \in A} \langle c_a \rangle$ ist. Für diese besonderen linearen Programme gibt es jedoch effiziente kombinatorische Algorithmen und Spezialversionen des Simplexalgorithmus, die außerordentlich schnell arbeiten.

Wir werden eines dieser Verfahren vorstellen, dass auf Ford und Fulkerson zurückgeht. Die Idee hinter dieser Methode kann wie folgt erläutert werden. Man starte mit dem zulässigen (s, t)-Fluss, z. B. mit $x_a = 0$ für alle $a \in A$. Hat man einen zulässigen (s, t)-Fluss,

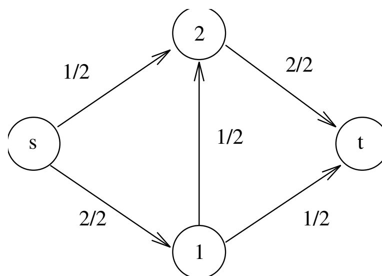


Abbildung 6.1: Digraph ohne gerichteten (s,t)-Weg, der eine Flusserhöhung ermöglicht.

dann versuche man im gegebenen Digraphen einen gerichteten Weg von s nach t zu finden, auf dem zusätzlich ein positiver Wert durch das Netzwerk „geschoben" werden kann. Geht dies, so erhöht man den gegenwärtigen Fluss und fährt fort. Die Suche nach einem gerichteten (s,t)-Weg, der die Erhöhung des Flusswertes erlaubt, führt allerdings nicht direkt zum gewünschten Erfolg. Betrachten wir z. B. den Digraphen D in Abbildung 6.1, bei dem die erste Zahl des zu einem Bogen gehörenden Zahlenpaares den gegenwärtigen Flusswert des Bogens anzeigt und die zweite Zahl die Kapazität des Bogen angibt. Der gegenwärtige Fluss hat den Wert 3, und offenbar hat der Maximalfluss den Wert 4. Es gibt im Digraphen von Abbildung 6.1 aber keinen gerichteten (s,t)-Weg auf dem der gegenwärtige Fluss verbessert werden könnte. Auf allen drei gerichteten (s,t)-Wegen ist mindestens ein Bogenfluss an seiner maximalen Kapazität. Eine Möglichkeit, den Fluss entlang eines *ungerichteten* Weges zu erhöhen, haben wir jedoch. Wir betrachten den $[s,t]$-Weg P mit den Bögen $(s,2)$, $(1,2)$, $(1,t)$ und erhöhen den Fluss der Bögen $(s,2)$, $(1,t)$ um jeweils 1, erniedrigen den Fluss durch (1,2) um 1. Dadurch wird weder eine der Kapazitätsbedingungen (6.2) noch eine der Flusserhaltungsbedingungen verletzt, aber der Flusswert um 1 erhöht. Wir treffen daher folgende Definition.

(6.8) Definition. *Sei $D = (V,A)$ ein Digraph mit Bogenkapazitäten c_a für alle $a \in A$, seien $s,t,v \in V$, $s \neq t$, und sei x ein zulässiger (s,t)-Fluss in D. In einem (ungerichteten) $[s,v]$-Weg P nennen wir einen Bogen (i,j), der auf P in Richtung s nach v verläuft,* Vorwärtsbogen*, andernfalls heißt (i,j)* Rückwärtsbogen. *P heißt* augmentierender $[s,v]$-Weg *(bezüglich des (s,t)-Flusses x), falls $x_{ij} < c_{ij}$ für jeden Vorwärtsbogen (i,j) und $x_{ij} > 0$ für jeden Rückwärtsbogen (i,j) gilt. Wenn wir nur* augmentierender Weg *sagen, so meinen wir immer einen augmentierenden $[s,t]$-Weg.* △

Im oben angegebenen Weg P des in Abbildung 6.1 gezeigten Digraphen ist $(1,2)$ ein Rückwärtsbogen, $(s,2)$ und $(1,t)$ sind Vorwärtsbögen. P selbst ist augmentierend bezüglich des gegebenen Flusses. Der folgende Satz liefert ein Optimalitätskriterium.

(6.9) Satz. *Ein zulässiger (s,t)-Fluss x in einem Digraphen D mit Bogenkapazitäten ist genau dann maximal, wenn es in D keinen bezüglich x augmentierenden $[s,t]$-Weg gibt.* △

Beweis. Ist P ein bezüglich x augmentierender $[s,t]$-Weg, dann sei

$$\varepsilon := \min \begin{cases} c_{ij} - x_{ij} & \text{falls } (i,j) \in P \text{ Vorwärtsbogen,} \\ x_{ij} & \text{falls } (i,j) \in P \text{ Rückwärtsbogen.} \end{cases} \tag{6.10}$$

Setzen wir

$$x'_{ij} := \begin{cases} x_{ij} + \varepsilon & \text{falls } (i,j) \in P \text{ Vorwärtsbogen,} \\ x_{ij} - \varepsilon & \text{falls } (i,j) \in P \text{ Rückwärtsbogen,} \\ x_{ij} & \text{falls } (i,j) \in A \setminus P, \end{cases} \tag{6.11}$$

dann ist offenbar x'_{ij} ein zulässiger (s,t)-Fluss mit $\text{val}(x') = \text{val}(x) + \varepsilon$. Also kann x nicht maximal sein.

Angenommen x besitzt keinen augmentierenden Weg. Dann sei W die Knotenmenge, die aus s und denjenigen Knoten $v \in V$ besteht, die von s aus auf einem bezüglich x augmentierenden $[s,v]$-Weg erreicht werden können. Definition (6.8) impliziert $x_a = c_a$ für alle $a \in \delta^+(W)$ und $x_a = 0$ für alle $a \in \delta^-(W)$. Daraus ergibt sich $\text{val}(x) = x(\delta^+(W)) - x(\delta^-(W)) = x(\delta^+(W)) = c(\delta^+(W))$. Aufgrund von Lemma (6.4)(b) ist somit x maximal. □

Der Beweis von Satz (6.9) liefert einen Schnitt $\delta^+(W)$ mit $\text{val}(x) = c(\delta^+(W))$. Zusammen mit Lemma (6.4)(b) ergibt dies einen kombinatorischen Beweis des Max-Flow-Min-Cut-Theorems. Aus dem Beweis von Satz (6.9) folgt ebenfalls, dass das lineare Programm (6.5) ganzzahlige Optimallösungen hat, falls alle Kapazitäten ganzzahlig sind.

(6.12) Satz. *Sei $D = (V,A)$ ein Digraph mit ganzzahligen Bogenkapazitäten $c_a \geq 0$, und seien $s,t \in V$, $s \neq t$. Dann gibt es einen maximalen zulässigen (s,t)-Fluss, der ganzzahlig ist.* △

Beweis. Wir führen einen Induktionsbeweis über die Anzahl der „Additionen“ augmentierender Wege. Wir starten mit dem Nullfluss. Haben wir einen ganzzahligen Flussvektor und ist dieser nicht maximal, so bestimmen wir den Wert ε durch (6.10). Nach Voraussetzung ist ε ganzzahlig, und folglich ist der neue durch (6.11) festgelegte Flussvektor ebenfalls ganzzahlig. Bei jeder Augmentierung erhöhen wir den Flusswert um mindestens eins. Da der maximale Flusswert endlich ist, folgt die Behauptung aus Satz (6.9). □

Wir können nun den Ford-Fulkerson-Algorithmus angeben:

(6.13) Ford-Fulkerson-Algorithmus.

Eingabe: Digraph $D = (V,A)$ mit Bogenkapazitäten $c_a \in \mathbb{R}$, $c_a \geq 0$ für alle Bögen $a \in A$ und zwei Knoten $s,t \in V$, $s \neq t$.

Ausgabe: Ein zulässiger (s,t)-Fluss x mit maximalem Wert $\text{val}(x)$ und ein kapazitätsminimaler (s,t)-Schnitt $\delta^+(W)$.

1. (Initialisierung) Sei $x = (x_{ij}) \in \mathbb{R}^A$ ein zulässiger (s,t)-Fluss. Hier verwendet man am besten eine schnelle Heuristik zur Auffindung eines „guten“ Flusses. Wenn einem nichts einfällt, setzt man z. B. $x_{ij} = 0$ für alle $(i,j) \in A$. Lege folgende Datenstrukturen an:

W	Menge der markierten Knoten
U	Menge der markierten, aber noch nicht überprüften Knoten
VOR	$(n-1)$-Vektor, in dem der Vorgänger eines Knoten v auf einem augmentierenden $[s,v]$-Weg gespeichert wird
EPS	$(n-1)$-Vektor, zur sukzessiven Berechnung von (6.10)

Markieren und Überprüfen

2. Setze $W := \{s\}$, $U := \{s\}, \text{EPS}(s) := +\infty$.

3. Ist $U = \emptyset$, dann gehe zu 9.

4. Wähle einen Knoten $i \in U$ aus und setze $U := U \setminus \{i\}$.

5. Führe für alle Bögen $(i,j) \in A$ mit $j \notin W$ Folgendes aus:
   Ist $x_{ij} < c_{ij}$, dann setze

$$\text{EPS}(j) := \min\{c_{ij} - x_{ij}, \text{EPS}(i)\},$$
$$\text{VOR}(j) := +i,\ W := W \cup \{j\},\ U := U \cup \{j\}.$$

6. Führe für alle Bögen $(j,i) \in A$ mit $j \notin W$ Folgendes aus:
   Ist $x_{ji} > 0$, dann setze

$$\text{EPS}(j) := \min\{x_{ji}, \text{EPS}(i)\},$$
$$\text{VOR}(j) := -i,\ W := W \cup \{j\},\ U := U \cup \{j\}.$$

7. Gilt $t \in W$, gehe zu 8, andernfalls zu 3.

Augmentierung

8. Konstruiere einen augmentierenden Weg und erhöhe den gegenwärtigen Fluss um $\text{EPS}(t)$, d. h. bestimme $j_1 = |\text{VOR}(t)|$, falls $\text{VOR}(t) > 0$, setze $x_{j_1 t} := x_{j_1 t} + \text{EPS}(t)$, andernfalls setze $x_{tj_1} := x_{tj_1} - \text{EPS}(t)$. Dann bestimme $j_2 := |\text{VOR}(j_1)|$, falls $\text{VOR}(j_1) > 0$, setze $x_{j_2 j_1} := x_{j_2 j_1} + \text{EPS}(t)$, andernfalls $x_{j_1 j_2} := x_{j_1 j_2} - \text{EPS}(t)$ usw. bis der Knoten s erreicht ist. Gehe zu 2.

Bestimmung eines minimalen Schnittes

9. Der gegenwärtige (s,t)-Fluss x ist maximal und $\delta^+(W)$ ist ein (s,t)-Schnitt minimaler Kapazität. STOP. △

Aus den Sätzen (6.9) und (6.12) folgt, dass Algorithmus (6.13) für ganzzahlige Kapazitäten korrekt arbeitet und nach endlicher Zeit abbricht. Sind die Daten rational, so kann man (wie üblich) alle Kapazitäten mit dem kleinsten gemeinsamen Vielfachen ihrer Nenner multiplizieren. Man erhält so ein äquivalentes ganzzahliges Maximalflussproblem.

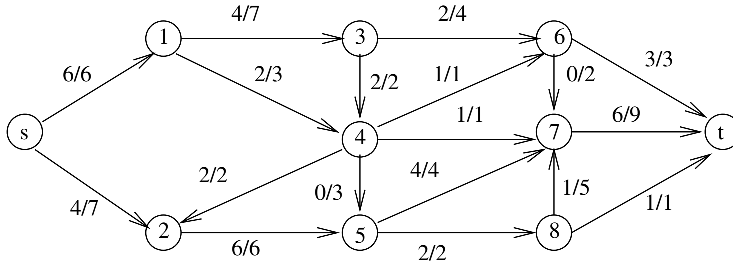


Abbildung 6.2: Beispiel-Digraph für den Ford-Fulkerson-Algorithmus.

Also funktioniert (6.13) auch bei rationalen Daten. Lässt man (zumindest theoretisch) auch irrationale Kapazitäten zu, so kann man Beispiele konstruieren, bei denen Algorithmus (6.13) nicht nach endlicher Zeit abbricht. Aber auch bei ganzzahligen Daten gibt es Probleme. Ein Durchlauf der Markierungs- und Überprüfungsphase und der Augmentierungsphase kann offenbar in $O(m)$, $m = |A|$, Schritten durchgeführt werden. Nach jedem Durchlauf wird der Flusswert um mindestens 1 erhöht. Ist also v der Wert des maximalen (s,t)-Flusses, so ist die Laufzeit von (6.13) $O(m \cdot v)$. Diese Laufzeit ist nicht polynomial in $n + m + \sum_{a \in A} \langle c_a \rangle$, und wenn man die im Verfahren (6.13) noch nicht exakt spezifizierten Schritte ungeschickt ausführt, kann man tatsächlich zu exorbitanten Laufzeiten kommen. Allerdings haben Edmonds and Karp (1972) gezeigt:

(6.14) Satz. *Falls in Algorithmus* (6.13) *jeder Augmentierungsschritt entlang eines augmentierenden* $[s,t]$-*Weges mit minimaler Bogenzahl durchgeführt wird, dann erhält man einen Maximalfluss nach höchstens* $\frac{mn}{2}$ *Augmentierungen. Also ist die Gesamtlaufzeit dieser Version des Verfahrens* (6.13) $O(m^2 n)$. △

Satz (6.14) gilt für beliebige (auch irrationale) Bogenkapazitäten. Es ist in diesem Zusammenhang interessant zu bemerken, dass praktisch jeder, der Verfahren (6.13) implementiert, die Edmonds-Karp-Vorschrift einhält. Üblicherweise arbeitet man die Knoten in Breadth-First-Reihenfolge ab. Dies führt zu augmentierenden Wegen minimaler Bogenzahl. Das heißt, man implementiert die Menge U der markierten und noch nicht abgearbeiteten Knoten als Schlange. Wird ein Knoten in Schritt 5 oder 6 zu U hinzugefügt, so kommt er an das Ende der Schlange. In Schritt 4 wird immer der Knoten $i \in U$ gewählt, der am Anfang der Schlange steht.

(6.15) Beispiel. Wir betrachten den in Abbildung 6.2 dargestellten Digraphen. Die erste Zahl des Zahlenpaares bei einem Bogen gibt den gegenwärtigen Fluss durch den Bogen an, die zweite die Kapazität des Bogens. In Abbildung 6.2 starten wir also mit einem Fluss des Wertes 10.

Wir führen einen Durchlauf der Markierungs- und Überprüfungsphase vor. Im weiteren sei

$$\begin{aligned} \text{VOR} &= (\text{VOR}(1), \text{VOR}(2), \ldots, \text{VOR}(8), \text{VOR}(t)) \\ \text{EPS} &= (\text{EPS}(1), \text{EPS}(2), \ldots, \text{EPS}(8), \text{EPS}(t)). \end{aligned}$$

Das Verfahren beginnt wie folgt:

2. $W := \{s\}$, $U := \{s\}$.
3. –
4. Wir wählen $s \in U$ und setzen $U := \emptyset$.
5. $W := \{s, 2\}$, $U := \{2\}$, $\text{VOR} = (-, +s, -, -, -, -, -, -, -)$,
   $\text{EPS} = (-, 3, -, -, -, -, -, -, -)$.
6. –
7. –
3. –
4. Wir wählen $2 \in U$, $U := \emptyset$.
5. –
6. $W := \{s, 2, 4\}$, $U := \{4\}$, $\text{VOR} = (-, +s, -, -2, -, -, -, -, -)$,
   $\text{EPS} = (-, 3, -, 2, -, -, -, -, -)$.
7. –
3. –
4. Wir wählen $4 \in U$, $U := \emptyset$.
5. $W := \{s, 2, 4, 5\}, U := \{5\}$, $\text{VOR} = (-, +s, -, -2, +4, -, -, -, -)$,
   $\text{EPS} = (-, 3, -, 2, 2, -, -, -, -)$.
6. $W := \{s, 2, 4, 5, 1, 3\}, U := \{5, 1, 3\}$, $\text{VOR} = (-4, +s, -4, -2, +4, -, -, -, -)$,
   $\text{EPS} = (2, 3, 2, 2, 2, -, -, -, -)$.
7. –
3. –
4. Wir wählen $5 \in U$, $U := \{1, 3\}$.
5. –
6. –
7. –
3. –
4. Wir wählen $1 \in U$, $U := \{3\}$.
5. –
6. –
7. –
3. –
4. Wir wählen $3 \in U$, $U := \emptyset$.
5. $W := \{s, 2, 4, 5, 1, 3, 6\}, U := \{6\}$, $\text{VOR} = (-4, +s, -4, -2, +4, +3, -, -, -)$,
   $\text{EPS} = (2, 3, 2, 2, 2, 2, -, -, -)$.
6. –
7. –
3. –
4. Wir wählen $6 \in U$, $U := \emptyset$.
5. $W := \{s, 2, 4, 5, 1, 3, 6, 7\}, U := \{7\}$, $\text{VOR} = (-4, +s, -4, -2, +4, +3, +6, -, -)$,
   $\text{EPS} = (2, 3, 2, 2, 2, 2, 2, -, -)$.
6. –
7. –
3. –
4. Wir wählen $7 \in U$, $U := \emptyset$.
5. $W := \{s, 2, 4, 5, 1, 3, 6, 7, t\}, U := \{t\}$,
   $\text{VOR} = (-4, +s, -4, -2, +4, +3, +6, -, +7)$, $\text{EPS} = (2, 3, 2, 2, 2, 2, 2, -, 2)$.
6. (Hier wird noch 8 markiert, das ist aber irrelevant, da t bereits markiert ist)
7. $t \in W$

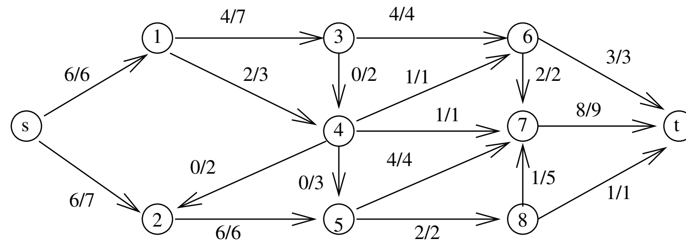


Abbildung 6.3: Maximaler Fluss für den Beispiel-Digraph aus Abbildung 6.2.

8. Es gilt

$$\begin{aligned} \text{VOR}(t) &= +7 \\ \text{VOR}(7) &= +6 \\ \text{VOR}(6) &= +3 \\ \text{VOR}(3) &= -4 \\ \text{VOR}(4) &= -2 \\ \text{VOR}(2) &= +s \end{aligned}$$

also ist der augmentierende Weg mit $\text{EPS}(t) = 2$ der folgende $(s, 2)$, $(4, 2)$, $(3, 4)$, $(3, 6)$, $(6, 7)$, $(7, t)$. Der neue Fluss ist in Abbildung 6.3 dargestellt. Dieser (s, t)-Fluss ist maximal, ein (s, t)-Schnitt minimaler Kapazität ist $\delta^+(\{s, 2\})$, ein anderer $\delta^+(\{s, 1, 2, 3, 4, 5\})$. △

6.3 Einige Anwendungen

In diesem Abschnitt geht es nicht um praktische Anwendungen, sondern um Anwendungen der im Vorhergehenden angegebenen Sätze und Algorithmen zur Lösung anderer mathematischer (Optimierungs-)Probleme.

Matchings maximaler Kardinalität in bipartiten Graphen In (3.9) haben wir das bipartite Matchingproblem kennengelernt. Wir wollen nun zeigen, wie man die Kardinalitätsversion dieses Problems, d. h. alle Kantengewichte sind 1, mit Hilfe eines Maximalflussverfahrens lösen kann.

Ist also $G = (V, E)$ ein bipartiter Graph mit Bipartition V_1, V_2, so definieren wir einen Digraphen $D = (W, A)$ wie folgt. Wir wählen zwei neue Knoten, sagen wir s und t, und setzen $W := V \cup \{s, t\}$. Die Bögen von D seien die folgenden. Ist $e = uv \in E$ eine Kante von G, so geben wir dieser die Richtung von V_1 nach V_2. Ist also $u \in V_1$ und $v \in V_2$, so wird aus $uv \in E$ der Bogen (u, v) andernfalls der Bogen (v, u). Ferner enthält D die Bögen (s, u) für alle $u \in V_1$ und die Bögen (v, t) für alle $v \in V_2$. Alle Bögen von D erhalten die Kapazität 1. Die Konstruktion von D aus G ist in Abbildung 6.4 an einem Beispiel dargestellt.

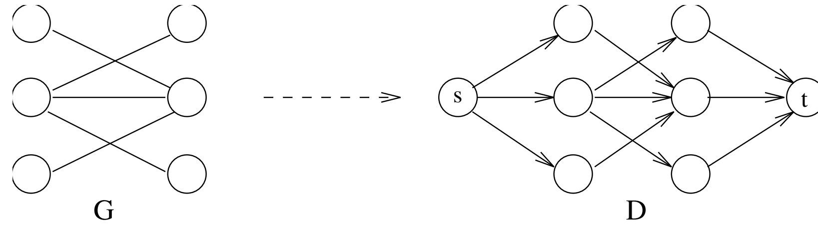


Abbildung 6.4: Transformation eines bipartiten Matchingproblems in ein Maximalflussproblem.

(6.16) Satz. *Ist G ein bipartiter Graph und D der wie oben angegeben aus G konstruierte Digraph, dann ist der Wert eines maximalen zulässigen (s,t)-Flusses x in D gleich dem Wert eines maximalen Matchings in D. Ferner kann ein maximales Matching M direkt aus x konstruiert werden.* △

Beweis. Hausaufgabe. □

Zusammenhangsprobleme in Graphen und Digraphen Mit Hilfe von Maximalflussalgorithmen können ferner für einen Digraphen die starke Zusammenhangszahl und die starke Bogenzusammenhangszahl in polynomialer Zeit bestimmt werden. Analog können in einem ungerichteten Graphen die Zusammenhangszahl und die Kantenzusammenhangszahl in polynomialer Zeit ermittelt werden.

Mehrere Quellen und Senken Die Festlegung, dass wir in einem Digraphen einen Fluss von nur einer Quelle zu nur einer Senke schicken wollen, scheint auf den ersten Blick eine starke Einschränkung zu sein. Jedoch können Maximalflussprobleme mit mehreren Quellen und Senken sehr einfach auf das von uns behandelte Problem zurückgeführt werden.

Gegeben sei ein Digraph $D = (V, A)$ mit Bogenkapazitäten $c(a) \geq 0$ für alle $a \in A$. Ferner seien $S = \{s_1, \dots, s_p\} \subseteq V$ Quellen und $T = \{t_1, \dots, t_q\} \subseteq V$ Senken. Es gelte $S \cap T = \emptyset$. Ein zulässiger (S,T)-Fluss in D ist ein Vektor $x \in \mathbb{R}^A$ mit folgenden Eigenschaften

$$\begin{aligned} 0 \leq x_a \leq c_a \qquad & \text{für alle } a \in A \\ x(\delta^-(v)) = x(\delta^+(v)) \qquad & \text{für alle } v \in V \setminus (S \cup T). \end{aligned}$$

Der Wert eines zulässigen (S,T)-Flusses x ist definiert als

$$\text{val}(x) := \sum_{s \in S} (x(\delta^+(s)) - x(\delta^-(s))).$$

Die Bestimmung eines maximalen (S,T)-Flusses in D kann wie folgt auf ein Maximalflussproblem in einem Digraphen $D' = (V', A')$ mit einer Quelle und einer Senke zurückgeführt werden. Wir wählen zwei neue Knoten s, t und setzen

$$V' := V \cup \{s, t\}.$$

Der Knoten s ist die Quelle, t ist die Senke von D'. Ferner sei

$$\begin{aligned} A' &:= A \cup \{(s, s_i) \mid i = 1, \ldots, p\} \cup \{(t_i, t) \mid i = 1, \ldots, q\} \\ c'(a) &:= c(a) \text{ für alle } a \in A \\ c(a) &:= M \text{ für alle } a \in A' \setminus A. \end{aligned}$$

Es reicht z. B. $M := \sum_{a \in A} c(a) + 1$ zu wählen. Man überlegt sich sofort, dass jedem zulässigen (s,t)-Fluss in D' ein zulässiger (S,T)-Fluss in D mit gleichem Wert entspricht. Also liefert ein maximaler (s,t)-Fluss in D' einen maximalen (S,T)-Fluss in D.

Separationsalgorithmen Maximalfluss-Algorithmen spielen eine wichtige Rolle in sogenannten Schnittebenenverfahren für die ganzzahlige Optimierung. So treten etwa bei der Lösung von Travelling-Salesman-Problemen und Netzwerkentwurfsproblemen (Telekommunikation, Wasser- und Stromnetzwerke) Ungleichungen des Typs

$$\sum_{u \in W} \sum_{v \in V \setminus W} x_{uv} \geq f(w) \quad \forall W \subseteq V$$

auf, wobei $f(w)$ eine problemspezifische Funktion ist. Die Anzahl dieser Ungleichungen ist exponentiell in $|V|$. Häufig kann man jedoch in einer Laufzeit, die polynomial in $|V|$ ist, überprüfen, ob für einen gegebenen Vektor x^* alle Ungleichungen dieser Art erfüllt sind oder ob x^* eine der Ungleichungen verletzt. Algorithmen, die so etwas leisten, werden Separationsalgorithmen genannt. Beim TSP zum Beispiel können die „Schnittungleichungen“ durch Bestimmung eines kapazitätsminimalem Schnittes (mit Hilfe eines Maximalflussalgorithmus) überprüft werden.

Literaturverzeichnis

R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows, Handbooks in Operations Research and Management Science*, volume 1, chapter Optimization, pages 211–360. Elsevier, North-Holland, Amsterdam, 1989.

R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows, Theory, Algorithms and Applications*. Paramount Publishing International, Prentice Hall, New York, 1993.

M. O. Ball, T. L. Magnanti, C. L. Monma, and G. L. Nemhauser, editors. *Handbooks in Operations Research and Management Science*, volume 7: Network Models. North-Holland, Amsterdam, 1995a.

M. O. Ball, T. L. Magnanti, C. L. Monma, and G. L. Nemhauser, editors. *Handbooks in Operations Research and Management Science*, volume 8: Network Routing. North-Holland, Amsterdam, 1995b.

J. Edmonds and R. M. Karp. Theoretical improvement in algorithmic efficiency of network flow problems. *J. ACM*, (19):248–264, 1972.

L. R. Ford, Jr. and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

A. Frank. Connectivity and network flows. In R. L. Graham et al., editors, *Handbook of Combinatorics*, chapter 2, pages 111–177. North-Holland, Amsterdam, 1995.

J. L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, 1962.

A. F. P. Elias and C. E. Shannon. Note on maximum flow through a network. *IRE Trans. on Inform. Theory*, (2):117–119, 1956.

A. Schrijver. *Combinatorial Optimization – Polyhedra and Efficiency*. Springer-Verlag, Berlin, 2003.

E. T. V. Goldberg and R. E. Tarjan. Network flow algorithms. In in B. Korte et al. (eds.), editor, *Paths, Flows, and VLSI-Layout*. Springer-Verlag, Berlin, 1990.

7 Flüsse mit minimalen Kosten

Das im vorhergehenden Kapitel behandelte Maximalflussproblem ist eines der Basisprobleme der Netzwerkflusstheorie. Es gibt noch weitere wichtige und anwendungsreiche Netzwerkflussprobleme. Wir können hier jedoch aus Zeitgründen nur wenige dieser Probleme darstellen und analysieren. Der Leser, der an einer vertieften Kenntnis der Netzwerkflusstheorie interessiert ist, sei auf die am Anfang von Kapitel 6 erwähnten Bücher und Übersichtsartikel verwiesen.

7.1 Flüsse mit minimalen Kosten

Häufig tritt das Problem auf, durch ein Netzwerk nicht einen maximalen Fluss senden zu wollen, sondern einen Fluss mit vorgegebenem Wert, der bezüglich eines Kostenkriteriums minimale Kosten verursacht. Wir wollen hier nur den Fall einer linearen Kostenfunktion behandeln, obwohl gerade auch konkave und stückweise lineare Kosten (bei Mengenrabatten) eine wichtige Rolle spielen.

Sind ein Digraph $D = (V, A)$ mit Bogenkapazitäten $c(a) \geq 0$ für alle $a \in A$ und Kostenkoeffizienten $w(a)$ für alle $a \in A$ gegeben, sind $s, t \in V$ zwei verschiedene Knoten, und ist f ein vorgegebener Flusswert, dann nennt man die Aufgabe, einen zulässigen (s, t)-Fluss x mit Wert f zu finden, dessen Kosten $\sum_{a \in A} w_a x_a$ minimal sind, ein *Minimalkosten-Netzwerkflussproblem.* Analog zur LP-Formulierung (6.5) des Maximalflussproblems kann man ein Minimalkosten-Flussproblem als lineares Programm darstellen. Offenbar ist jede Optimallösung des linearen Programms (7.1) ein kostenminimaler c-kapazitierter (s, t)-Fluss mit Wert f.

$$
\begin{aligned}
\min \quad & \sum_{a \in A} w_a x_a \\
& x(\delta^-(v)) - x(\delta^+(v)) = 0 \quad \forall v \in V \setminus \{s, t\} \\
& x(\delta^-(t)) - x(\delta^+(t)) = f \\
& 0 \leq x_a \leq c_a \quad \forall a \in A
\end{aligned}
\tag{7.1}
$$

Minimalkosten-Flussprobleme kann man daher mit Algorithmen der linearen Optimierung lösen. In der Tat gibt es besonders schnelle Spezialversionen des Simplexalgorithmus für Probleme des Typs (7.1). Sie werden *Netzwerk-Simplexalgorithmen* genannt. Wir werden in Abschnitt 7.3 genauer auf diesen Algorithmentyp eingehen.

Es gibt viele kombinatorische Spezialverfahren zur Lösung von Minimalkosten-Flussproblemen. Alle „Tricks" der kombinatorischen Optimierung und Datenstrukturtechniken der Informatik werden benutzt, um schnelle Lösungsverfahren für (7.1) zu produzieren.

Ein Ende ist nicht abzusehen. Es gibt (zurzeit) kein global bestes Verfahren, weder bezüglich der beweisbaren Laufzeit, noch in Bezug auf Effizienz im praktischen Einsatz. Die Literatur ist allerdings voll mit Tabellen mit derzeitigen „Weltrekorden“ bezüglich der worst-case-Laufzeit unter speziellen Annahmen an die Daten. Alle derzeit gängigen Verfahren können — gut implementiert — Probleme des Typs (7.1) mit Zigtausenden von Knoten und Hunderttausenden oder gar Millionen von Bögen mühelos lösen.

Wir haben in dieser Vorlesung nicht genügend Zeit, um auf diese Details und Feinheiten einzugehen. Wir werden zunächst ein kombinatorisches Verfahren und die zugrundeliegende Theorie vorstellen. Um den Algorithmus und den Satz, auf dem seine Korrektheit beruht, darstellen zu können, führen wir einige neue Begriffe ein.

Sei x ein zulässiger (s,t)-Fluss in D und sei C ein (nicht notwendigerweise gerichteter) Kreis in D. Diesem Kreis C können wir offenbar zwei Orientierungen geben. Ist eine Orientierung von C gewählt, so nennen wir einen Bogen auf C, der in Richtung der Orientierung verläuft, *Vorwärtsbogen*, andernfalls nennen wir ihn *Rückwärtsbogen*. Ein Kreis C heißt *augmentierend* bezüglich x, wenn es eine Orientierung von C gibt, so dass $x_a < c_a$ für alle Vorwärtsbögen $a \in C$ und dass $0 < x_a$ für alle Rückwärtsbögen $a \in C$ gilt (vergleiche Definition (6.8)). Ein Kreis kann offenbar bezüglich beider, einer oder keiner Richtung augmentierend sein. Sprechen wir von einem augmentierenden Kreis C, so unterstellen wir fortan, dass eine Orientierung von C fest gewählt ist, bezüglich der C augmentierend ist.

Die Summe der Kostenkoeffizienten der Vorwärtsbögen minus der Summe der Kostenkoeffizienten der Rückwärtsbögen definieren wir als die *Kosten eines augmentierenden Kreises*. (Wenn ein Kreis in Bezug auf beide Orientierungen augmentierend ist, können die beiden Kosten verschieden sein!) Das zentrale Resultat dieses Abschnitts ist das Folgende.

(7.2) Satz. *Ein zulässiger (s,t)-Fluss x in D mit Wert f hat genau dann minimale Kosten, wenn es bezüglich x keinen augmentierenden Kreis mit negativen Kosten gibt.* △

Beweis. Wir beweisen zunächst nur die triviale Richtung. (Satz (7.6) liefert die Rückrichtung.) Gibt es einen augmentierenden Kreis C bezüglich x, so setzen wir:

$$\varepsilon := \min \begin{cases} c_{ij} - x_{ij} & (i,j) \in C \text{ Vorwärtsbogen,} \\ x_{ij} & (i,j) \in C \text{ Rückwärtsbogen.} \end{cases} \tag{7.3}$$

Definieren wir

$$x'_{ij} := \begin{cases} x_{ij} + \varepsilon & \text{falls } (i,j) \in C \text{ Vorwärtsbogen,} \\ x_{ij} - \varepsilon & \text{falls } (i,j) \in C \text{ Rückwärtsbogen,} \\ x_{ij} & \text{falls } (i,j) \in A \setminus C, \end{cases} \tag{7.4}$$

dann ist $x' \in \mathbb{R}^A$ trivialerweise ein zulässiger (s,t)-Fluss mit Wert $\text{val}(x') = f$. Hat der augmentierende Kreis C negative Kosten $\gamma < 0$, dann gilt offenbar $\sum_{(i,j)\in A} w_{ij} x'_{ij} = \sum_{(i,j)\in A} w_{ij} x_{ij} + \varepsilon\gamma$. Gibt es also einen augmentierenden Kreis bezüglich x mit negativen Kosten, dann kann x nicht kostenminimal sein. □

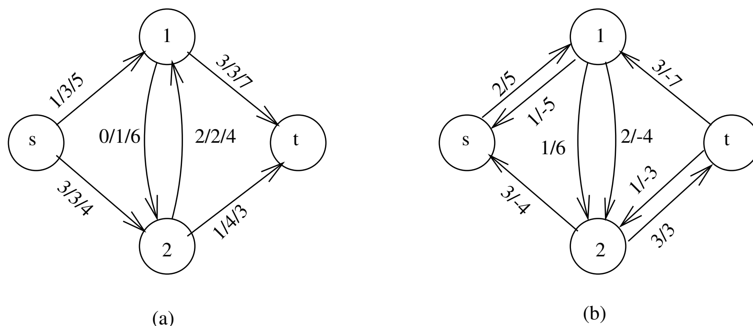


Abbildung 7.1: Ein Digraph D mit Fluss x und ein augmentierendes Netzwerk bzgl. x.

Um die umgekehrte Richtung zu beweisen, müssen wir etwas mehr Aufwand treiben, den wir allerdings direkt bei der Darstellung des Algorithmus benutzen können. Ist x ein zulässiger (s,t)-Fluss mit Wert f, dann definieren wir einen Digraphen (genannt *augmentierendes Netzwerk bezüglich* x) $N = (V, \overline{A}, \overline{c}, \overline{w})$ wie folgt: Es sei

$$A_1 := \{(u,v) \in A \mid x_{uv} < c_{uv}\}, \quad A_2 := \{(v,u) \mid (u,v) \in A \text{ und } x_{uv} > 0\}.$$

Ist $a \in A$, so schreiben wir a_1 bzw. a_2 um den zugehörigen Bogen aus A_1 bzw. A_2 zu bezeichnen. Schreiben wir für $a \in A$ eine Formel wie etwa $x'(a) = x(a) + \overline{x}(a_1) - \overline{x}(a_2)$ und ist einer der Bögen a_1 bzw. a_2 nicht definiert (d. h. es gilt entweder $x_a = c_a$ oder $x_a = 0$), dann ist der Wert $\overline{x}(a_1)$ bzw. $\overline{x}(a_2)$ als Null zu betrachten. Wir setzen $\overline{A} := A_1 \dot{\cup} A_2$ (man beachte, dass $\overline{A}$ parallele Bögen enthalten kann). Ferner sei für $\overline{a} \in \overline{A}$

$$\overline{c}(\overline{a}) := \begin{cases} c(a) - x(a) & \text{falls } \overline{a} = a_1, \\ x(a) & \text{falls } \overline{a} = a_2, \end{cases}$$

$$\overline{w}(\overline{a}) := \begin{cases} w(a) & \text{falls } \overline{a} = a_1, \\ -w(a) & \text{falls } \overline{a} = a_2. \end{cases}$$

In Abbildung 7.1(a) ist ein Digraph mit einem (s,t)-Fluss x des Wertes 4 dargestellt. Die drei Zahlen bei einem Bogen a geben an: Fluss durch a / Kapazität von a / Kosten von a. Das augmentierende Netzwerk N bezüglich D und x ist in 7.1(b) gezeichnet. Die beiden Zahlen bei einem Bogen a in 7.1(b) geben an: Kapazität von a / Kosten von a.

Der (ungerichtete) Kreis $\{(2,t),(1,t),(2,1)\}$ in Abb. 7.1(a) ist ein augmentierender Kreis mit Kosten $3 - 7 - 4 = -8$. Dieser Kreis entspricht in (b) dem gerichteten Kreis $\{(2,t),(t,1),(1,2)\}$, wobei von den beiden zwischen 1 und 2 parallel verlaufenden Bögen natürlich der mit negativen Kosten zu wählen ist. Aufgrund der Konstruktion von N ist folgende Beziehung offensichtlich:

(7.5) Lemma. *Ist $D = (V,A)$ ein Digraph mit Kapazitäten $c \in \mathbb{R}^A_+$ und Kosten $w \in \mathbb{R}^A$, ist x ein zulässiger (s,t)-Fluss in D, und ist $N = (V, \overline{A}, \overline{c}, \overline{w})$ das zu D und x gehörige augmentierende Netzwerk, dann entspricht jeder augmentierende Kreis in D genau einem gerichteten Kreis in N. Die Kosten eines augmentierenden Kreises in D stimmen überein mit der Summe der Kostenkoeffizienten des zugehörigen gerichteten Kreises in N.* △

Damit ist unser Exkurs zur Definition von augmentierenden Netzwerken beendet. Wir formulieren nun Theorem (7.2) unter Benutzung dieses neuen Konzepts um.

(7.6) Satz. *Sei x ein zulässiger (s,t)-Fluss in D mit Wert f, und $N = (V, \overline{A}, \overline{c}, \overline{w})$ sei das bezüglich x und D augmentierende Netzwerk, dann gilt folgendes. Der Fluss x ist unter allen zulässigen (s,t)-Flüssen in D mit Wert f genau dann kostenminimal, wenn es in N keinen gerichteten Kreis mit negativen Kosten gibt.* △

Beweis. Gibt es in N einen gerichteten Kreis mit negativen Kosten, so folgt analog zum Beweis des einfachen Teils von (7.2), dass x nicht minimal ist.

Nehmen wir umgekehrt an, dass x nicht kostenminimal ist, dann müssen wir in N einen gerichteten Kreis mit negativen Kosten finden. Sei also x' ein zulässiger (s,t)-Fluss in D mit Wert f und $w^T x' < w^T x$. Für jeden Bogen $\overline{a} \in \overline{A}$ setzen wir

$$\overline{x}(\overline{a}) := \begin{cases} \max\{0, x'(a) - x(a)\}, & \text{falls } \overline{a} = a_1 \in A_1 \\ \max\{0, x(a) - x'(a)\}, & \text{falls } \overline{a} = a_2 \in A_2 \end{cases}$$

Diese Definition ist genau so gewählt, dass gilt

$$\overline{x}(a_1) - \overline{x}(a_2) = x'(a) - x(a), \quad \forall\ a \in A.$$

Wir weisen zunächst einige Eigenschaften von $\overline{x} \in \mathbb{R}^{\overline{A}}$ nach.

Behauptung 1. $\overline{x}$ ist ein zulässiger (s,t)-Fluss in N mit Wert 0 und $\overline{w}^T \overline{x} < 0$. △

Beweis. Wir zeigen zunächst, dass $\overline{x}$ negative Kosten hat. Für alle $a \in A$ gilt offenbar

$$\overline{x}(a_1)\overline{w}(a_1) + \overline{x}(a_2)\overline{w}(a_2) = (x'(a) - x(a))w(a),$$

und daraus folgt:

$$\begin{aligned} \sum_{\overline{a} \in \overline{A}} \overline{w}(\overline{a})\overline{x}(\overline{a}) &= \sum_{a \in A} (\overline{w}(a_1)\overline{x}(a_1) + \overline{w}(a_2)\overline{x}(a_2)) \\ &= \sum_{a \in A} (x'(a) - x(a))w(a) \\ &= w^T x' - w^T x < 0. \end{aligned}$$

Der Vektor $\overline{x}$ erfüllt nach Definition die Kapazitätsbedingungen $0 \leq \overline{x}(\overline{a}) \leq \overline{c}(\overline{a})$. Wir

zeigen nun, dass $\overline{x}$ auch die Flusserhaltungsbedingungen für alle $v \in V$ erfüllt.

$$\begin{aligned}
\overline{x}(\delta_N^+(v)) - \overline{x}(\delta_N^-(v)) &= \sum_{a\in\delta^+(v)} (\overline{x}(a_1) - \overline{x}(a_2)) - \sum_{a\in\delta^-(v)} (\overline{x}(a_1) - \overline{x}(a_2)) \\
&= \sum_{a\in\delta^+(v)} (x'(a) - x(a)) - \sum_{a\in\delta^-(v)} (x'(a) - x(a)) \\
&= \Big(\sum_{a\in\delta^+(v)} x'(a) - \sum_{a\in\delta^-(v)} x'(a)\Big) \\
&\quad - \Big(\sum_{a\in\delta^+(v)} x(a) - \sum_{a\in\delta^-(v)} x(a)\Big) \\
&= \begin{cases} 0 - 0 & \text{falls } v \in V \setminus \{s,t\} \\ \operatorname{val}(x') - \operatorname{val}(x) & \text{falls } v = s \\ -\operatorname{val}(x') + \operatorname{val}(x) & \text{falls } v = t \end{cases} \\
&= 0.
\end{aligned}$$

Daraus folgt, dass $\overline{x}$ die Flußerhaltungsbedingungen erfüllt und dass $\operatorname{val}(\overline{x}) = 0$ gilt. Damit ist Behauptung 1 bewiesen. □

Behauptung 2. Es gibt zulässige (s,t)-Flüsse $x_1, \ldots, x_k \in \mathbb{R}^{\overline{A}}$, $k \leq |\overline{A}|$, so dass folgendes gilt:

(a) $\overline{x}(\overline{a}) = \sum_{i=1}^{k} x_i(\overline{a})$ für alle $\overline{a} \in \overline{A}$.

(b) Für jeden (s,t)-Fluss x_i, $i \in \{1, \ldots, k\}$ gibt es einen gerichteten Kreis $\overline{C}_i$ in N und eine positive Zahl α_i, so dass $x_i(\overline{a}) = \alpha_i$ für alle $\overline{a} \in \overline{C}_i$ und $x_i(\overline{a}) = 0$ für alle $\overline{a} \in \overline{A} \setminus \overline{C}_i$. △

Beweis. Sei p die Anzahl der Bögen $\overline{a} \in \overline{A}$ mit $\overline{x}(\overline{a}) \neq 0$. Da $x \neq x'$ gilt $p \geq 1$. Sei v_0 ein Knoten, so dass ein Bogen $(v_0, v_1) \in \overline{A}$ existiert mit $\overline{x}((v_0, v_1)) \neq 0$. Da in v_1 die Flusserhaltungsbedingung gilt, muss es einen Bogen $(v_1, v_2) \in \overline{A}$ geben mit $\overline{x}((v_1, v_2)) \neq 0$. Fahren wir so weiter fort, so erhalten wir einen gerichteten Weg $v_0, v_1, v_2, \ldots$. Da N endlich ist, muss irgendwann ein Knoten auftreten, der schon im bisher konstruierten Weg enthalten ist. Damit haben wir einen gerichteten Kreis $\overline{C}_p$ gefunden. Sei α_p der kleinste Wert $\overline{x}(\overline{a})$ der unter den Bögen $\overline{a}$ des Kreises $\overline{C}_p$ auftritt. Definieren wir

$$x_p(\overline{a}) := \begin{cases} \alpha_p & \text{für alle } \overline{a} \in \overline{C}_p, \\ 0 & \text{sonst,} \end{cases}$$

so ist $x_p \in \mathbb{R}^{\overline{A}}$ ein (s,t)-Fluss mit Wert 0. Setzen wir nun $\overline{x}_p := \overline{x} - x_p$, so ist $\overline{x}_p$ ein (s,t)-Fluss mit Wert 0, und die Zahl der Bögen $\overline{a} \in \overline{A}$ mit $\overline{x}_p(\overline{a}) \neq 0$ ist kleiner als p. Führen wir diese Konstruktion iterativ fort bis $\overline{x}_p = 0$ ist, so haben wir die gesuchten Kreise gefunden. □

Damit können wir den Beweis von (7.6) beenden. Für die nach Behauptung 2 existierenden (s,t)-Flüsse x_i gilt offenbar

$$\overline{w}^T\overline{x} = \sum_{i=1}^{k} \overline{w}^T x_i.$$

Da $\overline{w}^T\overline{x} < 0$ nach Behauptung 1 ist, muss einer der Werte $\overline{w}^T x_i$ kleiner als Null sein, dass heißt, wir haben in N einen gerichteten Kreis mit negativen Kosten gefunden. □

Satz (7.2) bzw. Satz (7.6) sind Optimalitätskriterien für zulässige (s,t)-Flüsse. Man kann beide Aussagen — was algorithmisch noch wichtiger ist — benutzen, um zu zeigen, dass Kostenminimalität erhalten bleibt, wenn man entlang Wegen minimaler Kosten augmentiert.

(7.7) Satz. *Sei $D = (V,A)$ ein Digraph mit gegebenen Knoten $s,t \in V$, Kapazitäten $c \in \mathbb{R}^A_+$ und Kosten $w \in \mathbb{R}^A$. Sei x ein zulässiger (s,t)-Fluss in D mit Wert f, der kostenminimal unter allen (s,t)-Flüssen mit Wert f ist, und sei $N = (V,\overline{A},\overline{c},\overline{w})$ das zugehörige augmentierende Netzwerk. Sei P ein (s,t)-Weg in N mit minimalen Kosten $\overline{w}(P)$, und sei $\overline{x}$ ein zulässiger (s,t)-Fluss in N, so dass $\overline{x}(\overline{a}) > 0$ für alle $\overline{a} \in P$ und $\overline{x}(\overline{a}) = 0$ für alle $\overline{a} \in \overline{A} \setminus P$, dann ist der Vektor $x' \in \mathbb{R}^A$ definiert durch*

$$x'(a) := x(a) + \overline{x}(a_1) - \overline{x}(a_2) \quad \text{für alle } a \in A$$

ein zulässiger (s,t)-Fluss in D mit Wert $f+\text{val}(\overline{x})$, der kostenminimal unter allen Flüssen dieses Wertes in D ist. △

Beweis. Trivialerweise ist $x' \in \mathbb{R}^A$ ein zulässiger (s,t)-Fluss mit Wert $f + \text{val}(\overline{x})$. Wir zeigen, dass x' kostenminimal ist. Angenommen, dies ist nicht der Fall, dann gibt es nach Satz (8.6) einen negativen gerichteten Kreis C' im bezüglich x' augmentierenden Netzwerk $N' = (V,A',c',w')$. Wir beweisen, dass dann auch ein negativer gerichteter Kreis in N bezüglich x existiert.

Wir bemerken zunächst, dass das augmentierende Netzwerk N' aus N dadurch hervorgeht, dass die Bögen aus $P \subseteq \overline{A}$ neue Kapazitäten erhalten und möglicherweise ihre Richtung und damit ihre Kosten ändern. Sei $\overline{B} \subseteq P$ die Menge der Bögen aus $\overline{A}$, die in N' eine andere Richtung als in N haben, und sei $B' := \{(v,u) \in A' \mid (u,v) \in \overline{B}\}$.

Wir untersuchen nun den gerichteten Kreis $C' \subseteq A'$ in N' mit negativen Kosten. Gilt $C' \cap B' = \emptyset$, so ist C' in $\overline{A}$ enthalten und somit ein negativer Kreis in N. Dann wäre x nach (8.6) nicht kostenoptimal, was unserer Voraussetzung widerspricht. Wir können daher annehmen, dass $C' \cap B' \neq \emptyset$ gilt.

Der Beweis verläuft nun wie folgt. Wir konstruieren aus dem (s,t)-Weg P und dem gerichteten Kreis C' einen (s,t)-Weg $Q \subseteq \overline{A}$ und einen gerichteten Kreis $K' \subseteq A'$ mit den Eigenschaften

$$\overline{w}(P) + w'(C') \geq \overline{w}(Q) + w'(K')$$
$$|C' \cap B'| > |K' \cap B'|.$$

Durch iterative Wiederholung dieser Konstruktion erhalten wir nach höchstens $|B'|$ Schritten einen (s,t)-Weg in N und einen gerichteten Kreis in N', dessen Kosten negativ sind und der keinen Bogen aus B' enthält. Folglich ist dieser Kreis ein negativer Kreis in N. Widerspruch!

Die Konstruktion verläuft wie folgt. Da $C' \cap B' \neq \emptyset$, gibt es einen Bogen $(u,v) \in P$ mit $(v,u) \in C'$. Wir wählen denjenigen Bogen $(u,v) \in P$, der auf dem Weg von s nach t entlang P der erste Bogen mit $(v,u) \in C'$ ist. Wir unterscheiden zwei Fälle.

Fall 1: $C' \cap B'$ enthält mindestens zwei Bögen.
Sei (y,x) der nächste Bogen auf dem gerichteten Kreis C' nach (v,u), der in B' ist. Wir konstruieren einen (s,t)-Pfad $\overline{P} \subseteq \overline{A}$ wie folgt. Wir gehen von s aus entlang P nach u, dann von u entlang C' nach y und von y entlang P nach t. Starten wir nun in v, gehen entlang P zu x und dann entlang C' nach v, so erhalten wir eine geschlossene gerichtete Kette $P' \subseteq A'$. Aus $\overline{w}_{uv} = -w'_{vu}$ und $\overline{w}_{xy} = -w'_{yx}$ folgt, dass $\overline{w}(P) + w'(C') = \overline{w}(\overline{P}) + w'(P')$ gilt. $\overline{P}$ ist die Vereinigung von gerichteten Kreisen $C'_1, \dots, C'_k \subseteq \overline{A}$ mit einem gerichteten (s,t)-Weg $Q \subseteq \overline{A}$, und P' ist die Vereinigung von gerichteten Kreisen $C'_{k+1}, \dots, C'_r \subseteq A'$. Da P kostenminimal in N ist, gilt $\overline{w}(P) \leq \overline{w}(Q)$, und somit gibt es wegen $\overline{w}(\overline{P}) + w'(P') = \overline{w}(Q) + \sum\limits_{i=1}^{k} \overline{w}(C'_i) + \sum\limits_{i=k+1}^{r} w'(C'_i)$ mindestens einen gerichteten Kreis in A', sagen wir K', der negative Kosten hat. Nach Konstruktion gilt $|K' \cap B'| < |C' \cap B'|$.

Fall 2: Der Bogen (u,v) ist der einzige Bogen auf P mit $(v,u) \in C' \cap B'$.
Wir konstruieren einen gerichteten (s,t)-Pfad $\overline{P} \subseteq \overline{A}$ wie folgt. Wir starten in s und folgen P bis u, dann folgen wir dem gerichteten Weg von u entlang C' bis v und dann wieder dem gerichteten Weg von v entlang P bis t. Offenbar ist $\overline{P}$ in $\overline{A}$ enthalten und ein gerichteter (s,t)-Pfad in N. Aus $\overline{w}_{uv} = -w'_{vu}$ folgt direkt $\overline{w}(P) + w'(C') = \overline{w}(\overline{P})$. Der gerichtete (s,t)-Pfad $\overline{P}$ ist die Vereinigung eines (s,t)-Weges Q und einiger gerichteter Kreise $C'_1, \dots, C'_k$. Da P ein (s,t)-Weg in N mit minimalen Kosten ist, gilt $\overline{w}(Q) \geq \overline{w}(P)$, und aus $\overline{w}(Q) = \overline{w}(\overline{P}) - \sum_{i=1}^{k} \overline{w}(C'_1)$ folgt, dass mindestens einer der Kreise C'_i negativ ist. Da alle C'_i in $\overline{A}$ enthalten sind, enthält $\overline{A}$ einen negativen Kreis. Widerspruch! □

Damit können wir nun einen Algorithmus zur Lösung des Minimalkosten-Flussproblems angeben.

(7.8) Algorithmus.

Eingabe: Digraph $D = (V, A)$, mit Kapazitäten $c \in \mathbb{R}^A_+$ und Kosten $w \in \mathbb{R}^A$, zwei verschiedene Knoten $s, t \in V$ und ein Flusswert f.

Ausgabe: Ein zulässiger (s,t)-Fluss x mit Wert f, der kostenminimal unter allen zulässigen (s,t)-Flüssen mit Wert f ist, oder die Aussage, dass kein zulässiger (s,t)-Fluss mit Wert f existiert.

1. Setze $x(a) = 0$ für alle $a \in A$ (bzw. starte mit einem zulässigen (s,t)-Fluss mit Wert nicht größer als f).

2. Konstruiere das augmentierende Netzwerk $N = (V, \overline{A}, \overline{c}, \overline{w})$ bezüglich D und x.
3. Wende einen Kürzeste-Wege-Algorithmus (z. B. den Floyd-Algorithmus (5.25)) an, um im Digraphen $N = (V, \overline{A})$ mit den "Bogenlängen" $\overline{w}(\overline{a})$, $\overline{a} \in \overline{A}$, einen negativen gerichteten Kreis C zu finden. Gibt es keinen, dann gehe zu 5.
4. (Augmentierung entlang C)

   Bestimme $\varepsilon := \min\{\overline{c}(\overline{a}) \mid \overline{a} \in C\}$, setze für $a \in A$

$$x(a) := \begin{cases} x(a) + \varepsilon & \text{falls } a_1 \in C \\ x(a) - \varepsilon & \text{falls } a_2 \in C \\ x(a) & \text{andernfalls} \end{cases}$$

   und gehe zu 2. (Hier erhalten wir einen Fluss mit gleichem Wert und geringeren Kosten.)
5. Ist $\text{val}(x) = f$, STOP, gib x aus.
6. Bestimme mit einem Kürzeste-Wege-Algorithmus (z. B. einer der Varianten des Moore-Bellman-Verfahrens (5.20), es gibt keine negativen Kreise!) einen (s,t)-Weg P in N mit minimalen Kosten $\overline{w}(P)$.
7. Gibt es in N keinen (s,t)-Weg, dann gibt es in D keinen zulässigen (s,t)-Fluss mit Wert f, STOP.
8. (Augmentierung entlang P)

   Bestimme $\varepsilon' := \min\{\overline{c}(\overline{a}) \mid \overline{a} \in P\}$, $\varepsilon := \min\{\varepsilon', f - \text{val}(x)\}$, setze für $a \in A$

$$x(a) := \begin{cases} x(a) + \varepsilon & \text{falls } a_1 \in P \\ x(a) - \varepsilon & \text{falls } a_2 \in P \\ x(a) & \text{andernfalls,} \end{cases}$$

   konstruiere das bzgl. x und D augmentierende Netzwerk $N = (V, \overline{A}, \overline{c}, \overline{w})$ und gehe zu 5. △

Die Korrektheit des Algorithmus folgt unmittelbar aus (7.6) und (7.7). Wir wollen nun die Laufzeit abschätzen. Hat D nur nichtnegative Kosten $w(a)$ bzw. enthält D keinen augmentierenden Kreis mit negativen Kosten, so ist der Nullfluss eine kostenoptimaler Fluss mit Wert Null, und die Schleife über die Schritte 2, 3 und 4 braucht nicht durchlaufen zu werden. Sind alle Kapazitäten ganzzahlig, so wird der Flusswert in Schritt 8 um jeweils mindestens eine Einheit erhöht. Also sind höchstens f Aufrufe einen Kürzesten-Wege-Algorithmus erforderlich.

(7.9) Satz. *Ist $D = (V, A)$ ein Digraph mit ganzzahligen Kapazitäten $c(a)$ und nichtnegativen Kosten $w(a)$, und sind s, t zwei verschiedene Knoten und $f \in \mathbb{Z}_+$ ein vorgegebener Flußwert, so findet Algorithmus (7.8) in $\mathcal{O}(f|V|^3)$ Schritten einen kostenminimalen zulässigen (s,t)-Fluss mit Wert f, falls ein solcher existiert.* △

Der Algorithmus ist in dieser Form nicht polynomial, da seine Laufzeit polynomial in der Kodierungslänge $\langle f \rangle$ sein müsste. Ferner ist nicht unmittelbar klar, wie lange er läuft, wenn negative Kosten erlaubt sind, da die Anzahl der Kreise mit negativen Kosten, auf denen der Fluss verändert werden muß, nicht ohne weiteres abgeschätzt werden kann. Diese Schwierigkeiten können durch neue Ideen (Augmentierung entlang Kreisen mit minimalen durchschnittlichen Kosten $\overline{w}(C)/|C|$, Skalierungstechniken) überwunden werden, so dass Versionen von Algorithmus (7.8) existieren, die polynomiale Laufzeit haben. Aus Zeitgründen können diese Techniken hier nicht dargestellt werden. Es sei hierzu wiederum auf die schon mehrfach erwähnten Übersichtsartikel und das Buch von Ahuja et al. (1993) verwiesen, die auch ausführlich auf die historische Entwicklung eingehen. Der Aufsatz von M. Shigeno and McCormick (2000) präsentiert zwei Skalierungsmethoden und gibt dabei eine gute Vergleichsübersicht über viele der bekannten Min-Cost-Flow-Algorithmen.

7.2 Transshipment-, Transport- u. Zuordnungsprobleme

Wie beim Maximalflussproblem ist es natürlich möglich, Minimalkosten-Flussprobleme, bei denen mehrere Quellen und Senken vorkommen und bei denen von Null verschiedene untere Kapazitätsschranken für die Bögen auftreten, zu lösen. Sie können auf einfache Weise auf das Standardproblem (7.1) reduziert werden.

Es gibt noch eine Reihe von Varianten des Minimalkosten-Flussproblems, die in der Literatur große Beachtung gefunden und viele Anwendungen haben. Ferner gibt es für alle dieser Probleme Spezialverfahren zu ihrer Lösung. Aus Zeitgründen können wir diese nicht behandeln. Wir wollen diese Probleme jedoch zumindest aus „Bildungsgründen" erwähnen und zeigen, wie sie in Minimalkosten-Flussprobleme transformiert werden können.

(7.10) Transshipment-Probleme (Umladeprobleme). Gegeben sei ein Digraph $D = (V, A)$, dessen Knotenmenge zerlegt sei in drei disjunkte Teilmengen V_a, V_n und V_u. Die Knoten aus V_a bezeichnen wir als *Angebotsknoten*. (Bei ihnen fließt ein Strom in das Netzwerk ein.) Die Knoten V_n bezeichnen wir als *Nachfrageknoten* (bei ihnen verläßt der Strom das Netz), und die Knoten V_u werden als *Umladeknoten* bezeichnet (hier wird der Fluss erhalten). Jedem Bogen $a \in A$ sind eine Kapazität $c(a)$ und ein Kostenkoeffizient $w(a)$ zugeordnet. Ferner sei bei jedem Angebotsknoten $v \in V_a$ die Menge $a(v)$ verfügbar, und bei jedem Nachfrageknoten die Menge $b(v)$ erwünscht. Die Aufgabe, einen Plan zu ermitteln, der Auskunft darüber gibt, von welchen Anbietern aus über welche Transportwege der Bedarf der Nachfrager zu decken ist, damit die Kosten für alle durchzuführenden Transporte minimiert werden, heißt *Umladeproblem*. $\triangle$

Offenbar kann man ein Umladeproblem wie in (7.10) angegeben als lineares Programm

schreiben.

$$
\begin{aligned}
\min \quad & \sum w(a)x(a) \\
x(\delta^-(v)) - x(\delta^+(v)) &= \begin{cases} 0 & \text{falls } v \in V_u \\ b(v) & \text{falls } v \in V_n \\ -a(v) & \text{falls } v \in V_a \end{cases} \\
0 \leq x(a) &\leq c(a) \quad \forall a \in A.
\end{aligned}
\tag{7.11}
$$

Problem (7.10) ist offensichtlich höchstens dann lösbar, wenn $\sum_{v \in V_n} b(v) = \sum_{v \in V_a} a(v)$ gilt. Das lineare Programm (7.11) kann in ein Minimalkosten-Flussproblem (7.1) wie folgt transformiert werden. Wir führen eine (künstliche) Quelle s und eine künstliche Senke t ein. Die Quelle s verbinden wir mit jedem Knoten $v \in V_a$ durch einen Bogen (s, v) mit Kosten Null und Kapazität $a(v)$. Jeden Knoten $v \in V_n$ verbinden wir mit der Senke t durch einen Bogen (v, t) mit Kosten null und Kapazität $b(v)$. Offenbar liefert der kostenminimale (s, t)-Fluss in diesem neuen Digraphen mit Wert $\sum_{v \in V_n} b(v)$ eine optimale Lösung des Umladeproblems.

(7.12) Transportprobleme. Ein Transshipment-Problem, bei dem $V_u = \emptyset$ gilt, heißt Transportproblem. Hier wird also von Erzeugern direkt zu den Kunden geliefert, ohne den Zwischenhandel einzuschalten. △

(7.13) Zuordnungsproblem. Ein Transportproblem, bei dem $|V_a| = |V_n|$, $b(v) = 1$ für alle $v \in V_n$ und $a(v) = 1 \; \forall \, v \in V_a$ gilt, heißt Zuordnungsproblem (vergleiche (3.9)). △

Zur Lösung von Zuordnungs- und Transportproblemen gibt es besonders schnelle Algorithmen, die erheblich effizienter als die Algorithmen zur Bestimmung kostenminimaler Flüsse sind. Näheres hierzu wird in den Übungen behandelt. Aber auch in diesem Bereich kann man nicht — wie bei Algorithmen zur Bestimmung von Flüssen mit Minimalkosten – davon sprechen, dass irgendein Verfahren das schnellste (in der Praxis) ist. Immer wieder gibt es neue Implementationstechniken, die bislang unterlegene Algorithmen erheblich beschleunigen und anderen Verfahren überlegen erscheinen lassen. Siehe hierzu (Ahuja et al., 1993).

Wir behandeln hier das Zuordnungsproblem (englisch: assignment problem) stiefmütterlich als Spezialfall des Minimalkosten-Flussproblems. Seine Bedeutung für die Entwicklung der Algorithmen zur Lösung kombinatorischer Optimierungsprobleme ist jedoch erheblich. Im Jahr 1955 veröffentlichte Harold Kuhn (Kuhn, 1955) einen Algorithmus, den er, um die Bedeutung von Ideen zweier ungarischer Mathematiker (D. König und J. Egerváry) hervorzuheben, „ungarische Methode“ nannte. Dieser Algorithmus ist ein Vorläufer der heute so genannten „Primal-Dual-Verfahren“ und erwies sich (nach einer Modifikation von Munkres) als polynomialer Algorithmus zur Lösung von Zuordnungsproblemen. Der ungarische Algorithmus war Vorbild für viele andere Methoden zur Lösung kombinatorischer Optimierungsprobleme, und er wurde so oft zitiert, dass der Aufsatz (Kuhn, 1955) von der Zeitschrift *Naval Logistics Quarterly* im Jahre 2004 als wichtigstes Paper gewählt wurde, das seit Bestehen der Zeitschrift in dieser erschienen ist.

Vollkommen überraschend war die Entdeckung eines französischen Mathematikers im Jahre 2006, dass die ungarische Methode bereits um 1850 von Carl Gustav Jacob Jacobi (1804 in Potsdam – 1851 in Berlin) beschrieben wurde. Siehe hierzu Abschnitt 2 in (Grötschel, 2008) und die URLs:

`http://www.lix.polytechnique.fr/~ollivier/JACOBI/jacobiEngl.htm`
`http://en.wikipedia.org/wiki/Hungarian_method`
`http://www.zib.de/groetschel/pubnew/paper/groetschel2008_pp.pdf`

7.3 Der Netzwerk-Simplex-Algorithmus

Wie schon in der Einführung erwähnt kann der Simplexalgorithmus zur Lösung allgemeiner linearer Optimierungsprobleme zur Lösung des Minimalkosten-Flussproblems spezialisiert werden. Diese *Netzwerk-Simplex-Algorithmen* nutzen die spezielle Struktur des Minimalkosten-Flussproblems aus, um die Operationen des Simplexalgorithmus besonders effizient umzusetzen. Die Grundstruktur eines Netzwerk-Simplex-Algorithmus kann jedoch auch ohne Kenntnis des Simplexalgorithmus verstanden werden. Betrachten wir dazu das Minimalkosten-Flussproblem in der allgemeinen Form

$$\begin{aligned} \min \quad & c^T x \\ x(\delta^-(i)) - x(\delta^+(i)) &= b_i \quad \forall i \in V, && (7.14)\\ l_a \le x_a &\le u_a \quad \forall a \in A, && (7.15) \end{aligned}$$

für einen Digraphen $D = (V = \{1, \dots, n\}, A)$ mit n Knoten und m Bögen. Die Bogenbewertung $c\colon A \to \mathbb{R}$ definiert „Kosten“, während die Bogenbewertungen $l, u\colon A \to \mathbb{R}_+$ untere bzw. obere Schranken für den Fluss auf einem Bogen angeben und damit die Kapazität des Bogens definieren. Die Knotenbewertung $b\colon V \to \mathbb{R}$ beschreibt die Einspeisung ($b_i \geq 0$, $i \in V$) bzw. Ausspeisung ($b_i \leq 0$, $i \in V$) an den Knoten. Wir nehmen in diesem Abschnitt an, dass D zusammenhängend ist und die Beziehung $\sum_{i \in V} b_i = 0$ gilt. In Analogie zur Terminilogie für (s,t)-Flüsse bezeichnen wir einen Vektor $x \in \mathbb{R}^A$, der (7.14) erfüllt, als *Fluss*. Erfüllt x zusätzlich (7.15), so sprechen wir von einem *zulässigen Fluss*.

(Optimale) Zulässige Flüsse des Minimalkosten-Flussproblems lassen sich kombinatorisch über aufspannende Bäume beschreiben, die den Fluss definieren.

(7.16) Definition (Baum-Lösung). *Ein Flussvektor x zusammen mit einem aufspannenden Baum T für $D = (V, A)$ und Bogenmengen $L, U \subseteq A$ heißt* Baum-Lösung, *wenn gilt:*

- *T, L und U sind eine Partition von A,*
- *$x_a = l_a$ für alle $a \in L$,*
- *$x_a = u_a$ für alle $a \in U$.*

Erfüllt (x, T, L, U) außerdem alle Bedingungen $l \leq x \leq u$ (d. h. auch für die Bögen in T), so heißt (x, T, L, U) zulässige Baum-Lösung. △

Wie wir später sehen werden (Satz (7.24)) genügt es, sich bei der Suche nach kostenminimalen Flüssen auf Baum-Lösungen zu beschränken. Baum-Lösungen haben die Eigenschaft, dass sich die Flusswerte auf den Nicht-Baum-Kanten über die Flusserhaltungsbedingungen aus den Flusswerten der Baum-Kanten eindeutig ergeben.

(7.17) Lemma. *Sei (T, L, U) eine Partition von A und T ein aufspannender Baum für $D = (V, A)$. Dann existiert ein (nicht unbedingt zulässiger) eindeutiger Fluss x mit $x_a = l_a$ für alle $a \in L$ und $x_a = u_a$ für alle $a \in U$.* △

Beweis. Sei i ein Blatt von T und $a \in T$ der mit i inzidente Bogen sowie $M := L \cup U$ die Menge der Bögen mit festem Flusswert. Da i ein Blatt ist, sind alle anderen mit i inzidenten Bögen in M enthalten. Auf der linken Seite der Flusserhaltungsbedingung $x(\delta^-(i)) - x(\delta^+(i)) = b_i$ sind daher alle Terme außer x_a konstant, d. h. der Wert von x_a ist durch diese Gleichung eindeutig bestimmt. Wenn wir nun x_a auf diesen Wert festsetzen und a aus T entfernen und in die Menge M aufnehmen, können wir dieses Verfahren mit dem übrigen Baum T und der aktualisierten Menge M fortsetzen. Schließlich ergibt sich ein (nicht notwendig zulässiger) Fluss für jeden Bogen des ursprünglichen Baumes T. □

Die Grundidee des Netzwerk-Simplex-Algorithmus ist, eine bekannte zulässige Baum-Lösung in eine andere, kostengünstigere Baum-Lösung „umzubauen“. Konkret passiert dies, indem wir einen Bogen $e \in A \setminus T$ (engl. „entering arc“) zu T hinzunehmen und einen Bogen $\ell \in T$ (engl. „leaving arc“) aus T entfernen. Damit die Baum-Eigenschaft erhalten bleibt, sollte ℓ natürlich auf dem eindeutigen Kreis C in $T + e$ liegen. Wir wählen die Orientierung von C so, dass e ein Vorwärtsbogen auf C ist; F und B bezeichnen die Mengen die Vorwärts- bzw. Rückwärtsbögen in C. Damit die Aufnahme von e den Zielfunktionswert verbessert, müssen wir den Fluss auf e erhöhen ($e \in L$) oder verringern ($e \in U$). Wenn wir den Fluss auf e um ε verändern, erhalten wir einen neuen Fluss x', indem wir alle Flüsse auf Bögen von C gemäß Gleichung (7.4) anpassen, sodass die Flusserhaltung weiter gewährleistet ist. Damit x' auch die Kapazitätsschranken einhält, dürfen wir ε nur so groß wählen, dass x' auf C eine Kapazitätsschranke erreicht, d. h. ε ist

$$\varepsilon := \min\big\{\min\{x_a - l_a \mid a \in B\}, \min\{u_a - x_a \mid a \in F\}\big\}. \tag{7.18}$$

Es ist durchaus möglich, dass $\varepsilon = 0$ auftritt. Sei ℓ einer der Bögen in T, der durch die Änderung des Flusses um ε seine Kapazitätsschranke erreicht (Achtung: hier kann auch $e = \ell$ gelten). Die Baum-Lösung für den neuen Fluss x' ergibt sich als

$$\begin{aligned} T &:= (T - \ell) + e, \\ L &:= \begin{cases} (L - e) + \ell & \text{falls } x'_\ell = l_a, \\ (L - e) & \text{sonst,} \end{cases}, \\ U &:= \begin{cases} (U - e) + \ell & \text{falls } x'_\ell = u_a, \\ (U - e) & \text{sonst.} \end{cases} \end{aligned} \tag{7.19}$$

Natürlich sollte e so gewählt werden, dass sich die Kostern verringern. Die Kosten von x und x' unterscheiden sich um

$$\varepsilon\Big(\sum_{a\in F} c_a - \sum_{a\in B} c_a,\Big) \tag{7.20}$$

d. h. die Kosten des Kreises C (die Differenz der Summen in den Klammern) sollten negativ sein. Die Berechnung der Kosten von C kann durch die Einführung von *Knotenpreisen* $y = (y_i)_{i\in V}$ vereinfacht werden. Die Idee (und ökonomische Interpretation) der Knotenpreise ist, dass y_i den Wert des transportierten Gutes (das hier abstrakt als Fluss behandelt wird) an Knoten $i \in V$ angibt. Diese Knotenpreise können lokal unterschiedlich sein, da die Transportkosten zu einem Knoten (die indirekt auch durch die Bogenkapazitäten bestimmt werden) sich von Knoten zu Knoten unterscheiden können. Sie sind allerdings nicht völlig unabhängig voneinander: Wenn das Gut zum Preis y_i an Knoten i bereitgestellt werden kann und für den Transport zu Knoten j die Kosten c_{ij} anfallen, so sollte der Preis an Knoten j *genau* $y_i + c_{ij}$ betragen. Ist der Preis geringer, ist er nicht kostendeckend, ist er höher, würde jemand anderes an Knoten i kaufen und den Transport selbst durchführen. Die Knotenpreise sollten daher das Gleichungssystem

$$y_i + c_{ij} = y_j \quad \forall (i,j) \in T \tag{7.21}$$

erfüllen. Dieses Gleichungssystem hat n Variablen, jedoch nur $(n-1)$ Gleichungen, die Lösung ist also nicht eindeutig. Allerdings ist die *Differenz* $y_i - y_j$ für $(i,j) \in T$ eindeutig und mit der Setzung $y_n = 0$ ergibt sich eine eindeutige Lösung für jeden aufspannenden Baum T. Wir definieren die *reduzierten Kosten* $\bar{c}_{ij}$ eines Bogens $(i,j) \in A$ als

$$\bar{c}_{ij} := c_{ij} + y_i - y_j \quad \forall (i,j) \in A. \tag{7.22}$$

Offensichtlich gilt $\bar{c}_{ij} = 0$ für $(i,j) \in T$. Es zeigt sich, dass die reduzierten Kosten von e genau den Kosten des von e induzierten Kreises C entsprechen.

(7.23) Lemma. *Sei $e \in A \setminus T$, C der eindeutige Kreis in $T + e$ sowie F und B die Menge der Vorwärts- und Rückwärtsbögen in C mit $e \in F$. Dann gilt*

$$\bar{c}_e = \sum_{a\in F} c_a - \sum_{a\in B} c_a.$$

△

Beweis. Es gilt

$$\begin{aligned}\sum_{a\in F} c_a - \sum_{a\in B} c_a &= \sum_{(i,j)\in F} (\bar{c}_{ij} - y_i + y_j) - \sum_{(i,j)\in B} (\bar{c}_{ij} - y_i + y_j)\\ &= \bar{c}_e + \underbrace{\sum_{(i,j)\in F} (-y_i + y_j) - \sum_{(i,j)\in B} (-y_i + y_j)}_{=0}\end{aligned}$$

wegen $\bar{c}_a = 0$ für $a \in T$ und weil die beiden Summen zusammen jeden Knoten des Kreises C jeweils einmal mit positivem und negativem Vorzeichen enthalten. □

Die reduzierten Kosten lassen sich, die Knotenpreise y vorausgesetzt, leichter berechnen als die Kosten eines Kreises. Intuitiv geben die reduzierten Kosten eines Bogens an, wie sich die Kosten des Gesamtflusses ändern, wenn man den Fluss auf dem Bogen um eine Einheit erhöht. Sie liefern daher das folgende Optimalitätskriterium.

(7.24) Satz. *Eine zulässige Baum-Lösung (x, T, L, U) ist optimal, wenn ein Knotenpreisvektor y existiert mit*

$$\bar{c}(a) \begin{cases} = 0 & a \in T, \\ \geq 0 & a \in L, \\ \leq 0 & a \in U. \end{cases}$$

△

Beweis. Sei x ein Fluss. Dann gilt

$$\begin{aligned} \bar{c}(x) &= \sum_{(i,j)\in A} x_{ij}(c_{ij} + y_i - y_j) \\ &= \sum_{(i,j)\in A} x_{ij}c_{ij} + \sum_{i\in V} \sum_{(i,j)\in A} x_{ij}y_i - \sum_{j\in V} \sum_{(i,j)\in A} x_{ij}y_j \\ &= c(x) + \sum_{i\in V} y_i \Bigg(\sum_{(i,j)\in A} x_{ij} - \sum_{(j,i)\in A} x_{ij} \Bigg) \\ &= c(x) + \sum_{i\in V} y_i d_i, \end{aligned}$$

d. h. die reduzierten Kosten $\bar{c}(x)$ eines Flusses x unterscheiden sich nur um eine Konstante von den Kosten $c(x)$. Ist x optimal bzgl. der reduzierten Kosten, ist es daher auch optimal bzgl. der ursprünglichen Kosten.

Sei nun (x, T, L, U) eine zulässige Baum-Lösung mit zugehörigem Knotenpreisvektor y und darüber definierten reduzierten Kosten $\bar{c}$, die die im Satz genannten Voraussetzungen erfüllen. Betrachten wir einen weiteren zulässigen Fluss x'. Es gilt dann:

$$\begin{aligned} \bar{c}(x') &= \sum_{a\in T} x'_a \bar{c}_a + \sum_{a\in L} x'_a \bar{c}_a + \sum_{a\in U} x'_a \bar{c}_a \\ &= \sum_{a\in L} x'_a \bar{c}_a + \sum_{a\in U} x'_a \bar{c}_a \\ &\geq \sum_{a\in L} l_a \bar{c}_a + \sum_{a\in U} u_a \bar{c}_a \\ &\geq \sum_{a\in A} x_a \bar{c}_a = \bar{c}(x). \end{aligned}$$

Also ist x optimal bezüglich $\bar{c}$ und damit auch bezüglich c. □

Insbesondere zeigt der zweite Teil des Beweises von Satz (7.24), dass stets eine optimale Baum-Lösung existiert, falls es überhaupt einen zulässigen Fluss gibt. Der Netzwerk-

Simplex-Algorithmus nutzt dies aus, indem ausschließlich zulässige Baum-Lösungen erzeugt werden. Noch offen ist die Konstruktion einer ersten zulässigen Baum-Lösung. Anders als beim Maximalflussproblem ist dies für das Minimalkosten-Flussproblems nicht trivial, da der Nullfluss nicht notwendig zulässig ist. Wir können jedoch aus jeder Instanz des Minimalkosten-Flussproblems $\mathcal{I} = (D = (V, A), c, l, u)$ eine abgeleitete Instanz $\mathcal{I}' = (D' = (V', A'), c', l', u')$ konstruieren,

- für die wir einen zulässigen Fluss angeben können und
- deren optimale Lösungen im Falle der Zulässigkeit von $\mathcal{I}$ denen von $\mathcal{I}$ entsprechen.

Dazu definieren wir den „Nettobedarf“ $\tilde{b}(i)$ für Knoten in V als

$$\tilde{b}_i := b_i + l(\delta^+(i)) - l(\delta^-(i)), \tag{7.25}$$

also als Differenz des Mindestbedarfs und der Mindestlieferung. Ein Knoten $i \in V$ ist „unterversorgt“, wenn sein Bedarf b_i nicht durch die eingehenden Bögen gedeckt werden kann, d. h. wenn $\tilde{b}_i < 0$ und „überversorgt“ sonst. Wir setzen

$$\begin{aligned} V^+ &:= \{i \in V \mid \tilde{b}_i \geq 0\}, \\ V^- &:= \{i \in V \mid \tilde{b}_i \geq 0\}. \end{aligned}$$

Der Digraph $D' = (V', A')$ enthält einen zusätzlichen Knoten k mit Bedarf $b'_k = 0$ sowie einen Bogen (k, i) für jeden Knoten $i \in V^-$ und einen Bogen (i, k) für jeden Knoten $i \in V^+$. Die Kosten für diese zusätzlichen Bögen werden so hoch gesetzt, dass sie in keiner Optimallösung für $\mathcal{I}'$ gewählt werden, falls $\mathcal{I}$ zulässig ist, also z. B.

$$M := 1 + \frac{1}{2}|V| \max\{|c_a| \mid a \in A\}.$$

Die erweiterte Instanz $\mathcal{I}'$ ist dann gegeben durch:

$$\begin{aligned} V' &:= V \cup \{k\} \\ A' &:= A \cup \{(k,i) \mid i \in V^-\} \cup \{(i,k) \mid i \in V^+\} \\ c'_a &:= \begin{cases} c_a & a \in A \\ M & a \in A' \setminus A \end{cases} \\ b'_i &:= \begin{cases} b_i & i \in V \\ 0 & i \in V' \setminus V \end{cases} \\ l'_a &:= \begin{cases} l_a & a \in A \\ 0 & a \in A' \setminus A \end{cases} \\ u'_a &:= \begin{cases} c_a & a \in A \\ \tilde{b}_i + 1 & a \in A' \setminus A \end{cases} \end{aligned} \tag{7.26}$$

(7.27) Satz. *Sei $\mathcal{I}$ eine Instanz eines Minimalkosten-Flussproblems und $\mathcal{I}'$ eine weitere, gemäß Gleichung* (7.26) *definierte Instanz. Dann gilt:*

(a) *Es existiert eine zulässige Lösung für $\mathcal{I}'$. Genauer bilden*

$$T := A' \setminus A, \quad L := A, \quad U := \emptyset$$

und der durch T definierte Fluss x eine zulässige Baum-Lösung von $\mathcal{I}'$.

(b) *Gilt in einer Optimallösung x von $\mathcal{I}'$ $x_a > 0$ für einen Bogen $a \in A' \setminus A$, so existiert kein zulässiger Fluss für $\mathcal{I}$.*

(c) *Ist x ein optimaler zulässiger Fluss für $\mathcal{I}'$ mit $x_a = 0$ für $a \in A' \setminus A$, so ist x eingeschränkt auf A ein optimaler zulässiger Fluss für $\mathcal{I}$.* △

Beweis. Hausaufgabe. □

(7.28) Algorithmus Netzwerk-Simplex-Algorithmus.

Eingabe: Zusammenhängender Digraph $D = (V, A)$, mit Kapazitäten $l, u \colon A \to \mathbb{R}_+^A$ und Kosten $w \in \mathbb{R}^A$ sowie Bedarfe an den Knoten $b \colon V \to \mathbb{R}$ mit $\sum_{i \in V} b_i = 0$.

Ausgabe: Ein zulässiger Fluss x, der kostenminimal unter allen zulässigen Flüssen ist, oder die Aussage, dass kein zulässiger Fluss existiert.

1. *Initialisierung:* Erstelle $\mathcal{I}'$ aus $\mathcal{I}$. Alle weiteren Schritte beziehen sich auf $\mathcal{I}'$. Setze

$$T := A' \setminus A, \quad L := A, \quad U := \emptyset$$

   und bestimme den entsprechenden Fluss x.

2. *Berechnung Knotenpreise:* Bestimme neue Knotenpreise y durch Lösen des Gleichungssystems (7.21) und $y_k = 0$.

3. *Optimalitätstest:* Wenn es kein $a \in L \cup U$ gibt mit

$$a \in L \text{ und } \bar{c}_a < 0 \quad \text{oder} \quad a \in U \text{ und } \bar{c}_a > 0, \tag{7.29}$$

   dann STOP: Wenn es ein $a \in A' \setminus A$ gibt mit $x_a > 0$, dann existiert kein zulässiger Fluss für $\mathcal{I}$. Sonst gib die Einschränkung von x auf A als optimalen Fluss aus.

4. *Pricing:* Wähle einen Bogen $e \in L \cup U$, der (7.29) erfüllt.

5. *Augmentieren:* Finde Kreis C in $T + e$ und bestimme die Flussänderung ε gemäß (7.18). Aktualisiere den Fluss x gemäß (7.4) und bestimme einen Bogen ℓ, der seine untere oder obere Kapazitätsschranke erreicht hat.

6. *Update:* Aktualisiere die Baum-Lösung gemäß (7.19) und gehe zu Schritt 2. △

Algorithmus (7.28) muss nicht terminieren, da es möglich ist, dass nur Bögen e mit reduzierten Kosten 0 gewählt werden können. In diesem Fall verändert sich der Kostenwert

nicht und es kann zu *Cycling* kommen, d. h. der Algorithmus gerät in eine Endlosschleife. Um dies zu vermeiden, muss man den Bogen, der den Baum T verlässt, sorgfältig auswählen.

Ein von A. Löbel (Konrad-Zuse-Zentrum, Berlin) implementierter Code dieser Art, es handelt sich um einen sogenannten primal-dualen Netzwerk-Simplex-Algorithmus, ist auf dem ZIB-Server für akademische Nutzung verfügbar, siehe URL:

`http://www.zib.de/Optimization/Software/Mcf/`

Mit diesem Code namens MCF können Minimalkosten-Flussprobleme mit Zigtausenden Knoten und Hundertmillionen Bögen in wenigen Minuten gelöst werden. MCF findet derzeit u. a. in verschiedenen Planungssystemen für den öffentlichen Nahverkehr Anwendung. MCF ist als einer der Integer-Benchmark Codes in die SPEC CPU2006-Suite aufgenommen worden, mit der Leistungsevaluierungen moderner Computersysteme vorgenommen werden, siehe URL:

`http://www.spec.org`.

Literaturverzeichnis

R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows, Handbooks in Operations Research and Management Science*, volume 1, chapter Optimization, pages 211–360. Elsevier, North-Holland, Amsterdam, 1989.

R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows, Theory, Algorithms and Applications.* Paramount Publishing International, Prentice Hall, New York, 1993.

D. Bertsimas and J. N. Tsitsiklis. *Introduction to Linear Optimization.* Athena Scientific, 1997.

V. Chvátal. *Linear Programming.* Freeman, 1983.

M. Grötschel. Tiefensuche: Bemerkungen zur Algorithmengeschichte. In H. Hecht, R. Mikosch, I. Schwarz, H. Siebert, and R. Werther, editors, *Kosmos und Zahl - Beiträge zur Mathematik- und Astronomiegeschichte, zu Alexander von Humboldt und Leibniz*, volume 58 of *Boethius*, pages 331–346. Franz Steiner Verlag, 2008. URL `http://www.steiner-verlag.de/titel/56734.html`.

T. C. Hu. *Integer Programming and Network Flows.* Addison-Wesley, Reading, Massachusetts, 1969.

P. A. Jensen and J. W. Barnes. *Network Flow Programming.* Wiley & Sons, New York, 1980.

D. Jungnickel. *Graphs, Networks and Algorithms.* Springer, 4. edition, 2012.

J. Kennington and R. Helgason. *Algorithms for Network Programming.* Wiley & Sons, New York, 1980.

H. W. Kuhn. The hungarian method for the assignment problem. *Naval Logistics Quarterly*, 2:83–97, 1955.

J. L. R. Ford and D. R. Fulkerson. *Flows in Networks.* Princeton University Press, Princeton, 1962.

E. L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart & Winston, New York, 1976.

S. I. M. Shigeno and S. T. McCormick. Relaxed most negative cycle and most positive cut canceling algorithms for minimum cost flow. *Mathematics of Operations Research*, 25:76–104, 2000.

E. T. V. Goldberg and R. E. Tarjan. Network flow algorithms. In in B. Korte et al. (eds.), editor, *Paths, Flows, and VLSI-Layout.* Springer-Verlag, Berlin, 1990.