Graphen- und Netzwerkalgorithmen

(Algorithmische Diskrete Mathematik I)

Skriptum zur Vorlesung im SS 2003

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

Version vom 21. August 2003

Vorwort

Bei dem vorliegenden Skript handelt es sich um die Ausarbeitung der vierstündigen Vorlesung "Graphen- und Netzwerkalgorithmen", die die grundlegende Vorlesung des dreisemestrigen Zyklus "Algorithmische Diskrete Mathematik" bildet. Diese Vorlesung wurde von mir im SS 2003 an der TU Berlin gehalten.

Ziel der Vorlesung ist eine Einführung in die Theorie der Graphen und Netzwerke, wobei auf algorithmische Aspekte besonderer Wert gelegt wird. Vorkenntnisse sind nicht erforderlich; alle 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. Verfahren, die lineare oder ganzzahlige Optimierung benutzen, werden in dieser Vorlesung nicht vorgestellt.

Es gibt kein einzelnes Buch, das den gesamten, in dieser Vorlesung abgehandelten Themenkreis abdeckt. Daher sind in die einzelnen Kapitel Literaturhinweise eingearbeitet worden. Lesenswerte Einführungen in die kombinatorische Optimierung, in algorithmische Aspekte der Graphen- und Netzwerktheorie (bzw. gewisse Teilbereiche dieser Gebiete) sind die in den Literaturverzeichnissen von Kapitel 1 und 2 aufgeführten Bücher Ahuja, Magnanti, Orlin (1993) und Cook, Cunningham, Pulleyblank, Schrijver (1998), Diestel (1996), Jungnickel (1994), Korte, Vygen (2002), Schrijver (2003), West (1996) und der Übersichtsartikel Grötschel und Lovász (1995).

Dies ist ein Vorlesungsskript und kein Buch. Obwohl ich mit der gebotenen Sorgfalt geschrieben habe, war nicht genügend Zeit für intensives 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.

Sommer 2003 M. Grötschel

Inhaltsverzeichnis

Kapitel 1

Graphen, Hypergraphen, Matroide: wichtige Definitionen und Bezeichnungen

Bei der nachfolgenden Zusammenstellung von Begriffen und Bezeichnungen aus der Graphen-, Hypergraphen- und Matroidtheorie 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.

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 Fettdruck 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 über 260 Verweise. Bei rund 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 (2000), Bondy and Murty (1976) und West (2001). Stärker algorithmisch orientiert und anwendungsbezogen sind z. B. Ebert (1981), Golombic (1980) sowie Jungnickel (1994) und Walther, H. and Nägler, G. (1987).

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). 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 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**. 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: $\mathbf{deg}(v)$) 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 $\boldsymbol{E(W)}$die Menge aller Kanten von G mit beiden Endknoten in W und mit $\boldsymbol{V(F)}$ 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 $\boldsymbol{W}$ 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 $\boldsymbol{K_n}$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 $\boldsymbol{K_{m,n}}$.

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 $\alpha(\mathbf{G})$ (bzw. $\omega(\mathbf{G})$); die kleinste Kardinalität einer Knotenüberdeckung mit $\tau(\mathbf{G})$.

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 $\boldsymbol{k}$**-Faktor**, 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 $\boldsymbol{\nu(G)}$, die kleinste Kardinalität einer Kantenüberdeckung mit $\boldsymbol{\rho(G)}$.

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 $\chi(\mathbf{G})$ (bzw. $\overline{\chi}(\mathbf{G})$), die minimale Anzahl von Matchings in einer Kantenfärbung mit $\boldsymbol{\gamma(G)}$. 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.

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) nicht-leeren **Knotenmenge** V und einer (endlichen) Menge A von **Bögen** (oder gerichteten Kanten; englisch: arc). Eine **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.

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). 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 $(v_0, \ldots, v_k)$ ein 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 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**. Eine Kette, in der alle Kanten oder Bögen verschieden sind, heißt **Pfad**. 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 $[\mathbf{u}, \mathbf{v}]$**-Wegen**, $(\mathbf{u}, \mathbf{v})$**-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 (2000) 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 der Anfangsknoten und alle inneren Knoten voneinander verschieden sind, heißt **Kreis** (**gerichteter Kreis**). Offensichtlich enthält jeder geschlossene Pfad einen Kreis.

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. 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**. 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, 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.

1.5 Hypergraphen

Sei V eine endliche Menge und $\mathcal{E} = (E_i \mid i \in I)$ eine endliche Familie von Teilmengen von V. $\mathcal{E}$ heißt endlicher **Hypergraph** auf V, falls $E_i \neq \emptyset$ für alle $i \in I$ und $\bigcup_{i \in I} E_i = V$ gilt. Manchmal nennt man auch das Paar $(V, \mathcal{E})$ **Hypergraph**. $|V|$ ist die **Ordnung** des Hypergraphen, die Elemente von V heißen **Knoten**, die Mengen E_i **Hyperkanten** (oder kurz Kanten).

Die Hyperkanten mit genau einem Element heißen **Schlingen**. Ein Hypergraph heißt **einfach**, wenn alle Kanten voneinander verschieden sind. In diesem Falle ist $\mathcal{E}$ eine Teilmenge von 2^V. (2^V bezeichnet die Potenzmenge von V.)

Jeder Graph ohne isolierte Knoten kann also als Hypergraph aufgefasst werden, dessen (Hyper-) Kanten höchstens 2 Elemente enthalten. Ein einfacher Graph ohne isolierte Knoten ist ein einfacher Hypergraph mit $|E_i| = 2$ für alle $i \in I$.

Es ist gelegentlich nützlich, graphentheoretische Objekte als Hypergraphen aufzufassen. Sei z. B. $D = (V, A)$ ein Digraph, und seien $s, t \in V$ zwei festgewählte Knoten. Sei $B \subseteq A$ die Menge der Bögen, die auf mindestens einem (s,t)-Weg liegen. Seien $\mathcal{E}_1 := \{P \subseteq B \mid P$ ist die Bogenmenge eines (s,t)-Weges $\}$, $\mathcal{E}_2 := \{C \subseteq B \mid C$ ist ein (s,t)-Schnitt in $(V, B)\}$, dann sind $(B, \mathcal{E}_1)$ und $(B, \mathcal{E}_2)$ zwei interessante Hypergraphen, die uns in der Theorie der Blocker wiederbegegnen werden. In dieser Theorie sind besonders **Antiketten** (oder Clutter) von Interesse. Das sind Hypergraphen $(V, \mathcal{E})$, so dass für je zwei Kanten $E, F \in \mathcal{E}$ weder $E \subseteq F$ noch $F \subseteq E$ gilt. Der oben definierte Hypergraph $(B, \mathcal{E}_1)$ ist für jeden Digraphen D eine Antikette, dies gilt jedoch nicht für den Hypergraphen $(B, \mathcal{E}_2)$.

1.6 Matroide, Unabhängigkeitssysteme

Ist E eine endliche Menge und $\mathcal{I} \subseteq 2^E$ eine Menge von Teilmengen von E, dann heißt $\mathcal{I}$ oder das Paar $(E, \mathcal{I})$ **Unabhängigkeitssystem** auf E, falls gilt:

(I.1) $\emptyset \in \mathcal{I}$

(I.2) $I \subseteq J \in \mathcal{I} \Longrightarrow I \in \mathcal{I}$.

Die Elemente von $\mathcal{I}$ heißen **unabhängige Mengen**, die Mengen in $2^E \setminus \mathcal{I}$ **abhängige Mengen**. $(E, \mathcal{I} \setminus \{\emptyset\})$ ist also ein einfacher Hypergraph. Beispiele graphentheoretisch interessanter Unabhängigkeitssysteme sind etwa die Menge aller Matchings $\{M \subseteq E \mid M$ Matching $\}$ oder die Menge aller stabilen Knotenmengen $\{S \subseteq V \mid S$ stabil $\}$ eines Graphen $G = (V, E)$ oder die Menge aller Branchings $\{B \subseteq A \mid (V, B)$ Branching $\}$ eines Digraphen $D = (V, A)$.

Ein Unabhängigkeitssystem $(E, \mathcal{I})$ heißt **Matroid**, falls gilt

(I.3) $\quad I, J \in \mathcal{I}, |I| = |J| + 1 \Longrightarrow \exists e \in I \setminus J$ mit $J \cup \{e\} \in \mathcal{I}$.

Ist $(E, \mathcal{I})$ ein Unabhängigkeitssystem und $F \subseteq E$, dann heißt jede Menge $B \subseteq F$ mit $B \in \mathcal{I}$, die in keiner weiteren Menge mit diesen beiden Eigenschaften enthalten ist, **Basis** von F. Die Zahl

$$r(F) := \max\{|B| \mid B \text{ Basis von } F\}$$

ist der **Rang** von F bezüglich $(E, \mathcal{I})$. Für Matroide $(E, \mathcal{I})$ folgt aus (I.3), dass für jede Menge $F \subseteq E$ alle Basen von F die gleiche Kardinalität besitzen.

Eine abhängige Menge $C \subseteq E$, die die Eigenschaft hat, dass $C \setminus \{e\}$ unabhängig ist für alle $e \in C$, heißt **Zirkuit** (oder Kreis) des Unabhängigkeitssytems $(E, \mathcal{I})$.

Ein klassisches Beispiel für Matroide ist das folgende. Sei $G = (V, E)$ ein Graph. Dann ist $\mathcal{I} := \{F \subseteq E \mid (V, F) \text{ Wald}\}$ das Unabhängigkeitssystem eines Matroids auf E. Dieses Matroid wird das **graphische Matroid** bezüglich G genannt. Die Zirkuits von $(E, \mathcal{I})$ sind die Kreise von G, die Basen einer Kantenmenge F sind die Vereingungen von aufspannenden Bäumen der Komponenten von $(V(F), F)$.

Ist E eine Menge und $\mathcal{I} \subseteq 2^E$ ein Mengensystem, so nennen wir eine Menge $M \in \mathcal{I}$ **maximal** (bzw. **minimal**) bezüglich $\mathcal{I}$, wenn es keine Menge $M' \in \mathcal{I}$ gibt mit $M \subset M'$ (bzw. $M' \subset M$). $M \in \mathcal{I}$ heißt ein **größtes** (bzw. **kleinstes**) Element von $\mathcal{I}$, wenn $|M' | \leq | M|$ (bzw. $|M' | \geq |M|$) gilt für alle $M' \in \mathcal{I}$. So sind z. B. bezüglich eines Unabhängigkeitssystems $(E, \mathcal{I})$ und einer Menge $F \subseteq E$ die Basen von F genau die maximalen Elemente von $\mathcal{I}_F := \{I \in \mathcal{I} \mid I \subseteq F\}$. Nicht jede Basis ist auch ein größtes Element von $\mathcal{I}_F$. (Z. B. sind nicht alle maximalen Matchings (maximalen stabilen Knotenmengen) in einem Graphen größte Matchings (größte stabile Mengen).) Ist $(E, \mathcal{I})$ jedoch ein Matroid, so sind alle maximalen Elemente von $\mathcal{I}_F$ auch größte Elemente von $\mathcal{I}_F$. (In einem zusammenhängenden Graphen sind z. B. alle maximalen Wälder aufspannende Bäume.)

Gute Einführungen in die Matroidtheorie sind die Bücher Oxley (1992) und Welsh (1980).

1.7 Liste einiger in der Graphentheorie verwendeter Symbole

G	bezeichnet meistens einen Graphen
D	bezeichnet meistens einen Digraphen
n	Anzahl der Knoten eines Graphen (wenn möglich)
m	Anzahl der Kanten eines Graphen (wenn möglich)
$\alpha(G)$	Stabilitätszahl = maximale Kardinalität einer stabilen Menge in G
$\omega(G)$	Cliquenzahl = maximale Kardinalität einer Clique in G
$\tau(G)$	Knotenüberdeckungszahl = minimale Kardinalität einer Knotenüberdeckung von G
$\rho(G)$	Kantenüberdeckungszahl = minimale Kardinalität einer Kantenüberdeckung von G
$\nu(G)$	Matchingzahl = maximale Kardinalität eines Matchings in G
$\chi(G)$	Färbungszahl = chromatische Zahl = minimale Anzahl von stabilen Mengen in einer Knotenfärbung von G
$\overline{\chi}(G)$	Cliquenüberdeckungszahl = minimale Anzahl von Cliquen in einer Cliquenüberdeckung von G
$\gamma(G)$	Kantenfärbungszahl = chromatischer Index = minimale Anzahl von Matchings in einer Kantenfärbung von G
$\Delta(G)$	maximaler Grad eines Knotens von G
$\delta(G)$	minimaler Grad eines Knotens von G
$\delta(W)$	der durch die Knotenmenge W induzierte Schnitt
$\delta^+(W)$	Menge aller Bögen mit Anfangsknoten in W und Endknoten in $V \setminus W$
$\delta^-(W)$	Menge aller Bögen mit Endknoten in W und Anfangsknoten in $V \setminus W$
$\kappa(G)$	Zusammenhangszahl = maximales κ, so dass G κ-zusammenhängend ist
$\vec{\kappa}(D)$	starke Zusammenhangszahl = maximales κ, so dass G stark κ-zusammenhängend ist
$\lambda(G)$	Kantenzusammenhangszahl = maximales κ, so dass G κ-kantenzusammenhängend ist
$\vec{\lambda}(G)$	starke Bogenzusammenhangszahl = maximales κ, so dass D stark κ-bogenzusammenhängend ist
$\Gamma(W)$	Menge der Nachbarn der Knotenmenge W
$G[W]$	der von der Knotenmenge W induzierte Untergraph von G
$G - W$	der aus G durch Entfernung der Knotenmenge W entstehende Graph
$G - F$	der aus G durch Entfernung der Kantenmenge F entstehende Graph
$G \cdot W$	der aus G durch Kontraktion der Knotenmenge W entstehende Graph
$G \cdot F$	der aus G durch Kontraktion der Kantenmenge F entstehende Graph
$\deg(v)$	Grad des Knoten v
$E(W)$	Menge aller Kanten von G mit beiden Endknoten in W
$A(W)$	Menge alle Bögen von D mit Anfangs- und Endknoten in W
$V(F)$	Menge aller Knoten aus G (bzw. D) die Endknoten (bzw. Anfangs- oder Endknoten) mit mindestens einer Kante (bzw. eines Bogens) aus F sind
K_n	vollständiger Graph mit n Knoten
$K_{m,n}$	vollständig bipartiter Graph mit $\lvert V_1\rvert = m$ und $\lvert V_2\rvert = n$
$L(G)$	Kantengraph von G
$\overline{G}$	Komplement von G

Literaturverzeichnis

Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1993). *Network Flows, Theory, Algorithms and Applications*. Pearson Education, Prentice Hall, New York, first edition.

Aigner, M. (1984). *Graphentheorie: eine Entwicklung aus dem 4-Farben-Problem* . Teubner Verlag, Studienbücher: Mathematik, Stuttgart.

Berge, C. and Ghouila-Houri, A. (1969). *Programme, Spiele, Transportnetze*. Teubner Verlag, Leipzig.

Bollobás, B. (1998). *Modern Graph Theory*. Springer Verlag, New York.

Bondy, J. A. and Murty, U. S. R. (1976). *Graph Theory with Applications*. American Elsevier, New York and Macmillan, London.

Cook, W. J., Cunningham, W. H., Pulleyblank, W. R., and Schrijver, A. (1998). *Combinatorial Optimization*. John Wiley & Sons, Inc., New York.

Diestel, R. (1996). *Graphentheorie*. Springer-Verlag, Heidelberg, 1. edition.

Diestel, R. (2000). *Graphentheorie*. Springer-Verlag, Heidelberg, 2. edition.

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

Ebert, J. (1981). *Effiziente Graphenalgorithmen*. Akademische Verlagsgesellschaft, Wiesbaden.

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

Grötschel, M. and Lovász, L. (1995). Combinatorial Optimization. In Graham, R. L., Grötschel, M., and Lovász, L., editors, *Handbook of Combinatorics, Volume II*, pages 1541–1597. Elsevier (North-Holland).

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

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

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

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

Korte, B. and Vygen, J. (2002). *Combinatorial Optimization: Theory and Algorithms*, volume 21 of *Algorithms and Combinatorics*. Springer, Berlin, second edition.

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

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

Schrijver, A. (2003). *Combinatorial Optimization – Polyhedra and Efficiency*. Springer-Verlag, Berlin.

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

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

Welsh, D. J. A. (1980). *Matroid Theory*. Academic Press, New York.

West, D. B. (1996). *Introduction to Graph Theory*. Prentice Hall, Upper Saddle River, first edition.

West, D. B. (2001). *Introduction to Graph Theory*. Prentice Hall, Upper Saddle River, second edition.

Kapitel 2

Optimierungsprobleme auf Graphen: eine Einführung

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

2.1 Kombinatorische Optimierungsprobleme

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

(2.1) 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 (2.1) 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.

(2.2) Kombinatorisches 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 $c : E \rightarrow \mathbb{R}$. 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 (2.2) 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 (2.2) bereits sehr speziell strukturiert. Aber Problem (2.2) 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 (2.2) 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. 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.

2.2 Klassische Fragestellungen der Graphentheorie

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

(2.3) Euler und das Königsberger Brückenproblem. Fast jedes Buch über Graphentheorie enthält einen Stadtplan von Königsberg und erläutert, wie Euler die Königsberger Karte zu dem Graphen aus Abbildung 2.1 “abstrahiert” hat.

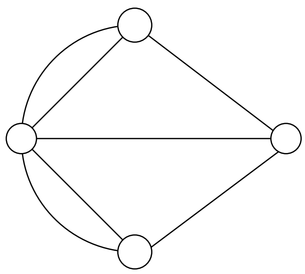


Abb. 2.1

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 2.1 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. □

(2.4) Das Haus vom Nikolaus. Jeder kennt die Aufgabe aus dem Kindergarten: Zeichne das Haus des Nikolaus, siehe Abbildung 2.2, in einem Zug!

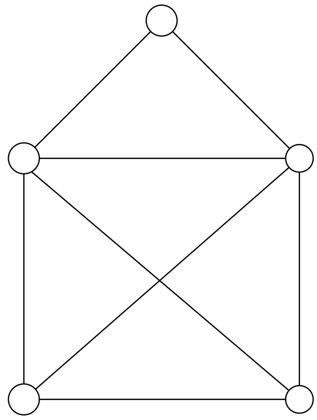


Abb. 2.2

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

(2.5) 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 2.3, 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

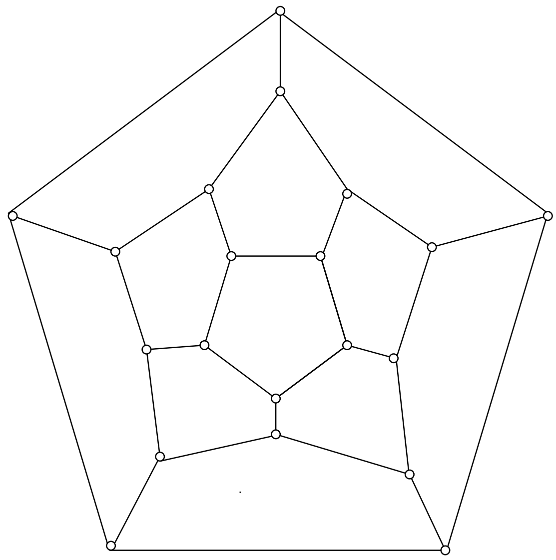


Abb. 2.3

Spiels mit dem Namen "The Icosian Game" befindet sich noch in der Bibliothek des Trinity College in Dublin, Irland, siehe Abbildung 2.4. Die Aufgabe, in

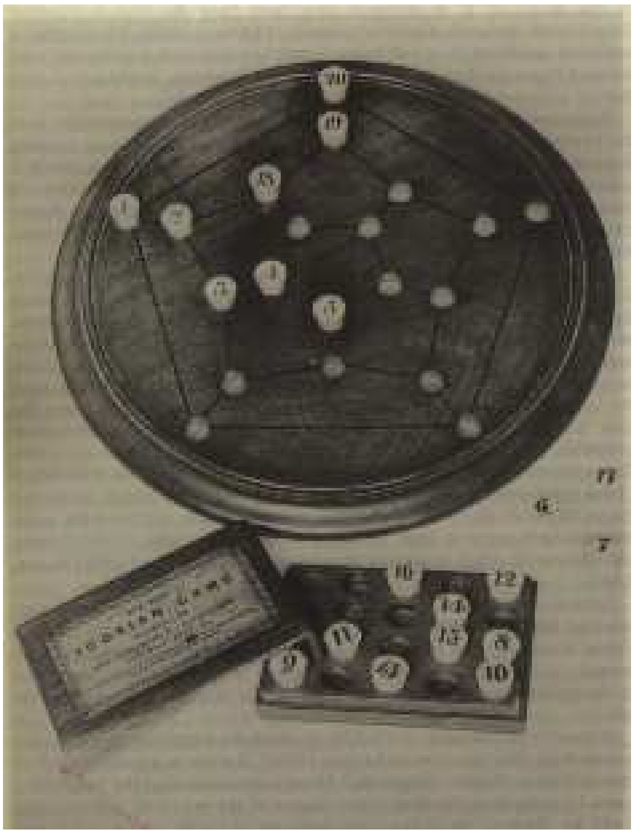


Abb. 2.4

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 u.a. in Hoffman and Wolfe (1985).

□

(2.6) Färbung von Landkarten. Nach Aigner (1984), der die Entwicklung der Graphentheorie anhand der vielfältigen Versuche, das 4-Farben-Problem zu lösen, darstellt, 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 2.5 (a) zeigt die Karte der deutschen Bundesländer. Der “Bundesländergraph” in Abbildung 2.5 (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.

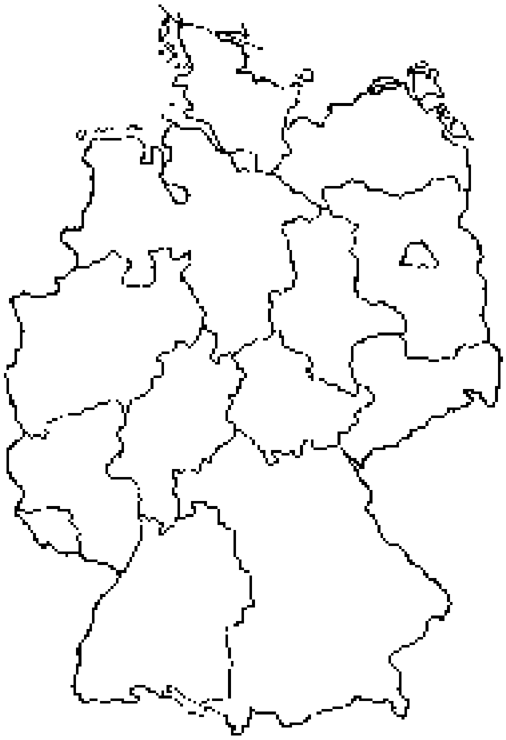


Abb. 2.5 (a)

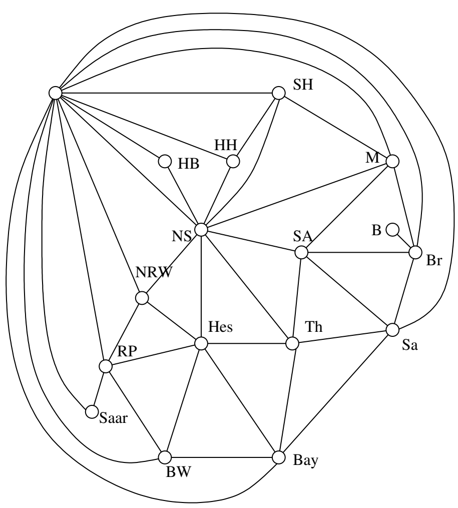


Abb. 2.5 (b)

“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`.

(2.7) 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. $\square$

2.3 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.

(2.8) 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 $u, v \in V$ 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 Bordsysteme 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".

(2.9) 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.) 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 3), 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 $u_i v_j \in E$ gilt. Das Gewicht $c'((w_i, w_j))$ des Bogens (w_i, w_j) sei das Gewicht $c(u_i v_j)$ der Kante $u_i v_j$. Ist B eine minimale Lösung des gerichteten Zuordnungsproblems bezüglich D und c', so ist

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

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 $E := \{u_i v_j \mid (w_i, w_j) \in A\}$. Es seien

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

und

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

Nach Konstruktion gilt, dass jedes Matching in G mit k Kanten ein geringeres Gewicht hat als ein Matching mit $k+1$ Kanten, $k = 0, \dots, n-1$. 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

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

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

(2.10) 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. 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**. □

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 $V = \{v_1, \dots, v_n\}$ gilt. Die Graphen $G_1 = (U, E_1)$ mit $U := \{u_1, \dots, u_n\}$ und $G_2 = (W, E_2)$ mit $W := \{w_1, \dots, w_n\}$ seien knotendisjunkte isomorphe Bilder (also Kopien) von G, so dass die Abbildungen $v_i \mapsto u_i$ und $v_i \mapsto w_i$, $i = 1, \dots, n$ 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 $u_i w_i$. Diese neuen Kanten $u_i w_i$ erhalten das Gewicht $c(u_i w_i) = 0$. 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 $u_i w_i$, siehe Abbildung 2.6. Man überlegt sich leicht, dass jedes perfekte Matching in G' einem ungeraden $[u, v]$-Weg in G entspricht und dass jedes minimale perfekte Matching in G' einen minimalen ungeraden $[u, v]$-Weg bestimmt.

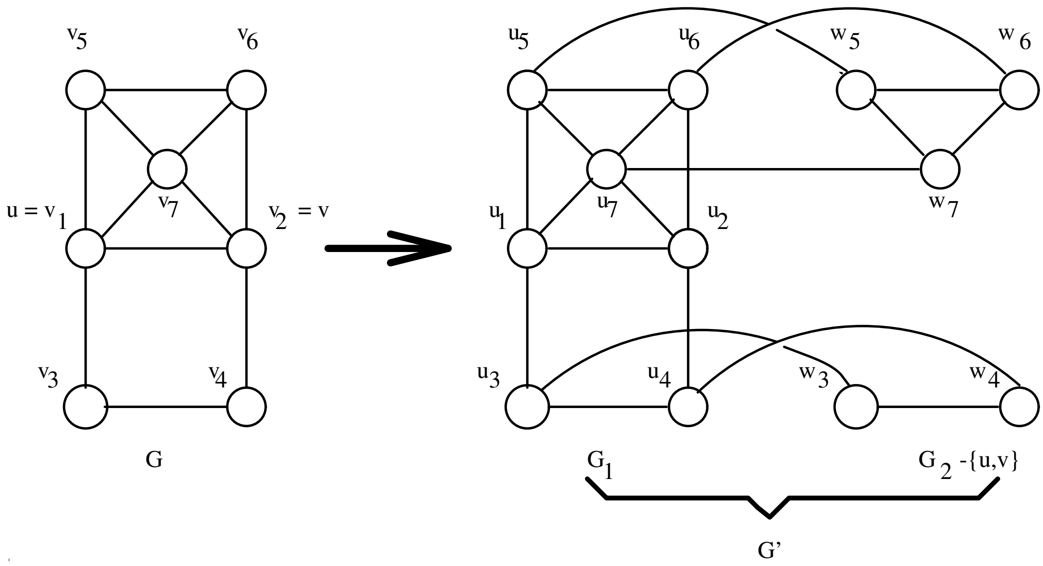


Abb. 2.6

In Abbildung 2.6 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.

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

(2.11) 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.

(2.12) 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 seiner Varianten) auf. Abbildung 2.7 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 2.7 zeigt eine optimale Lösung dieses 442-Städte-TSP. Die Bohrmaschine muss eine Weglänge von 50.069 Einheiten zurückzulegen.

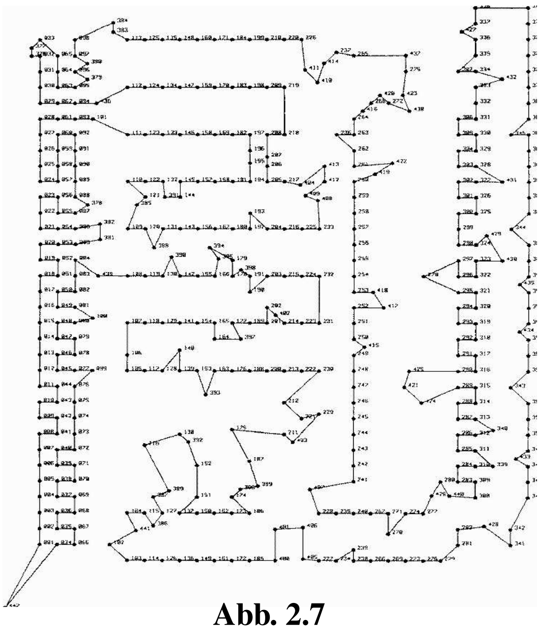


Abb. 2.7

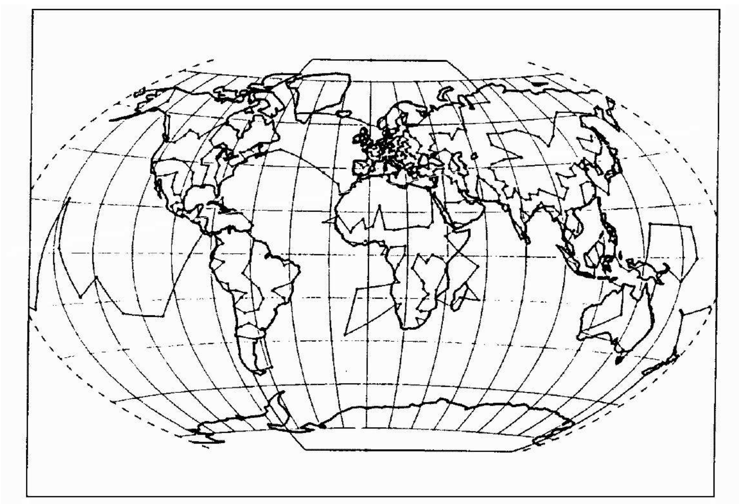


Abb. 2.8

Abbildung 2.8 zeigt 666 Städte auf der Weltkugel. Wählt man die "Luftliniendistanz" (bezüglich eines Großkreises auf der Kugel) als Entfernung zwischen zwei

Städten, so zeigt Abbildung 2. 8 eine kürzeste Rundreise durch die 666 Orte dieser Welt. Die Länge dieser Reise ist 294 358 km lang. Abbildungen 2.7 und 2.8 sind Grötschel and Holland (1991) entnommen. In 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/iwr/comopt/software/TSPLIB95/`

Eine weitere Webpage zum TSP mit lauffähigen Codes etc. ist:

`http://www.densis.fee.unicamp.br/~moscato/TSPBIB_home.html`

In Abbildung 2.9 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.

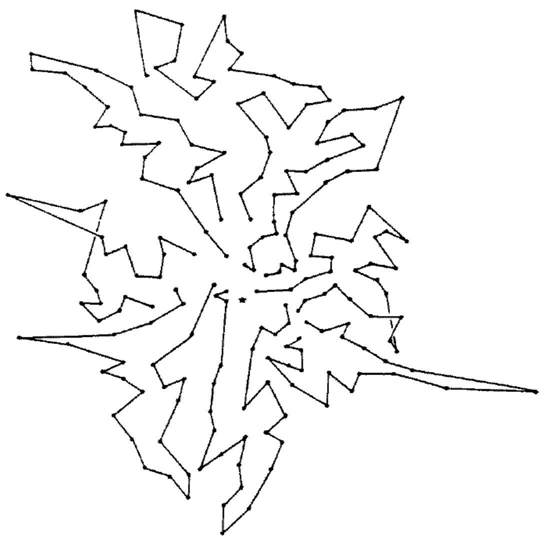


Abb. 2.9

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

- Bestimmung einer optimalen Durchlaufreihenfolge der Flüssigkeiten (Chargen) in einer Mehrproduktenpipeline,
- Bestimmung der optimalen Verarbeitungsfolge von Lacken in einer Großlackiererei,

bei diesen beiden Problemen sollen Reinigungszeiten minimiert werden,

- 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.

Umfangreiche Information über das TSP, seine Varianten und Anwendungen kann man in dem Sammelband Lawler et al. (1985) finden.

(2.13) 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.

(2.14) 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 $S_1, \dots, S_t$ von G zu suchen, so dass jeder Knoten in mindestens b_v dieser stabilen Mengen enthalten und t minimal ist, heiß t (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 $M_1, \dots, M_s$, so dass jede Kante in mindestens c_e dieser Matchings enthalten und s so klein wie möglich ist. $\square$

Das geographische Färbungsproblem ist uns schon in (2.6) 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.

(2.15) Schnitt-Probleme. 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 7). 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

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

beschreiben 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:

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

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 (sowie weitere Literaturhinweise) kann man in Barahona et al. (1988), Barahona, Grötschel, Jünger, Reinelt (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.

(2.16) 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 (2.12)) 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. □

(2.17) 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 (2.17) 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.

(2.18) 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 derzeit z. B. in den USA Glasfasernetzwerke für so genannte LATA-Netze entworfen und ausgelegt, siehe Grötschel et al. (1992) und Grötschel et al. (1995) (von einer großen Stadt in den USA) das Netzwerk der möglichen direkten Kabelverbindungen, Abbildung 2.10 (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. □

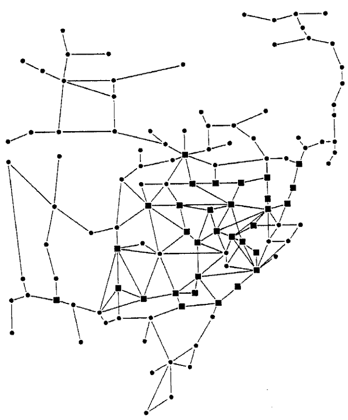


Abb. 2.10a

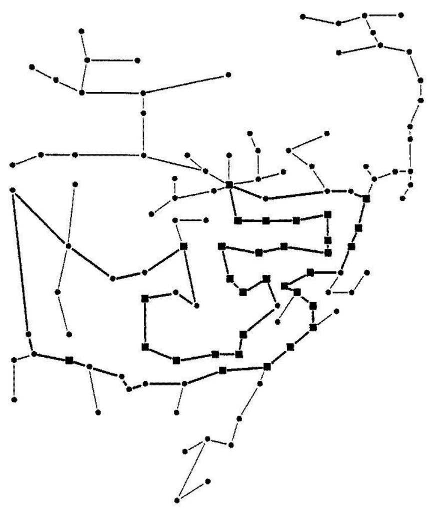


Abb. 2.10b

Literaturverzeichnis

Aigner, M. (1984). *Graphentheorie: eine Entwicklung aus dem 4-Farben-Problem*. Teubner Verlag, Studienbücher: Mathematik, Stuttgart.

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

Berge, C. (1989). *Hypergraphs, Combinatorics of Finite Sets*, volume 45. North-Holland Mathematical Library, Amsterdam.

Berge, C. and Ghouila-Houri, A. (1969). *Programme, Spiele, Transportnetze*. Teubner Verlag, Leipzig.

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

Bondy, J. A. and Murty, U. S. R. (1976). *Graph Theory with Applications*. American Elsevier, New York and Macmillan, London.

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

Ebert, J. (1981). *Effiziente Graphenalgorithmen*. Akademische Verlagsgesellschaft, Wiesbaden.

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

Graham, R. L., Grötschel, M., and Lovász, L., editors (1995a). *Handbook of Combinatorics, Volume I*. Elsevier (North-Holland); The MIT Press, Cambridge, Massachusetts.

Graham, R. L., Grötschel, M., and Lovász, L., editors (1995b). *Handbook of Combinatorics, Volume II*. Elsevier (North-Holland); The MIT Press, Cambridge, Massachusetts.

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

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

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

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

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

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

Hoffman, A. J. and Wolfe, P. (1985). History. In et all (Hrsg.), E. L. L., editor, *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, pages 1–16. John Wiley & Sons, Chichester.

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

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

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

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

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

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

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

Stoer, M. (1992). Design of Survivable Networks. In *Lecture Notes in Mathematics*, volume 1531. Springer Verlag, Berlin, Heidelberg.

Truemper, K. (1992). *Matroid Decomposition*. Academic Press, Boston.

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

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

Welsh, D. J. A. (1976). *Matroid Theory*. Academic Press, London.

Kapitel 3

Komplexitätstheorie, Speicherung von Daten

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 informellen Niveau. Gute Bücher zur Einführung in die Komplexitätstheorie sind Garey and Johnson (1979) und Papadimitriou (1994), weiterführende Aspekte werden z. B. in Wagner and Wechsung (1986) und van Leeuwen (1990) behandelt. Shmoys und Tardos (1995) ist ein guter Übersichtsartikel. Einen detaillierten Überblick über Komplexitätsklassen gibt Johnson (1990).

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 3.3. Zur Vertiefung dieses Gebietes empfehlen wir Aho et al. (1974), Cormen et al. (1990), Mehlhorn (1984), Meinel (1991), Ottmann and Widmayer (1990) und Tarjan (1983).

3.1 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.,

$$\langle n \rangle := \lceil \log_2(|n|+1) \rceil + 1. \tag{3.1}$$

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 genau die benötigten $\langle r \rangle$ Zellen besitzt.

Die Kodierungslänge eines Vektors $x = (x_1, \dots, x_n)^T \in \mathbb{Q}^n$ 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 3.3 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.

(3.2) 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*

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

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*

$$s_A(n) \leq q(n) \quad \text{für alle } n \in \mathbb{N}.$$

□

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 die Benutzung eines Speicherplatzes in die Berechnung der Laufzeitfunktion eingeht.

(3.3) Hausaufgabe. *Bestimmen Sie die Laufzeitfunktion und die Speicherplatzfunktion des folgenden Algorithmus:*

Input: *ganze Zahl n*

$k := \langle n \rangle$

DO $i = 1$ TO k:

$n := n \cdot n \cdot n$

END

Gib n aus.

□

3.2 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 3.3 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. Enthält er einen, und jemand hat ihn 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.

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

(a) *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.*

(b) *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.*

□

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 in (b) 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 (3.4) (a) und (b) 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 (a) und (b) aus (3.4) 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. 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.

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

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. Das 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 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.

(3.5) 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.*

□

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.

(3.6) 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 beweisen. 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. Seien s und t die kleinste bzw. größte Zahl, die als Entfernung vorkommt. Da jede Rundreise n Kanten enthält, ist keine Rundreise kürzer als ns und keine länger als nt. Wir fragen nun den Algorithmus zur Lösung des TSP-Entscheidungsproblems, ob es eine Rundreise gibt, deren Länge nicht größer als $\frac{n(t+s)}{2}$ ist. Ist das so, fragen wir, ob es eine Rundreise gibt, deren Länge höchstens $\frac{n(t+s)}{4}$ ist, andernfalls fragen wir, ob es eine Rundreise gibt mit Länge höchstens $\frac{3n(t+s)}{4}$. 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 $(\lceil \log_2(n(t-s)) \rceil + 1)$-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.

In der nachfolgenden Tabelle haben wir einige der Beispiele von kombinatorischen Optimierungsproblemen aufgelistet, die wir in früheren Abschnitten eingeführt haben und die $\mathcal{NP}$-schwer sind:

– 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 & Johnson (1979) aufgeführt. Probleme, die mit diesem Themenkreis zusammenhängen, wurden auch in einer seit 1981 über mehr als zehn Jahre laufenden Serie von Aufsätzen von D. S. Johnson mit dem Titel "The $\mathcal{NP}$-completeness column: an ongoing guide" im *Journal of Algorithms* behandelt. Der (vorläufig?) letzte Artikel (No. 23) ist 1992 erschienen. Im Internet finden Sie unter der URL

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

ein "Compendium of $\mathcal{NP}$ Optimization Problems".

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.

3.3 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.

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.

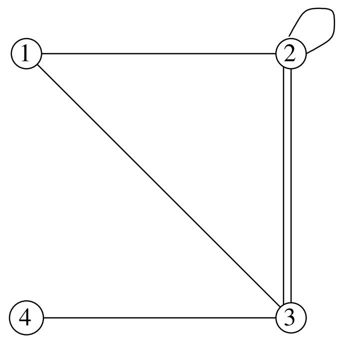


Abb. 3.1

Eine mögliche Kantenliste für den Graphen aus Abbildung 3.1 ist die folgende:

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

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.

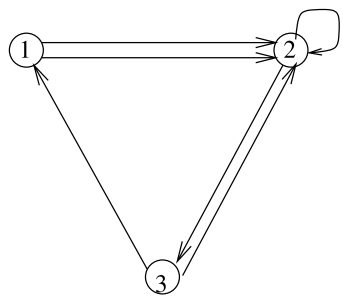


Abb. 3.2

Eine Bogenliste des Digraphen aus Abbildung 3.2 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.

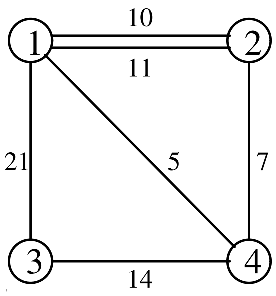


Abb. 3.3

Der gewichtete Graph aus Abbildung 3.3 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.

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

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

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 3.1 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

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

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 $V = \{1, 2, \dots, n\}$ ohne Schleifen ist definiert durch

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

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.

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 3.1 sieht die Adjazenzliste wie folgt aus

	Knotennummer ↓		
4	1	2	2, 3
	2	5	1, 3, 3, 2, 2
	3	4	1, 2, 2, 4
	4	1	3
		↑ Grad	

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) 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 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.

(3.7) Depth-First-Search.

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

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

Alle Knoten $v \in V$ seien unmarkiert.

1. Setze $T := \emptyset$.
2. Für alle $v \in V$ führe aus:

   Ist v unmarkiert, dann CALL SEARCH(v).

   END.
3. Gib T aus.

Rekursives Unterprogramm

PROCEDURE SEARCH(v)

1. Markiere v.
2. Für alle Knoten $w \in N(v)$ führe aus:

   3. Ist w unmarkiert, setze $T := T \cup \{vw\}$ und CALL SEARCH(w).

   END.

END SEARCH. □

In Algorithmus (3.7) 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 (3.7) 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

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

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

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. (1990). *Introduction to Algorithms*. MIT Press, Cambridge, Mass. (wurde mehrfach neu aufgelegt).

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman and Company, New York.

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

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

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

Meinel, C. (1991). *Effiziente Algorithmen. Entwurf und Analyse*. Fachbuchverlag, Leipzig, 1. edition.

Ottmann, T. and Widmayer, P. (1990). *Algorithmen und Datenstrukturen*. BI Wissenschaftsverlag, Mannheim, Reihe Informatik, Band 70, Hrsg. K. H. Böhling and U. Kulisch and H. Maurer edition.

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

Shmoys, D. B. and Tardos, E. (1995). *Computational Complexity*, chapter 29, pages 1599–1645. North-Holland, Amsterdam, R. L. Graham et al. (Hrsg.) edition.

Tarjan, R. E. (1983). Data Structures and Network Algorithms. In *Regional Conference Series in Applied Mathematics*, number 44 in CMBS-NSF Regional conference series in applied mathematics, page 131, Philadelphia. second printing 1985.

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

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

Kapitel 4

Minimale Bäume, maximale Branchings

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.

4.1 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.

Lemma (4.1) *Ein Baum $G = (V, E)$ mit $|V| \geq 2$ hat mindestens zwei Knoten mit Grad 1.*

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. □

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?

Lemma (4.2)

(a) *Für jeden Graphen $G = (V, E)$ gilt: $2\,|E| = \sum_{v \in V} \deg(v)$.*

(b) *Für jeden Baum $G = (V, E)$ gilt: $|E| = |V| - 1$.*

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 (4.1) 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. □

Lemma (4.2) impliziert übrigens auch, dass ein Baum mindestens zwei Knoten mit Grad 1 hat. Wie überlegt man sich das?

Lemma (4.3) *Ein Graph $G = (V, E)$ mit mindestens 2 Knoten und mit weniger als $|V| - 1$ Kanten ist unzusammenhängend.*

Beweis. 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 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.

Satz (4.4) *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) $\Longleftrightarrow$ (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 (4.2).

(2) $\Longrightarrow$ (3) Ist G ein Graph, der (2) erfüllt, so gilt nach Voraussetzung $|E| = |V| - 1$ und nach Lemma (4.2) (a) $2\,|E| = \sum_{v \in V} \deg(v)$. Der Durchschnittsgrad $\deg(v)/|V|$ der Knoten von G ist damit kleiner als 2. In G gibt es daher 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 gilt mit höchstens $n \geq 2$ Knoten 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 (4.3). □

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.

Folgerung (4.5)

(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.

Satz (4.6) *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 $v \in V \setminus \{r\}$. □

4.2 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 (2.11) eingeführt. Beide Probleme sind sehr effizient lösbar und haben vielfältige Anwendungen. Einen umfassenden Überblick über die Geschichte dieser Probleme, ihre Anwendungen und die bekannten Lösungsverfahren gibt der Aufsatz Graham and Hell (1982).

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

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

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

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

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.

(4.7) GREEDY-MAX.

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

Output: *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.* □

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

Beweis : Hausaufgabe! □

Versuchen Sie, einen direkten Beweis für die Korrektheit von Algorithmus (4.7) zu finden. Im nachfolgenden Teil dieses Abschnitts und in Kapitel 5 werden wir Sätze angeben, aus denen Satz (4.8) 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.

(4.9) GREEDY-MIN.

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

Output: *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 (4.8) und unseren Überlegungen zur Reduktion des Waldproblems auf das Baumproblem und umgekehrt folgt.

(4.10) Satz. *Algorithmus (4.9) 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 (4.7) bzw. (4.9) 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 (4.9) häufig **Kruskal-Algorithmus** genannt.

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

(4.11) "Dualer" Greedy-Algorithmus.

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

Output: *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 (4.11) 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.

(4.12) Algorithmus.

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

Output: *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 (4.9) ist ein Spezialfall von Algorithmus (4.12). Überlegen Sie sich wieso!

(4.13) Satz. *Algorithmus (4.12) bestimmt einen aufspannenden Baum minimalen Gewichts.*

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 (4.12) 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-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 (4.12) ist, wird häufig PRIM-Algorithmus genannt.

(4.14) PRIM-Algorithmus.

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

Output: *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

$$T := T \cup \{uv\}$$
$$W := W \cup \{v\}$$
$$V := V \setminus \{v\}$$

und gehe zu 2. □

Das PRIM-Verfahren hat, bei geeigneten Datenstrukturen, eine Laufzeit von $O(m+n \log n)$ und 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 (4.14), das für die Bestimmung minimaler aufspannender Bäume im K_n konzipiert ist.

(4.15) PASCAL-Implementierung von Algorithmus (4.14).

```
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 -}
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 (4.9), (4.11) und (4.14) verdeutlicht.

(4.16) Beispiel. Wir betrachten den in Abbildung 4.1 dargestellten Graphen.

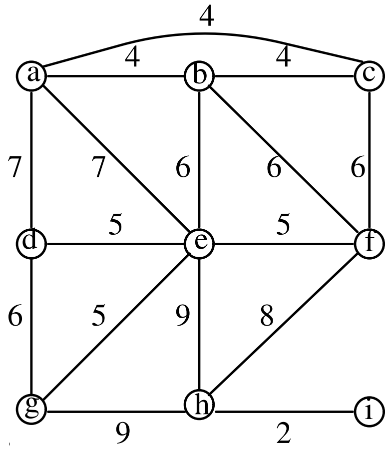


Abb. 4.1

Wir wenden Algorithmus (4.9) 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 (4.9) werden die in der Abbildung 4.2 gezeichneten Kanten ausgewählt.

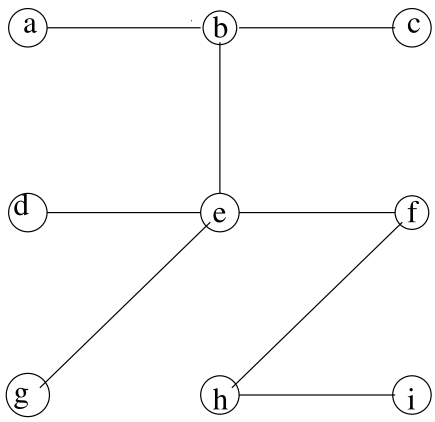


Abb. 4.2

Den Prim-Algorithmus (4.14) starten wir mit dem Knoten $w = a$. Es ergibt sich der in Abbildung 4.3 gezeichnete minimale aufspannende Baum. □

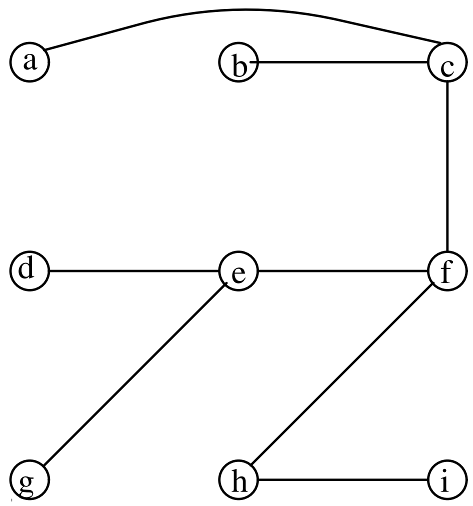


Abb. 4.3

Wie Beispiel (4.16) zeigt, muss ein minimaler Baum nicht eindeutig bestimmt sein. Überlegen Sie sich bitte, wie man feststellen kann, ob ein minimaler aufspannender Baum eindeutig ist.

4.2.1 Optimale Branchings und Arboreszenzen

Die Aufgaben, in einem Digraphen $D = (V, A)$ mit Bogengewichten c_{ij} für alle $(i, j) \in A$ ein Branching maximalen Gewichts bzw. eine Arboreszenz minimalen Gewichts zu finden, sind trivialerweise äquivalent. Wir wollen nun zeigen, dass man ein maximales Branching in polynomialer Zeit konstruieren kann. Der Algorithmus hierzu wurde unabhängig voneinander von Bock (1971), Chu and Liu (1965) und Edmonds (1967) entdeckt. Er ist erheblich komplizierter als die Algorithmen zur Bestimmung maximaler Wälder bzw. minimaler aufspannender Bäume. Insbesondere der Korrektheitsbeweis erfordert einigen Aufwand. Wir wollen bei der Beschreibung des Algorithmus die Inzidenzfunktion $\Psi : A \to V \times V$ eines Digraphen benutzen, da mit ihrer Hilfe die Technik des Schrumpfens von Knotenmengen etwas klarer beschrieben werden kann. Falls $a \in A$, schreiben wir $\Psi(a) := (t(a), h(a))$ um den Anfangs- und den Endknoten von a anzugeben.

(4.17) Der Branching-Algorithmus.

Input: *Ein schlingenfreier Digraph* $D = (V, A)$ *mit Bogengewichten* $c(a)$ *für alle* $a \in A$. $\Psi = (t, h) : A \to V \times V$ *sei die Inzidenzfunktion von* D.

Output: *Ein Branching* B, *dessen Gewicht maximal ist.*

1. *(Initialisierung):*

Setze $D_0 := D, V_0 := V, A_0 := A, \Psi_0 := (t_0, h_0) = \Psi = (t, h)$,

$B_0 := \emptyset$, $c_0 := c, i := 0$. *Alle Knoten aus* V_0 *seien unmarkiert.*

Phase I *(Greedy-Auswahl und Schrumpfung von Kreisen):*

2. *Sind alle Knoten aus* V_i *markiert, gehe zu 11.*
3. *Wähle einen unmarkierten Knoten* $v \in V_i$.
4. *Gilt* $c_i(a) \leq 0$ *für alle* $a \in \delta^-(v) \subseteq A_i$, *markiere* v *und gehe zu 2.*
5. *Unter allen Bögen aus* $\delta^-(v) \subseteq A_i$, *wähle einen Bogen* b *mit maximalem (positiven) Gewicht* $c_i(b)$.
6. *Ist* $B_i \cup \{b\}$ *ein Branching, setze* $B_i := B_i \cup \{b\}$, *markiere* v *und gehe zu 2.*
7. *Ist* $B_i \cup \{b\}$ *kein Branching, dann bestimme den gerichteten Kreis, der von* b *und einigen Bögen aus* B_i *gebildet wird. Seien* C_i *die Bogen- und* W_i *die Knotenmenge dieses Kreises.*
8. *Schrumpfe die Knotenmenge* W_i, *d. h. ersetze die Knotenmenge* W_i *durch einen neuen Knoten* w_i *(genannt Pseudoknoten) und definiere einen neuen Digraphen* $D_{i+1} := (V_{i+1}, A_{i+1})$ *mit Inzidenzfunktion* Ψ_{i+1} *und Gewichten* c_{i+1} *wie folgt:*

$$\begin{array}{ll} V_{i+1} & := (V_i \setminus W_i) \cup \{w_i\} \\ A_{i+1} & := A_i \setminus \{a \in A_i \mid t_i(a), h_i(a) \in W_i\} \end{array}$$

$$\left.\begin{array}{ll} \Psi_{i+1}(a) & := (w_i, h_i(a)) \\ c_{i+1}(a) & := c_i(a) \end{array}\right\} \quad \text{falls } t_i(a) \in W_i \text{ und } h_i(a) \notin W_i$$

$$\left.\begin{array}{ll} \Psi_{i+1}(a) & := \Psi_i(a) \\ c_{i+1}(a) & := c_i(a) \end{array}\right\} \quad \text{falls } t_i(a) \notin W_i \text{ und } h_i(a) \notin W_i$$

$$\left.\begin{array}{ll} \Psi_{i+1}(a) & := (t_i(a), w_i) \\ c_{i+1}(a) & := c_i(a) + c_i(b_i) - c_i(a_i) \end{array}\right\} \quad \text{falls } t_i(a) \notin W_i \text{ und } h_i(a) \in W_i$$

wobei in der letzten Zuweisung b_i *ein Bogen des Kreises* C_i *mit* $c(b_i) = \min\{c(b) \mid b \in C_i\}$ *ist und* a_i *der Bogen des Kreises* C_i *ist, der* $h_i(a)$ *als Endknoten hat.*

9. *Alle Knoten in* $V_{i+1} \setminus \{w_i\}$ *behalten die Markierung, die sie in* D_i *hatten. Der Pseudoknoten* w_i *sei unmarkiert.*
10. *Setze* $B_{i+1} := B_i \setminus C_i$, $i := i + 1$ *und gehe zu 2.*

Phase II *(Rekonstruktion eines maximalen Branchings):*

(Bei Beginn von Phase II sind alle Knoten des gegenwärtigen Digraphen D_i markiert, und die Bogenmenge B_i ist ein Branching in D_i mit maximalem Gewicht $c_i(B_i)$.)

11. *Falls $i = 0$, STOP! (Das Branching B_0 ist ein optimales Branching des gegebenen Digraphen $D = D_0$.)*

12. *Falls der Pseudoknoten w_{i-1} von D_i die Wurzel einer Arboreszenz des Branchings B_i ist, dann setze*

$$B_{i-1} := B_i \cup (C_{i-1} \setminus \{b_{i-1}\}),$$

wobei b_{i-1} ein Bogen des Kreises C_{i-1} ist, der unter den Bögen aus C_{i-1} minimales Gewicht $c_{i-1}(b_{i-1})$ hat.

Setze $i := i - 1$ und gehe zu 11.

13. *Gibt es einen Bogen b in B_i mit $h_i(b) = w_{i-1}$ (d. h. ist w_{i-1} keine Wurzel), dann bestimme den Bogen $a_{i-1} \in C_{i-1}$ mit $h_{i-1}(b) = h_{i-1}(a_{i-1})$ und setze*

$$B_{i-1} := B_i \cup (C_{i-1} \setminus \{a_{i-1}\}).$$

Setze $i := i - 1$ und gehe zu 11.

□

Der "Witz" von Algorithmus (4.17) liegt in der geschickten Definition der modifizierten Kosten in Schritt 8 für die Bögen aus $\delta^-(W_i)$ des Digraphen D_i. Die Greedy-Bogenauswahl hat einen gerichteten Kreis C_i mit Knotenmenge W_i produziert. Die Kosten des Kreises seien $K := c(C_i)$. Da ein Branching keinen Kreis enthalten darf, entfernt man den billigsten Bogen b_i und erhält damit einen gerichteten Weg $P_i := C_i \setminus \{b_i\}$ mit Kosten $K' := K - c(b_i)$. Dieser Weg ist ein potentieller Kandidat zur Aufnahme in das optimale Branching. Es gibt jedoch noch Alternativen zum Weg P, die zu prüfen sind. Statt einfach den billigsten Bogen aus K zu entfernen, kann man einen Bogen a mit Endknoten in W_i und Anfangsknoten außerhalb von W_i zum Kreis K hinzufügen. Um ein Branching zu erhalten, muß dann der eindeutig bestimmte Bogen a_i des Kreises K, der zum Endknoten von a führt, aus K entfernt werden. Dadurch erhält man einen gerichteten Weg P_a. Die Kosten einer solchen Modifikation des Kreises C_i zum Weg P_a sind:

$$K + c_i(a) - c_i(a_i) = K' + c_i(a) + c_i(b_i) - c_i(a_i).$$

Der neue Zielfunktionswert von a, $c_{i+1}(a) := c_i(a) + c_i(b_i) - c_i(a_i)$ mißt also den Wert des Weges P_a im Vergleich zur Wahl des Weges P. Ist $c(a)$ positiv, ist die Alternative P_a günstiger als P.

Bevor wir die Korrektheit des Algorithmus beweisen, besprechen wir ein Beispiel.

(4.18) Beispiel. Wir betrachten den in Abbildung 4.4 dargestellten Digraphen $D = (V, A)$ mit Knotenmenge $1, \dots, 7$ und den eingetragenen Bogengewichten.

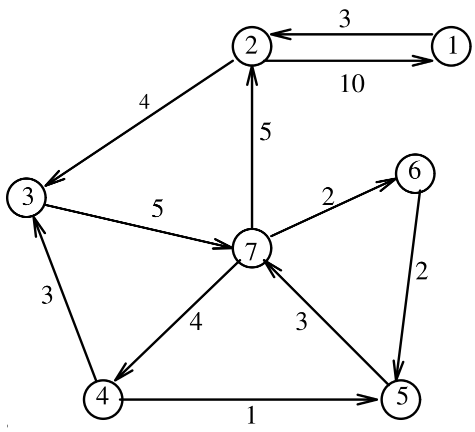


Abb. 4.4

Wir durchlaufen nun die einzelnen Schritte von Algorithmus (4.17). Die Bögen bezeichnen wir nicht mit einem Namen, sondern mit ihrem jeweiligen Anfangs- und Endknoten. In Schritt 1 initialisieren wir wie angegeben.

2. – (“–” heißt, die angegebene Bedingung ist nicht erfüllt, und wir machen nichts.)

3. Wir wählen Knoten 1.

4. –

5. $b = (2, 1)$.

6. $B_0 := \{(2, 1)\}$, wir markieren 1 und gehen zu 2.

2. –

3. Wir wählen Knoten 2.

4. –

5. $b = (7, 2)$.

6. $B_0 := \{(2, 1), (7, 2)\}$, wir markieren 2.

2. –

3. Wir wählen Knoten 3.

4. –

5. $b = (2, 3)$.

6. $B_0 := \{(2, 1), (7, 2), (2, 3)\}$, wir markieren 3.

2. –

3. Wir wählen Knoten 7.
4. –
5. $b = (3, 7)$.
6. –
7. $B_0 \cup \{(3, 7)\}$ enthält den gerichteten Kreis $C_0 = (3, 7, 2)$; es ist $W_0 = \{2, 3, 7\}$.
8. Der neue Digraph $D_1 = (V_1, A_1)$ mit $V_1 = \{w_0, 1, 4, 5, 6\}$ sieht wie in Abb. 4.5 gezeigt aus und hat die in Abb. 4.5 angegebenen Gewichte c_1. Gewichtsänderungen ergeben sich nur bei Bögen mit Anfangsknoten in $V_0 \setminus W_0$ und Endknoten in W_0, also

$$\begin{aligned} c_1((1, w_0)) &:= c_0((1, 2)) + c_0((2, 3)) - c_0((7, 2)) = 3 + 4 - 5 = 2, \\ c_1((4, w_0)) &:= c_0((4, 3)) + c_0((2, 3)) - c_0((2, 3)) = 3 + 4 - 4 = 3, \\ c_1((5, w_0)) &:= c_0((5, 7)) + c_0((2, 3)) - c_0((3, 7)) = 3 + 4 - 5 = 2. \end{aligned}$$

Markierungen von Knoten sind durch ein $\times$ angedeutet.

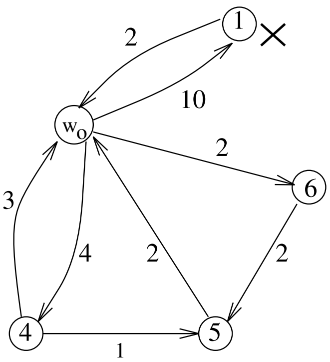


Abb. 4.5

9. In D_1 ist nur Knoten 1 markiert.
10. $B_1 = \{(w_0, 1)\}$.
2. –
3. Wir wählen Knoten 4.
4. –
5. $b = (w_0, 4)$.
6. $B_1 = \{(w_0, 1), (w_0, 4)\}$, wir markieren 4.
2. –

3. Wir wählen Knoten w_0.

4. –

5. $b = (4, w_0)$.

6. –

7. $B_1 \cup \{(4, w_0)\}$ enthält den Kreis $C_1 = \{(4, w_0), (w_0, 4)\}$ mit $W_1 = \{4, w_0\}$.

8. Der neue Digraph $D_2 = (V_2, A_2)$ hat die Knotenmenge $V_2 = \{w_1, 1, 5, 6\}$ und ist in Abbildung 4.6 mit seinen Bogengewichten und Markierungen dargestellt. Die Bogengewichtsänderungen ergeben sich aus:

$$\begin{aligned} c_2((1, w_1)) &:= c_1((1, w_0)) + c_1((4, w_0)) - c_1((4, w_0)) = 2 + 3 - 3 = 2, \\ c_2((5, w_1)) &:= c_1((5, w_0)) + c_1((4, w_0)) - c_1((4, w_0)) = 2 + 3 - 3 = 2. \end{aligned}$$

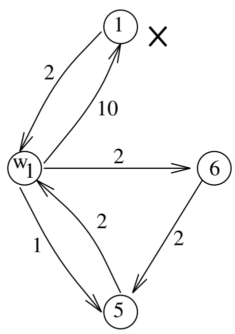


Abb. 4.6

9. In D_2 ist Knoten 1 markiert.

10. $B_2 = \{(w_1, 1)\}$.

2. –

3. Wir wählen Knoten 5.

4. –

5. $b = (6, 5)$.

6. $B_2 := \{(w_1, 1), (6, 5)\}$, wir markieren 5.

2. –

3. Wir wählen Knoten 6.

4. –

5. $b = (w_1, 6)$.

6. $B_2 := \{(w_1, 1), (6, 5), (w_1, 6)\}$, wir markieren 6.

2. –

3. Wir wählen Knoten w_1.

4. –

5. $b = (5, w_1)$.

6. –

7. $B_2 \cup \{(5, w_1)\}$ enthält den Kreis $C_2 = (w_1, 6, 5)$ mit $W_2 = \{w_1, 5, 6\}$.

8. Der neue Digraph $D_3 = (V_3, A_3)$ mit $V_3 = \{1, w_2\}$ ist in Abbildung 4.7 dargestellt.

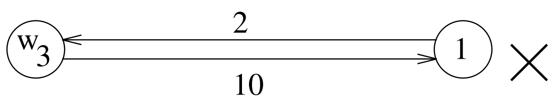


Abb. 4.7

9. In D_3 ist Knoten 1 markiert.

10. $B_3 = \{(w_2, 1)\}$.

2. –

3. Wir wählen Knoten w_2.

4. –

5. $b = (1, w_2)$.

6. –

7. $B_3 \cup \{(1, w_2)\}$ enthält den Kreis $C_3 = \{(1, w_2), (w_2, 1)\}$ mit $W_3 = \{1, w_2\}$.

8. und 9. Der neue Digraph $D_4 = (V_4, A_4)$ besteht aus nur einem unmarkierten Knoten w_3 und enthält keine Bögen.

10. $B_4 = \emptyset$.

2. –

3. Wir wählen Knoten w_3.

4. Wir markieren w_3.

2. Alle Knoten sind markiert.

Wir beginnen nun mit Phase II, der Rekonstruktion des optimalen Branching B von D aus
den Branchings B_i, $i = 4, 3, 2, 1, 0$.

11. $i = 4$.

12. w_3 ist Wurzel (des leeren Branchings B_4).

    $B_3 := B_4 \cup (C_3 \setminus \{b_{i-1}\}) = \emptyset \cup (\{(1, w_2), (w_2, 1)\} \setminus \{(1, w_2)\}) = \{(w_2, 1)\}$.

11. $i = 3$.

12. w_2 ist Wurzel von B_3.

    $B_2 = B_3 \cup (C_2 \setminus \{b_2\}) = \{(w_1, 1)\} \cup (\{(w_1, 6), (6, 5), (5, w_1)\} \setminus \{(w_1, 6)\}) = \{(w_1, 1), (6, 5), (5, w_1)\}$.

    (Man beachte hier, dass der Bogen $(w_2, 1) \in B_3 \subseteq A_3$ dem Bogen $(w_1, 1) \in A_2$ entspricht.)

11. $i = 2$.

12. –

13. Der Bogen $b = (5, w_1) \in A_2$ entspricht dem Bogen $(5, w_0) \in A_1$, also hat $a_1 = a_{i-1} = (4, w_0) \in C_1$ denselben Endknoten wie b.

    $B_1 := B_2 \cup (C_1 \setminus \{a_1\}) = \{(w_0, 1), (6, 5), (5, w_0)\} \cup (\{(4, w_0), (w_0, 4)\} \setminus \{(4, w_0)\}) = \{(w_0, 1), (6, 5), (5, w_0), (w_0, 4)\}$.

11. $i = 1$.

12. –

13. Der Bogen $b = (5, w_0) \in B_1 \subseteq A_1$ entspricht dem Bogen $(5, 7) \in A_0$, also haben $a_0 = (3, 7) \in C_0$ und b denselben Endknoten.

    $B_0 := B_1 \cup (C_0 \setminus \{a_0\}) = \{(2, 1), (6, 5), (5, 7), (7, 4)\} \cup \{(2, 3), (3, 7), (7, 2)\} \setminus \{(3, 7)\} = \{(2, 1), (6, 5), (5, 7), (7, 4), (2, 3), (7, 2)\}$.

11. $i = 0$, STOP, B_0 ist optimal.

Die Phase II des “Aufblasens” stellen wir noch einmal graphisch in Abbildung 4.8 dar. Das optimale Branching $B = B_0 \subseteq A$ hat das Gewicht $2+3+4+5+4+10 = 28$. $\square$

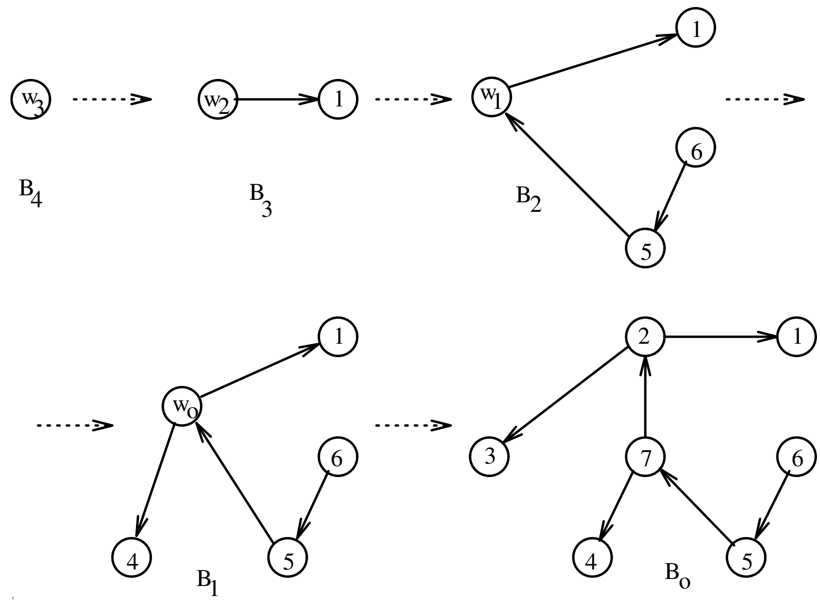


Abb. 4.8

(4.19) Beispiel. Wir betrachten den in Abbildung 4.9 dargestellten Digraphen mit den eingezeichneten Bogengewichten. Wir durchlaufen den Branching-Algorithmus nicht mehr so explizit, wie im vorangegangenen Beispiel, sondern stellen nur noch die Schrumpfungsphasen graphisch dar.

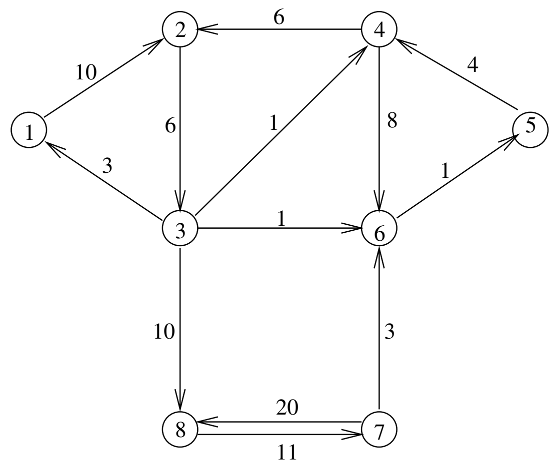


Abb. 4.9

Wir wählen in Schritt 3 nacheinander die Knoten 1, 2, 3 und erhalten den gerichteten Kreis $C_0 = (1, 2, 3)$ mit der Knotenmenge $W_0 = \{1, 2, 3\}$, die zu einem neuen Knoten w_0 wie in Abbildung 4.10 gezeigt, geschrumpft wird.

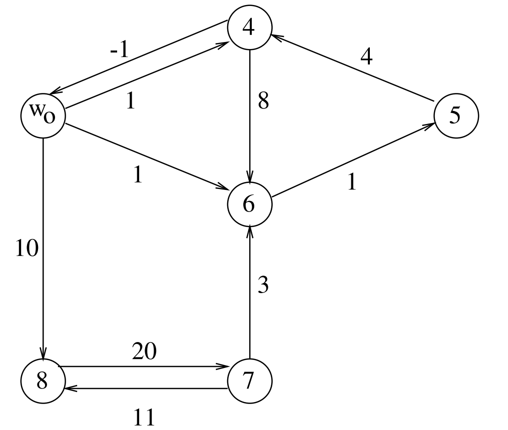


Abb. 4.10

Nun wählen wir in Schritt 3 nacheinander die Knoten w_0, 4, 5, 6. (Da alle in w_0 endenden Bögen negatives Gewicht haben, wird w_0 in Schritt 4 markiert, aber kein Bogen mit Endknoten w_0 gewählt.) Es ergibt sich der Kreis $C_1 = (4, 6, 5)$ mit $W_1 = \{4, 5, 6\}$. Der Schrumpfungsprozess ergibt den in Abbildung 4.11 gezeigten Digraphen.

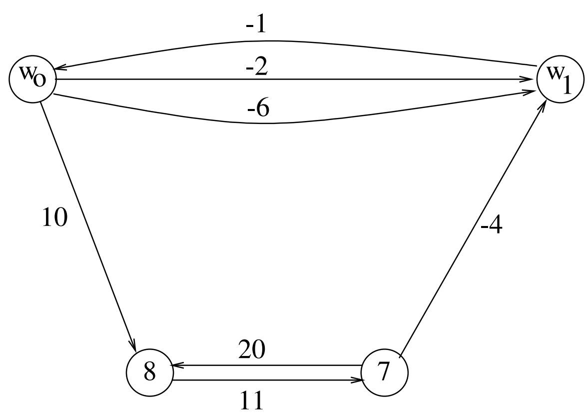


Abb. 4.11

In Schritt 3 wählen wir nun nacheinander die Knoten w_1, 7, 8. Es ergibt sich der gerichtete Kreis $C_2 = (7, 8)$ mit $W_2 = \{7, 8\}$. Wir schrumpfen W_2 zu w_2, wie in Abbildung 4.12 gezeigt.

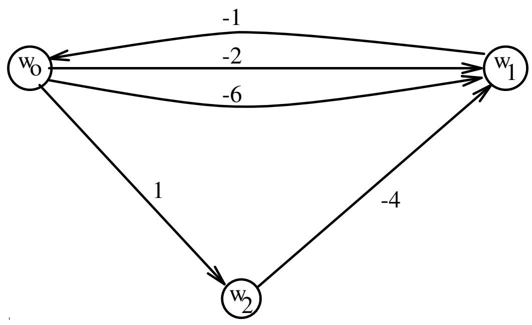


Abb. 4.12

Nun ist lediglich w_2 noch nicht markiert. Wir erhalten den Bogen (w_0, w_2), der ein optimales Branching B_3 des in Abbildung 4.12 gezeigten Digraphen D_3 ist. Alle Knoten von D_3 sind markiert. Wir beginnen die Phase II und rekonstruieren das optimale Branching von D. Dieser Vorgang ist in Abbildung 4.13 dargestellt. Das optimale Branching besteht aus den Bögen $(1, 2), (2, 3), (3, 8), (8, 7), (5, 4)$ und $(4, 6)$. □

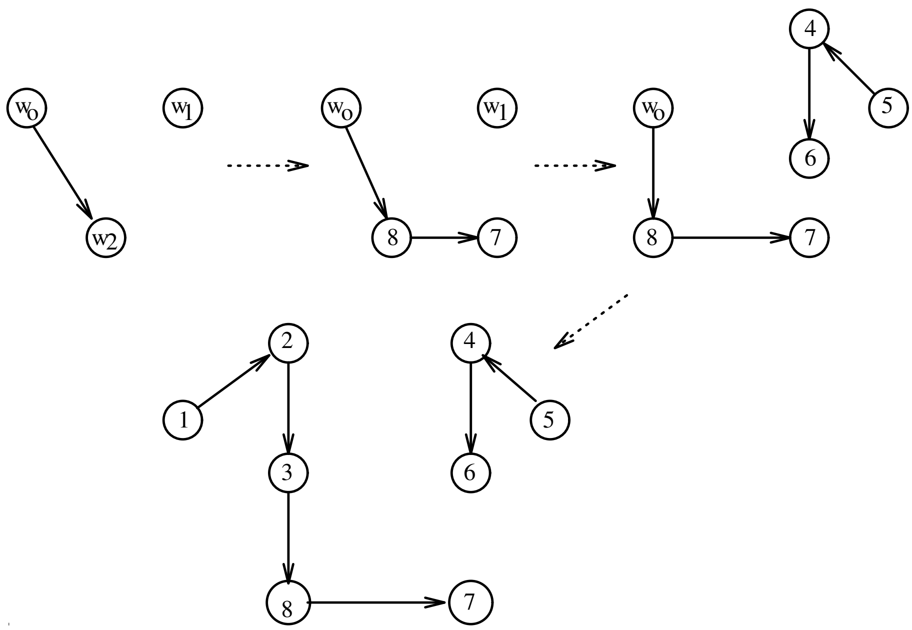


Abb. 4.13

Der Branching-Algorithmus (4.17) kann so implementiert werden, dass seine Laufzeit für einen Digraphen mit m Bögen und n Knoten $O(m \log n)$ bzw. bei Digraphen mit vielen Bögen $O(n^2)$ beträgt, siehe hierzu Tarjan (1977) und Camerini et al. (1979). Im letztgenannten Artikel werden einige inkorrekte Implementierungsdetails des Tarjan-Aufsatzes richtiggestellt. Zum Abschluss des Kapitels wollen wir die Korrektheit des Branching-Algorithmus (4.17) beweisen.

(4.20) Lemma. *Bei der i-ten Iteration der Phase I sei in Schritt 7 bei der Addition des Bogens b zum Branching B_i der gerichtete Kreis $C_i \subseteq B_i \cup \{b\}$ mit Knotenmenge W_i gefunden worden. Dann existiert in $D_i = (V_i, A_i)$ ein bezüglich der Gewichtsfunktion c_i maximales Branching B^* mit folgenden Eigenschaften:*

(a) $\left| B^* \cap \Big(\big(\bigcup\limits_{v \in W_i} \delta^-(v) \big) \setminus C_i \Big) \right| \leq 1,$

(b) $|B^* \cap C_i| = |C_i| - 1.$

Beweis : Wir nehmen an, dass Behauptung (a) falsch ist.
Setze $\Delta_i := (\bigcup\limits_{v \in W_i} \delta^-(v)) \setminus C_i$. Wir wählen ein Branching B^* von D_i maximalen Gewichts bezüglich c_i, so dass die Anzahl der Bögen in $\Delta := B^* \cap \Delta_i$ minimal ist. Zur Notationsvereinfachung nehmen wir an, dass $C_i = (1, 2, \dots, r)$ und $\Delta = \{(u_j, v_j) \mid j = 1, \dots, k\}$ gilt (nach Annahme gilt $k \geq 2$). Wir können ferner o. B. d. A. annehmen, dass die Knoten $v_j \in W_i$ so numeriert sind, dass $v_p < v_q$ gilt, falls $p < q$.

Wir untersuchen nun die Mengen $B_j^* := (B^* \setminus \{(u_j, v_j)\}) \cup \{(v_j - 1, v_j)\}$ für $j = 1, \dots, k$. Wäre für irgendeinen Bogen $(u_j, v_j) \in \Delta$ die Menge B_j^* ein Branching, so wäre aufgrund der Auswahlvorschriften in Schritt 5 $c_i(B_j^*) - c_i(B^*) = c_i((v_j - 1, v_j)) - c_i((u_j, v_j)) \geq 0$. D. h. B_j^* wäre ein maximales Branching mit weniger Bögen in Δ_i als B^*, ein Widerspruch zu unserer Minimalitätsannahme. Folglich enthält B_j^* einen Kreis, der $(v_j - 1, v_j)$ enthält. Da in B_j^* keine zwei Bögen einen gemeinsamen Endknoten besitzen, ist dieser Kreis gerichtet. Daraus folgt, dass B_j^* — und somit B^* — einen gerichteten $(v_j, v_j - 1)$-Weg P_j enthält. Enthielte P_j nur Bögen des Kreises C_i, dann ergäbe sich $P_j = C_i \setminus \{(v_j - 1, v_j)\}$; daraus folgte, dass außer v_j kein weiterer Knoten aus W_i Endknoten eines Bogens aus Δ sein könnte. Dies widerspräche unserer Annahme $k \geq 2$.

Mithin enthält P_j mindestens einen Bogen aus Δ. Sei d der letzte (gesehen in Richtung v_j nach $v_j - 1$) Bogen von P_j, der in Δ liegt. Dann muss d der Bogen (u_{j-1}, v_{j-1}) sein, da jeder gerichtete Weg in B^* von irgendeinem Knoten $v \in V \setminus \{v_{j-1}, v_{j-1} + 1, v_{j-1} + 2, \dots, v_j - 1\}$ zum Knoten v_{j-1} den Weg $(v_{j-1}, v_{j-1} + 1, \dots, v_j - 1)$ enthalten muß und v_{j-1} in B^* nur über den Bogen (u_{j-1}, v_{j-1}) zu erreichen ist. Mithin enthält B^* einen gerichteten Weg $\overline{P}_j$ von v_j nach v_{j-1}. Diese Aussage gilt für $j = 1, \dots, k$ (wobei $v_0 = v_k$ zu setzen ist). Setzen wir diese Wege wie folgt zusammen

$$v_k, \overline{P}_k, v_{k-1}, \overline{P}_{k-1}, v_{k-2}, \dots, v_2 \overline{P}_2, v_1, \overline{P}_1, v_0 = v_k,$$

so ergibt sich eine geschlossene gerichtete Kette. Diese Kette enthält einen gerichteten Kreis, und dieser Kreis ist eine Teilmenge von B^*. Widerspruch! Damit ist der Beweis von (a) erledigt.

Sei nun B^* ein maximales Branching, das (a) erfüllt und für das $|C_i \cap B^*|$ maximal ist. Enthält B^* keinen Bogen aus Δ_i, so ist nach (a) $B^* \cap (A(W_i) \cup \delta^-(W_i)) \subseteq C_i$. Trivialerweise muss dann $|B^* \cap C_i| = |C_i| - 1$ gelten. Enthält B^* einen Bogen aus

Δ_i, sagen wir a, dann sei a_i der Bogen aus C_i mit $h_i(a_i) = h_i(a)$. Gilt $|B^* \cap C_i| < |C_i| - 1$, so ist $B' := (B^* \setminus \{a\}) \cup \{a_i\}$ ein Branching, für das aufgrund von Schritt 5 gilt $c_i(B') \geq c_i(B^*)$. Dies widerspricht unserer Annahme, dass $|C_i \cap B^*|$ maximal ist. Damit ist auch (b) gezeigt. □

(4.21) Satz. *Zu jedem Digraphen $D = (V, A)$ mit Gewichten $c(a)$ für alle $a \in A$ liefert Algorithmus (4.17) ein maximales Branching.*

Beweis : Führen wir die Iteration der Phase I zum letzten Mal (sagen wir k-ten Mal) aus, so haben alle Bögen d des gegenwärtigen Branchings $B_k \subseteq A_k$ die Eigenschaft $c_k(d) \geq c_k(a)$ für alle Bögen a mit $h_k(a) = h_k(d)$ (wegen Schritt 5). Für die Knoten $v \in V_k$, die nicht Endknoten eines Bogens aus B_k sind, gilt wegen Schritt 4 $c_k(a) \leq 0$ für alle $a \in \delta^-(v)$. B_k ist also offensichtlich ein maximales Branching in D_k, das nur Bögen mit positivem c_k-Gewicht enthält. Durch Induktion zeigen wir nun: Ist B_i ein maximales Branching von D_i bezüglich c_i mit $c_i(b) > 0 \; \forall \, b \in B_i$, so ist das in den Schritten 12 oder 13 definierte Branching B_{i-1} maximal in D_{i-1} bezüglich c_{i-1} mit $c_{i-1}(b) > 0 \; \forall \, b \in B_{i-1}$. Der Induktionsanfang für $i = k$ ist durch die obige Bemerkung gegeben.

Wir nehmen an, dass die Behauptung für i, $0 < i \leq k$, richtig ist und wollen zeigen, dass sie auch für $i - 1$ gilt. Sei also B_i das durch den Algorithmus gefundene maximale Branching von D_i bezüglich c_i, sei $C_{i-1} \subseteq B_{i-1} \cup \{b\}$ der in Schritt 7 gefundene gerichtete Kreis mit Knotenmenge W_{i-1}, sei $b_{i-1} \in C_{i-1}$ ein Bogen minimalen c_{i-1}-Gewichts, und sei B^* ein bezüglich c_{i-1} maximales Branching von D_{i-1}, das die Bedingungen (a) und (b) von Lemma (4.20) erfüllt.

1. Fall: $|B^* \cap \delta^-(W_{i-1})| = 1$, sagen wir $\{a\} = B^* \cap \delta^-(W_{i-1})$. Dann hat B^* die Form $B^* = B' \cup \{a\} \cup (C_{i-1} \setminus \{a_{i-1}\})$, wobei $B' \subseteq A_{i-1} \setminus (\delta^-(W_{i-1}) \cup A(W_{i-1}))$ und $h_{i-1}(a) = h_{i-1}(a_{i-1})$ gilt, und $B' \cup \{a\}$ ein Branching in D_i ist. Da B_i ein c_i-maximales Branching von D_i ist, gilt

$$\begin{aligned} c_{i-1}(B^*) \; &= c_{i-1}(B' \cup \{a\}) + c_{i-1}(C_{i-1} \setminus \{a_{i-1}\}) \\ &= c_i(B' \cup \{a\}) - c_{i-1}(b_{i-1}) + c_{i-1}(a_{i-1}) + c_{i-1}(C_{i-1} \setminus \{a_{i-1}\}) \\ &\leq c_i(B_i) + c_{i-1}(C_{i-1} \setminus \{b_{i-1}\}) \end{aligned}$$

Enthält B_i keinen Bogen aus $\delta^-(W_{i-1})$, so ist $B_i \cup (C_{i-1} \setminus \{b_{i-1}\})$, das in Schritt 12 konstruierte Branching B_{i-1}. Aus $c_i(B_i) = c_{i-1}(B_i)$ folgt $c_{i-1}(B_{i-1}) \geq c_{i-1}(B^*)$ und damit die Maximalität von B_{i-1} in D_{i-1} bezüglich c_{i-1}. Enthält B_i einen Bogen $b \in \delta^-(W_{i-1})$ und ist $d_{i-1} \in C_{i-1}$ mit $h_{i-1}(d_{i-1}) = h_{i-1}(b)$, so gilt $B_{i-1} = B_i \cup (C_{i-1} \setminus \{d_{i-1}\})$ und folglich

$$\begin{aligned} c_{i-1}(B_{i-1}) \; &= c_i(B_i) - c_{i-1}(b_{i-1}) + c_{i-1}(d_{i-1}) + c_{i-1}(C_{i-1} \setminus \{d_{i-1}\}) \\ &= c_i(B_i) + c_{i-1}(C_{i-1} \setminus \{b_{i-1}\}) \geq c_{i-1}(B^*). \end{aligned}$$

Also ist B_{i-1} maximal.

2. Fall: $B^* \cap \delta^-(W_{i-1}) = \emptyset$. Der Beweis verläuft analog zum 1. Fall.

Sei also B_i maximal in D_i bezüglich der Gewichtsfunktion c_i. Sei $C_{i-1} \subseteq B_{i-1} \cup \{b\}$ der in Schritt 7 gefundene gerichtete Kreis mit Knotenmenge W_{i-1}, und sei B^* ein bezüglich c_{i-1} maximales Branching von D_{i-1}, das eine minimale Zahl von Bögen aus $\delta^-(W_{i-1})$ enthält. Ein solches Branching hat nach Lemma (4.20) (a) höchstens einen Bogen in $\delta^-(W_{i-1})$. Wir setzen $B^* = B' \cup B'' \cup B'''$ mit

$$\begin{aligned} B' &:= B^* \setminus (A(W_{i-1}) \cup \delta^-(W_{i-1}), \\ B'' &:= B^* \cap A(W_{i-1}), \\ B''' &:= B^* \cap \delta^-(W_{i-1}). \end{aligned}$$

Nach Annahme ist B''' entweder leer oder elementig. Sei B_i das durch den Algorithmus bestimmte Branching von D_i, das nach Induktionsannahme in D_i maximal bezüglich der Gewichte c_i ist (und dessen Bögen positive c_i-Gewichte haben). Sei C_{i-1} der in Schritt 7 der Iteration $i-1$ gefundene gerichtete Kreis, und b_{i-1} sei ein Bogen des Kreises C_{i-1} mit kleinstem (positivem) c_{i-1}-Gewicht.

1. Fall: $B''' = \emptyset$. Nach Lemma (4.20) (b) ist dies gleichwertig mit $c_i(a) \leq 0$ für alle $a \in \delta^-(w_{i-1}) \subseteq A_i$. Also ist $B_i \cap \delta^-(w_{i-1}) = \emptyset$. Aufgrund der Auswahl der Bögen in Schritt 5 gilt $c_{i-1}(C_{i-1} \setminus \{b_{i-1}\}) \geq c_{i-1}(B'')$. Ferner ist B' ein Branching in D_i, woraus $c_i(B') \leq c_i(B_i)$ folgt. Da w_{i-1} eine Wurzel in B_i ist, gilt nach Schritt 12 $B_{i-1} = B_i \cup (C_{i-1} \setminus \{b_{i-1}\})$. Daraus ergibt sich

$$\begin{aligned} c_{i-1}(B_{i-1}) &= c_{i-1}(B_i) + c_{i-1}(C_{i-1} \setminus \{b_{i-1}\}) = c_i(B_i) + c_{i-1}(C_{i-1} \setminus \{b_{i-1}\}) \\ &\geq c_i(B') + c_{i-1}(B'') = c_{i-1}(B') + c_{i-1}(B'') = c_{i-1}(B^*). \end{aligned}$$

Mithin ist B_{i-1} c_{i-1}-maximal in D_{i-1}.

2. Fall: $B''' = \{d\}$. Es sei d_{i-1} der Bogen in C_{i-1} mit $h_{i-1}(d_{i-1}) = h_{i-1}(d)$. Nach Lemma (4.20) (b) ist $B''' \neq \emptyset$ gleichwertig damit, dass mindestens ein Bogen aus $\delta^-(w_{i-1})$ positives c_i-Gewicht hat. Aus den Vorschriften in (4.17) folgt, dass w_{i-1} keine Wurzel von B_i sein kann. Das heißt, es gibt einen Bogen $\overline{a}_i \in B_i$ mit $h_i(\overline{a}_i) = w_{i-1}$. Ferner sei a_{i-1} der Bogen in C_{i-1} mit $h_{i-1}(\overline{a}_i) = h_{i-1}(a_{i-1})$. Nach Schritt 13 ist dann $B_{i-1} = B_i \cup (C_{i-1} \setminus \{a_{i-1}\})$. Ferner sei b_{i-1} ein Bogen aus C_{i-1} mit kleinstem c_{i-1}-Gewicht. Aufgrund der Auswahlvorschriften in Schritt 5 gilt $c_{i-1}(C_{i-1}) \geq c_{i-1}(B'') + c_{i-1}(d)$. Ferner ist $B' \cup \{d\}$ ein Branching in D_i, woraus $c_i(B_i) \geq c_i(B') + c_i(d)$ folgt. Daraus ergibt sich nun

$$\begin{aligned} c_{i-1}(B^*) &= c_{i-1}(B') + c_{i-1}(d) + c_{i-1}(B'') \\ &= c_i(B') + c_i(d) - c_{i-1}(b_{i-1}) + c_{i-1}(d_{i-1}) + c_{i-1}(B'') \\ &\leq c_i(B_i) - c_{i-1}(b_{i-1}) + c_{i-1}(d_{i-1}) + c_{i-1}(B'') \\ &= c_{i-1}(B_i) + c_{i-1}(b_{i-1}) - c_{i-1}(a_{i-1}) - c_{i-1}(b_{i-1}) + c_{i-1}(d_{i-1}) + c_{i-1}(B'') \\ &\leq c_{i-1}(B_i) - c_{i-1}(a_{i-1}) + c_{i-1}(C_{i-1}) \\ &= c_{i-1}(B_{i-1}). \end{aligned}$$

Damit ist die Behauptung bewiesen. □

Literaturverzeichnis

Bock, F. (1971). *An algorithm to construct a minimum directed spanning tree in a directed network*, pages 29–44. Gordon and Breach, New York, in: B. Avi-Itzhack (ed.), Developments in Operations Research edition.

Camerini, P. M., Fratta, L., and Maffioli, F. (1979). A Note on Finding Optimum Branchings. *Networks*, 9:309–312.

Chu, Y. J. and Liu, T. H. (1965). On the shortest arborescence of a directed graph. *Scientia Sinica*, 4:1396–1400.

Edmonds, J. (1967). Optimum branchings. *Journal of Research of the National Bureau of Standards, Section B*, 71:233–240.

Graham, R. L. and Hell, P. (1982). On the History of the Minimum Spanning Tree Problem. *Annals of the History of Computing*, 7:43–57.

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

Tarjan, R. E. (1977). Finding Optimum Branchings. *Networks*, 7:25–35.

Kapitel 5

Matroide und Unabhängigkeitssysteme

Wir werden nun einige sehr allgemeine Klassen von kombinatorischen Optimierungsproblemen kennenlernen. Diese enthalten die im vorigen Kapitel betrachteten Probleme. Die in Kapitel 4 besprochenen Algorithmen sind ebenfalls auf die neuen Probleme übertragbar. Allerdings treten bei der algorithmischen Behandlung dieser Probleme einige Schwierigkeiten auf, die wir eingehend diskutieren müssen.

Gute Bücher über Matroidtheorie sind Oxley (1992), Truemper (1992) und Welsh (1976), ein neuerer Übersichtsartikel Welsh (1995). Optimierungsprobleme auf Matroiden werden z. B. in Bixby and Cunningham (1995) ausführlich behandelt.

5.1 Allgemeine Unabhängigkeitssysteme

E sei im folgenden immer eine endliche Menge, 2^E bezeichne die Menge aller Teilmengen von E.

(5.1) Definition. *Eine Menge $\mathcal{I} \subseteq 2^E$ heißt* **Unabhängigkeitssystem** *(oder* **monotones Mengensystem***) auf E, wenn $\mathcal{I}$ die folgenden Axiome erfüllt:*

(I.1) $\emptyset \in \mathcal{I}$,

(I.2) $F \subseteq G \in \mathcal{I} \Longrightarrow F \in \mathcal{I}$.

Häufig wird auch das Paar $(E, \mathcal{I})$ Unabhängigkeitssystem genannt. Die Teilmengen von E, die in $\mathcal{I}$ enthalten sind, heißen **unabhängige Mengen***, alle übrigen Teilmengen von E heißen* **abhängige Mengen**. □

Mit jedem Unabhängigkeitssystem $(E,\mathcal{I})$ sind auf kanonische Weise andere Mengensysteme gegeben, die wir nun kurz einführen wollen.

Die bezüglich Mengeninklusion minimalen abhängigen Teilmengen von E heißen **Zirkuits** (oder **Kreise**), d. h. $C \subseteq E$ ist ein Zirkuit, wenn C abhängig ist und wenn C keine von sich selbst verschiedene abhängige Teilmenge enthält. Die Menge aller Zirkuits (bzgl. eines Unabhängigkeitssystems $\mathcal{I}$) heißt **Zirkuitsystem** und wird mit $\mathcal{C}$ bezeichnet.

Ist $F \subseteq E$, so heißt jede Teilmenge von F, die unabhängig ist und die in keiner anderen unabhängigen Teilmenge von F enthalten ist, **Basis** von F, d. h.

$$B \text{ Basis von } F \iff (B, B' \in \mathcal{I}, B \subseteq B' \subseteq F \Longrightarrow B = B').$$

Die Menge aller Basen der Grundmenge E heißt **Basissystem** (bzgl. $\mathcal{I}$) und wird mit $\mathcal{B}$ bezeichnet.

Für jede Menge $F \subseteq E$ heißt die ganze Zahl

$$r(F) := \max\{|B| \mid B \text{ Basis von } F\} \tag{5.2}$$

Rang von F. Die Rangfunktion r ist also eine Funktion, die 2^E in die nicht-negativen ganzen Zahlen abbildet.

Offenbar induziert jedes Unabhängigkeitssystem $\mathcal{I}$ auf E ein eindeutig bestimmtes Zirkuitsystem, ein eindeutig bestimmtes Basissystem und eine eindeutig bestimmte Rangfunktion. Es gilt auch die Umkehrung.

Zirkuitsysteme und Basissysteme sind nach Definition **Antiketten (Clutter)**, d. h. Systeme von Mengen, so dass keine zwei Mengen ineinander enthalten sind.

Ist $\mathcal{B} \neq \emptyset$ eine Antikette auf E, so ist

$$\mathcal{I} := \{I \subseteq E \mid \exists\, B \in \mathcal{B} \text{ mit } I \subseteq B\} \tag{5.3}$$

ein Unabhängigkeitssystem auf E, und $\mathcal{B}$ ist das zu $\mathcal{I}$ gehörige Basissystem.

Ist $\mathcal{C} \neq \{\emptyset\}$ eine Antikette auf E, so ist

$$\mathcal{I} := \{I \subseteq E \mid I \text{ enthält kein Element von } \mathcal{C}\} \tag{5.4}$$

ein Unabhängigkeitssystem, und $\mathcal{C}$ ist das zu $\mathcal{I}$ gehörige Zirkuitsystem.

Die oben definierte Rangfunktion hat folgende Eigenschaften. Sie ist **subkardinal**, d. h. für alle $F \subseteq E$ gilt

$$r(F) \leq |F|,$$

sie ist **monoton**, d. h. für alle $F, G \subseteq E$ gilt

$$F \subseteq G \Longrightarrow r(F) \leq r(G),$$

und sie ist **stark subadditiv**, d. h. für alle $F \subseteq E$, für alle ganzen Zahlen $k \geq 1$ und für alle Familien $(F_i, i \in K)$ von Teilmengen von F mit der Eigenschaft, dass $|\{i \in K \mid e \in F_i\}| = k$ für alle $e \in F$, gilt

$$k \cdot r(F) \leq \sum_{i \in K} r(F_i).$$

Ist $r : 2^E \to \mathbb{Z}_+$ eine subkardinale, monotone, stark subadditive Funktion, so ist

$$\mathcal{I} := \{I \subseteq E \mid r(I) = |I|\} \tag{5.5}$$

ein Unabhängigkeitssystem, dessen Rangfunktion die Funktion r ist.

Unabhängigkeitssysteme $\mathcal{I}$ auf einer Grundmenge E definieren also mathematische Strukturen, die äquivalent durch Zirkuitsysteme, Basissysteme oder Rangfunktionen gegeben werden können. Unabhängigkeitssysteme sind sehr allgemeine Objekte und besitzen zu wenig Struktur, um tiefliegende Aussagen über sie machen zu können.

Sei für jedes Element $e \in E$ ein "Gewicht" $c_e \in \mathbb{R}$ gegeben. Für $F \subseteq E$ setzen wir wie üblich

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

Das Problem, eine unabhängige Menge $I^* \in \mathcal{I}$ zu finden, so dass $c(I^*)$ maximal ist, heißt **Optimierungsproblem über einem Unabhängigkeitssystem** $\mathcal{I}$, d. h. wir suchen

$$\max\{c(I) \mid I \in \mathcal{I}\}. \tag{5.6}$$

Offenbar macht es hier keinen Sinn, Gewichte c_e zu betrachten, die nicht positiv sind. Denn wenn $I^* \in \mathcal{I}$ optimal ist, gilt $I' := I^* \setminus \{e \in E \mid c_e \leq 0\} \in \mathcal{I}$ (wegen (I.2)) und $c(I') \geq c(I^*)$, also ist I' ebenfalls eine optimale unabhängige Menge. Deswegen werden wir uns im Weiteren bei Optimierungsproblemen über Unabhängigkeitssystemen auf Gewichtsfunktionen beschränken, die positiv oder nicht-negativ sind.

Bei Optimierungsproblemen über Basissystemen ist es dagegen auch sinnvoll, negative Gewichte zuzulassen. Ist ein Basissystem $\mathcal{B}$ auf der Grundmenge E gegeben und sind $c_e \in \mathbb{R}$ Gewichte, so nennen wir

$$\min\{c(B) \mid B \in \mathcal{B}\} \tag{5.7}$$

Optimierungsproblem über einem Basissystem.

(5.8) Beispiele.

(a) Sei $G = (V, E)$ ein Graph. Eine Knotenmenge $S \subseteq V$ heißt **stabil (Clique)**, falls je zwei Knoten aus S nicht benachbart (benachbart) sind. Die Menge der stabilen Knotenmengen (Cliquen) ist ein Unabhängigkeitssystem auf V. Die Aufgabe – bei gegebenen Knotengewichten – eine gewichtsmaximale stabile Menge (Clique) zu finden, ist ein Optimierungsproblem über einem Unabhängigkeitssystem, vergleiche (2.13).

(b) Ein **Wald** in einem Graphen $G = (V, E)$ ist eine Kantenmenge, die keinen Kreis enthält. Die Menge aller Wälder bildet ein Unabhängigkeitssystem auf E. Das Problem, einen maximalen Wald in G zu finden, war Hauptthema von Kapitel 4, siehe auch (2.11). Offenbar ist in einem zusammenhängenden Graphen G die Menge der aufspannenden Bäume genau die Menge der Basen des Unabhängigkeitssystems der Wälder. Das Problem, einen minimalen aufspannenden Baum zu finden, ist somit ein Optimierungsproblem über einem Basissystem.

(c) Ein **Matching** in einem Graphen $G = (V, E)$ ist eine Kantenmenge $M \subseteq E$, so dass jeder Knoten aus V in höchstens einer Kante aus M enthalten ist. Die Menge aller Matchings ist ein Unabhängigkeitssystem auf E. Das Matchingproblem (2.10) ist also ein Optimierungsproblem über einem Unabhängigkeitssystem. Die Aufgabe, in einem vollständigen Graphen mit Kantengewichten ein minimales perfektes Matching zu finden, ist ein Optimierungsproblem über einem Basissystem.

(d) Gegeben sei ein vollständiger Digraph $D_n = (V, A)$ mit n Knoten und Bogenlängen c_{ij} für alle $(i, j) \in A$. Eine **Tour (gerichteter Hamiltonkreis)** ist ein gerichteter Kreis in D_n, der jeden Knoten enthält. Die Aufgabe, eine Tour mit minimalem Gewicht zu finden, heißt **asymmetrisches Travelling-Salesman-Problem**, siehe (2.12). Die Menge $\mathcal{T}$ aller Touren ist kein Unabhängigkeitssystem, jedoch eine Antikette, also Basissystem eines Unabhängigkeitssystems. Das asymmetrische TSP ist also ein Optimierungsproblem über einem Basissystem. Wir können es aber auch als Optimierungsproblem über einem Unabhängigkeitssystem auffassen. Dies geht wie folgt: Setzen wir

$$\tilde{\mathcal{T}} := \{I \subseteq A \mid \exists T \in \mathcal{T} \text{ mit } I \subseteq T\},$$

$$c'_{ij} := \max\{|c_{ij}| \mid (i, j) \in A\} + 1 - c_{ij},$$

so ist $\tilde{\mathcal{T}}$ ein Unabhängigkeitssystem, und jede Lösung von $\max\{c'(I) \mid I \in \tilde{\mathcal{T}}\}$ ist eine Tour, die – bezüglich der Gewichte c_{ij} – minimales Gewicht hat. (Auf die gleiche Weise kann man viele andere Optimierungsprobleme über Basissystemen in Optimierungsprobleme über Unabhängigkeitssystemen überführen.) Ebenso ist das symmetrische TSP ein Optimierungsproblem über einem Basissystem.

(e) Gegeben sei ein gerichteter Graph $D = (V, A)$. Ein **Branching** in D ist eine Bogenmenge $B \subseteq A$, die keinen Kreis (im ungerichteten Sinne) enthält, und die die Eigenschaft hat, dass jeder Knoten $v \in V$ Endknoten von höchstens einem Bogen aus B ist. Die Menge aller Branchings ist ein Unabhängigkeitssystem auf A, siehe Kapitel 4 und (2.11). Das Problem, in einem vollständigen Digraphen eine minimale aufspannende Arboreszenz zu finden, ist ein Optimierungsproblem über einem Basissystem. $\square$

Überlegen Sie sich, welche der übrigen Beispiele aus Abschnitt 2.3 als Optimierungsprobleme über Unabhängigkeits- oder Basissystemen aufgefasst werden können und welche nicht.

Verschiedene praktische Fragestellungen führen auch zu Optimierungsproblemen über Zirkuits. Sind die Elemente $e \in E$ durch Gewichte c_e bewertet, so kann man das Problem untersuchen, ein Zirkuit $C \in \mathcal{C}$ zu finden, das minimales Gewicht $c(C)$ hat. Die Aufgabe, in einem Graphen einen kürzesten Kreis zu bestimmen, ist z. B. von diesem Typ.

Allgemeiner noch ist folgende Frage von Interesse. Wir sagen, dass eine Menge $Z \subseteq E$ ein **Zyklus** ist, wenn Z die Vereinigung von paarweise disjunkten Zirkuits ist, d. h. wenn es Zirkuits $C_1, \ldots, C_k$ gibt mit $C_i \cap C_j = \emptyset$, $1 \leq i < j \leq k$, so dass $Z = \bigcup_{i=1}^{k} C_i$. Sind die Elemente $e \in E$ mit Gewichten c_e belegt, so sucht man nach einem Zyklus maximalem Gewichts. Das Chinesische Postbotenproblem (siehe (2.12)) und das Max-Cut-Problem (2.15) sind z. B. von diesem Typ. Aus Zeitgründen können wir auf Optimierungsprobleme über Zirkuits bzw. Zyklen nicht eingehen, siehe hierzu Barahona and Grötschel (1986) und Grötschel and Truemper (1989).

5.2 Matroide

Wie die Beispiele aus dem vorigen Abschnitt zeigen, enthalten Optimierungsprobleme über Unabhängigkeitssystemen sowohl polynomial lösbare als auch $\mathcal{NP}$-vollständige Probleme. Man wird daher nicht erwarten können, dass für diese Probleme eine "gute" Lösungstheorie existiert. Wir wollen nun eine Spezialklasse von Unabhängigkeitssystemen einführen, für die es so etwas gibt. In einem noch zu präzisierenden Sinn (siehe Folgerung (5.16)) ist dies die Klasse der Unabhängigkeitssysteme, für die der Greedy-Algorithmus (siehe (4.7)) eine Optimallösung liefert.

(5.9) Definition. *Ein* **Matroid** M *besteht aus einer Grundmenge* E *zusammen mit einem Unabhängigkeitssystem* $\mathcal{I} \subseteq 2^E$, *das eine der folgenden äquivalenten Bedingungen erfüllt:*

(I.3) $\quad I, J \in \mathcal{I}, |I| = |J| - 1 \Longrightarrow \exists\, j \in J \setminus I$ *mit* $I \cup \{j\} \in \mathcal{I}$,

(I.3′) $\quad I, J \in \mathcal{I}, |I| < |J| \Longrightarrow \exists\, K \subseteq J \setminus I$ *mit* $|I \cup K| = |J|$, *so daß* $I \cup K \in \mathcal{I}$,

(I.3″) $\quad F \subseteq E$ *und* B, B' *Basen von* $F \Longrightarrow |B| = |B'|$. □

Das heißt also, das Unabhängigkeitssystem eines Matroids auf E ist ein Mengensystem $\mathcal{I} \subseteq 2^E$, das die Axiome (I.1), (I.2) und eines der Axiome (I.3), (I.3′), (I.3″) erfüllt. Ein solches System erfüllt automatisch auch die beiden übrigen der drei Axiome (I.3), (I.3′), (I.3″).

Ein Wort zur Terminologie! Nach der obigen Definition ist ein Matroid M ein Paar $(E, \mathcal{I})$ mit den oben aufgeführten Eigenschaften. Wenn klar ist, um welche Grundmenge E es sich handelt, spricht man häufig auch einfach von dem Matroid $\mathcal{I}$, ohne dabei E explizit zu erwähnen. Man sagt auch, M (bzw. $\mathcal{I}$) ist ein Matroid auf E.

In Definition (5.9) haben wir von den "äquivalenten Bedingungen" (I.3), (I.3′), (I.3″) gesprochen. Diese Äquivalenz muss natürlich bewiesen werden. Da wir hier jedoch keine Vorlesung über Matroide halten wollen, können wir auf die Beweise (aus Zeitgründen) nicht eingehen. Das gleiche gilt für die nachfolgend gemachten Aussagen über Zirkuit-, Basissysteme und Rangfunktionen. Beweise der hier gemachten Aussagen findet der interessierte Leser z.B. in Oxley (1992) und Welsh (1976).

Wie wir bereits gesehen haben, können Unabhängigkeitssysteme über Zirkuits, Basen oder Rangfunktionen beschrieben werden. Ist $M = (E, \mathcal{I})$ ein Matroid und sind $\mathcal{C}, \mathcal{B}, r$ das zugehörige Zirkuit-, Basissystem bzw. die zugehörige Rangfunktion, so ist natürlich $\mathcal{I}$ durch die Angabe von $\mathcal{C}$, $\mathcal{B}$ oder r eindeutig beschrieben. Gelegentlich werden daher auch die Paare $(E, \mathcal{C})$, $(E, \mathcal{B})$, (E, r) als Matroide bezeichnet, meistens dann, wenn es bei einer speziellen Untersuchung sinnvoller erscheint, mit Zirkuits, Basen oder Rangfunktionen statt mit Unabhängigkeitssystemen zu arbeiten.

Sicherlich induzieren nicht alle Zirkuit- oder Basissysteme oder alle Rangfunktionen Unabhängigkeitssysteme von Matroiden. Diese Antiketten bzw. Funktionen müssen spezielle Eigenschaften haben, damit das zugehörige Unabhängigkeitssystem das Axiom (I.3) erfüllt. Einige solcher Eigenschaften wollen wir kurz auflisten.

(5.10) Satz.

(a) Eine Antikette $\mathcal{C} \subseteq 2^E$, $\mathcal{C} \neq \{\emptyset\}$, ist das **Zirkuitsystem eines Matroids** *auf E genau dann, wenn eine der beiden folgenden äquivalenten Bedingungen erfüllt ist:*

(C.1) $C_1, C_2 \in \mathcal{C}, C_1 \neq C_2, z \in C_1 \cap C_2 \Longrightarrow \exists\, C_3 \in \mathcal{C}$ *mit* $C_3 \subseteq (C_1 \cup C_2) \setminus \{z\}$,

(C.1′) $C_1, C_2 \in \mathcal{C}, C_1 \neq C_2, y \in C_1 \setminus C_2 \Longrightarrow \forall x \in C_1 \cap C_2 \,\exists\, C_3 \in \mathcal{C}$ *mit*
$y \in C_3 \subseteq (C_1 \cup C_2) \setminus \{x\}$.

(b) Eine Antikette $\mathcal{B} \subseteq 2^E$, $\mathcal{B} \neq \emptyset$, ist das **Basissystem eines Matroids** *auf E genau dann, wenn das folgende Axiom erfüllt ist:*

(B.1) $B_1, B_2 \in \mathcal{B}, x \in B_1 \setminus B_2 \Longrightarrow \exists\, y \in B_2 \setminus B_1$ *mit* $(B_1 \cup \{y\}) \setminus \{x\} \in \mathcal{B}$.

(c) Eine Funktion $r : 2^E \to \mathbb{Z}$ ist die **Rangfunktion eines Matroids** *auf E genau dann, wenn eines der beiden folgenden äquivalenten Axiomensysteme erfüllt ist:*

(R.1) $r(\emptyset) = 0$,

(R.2) $F \subseteq E, e \in E \Longrightarrow r(F) \leq r(F \cup \{e\}) \leq r(F) + 1$,

(R.3) $F \subseteq E, f, g \in E$ *mit* $r(F \cup \{g\}) = r(F \cup \{f\}) = r(F) \Longrightarrow r(F \cup \{g, f\}) = r(F)$,

beziehungsweise

(R.1′) $F \subseteq E \Longrightarrow 0 \leq r(F) \leq |F|$, *($r$ ist subkardinal)*

(R.2′) $F \subseteq G \subseteq E \Longrightarrow r(F) \leq r(G)$, *($r$ ist monoton)*

(R.3′) $F, G \subseteq E \Longrightarrow r(F \cup G) + r(F \cap G) \leq r(F) + r(G)$. *($r$ ist submodular)* $\square$

Es gibt noch einige hundert weitere äquivalente Definitionen von Matroiden (die Äquivalenz ist – auch in den obigen Fällen – nicht immer offensichtlich). Wir wollen uns jedoch mit den obigen begnügen.

Ist $\mathcal{I}_1$ ein Matroid auf einer Grundmenge E_1 und $\mathcal{I}_2$ ein Matroid auf E_2, so heißen $\mathcal{I}_1$ und $\mathcal{I}_2$ **isomorph**, falls es eine bijektive Abbildung $\varphi : E_1 \to E_2$ gibt mit

$$\varphi(F) \text{ ist unabhängig in } \mathcal{I}_2 \Longleftrightarrow F \text{ ist unabhängig in } \mathcal{I}_1.$$

Eine Teilmenge $F \subseteq E$ heißt **abgeschlossen** (bezüglich $\mathcal{I}$), falls gilt

$$r(F) < r(F \cup \{e\}) \quad \text{für alle } e \in E \setminus F.$$

Die Matroidtheorie kann man als eine gemeinsame Verallgemeinerung gewisser Aspekte der Graphentheorie und der linearen Algebra ansehen. Die beiden Beispiele, aus denen die Matroidtheorie entstanden ist, wollen wir daher zuerst vorstellen.

(5.11) Beispiele.

(a) **Graphische Matroide**

Das in (5.5) (b) definierte Unabhängigkeitssystem der Wälder eines Graphen $G = (V, E)$ ist ein Matroid. Ist $F \subseteq E$, so zerfällt (V, F) in Zusammenhangskomponenten $G_1 = (V_1, F_1), \ldots, G_k = (V_k, F_k)$. Die Basen der Zusammenhangskomponenten $G_1, \ldots, G_k$ sind die aufspannenden Bäume von $G_1, \ldots, G_k$. Jede Vereinigung von aufspannenden Bäumen von $G_1, \ldots, G_k$ ist eine Basis von F. Die Zirkuits von $\mathcal{I}$ sind die Kreise des Graphen G (daher der Name!). Der Rang in einer Menge $F \subseteq E$ ist gegeben durch

$$r(F) = |V(F)| - \text{Anzahl } k \text{ der Komponenten von } (V(F), F),$$

wobei $V(F)$ die Menge aller Knoten $v \in V$ bezeichnet, die in mindestens einer Kante aus F enthalten sind, vergleiche Folgerung (4.5).

Die Matroide, die wie oben angegeben auf einem Graphen definiert werden können (bzw. isomorph zu solchen sind), heißen **graphische Matroide**.

(b) **Matrix-Matroide**

Sei $A = (a_{ij})$ eine (m, n)-Matrix über einem beliebigen Körper K mit Spaltenvektoren $A_{.1}, \ldots, A_{.n}$. $E = \{1, \ldots, n\}$ sei die Menge der Spaltenindizes von A. Eine Teilmenge $F \subseteq E$ heißt unabhängig, wenn die Vektoren $A_{.j}$, $j \in F$, linear unabhängig in K^m sind. Da jede Teilmenge einer linear unabhängigen Menge wiederum linear unabhängig ist, ist das so definierte Mengensystem $\mathcal{I}$ offenbar ein Unabhängigkeitssystem. Die Menge aller Basen von E ist die Menge aller $B \subseteq E$, so dass die Vektoren $A_{.j}$, $j \in B$, eine Basis des durch die Spaltenvektoren von A aufgespannten linearen Teilraums von K^m bilden. Das Basisaxiom (B.1) ist in diesem Falle aufgrund des Steinitz'schen Austauschsatzes erfüllt. (Dieser Satz war die Motivation für (B.1).) Matroide, die auf die hier angegebene Weise konstruiert werden können, heißen **Matrix-Matroide**. Die Rangfunktion von $\mathcal{I}$ entspricht der aus der linearen Algebra bekannten Rangfunktion des K^m beschränkt auf die Spalten von A. Ist $\mathcal{I}$ ein Matroid auf E und gibt es einen Körper K und eine (m, n)-Matrix A über K, so dass $\mathcal{I}$ zu dem Matrix-Matroid bezüglich A isomorph ist, dann heißt $\mathcal{I}$ **repräsentierbar** (über K). (Natürlich kann man hier etwas verallgemeinern und anstelle von Körpern Schiefkörper oder andere geeignete Objekte betrachten und Repräsentierbarkeit über diesen studieren.) Matroide, die über dem zweielementigen Körper $GF(2)$ repräsentierbar sind, heißen **binär**. Eine umfassende Untersuchung dieser wichtigen Klasse von Matroiden findet sich in Truemper (1992). Matroide, die über allen Körpern repräsentierbar sind, nennt man **regulär**. □

Es folgt nun eine Liste weiterer interessanter Matroide.

(5.12) Beispiele.

(a) **Cographische Matroide**

Gegeben sei ein Graph $G = (V, E)$. Ein **Cokreis** ist eine Kantenmenge, deren Entfernung aus G die Komponentenzahl erhöht und die (mengeninklusionsweise) minimal bezüglich dieser Eigenschaft ist. Ein **Schnitt** ist eine Kantenmenge der Form

$$\delta(W) = \{ij \in E \mid i \in W, j \in V \setminus W\},\ W \subseteq V.$$

Jeder Cokreis ist offenbar ein Schnitt, und es ist einfach einzusehen, dass die Cokreise gerade die minimalen nicht-leeren Schnitte von G sind. Sind $\delta(W)$ und $\delta(W')$ verschiedene Cokreise und ist die Kante ij in beiden enthalten, so ist $\delta(W \triangle W')$ ein Schnitt der ij nicht enthält. (Hier bezeichnet $W \triangle W'$ die **symmetrische Differenz** $(W \cup W') \setminus (W \cap W')$.) Ist $\delta(W \triangle W')$ kein Cokreis, so enthält er einen Cokreis, der natürlich ij auch nicht enthält. Daraus folgt, dass die Antikette der Cokreise das Axiom (C.1) erfüllt und somit das Zirkuitsystem eines Matroids auf E ist. Ein Matroid, das isomorph zu einem so definierten Matroid ist, heißt **cographisch**. Ist $G = (V, E)$ ein zusammenhängender Graph und M das cographische Matroid auf G, so ist das Basissystem $\mathcal{B}$ von M gegeben durch

$$\mathcal{B} = \{B \subseteq E \mid \exists \text{ aufspannender Baum } T \subseteq E \text{ mit } B = E \setminus T\}.$$

(b) **Uniforme Matroide**

Sei E eine Menge mit n Elementen, dann ist die Menge aller Teilmengen von E mit höchstens k Elementen ein Matroid auf E. Dieses Matroid heißt **uniform** und ist durch die Angabe von k und n bis auf Isomorphie eindeutig bestimmt. Dieses Matroid wird mit

$U_{k,n}$ bezeichnet. Das Basissystem von $U_{k,n}$ wird durch die Menge der Teilmengen von E mit genau k Elementen gebildet. Das Zirkuitsystem von $U_{k,n}$ besteht aus den Teilmengen von E mit $k+1$ Elementen. Die Matroide $U_{n,n}$ (d. h. die Matroide, in denen alle Mengen unabhängig sind) heißen **frei** (oder trivial). Für freie Matroide gilt $\mathcal{C} = \emptyset$, $\mathcal{B} = \{E\}$, $r(F) = |F|$ für alle $F \subseteq E$. Das uniforme Matroid $U_{2,4}$ ist das kleinste nicht binäre Matroid.

(c) **Partitionsmatroide**

Sei E eine endliche Menge, und $E_1, \ldots, E_k$ seien nicht-leere Teilmengen von E mit $E_i \cap E_j = \emptyset$, $i \neq j$, und $\bigcup_{i=1}^k E_i = E$. Seien $b_1, \ldots, b_k$ nicht-negative ganze Zahlen, dann ist

$$\mathcal{I} := \{I \subseteq E \mid |I \cap E_i| \leq b_i, i = 1, \ldots, k\}$$

ein Matroid auf E, genannt **Partitionsmatroid**.

(d) **Transversalmatroide**

Sei E eine endliche Menge, $(E_i, i \in I)$ sei eine endliche Familie von Teilmengen von E. Eine Teilmenge $T \subseteq E$ ist eine **teilweise Transversale** (oder partielles Repräsentantensystem) von $(E_i, i \in I)$, falls es eine Indexmenge $J \subseteq I$ gibt mit $|J| = |T|$ und eine Bijektion $\pi : T \to J$, so dass $t \in E_{\pi(t)}$ für alle $t \in T$. Die Menge aller teilweisen Transversalen ist das Unabhängigkeitssystem eines Matroids auf E. (Dieses Ergebnis ist nicht trivial und hatte einen wesentlichen Einfluss auf die Transversaltheorie.) □

Wir wollen nun noch einige konkrete Beispiele von Matroiden angeben und ihre Basis-, Zirkuitsysteme etc. explizit auflisten.

Betrachten wir den folgenden in Abbildung 5.1 dargestellten Graphen $G = (V, E)$ mit $E = \{1, 2, \ldots, 13\}$. Das Zirkuitsystem $\mathcal{C}$ des graphischen Matroids auf E ist gegeben durch die Menge aller Kreise in G, d. h.

$$\mathcal{C} = \{\{1, 2, 3\}, \{4, 5, 6, 7\}, \{9, 10, 11\}, \{11, 12, 13\}, \{9, 10, 12, 13\}\}.$$

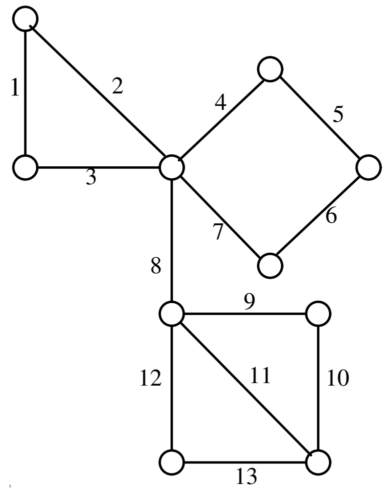


Abb. 5.1

Das Zirkuitsystem $\mathcal{C}^*$ des cographischen Matroids auf E ist gegeben durch die Menge aller minimalen Schnitte, d. h.

$$\mathcal{C}^* = \{\{1,2\},\{1,3\},\{2,3\},\{4,5\},\{4,6\},\{4,7\},\{5,6\},\{5,7\},\{6,7\},$$

$$\{8\},\{9,10\},\{9,11,12\},\{9,11,13\},\{10,11,12\},\{10,11,13\},\{12,13\}\}\,.$$

Das Basissystem $\mathcal{B}$ des graphischen Matroids des folgenden Graphen $G = (V, E)$ (siehe Abbildung 5.2) mit $E = \{1, 2, 3, 4\}$ ist gegeben durch

$$\mathcal{B} = \{\{1,2,3\},\{1,2,4\},\{1,3,4\}\},$$

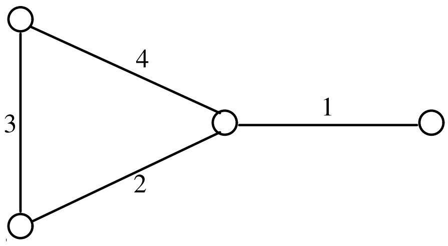


Abb. 5.2

und das Basissystem $\mathcal{B}^*$ des cographischen Matroids bezüglich G ist die folgende Antikette

$$\mathcal{B}^* = \{\{4\},\{3\},\{2\}\}.$$

Das graphische Matroid des in Abbildung 5.3 dargestellten Graphen ist das uniforme Matroid $U_{2,3}$. Uniforme Matroide können also auch isomorph zu graphischen sein. Das uniforme Matroid $U_{2,4}$ ist jedoch nicht graphisch.

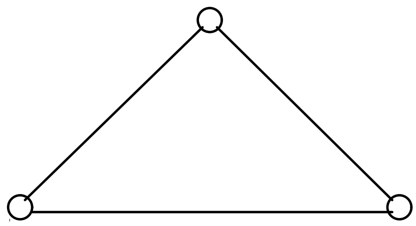


Abb. 5.3

Betrachten wir die Matrix

$$A = \begin{pmatrix} 1 & -1 & 1 & -1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

als Matrix über dem Körper $\mathbb{R}$ oder $\mathbb{Q}$. Das Zirkuitsystem $\mathcal{C}$ des Matrix-Matroids M bezüglich A ist offenbar gegeben durch

$$\mathcal{C} = \{\{1,2,3\},\{2,3,4\},\{1,4\}\},$$

und das Basissystem $\mathcal{B}$ des Matrix-Matroids M ist

$$\mathcal{B} = \{\{1,2\},\{1,3\},\{2,3\},\{2,4\},\{3,4\}\}.$$

Wir wollen nun noch einen einfachen, aber interessanten Zusammenhang zwischen Unabhängigkeitssystemen und Matroiden erwähnen.

(5.13) Satz. *Jedes Unabhängigkeitssystem ist als Durchschnitt von Matroiden darstellbar, d. h. ist $\mathcal{I}$ ein Unabhängigkeitssystem auf E, dann gibt es Matroide $\mathcal{I}_1, \dots, \mathcal{I}_k$ auf E mit*

$$\mathcal{I} = \bigcap_{i=1}^{k} \mathcal{I}_i \,.$$

Beweis : Sei $\mathcal{C}$ das zu $\mathcal{I}$ gehörige Zirkuitsystem. Jedes Zirkuit $C \in \mathcal{C}$ definiert eine Antikette $\{C\} \subseteq 2^E$, die trivialerweise das Axiom (C.1) aus (5.7) erfüllt. Also ist das zu dem Zirkuitsystem $\{C\}$ gehörige Unabhängigkeitssystem $\mathcal{I}_C$ ein Matroid. Wir behaupten nun

$$\mathcal{I} = \bigcap_{C \in \mathcal{C}} \mathcal{I}_C \,.$$

Ist $I \in \mathcal{I}$, so ist kein Zirkuit $C \in \mathcal{C}$ in I enthalten, folglich ist nach (5.4) $I \in \mathcal{I}_C$ für alle $C \in \mathcal{C}$. Sei umgekehrt $I \in \mathcal{I}_C$ für alle $C \in \mathcal{C}$, so heißt dies, dass kein Zirkuit $C \in \mathcal{C}$ in I enthalten ist, und somit, dass I ein Element von $\mathcal{I}$ ist. $\square$

Die im Beweis von Satz (5.13) angegebene Konstruktion zur Darstellung eines Unabhängigkeitssystems als Durchschnitt von Matroiden produziert i. a. eine riesige Zahl von Matroiden, die das Gewünschte leisten. Häufig kommt man mit viel weniger Matroiden aus.

Betrachten wir z. B. die Menge der Branchings $\mathcal{I} \subseteq 2^A$ in einem Digraphen $D = (V, A)$. Man kann einfach zeigen, dass das zugehörige Zirkuitsystem $\mathcal{C}$ aus den inklusionsminimalen Mengen der Vereinigung der folgenden Antiketten $\mathcal{C}_1$ und $\mathcal{C}_2$ besteht:

$$\mathcal{C}_1 := \{C \subseteq A \mid |C| = 2, \text{ u. die Endknoten der beiden Bögen in } C \text{ sind identisch}\},$$

$$\mathcal{C}_2 := \{C \subseteq A \mid C \text{ ist ein Kreis (die Bogenrichtungen spielen keine Rolle)}\}.$$

$\mathcal{C}_1$ ist das Zirkuitsystem eines Partitionsmatroids auf A, dessen Unabhängigkeitssystem gegeben ist durch $\{B \subseteq A \mid |B \cap \delta^-(v)| \leq 1 \;\forall\, v \in V\}$, und $\mathcal{C}_2$ ist das Zirkuitsystem des graphischen Matroids auf D (hierbei wird D als Graph aufgefasst, d. h. die Bogenrichtungen werden ignoriert). Daraus folgt, dass das Unabhängigkeitssystem der Branchings Durchschnitt von 2 Matroiden ist. Für den vollständigen Digraphen D mit n Knoten hätte die Konstruktion im Beweis von Satz (5.13) insgesamt

$$n\binom{n-1}{2} + \sum_{k=2}^{n} \binom{n}{k}(k-1)!$$

verschiedene Matroide geliefert.

Das Unabhängigkeitssystem $\tilde{\mathcal{T}}$ der Teilmengen von Touren im vollständigen Digraphen D_n mit n Knoten, siehe (5.8) (d), ist als Durchschnitt von 3 Matroiden darstellbar (Hausaufgabe). Also ist das asymmetrische Travelling-Salesman-Problem als Optimierungsproblem über dem Durchschnitt von 3 Matroiden formulierbar.

5.3 Orakel

Um Eigenschaften von Matroiden oder Unabhängigkeitssystemen überprüfen zu können, muss man sich natürlich fragen, wie man Matroide geeignet darstellt, um z. B. ein Computerprogramm schreiben zu können, das Matroide als Input akzeptiert.

Betrachten wir zum Beispiel das in (5.6) formulierte Optimierungsproblem $\max\{c(I) | I \in \mathcal{I}\}$ über einem Unabhängigkeitssystem $\mathcal{I}$ auf E (Spezialfall: $(E, \mathcal{I})$ ist ein Matroid). Ist $\mathcal{I}$ als Liste aller unabhängigen Mengen gegeben, so ist das Optimierungsproblem völlig trivial. Wir durchlaufen die Liste, rechnen für jede Menge I der Liste den Wert $c(I)$ aus und wählen eine Menge I^*, so dass $c(I^*)$ maximal ist. Die Laufzeit dieses trivialen Algorithmus ist linear in der Inputlänge des Unabhängigkeitssystems.

Viele der Matroide und Unabhängigkeitssysteme, die wir in den voraufgegangenen Abschnitten definiert haben, sind jedoch in wesentlich kompakterer Form gegeben. Zum Beispiel ist ein Matrix-Matroid (5.11) (b) durch eine Matrix A (mit der Information, dass $I \subseteq E$ unabhängig genau dann ist, wenn die Spalten $A_{.i}$, $i \in I$, linear unabhängig sind) gegeben, ein graphisches Matroid (5.11) (a) durch einen Graphen $G = (V, E)$ (zusammen mit der Information, dass $I \subseteq E$ unabhängig ist genau dann, wenn I keinen Kreis enthält), ein cographisches Matroid (5.12) (a) durch einen Graphen $G = (V, E)$ (zusammen mit der Information, dass $C \subseteq E$ ein Zirkuit ist genau dann, wenn C ein Cokreis ist). Würden wir die Inputlänge des Matroids als die Länge der Kodierung der Matrix A (für (5.11) (b)) bzw. als die Länge der Kodierung des Graphen G (für (5.11) (a)) definieren, hätte unser trivialer Enumerationsalgorithmus exponentielle Laufzeit in der Kodierungslänge dieses kompakten Inputs.

Die Frage ist also: Was ist eine "geeignete" Kodierung eines Matroids? Motiviert durch die gerade angegebenen Beispiele könnte man glauben, dass die Angabe einer Liste aller unabhängigen Mengen eine unökonomische Methode sei. Jedoch geht es im allgemeinen nicht viel besser. Man kann nämlich zeigen, dass es mindestens

$$2^{2^{n/2}} \text{ Matroide mit } n \text{ Elementen}$$

gibt. Daraus folgt, dass es bei jeder beliebigen Kodierungsart immer Matroide gibt, deren Kodierung eine Länge hat, die mindestens $2^{n/2}$ beträgt.

Aufgrund dieser Tatsache hat sich ein anderes Konzept der Repräsentation von Matroiden durchgesetzt und als außerordentlich nützlich erwiesen. Matroide werden durch "Orakel" dargestellt.

Wir treffen folgende Definition. Die **Kodierungslänge eines Matroids** $M = (E, \mathcal{I})$ ist die Anzahl der Elemente von E. Das Matroid selbst ist in einem Orakel "versteckt", wobei das Orakel als ein Computerprogramm (Subroutine) interpretiert werden kann, das Fragen eines speziellen Typs beantwortet. Wir geben einige Beispiele (diese gelten natürlich nicht nur für Matroide, sondern analog auch für Unabhängigkeitssysteme) und gehen davon aus, dass wir die Grundmenge E kennen.

(5.14) Beispiele für Orakel.

(a) **Unabhängigkeitsorakel**

Für jede Menge $F \subseteq E$ können wir das Orakel fragen, ob F unabhängig ist. Das Orakel antwortet mit "ja" oder "nein".

(b) **Zirkuitorakel**

Für jede Menge $F \subseteq E$ können wir das Orakel fragen, ob F ein Zirkuit ist. Das Orakel antwortet mit "ja" oder "nein".

(c) **Basisorakel**

Für jede Menge $F \subseteq E$ können wir das Orakel fragen, ob F eine Basis von E ist. Das Orakel antwortet mit "ja" oder "nein".

(d) **Rangorakel**

Für jede Menge $F \subseteq E$ können wir das Orakel fragen, wie groß der Rang von F ist. Die Antwort des Orakels ist "$r(F)$". □

Man sieht sofort, dass dieses theoretische Orakelkonzept dem Konzept von Unterprogrammen der Programmierung entspricht.

Wenn wir also Algorithmen betrachten wollen, die Eigenschaften von Matroiden überprüfen, so gehen wir immer davon aus, dass ein Matroid (oder Unabhängigkeitssystem) durch die Grundmenge E und ein Orakel gegeben ist, das im Verlaufe des Algorithmus befragt werden kann. Bei der Komplexitätsanalyse von Algorithmen für Matroide (Unabhängigkeitssysteme) wird dann ein Orakelaufruf (Unterprogrammaufruf) als ein Schritt gezählt. Achtung! Die Laufzeit beim Lesen der Antwort wird wie üblich berechnet. Gibt also ein Orakel eine Antwort, die z. B. $2^{|E|}$ Speicherplätze benötigt, so hat der Algorithmus automatisch eine exponentielle Laufzeit. Bei den in (5.14) angegebenen Orakeln kommt so ein Fall allerdings nie vor! Man nennt Verfahren, die Orakel aufrufen können, **Orakelalgorithmen**. Ist ihre Laufzeit in dem oben angegebenen Sinne polynomial in $|E|$, so sagt man, dass die Verfahren **orakelpolynomial** sind.

Hat man einen Algorithmus, dessen Laufzeit orakelpolynomial in $|E|$ ist, so ist der Algorithmus für alle die Matroide (Unabhängigkeitssysteme) im üblichen Sinne polynomial, für die das Orakel durch einen in $|E|$ polynomialen Algorithmus realisiert werden kann.

Dies ist zum Beispiel für Matrix-Matroide der Fall. Ein Unabhängigkeitsorakel kann man bei einer gegebenen Matrix durch Rangbestimmung mit Gauß-Elimination realisieren (analog die drei anderen Orakel). Dies gilt auch für graphische und cographische Matroide, die durch einen Graphen $G = (V, E)$ gegeben sind. Zum Beispiel kann man die Unabhängigkeit einer Menge F im graphischen Matroid (F enthält keinen Kreis) durch Depth-First-Search testen; auch die übrigen drei Orakel kann man durch Algorithmen realisieren deren Laufzeit polynomial in $|E|$ ist.

Eine wichtige Frage ergibt sich sofort. Sind die verschiedenen Orakel algorithmisch "äquivalent"? Das heißt, kann man ein Orakel durch ein anderes Orakel in orakelpolynomialer Zeit simulieren und umgekehrt? Zum Beispiel: Ist ein Unabhängigkeitsorakel gegeben, kann man dann einen Algorithmus entwerfen (der nur dieses Orakel benutzt und orakelpolynomial ist), der für jede Menge $F \subseteq E$ korrekt entscheidet, ob F ein Zirkuit ist oder nicht?

Für Matroide und die vier oben angegebenen Orakel wurde diese Frage von Hausmann and Korte (1981) wie folgt beantwortet.

(5.15) Satz. *Sei $M = (E, \mathcal{I})$ ein beliebiges Matroid.*

(a) Das Unabhängigkeitsorakel und das Rangorakel sind bezüglich M "äquivalent".

(b) *Das Basisorakel und das Zirkuitorakel können durch das Unabhängigkeitsorakel sowie durch das Rangorakel in orakelpolynomialer Zeit simuliert werden.*

(c) *Es gibt keine anderen orakelpolynomialen Beziehungen zwischen diesen Orakeln.*

□

Man kann diesen Satz graphisch durch Abbildung 5.4 veranschaulichen. Ein Pfeil in diesem Diagramm besagt, dass das Orakel am Ende des Pfeils durch polynomial viele Aufrufe des Orakels an der Spitze des Pfeils simuliert werden kann. Ist zwischen zwei Kästchen kein Pfeil (in irgendeiner der beiden Richtungen), so besagt dies auch, dass es keine orakelpolynomiale Simulation dieser Art gibt. Z. B. kann man das Basisorakel nicht durch das Zirkuitorakel in orakelpolynomialer Zeit simulieren und umgekehrt. Dies zeigt man dadurch, dass man eine Klasse von Beispielen angibt, bei denen zur Entscheidung der Frage, ob eine Menge I eine Basis (ein Zirkuit) ist, eine Anzahl von Aufrufen des Zirkuitorakels (Basisorakels) notwendig ist, die exponentiell in $|E|$ ist.

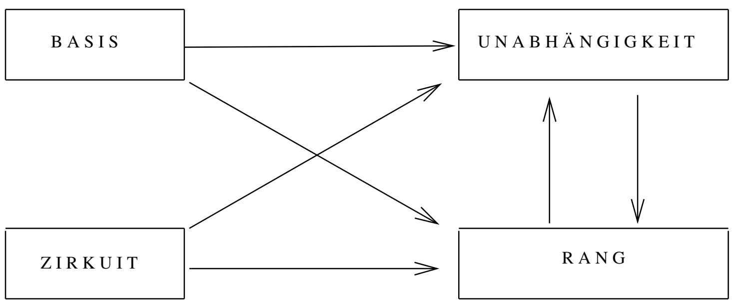


Abb. 5.4

Hausmann and Korte (1980) haben dieselbe Fragestellung auch für allgemeine Unabhängigkeitssysteme untersucht, die – wie wir in Abschnitt 5.1 gesehen haben – äquivalent durch Zirkuitsysteme, Basissysteme oder Rangfunktionen gegeben werden können. Der entsprechende Satz ist bildlich in Abbildung 5.5 veranschaulicht (Interpretation wie bei Abbildung 5.4).

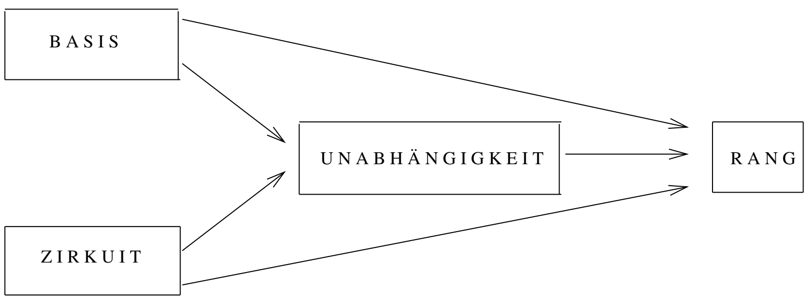


Abb. 5.5

Insbesondere folgt, dass bei Unabhängigkeitssystemen das Rangorakel das stärkste ist. Durch dieses lassen sich die übrigen Orakel in orakelpolynomialer Zeit simulieren. Jedoch sind hier keine zwei Orakel algorithmisch "äquivalent" (sie sind natürlich logisch äquivalent).

5.4 Optimierung über Unabhängigkeitssystemen

Wir wollen nun die Frage untersuchen, ob bzw. wie gut ein Optimierungsproblem der Form (5.6)

$$\max\{c(I) \mid I \in \mathcal{I}\},$$

wobei $\mathcal{I}$ ein Unabhängigkeitssystem auf einer Grundmenge E ist, gelöst werden kann. Wir betrachten dazu den folgenden trivialen Algorithmus, vergleiche (4.1).

(5.16) Greedy-Algorithmus für Unabhängigkeitssysteme.

Input: *Grundmenge $E = \{1, \dots, n\}$ mit Gewichten $c_i \in \mathbb{R}$ für alle $i \in E$. Ferner ist ein Unabhängigkeitssystem $\mathcal{I} \subseteq 2^E$ durch ein Unabhängigkeitsorakel (siehe (5.14) (a)) gegeben.*

Output: *Eine unabhängige Menge $I_g \in \mathcal{I}$.*

1. *Sortiere die Gewichte in nicht aufsteigender Reihenfolge (d. h. nach Beendigung von Schritt 1 können wir annehmen, dass $c_1 \geq c_2 \geq \ldots \geq c_n$ gilt).*
2. *Setze $I := \emptyset$.*
3. FOR $i = 1$ TO n DO:

   Ist $c_i \leq 0$, gehe zu 4.

   Ist $I \cup \{i\}$ unabhängig (Orakelaufruf), dann setze $I := I \cup \{i\}$.
4. *Setze $I_g := I$ und gib I_g aus.* □

Der Greedy-Algorithmus durchläuft (nach der Sortierung in Schritt 1) die Elemente von E genau einmal, entweder er nimmt ein Element in die Menge I auf, oder er verwirft es für immer. Die am Ende des Verfahrens gefundene Lösung I_g nennen wir **Greedy-Lösung**.

Für eine Menge $F \subseteq E$ setzen wir

(5.17) $$r_u(F) := \min\{|B| \mid B \text{ Basis von } F\}$$

und nennen $r_u(F)$ den **unteren Rang von** F. Wir setzen

$$q = \min_{F \subseteq E, r(F) > 0} \frac{r_u(F)}{r(F)}$$

und nennen q den Rangquotient von $\mathcal{I}$. Ist $(E, \mathcal{I})$ ein Matroid, so gilt nach (I.3″) natürlich $r_u(F) = r(F)$ für alle $F \subseteq E$ und somit $q = 1$.. Das folgende Resultat wurde von Jenkyns (1976) bewiesen.

(5.18) Satz. *Sei $\mathcal{I}$ ein Unabhängigkeitssystem auf E, seien I_g eine Greedy-Lösung von (5.16) und I_0 eine optimale Lösung von (5.6), dann gilt*

$$q \leq \frac{c(I_g)}{c(I_0)} \leq 1,$$

und für jedes Unabhängigkeitssystem gibt es Gewichte $c_i \in \{0, 1\}$, $i \in E$, so dass die erste Ungleichung mit Gleichheit angenommen wird.

Beweis : Wir können o. B. d. A. annehmen, dass $c_i > 0$ für $i = 1, \ldots, n$ und dass $c_1 \geq c_2 \geq \ldots \geq c_n$ gilt. Aus technischen Gründen führen wir ein Element $n+1$ ein mit $c_{n+1} = 0$. Wir setzen

$$E_i := \{1, \ldots, i\}, i = 1, \ldots, n.$$

Es gilt dann offenbar:

$$\begin{aligned} &(1) \qquad c(I_g) = \textstyle\sum_{i=1}^n |I_g \cap E_i|(c_i - c_{i+1}), \\ &(2) \qquad c(I_0) = \textstyle\sum_{i=1}^n |I_0 \cap E_i|(c_i - c_{i+1}). \end{aligned}$$

Da $I_0 \cap E_i \subseteq I_0$, gilt $I_0 \cap E_i \in \mathcal{I}$, und somit $|I_0 \cap E_i| \leq r(E_i)$. Die Vorgehensweise des Greedy-Algorithmus impliziert, dass $I_g \cap E_i$ eine Basis von E_i ist, also dass $|I_g \cap E_i| \geq r_u(E_i)$ gilt. Daraus folgt

$$|I_g \cap E_i| \geq |I_0 \cap E_i| \frac{r_u(E_i)}{r(E_i)} \geq |I_0 \cap E_i|\, q, \quad i = 1, \ldots, n$$

und somit

$$\begin{aligned} c(I_g) &= \textstyle\sum_{i=1}^n |I_g \cap E_i|(c_i - c_{i+1}) \\ &\geq \textstyle\sum_{i=1}^n (|I_0 \cap E_i|\, q)(c_i - c_{i+1}) \\ &= q \textstyle\sum_{i=1}^n |I_0 \cap E_i|(c_i - c_{i+1}) \\ &= c(I_0) q. \end{aligned}$$

Die Ungleichung $1 \geq \frac{c(I_g)}{c(I_0)}$ ist trivial. Damit haben wir die Gültigkeit der Ungleichungskette bewiesen.

Sei nun $F \subseteq E$ mit $q = \frac{r_u(F)}{r(F)}$. Wir können annehmen, dass $F = \{1, \ldots, k\}$ gilt und dass $B = \{1, \ldots, p\} \subseteq F$ eine Basis von F ist mit $|B| = r_u(F)$. Wir setzen $c_i = 1$,

$i = 1, \dots, k$, und $c_i = 0$, $i = k+1, \dots, n$. Dann liefert der Greedy-Algorithmus die Greedy-Lösung $I_g = B$ mit $c(I_g) = r_u(F)$, während für jede Optimallösung I_0 gilt $c(I_0) = r(F)$. Also wird für diese spezielle $0/1$-Zielfunktion die linke Ungleichung in der Aussage des Satzes mit Gleichheit angenommen. $\square$

(5.19) Folgerung. *Sei $\mathcal{I}$ ein Unabhängigkeitssystem auf E. Dann sind äquivalent:*

(a) $(E, \mathcal{I})$ *ist ein Matroid.*

(b) Für alle Gewichte $c \in \mathbb{R}^E$ liefert der Greedy-Algorithmus (5.16) eine Optimallösung von (5.6).

(c) Für alle Gewichte mit $0/1$-Koeffizienten liefert der Greedy-Algorithmus (5.16) eine Optimallösung von (5.6). $\square$

(5.19) wurde von verschiedenen Autoren unabhängig voneinander bewiesen. Edmonds (1971) zeigte insbesondere, wie man den Greedy-Algorithmus für Matroide als ein Verfahren zur Lösung spezieller linearer Programme deuten kann. Er liefert in der Tat auch duale Lösungen. Wir können hier jedoch nicht auf diese Zusammenhänge eingehen.

Da der Algorithmus (4.7) GREEDY-MAX offenbar eine Spezialisierung des Greedy-Algorithmus (5.16) für das graphische Matroid ist, liefert Folgerung (5.19) einen weiteren Beweis von Satz (4.8).

Satz (5.18) ist ein Prototyp von Abschätzungssätzen für die Qualität von heuristischen Lösungen, wie wir sie später noch mehrfach kennenlernen werden. Der Greedy-Algorithmus (5.16) ist offenbar orakelpolynomial. Immer dann, wenn wir den Orakelaufruf (Unabhängigkeitstest) in Schritt 3 durch einen polynomialen Algorithmus realisieren können, ist der Greedy-Algorithmus ein polynomialer Algorithmus (im üblichen Sinne). Eine solche polynomiale Realisation ist z. B. auf triviale Weise für das Cliquenproblem, Stabile-Menge-Problem, Travelling-Salesman-Problem und azyklische Subdigraphenproblem möglich. Würde der Greedy-Algorithmus auch in diesen Fällen immer Optimallösungen liefern, hätten wir einen Beweis für $\mathcal{P}$=$\mathcal{NP}$ gefunden, da alle gerade aufgelisteten Probleme $\mathcal{NP}$-schwer sind. Wir können also nicht erwarten, dass der Greedy-Algorithmus immer optimale Antworten produziert. Die Frage, wie schlecht im schlechtest möglichen Falle eine Greedy-Lösung ist, beantwortet Satz (5.18) auf bestmögliche Weise. Er gibt eine sogenannte **Gütegarantie** an, die besagt, dass der Wert $c(Ig)$ jeder Greedy-Lösung nicht schlechter ist als $qc(Io)$. Der Quotient

$$q := \min_{F \subseteq E} \frac{r_u(F)}{r(F)},$$

heißt in der Literatur auch **Rangquotient**. Der Rangquotient liefert als eine Gütegarantie für die Lösungsqualität des Greedy-Algorithmus. Der Rangquotient ist im allgemeinen nicht einfach auszurechnen. Eine Abschätzung wird gegeben durch:

(5.20) Satz. *Sei $\mathcal{I}$ ein Unabhängigkeitssystem auf E, und $(E, \mathcal{I}_i)$ $i = 1, \dots, k$ sei eine minimale Zahl von Matroiden mit $\mathcal{I} = \bigcap_{i=1}^k \mathcal{I}_i$, dann gilt*

$$\min_{F \subseteq E} \frac{r_u(F)}{r(F)} \geq \frac{1}{k}.$$

□

Daraus folgt zum Beispiel: Ist $\mathcal{I}$ ein Unabhängigkeitssystem, das Durchschnitt von 2 Matroiden ist, dann ist der Wert der Greedy-Lösung mindestens halb so groß wie der Wert der Optimallösung von (5.6). Insbesondere liefert also der Greedy-Algorithmus für Branchingprobleme Lösungen, deren Wert mindestens die Hälfte des Optimums beträgt.

Für das Branching-Problem haben wir in Kapitel 4 einen polynomialen Lösungsalgorithmus kennengelernt. Es gibt also offenbar auch Probleme über Unabhängigkeitssystemen, die keine Matroide sind und für die effiziente Optimierungsverfahren existieren. Ein sehr tiefliegendes Resultat ist der folgende von Edmonds (1979) und Lawler (1975) gefundene Satz.

(5.21) Satz. *Seien $(E, \mathcal{I}_1)$ und $(E, \mathcal{I}_2)$ zwei Matroide gegeben durch Unabhängigkeitsorakel, dann gibt es einen Algorithmus, der für jede beliebige Zielfunktion c das Problem $\max\{c(I) \mid I \in \mathcal{I}_1 \cap \mathcal{I}_2\}$ in orakelpolynomialer Zeit löst.* □

Der Algorithmus, der den Beweis des obigen Satzes liefert, ist relativ kompliziert, und sein Korrektheitsbeweis benötigt Hilfsmittel aus der Matroidtheorie, die uns hier nicht zur Verfügung stehen. Deshalb verzichten wir auf eine Angabe und Analyse dieses (sehr interessanten) Verfahrens.

Man fragt sich nun natürlich sofort, ob auch über dem Durchschnitt von 3 Matroiden in (orakel-) polynomialer Zeit optimiert werden kann. Dies geht (vermutlich) i. a. nicht. Man kann zeigen, dass das Optimierungsproblem für Unabhängigkeitssysteme, die Durchschnitte von 3 Matroiden sind, Probleme enthält, die $\mathcal{NP}$-schwer sind (z. B. das asymmetrische TSP).

Wir wollen uns nun noch kurz dem Optimierungsproblem (5.7) über Basissystemen zuwenden. Wie bei Bäumen und Wäldern vorgeführt (siehe Abschnitt 4.2), kann man ein Maximierungsproblem des Typs (5.6) mit Hilfe einer polynomialen Transformation in ein Minimierungsproblem (5.7) überführen und umgekehrt. Analog lässt sich auch aus (5.16) ein Minimierungsalgorithmus ableiten.

(5.22) Greedy-Minimierungsalgorithmus für Basissysteme.

Input: *Grundmenge $E = \{1, \dots, n\}$ mit Gewichten $c_i \in \mathbb{R}$ für alle $i \in E$, ein Unabhängigkeitssystem $\mathcal{I} \subseteq 2^E$ gegeben durch ein Unabhängigkeitsorakel.*

Output: *Eine Basis $B_g \in \mathcal{I}$.*

1. *Sortiere die Gewichte in nicht absteigender Reihenfolge (nach Beendigung von Schritt 1 gelte $c_1 \leq c_2 \leq \ldots \leq c_n$).*

2. *Setze $I := \emptyset$.*

3. FOR $i = 1$ TO n DO*:*

   Ist $I \cup \{i\} \in \mathcal{I}$ (Orakelaufruf), dann setze $I := I \cup \{i\}$.

4. *Setze $B_g := I$ und gib B_g aus.*

□

Offenbar ist B_g eine Basis der Grundmenge E des Unabhängigkeitssystems $\mathcal{I}$, gehört also zum zugehörigen Basissystem $\mathcal{B}$. Folgerung (5.19) impliziert:

(5.23) Satz. *Sei $\mathcal{I}$ ein Unabhängigkeitssystem auf E und $\mathcal{B}$ das zugehörige Basissystem. Dann sind äquivalent:*

(a) *$(E, \mathcal{I})$ ist ein Matroid.*

(b) *Für alle Gewichte $c \in \mathbb{R}^E$ liefert der Greedy-Algorithmus (5.22) eine Optimallösung von $\min\{c(B) \mid B \in \mathcal{B}\}$.*

(c) *Für alle Gewichte $c \in \{0,1\}^E$ liefert der Greedy-Algorithmus (5.22) eine Optimallösung von $\min\{c(B) \mid B \in \mathcal{B}\}$.*

□

Sortiert man in Schritt 1 von (5.22) die Gewichte in nicht aufsteigender Reihenfolge, so erhält man offenbar eine Basis maximalen Gewichts. Satz (5.23) gilt analog, wenn man "min" durch "max" ersetzt.

Leider kann man die schöne Abschätzung aus Satz (5.18) für die Gütegarantie von (5.16) nicht ohne weiteres auf Algorithmus (5.22) übertragen. In der Tat gibt es keine universelle Konstante, die bezüglich eines Problems die Qualität des Ergebnisses von (5.22) unabhängig von den Zielfunktionskoeffizienten beschränkt. Betrachten wir z. B. das Cliquen-Problem auf dem in Abbildung 5.6 dargestellten Graphen

Abb. 5.6

mit der Gewichtsfunktion $c_1 = 1$, $c_2 = M > 2$, $c_3 = 2$. Das zugehörige Unabhängigkeitssystem hat die Basen $\{1,2\}$ und $\{3\}$. Bei Anwendung von (5.22) erhalten wir immer die Greedylösung $B_g = \{1,2\}$ mit Gewicht $1 + M$, während die Optimalbasis B_0 das Gewicht 2 hat. Der Quotient aus B_g und B_0 geht also gegen ∞ falls $M \to \infty$.

5.5 Ein primal-dualer Greedy-Algorithmus

Die Matroidtheorie ist aufgrund ihres technischen Apparates besonders gut dazu geeignet, eine gemeinsame Verallgemeinerung des primalen und des dualen Greedy-Algorithmus zur Berechnung minimaler aufspannender Bäume (siehe (4.9) und (4.11)) zu formulieren. Grundlegend hierfür ist der folgende

(5.24) Satz und Definition. *Sei M ein Matroid auf einer Grundmenge E mit Basissystem $\mathcal{B}$. Setze $\mathcal{B}^* := \{E \setminus B \mid B \in \mathcal{B}\}$. Dann ist $\mathcal{B}^*$ das Basissystem eines Matroids M^* auf E. M^* wird das zu M* **duale Matroid** *genannt.* □

Offensichlich gilt für ein Basissystem

$$\mathcal{B}^{**} = \mathcal{B},$$

d. h. das zu M^* duale Matroid ist das Matroid M. Wir haben bereits ein Matroid und das dazu duale Matroid kennengelernt. Ist nämlich M das graphische Matroid auf einem Graphen G (siehe (5.11)(a)), so ist das cographische Matroid auf G (siehe (5.12)(a)) das zu M duale Matroid M^*.

Für Matrix-Matroide (siehe (5.11)(b)) kann man auf einfache Weise das duale Matroid konstruieren. Wir nehmen an, dass M durch die $m \times n$-Matrix A (über irgendeinem Körper) definiert ist. O.B.d.A. können wir annehmen, dass A vollen Zeilenrang m hat. Mit Hilfe der Gauß-Elimination bringen wir A in **Standardform**, d. h. wir formen A so um, dass $A = (I, B)$ gilt, wobei I die $m \times m$-Einheitsmatrix ist. (Möglichweise sind hierzu Zeilen- und Spaltenverbindungen erforderlich.) Man kann nun zeigen, dass das zu M duale Matroid M^* durch die Matrix

$$(-B^T, I)$$

definiert ist. Hierbei ist I eine $(n-m, n-m)$-Einheitsmatrix.

In der Matroidtheorie hat sich die folgende Sprechweise eingebürgert. Ist M ein Matroid und A eine Basis oder ein Zirkuit des dualen Matroids M^*, so nennt man A auch eine **Cobasis** oder ein **Cozirkuit** von M. Analog heißt der Rang von A bezüglich der Rangfunktion von M^* auch der **Corang** von A. Ein **Zyklus** ist die Vereinigung von paarweise disjunkten Zirkuits, analog ist ein **Cozyklus** die Vereinigung paarweise disjunkter Cozirkuits. Die nachfolgenden (einfachen) Beobachtungen sind für die anschließenden Betrachtungen wichtig.

(5.25) Satz.

(a) *Ein System $\mathcal{C}$ von Teilmengen von E ist genau dann die Zirkuitmenge eines Matroids M auf E, wenn $\mathcal{C}$ genau die minimalen, nichtleeren Teilmengen $C \subseteq E$ enthält, so dass gilt: $|C \cap C^*| \neq 1$ für alle Cozirkuits C^* von M.*

(b) *Sei B eine Basis (Cobasis) eines Matroids M, und sei $e \in E \setminus B$, dann enthält $B \cup \{e\}$ genau ein Zirkuit (Cozirkuit) C_e, der e enthält. C_e heißt der durch e und B erzeugte* **fundamentale Zirkuit (Cozirkuit)**. □

(5.26) Satz. *Sei M ein Matroid auf E mit Gewichten c_e für alle $e \in E$. Sei B eine Basis von M. Dann sind die folgenden Aussagen äquivalent:*

(i) *B ist eine gewichtsminimale Basis.*

(ii) *Für jedes Element $e \in E \setminus B$ gilt*

$c_e \geq c_f$ *für alle f aus dem von e und B erzeugten fundamentalen Zirkuit K_e.*

(iii) *Für jeden Cozirkuit C gilt*

$$\min\{c_e \mid e \in C\} = \min\{c_e \mid e \in B \cap C\}$$

(iv) *$E \setminus B$ ist eine gewichtsmaximale Cobasis.*

(v) *Für jedes Element $e \in B$ gilt*

$c_e \leq c_f$ *für alle f aus dem von e und $E \setminus B$ erzeugten fundamentalen Cozirkuit C_e.*

(vi) *Für jeden Zirkuit K gilt*

$$\max\{c_e \mid e \in K\} = \max\{c_e \mid e \in (E \setminus B) \cap K\}.$$

Beweis :

(i) $\Leftrightarrow$ (iv). Trivial.

(i) $\Rightarrow$ (ii). Angenommen, es existieren $e \in E \setminus B$ und $f \in K_e$ mit $c_e < c_f$. Dann ist $(B \setminus \{f\}) \cup \{e\}$ eine Basis mit kleinerem Gewicht als B. Widerspruch!

(ii) $\Rightarrow$ (iii). Angenommen, es existieren ein Cozirkuit C und $e \in C \setminus B$ mit $c_e < \min\{c_f \mid f \in B \cap C\}$. Sei K_e der durch e mit B erzeugte fundamentale Zirkuit. Dann existiert nach (5.25)(a) ein Element $f \in (K_e \setminus \{e\}) \cap C$. Aber dann gilt für $e \in E \setminus B$ und $f \in B : c_e < c_f$. Widerspruch!

(iii) $\Rightarrow$ (i). Sei B' eine gewichtsminimale Basis, so dass $|B \cap B'|$ so groß wie möglich ist. Angenommen, es existiert ein Element, $f \in B' \setminus B$. Sei C_f der von f mit $E \setminus B'$ erzeugte fundamentale Cozirkuit. Da B eine Basis ist, gilt $B \cap C_f \neq \emptyset$. Sei $g \in B \cap C_f$ mit $c_g = \min\{c_e \mid e \in B \cap C_f\}$. Nach (iii) gilt wegen $f \in C_f : c_f \geq c_g$. Nun aber ist $B'' := (B' \setminus \{f\}) \cup \{g\}$ eine Basis von M mit $c(B'') \leq c(B')$ und $|B \cap B''| > |B \cap B'|$. Widerspruch! Daraus folgt $B = B'$, und somit ist B gewichtsminimal.

(iv) $\Rightarrow$ (v) $\Rightarrow$ (vi) $\Rightarrow$ (iv) folgt aus Dualitätsgründen. $\square$

(5.27) Primal-dualer Greedy-Algorithmus.

Input: *Grundmenge E mit Gewichten c_e für alle $e \in E$ und ein Matroid $M = (E,\mathcal{I})$ durch ein Unabhängigkeitsorakel.*

Output: *Eine gewichtsminimale Basis B von M und eine gewichtsmaximale Cobasis B^* von M.*

(Terminologie: Wir nennen eine Menge S durch eine Menge F **überdeckt**, falls $F \cap S \neq \emptyset$.)

1. *Setze $B := B^* := \emptyset$.*
2. *Führe einen (beliebigen) der beiden folgenden Tests aus:*
   - (a) *Sind alle Cozirkuits von M durch B überdeckt, STOP.*
     (Gib B und $B^* := E \setminus B$ aus.)
   - (b) *Sind alle Zirkuits von M durch B^* überdeckt, STOP.*
     (Gib B^* und $B := E \setminus B^*$ aus.)
3. *Führe eine (beliebige) der folgenden beiden Operationen (a), (b) aus:*

(a) (i) *Wähle einen nicht-leeren Cozyklus C von M, der nicht durch B überdeckt ist.*
- (ii) *Bestimme $f \in C$ mit $c_f = \min\{c_e \mid e \in C\}$.*
- (iii) *Setze $B := B \cup \{f\}$ und gehe zu 2.*

(b) (i) *Wähle einen nicht-leeren Zyklus Z von M, der nicht durch B^* überdeckt ist.*
- (ii) *Bestimme $g \in Z$ mit $c_g = \max\{c_e \mid e \in Z\}$.*
- (iii) *Setze $B^* := B^* \cup \{g\}$ und gehe zu 2.*

(5.28) Satz. *Der obige Algorithmus (5.27) funktioniert.*

Beweis : Wir beweisen durch Induktion nach $|B| + |B^*|$, dass die jeweils während des Ablaufs des Verfahrens konstruierten Mengen B und B^* in einer gewichtsminimalen Basis bzw. gewichtsmaximalen Cobasis enthalten sind.

Für den Induktionsanfang $B = B^* = \emptyset$ ist die Behauptung trivial.

Wir nehmen an, dass der Satz gilt, wenn $|B| + |B^*| \leq k$, d.h., wenn wir Schritt 2 und 3 höchstens k-mal ausgeführt haben. Wir beginnen jetzt mit der $(k+1)$-ten Ausführung von

Schritt 3 und führen (a) aus. Sei B die bisher konstruierte Menge und B' eine gewichtsminimale Basis mit $B \subseteq B'$. Wir wählen in (i) einen Cozyklus C und in (ii) ein Element $f \in C$. Gilt $f \in B'$, so ist alles klar. Anderenfalls sei $g \in C \cap B'$ ein Element, das mit f auf einem Cozirkuit liegt. Dann muss wegen (ii) gelten $c_g \geq c_f$. $B'' := (B' \setminus \{g\}) \cup \{f\}$ ist damit eine Basis mit $c(B'') \leq c(B')$ und $\{f\} \cup B \subseteq B''$. Also ist $B \cup \{f\}$ in einer gewichtsminimalen Basis enthalten.

Führen wir (b) aus, so folgt die Behauptung analog.

Stellen wir in 2 (a) fest, dass B eine Basis ist, so ist nach Induktion $c(B)$ minimal und $E \setminus B$ eine maximale Cobasis. Analog schließen wir im Falle, dass B^* in 2 (b) als Cobasis erkannt wird. $\square$

Algorithmus (5.27) ist aufgrund der vielen Freiheitsgrade ein äußerst flexibel einsetzbares Instrument zur Konstruktion optimaler Basen und Cobasen. Durch geeignete Spezialisierung erhält man alle in Kapitel 4 vorgestellten Algorithmen zur Bestimmung minimaler Bäume und einige weitere derartige Verfahren.

Literaturverzeichnis

Barahona, F. and Grötschel, M. (1986). On the cycle polytope of a binary matroid. *Journal of Combinatorial Theory, Series B*, 40:40–62.

Bixby, R. E. and Cunningham, W. H. (1995). *Matroid Optimization and Algorithms*, volume 1, chapter 11, pages 551–609. North-Holland, Amsterdam.

Edmonds, J. (1971). Matroids and the greedy algorithm. *Mathematical Programming*, 1:127–136.

Edmonds, J. (1979). Matroid intersection. *Annals of Discrete Mathematics*, 4:39–49.

Grötschel, M. and Truemper, K. (1989). Decomposition and Optimization over Cycles in Binary Matroids. *Journal of Combinatorial Theory, Series B*, 46(3):306–337.

Hausmann, D. and Korte, B. (1980). The relative strength of oracles for independence systems. In Frehse, J., Pallaschke, D., and Trottenberg, U., editors, *Special topics of applied mathematics, functional analysis, numerical analysis, and optimization*, pages 195–211. North-Holland, Amsterdam.

Hausmann, D. and Korte, B. (1981). Algorithmic versus axiomatic definitions of matroids. *Mathematical Programming Study*, 14:98–111.

Jenkyns, T. A. (1976). The efficacy of the greedy algorithm. In Hoffman, F., Lesniak, L., and et.al., R. M., editors, *Proceedings of the 7th Southeastern Conference on Combinatorics, Graph Theory and Computing*, Baton Rouge.

Lawler, E. L. (1975). Matroid intersection algorithms. *Mathematical Programming*, 9:31–56.

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

Truemper, K. (1992). *Matroid Decomposition*. Academic Press, Boston.

Welsh, D. J. A. (1976). *Matroid Theory*. Academic Press, London.

Welsh, D. J. A. (1995). Matroids: Fundamental Concepts. In et al. (Hrsg.), R. L. G., editor, *Handbook of Combinatorics*, volume I, chapter 9, pages 481–526. North-Holland, Amsterdam.

Kapitel 6

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.

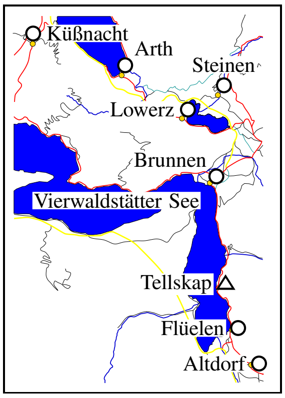
Kürzeste-Wege-Probleme spielen in der kombinatorischen Optimierung eine große Rolle. Es ist daher nicht überraschend, dass es zu diesem Problemkreis eine außerordentlich umfangreiche Literatur und sehr viele Lösungsvorschläge gibt. Wenn man dann noch Variationen hinzunimmt wie: Berechnung längster Wege 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), 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 die kürzesten 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 Abb. 6.2.
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.

Figur 6.1: F. Schiller.



Figur 6.2: Vierwaldstätter See.

Figur 6.3: W. Tell.

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.

6.1 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 $a \in A$ 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, sagen wir u, mit der kleinsten temporären Distanz ist der Knoten, der am nächsten zu s liegt. 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.

(6.1) DIJKSTRA-Algorithmus.

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

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

Datenstrukturen:

$\mathrm{DIST}(v)$ *(= Länge des kürzesten* (s, v)*-Weges),*

$\mathrm{VOR}(v)$ *(= Vorgänger von v im kürzesten (s,v)-Weg).*

1. *Setze:*
$$\begin{aligned} \mathrm{DIST}(s) &:= 0 & \\ \mathrm{DIST}(v) &:= c((s,v)) & \forall v \in V \text{ mit } (s,v) \in A \\ \mathrm{DIST}(v) &:= +\infty & \forall v \in V \text{ mit } (s,v) \notin A \\ \mathrm{VOR}(v) &:= s & \forall v \in V \setminus \{s\} \end{aligned}$$
Markiere s, alle übrigen Knoten seien unmarkiert.

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

3. *Für alle unmarkierten Knoten v mit $(u,v) \in A$ führe aus:*
$$\text{Falls } \quad \mathrm{DIST}(v) > \mathrm{DIST}(u) + c((u,v)) \text{ setze:}$$
$$\mathrm{DIST}(v) := \mathrm{DIST}(u) + c((u,v)) \text{ und } \mathrm{VOR}(v) := u.$$

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

5. *Für alle markierten Knoten v ist $\mathrm{DIST}(v)$ die Länge eines kürzesten (s,v)-Weges. Falls v markiert ist und $\mathrm{DIST}(v) < +\infty$, so ist $\mathrm{VOR}(v)$ 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 $\mathrm{DIST}(v) = +\infty$, so heißt das, dass es in D keinen (s,v)-Weg gibt.)* □

Zur Notationsvereinfachung für den nachfolgenden Beweis bezeichnen wir mit $\mathrm{DIST}_k(v)$ den Wert der in (6.1) 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 $\mathrm{DIST}_k(v_1) \le \ldots \le \mathrm{DIST}_k(v_k) \le \mathrm{DIST}_k(v)$ für alle bisher unmarkierten Knoten v.

(6.2) Satz. *Der Dijkstra-Algorithmus arbeitet korrekt.*

Beweis : Wir zeigen durch Induktion über die Anzahl k markierter Knoten Folgendes: Ist v markiert, so enthält $\mathrm{DIST}_k(v)$ die Länge eines kürzesten (s,v)-Weges, ist v unmarkiert, so enthält $\mathrm{DIST}_k(v)$ die Länge eines kürzesten (s,v)-Weges, wobei als innere Knoten nur markierte Knoten zugelassen sind. (Falls $\mathrm{DIST}_k(v) = +\infty$, 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:

(6.3) 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 (4.14) und der DIJKSTRA-Algorithmus (6.1) (im Wesentlichen) identische Algorithmen sind. Sie unterscheiden sich lediglich bezüglich einer Gewichtstransformation. In Schritt 3 von (4.14) wird $\min\{c(e) \mid e \in \delta(W)\}$ gesucht, in Schritt 2 von (6.1) 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).

6.2 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 (6.2) 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 $\mathrm{DIST}(v)$ wieder mit $+\infty$ oder mit $c((s, v))$ ($\mathrm{DIST}(v)$ enthält also die Länge des kürzesten zur Zeit bekannten (s, v)-Weges mit einer bestimmten Eigenschaft) und setzen wie in (6.1) $\mathrm{VOR}(v) = s$. Nun versuchen wir, die Distanzen $\mathrm{DIST}(v)$ sukzessive zu reduzieren. Finden wir einen Bogen (u, v) mit $\mathrm{DIST}(u) + c((u, v)) < \mathrm{DIST}(v)$, so setzen wir $\mathrm{DIST}(v) := \mathrm{DIST}(u) + c((u, v))$ und $\mathrm{VOR}(v) := u$. Wir führen diese Iteration so lange fort, bis kein Wert $\mathrm{DIST}(u)$ 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 6.1 mit den dort eingetragenen Gewichten.

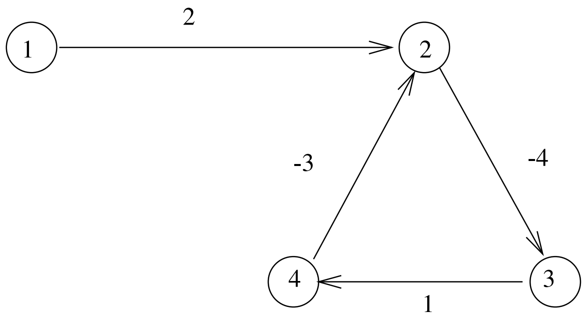


Abb. 6.1

Wir initialisieren mit $\text{DIST}(1) = 0$, $\text{DIST}(2) = 2$, $\text{DIST}(3) = \text{DIST}(4) = +\infty$, $\text{VOR}(i) = 1, i = 1, 2, 3, 4$. Wir stellen fest, dass $\text{DIST}(3) > \text{DIST}(2) + c((2,3)) = -2$, und setzen $\text{DIST}(3) = -2$, $\text{VOR}(3) = 2$. 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.

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

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

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

Datenstrukturen:

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

1. *Setze:*

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

$$\text{VOR}(v) := s.$$

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

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

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

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

   END 3.

   END 2.

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

(6.5) Satz. *Algorithmus (6.4) 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 $\mathrm{DIST}(v)$ 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 $\mathrm{DIST}(u)$ 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 $\mathrm{DIST}(u)$ 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 $\mathrm{DIST}(u)$, $1 \le u \le v$, die Länge eines kürzesten (s, u)-Weges. Da ein $(s, v + 1)$-Weg entweder von s direkt nach $v + 1$ führt (das Gewicht dieses Weges ist in $\mathrm{DIST}(v + 1)$ 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:

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

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

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

Da das Verfahren (6.4) 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:

(6.6) MOORE-BELLMAN-Algorithmus.

Input: *Digraph* $D = (V, A)$, *Gewichte* $c(a)$ *für alle* $a \in A$ *(können auch negativ sein), ein Knoten* $s \in V$.

Output: *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: $\mathrm{DIST}(v)$, $\mathrm{VOR}(v)$ *für alle* $v \in V$ *(wie in (6.1))*

1. *Setze:*

$$\mathrm{DIST}(s) := 0,$$
$$\mathrm{DIST}(v) := \infty \;\forall\, v \in V \setminus \{s\}$$
$$\mathrm{VOR}(v) := s \quad \forall\, v \in V.$$

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* $\mathrm{DIST}(u) + c((u, v)) < \mathrm{DIST}(v)$,
   setze $\mathrm{DIST}(v) := \mathrm{DIST}(u) + c((u, v))$ *und* $\mathrm{VOR}(v) := u$.
   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* $\mathrm{DIST}(u) + c((u, v)) < \mathrm{DIST}(v)$,
   setze $\mathrm{DIST}(v) := \mathrm{DIST}(u) + c((u, v))$
   und $\mathrm{VOR}(v) := u$.
   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 $\mathrm{DIST}(u) + c((u, v)) < \mathrm{DIST}(v)$

a) setze $\mathrm{DIST}(v) := \mathrm{DIST}(u) + c((u, v))$ und $\mathrm{VOR}(v) := u$,

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 $\mathrm{DIST}(v) < +\infty$, so enthält $\mathrm{DIST}(v)$ die Länge eines kürzesten (s, v)-Weges, und aus $\mathrm{VOR}(v)$ kann wie üblich ein kürzester (s, v)-Weg rekonstruiert werden. Ist $\mathrm{DIST}(v) = +\infty$, 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.

(6.7) 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 (6.6) angenommen, dass $s = 1$ gilt und die Knoten mit $1, 2, \dots, n$ 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 $\mathrm{DIST}(v, m)$ den Inhalt des Vektors $\mathrm{DIST}(v)$ nach Beendigung der m-ten Iteration der äußeren Schleife. Wir behaupten nun, dass $\mathrm{DIST}(v, m)$, $0 \le m \le n - 2$, die Länge des kürzesten aller der $(1, v)$-Wege enthält, bei denen nur m Richtungswechsel erlaubt sind. 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 (6.4) (angewendet auf $s = 1$), dessen

Korrektheit wir in Satz (6.5) bewiesen haben. $\text{DIST}(v, 0)$ enthält somit die Länge des kürzesten $(1, v)$-Weges ohne Richtungswechsel, die Behauptung für $m = 0$ ist 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 (s, v)-Wege mit höchstens $m + 1$ Richtungswechseln besteht aus folgenden Wegen:

(a) (s, v)-Wege mit höchstens m Richtungswechseln,

(b) (s, v)-Wege mit genau $m + 1$ Richtungswechseln.

Die Minimallänge der Wege in (a) kennen wir bereits, sie ist nach Induktionsvoraussetzung $\text{DIST}(v, m)$. Da nach Annahme der erste Bogen eines jeden Weges ein Aufwärtsbogen ist, ist für einen (s, v)-Weg mit $m + 1$ Richtungswechseln und $m + 1$ ungerade, 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 (s, 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 (s, w)-Weges mit höchstens $m + 1$ Richtungswechseln ist. Zur Bestimmung der Länge eines kürzesten $(s, u - 1)$-Weges müssen wir die Länge eines kürzesten $(s, u - 1)$-Weges mit höchsten m Richtungswechseln (diese ist in $\text{DIST}(u - 1, m)$ gespeichert) vergleichen mit der Länge eines kürzesten $(s, u - 1)$-Weges mit genau $m + 1$ Richtungswechseln. Im letzteren Falle ist der letzte Bogen eines solchen $(s, u - 1)$-Weges ein Abwärtsbogen. Der vorletzte Knoten w erfüllt also $u - 1 < w \leq n$. Ist für $u - 1 < w \leq n$ P ein kürzester (s, w)-Weg, und hat P höchstens m Richtungswechsel, so ist, falls $(w, u - 1) \in A$, $P \cup \{(w, u - 1)\}$ ein $(s, u - 1)$-Weg mit höchstens m Richtungswechseln. Hat P $m + 1$ Richtungswechsel, so ist der letzte Bogen in P nach obiger Bemerkung ein Abwärtsbogen, also hat auch $P \cup \{(w, u - 1)\}$ genau $m + 1$ Richtungswechsel. Nach Induktionsvoraussetzung befindet sich in $\text{DIST}(w, m + 1)$ für $u - 1 < w \leq n$ die Länge eines kürzesten (s, w)-Weges mit höchstens $m + 1$ Richtungswechseln, also ist $\min\{\text{DIST}(w, m + 1) + c(w, u - 1) \mid u - 1 < w \leq n\}$ nicht größer als die Länge des kürzesten $(s, u - 1)$-Weges mit genau $m + 1$ Richtungswechseln. Dieses Minimum wird offenbar in der Schleife 6 bestimmt und mit $\text{DIST}(u - 1, m)$ verglichen. Damit ist gezeigt, dass der Induktionsschritt der zweiten Induktion korrekt ist. Hieraus folgt die Korrektheit der ersten Induktion, und wir sind fertig. $\square$

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.

(6.8) 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 $\mathrm{DIST}(v)$ für mindestens einen Knoten $v \in V$ geändert wird. □

Der Beweis dieser Bemerkung sei dem Leser überlassen. Im nächsten Abschnitt werden wir auf das Thema "negative Kreise" 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 (siehe Übungsaufgabe). 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.

6.3 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.

(6.9) FLOYD-Algorithmus.

Input: *Digraph $D = (V, A)$, $V = \{1, \dots, n\}$ mit Gewichten $c(a)$ (können auch negativ sein), für alle $a \in A$.*

Output: *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:

$$w_{ij} := \begin{cases} c((i,j)) & \textit{falls } (i,j) \in A \\ +\infty & \textit{andernfalls} \end{cases}$$

$$p_{ij} := \begin{cases} i & \textit{falls } (i,j) \in A \\ 0 & \textit{andernfalls (bedeutet, zur Zeit kein Weg bekannt)} \end{cases}$$

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 $i = v_k$ 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

$$(i = v_s, v_{s-1}, v_{s-2}, \ldots, v_1, j)$$

ein kürzester (i, j)-Weg. Überzeugen Sie sich, dass dies stimmt!

(6.10) 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 von der äußeren Schleife von 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 $w_{ii}^l < 0$ 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 2 zum $(l + 1)$-sten Male durchlaufen. Bei diesem Durchlauf haben wir folgenden Schritt ausgeführt.

$$\text{Falls } w_{ij}^l > w_{i,l+1}^l + w_{l+1,j}^l, \text{ dann setze } w_{ij}^{l+1} := w_{i,l+1}^l + w_{l+1,j}^l,$$

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_{ij}^l ist, setzen wir $w_{ij}^{l+1} := w_{i,l+1}^l + w_{l+1,j}^l$, andernfalls $w_{ij}^{l+1} = w_{ij}^l$. Daraus folgt die Behauptung, es sei denn, $w_{ij}^l > w_{i,l+1}^l + w_{l+1,j}^l$ 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. $w_{ij}^l \leq c(\overline{K})$. Aus $c(K) = w_{i,l+1}^l + w_{l+1,j}^l < w_{ij}^l$ folgt, dass mindestens einer der aus K entfernten gerichteten Kreise eine negative Länge hat. Für jeden Knoten i dieses negativen Kreises muss folglich $w_{ii}^{l+1} < 0$ gelten. Daraus folgt die Behauptung. □

Der FLOYD-Algorithmus liefert also explizit einen Kreis negativer Länge, falls ein solcher existiert.

(6.11) Folgerung. *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 (6.9) 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 wohl so?

6.4 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 so genannte 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.

(6.12) Satz. *Sei $D = (V, A)$ ein Digraph, und seien $s, t \in V$. 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 k bogendisjunkte (s, t)-Schnitte , so hat jeder (s, t)-Weg mindestens die Länge k. Also 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 einen 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.

(6.13) Satz. *Seien $D = (V, A)$ ein Digraph, $s, t \in V$ und $c(a) \in \mathbb{Z}_+$ für alle $a \in A$. Dann ist die kürzeste Länge eines (s, t)-Weges gleich der maximalen Anzahl k von (nicht notwendig verschiedenen) (s, t)-Schnitten $c_1, ..., c_k$, so dass jeder Bogen $a \in A$ in höchstens $c(a)$ Schnitten c_i liegt.*

Beweis : Sei P ein (s,t)-Weg und seien $c_1,...,c_k$ (s,t)-Schnitte wie im Satz gefordert, dann gilt

$$c(P) = \sum_{a\in P} c(a) \geq \sum_{a\in P} (\text{Anzahl von } i \text{ mit } a \in c_i) = \sum_{i=1}^{k} |c_i \cap P| \geq \sum_{i=1}^{k} 1 = k$$

Also ist das Minimum nicht kleiner als das Maximum.

Wählen wir die (s,t)-Schnitte $c_i := \delta^+(V_i)$, mit V_i = Menge der Knoten aus V, die höchstens die Entfernung $i-1$ von s haben, $i = 1,...,d$, dann sehen wir, dass Gleichheit gilt. □

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 Vewendung $n = |V| \geq 2^{12^{20}}$ 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.

$O(l_n\mathcal{C} + m + nl_nl_nl_nn)$, wobei $\mathcal{C}$ das größte Kantengewicht bezeichnet.)

Bei Routenplanern, wie sie z.B. im Internet oder 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 lösen.

Literaturverzeichnis

Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1993). *Network Flows, Theory, Algorithms and Applications*. Pearson Education, Prentice Hall, New York, first edition.

Bellman, R. E. (1958). On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numer. Math.*, 1:269–271.

Domschke, W. (1972). *Kürzeste Wege in Graphen*. Verlag A. Hain, Meisenheim am Glan.

Floyd, R. W. (1962). Algorithm 97, Shortest path. *Communications of the ACM*, 5(6):345.

Glover, F., Klingman, D. D., and Phillips, N. V. (1985). A New Polynomially Bounded Shortest Path Algorithm. *Operations Research*, 33(1):65–73.

Lawler, E. L. (1976). *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart & Winston, New York.

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

Moore, E. F. (1959). The shortest path through a maze. In *Proc. Int. Symp. on Theory of Switching Part II*, pages 285–292, Cambridge, Mass. Harvard University Press.

Schrijver, A. (2003). *Combinatorial Optimization – Polyhedra and Efficiency*. Springer-Verlag, Berlin.

Syslo, M. M., Deo, N., and Kowalik, J. S. (1983). *Discrete Optimization Algorithms (with PASCAL programs*. Prentice Hall, Englewood Cliffs, N.J.

Thorup, M. (1997). Undirected single shortest paths in linear time. In *Proceedings of the 38th IEE Symposium on Foundations of Comp. Sci. (FOCS)*, pages 12–21.

Kapitel 7

Maximale Flüsse in Netzwerken

In diesem Kapitel behandeln wir ein in sowohl theoretischer als auch praktischer Hinsicht außerordentlich 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, 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.

Darüber hinaus treten Netzwerkflussprobleme in vielfältiger Weise als Unter- oder Hilfsprobleme bei der Lösung komplizierter Anwendungsprobleme auf. Insbesondere werden Netzwerkflussalgorithmen sehr häufig als Separierungsroutinen bei Schnittebenenverfahren eingesetzt.

Das klassische Werk der Netzwerkflusstheorie ist das Buch Ford Jr. 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), Goldberg et al. (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.

7.1 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 $x : A \to \mathbb{R}$ (bzw. ein Vektor $x \in \mathbb{R}^A$) heißt **zulässiger (s,t)-Fluß**, wenn die folgenden beiden Bedingungen erfüllt sind:

(7.1) $\quad 0 \leq x_a \leq c_a \quad$ für alle $a \in A$ $\quad$ (Kapazitätsbedingungen)

(7.2) $\quad \sum_{a \in \delta^-(v)} x_a = \sum_{a \in \delta^+(v)} x_a \quad$ für alle $v \in V \setminus \{s, t\}$ $\quad$ (Flußerhaltungsbedingungen)

Ist $x \in \mathbb{R}^A$ ein zulässiger (s,t)-Fluss, dann heißt

(7.3) $$\operatorname{val}(x) := \sum_{a \in \delta^+(s)} x_a - \sum_{a \in \delta^-(s)} x_a$$

der **Wert** des (s,t)-Flusses x.

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 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 $t \notin V \setminus W$ 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:

(7.4) Lemma. *(a) Seien $s \in W$, $t \notin W$, dann gilt für jeden zulässigen (s,t)-Fluss x:*

$$\text{val}(x) = \sum_{a \in \delta^+(W)} x_a - \sum_{a \in \delta^-(W)} x_a.$$

(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 (7.2) folgt

$$\begin{aligned} \text{val}(x) &= \sum_{a \in \delta^+(s)} x_a - \sum_{a \in \delta^-(s)} x_a = \sum_{v \in W} \Big(\sum_{a \in \delta^+(v)} x_a - \sum_{a \in \delta^-(v)} x_a \Big) \\ &= \sum_{a \in \delta^+(W)} x_a - \sum_{a \in \delta^-(W)} x_a. \end{aligned}$$

(b) Seien $\delta^+(W)$ ein beliebiger (s,t)-Schnitt und x ein zulässiger (s,t)-Fluss, dann gilt wegen (a), (7.1) und (7.2)

$$\text{val}(x) = \sum_{a \in \delta^+(W)} x_a - \sum_{a \in \delta^-(W)} x_a \le \sum_{a \in \delta^+(W)} c_a = c(\delta^+(W)).$$

□

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 in der Vorlesung "Lineare Optimierung" behandelt werden.

Wir schreiben zunächst die Aufgabe, einen maximalen (s,t)-Fluss wert in D zu finden, als lineares Programm. Dieses lautet wie folgt:

$$\max \sum_{a \in \delta^+(s)} x_a - \sum_{a \in \delta^-(s)} x_a \quad (= x(\delta^+(s)) - x(\delta^-(s)))$$

$$(7.5) \qquad \begin{aligned} x(\delta^-(v)) - x(\delta^+(v)) &= \sum_{a \in \delta^-(v)} x_a - \sum_{a \in \delta^+(v)} x_a = 0 \ \text{ für alle } v \in V \setminus \{s,t\}, \\ & 0 \le x_a \le c_a \text{ für alle } a \in A. \end{aligned}$$

Jede zulässige Lösung von (7.5) ist also ein zulässiger (s,t)-Fluss, und jede optimale Lösung ein optimaler (s,t)-Fluss. Um das zu (7.5) 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 (7.5) dual.

$$\begin{array}{llll}
\min \sum\limits_{a \in A} c_a y_a & & & \\
y_a + z_v - z_u \geq 0 & \text{falls} & a = (u,v) \in A(V \setminus \{s,t\}) \\
y_a \quad\;\; - z_u \geq -1 & \text{falls} & a = (u,s), u \neq t \\
y_a + z_v \qquad \geq 1 & \text{falls} & a = (s,v), v \neq t \\
y_a \quad\;\; - z_u \geq 0 & \text{falls} & a = (u,t), u \neq s \\
y_a + z_v \qquad \geq 0 & \text{falls} & a = (t,v), v \neq s \\
y_a \qquad\qquad \geq 1 & \text{falls} & a = (s,t) \\
y_a \qquad\qquad \geq -1 & \text{falls} & a = (t,s) \\
\qquad\quad y_a \geq 0 & \text{für alle} & a \in A
\end{array}$$

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:

$$(7.6) \qquad \begin{array}{ll}
\min \sum\limits_{a \in A} c_a y_a & \\
y_a + z_v - z_u \geq 0 & \text{für alle } a = (u,v) \in A \\
z_s = 1 & \\
z_t = 0 & \\
y_a \geq 0 & \text{für alle } a \in A.
\end{array}$$

Wir benutzen nun (7.5) und (7.6), um folgenden berühmten Satz, der auf Ford Jr. and Fulkerson (1956) und Elias et al. (1956) zurückgeht, zu beweisen.

(7.7) Das 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 (s,t)-Flusses gleich der minimalen Kapazität eines (s,t)-Schnittes.*

Beweis : Aufgrund von Lemma (7.4) 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 (7.5) 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 (7.5) dualen Programms (7.6) 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

$x_a \le c_a$ des primalen Programms (7.5) 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 $z^*_v > 0$, $z^*_u \le 0$ und somit (da $y^*_a \ge 0$) $y^*_a - z^*_u + z^*_v \ge z^*_v - z^*_u > 0$. Die Ungleichung ist also "locker". Der Satz vom komplementären Schlupf impliziert nun, dass die zugehörige Primalvariable $x_a \ge 0$ 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 (7.6) können wir in polynomialer Zeit einen (s,t)-Schnitt konstruieren, der den gleichen Wert wie die Optimallösung von (7.6) 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 (7.6) konstruieren, und daraus folgt, dass das lineare Programm (7.6) immer auch ganzzahlige optimale Lösungen hat. Wir können somit also das ganzzahlige Programm bestehend aus (7.6) plus Ganzzahligkeitsbedingungen für y_a und z_v durch das im Beweis von (7.7) angegebene Verfahren lösen.

Wir werden im nächsten Abschnitt zeigen, dass auch das LP (7.5) 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).

7.2 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 (7.5) und (7.6) 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 drei dieser Verfahren vorstellen.

Das erste dieser Verfahren geht auf Ford und Fulkerson zurück. Die Idee hinter dieser Methode kann wie folgt erläutert werden. Man starte mit dem zulässigen (s,t)-Fluss $x_a = 0$ für alle $a \in A$. Hat man einen zulässigen (s,t)-Fluss, 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 7.1, 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.

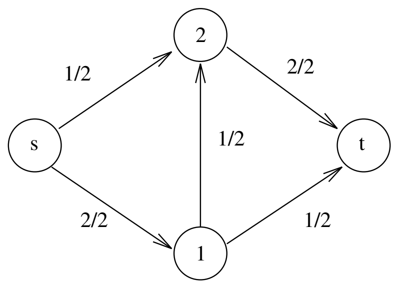


Abb. 7.1

Der gegenwärtige Fluss hat den Wert 3, und offenbar hat der Maximalfluss den Wert 4. Es gibt im Digraphen von Abbildung 7.1 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 (7.1) noch eine der Flusserhaltungsbedingungen verletzt, aber der Flusswert um 1 erhöht. Wir treffen daher folgende Definition.

(7.8) 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) gilt 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 7.1 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.

(7.9) Satz. *Ein (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

(7.10) $$\varepsilon := \min \begin{cases} c_{ij} - x_{ij} & \text{falls } (i,j) \in P \text{ Vorwärtsbogen,} \\ x_{ij} & \text{falls } (i,j) \in P \text{ Rückwärtsbogen.} \end{cases}$$

Setzen wir

(7.11) $$x'_{ij} := \begin{cases} x_{ij} + \varepsilon & \text{falls } (i,j) \in P \text{ Vorwärtsbogen,} \\ x_{ij} - \varepsilon & \text{falls } (i,j) \in P \text{ Rückwärtsbogen,} \\ x_{ij} & \text{falls } (i,j) \in A \setminus P, \end{cases}$$

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 (7.8) 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 (7.4) (b) ist somit x maximal. $\square$

Der Beweis von Satz (7.9) liefert einen Schnitt $\delta^+(W)$ mit $\text{val}(x) = c(\delta^+(W))$. Zusammen mit Lemma (7.4) (b) ergibt dies einen kombinatorischen Beweis des Max-Flow Min-Cut-Theorems. Aus dem Beweis von Satz (7.9) folgt ebenfalls, dass das lineare Programm (7.5) ganzzahlige Optimallösungen hat, falls alle Kapazitäten ganzzahlig sind.

(7.12) 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 (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 (7.10). Nach Voraussetzung ist ε ganzzahlig, und folglich ist der neue durch (7.11) festgelegte Flussvektor ebenfalls ganzzahlig. Bei jeder Augmentierung erhöhen wir den Flusswert um mindestens eins. Da der maximale Flusswert endlich ist, folgt die Behauptung aus (7.9). $\square$

Wir können nun den Ford-Fulkerson-Algorithmus angeben:

(7.13) FORD-FULKERSON-Algorithmus.

Input: *Digraph* $D = (V, A)$ *mit Bogenkapazitäten* $c_a \in \mathbb{R}$, $c_a \geq 0$ *für alle Bögen* $a \in A$ *und zwei Knoten* $s, t \in V$, $s \neq t$.

Output: *Ein zulässiger* (s, t)*-Fluss* x *mit maximalem Wert* $\operatorname{val}(x)$ *und ein kapazitätsminimaler* (s, t)*-Schnitt* $\delta^+(W)$.

1. *(Initialisierung) Sei* $x = (x_{ij}) \in \mathbb{R}^A$ *ein zulässiger* (s,t)*-Fluss, 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 auf einem augmentierenden* $[s, v]$*-Weg gespeichert wird)*
   - EPS *(=* $(n-1)$*-Vektor, zur sukzessiven Berechnung von (7.10).).*

Markieren und Überprüfen

2. *Setze* $W := \{s\}, U := \{s\}, \mathrm{EPS}(s) := +\infty$.
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*
   $\mathrm{VOR}(j) := +i$, $\mathrm{EPS}(j) := \min\{c_{ij} - x_{ij}, \mathrm{EPS}(i)\}$, $W := W \cup \{j\}$, $U := U \cup \{j\}$.
6. *Führe für alle Bögen* $(j, i) \in A$ *mit* $j \notin W$ *folgendes aus:*
   Ist $x_{ji} > 0$, *dann setze*
   $\mathrm{VOR}(j) := -i$, $\mathrm{EPS}(j) := \min\{x_{ji}, \mathrm{EPS}(i)\}$, $W := W \cup \{j\}$, $U := U \cup \{j\}$.
7. *Gilt* $t \in W$, *gehe zu 8, andernfalls zu 3.*

Augmentierung

8. *Konstruiere einen augmentierenden Weg und erhöhe den gegenwärtigen Fluss um* $\mathrm{EPS}(t)$, *d. h. bestimme* $j_1 = |\mathrm{VOR}(t)|$, *falls* $\mathrm{VOR}(t) > 0$, *setze* $x_{j_1 t} := x_{j_1 t} + \mathrm{EPS}(t)$, *andernfalls setze* $x_{t j_1} := x_{t j_1} - \mathrm{EPS}(t)$. *Dann bestimme* $j_2 := |\mathrm{VOR}(j_1)|$, *falls* $\mathrm{VOR}(j_1) > 0$, *setze* $x_{j_2 j_1} := x_{j_2 j_1} + \mathrm{EPS}(t)$, *andernfalls* $x_{j_1 j_2} := x_{j_1 j_2} - \mathrm{EPS}(t)$ *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 (7.9) und (7.12) folgt, dass Algorithmus (7.13) 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.

Also funktioniert (7.13) auch bei rationalen Daten. Lässt man (zumindest theoretisch) auch irrationale Kapazitäten zu, so kann man Beispiele konstruieren, bei denen Algorithmus (7.13) 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 $0(m)$, $m = |A|$, 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 (7.13) $0(m \cdot v)$. Diese Laufzeit ist nicht polynomial in $n + m + \sum_{a \in A} \langle c_a \rangle$, und wenn man die im Verfahren (7.13) noch nicht exakt spezifizierten Schritte ungeschickt ausführt, kann man tatsächlich zu exorbitanten Laufzeiten kommen. Allerdings haben Edmonds and Karp (1972) gezeigt:

(7.14) Satz. *Falls in Algorithmus (7.13) 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 (7.13) $0(m^2n)$.* □

Satz (7.14) gilt für beliebige (auch irrationale) Bogenkapazitäten. Es ist in diesem Zusammenhang interessant zu bemerken, dass praktisch jeder, der Verfahren (7.13) implementiert, die Edmonds-Karp-Vorschrift einhält. Üblicherweise arbeitet man die Knoten in Breadth-First-Strategie 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.

(7.15) Beispiel. Wir betrachten den in Abbildung 7.2 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 7.2 starten wir also mit einem Fluss des Wertes 10.

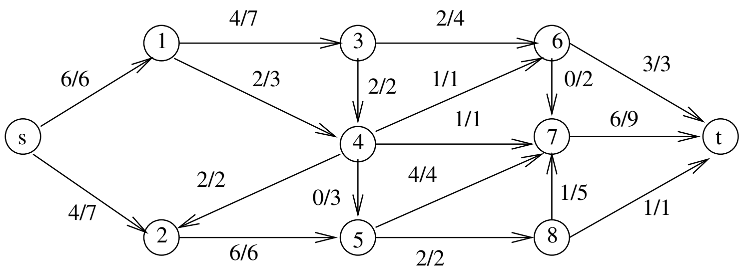


Abb. 7.2

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 $s \in U, U := \emptyset$.
5. $W := \{s, 2\}$, $U := \{2\}$
VOR $= (-, +s, -, -, -, -, -, -, -)$
EPS $= (-, 3, -, -, -, -, -, -, -)$.
6. –
7. –
3. –
4. Wir wählen $2 \in U, U := \emptyset$.
5. –
6. $W := \{s, 2, 4\}$, $U := \{4\}$
VOR $= (-, +s, -, -2, -, -, -, -, -)$
EPS $= (-, 3, -, 2, -, -, -, -, -)$.
7. –
3. –
4. Wir wählen $4 \in U, U := \emptyset$.
5. $W := \{s, 2, 4, 5\}$, $U := \{5\}$
VOR $= (-, +s, -, -2, +4, -, -, -, -)$
EPS $= (-, 3, -, 2, 2, -, -, -, -)$.
6. $W := \{s, 2, 4, 5, 1, 3\}$, $U := \{5, 1, 3\}$
VOR $= (-4, +s, -4, -2, +4, -, -, -, -)$
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. $W := \{s, 2, 4, 5, 1, 3, 6\}$, $U := \{6\}$
VOR $= (-4, +s, -4, -2, +4, +3, -, -, -)$
EPS $= (2, 3, 2, 2, 2, 2, -, -, -)$.
6. –
7. –
3. –
4. Wir wählen $6 \in U, U := \emptyset$.
5. $W := \{s, 2, 4, 5, 1, 3, 6, 7\}$, $U := \{7\}$
VOR $= (-4, +s, -4, -2, +4, +3, +6, -, -)$
EPS $= (2, 3, 2, 2, 2, 2, 2, -, -)$.
6. –
7. –
3. –

4. Wir wählen $7 \in U$, $U := \emptyset$.
5. $W := \{s, 2, 4, 5, 1, 3, 6, 7, t\}$, $U := \{t\}$
VOR $= (-4, +s, -4, -2, +4, +3, +6, -, +7)$
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. $t \in W$
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 7.3 dargestellt. Dieser (s, t)-Fluss ist maximal, ein (s, t)-Schnitt minimaler Kapazität ist $\delta^+(\{s, 2\})$, ein anderer $\delta^+(\{s, 1, 2, 3, 4, 5\})$. $\square$

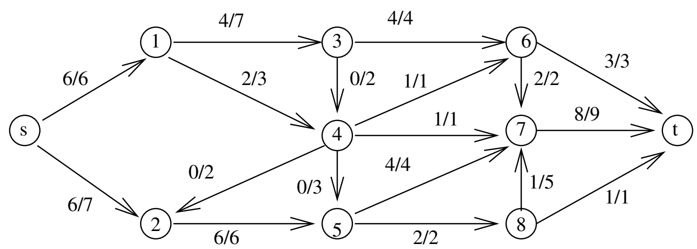


Abb. 7.3

7.3 Der Dinic-Malhorta-Kumar-Maheshwari-Algorithmus.

In diesem Abschnitt beschreiben wir einen Algorithmus zur Lösung des Maximalflussproblems, dessen Grundgerüst auf Dinic (1970) zurückgeht. Die wichtigste Phase dieses Algorithmus wurde in Malhorta et al. (1978) entscheidend verbessert. Wir wollen den im nachfolgenden dargestellten Algorithmus nach den vier oben aufgeführten Autoren benennen. Zur Abkürzung schreiben wir einfach **DMKM-Algorithmus**. Aus Zeitgründen wird nur eine relativ informelle Beschreibung des DMKM-Algorithmus gegeben. Sie basiert auf Syslo et al. (1983). Eine sehr detaillierte Analyse des Verfahrens mit allen notwendigen Korrektheitsbeweisen und einer sorgfältigen Abschätzung der Laufzeit kann man in Mehlhorn (1984) finden.

Der DMKM-Algorithmus verfolgt die folgende generelle Strategie. Statt wie im Ford-Fulkerson-Algorithmus im gesamten Digraphen D mit gegebenem (s, t)-Fluss x einen augmentierenden $[s, t]$-Weg zu suchen, wird aus D und x ein azyklischer Digraph N (genannt geschichtetes Netzwerk) konstruiert, der mit weniger Zeitaufwand zu bearbeiten ist. Ein besonderes Merkmal dieser Konstruktion ist, dass N genau dann den Knoten t nicht enthält, wenn der gegenwärtige (s, t)-Fluss x maximal ist. Ist t in N enthalten, so versucht man einen möglichst großen (s, t)-Fluss in N zu finden. Nach Konstruktion kann jeder (s, t)-Fluss in N zum (s, t)-Fluss x in D "augmentiert" werden, und man kann ein neues geschichtetes Netzwerk N bestimmen. Da die Bestimmung eines maximalen (s, t)-Flusses in N zu aufwendig ist, begnügt man sich mit der Bestimmung eines sogenannten saturierten (s, t)-Flusses in N. Ein solcher ist relativ leicht zu finden. Das Verfahren zur Auffindung eines saturierten (s, t)-Flusses in N garantiert, dass ein (s, t)-Fluss mit positivem Wert in N gefunden wird, falls ein solcher existiert. Daraus folgt bereits, dass das Verfahren funktioniert. Eine weitere Besonderheit ist die Methode zur sukzessiven Verbesserung des (s, t)-Flusses in N bis zur Erreichung eines saturierten (s, t)-Flusses. Hier wird nicht bogenorientiert gearbeitet; man versucht stattdessen, durch die vorhandenen Knoten soviel Fluss wert wie möglich "durchzuschieben". Dabei bestimmt man einen Flusswert, der mit Sicherheit zum gegenwärtigen Fluss "augmentiert" werden kann; ferner ist dieser Fluss relativ einfach auf die Bögen zu verteilen. Bevor wir die Phasen des DMKM-Algorithmus einzeln beschreiben, fassen wir das formale Konzept des Verfahrens zusammen.

(7.16) DMKM-Algorithmus (Überblick).

Input: *Digraph $D = (V, A)$ mit nichtnegativen Bogenkapazitäten c_a für alle $a \in A$, zwei Knoten $s, t \in V$.*

Output: *Maximaler (s, t)-Fluss x.*

1. *Setze $x_a = 0$ für alle $a \in A$.*
2. *Konstruiere aus D und x ein geschichtetes Netzwerk $N = (W, B)$ mit Bogenkapazitäten $\overline{c}_a$ für alle $a \in B$.*
3. *Ist $t \notin W$, STOP. Der gegenwärtige (s, t)-Fluss ist maximal.*
4. *Bestimme einen saturierten (s, t)-Fluss $\overline{x}$ in $N = (W, B)$.*
5. *Erhöhe den Fluss x durch "Augmentierung" von $\overline{x}$ und gehe zu 2.* □

Saturierte (s, t)-Flüsse in geschichteten Netzwerken

Wir beginnen mit der Darstellung eines wichtigen Teils des DMKM-Algorithmus, der Bestimmung eines saturierten Flusses in einem geschichteten Netzwerk.

Es sei $N = (W, B)$ ein Digraph mit nicht-negativen Bogenkapazitäten $\overline{c}(a)$ für alle $a \in B$, und s, t seien zwei voneinander verschiedene Knoten von N. Zusätzlich nehmen wir an, dass N azyklisch ist, also keinen gerichteten Kreis enthält, und dass die Knotenmenge W in "Schichten" $V_1, \ldots, V_k$ mit folgenden Eigenschaften zerlegt werden kann:

$$(7.17) \qquad \begin{aligned} V_1 &= \{s\}, \\ V_i &= \{v \in W \mid \exists\, u \in V_{i-1} \text{ mit } (u,v) \in B\} \quad i = 2, \ldots, k-1, \\ V_k &= \{t\}, \\ B &= \bigcup_{i=1}^{k-1} \{(u,v) \in B \mid u \in V_i, v \in V_{i+1}\}. \end{aligned}$$

Einen derartigen Digraphen nennnen wir ein **geschichtetes Netzwerk**. In einem geschichteten Netzwerk, sind also Knoten aus V_i nicht durch einen Bogen verbunden, ebenso gibt es für Knoten in V_i und in V_j mit $i < j - 1$ keinen Bogen, der sie miteinander verbindet. Für jeden Knoten $v \in V_i$ haben alle (s, v)-Wege die Länge $i - 1$, speziell haben alle (s, t)-Wege die Länge $k - 1$. Für einen Bogen $(u, v) \in B$ gibt es immer einen Index i mit $u \in V_i$, $v \in V_{i+1}$.

Ist x ein (s, t)-Fluss in einem (beliebigen) Digraphen $D = (V, A)$ mit Bogenkapazitäten $c(a)$, so nennen wir den **Bogen** $a \in A$ **saturiert**, falls $x_a = c_a$ gilt. Ein $\boldsymbol{(s,t)}$**-Weg** P heißt **saturiert** (bzgl. x), falls mindestens ein Bogen aus P saturiert ist. Der $\boldsymbol{(s,t)}$**-Fluß** x in D heißt **saturiert**, wenn jeder (s, t)-Weg in D saturiert ist. Man beachte, dass jeder maximale Fluss saturiert ist, dass aber nicht jeder saturierte Fluss auch maximal ist (vergleiche Abbildung 7.1). Wir zeigen nun, wie man einen saturierten Fluss in einem geschichteten Netzwerk bestimmen kann.

Sei also $N = (W, B)$ ein geschichtetes Netzwerk mit Schichten $V_1, \ldots, V_k$ und Bogenkapazitäten $\overline{c}(a) \geq 0$. Ferner sei x ein zulässiger (s, t)-Fluss in N. Im DMKM-Algorithmus wird für jeden Knoten $v \in W \setminus \{s, t\}$ der maximale zusätzliche Flusswert, der durch den Knoten v geschickt werden kann, bestimmt. Diesen Wert nennen wir **Potenzial** von v und bezeichnen ihn mit $\mathrm{POT}(v)$. Er kann wie folgt bestimmt werden.

$$(7.18) \quad \mathrm{POT}(v) := \min\{ \sum_{a \in \delta^-(v)} (\overline{c}(a) - x(a)), \sum_{a \in \delta^+(v)} (\overline{c}(a) - x(a))\}, \ v \in W \setminus \{s, t\}.$$

Unter allen Knoten $v \in W \setminus \{s, t\}$ wird ein Knoten mit minimalem Potenzial bestimmt. Dieser Knoten, sagen wir r, wird **Referenzknoten** genannt, d. h.

$$(7.19) \qquad \mathrm{POT}(r) = \min\{\mathrm{POT}(v) \mid v \in W \setminus \{s, t\}\}.$$

Abbildung 7.4 zeigt ein geschichtetes Netzwerk mit 5 Schichten und 10 Knoten; es gilt $s = 1$, $t = 10$. Wie üblich gibt die erste Zahl eines Zahlenpaares bei einem Bogen den gegenwärtigen Fluss durch den Bogen und die zweite Zahl die Bogenkapazität an. Aus (7.18) folgt

$$\begin{aligned} &\mathrm{POT}(2) = 4, \ \mathrm{POT}(3) = 4, \ \mathrm{POT}(4) = 5, \ \mathrm{POT}(5) = 3, \\ &\mathrm{POT}(6) = 9, \ \mathrm{POT}(7) = 4, \ \mathrm{POT}(8) = 4, \ \mathrm{POT}(9) = 5. \end{aligned}$$

Der Referenzknoten ist somit eindeutig bestimmt, es ist der Knoten 5.

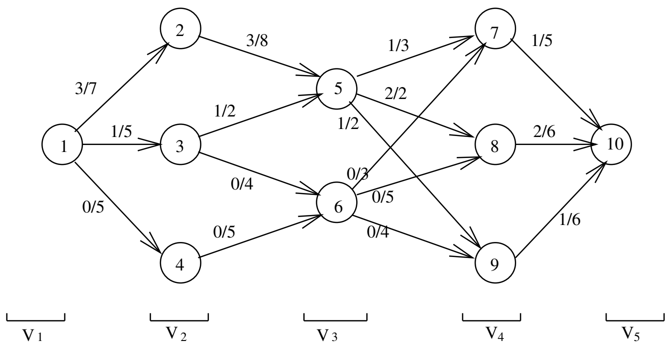


Abb. 7.4

Wir wollen nun vom Referenzknoten r ausgehend den zusätzlichen Flusswert $\mathrm{POT}(r)$ durch das gesamte Netzwerk schicken. Wir nehmen an, dass er in der Schicht V_i liegt. Zunächst verteilen wir $\mathrm{POT}(r)$ auf die Bögen, die aus r hinausgehen. Dies ist wegen $\mathrm{POT}(r) \leq \sum_{a \in \delta^+(r)} (\overline{c}(a) - x(a))$ möglich. Der zusätzliche Fluss $\mathrm{POT}(r)$ kommt also in der Schicht V_{i+1} an. Für jeden Knoten $v \in V_{i+1}$ verteilen wir die Flussmenge, die in v ankommt auf die Bögen, die aus v herausgehen. Aufgrund von (7.18) ist auch dies möglich. Damit gelangt der gesamte Wert $\mathrm{POT}(r)$ zur Schicht V_{i+2}, und wir fahren so fort bis der Fluss $\mathrm{POT}(r)$ den Zielknoten $t \in V$ erreicht. Das Verteilen von $\mathrm{POT}(r)$ auf die verschiedenen Bögen erfolgt durch einfache Listenverarbeitung (z. B. Breadth-First).

Nun gehen wir rückwärts, um aus s den Flusswert $\mathrm{POT}(r)$ "herauszuziehen". Wir verteilen den Flusswert $\mathrm{POT}(r)$ auf alle Bögen, die in r enden. Für alle Bögen $v \in V_{i-1}$ verteilen wir den Anteil des Flusswertes $\mathrm{POT}(r)$, der rückwärts in v angekommen ist, auf die Bögen, die in v enden. Die Wahl des Referenzknoten r garantiert, dass dies möglich ist. Und wir machen auf diese Weise weiter, bis wir zur Quelle s gelangt sind. Durch dieses "Durchschieben" und "Herausziehen" wird also der Wert des Ausgangsflusses in N um $\mathrm{POT}(r)$ erhöht. Wenden wir dieses Verfahren auf unser Beispielnetzwerk aus Abbildung 7.4 an, so erhalten wir den in Abbildung 7.5 dargestellten Fluss.

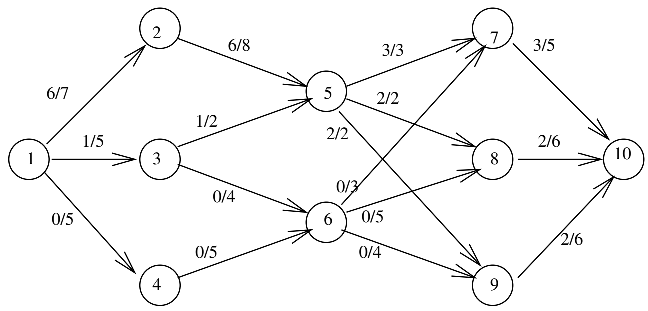


Abb. 7.5

Alle Bögen mit $x_a = \overline{c}_a$ sind saturiert. Ist der gegenwärtige (s,t)-Fluss in N nicht saturiert, muss es noch einen nicht saturierten (s,t)-Weg in N geben. Kein nicht saturierter (s,t)-Weg in N kann einen saturierten Bogen enthalten. Daher können wir alle saturierten Bögen aus N entfernen. Aufgrund unseres Vorgehens gilt für den Referenzknoten r, dass entweder alle Bögen aus $\delta^-(r)$ oder alle Bögen aus $\delta^+(r)$ saturiert sind (in unserem Beispiel aus Abbildung 7.5 sind alle Bögen aus $\delta^+(5)$ saturiert). Falls es also einen noch nicht saturierten (s,t)-Weg geben sollte, kann dieser den Knoten r nicht enthalten. Wir können also auch den Knoten r und alle weiteren Bögen, die r als Anfangs- oder Endknoten enthalten, aus N entfernen. Die Herausnahme von r und der saturierten Bögen kann bewirken, dass ein Knoten $v \in W$ auf keinem (s,t)-Weg in dem so reduzierten Digraphen liegt. (Dies gilt in Abbildung 7.5 nach Wegnahme der saturierten Bögen und des Knoten 5 für den Knoten 2.) Alle derartigen Knoten und die mit ihnen inzidenten Bögen können ebenfalls aus N entfernt werden. Wir entfernen Knoten und Bögen so lange, bis jede Menge $\delta^-(v)$ und $\delta^+(v)$ der noch verbleibenden Knoten v mindestens eine nicht saturierte Kante enthält. Werden s oder t entfernt, können wir aufhören, denn dann ist der gegenwärtige Fluss offenbar saturiert. Damit ist eine Iteration des Verfahrens zur Bestimmung eines saturierten (s,t)-Flusses beschrieben.

Diese Herausnahme von Knoten und Bögen aus N induziert natürlich eine Änderung der Knotenpotentiale der vorhandenen Knoten. Wir können somit mit dem reduzierten Netzwerk von neuem beginnen, durch (7.18) die Potentiale bestimmen und mit der obigen Methode fortfahren.

Das geschichtete Netzwerk, das aus dem Netzwerk aus Abbildung 7.5 durch die obige Reduktionsmethode entsteht, ist in Abbildung 7.6 dargestellt.

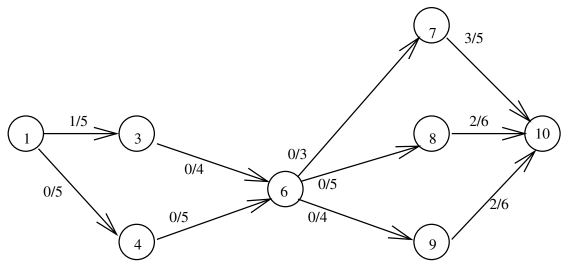


Abb. 7.6

Man beachte, dass die in Abbildung 7.6 angegebenen Flusswerte keinen zulässigen Fluss des Digraphen 7.6 darstellen. Fügen wir die aus dem Digraphen entfernten Knoten und Bögen mit den für sie bereits bestimmten Fluss werten hinzu, so erhalten wir einen zulässigen Fluss im ursprünglichen Digraphen aus Abbildung 7.5. Die Berechnung der Potentiale der Knoten aus Abbildung 7.6 ergibt

$$\mathrm{POT}(3) = 4,\ \mathrm{POT}(4) = 5,\ \mathrm{POT}(6) = 9,$$
$$\mathrm{POT}(7) = 2,\ \mathrm{POT}(8) = 4,\ \mathrm{POT}(9) = 4.$$

Als Referenzknoten müssen wir nun den Knoten 7 wählen und $\mathrm{POT}(7)$ aus $s = 1$ “herausziehen” bzw. in $t = 10$ “hineinschieben”.

Damit ist das Verfahren zur Bestimmung eines saturierten (s,t)-Flusses in geschichteten Netzwerken beschrieben. Wir starten zunächst mit dem Nullfluss, bestimmen die Potentiale aller Knoten $v \in V \setminus \{s,t\}$, wählen das kleinste Potential mit dem Referenzknoten r und schieben $\mathrm{POT}(r)$ durch das Netzwerk. Dann reduzieren wir das Netzwerk iterativ um alle saturierten Bögen (und einige Knoten) und beginnen von neuem. Dies führen wir so lange durch, bis es im gegenwärtigen reduzierten Digraphen keinen Weg von s nach t mehr gibt. Da bei jeder Reduktion mindestens ein Knoten entfernt wird, ist dies nach spätestens $n - 1$ Iterationen der Fall.

Bestimmung eines geschichteten Netzwerks

Wir wissen nun, wie in einem geschichteten Netzwerk ein saturierter (aber nicht notwendig maximaler) (s,t)-Fluss bestimmt werden kann. Diese Methode wollen wir uns zunutze machen, um in einem beliebigen Digraphen einen maximalen Fluss zu konstruieren. Dabei gehen wir in mehreren Stufen wie folgt vor.

Zunächst konstruieren wir aus dem gegebenen Digraphen D ein geschichtetes Netzwerk, sagen wir N. Vom Nullfluss ausgehend bestimmen wir einen saturierten (s,t)-Fluss x in N. Diesen (s,t)-Fluss x können wir (nach Konstruktion) in einen (s,t)-Fluss in D transformieren. Aus D und dem derzeitigen Fluss x konstruieren wir ein neues geschichtetes Netzwerk N' und bestimmen wiederum einen saturierten (s,t)-Fluss x' in N'. Den Fluss x' kann man (wie bei der Addition augmentierender Wege) zu x hinzufügen und erhält einen (s,t)-Fluss in D mit Wert $\mathrm{val}(x) + \mathrm{val}(x')$. Nun bestimmen wir aus D und dem neuen (s,t)-Fluss ein weiteres geschichtetes Netzwerk und fahren so fort, bis aus D und dem gegenwärtigen (s,t)-Fluss kein neues geschichtetes Netzwerk mehr konstruiert werden kann, das den Knoten t enthält. Die Regeln zur Konstruktion des geschichteten Netzwerkes aus D und dem gegenwärtigen Fluss sind so gestaltet, dass es dann zu D und dem gegenwärtigen (s,t)-Fluss keinen augmentierenden (s,t)-Weg mehr gibt. Aus Satz (7.9) folgt dann, dass der gegenwärtige (s,t)-Fluss maximal ist. Aus der letzten Bemerkung ist klar, wie wir bei der Auswahl der Bögen vorzugehen haben. Die Schichten des Netzwerkes werden nach dem Breadth-First-Prinzip bestimmt.

Ist also $D = (V,A)$ mit Bogengewichten $c(a)$ für alle $a \in A$ ein Digraph, sind s,t zwei verschiedene Knoten aus D, und ist x ein zulässiger (s,t)-Fluss, dann wird aus D, c und x ein geschichtetes Netzwerk $N = (W,F)$ mit Kapazitäten $\overline{c}(a)$ wie folgt konstruiert:

Es seien

(7.20)
$$\begin{aligned} A_1 &:= \{(u,v) \mid (u,v) \in A \text{ und } x_{uv} < c_{uv}\}, \\ A_2 &:= \{(v,u) \mid (u,v) \in A \text{ und } x_{uv} > 0\}, \end{aligned}$$

$$(7.21) \qquad \begin{array}{ll} \overline{c}_{uv} := c_{uv} - x_{uv} & \text{für alle } (u,v) \in A_1, \\ \overline{c}_{uv} := x_{vu} & \text{für alle } (u,v) \in A_2. \end{array}$$

Auf den Bögen aus A_1 kann der Fluss erhöht, auf denen aus A_2 erniedrigt werden. Ein Bogen kann sowohl in A_1 als auch in A_2 sein. Mit $A_1 \dot\cup A_2$ bezeichnen wir die "Vereinigung" von A_1 und A_2, bei der ein Bogen zweimal auftreten kann. Ist $(u,v) \in A$ mit $x_{uv} < c_{uv}$, so schreiben wir, wenn es für das Verständnis hilfreich ist, $(u,v)_1$, um den zu (u,v) gehörigen Bogen aus A_1 zu bezeichnen. Ist $(u,v) \in A$ mit $x_{uv} > 0$, so schreiben wir auch analog $(v,u)_2$, um den zugehörigen Bogen (v,u) aus A_2 zu bezeichnen.

Nun setzen wir $V_1 := \{s\}$. Ist V_i bestimmt, dann sei

$$(7.22) \qquad V_{i+1} := \{v \in V \setminus (V_1 \cup \ldots \cup V_i) \mid \exists\, u \in V_i \text{ mit } (u,v) \in A_1 \dot\cup A_2\}.$$

Sobald eine solche Knotenmenge, sagen wir V_k, den Knoten t enthält, brechen wir ab und setzen $V_k := \{t\}$ und $W' := \bigcup_{i=1}^k V_i$. Ferner sei

$$F' := (A_1 \cap \bigcup_{i=1}^{k-1} (V_i \times V_{i+1})) \dot\cup (A_2 \cap \bigcup_{i=1}^{k-1} (V_i \times V_{i+1})).$$

Der Digraph (W', F') ist ein geschichtetes Netzwerk mit den in (7.21) definierten Bogenkapazitäten. Man beachte, dass $\delta^-(v) \neq \emptyset$ für alle Knoten $v \in W' \setminus \{s\}$ gilt; es kann aber Knoten $v \in W' \setminus \{t\}$ geben mit $\delta^+(v) = \emptyset$. Solche Knoten können wir entfernen. Wir tun dies sukzessive und erhalten das "endgültige" geschichtete Netzwerk (W, F) durch Entfernen von Knoten und Bögen wie im vorigen Abschnitt angegeben.

Betrachten wir den in Abbildung 7.7 dargestellen kapazitierten Digraphen mit dem angegebenen (s,t)-Fluss.

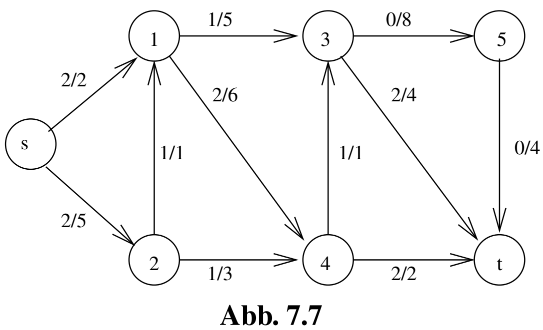


Abb. 7.7

In Abbildung 7.8 sind die in (7.20) definierten zugehörigen Bogenmengen A_1 und A_2 angegeben mit den durch (7.21) definierten Kapazitäten. Die Bögen aus A_2 sind gestrichelt gezeichnet, die aus A_1 mit durchgehenden Linien.

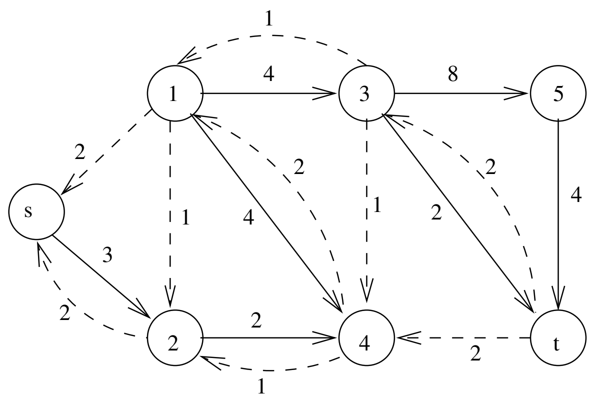


Abb. 7.8

Mit dem oben beschriebenen Verfahren erhalten wir nun das in Abbildung 7.9 dargestellte geschichtete Netzwerk mit 6 Schichten.

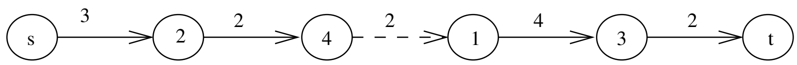


Abb. 7.9

Das geschichtete Netzwerk N ist so konstruiert, dass jeder zulässige (s,t)-Fluss in N eine Vereinigung augmentierender (s,t)-Wege in D ist, genauer:

(7.23) Satz. *Sei $D = (V, A)$ ein kapazitierter Digraph, und $N = (W, F)$ sei das oben aus D und dem zulässigen (s,t)-Fluss x konstruierte geschichtete Netzwerk mit den Kapazitäten $\overline{c}$, dann gilt:*

(a) *x ist ein maximaler (s,t)-Fluss in D genau dann, wenn $t \notin W$.*

(b) *Ist $\overline{x}$ ein zulässiger (s,t)-Fluss in N, dann ist $x' \in \mathbb{R}^A$ definiert durch*

$$x'_a := x_a + \overline{x}_{a_1} - \overline{x}_{a_2} \qquad \text{für alle } a = (u,v) \in A$$

ein zulässiger (s,t)-Fluss in D mit Wert $\mathrm{val}(x) + \mathrm{val}(\overline{x})$.

(Die Formel in (b) zur Berechnung von x'_a ist wie folgt zu interpretieren. Ist $a = (u,v) \in A$ und $0 < x_{uv} < c_{uv}$, so ist $a_1 = (u,v) \in A_1$ und $a_2 = (v,u) \in A_2$ und die Formel ist wohldefiniert. Ist $x_a = 0$, so ist $(v,u) \notin A_2$ und $\overline{x}_{vu}$ ist als Null zu betrachten. Ist $x_{uv} = c_{uv}$, so gilt $(u,v) \notin A_1$ und $\overline{x}_{uv}$ ist als Null zu betrachten.)

Beweis : (b) Wir müssen zeigen, dass x' die Kapazitäts- und die Flusserhaltungsbedingungen (7.1), (7.2) erfüllt. Sei $(u, v) \in A$ beliebig, dann gilt

$$\begin{array}{llll} 0 \le x_{uv} - \overline{x}_{vu}, & \text{da} & x_{uv} & = \overline{c}_{vu} \ge \overline{x}_{vu} \\ \le x_{uv} + \overline{x}_{uv} - \overline{x}_{vu}, & \text{da} & \overline{x}_{uv} & \ge 0 \\ = x'_{uv} & & & \\ \le x_{uv} + \overline{x}_{uv}, & \text{da} & \overline{x}_{vu} & \ge 0 \\ \le c_{uv}, & \text{da} & \overline{c}_{uv} & = c_{uv} - x_{uv} \ge \overline{x}_{uv}. \end{array}$$

Also ist (7.1) erfüllt. Sei nun $v \in V \setminus \{s, t\}$ ein beliebiger Knoten.

$$\begin{array}{ll} x'(\delta^-(v)) - x'(\delta^+((v)) & = x(\delta^-(v)) + \overline{x}(\delta^-(v) \cap A_1) - \overline{x}(\delta^+(v) \cap A_2) \\ & -x(\delta^+(v)) - \overline{x}(\delta^+(v) \cap A_1) + \overline{x}(\delta^-(v) \cap A_2) \\ & = (x(\delta^-(v)) - x(\delta^+(v))) + (\overline{x}(\delta^-(v)) - \overline{x}(\delta^+(v))) \\ & = 0 + 0 = 0. \end{array}$$

Der Wert von x' ist offenbar $\text{val}(x) + \text{val}(\overline{x})$.

(a) Jedem (s, t)-Weg in N entspricht ein augmentierender $[s, t]$-Weg in D. Nach Konstruktion gibt es in D einen augmentierenden $[s, t]$-Weg genau dann, wenn es in N einen (s, t)-Weg gibt. Die Behauptung folgt dann direkt aus (b) und Satz (7.9). □

Damit ist der DMKM-Algorithmus vollständig beschrieben. Unter Verwendung geeigneter Datenstrukturen kann man folgendes zeigen (siehe Mehlhorn (1984)):

(7.24) Satz. *Der DMKM-Algorithmus findet in einem kapazitierten Digraphen mit n Knoten und m Bögen einen maximalen (s, t)-Fluss in $0(n^3)$ Schritten.* □

Eine Implementation des DMKM-Algorithmus in PASCAL, in der alle oben informell dargestellten Details explizit ausgeführt sind, kann man in Syslo, Deo und Kowalik (1983) finden.

7.4 Ein generischer Präfluss–Algorithmus

In seiner Doktorarbeit aus dem Jahre 1987 und einigen z. T. vorher erschienenen gemeinsamen Veröffentlichungen mit anderen Autoren hat A. V. Goldberg einige neue und einige bekannte Ideen auf originelle Weise kombiniert und dabei die Grundlagen zu einer neuen Klasse von Maximalfluss–Algorithmen gelegt. In einer Vielzahl von Folgeveröffentlichungen verschiedener Autoren sind diese Ideen ergänzt und verfeinert worden, und die Laufzeiten der Algorithmen sind durch verbesserte Datenstrukturen verringert worden. Diese Algorithmen werden in der englischen Literatur häufig preflow-push- oder push-relabel-Algorithmen genannt. Wir nennen sie hier Präfluss–Algorithmen. Überblicke zu diesem Themenkreis finden sich u.a. in den Artikeln Ahuja et al. (1989) und

Goldberg et al. (1990) sowie in dem Buch Ahuja et al. (1993). Implementierungswettbewerbe in letzter Zeit deuten an, dass die besten Präfluss–Algorithmen bei großen Graphen in der Praxis schneller sind als die bisher bekannten Verfahren, wobei nicht unbedingt die bezüglich der worst-case-Laufzeit schnellsten Algorithmen auch in der praktischen Ausführung am schnellsten sind. Die Entwicklung – speziell hinsichtlich praktisch effizienter Implementierungen – ist noch nicht abgeschlossen.

Es hat sich aus Notationsgründen eingebürgert, Präfluss–Algorithmen etwas anders darzustellen als die übrigen Maximalfluss–Verfahren. Wir folgen hier diesem Brauch.

Wie immer ist ein Digraph $D = (V, A)$ mit nichtnegativen Bogenkapazitäten $c(a)$ gegeben. (Wir lassen wie üblich zu, dass $c(a)$ auch den Wert ∞ annehmen kann.) Wir verlangen, dass $D = (V, A)$ **keine parallelen Bögen** besitzt, also **einfach** ist, und ferner, dass D zu jedem Bogen (u, v) auch seinen "Gegenbogen" (v, u) enthält. Digraphen mit letzterer Eigenschaft heißen **symmetrisch**.

Wenn (wie üblich) ein beliebiger Digraph gegeben ist, so können wir ihn in diese Standardform wie folgt transformieren. Parallele Bögen ersetzen wir durch einen einzigen Bogen, wobei wir die entsprechenden Bogenkapazitäten aufaddieren. Ist zu einem Bogen (u, v) noch kein Gegenbogen (v, u) vorhanden, so fügen wir (v, u) zum gegenwärtigen Digraphen mit der Kapazität $c(v, u) = 0$ hinzu. Jedem Maximalfluss in diesem transformierten symmetrischen und einfachen Digraphen entspricht offensichtlich ein Maximalfluss im ursprünglichen Digraphen und umgekehrt.

Ab jetzt gehen wir daher in diesem Abschnitt davon aus, dass Digraphen symmetrisch und einfach sind.

Ein **Pseudofluss** auf $D = (V, A)$ ist eine Abbildung $x : A \to \mathbb{R}$ mit den folgenden Eigenschaften

(7.25) $$x(u, v) \leq c(u, v) \quad \text{für alle} \quad (u, v) \in A$$

(7.26) $$x(u, v) = -x(v, u) \quad \text{für alle} \quad (u, v) \in A.$$

In der **Kapazitätsbeschränkung** (7.25) verzichtet man hier auf die übliche Nichtnegativitätsschranke, weil die **Antisymmetrie-Bedingung** (7.26) für den Pseudofluss eine notationstechnische Erleichterung bringt. Hinter (7.26) steckt keine tiefsinnige Überlegung!

Ist ein Pseudofluss x gegeben, so definiert man eine **Überschuss -Abbildung** $e : V \to \mathbb{R}$ (e steht für "excess") durch

(7.28) $$e(v) := \sum_{a \in \delta^-(v)} x(a) \quad \text{für alle} \quad v \in V.$$

Beim Vorliegen mehrerer Pseudoflüsse, sagen wir x und x', schreibt man $e_x(v)$ bzw. $e_{x'}(v)$, um zu kennzeichnen bzgl. welches Pseudoflusses der Überschuss definiert ist. Ist

$e(v)$ nichtnegativ, so spricht man von einem **Überschuss** bei v, sonst von einem **Defizit**. Die **Residualkapazität** bezüglich x ist eine Abbildung $r := A \to \mathbb{R}$ definiert durch

$$(7.29) \qquad r(u,v) := c(u,v) - x(u,v) \quad \text{für alle} \quad (u,v) \in A.$$

Wiederum schreibt man $r_x(u,v)$, wenn man den betrachteten Pseudofluss hervorheben will. Der **Residualdigraph** bezüglich x ist der Digraph $D_x = (V, A_x)$ mit

$$A_x := \{(u,v) \in A \mid r_x(u,v) > 0\}.$$

Ein **(s,t)-Präfluss** oder kurz **Präfluss** ist ein Pseudofluss x, bei dem für alle Knoten $v \in V \setminus \{s,t\}$ der Überschuss $e_x(v)$ nichtnegativ ist. Die Flusserhaltungsbedingung (7.2) liest sich in der gegenwärtigen Notation $e(v) = 0$. Also ist ein (s,t)-Fluss ein Pseudo- (oder Prä-) Fluss mit $e(v) = 0$ für alle $v \in V \setminus \{s,t\}$.

Wir beschreiben nun einen generischen Präfluss–Algorithmus, in Anlehnung an Goldberg and Tarjan (1988).

(7.30) Definition. *Sei $D = (V,A)$ ein Digraph mit Kapazitäten $c(a) \geq 0$ für alle $a \in A$, seien $s,t \in V, s \neq t$, und sei x ein (s,t)-Präfluss . Eine* **Entfernungsmarkierung** *(distance labeling) ist eine Funktion $d : V \to \mathbb{Z}_+$ mit den Eigenschaften:*

$$\begin{aligned} d(t) &= 0, \quad d(s) = n = |V|, \\ d(u) &\leq d(v) + 1 \qquad \text{für alle Bögen } (u,v) \in A_x \text{ des Residualdigraphen } D_x. \end{aligned}$$

□

Hinter dieser Begriffsbildung liegt der folgende Gedanke. Wir setzen die "Länge" aller Bögen $a \in A_x$ des Residualdigraphen D_x mit 1 fest, nur die Länge von (s,t) wird n gesetzt. Für jeden Knoten $v \in V \setminus \{t\}$ können wir dann seine Entfernung $\text{dist}_{D_x}(v,t)$ zur Senke t berechnen, also die Länge des kürzesten (v,t)-Weges in D_x. Man kann zeigen, dass für alle $v \in V$ der Wert $d(v)$ eine untere Schranke dieser Enfernung $\text{dist}_{D_x}(v,t)$ ist.

Der generische Präfluss-Algorithmus hat zu jeder Zeit der Algorithmus-Ausführung einen Präfluss x und eine Entfernungsmarkierung d. Er schreibt x und d fort, wobei Schuboperationen (push) und Markierungsänderungen (relabel) vorgenommen werden. Um diese beschreiben zu können, treffen wir noch einige Definitionen. Wir nennen einen Knoten v **aktiv**, wenn $v \notin \{s,t\}$ und $e_x(v) > 0$ gilt. Ein Präfluss ist also ein Fluß, wenn kein Knoten aktiv ist. Ein Bogen heißt **erlaubt**, wenn $(u,v) \in A_x$ (also im Residualdigraphen) ist und $d(u) = d(v) + 1$ gilt.

Der generische Präfluss -Algorithmus beginnt mit dem Präfluss, der auf den Bögen $a \in \delta^+(s)$ den Wert $x(a) = c(a)$ und auf allen übrigen den Wert Null hat. Der Algorithmus führt dann in beliebiger Reihenfolge die Fortschreibungsoperationen aus. Formal kann man dies wie folgt beschreiben.

(7.31) Generischer Präfluss -Algorithmus.

Input: *(einfacher, symmetrischer) Digraph $D = (V,A)$ mit Kapazitäten $c(a) \geq 0$ für alle $a \in A$ und $s,t \in V, s \neq t$.*

Output: *maximaler* (s,t)*-Fluss* x.

Inititialisierung:

$$
\begin{array}{rll}
\textit{Setze} \quad x(s,v) := c(s,v) & \textit{für alle} & (s,v) \in \delta^+(s), \\
x(v,s) := -c(s,v) & \textit{für alle} & (v,s) \in \delta^-(s), \\
x(a) := 0 & \textit{für alle} & a \in A \setminus (\delta^+(s) \cup \delta^-(s)). \\
\textit{Setze} \quad e(v) := \sum_{a\in\delta^-(v)} x(a) & \textit{für alle} & v \in V, \\
d(s) := n, & & \\
d(v) := 0 & \textit{für alle} & v \in V \setminus \{s\}.
\end{array}
$$

Schleife: *Solange es noch* **aktive** *Knoten gibt, wähle einen aktiven Knoten* u.
Gibt es noch erlaubte Bögen mit Anfangsknoten u*, wähle einen solchen Bogen* (u,v) *und führe PUSH* (u,v) *aus; gibt es keinen erlaubten Bogen mit Anfangsknoten* u*, führe RELABEL* (u) *aus.*

RELABEL (u): *Ersetze* $d(u)$ *durch* $\min\{d(v) \mid (u,v) \in A_x\} + 1$.

PUSH (u,v): *Erhöhe den Fluss auf dem Bogen* (u,v) *um den Wert* δ*, wobei* $0 \le \delta \le \min\{e_x(u), r_x(u,v)\}$ *gelten muss.* □

Die PUSH-Operation schiebt also zusätzlichen Fluss durch einen Bogen. Im allgemeinen, und so machen wir es hier auch, wird man natürlich den Fluss um den größtmöglichen Wert $\delta := \min\{e(v), r(u,v)\}$ erhöhen; es gibt jedoch Skalierungsalgorithmen, bei denen kleinere Werte gewählt werden. Falls $\delta = r(u,v)$, so nennt man einen solchen Schub **saturierend**. Nach einem saturierenden Schub ist ein Bogen nicht mehr erlaubt, da der Fluss durch ihn die Kapazitätsgrenze erreicht hat und er aus dem Residualgraphen entfernt wird.

Die Entfernungsmarkierung von u wird dann erhöht, wenn u kein erlaubter Bogen verlässt, weil jeder Bogen $(u,v) \in A$ entweder keine Restkapazität mehr hat oder die Entfernungsmarken der Nachfolger von u zu hoch sind. Das Ziel der RELABEL-Operation ist, mindestens einen weiteren erlaubten Bogen zu erzeugen.

Der generische Präfluss-Algorithmus muss keineswegs mit der Entfernungsmarkierung $d(v) = 0$ für alle $v \in V \setminus \{s\}$ beginnen. In der Praxis hat es sich als außerordentlich nützlich erwiesen, die Werte $d(v)$ so gut wie möglich mit $\text{dist}_{D_x}(v,t)$ in Übereinstimmung zu bringen. Dies ist zwar rechnerisch teuer, lohnt sich jedoch. Man setzt zu Beginn einfach $d(v) = \text{dist}(v,t)$ für alle $v \in V \setminus \{s\}$ (durch breadth first search) fest. Dann schreibt man $d(v)$ durch die RELABEL-Operation fort. Man sollte aber $d(v)$ periodisch neu berechnen, um die Werte wieder in Einklang mit den Distanzen im gegenwärtigen Residualgraphen zu bringen.

Die Initialisierung erledigt verschiedene wichtige Aufgaben. Zunächst wird jeder Nachfolger von s mit einem positiven Überschuss versehen, so dass der Algorithmus mit einem aktiven Knoten beginnen kann. Ferner ist nach der Initialisierung keiner der Bögen aus

$\delta^+(s)$ erlaubt, da alle PUSH-Operationen saturierend waren. Da alle (v,t)-Wege in D_x höchstens die Länge $n-1$ haben, wird durch $d(s) = n$ sichergestellt, dass $d(u) \leq d(s)+1$ für alle $(u,s) \in A_x$ gilt. Man beachte, dass D_x keinen gerichteten (s,t)-Weg enthält und damit $d(s) = n$ auch eine untere Schranke für $\text{dist}_{D_x}(s,t)$ ist. Da die Entfernungsmarkierungen im Verlaufe des Verfahrens nicht fallen, bleibt garantiert, dass D_x während der Ausführung des Präfluss -Algorithmus nie einen gerichteten (s,t)-Weg enthalten wird. Es wird also nie mehr nötig sein, aus s Fluss herauszuschieben.

Es ist für das Verständnis des Vorgehens sehr hilfreich, sich den Präfluss -Algorithmus anhand eines Rohrleitungssystems vorzustellen. Die Bögen des gegebenen Graphen repräsentieren Rohre, die flexible Verbindungsstücke haben und bewegt werden können. Die Knoten stellen die Rohrverbindungen dar. Eine Entfernungsmarkierung misst, wie weit ein Knoten vom Boden entfernt ist. In diesem Netzwerk wollen wir Wasser von der Quelle zur Senke schicken, wobei Wasser in den Rohren jeweils nur abwärts fließen kann in Richtung der Senke. Gelegentlich gerät der Fluss in eine "lokale Senke", nämlich dann, wenn ein Knoten keinen Nachbarn hat, der tiefer liegt. In diesem Falle heben wir (durch ein RELABEL) den Knoten (auf eine Ebene höher als sein niedrigster Nachbar) an, und das Wasser kann wieder abfließen. Da wir die Knoten immer weiter anheben, fließt der verbleibende Überschuss (der die Senke nicht mehr erreichen kann) an den inneren Knoten des Netzwerkes irgendwann zurück zur Quelle. Der Algorithmus endet, wenn die maximale Wassermenge von der Quelle zur Senke fließt und kein Knoten aus $V \setminus \{s,t\}$ mehr Überschuss hat, also ein "richtiger" Fluss fließt.

(7.32) Beispiel. Wir betrachten das in Abb. 7.10 (a) angegebene Netzwerk mit den bei den Bögen aufgelisteten Kapazitäten. Dieser Digraph ist symmetrisch, aber die jeweiligen "Gegenpfeile" sind aus Übersichtlichkeitsgründen nicht gezeichnet. Diese haben alle die Kapazität 0.

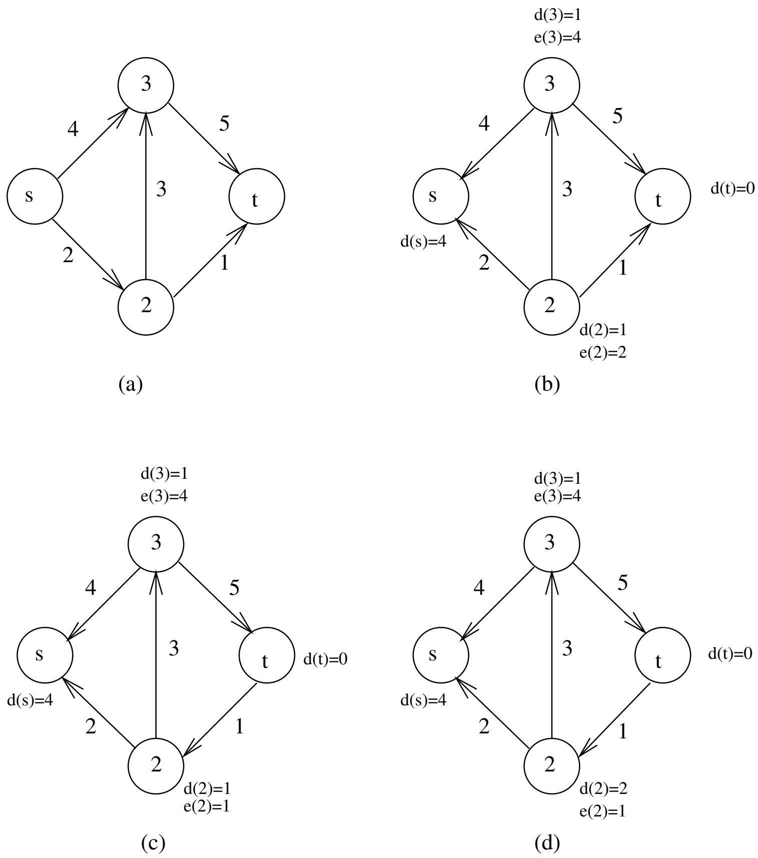


Abb. 7.10

Wir führen mit dem Netzwerk aus Abb. 7.10 (a) den Initialisierungsschritt aus. Daraus ergibt sich der in Abb. 7.10 (b) gezeichnete Residualgraph mir den angedeuteten Überschüssen $e(v)$ und den durch Kürzeste-Wege-Berechnung bestimmten Entfernungsmarkierungen $d(v)$. Die Knoten 2 und 3 sind aktiv. Wir wählen Knoten 2 aus. Da der Bogen $(2,t)$ eine Restkapazität von 1 hat und da $d(2) = d(t) + 1$ ist, ist der Bogen $(2,t)$ erlaubt. Da der Überschuss von Knoten 2 den Wert 2 hat, können wir einen Fluss vom Wert $\delta = \min\{r(2,t), e(2)\} = \min\{1,2\} = 1$ durch den Bogen $(2,t)$ schieben. Diese Fluss erhöhung reduziert den Überschuss von Knoten 2 auf 1, der Bogen (2,4) wird, da der Fluss durch ihn die Kapazitätsgrenze erreicht hat (der Schub also saturierend war) aus dem Residualgraphen entfernt, der Gegenbogen (t,2) wird mit Kapazität 1 hinzugefügt. Der neue Residualgraph ist in Abb. 7.10 (c) gezeigt. Der Knoten 2 ist immer noch aktiv. Wir können ihn also auswählen. Die Bögen $(2,s)$ und (2,3) haben positive Restkapazität, aber die Entfernungsbedingung für die Bögen (u,i) (d.h., $d(u) = d(i)+1$) ist nicht erfüllt. Die Bögen sind also nicht erlaubt. Wir müssen daher ein RELABEL machen; damit erhält der Knoten 2 die neue Entfernungsmarkierung $d(2) := \min\{d(3), d(s)\} + 1 = \min\{1,4\} + 1 = 2$, siehe Abb. 7.10 (d). □

Der Korrektheitsbeweis für den generischen Präfluss -Algorithmus basiert auf den folgenden Überlegungen. (Aus Zeitgründen geben wir die Beweise nicht vollständig an und verweisen auf die Übersichtsartikel Ahuja et al. (1989) und Goldberg et al. (1990) und die dort zitierte Originalliteratur.)

Wir gehen im weiteren davon aus, dass ein einfacher symmetrischer Digraph $D = (V, A)$ mit Bogenkapazitäten $c(a) \geq 0$ und zwei Knoten $s, t \in V, s \neq t$ gegeben sind und dass $n = |V|$ und $m = |A|$ gilt.

(7.33) Lemma. *In jedem beliebigen Stadium der Ausführung des generischen Präfluss -Algorithmus mit zugehörigem Präfluss x gibt es von jedem Knoten u mit positivem Überschuss $e_x(u) > 0$ einen gerichteten Weg von u zur Quelle s im Residualgraphen D_x.* □

Dieses Lemma impliziert, dass bei jeder RELABEL-Operation das Minimum nicht über der leeren Menge gebildet wird. Die nächsten Aussagen sind für die Laufzeit-Analyse von Bedeutung.

(7.34) Lemma. *In jedem Stadium der Ausführung des generischen Präfluss -Algorithmus gilt $d(v) \leq 2n - 1$ für jeden Knoten $v \in V$.* □

(7.35) Lemma.

(a) *Die Entfernungsmarkierung eines Knoten wird höchstens $(2n - 1)$-mal erhöht.*

(b) *Insgesamt werden höchstens $2n^2 - 1$ RELABEL-Operationen ausgeführt.*

(c) *Die Anzahl der saturierenden PUSH-Operationen ist höchstens nm.*

(d) *Die Anzahl der nichtsaturierenden PUSH-Operationen ist höchstens $2n^2m$.*

(7.36) Satz. *Der generische Präfluss-Algorithmus endet mit einem maximalen Fluss. Er benötigt $\mathcal{O}(n^2m)$ Fortschreibungsoperationen. Mit geeigneten Datenstrukturen ist das in $\mathcal{O}(n^2m)$ arithmetischen Operationen durchführbar.*

Es gibt eine Vielzahl von Arbeiten zur Laufzeitverbesserung des generischen Präfluss -Algorithmus. Man kann solche z.B. durch eine geschickte Auswahl der aktiven Knoten und die Benutzung geeigneter Datenstrukturen erreichen. Wird z.B. in der Schleife immer der Knoten mit dem höchsten Überschuss gewählt, so reduziert sich die Laufzeit auf $\mathcal{O}(n^3)$. Eine trickreiche Analyse dieses Verfahrens zeigt sogar, dass die Laufzeit $\mathcal{O}(n^2\sqrt{m})$ beträgt. Mit Skalierungsmethoden kann man eine Laufzeitreduktion auf $\mathcal{O}(n^2 \log U)$ bzw. $\mathcal{O}(nm+n^2\sqrt{\log U})$ erreichen, wobei $U := \max\{c(u, v) \mid (u, v) \in A\}$.

Ein detaillierteres Eingehen auf diese "Tricks" würde den Rahmen der Vorlesung sprengen. Wir verweisen hierzu auf die beiden mehrfach erwähnten Übersichtsartikel, das Buch von Ahuja et al. (1993), Schrijver (2003) und die dort zitierte Originalliteratur.

Es sei noch auf einen fundamentalen Unterschied zwischen den Präfluss-Algorithmen und den anderen bisher besprochenen Maximalfluss-Algorithmen hingewiesen. Der Ford-Fulkerson und der DMKM-Algorithmus finden zunächst einen maximalen (s,t)-Fluss, dessen Maximalität sie durch Bestimmung eines (s,t)-Schnittes mit gleicher Kapazität nachweisen. Der Präfluss -Algorithmus findet "unterwegs" einen kapazitätsminimalen (s,t)-Schnitt (Überlegen Sie sich, wann das der Fall ist.) und muss anschließend durch Rückflussoperationen aus dem Präfluss einen Fluss machen. Ist man nur an der Bestimmung eines minimalen (s,t)-Schnittes interessiert, so kann man sich das "Rückschieben" sparen.

7.5 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 (2.9) 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 Maximalfluss verfahrens 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 7.11 an einem Beispiel dargestellt.

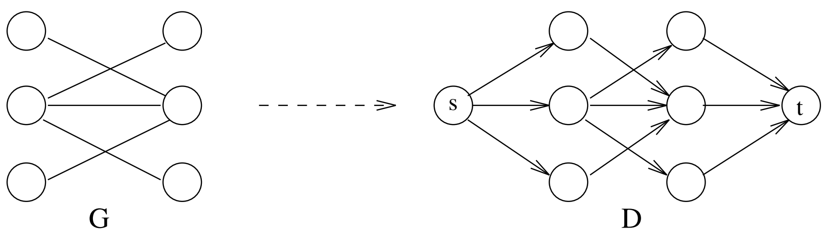


Abb. 7.11

(7.25) Satz. *Ist G ein bipartiter Graph und D der wie oben angegeben aus G konstruierte Digraph, dann ist der Wert eines maximalen (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 Maximalfluss algorithmen 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 Maximalfluss probleme 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 $S = \{s_1, \ldots, s_p\} \subseteq V$ Quellen und $T = \{t_1, \ldots, t_q\} \subseteq V$ 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

$$\begin{array}{ll} 0 \leq x_a \leq c_a & \text{für alle } a \in A \\ x(\delta^-(v)) = x(\delta^+(v)) & \text{für alle } v \in V \setminus (S \cup T). \end{array}$$

Der Wert eines zulässigen (S, T)-Flusses x ist definiert als

$$\mathrm{val}(x) := \sum_{s \in S} (x(\delta^+(s)) - x(\delta^-(s))).$$

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

$$\begin{array}{ll} A' & := A \cup \{(s, s_i) \mid i = 1, \ldots, p\} \cup \{(t_i, t) \mid i = 1, \ldots, q\} \\ c'(a) & := c(a) \;\; \text{für alle } a \in A \\ c(a) & := M \;\; \text{für alle } a \in A' \setminus A. \end{array}$$

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.

Literaturverzeichnis

Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1989). *Network Flows, Handbooks in Operations Research and Management Science*, volume 1, chapter Optimization, pages 211–360. Elsevier, North-Holland, Amsterdam, G. L. Nemhauser, A. H. G. Rinnooy Kan and M. J. Todd edition.

Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1993). *Network Flows, Theory, Algorithms and Applications*. Pearson Education, Prentice Hall, New York, first edition.

Ball, M. O., Magnanti, T. L., Monma, C. L., and Nemhauser, G. L. (1995a). *Handbooks in Operations Research and Management Science*, volume 7: Network Models. North-Holland, Amsterdam.

Ball, M. O., Magnanti, T. L., Monma, C. L., and Nemhauser, G. L. (1995b). *Handbooks in Operations Research and Management Science*, volume 8: Network Routing. North-Holland, Amsterdam.

Dinic, E. A. (1970). Algorithm for solution of a problem of maximal flow in a network with power estimation. *Soviet Math. Dokl.*, 11:1277–1280.

Edmonds, J. and Karp, R. M. (1972). Theoretical improvement in algorithmic efficiency of network flow problems. *J. ACM*, 19:248–264.

Elias, P., Feinstein, A., and Shannon, C. E. (1956). Note on maximum flow through a network. *IRE Trans. on Inform. Theory*, 2:117–119.

Ford Jr., L. R. and Fulkerson, D. R. (1956). Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404.

Ford Jr., L. R. and Fulkerson, D. R. (1962). *Flows in Networks*. Princeton University Press, Princeton.

Frank, A. (1995). Connectivity and Network Flows. In R. L. Graham et al. (Hrsg.), editor, *Handbook of Combinatorics*, chapter 2, pages 111–177. North-Holland, Amsterdam.

Goldberg, V. (1987). *Efficient graph algorithms for sequential and parallel computers*. PhD thesis, Laboratory for Computer Science, M.I.T., Cambridge. erhältlich als Technical Report TR-374.

Goldberg, V., Tardos, E., and Tarjan, R. E. (1990). Network Flow Algorithms. In B. Korte et al., editor, *Paths, Flows, and VLSI-Layout*. Springer-Verlag, Berlin.

Goldberg, V. and Tarjan, R. E. (1988). A new approach to the maximum flow problem. *J. Assoc. Comput. Mach*, 35(4):921–940.

Malhorta, V. M., Kumar, M. P., and Maheshwari, N. (1978). An $0(|V|^3)$ algorithm for finding the maximum flows in networks. *Inform. Process. Lett.*, 7:227–278.

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

Schrijver, A. (2003). *Combinatorial Optimization – Polyhedra and Efficiency*, volume A,B,C. Springer-Verlag, Berlin.

Syslo, M. M., Deo, N., and Kowalik, J. S. (1983). *Discrete Optimization Algorithms (with PASCAL programs*. Prentice Hall, Englewood Cliffs, N.J.

Kapitel 8

Weitere Netzwerkflussprobleme

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 7 erwähnten Bücher und Übersichtsartikel verwiesen.

8.1 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 (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 (7.5) des Maximalflussproblems kann man ein Minimalkosten-Flussproblem als lineares Programm darstellen. Offenbar ist jede Optimallösung des linearen Programms (8.1) ein kostenminimaler c-kapazitierter

(s, t)-Fluss mit Wert f.

$$
(8.1) \qquad \begin{array}{lll}
\min \sum_{a \in A} w_a x_a & & \\
x(\delta^-(v)) - x(\delta^+(v)) & = 0 & \forall v \in V \setminus \{s,t\} \\
x(\delta^-(t)) - x(\delta^+(t)) & = f & \\
0 \leq x_a & \leq c_a & \forall a \in A
\end{array}
$$

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 (8.1). Sie werden Netzwerk-Simplexalgorithmen genannt. Sie nutzen u.a. die Tatsache aus, dass die Basen von (8.1) durch aufspannende Bäume im Digraphen D repräsentiert werden können. Alle numerischen Unterprogramme wie Basis-Updates, Berechnung von reduzierten Kosten etc. können daher sehr effizient durch einfache kombinatorische Algorithmen erledigt werden.

Ein von A. Löbel (Konrad-Zuse-Zentrum, Berlin) implementierter Code dieser Art, es handelt sich um einen sogenannten primalen und dualen Netzwerk-Simplex-Algorithmus, ist auf dem ZIB-Server für akademische Nutzung verfügbar (`http://www.zib.de/Optimization/Software/Mcf/`). Mit diesem Code namens MCF können Minimalkosten-Flussprobleme mit mehreren tausenden 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 CPU2000 Suite aufgenommen worden, mit der Leistungsevaluierungen moderner Computersysteme vorgenommen werden, s. `http://spec.org`.

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 (8.1) zu produzieren. Ein Ende ist nicht abzusehen. Es gibt (zur Zeit) 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 (8.1) 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 lediglich 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 (7.8)). 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.

(8.2) 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. (Lemma (8.6) beweist die Rückrichtung.) Gibt es einen augmentierenden Kreis C bezüglich x, so setzen wir:

$$\varepsilon := \min \begin{cases} c_{ij} - x_{ij} & (i,j) \in C \text{ Vorwärtsbogen,} \\ x_{ij} & (i,j) \in C \text{ Rückwärtsbogen.} \end{cases} \tag{8.3}$$

Definieren wir

$$x'_{ij} := \begin{cases} x_{ij} + \varepsilon & \text{falls } (i,j) \in C \text{ Vorwärtsbogen,} \\ x_{ij} - \varepsilon & \text{falls } (i,j) \in C \text{ Rückwärtsbogen,} \\ x_{ij} & \text{falls } (i,j) \in A \setminus C, \end{cases} \tag{8.4}$$

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. □

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

$$A_1 := \{a \in A \mid x_a < c_a\}, \quad A_2 := \{(v,u) \mid (u,v) \in A \text{ und } x_{uv} > 0\}.$$

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 $\overline{a} \in \overline{A}$

$$\overline{c}(\overline{a}) \quad := \begin{cases} c(a) - x(a) & \text{falls } \overline{a} = a_1, \\ x(a) & \text{falls } \overline{a} = a_2, \end{cases}$$
$$\overline{w}(\overline{a}) \quad := \begin{cases} w(a) & \text{falls } \overline{a} = a_1, \\ -w(a) & \text{falls } \overline{a} = a_2. \end{cases}$$

In Abbildung 8.1 (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 8.2 (b) gezeichnet. Die beiden Zahlen bei einem Bogen a in 8.1 (b) geben an: Kapazität von a / Kosten von a.

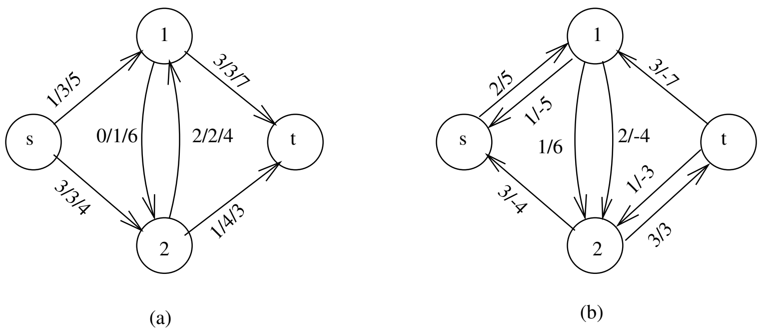


Abb. 8.1

Der Kreis $\{(2,t),(1,t),(2,1)\}$ in Abbildung 8.1 (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:

(8.5) 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 (8.2) unter Benutzung dieses neuen Konzepts um.

(8.6) 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 (8.2), 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

$$\overline{x}(\overline{a}) := \begin{cases} \max\{0, x'(a) - x(a)\}, & \text{falls } \overline{a} = a_1, \\ \max\{0, x(a) - x'(a)\}, & falls \overline{a} = a_2, \end{cases}$$

d. h., es gilt $\overline{x}(a_1) - \overline{x}(a_2) = x'(a) - x(a)$. 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:

$$\begin{aligned} \textstyle\sum_{\overline{a} \in \overline{A}} \overline{w}(\overline{a})\overline{x}(\overline{a}) &= \textstyle\sum_{a \in A} (\overline{w}(a_1)\overline{x}(a_1) + \overline{w}(a_2)\overline{x}(a_2)) \\ &= \textstyle\sum_{a \in A} (x'(a) - x(a))w(a) \\ &= w^T x' - w^T x < 0. \end{aligned}$$

Der Vektor $\overline{x}$ erfüllt trivialerweise 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.

$$\begin{aligned} \overline{x}(\delta_N^+(v)) - \overline{x}(\delta_N^-(v)) &= \textstyle\sum_{a \in \delta^+(v)} (\overline{x}(a_1) - \overline{x}(a_2)) - \sum_{a \in \delta^-(v)} (\overline{x}(a_1) - \overline{x}(a_2)) \\ &= \textstyle\sum_{a \in \delta^+(v)} (x'(a) - x(a)) - \sum_{a \in \delta^-(v)} (x'(a) - x(a)) \\ &= \Big(\textstyle\sum_{a \in \delta^+(v)} x'(a) - \sum_{a \in \delta^-(v)} x'(a)\Big) - \\ &\quad \Big(\textstyle\sum_{a \in \delta^+(v)} x(a) - \sum_{a \in \delta^-(v)} x(a)\Big) \\ &= \begin{cases} 0 - 0 & \text{falls } v \in V \setminus \{s,t\} \\ \text{val}(x') - \text{val}(x) & \text{falls } v = s \\ -\text{val}(x') + \text{val}(x) & \text{falls } v = t \end{cases} \\ &= 0. \end{aligned}$$

Daraus folgt, dass $\overline{x}$ die Flußerhaltungsbedingungen erfüllt und dass $\text{val}(\overline{x}) = 0$ 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 jedes $i \in \{1, \ldots, k\}$ 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 : Wir führen eine Induktion über die Anzahl p der Bögen $\overline{a} \in \overline{A}$ mit $\overline{x}(\overline{a}) \neq 0$ durch. Ist $p = 0$, so ist nichts zu zeigen. Sei also $p \geq 1$, und 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

$$
\begin{array}{ll}
x_p(\overline{a}) := \alpha_p & \text{für alle } \overline{a} \in \overline{C}_p, \\
x_p(\overline{a}) := 0 & \text{sonst,}
\end{array}
$$

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. Nach Induktionsvoraussetzung gilt die Behauptung für $\overline{x}_p$, und somit gilt sie nach Konstruktion auch für $\overline{x}$. Somit ist Behauptung 2 bewiesen.

Damit können wir den Beweis von (8.6) 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 (8.2), bzw. Satz (8.6) 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.

(8.7) 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*

$$
x'(a) := x(a) + \overline{x}(a_1) - \overline{x}(a_2) \qquad \text{für alle } a \in A
$$

ein zulässiger (s,t)-Fluss in D mit Wert $f + \operatorname{val}(\overline{x})$, 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

$$\begin{array}{ll} \overline{w}(P) + w'(C') & \geq \overline{w}(Q) + w'(K') \\ |C' \cap B'| & > |K' \cap B'|. \end{array}$$

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 $C'_1, \ldots, C'_k \subseteq \overline{A}$ mit einem gerichteten (s,t)-Weg $Q \subseteq \overline{A}$, und P' ist die Vereinigung von gerichteten Kreisen $C'_{k+1}, \ldots, C'_r \subseteq A'$. Da P kostenminimal in N ist, gilt $\overline{w}(P) \leq \overline{w}(Q)$, und somit gibt es wegen $\overline{w}(\overline{P}) + w'(P') = \overline{w}(Q) + \sum_{i=1}^{k} \overline{w}(C'_i) + \sum_{i=k+1}^{r} w'(C'_i)$ 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 $C'_1, \ldots, C'_k$. 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.

(8.8) Algorithmus.

Input: *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.*

Output: *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 (6.9)) 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 $\operatorname{val}(x) = f$, STOP, gib x aus.*

6. *Bestimme mit einem Kürzeste-Wege-Algorithmus (z. B. einer der Varianten des Moore-Bellman-Verfahrens (6.6), 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 $\varepsilon' := \min\{\overline{c}(\overline{a}) \mid \overline{a} \in P\}$, $\varepsilon := \min\{\varepsilon', f - \mathrm{val}(x)\}$, setze für $a \in A$

$$x(a) := \begin{cases} x(a) + \varepsilon & \text{falls } a_1 \in P \\ x(a) - \varepsilon & \text{falls } a_2 \in P \\ x(a) & andernfalls, \end{cases}$$

   konstruiere das bezüglich x und D augmentierende Netzwerk $N = (V, \overline{A}, \overline{c}, \overline{w})$ und gehe zu 5. □

Die Korrektheit des Algorithmus folgt unmittelbar aus (8.6) und (8.7). 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.

(8.9) 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 (8.8) 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 $\langle f \rangle$ 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 (8.8) 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, Magnanti und Orlin verwiesen, die auch ausführlich auf die historische Entwicklung eingehen. Der Aufsatz Shigeno, Iwata und McCormick (2000) präsentiert zwei neue Skalierungsmethoden und gibt dabei eine gute Vergleichsübersicht über die meisten der bekannten Min-Cost-Flow-Algorithmen.

8.2 Netzwerke mit Flussmultiplikatoren

Gelegentlich werden Flüsse in Netzwerken nicht konserviert, es können Verluste (Sickerverluste bei Wasserleitungen durch undichte Stellen) oder Umwandlungen (Geldtausch, chemische Prozesse) auftreten. In solchen Fällen kann man diese Verluste oder Transformationen durch sogenannte "Multiplikatoren" berücksichtigen bzw. modellieren. Dies geschieht dadurch, dass man jedem Bogen $a \in A$ nicht nur eine Kapazität $c(a)$ und einen Kostenkoeffizienten $w(a)$, sondern noch eine Zahl $m(a)$, den **Flussmultiplikator**, zuordnet. Bei den Zwischenknoten bekommt dann die Flusserhaltungsbedingung die folgende Form:

$$\sum_{a \in \delta^+(v)} x(a) - \sum_{a \in \delta^-(v)} m(a)x(a) = 0 \quad \forall\, v \in V \setminus \{s,t\},$$

und für die Quelle s bzw. die Senke t gilt:

$$\begin{array}{llr} \sum_{a \in \delta^+(s)} x(a) & - \sum_{a \in \delta^-(s)} m(a)x(a) = & f_s, \\ \sum_{a \in \delta^+(t)} x(a) & - \sum_{a \in \delta^-(t)} m(a)x(a) = & -f_t. \end{array}$$

Beim klassischen Maximalflußproblem gilt $m(a) = 1$ für alle $a \in A$, woraus $f_s = f_t$ folgt, hier jedoch gilt i. a. $f_s \neq f_t$. Der Betrag $f_s - f_t$ wird üblicherweise **Verlust des Flusses** x genannt.

Es gibt nun verschiedene Optimierungsfragen. Z. B. kann man bei gegebenem Wert f_s den Wert f_t maximieren oder bei gegebenem Wert f_t den Wert f_s minimieren oder $f_t - f_s$ maximieren. Analog kann man natürlich auch bei gegebenen Kostenkoeffizienten $w(a)$, für alle $a \in A$, einen kostenminimalen Fluss bestimmen, wobei entweder f_s oder f_t vorgegeben sind. Ist letzteres der Fall, so erhalten wir ein LP der folgenden Form

$$\begin{array}{ll} \min & \sum_{a \in A} w(a)x(a) \\ & \sum_{a \in \delta^+(v)} x(a) - \sum_{a \in \delta^-(v)} m(a)x(a) = \begin{cases} 0 & \text{falls } v \in V \setminus \{s,t\}, \\ -f_t & \text{falls } v = t, \end{cases} \\ & 0 \leq x_a \leq c(a) \quad \text{für alle } a \in A. \end{array}$$

Ein interesantes Devisenspekulationsproblem läßt sich als Anwendung des Maximalflussproblems mit Flussmultiplikatoren betrachten. Wir wollen auf dem Devisenmarkt Währungen ineinander umtauschen und dabei einen Gewinn erzielen. Unser Plan ist der folgende. Wir starten mit einer noch zu bestimmenden Summe an DM, tauschen diese in Fremdwährungen um, diese möglicherweise mehrmals weiter um, und zum Schluss tauschen wir alles wieder in DM zurück. Von Ihren Urlaubsreisen wissen Sie vermutlich, dass Sie dabei fast immer Geld verlieren. Vielleicht aber kann man es mit Mathematik etwas besser machen. In den Tabellen 8.1 und 8.2 sind die Umtauschkurse vom 16.09.1994 bzw. 16.12.1994 (Quelle: Handelsblatt) zwischen 6 wichtigen Währungen aufgelistet. Kann man mit diesen Kursen die angestrebte "wunderbare Geldvermehrung" bewerkstelligen?

	DM	£	US *sl*\$	SF	Yen	FF
DM	–	0.4103	0.6477	0.8297	64.0544	3.4171
£	2.4372	–	1.5785	2.0221	156.1136	8.3282
US *sl*\$	1.5440	0.6335	–	1.2810	98.9000	5.2760
SF	1.2053	0.4945	0.7806	–	77.2053	4.1187
1000 Yen	15.6117	6.4056	10.1112	12.9525	–	53.3468
10 FF	2.9265	1.2007	1.8954	2.4280	187.4526	–

Wechselkurse vom 16.09.1994 *(Handelsblatt)*
Tabelle 8.1

	DM	£	US *sl*\$	SF	Yen	FF
DM	–	0.4073	0.6365	0.8459	63.7770	3.4476
£	2.4552	–	1.5627	2.0768	156.5825	8.4644
US *sl*\$	1.5707	0.6399	–	1.3285	100.1800	5.4151
SF	1.1822	0.4815	0.7524	–	75.3950	4.0756
1000 Yen	15.6796	6.3864	9.9800	13.2635	–	54.0569
10 FF	2.9006	1.1814	1.8462	2.4536	184.9903	–

Wechselkurse vom 16.12.1994 sl(Handelsblatt)

Tabelle 8.2

Wir machen zunächst einige zusätzliche Annahmen. Wir gehen in unserem Beispiel davon aus, dass der Umtausch kursneutral erfolgt, dass also keine Gebühren anfallen. Diese Annahme ist natürlich nur für Banken bzw. Devisenhändler erfüllt (falls überhaupt). Da jedoch Gebühren i. a. proportial zur Umtauschsumme anfallen, kann man diese durch Abschläge im Kurs auf direkte Weise berücksichtigen. Wir probieren nun eine Umtauschsequenz anhand der Tabellen 8.1 und 8.2 aus. Diese ist in Abbildung 8.2 dargestellt. Wir starten mit DM 1 Million, tauschen dann in US \$, dann in französische Franc und wieder zurück in DM.

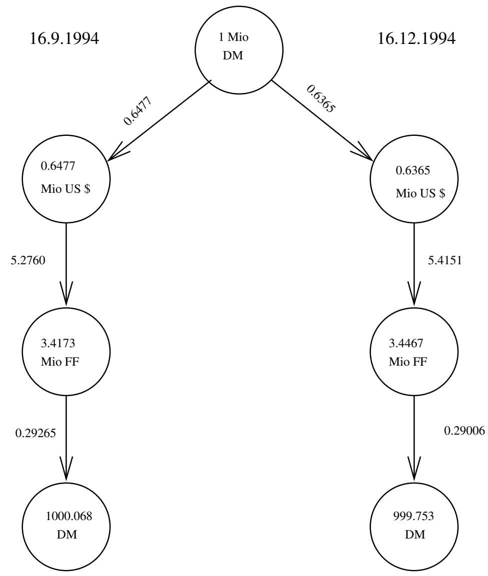


Abb. 8.2

Am 16.12.1994 ergibt sich dadurch ein Verlust von DM 247,– oder rund 0,02 % des eingesetzten Betrags, während sich am 16.09.1994 ein Gewinn von DM 63,– einstellt. Durch Erhöhung des eingesetzten Betrags kann man in unserem Modell den Gewinn beliebig erhöhen. Jedoch haben wir dabei nicht berücksichtigt, dass erhöhte Nachfrage bzw. erhöhtes Angebot zu Kursänderungen führt. In der Praxis ist es so, dass Devisenhändler die Umtauschkurse nur für gewisse maximale Geldbeträge garantieren und das auch nur für wenige Minuten. Dies liegt daran, dass sich alle Devisenbörsen der Welt gleichzeitig beobachten und Kursschwankungen an einer Börse sofort zu Kursänderungen bei den anderen führen. Je nach Sachlage kann man die Umtauschsummenbeschränkungen dadurch berücksichtigen, dass man Bogenkapazitäten einführt oder dass man Umtauschbeschränkungen an jeder Devisenbörse festlegt. In unserem Beispiel folgen wir der zweiten Methode. Wir nehmen an, dass Tauschvorgänge, die folgenden Grenzen an einem Börsenplatz nicht überschreiten:

London	:	£	2 000 000,
New York	:	sl\$	4 000 000,
Zürich	:	SF	10 000 000,
Tokio	:	Yen	800 000 000,
Paris	:	FF	25 000 000,

keine Kursänderungen induzieren. Damit können wir unser Problem als das folgende li-

neare Programm schreiben.
(8.11)

$$\max \sum_{a\in\delta^-(\mathrm{DM})} m(a)x(a) \quad - \sum_{a\in\delta^+(\mathrm{DM})} x(a)$$

$$\sum_{a\in\delta^+(v)} x(a) \quad - \sum_{a\in\delta^-(v)} m(a)x(a) = 0 \quad \forall\, v \in \{\$, \pounds, \mathrm{SF}, \mathrm{FF}, \mathrm{Yen}\}$$

$$\textstyle\sum_{a\in\delta^+(v)} x(a) \quad \le \begin{cases} 2.0 \text{ Mio} & \text{falls } v = \pounds \\ 4.0 \text{ Mio} & \text{falls } v = \$ \\ 10.0 \text{ Mio} & \text{falls } v = \mathrm{SF} \\ 800.0 \text{ Mio} & \text{falls } v = \mathrm{Yen} \\ 25.0 \text{ Mio} & \text{falls } v = \mathrm{FF}\} \end{cases}$$

$$x_a \quad \ge 0 \quad \forall\, a \in A.$$

Es gibt in der Literatur einige Vorschläge zur algorithmischen Behandlung der hier vorgestellten speziellen LP's. Aus Zeitgründen können wir darauf nicht eingehen. Wir haben das LP (8.11) für die zwei Probleme aus den Tabellen 8.1 und 8.2 mit dem Simplexalgorithmus gelöst. (Die Koeffizienten von (8.11) können z. B. wie folgt gelesen werden $m((\mathrm{DM}, sl\$))$ ist am 16.09.1994 gleich 0.6477 und am 16.12.1994 gleich 0.6365.) Es ergeben sich die in den Abbildungen 8.3 und 8.4 dargestellten Optimallösungen.

Man hätte also am 16.09.1994 unter Einsatz von 18,2 Mio DM einen Umtauschgewinn von DM 1004,– erzielen können. Durch die Summenbeschränkungen bei den einzelnen Währungen war kein höherer DM-Einsatz möglich.

Man beachte, dass natürlich aufgrund der im Modell nicht berücksichtigten Zeitbeschränkungen alle Umtauschaktionen innerhalb weniger Minuten (bzw. gleichzeitig) ausgeführt werden müssen, um den Gewinn zu realisieren. (Dieser wird natürlich durch die hohen Telefonkosten geschmälert.) Aber im Prinzip können diese Tauschvorgänge — auch nach Kursänderungen und damit verbundenen Neuberechnungen — immer wiederholt werden, so dass sich (theoretisch) relativ hohe Gewinne ansammeln können. Jedoch scheitert dieses Verfahren i. a. (zumindest für Nicht-Devisenhändler) am technisch nicht realisierbaren direkten Marktzugang. Nur wenige Händler haben eine Lizenz zur Durchführung dieser Transaktionen.

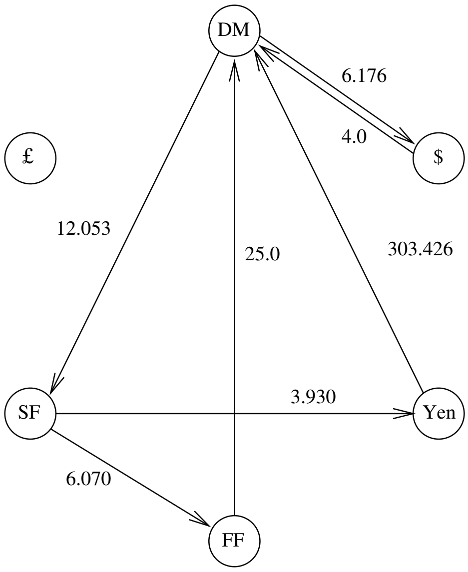


Abb.8.3

Einsatz:	18 228 248
Gewinn:	1 004

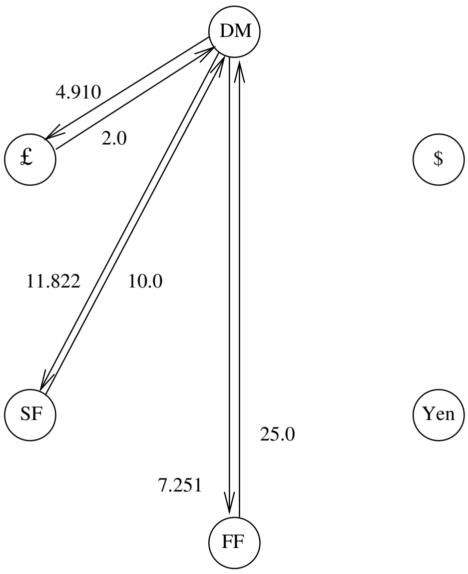


Abb.8.4

Einsatz:	23 983 535
Gewinn:	365

8.3 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 (8.1) 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.

(8.12) 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**. □

Offenbar kann man ein Umladeproblem wie in (8.13) angegeben als lineares Programm schreiben.

$$
\begin{aligned}
&\min \sum w(a)x(a) \\
(8.13)\qquad & x(\delta^-(v)) - x(\delta^+(v)) = \begin{cases} 0 & \text{falls } v \in V_u \\ b(v) & \text{falls } v \in V_n \\ -a(v) & \text{falls } v \in V_a \end{cases} \\
& 0 \le x(a) \le c(a) \quad \forall\, a \in A.
\end{aligned}
$$

(8.13) ist offensichtlich höchstens dann lösbar, wenn $\sum_{v \in V_n} b(v) = \sum_{v \in V_a} a(v)$ gilt. Das lineare Programm (8.13) kann in ein Minimalkosten-Flußproblem (8.1) 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.

(8.14) Transportprobleme. Ein Transshipment, bei dem $V_u = \emptyset$ gilt, heißt Transportproblem. Hier wird also von Erzeugern direkt zu den Kunden geliefert, ohne den Zwischenhandel einzuschalten. □

(8.15) 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 (2.9)). □

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).

Literaturverzeichnis

Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1989). *Network Flows, Handbooks in Operations Research and Management Science*, volume 1, chapter Optimization, pages 211–360. Elsevier, North-Holland, Amsterdam, G. L. Nemhauser, A. H. G. Rinnooy Kan and M. J. Todd edition.

Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1993). *Network Flows, Theory, Algorithms and Applications*. Pearson Education, Prentice Hall, New York, first edition.

Ford Jr., L. R. and Fulkerson, D. R. (1962). *Flows in Networks*. Princeton University Press, Princeton.

Goldberg, V., Tardos, E., and Tarjan, R. E. (1990). Network Flow Algorithms. In B. Korte et al., editor, *Paths, Flows, and VLSI-Layout*. Springer-Verlag, Berlin.

Hu, T. C. (1969). *Integer Programming and Network Flows*. Addison-Wesley, Reading, Massachusetts.

Jensen, P. A. and Barnes, J. W. (1980). *Network Flow Programming*. Wiley & Sons, Inc., New York.

Kennington, J. and Helgason, R. (1980). *Algorithms for Network Programming*. Wiley & Sons, Inc., New York.

Lawler, E. L. (1976). *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart & Winston, New York.

Shigeno, M., Iwata, S., and McCormick, S. T. (2000). Relaxed most negative cycle and most positive cut canceling algorithms for minimum cost flow. *Mathematics of Operations Research*, 25:76–104.

Kapitel 9

Primale Heuristiken für schwere Probleme: Eröffnungs- und Verbesserungsverfahren

Es ist leider so, dass viele der in der Praxis vorkommenden kombinatorischen Optimierungsprobleme groß und — im Sinne der Komplexitätstheorie — schwer sind. Ferner müssen viele dieser Probleme häufig in beschränkter Zeit mit nicht allzu großen Computern "gelöst" werden. Diesen Praxisanforderungen kann man mit zwei möglichen Ansätzen entsprechen. Entweder man entwickelt — falls das möglich ist — Verfahren, die für die vorliegenden speziellen Probleme exakt arbeiten und empirisch für die gegebenen Größenordnungen vertretbaren Rechenaufwand erfordern, oder man entwirft approximative Verfahren (Heuristiken), die in kurzer Zeit obere und untere Schranken für den Optimalwert liefern und somit gewisse Güteaussagen erlauben.

Für spezielle Anwendungsprobleme mit problemspezifischen Nebenbedingungen können natürlich individuelle Heuristiken entwickelt werden, die auf die besonderen Nebenbedingungen abgestellt sind. Es gibt jedoch einige Prinzipien, die (mit geeigneten Modifikationen) bei allen schwierigen kombinatorischen Optimierungsproblemen zur Entwicklung von Heuristiken eingesetzt werden können. Diese Prinzipien sollen in diesem Kapitel erläutert und (meistens) anhand des symmetrischen Travelling Salesman Problems eingehend erklärt werden.

Wir wollen solche Heuristiken, die zulässige Lösungen des vorliegenden Optimierungsproblems liefern, **primale Heuristiken** nennen. Es gibt unzählige Versuche, Heuristiken zu klassifizieren. Wir wollen diese Versuche hier nicht weiter kommentieren und werten. Im weiteren werden wir primale Heuristiken in zwei Verfahrensklassen aufteilen, also nur eine ganz grobe Gliederung vornehmen. Die beiden Klassen wollen wir mit Eröffnungsverfahren (oder Startheuristiken) und mit Verbesserungsverfahren bezeichnen.

Unter **Eröffnungsverfahren** versteht man solche Algorithmen für kombinatorische Optimierungsprobleme, die mit der leeren Menge (oder einer "trivialen Anfangsmenge") beginnend sukzessive eine zulässige Lösung aufbauen, wobei beim Aufbau **lokale Optimierungsüberlegungen** angestellt werden. Ein typisches Beispiel hierfür ist der uns bereits bekannte Greedy-Algorithmus.

Verbesserungsverfahren sind solche Algorithmen, die mit einer zulässigen Anfangslösung beginnen und wiederum unter Berücksichtigung lokaler Optimalitätsbedingungen gewisse Änderungen an der bisherigen Lösung vornehmen und so fortschreitend zu einer "lokalen Optimallösung" gelangen.

9.1 Eröffnungsheuristiken für symmetrisches TSP

Wir erinnern daran, dass das symmetrische Travelling Salesman Problem (TSP) die Aufgabe ist, in einem ungerichteten vollständigen Graphen $K_n = (V, E)$ mit Kantengewichten einen hamiltonschen Kreis (auch Tour genannt) zu finden, der möglichst geringes Gewicht hat. Das TSP ist das vermutlich am besten untersuchte kombinatorische Optimierungsproblem. Die Anzahl der Veröffentlichungen ist fast unübersehbar. Das TSP ist auch eine beliebte Spielwiese für Amateure (dazu gehören auch Wissenschaftler aus der Physik, den Ingenieurwissenschaften, der Biologie und einigen anderen Bereichen), die neue (oder bekannte, aber mit neuem Namen versehene) Heuristiken erfinden, häufig auf Analogien zu physikalischem oder biologischem Vorgehen fußend, und die nicht selten kühne Behauptungen über die Leistungsfähigkeit dieser Verfahren aufstellen. Manches davon findet den Weg in die Tagespresse.

Ein sehr gutes Buch zum TSP ist der Sammelband Lawler et al. (1985), welcher – obwohl schon einige Jahre alt – ausgezeichnete Information zum TSP enthält. Das Buch Reinelt (1994) enthält eine sorgfältige Studie für Heuristiken zur Bestimmung von unteren und oberen Schranken für den Wert einer optimalen Tour. Die zur Analyse der verschiedenen Verfahren benutzten TSP-Beispiele (fast alle aus der Praxis) sind von G. Reinelt elektronisch verfügbar gemacht worden. Diese Sammlung, genannt *TSPLIB*, ist inzwischen durch Hinzunahme sehr großer Problembeispiele erheblich erweitert worden und unter der URL

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

aufrufbar. Sehr gute Informationen zum TSP (u. a. Bilder der "Weltrekorde", d. h. der größten exakt gelösten TSPs) findet man auf der von D. Applegate, B. Bixby, V. Chvátal und B. Cook angelegten Webpage `http://www.keck.caam.rice.edu/tsp`.

Dieses Autorenteam hat derzeit den besten Code zur exakten Lösung großer TSPs. Eine weitere interessante Homepage zum TSP ist TSPBIB

`http://www.densis.fee.unicamp.br/~moscato/TSPBIB_home.html`

wo Verweise zu vielen anderen elektronischen Quellen zum TSP zu finden sind, u. a. zu Seiten, wo man die Ausführung einiger Heuristiken graphisch verfolgen kann.

Hamiltonsche Kreise

Bevor wir uns dem TSP zuwenden, wollen wir kurz der Frage nachgehen, unter welchen Bedingungen an einen Graphen auf die Existenz eines hamiltonschen Kreises geschlossen werden kann. Da das Problem $\mathcal{NP}$-vollständig ist, kann man natürlich keine vollständige Lösung des Problems erwarten. Alle hinreichenden Bedingungen fordern im Prinzip, dass ein Graph viele, gut verteilte Kanten haben muss, wenn er hamiltonsch sein soll. Eine einfache Idee ist, für jeden Knoten zu fordern, dass er einen hohen Grad hat. Die Idee funktioniert jedoch nicht direkt. Man überlegt sich leicht, dass es zu jedem $d \in \mathbb{N}$ einen Graphen G mit Minimalgrad $\delta(G) \geq d$ gibt, der keinen hamiltonschen Kreis besitzt. Ein Klassiker des Gebiets ist die folgende Beobachtung von Dirac aus dem Jahre 1952.

(9.1) Satz *Ist $G = (V, E)$ ein einfacher Graph mit $n = |V| \geq 3$ und Minimalgrad $\delta(G) \geq \frac{n}{2}$, so enthält G einen Hamiltonkreis.*

Beweis :

Sei $P = v_1, \ldots, v_k$ ein längster Weg in G. Hätte v_1 einen Nachbarn außerhalb von P, könnte der Weg verlängert werden, was wegen seiner Maximalität nicht geht. Analog liegen auch alle Nachbarn von v_k in P. Mindestens $\frac{n}{2}$ der höchsten $n-1$ Knoten $v_1, \ldots, v_{k-1}$ sind Vorgänger auf P eines Nachbarn von v_1, und mindestens $\frac{n}{2}$ der Knoten $v_1, \ldots, v_{k-1}$ sind Nachbarn von v_k. Folglich muss es einen Knoten, sagen wir $v_i, 1 \leq i < k$, geben mit $v_1 v_{i+1} \in E$ und $v_i v_k \in E$. Damit ist $C = v_1 v_{i+1} P v_k v_i P v_1$ ein Kreis in G. Gäbe es einen Knoten $v_0 \in V$, der nicht in C ist, so gäbe es (wegen $\deg(v_0) \geq \frac{n}{2}$) zwei Nachbarn von v_0, die auf C benachbart sind. Wir könnten also C verlängern und hätten damit auch einen Weg gefunden, der länger als P ist, Widerspruch. Also ist C ein hamiltonscher Kreis. $\square$

Die hinreichende Bedingung des Satzes von Dirac ist sukzessive von verschiedenen Autoren verfeinert worden. Wir geben hier zwei Resultate dieser Art an. Die Beweise verlaufen ähnlich, die Konstruktionen sind natürlich komplizierter. Das nächste Resultat stammt von Chvátal und ist aus dem Jahr 1972.

Eine Folge $d_1, d_2, \ldots, d_n$ nennen wir **Gradsequenz**, wenn es einen einfachen Graphen G mit n Knoten $1, \ldots, n$ gibt, so daß $\deg(i) = d_i, i = 1, \ldots, n$. Wir nennen eine Gradsequenz **hamiltonsch**, wenn jeder Graph mit dieser Gradsequenz hamiltonsch ist.

(9.2) Satz *Eine Gradsequenz $d_1, d_2, \ldots, d_n$ ist genau dann hamiltonsch, wenn für jedes*

$i, 1 \leq i < \frac{n}{2}$ *gilt:*

$$d_i \leq i \rightarrow d_{n-i} \geq n - i. \quad \square$$

Bondy und Chvátal beobachteten 1974, dass sich der Diracsche Beweis so modifizieren lässt, dass man folgende (stärkere) Bedingung erhält.

(9.3) Satz

Sei G ein einfacher Graph mit n Knoten, und seien $u, v \in V$ zwei nichtadjazente Knoten, so daß

$$\deg(u) + \deg(v) \geq n.$$

Dann gilt: G ist hamiltonsch genau dann, wenn $G + uv$ hamiltonsch ist. $\square$

Sei G ein beliebiger Graph mit n Knoten. Der **Abschluss** von G ist derjenige Graph, den man aus G dadurch erhält, dass man rekursiv für jedes Paar u, v nichtadjazenter Knoten mit Gradsumme $\geq n$ die Kante uv zu G hinzufügt. Man kann leicht zeigen, dass der Abschluss eindeutig bestimmt (also unabhängig von der Reihenfolge des Hinzufügens) ist. Satz (9.3) liefert dann durch Rekursion:

(9.4) Satz *Ein einfacher Graph G ist genau dann hamiltonsch, wenn sein Abschluss hamiltonsch ist.* $\square$

Ist insbesondere der Abschluss von G der vollständige Graph K_n, so kann man daraus schließen, dass G hamiltonsch ist.

Die Bedingungen der oben angegebenen Sätze implizieren, dass jeder Graph mit n Knoten, der eine solche Bedingung erfüllt, $\mathcal{O}(n^2)$ Kanten enthält. Die Theorie der Zufallsgraphen liefert schärfere Resultate. Man kann zeigen, dass ein zufälliger Graph mit $\mathcal{O}(n \log n)$ Kanten mit hoher Wahrscheinlichkeit hamiltonsch ist.

Warum werden Sätze vom Typ (9.1), (9.2) oder (9.3) in einem Kapitel über Heuristiken betrachtet? Ein Grund dafür (neben dem Wunsch des Vortragenden, ein paar interessante Resultate der Graphentheorie zu vermitteln) ist, dass die Beweise der Sätze algorithmisch sind. In der Tat sind sehr viele Beweise in der Graphentheorie algorithmisch; sie werden häufig nur nicht so formuliert, weil die Autoren entweder möglichst kurze Beweise geben wollen oder an Algorithmen nicht wirklich interessiert sind.

Schauen wir uns also den Beweis des Satzes von Dirac genauer an und interpretieren wir ihn als Heuristik.

Wir wählen einen beliebigen Knoten von $G = (V, E)$, sagen wir v; dann gehen wir zu einem Nachbarn von v, sagen wir v'; von v aus besuchen wir einen Nachbarn, den wir bisher nicht besucht haben. Wir führen diese Depth-First-Suche solange durch, bis wir zu einem Knoten kommen, dessen Nachbarn alle bereits in dem bisher konstruierten Weg sind. Dann gehen wir zurück zu v, wählen einen weiteren (noch nicht besuchten)

Nachbarn von v und verlängern den Weg so lange wie möglich. Wir erhalten auf diese Weise einen Weg $P = v_1, \dots, v_k$, so daß alle Nachbarn von v_1 und von v_k in P liegen. Dies ist eine Trivialheuristik zur Konstruktion eines maximalen (nicht notwendigerweise längsten) Weges in G.

Nun gehen wir genau wie im Beweis von Satz (9.1) vor. Aufgrund der Bedingung $\delta(G) \geq \frac{n}{2}$ können wir einen Knoten $v_i, 1 \leq i < k$, finden mit $v_1, v_k \in E$ und $v_1, v_k \in E$. Wir schließen P zu einem Kreis $C = v_1 v_{i+1} P v_k v_i P v_1$.

Gibt es einen Knoten $v_0 \in V$, der nicht in G ist, so hat v_0 (wegen $\deg(v_0) \geq \frac{n}{2}$) zwei Nachbarn in C, die auf C benachbart sind, sagen wir v_j und v_{j+1}. Wir entfernen die Kante $v_j v_{j+1}$ aus C und fügen die beiden Kanten $v_j v_0$ und $v_0 v_{j+1}$ ein, verlängern also C auf diese Weise. Gibt es einen weiteren Knoten außerhalb des neuen Kreises, so wiederholen wir diese Prozedur. Gibt es keinen Knoten mehr, der nicht in C liegt, so haben wir einen hamiltonschen Kreis gefunden.

Den Satz von Dirac kann man auch als Algorithmusanalyse interpretieren. Wir erfinden eine Heuristik (erst langen Weg konstruieren, diesen zum Kreis schließen, alle restlichen Knoten einbauen) und formulieren dann eine Bedingung (hier $\delta(G) \geq \frac{n}{2}$), so dass die Heuristik beweisbar einen hamiltonschen Kreis liefert.

Es ist recht instruktiv, graphentheoretische Sätze einmal auf die oben beschriebene Weise zu durchleuchten. Sehr häufig sind die Resultate und Beweise auf genau die hier beschriebene Weise interpretierbar.

TSP-Eröffnungsverfahren

Die im nachfolgenden beschriebenen TSP-Heuristiken folgen übrigens in verschiedener Hinsicht den gleichen Prinzipien, die oben dargelegt wurden. Bei der Auswahl der Ausbaustrategie (Verlängerung des bestehenden Weges oder Kreises) werden Auswahlregeln benutzt, die in einem spezifizierbaren Sinn "greedy" sind. Man versucht bei einer Eröffnungsheuristik, den nächsten Schritt so auszuführen, dass er bezüglich eines lokalen Kriteriums optimal ist. Natürlich muss gewährleistet sein, dass am Ende eine zulässige Lösung (ein hamiltonscher Kreis oder kurz Tour) gefunden wird.

Der Kern aller derartigen Verfahren ist die "geschickte" Wahl des lokalen Optimalitätskriteriums. Es muss algorithmisch einfach sein (aus Rechenzeitgründen), die lokalen Optima zu bestimmen. Zum anderen soll gewährleistet sein, dass das lokale Kriterium zu einer "guten" Lösung im Sinne der globalen Zielfunktion führt. Beim Greedy-Algorithmus bestand die Strategie darin, lokal das bezüglich des Gewichtes bestmögliche Element zu wählen , und wie Satz (5.18) zeigt, fährt man damit bei Maximierungsproblemen nicht allzu schlecht. Bei Minimierungsaufgaben ließ sich jedoch keine Gütegarantie finden.

(9.5) Eröffnungsverfahren für das symmetrische TSP.

Gegeben sei ein vollständiger Graph $K_n = (V, E), n \geq 3,$ *mit "Kantenlängen"* c_{ij} *für* *alle* $ij \in E$.

(a) **Nächster Nachbar (NN)**

1. *Wähle irgendeinen Knoten* $i \in V$, *markiere* i, *und setze* $T := \emptyset$, $p := i$.
2. *Sind alle Knoten markiert, setze* $T := T \cup \{ip\}$ *und STOP (*T *ist eine Tour).*
3. *Bestimme einen unmarkierten Knoten* j, *so dass*

$$c_{pj} = \min\{c_{pk} \mid k \text{ unmarkiert}\}.$$

4. *Setze* $T := T \cup \{pj\}$, *markiere* j, *setze* $p := j$ *und gehe zu 2.*

(b) **NEAREST INSERT (NI)**

1. *Wähle irgendeinen Kreis* C *der Länge drei.*
2. *Ist* $V \setminus V(C) = \emptyset$, *STOP,* C *ist eine Tour.*
3. *Bestimme einen Knoten* $p \in V \setminus V(C)$, *so dass ein Knoten* $q \in V(C)$ *existiert mit*

$$c_{pq} = \min\{\min\{c_{ij} \mid j \in V(C)\} \mid i \in V \setminus V(C)\}.$$

4. *Bestimme eine Kante* $uv \in C$ *mit*

$$c_{up} + c_{pv} - c_{uv} = \min\{c_{ip} + c_{pj} - c_{ij} \mid ij \in C\}.$$

5. *Setze* $C := (C \setminus \{uv\}) \cup \{up, pv\}$ *und gehe zu 2.*

(c) **FARTHEST INSERT (FI)**

Wie NI, es wird lediglich Schritt 3 ersetzt durch

3. *Bestimme einen Knoten* $p \in V \setminus V(C)$, *so dass ein Knoten* $q \in V(C)$ *existiert mit*

$$c_{pq} = \max\{\min\{c_{ij} \mid j \in V(C)\} \mid i \in V \setminus V(C)\}.$$

(d) **CHEAPEST INSERT (CI)**

Wie NI, es werden lediglich die Schritte 3 und 4 ersetzt durch

3. *Bestimme einen Knoten* $p \in V \setminus V(C)$ *und eine Kante* $uv \in C$ *mit*

$$c_{up} + c_{pv} - c_{uv} = \min\{c_{ik} + c_{kj} - c_{ij} \mid ij \in C, k \in V \backslash C\}.$$

(e) **SPANNING-TREE-Heuristik (ST)**

1. *Bestimme einen minimalen aufspannenden Baum* B *von* K_n.
2. *Verdopple alle Kanten aus* B, *um einen Graphen* (V, B_2) *zu erhalten.*
3. *(Da jeder Knoten in* (V, B_2) *einen geraden Grad hat und* (V, B_2) *zusammenhängend ist, enthält* (V, B_2) *eine Eulertour.) Bestimme eine Eulertour* C, *gib ihr eine Orientierung, wähle einen Knoten* $i \in V$, *markiere* i, *und setze* $p := i$, $T := \emptyset$.
4. *Sind alle Knoten markiert, setze* $T := T \cup \{ip\}$ *und STOP (*T *ist eine Tour).*

5. *Laufe von p entlang der Orientierung von C, bis ein unmarkierter Knoten, sagen wir q, erreicht ist. Setze $T := T \cup \{p, q\}$, markiere q, setze $p := q$ und gehe zu 4.*

(f) **CHRISTOFIDES Heuristik (CH)**

Wie (ST), lediglich Schritt 2 wird ersetzt durch

2. *Sei W die Menge der Knoten in (V, B) mit ungeradem Grad. (Man beachte: $|W|$ ist gerade!)*

a) *Bestimme im von W induzierten Untergraphen von K_n ein perfektes Matching M minimalen Gewichts.*

b) *Setze $B_2 := B \cup M$ (Achtung: hier können Kanten doppelt vorkommen!).* □

Wir wollen nun an einigen Beispielen die Wirkungsweisen der 7 oben angegebenen Heuristiken vorführen.

(9.6) Beispiel. Wir betrachten das durch Tabelle 9.1 definierte 6-Städte Problem (das **Rheinland-Problem**). Die Daten sind Straßenkilometer und entstammen einem Straßenatlas.

	A	B	D	F	K	W
Aachen	–	91	80	259	70	121
Bonn	91	–	77	175	27	84
Düsseldorf	80	77	–	232	47	29
Frankfurt	259	175	232	–	189	236
Köln	70	27	47	189	–	55
Wuppertal	121	84	29	236	55	–

Tabelle 9.1

Die geographische Lage der sechs Orte ist in Abb. 9.1 ungefähr maßstabgetreu angedeutet.

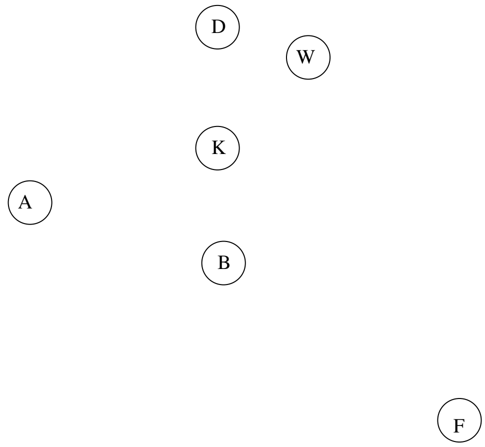


Abb. 9.1

(a) Wir wenden NN an

(a_1) Startpunkt A: Es ergibt sich die Rundreise

A — K — B — D — W — F — A

mit der Länge 698 km.

(a_2) Startpunkt F: Wir erhalten die Tour

F — B — K — D — W — A — F

der Länge 658 km.

(b) Wir wenden NI mit dem Startkreis K — D — B — K an. Dabei werden sukzessive die folgenden Kreise erzeugt

K — D — B — K, K — D — W — B — K,
K — A — D — W — B — K, K — A — D — W — F — B — K.

Die so entstandene Tour hat die Länge 617 km.

(c) Wir wenden FI mit dem Startkreis K — A — F — K an. Wir erhalten sukzessiv die folgenden Kreise:

K — A — F — K, K — A — F — W — K,
K — A — F — W — D — K, K — A — B — F — W — D — K.

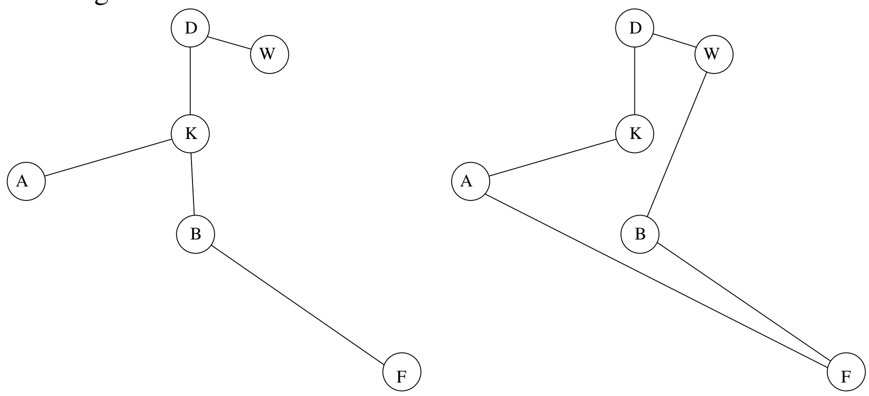
Die durch FI gefundene Tour hat die Länge 648 km.

(d) Mit CI und dem Startkreis A — B — F — A erhalten wir

A — B — F — A, A — K — B — F — A,
A — D — K — B — F — A, A — D — W — K — B — F — A.

Die so gefundene Tour hat die Länge 625 km.

(e) Der minimale aufspannende Baum B des Rheinland-Problems ist in Abbildung 9.2 (a) gezeigt. Er hat die Länge 348 km. Wir verdoppeln die Kanten von B, wählen die folgende orientierte Eulertour AK, KD, DW, WD, DK, KB, BF, FB, BK, KA und starten mit A. Daraus ergibt sich die in Abbildung 9.2 (b) gezeigte Rundreise der Länge 664 km.



(a) (b)

Abb. 9.2

(f) Im minimalen aufspannenden Baum B (siehe Abbildung 9.2 (a)) haben die Knoten A, K, W und F ungeraden Grad. Das minimale perfekte Matching des durch $\{$A, K, W, F$\}$ induzierten Untergraphen besteht aus den Kanten $M = \{$AK, WF$\}$. Sei $B_2 = B \cup M$. Wir wählen die orientierte Eulertour AK, KD, DW, WF, FB, BK, KA von (V, B_2), starten in A und erhalten die Tour A — K — D — W — F — B — A der Länge 648 km. Starten wir dagegen in B, so ergibt sich die Tour B — K — A — D — W — F — B der Länge 617 km.

Die durch unsere Heuristiken gefundene kürzeste Rundreise hat also die Länge 617 km und ist in Abbildung 9.3 gezeigt

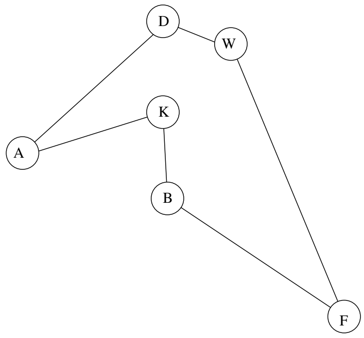


Abb. 9.3

□

Beispiel (9.2) zeigt deutlich, dass die Heuristiken durchaus unterschiedliche Ergebnisse liefern können. Selbst einunddieselbe Heuristik kann bei verschiedener Wahl der noch offenen Parameter (z. B. Startpunkt) zu stark voneinander abweichenden Lösungen führen.

Wir wollen nun untersuchen, ob man theoretisch etwas über die Güte der durch diese Heuristiken gefundenen Rundreisen sagen kann. Zunächst ein negatives Resultat bezüglich der Möglichkeit, eine approximative Lösung für das symmetrische TSP zu finden.

(9.7) Satz. *Für ein beliebiges symmetrisches TSP bezeichnen wir mit T_{opt} eine optimale Tour. Gibt es ein $\varepsilon > 0$ und einen polynomialen Algorithmus H, der für jedes symmetrische TSP eine Tour T_H liefert mit*

$$c(T_{\text{opt}}) \leq c(T_H) \leq (1+\varepsilon)c(T_{\text{opt}}),$$

dann ist $P = \mathcal{NP}$, d. h. das ε-Approximationsproblem des symmetrischen TSP ist $\mathcal{NP}$-vollständig.

Beweis : Wir wissen, dass das folgende Problem $\mathcal{NP}$-vollständig ist. (HAM) Gibt es einen hamiltonschen Kreis in einem Graphen G? Es ist klar, dass das ε-Approximationsproblem für das TSP in $\mathcal{NP}$ ist. Um zu zeigen, dass dieses Problem $\mathcal{NP}$-vollständig ist, beweisen wir folgendes: Wenn es einen Algorithmus H mit den obigen Eigenschaften gibt, dann gibt es auch einen polynomialen Algorithmus für (HAM).

Angenommen es existieren ein $\varepsilon > 0$ und ein polynomialer Algorithmus H mit obigen Eigenschaften. Sei $G = (V, E)$ ein beliebiger Graph der Ordnung n, und sei

$$M := \varepsilon\, n + 2\,.$$

Wir definieren ein TSP auf n-Städten durch die folgenden Entfernungen:

$$\begin{aligned} c_{ij} &= 1 && \text{falls} \quad ij \in E, \\ c_{ij} &= M && \text{sonst.} \end{aligned}$$

Aufgrund dieser Definition gilt:

$$G \quad \text{hamiltonsch} \iff c(T_{\text{opt}}) = n\,.$$

Ist T eine Tour, die kein hamiltonscher Kreis in G ist, so gilt

$$c(T) \geq n - 1 + M = n - 1 + \varepsilon\, n + 2 > (1+\varepsilon)n\,.$$

Sei nun T_H die von der Heuristik gefundene Lösung, und sei G hamiltonsch, so gilt $c(T_{\text{opt}}) = n \leq c(T_H) \leq (1+\varepsilon)c(T_{\text{opt}}) = (1+\varepsilon)n$. Da aber für jede Tour, die kein Hamiltonkreis in G ist, $c(T) > (1+\varepsilon)n$ gilt, muss T_H ein hamiltonscher Kreis in G sein, d. h. wir können einen hamiltonschen Kreis in G in polynomialer Zeit finden. $\square$

Die Situation des Satzes (9.8) trifft leider auf relativ viele kombinatorische Optimierungsprobleme zu. Das heißt, man kann in polynomialer Zeit nicht einmal eine feste Fehlerschranke garantieren. Ein Ausweg bleibt allerdings. Falls praktische Probleme vorliegen, sollte man versuchen, spezielle Strukturen zu finden und diese auszunutzen. Für die Theorie heißt das, man muss Spezialfälle des Problems bestimmen, für die Gütegarantien bewiesen werden können (und die möglichst die praxisrelevanten Beispiele umfassen).

Beim TSP ist der folgende Spezialfall von besonderer Praxisbedeutung. Wir nennen ein symmetrisches TSP **metrisch**, wenn für die Entfernungen die **Dreiecksungleichung** gilt, wenn also für alle Tripel i, j, k von Knoten gilt:

$$c_{ik} \leq c_{ij} + c_{jk}\,.$$

Bei Maschinensteuerungsproblemen (z.B. Bohren von Löchern in Leiterplatten) und bei geographischen Problemen ist die Dreiecksungleichung erfüllt. Das gilt jedoch häufig nicht für Entfernungstabellen in Straßenatlanten. Für euklidische TSP gilt (9.7) nicht.

(9.8) Satz. *Für metrische TSP und die Heuristiken aus (9.5) gelten die in Tabelle 9.2 angegebenen Gütegarantien. Die Laufzeitschranken gelten für beliebige Travelling Salesman Probleme.*

(Tabelle 9.2) ist wie folgt zu lesen. In der Spalte Laufzeit steht die Ordnung der Laufzeit, also ist z. B. NI ein $O(n^2)$-Algorithmus. In der Spalte ε besagt die dort angegebene Zahl, dass die zugehörige Heuristik immer eine Lösung liefert, deren Wert nicht größer als $(1+\varepsilon)c(T_{\text{opt}})$ ist. Also weichen z. B. die Werte der von der Christofides-Heuristik gefundenen Lösungen höchstens um 50 % vom Optimalwert ab.

Name	ε	Laufzeit
Nächster Nachbar	$\frac{1}{2}(\lceil\log n\rceil - 1)$	n^2
Nearest Insert	1	n^2
Farthest Insert	$\geq \frac{1}{2}$	n^2
Cheapest Insert	1	$n^2 \log n$
Spanning-Tree	1	n^2
Christofides	$\frac{1}{2}$	n^3

Tabelle 9.2

Beweis : Die Aussagen über die Laufzeiten folgen direkt aus den Darstellungen der Algorithmen in (9.5). Wir beweisen die Güteschranken von ST und CHRISTOFIDES.

Entfernen wir aus einer Tour eine Kante, so erhalten wir einen aufspannenden Baum. Da bei einem euklidischen TSP alle Kantengewichte nichtnegativ sind, folgt daraus, dass der Wert eines minimalen aufspannenden Baums B nicht größer ist als der der optimalen Tour T_{opt}.

Daher gilt für die Heuristik ST: $c(B_2) = \sum_{ij \in B} 2c_{ij} \leq 2c(T_{\text{opt}})$.

Die aus B_2 konstruierte Tour T entsteht dadurch, dass Wege in B_2 durch Kanten zwischen den Endknoten der Wege ersetzt werden. Da die Dreiecksungleichung gilt, ist der Wert dieser Kanten nicht größer als die Summe der Werte der Kanten der Wege. Daraus folgt $c(T) \leq c(B_2) \leq 2c(T_{\text{opt}})$, was zu zeigen war.

Bei der Christofides Heuristik müssen wir den Wert des minimalen perfekten Matchings M abschätzen. Seien $i_1, \ldots, i_{2m}$ die ungeraden Knoten des aufspannenden Baumes B und zwar so numeriert, wie sie in T_{opt} vorkommen, d. h. $T_{\text{opt}} = (i_1, \alpha_1, i_2, \alpha_2, \ldots, \alpha_{2m-1}, i_{2m}, \alpha_{2m})$, wobei die α_i (möglicherweise leere) Folgen von Knoten sind. Wir betrachten nun die beiden Matchings $M_1 = \{i_1 i_2, i_3 i_4, \ldots, i_{2m-1} i_{2m}\}$, $M_2 = \{i_2 i_3, i_4 i_5, \ldots, i_{2m-2} i_{2m-1}, i_{2m} i_1\}$. Aufgrund der Dreiecksungleichung gilt $c(T_{\text{opt}}) \geq c(M_1) + c(M_2)$. Da M optimal ist, gilt somit $c(M) \leq \frac{1}{2}\, c(T_{\text{opt}})$. Wie vorher folgt daraus für die Christofides-Tour T : $c(T) \leq c(B_2) = c(B) + c(M) \leq c(T_{\text{opt}}) + \frac{1}{2}\, c(T_{\text{opt}}) = \frac{3}{2}\, c(T_{\text{opt}})$. □

Die Gütegarantie der Christofides-Heuristik ist die beste derzeit bekannte Gütegarantie für das symmetrische TSP.

Kann man aus der Gütegarantie ablesen, wie gut sich eine Heuristik in der Praxis verhält? Leider kann man diese Frage nicht uneingeschränkt bejahen. Es scheint so, dass zur Beurteilung immer noch praktische Tests an vielen und "repräsentativen" (was auch immer das ist) Beispielen ausgeführt werden müssen. So wird z. B. in der Literatur relativ übereinstimmend berichtet, dass — bezüglich der hier vorgestellten Heuristiken — bis zu etwa 200 Knoten FI sehr häufig besser als die übrigen Verfahren ist. Für größere Probleme ist meistens die Christofides-Heuristik am besten, während FI die zweitbeste Heuristik ist. Bedenkt man allerdings, dass die Christofides-Heuristik als Unterprogramm ein (nicht trivial zu implementierendes) Verfahren zur Lösung von Matching Problemen benötigt, dann zeigt FI natürlich deutliche praktische Vorteile. Da der Christofides-Algorithmus ja ohnehin nur eine Heuristik ist, kann man natürlich den Matching-Algorithmus durch eine Matching-Heuristik ersetzen. Dadurch verliert man zwar die globale Gütegarantie, erhält aber doch eine empirisch ordentliche Heuristik, die relativ leicht zu codieren ist (als Matching-Heuristik kann man z. B. den Greedy-Algorithmus wählen). Gleichfalls sollte bemerkt werden, dass die Spanning-Tree-Heuristik trotz der nicht schlechten Gütegarantie (empirisch) in der Praxis im Vergleich mit anderen Verfahren nicht so gut abschneidet. Das gleiche gilt für CI und NI. Eine sorgfältige empirische Analyse der hier genannten und anderer Heuristiken findet man in Reinelt (1994).

Damit wollen wir es bezüglich Eröffnungsverfahren bewenden lassen und bemerken, dass sich die hier dargestellten Ideen mit etwas Einfühlungsvermögen auf alle übrigen kombinatorischen Optimierungsprobleme übertragen lassen. Bei der Wahl des "lokalen Optimierungskriteriums" ist es wichtig, nicht zu kurzsichtig zu wählen (NN ist z. B. eine relativ schlechte Heuristik). Man sollte hier globale Aspekte berücksichtigen. Dies ist z. B. bei FI gut gelöst, wie folgende (heuristische) Überlegung zeigt. Wählt man wie bei CI den lokalen bestmöglichen Einbau eines Knotens in den gegenwärtigen Kreis, so kann man in Fallen laufen, d. h. am Ende müssen einige ungünstig gelegene Knoten eingebaut werden, die in den bisherigen Kreis nicht “passen”. Bei FI versucht man, diese globalen Routenführungsfehler dadurch zu vermeiden, dass man immer denjenigen Knoten kostengünstigst einbaut, der gegenwärtig am weitesten entfernt liegt. Dadurch kann man am Ende nicht all zu viele Fehler machen.

9.2 Verbesserungsverfahren

Hat man durch ein Eröffnungsverfahren oder auf irgendeine andere Weise eine zulässige Lösung gefunden, so sollte man versuchen, die gegenwärtige Lösung durch Modifikationen zu verbessern. Die wichtigste Verfahrensklasse in der Kategorie der Verbesserungsheuristiken sind die **Austauschverfahren**. Die Idee hinter Austauschverfahren ist die folgende.

> Entferne einige Elemente aus der gegenwärtigen Lösung T, um eine Menge S zu erhalten. (Ist die Lösungsmenge ein Unabhängigkeitssystem, so ist S natürlich zulässig, i. a. (wie z. B. beim TSP) muss S keine zulässige Lösung sein.) Nun versuchen wir alle möglichen zulässigen Lösungen, die S enthalten, zu erzeugen. Falls dies einen zu großen Rechenaufwand erfordert, generiert man entweder nach einem festen Schema eine Teilmenge aller zulässigen Lösungen, die S enthalten, oder man benutzt wieder ein lokales Optimalitätskriterium, um eine derartige Lösung zu bestimmen. Ist unter den erzeugten zulässigen Lösungen eine, die besser als T ist, nennen wir diese T und wiederholen die Prozedur. Gibt es keine bessere Lösung als T, entfernen wir andere Elemente aus T und verfahren analog. Wir beenden das Verfahren, wenn alle (nach einer vorgegebenen Regel) möglichen Austauschschritte keine bessere Lösung produziert haben.

Für das symmetrische TSP wollen wir drei Varianten dieses Austauschverfahrens genauer darstellen.

(9.9) Austauschverfahren für das symmetrische TSP

(a) **Zweier-Austausch** *(2-***OPT***).*

1. *Wähle eine beliebige Anfangstour* $T = (i_1, i_2, \dots, i_n)$.
2. *Setze* $Z := \{\{i_p i_{p+1}, i_q i_{q+1}\} \mid p+1 \neq q, p \neq q, q+1 \neq p,\ 1 \leq p,\ q \leq n\}$.
3. *Für alle Kantenpaare* $\{i_p i_{p+1}, i_q i_{q+1}\} \in Z$ *führe aus:*

   Ist $c_{i_p i_{p+1}} + c_{i_q i_{q+1}} > c_{i_p i_q} + c_{i_{p+1} i_{q+1}}$, *setze*

   $T := (T \setminus \{i_p i_{p+1}, i_q i_{q+1}\}) \cup \{i_p i_q, i_{p+1}, i_{q+1}\}$ *und gehe zu 2.*
4. *Gib* T *aus.*

(b) r**-Austausch** *(*r**-OPT***).*

1. *Wähle eine beliebige Anfangstour T.*
2. *Sei Z die Menge aller r-elementigen Teilmengen von T.*
3. *Für alle $R \in Z$ führe aus:*

   Setze $S := T \setminus R$ und konstruiere alle Touren, die S enthalten. Gibt es unter diesen eine, die besser als T ist, nenne sie T und gehe zu 2.
4. *Gib T aus.*

(c) **Austausch zweier Knoten** *(2-***NODE-OPT***).*

1. *Wähle eine beliebige Anfangstour T.*
2. *Sei $Z = \{(v, y) \mid v$ und y nicht benachbart auf $T\}$.*
3. *Für alle Knotenpaare $(v, y) \in Z$ führe aus: O. B. d. A. habe T die folgende Gestalt:*

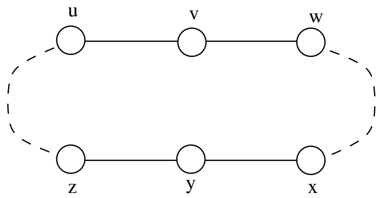


Abb. 9.4

Konstruiere aus T die folgenden 5 Touren

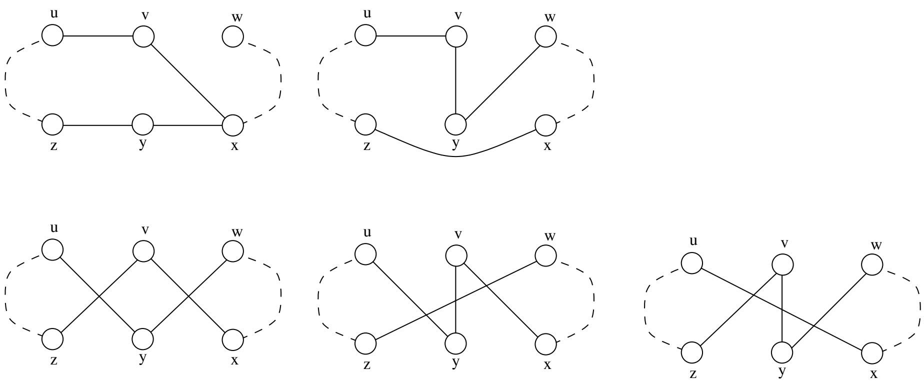


Abb. 9.5

Ist eine der Touren kürzer als T, nenne sie T und gehe zu 2.

4. Gib T aus.

□

(9.10) Bemerkung.

(a) Für einen Durchlauf der Schleife 3 benötigen 2-OPT und 2-NODE-OPT $O(n^2)$ Schritte, während r-OPT $O(n^r)$ Schritte benötigt. Es ist nicht bekannt, ob die drei Heuristiken aus (10.5) eine polynomiale Laufzeit haben.

(b) Man kann zeigen, dass für metrische TSP gilt:

$$c(T_{\text{opt}}) \le c(T_{r-\text{opt}}) \le (2 - \tfrac{2}{n})c(T_{\text{opt}}).$$

□

Für $r \geq 4$ ist das r-Austauschverfahren aus Laufzeitgründen praktisch nicht durchführbar, aber auch bei $r = 3$ treten bei größeren TSP-Instanzen erhebliche Rechenzeitprobleme auf. Dagegen laufen 2-OPT und 2-NODE-OPT empirisch mit vernünftigen Rechenzeiten.

Es ist üblich, eine Tour, die durch einen r-Austausch nicht mehr verbessert werden kann, r-**optimal** zu nennen. r-optimale Touren sind sogenannte "lokale Optima", aus denen man mit Hilfe des r-Austausches nicht "herauskommt". Man beachte, dass 2-NODE-OPT als eine Mischung des 2-OPT-, 3-OPT- und 4-OPT-Verfahrens angesehen werden kann, wobei alle Zweiertausche aber nur wenige der Dreier- und Vierertausche ausgeführt werden. Die Zahl der Dreier- und Vierertausche wird so stark reduziert, dass die Laufzeit für Schritt 3 $O(n^2)$ bleibt, aber immerhin einige "vielversprechende" Tausche vorgenommen werden. Daraus folgt, dass jede 2-NODE-optimale Tour auch 2-optimal ist.

In der Praxis sind die obigen Austauschverfahren sehr erfolgreich und können in der Regel die durch die Eröffnungsverfahren (10.1) gefundenen Lösungen verbessern. Erstaunlich dabei ist die (von vielen "Heuristikern" gemachte) Beobachtung, dass die Austauschverfahren häufig bessere Lösungen liefern, wenn man sie mit zufälligen oder relativ schlechten Startlösungen beginnt, als wenn man mit guten heuristischen Lösungen (etwa durch FI gefundene) beginnt.

Startet man z. B. 2-OPT mit der durch NN gefundenen Lösung des Rheinlandproblems (10.2) (a_1), so wird der in Abbildung 10.6 gezeigte 2-Austausch vollzogen.

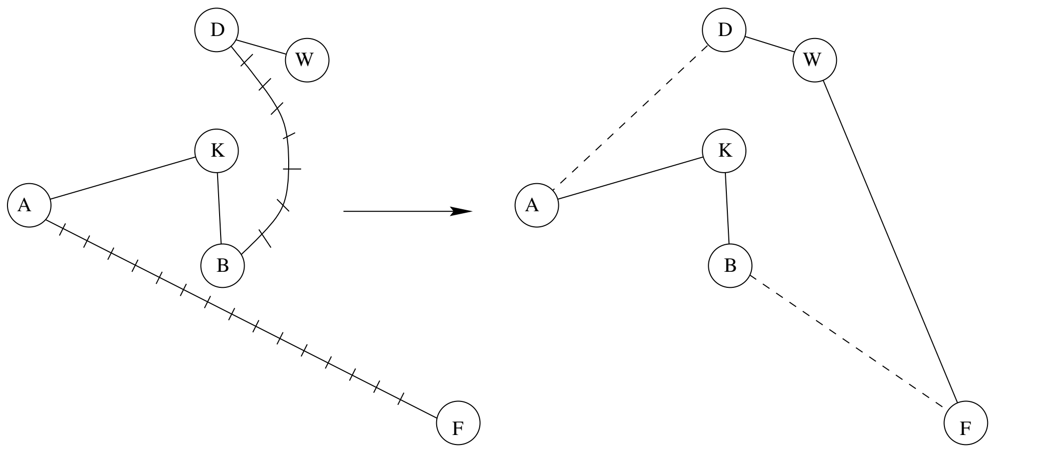


Abb. 9.6

Diese Tour hat die Länge 617 km. Es wurde also eine Verbesserung um 81 km erzielt.

Die Austausch-Verfahren für das TSP und andere Kombinatorische Optimierungsprobleme sind stark von S. Lin und B. W. Kernigham propagiert worden. In ihrem Aufsatz

> "An effective heuristic algorithm for the traveling-salesman problem", Operations Research 21 (1973) 498–516,

zeigen sie z. B. dass eine dynamische Mischung von 2-OPT und 3-OPT mit $O(n^2)$ Laufzeit empirisch sehr gute Lösungen in akzeptabler Laufzeit liefert. Es ist vermutlich nicht falsch, wenn man die Lin-Kernigham-Heuristik (bzw. Varianten davon) als die beste derzeit bekannte TSP-Heuristik bezeichnet.

Es ist jedoch keineswegs trivial, sie so zu implementieren, dass sie effektiv un in akzeptabler Laufzeit arbeitet. Eine sorgfältige Darstellung dieser Heuristik, der benutzten Datenstrukturen und vieler Implementationstricks findet man in

D. Applegate, R. Bixby, V. Chvàtal & W. Cook "Finding Tours in the TSP".

Dieser Aufsatz soll Kapitel 2 eines geplanten TSP-Buches der vier Autoren werden und ist über die Website `http://www.math.princeton.edu/tsp/papers/` elektronisch erhältlich.

Eine gute Übersicht über Heuristiken für das Travelling Salesman Problem gibt der Aufsatz Johnson and Papadimitriou (1985).

Im gleichen Buch wird im Aufsatz

> P. C. Gilmore, E. L. Lawler & D. B. Shmoys, "Well-solved special cases", pp. 87–143

gezeigt, dass gewisse Heuristiken für Spezialfälle des TSP immer Optimallösungen liefern.

Ein Verbesserungsverfahren, das in den letztern Jahren einige Aufmerksamkeit erregt hat, ist die Methode des "Simulated Annealing". Dies ist ein Verfahren, das erwachsen ist aus Simulationsmethoden der statistischen Mechanik. In dieser physikalischen Theorie untersucht man u. a. Phasenübergänge wie Kristallisation von Flüssigkeiten oder das Entstehen von Magneten. Derartige Phänomene treten bei gewissen kritischen Temperaturen auf (z. B. friert Wasser bei $0°$ C), und man möchte wissen, wie sich die Materie in diesem kritischen Temperaturbereich organisiert, um z. B. Kristalle zu bilden. Da man für verschiedene technische Prozesse Kristalle, ohne Strukturdefekte benötigt, möchte man wissen, wie man reale Flüssigkeiten kühlt und wärmt, so daß möglichst reine Kristalle entstehen.

Bevor man heutzutage reale Experimente mit derartigen physikalischen Systemen ausführt, werden meistens — besonders dann, wenn die Experimente teuer sind — Computersimulationen dieser Experimente durchgeführt. Hierzu wird ein abstraktes Modell des physikalischen Systems entworfen und implementiert und (zufälligen) Änderungen unterworfen, die z. B. in der Realität Kühlvorgängen entsprechen. Bei diesen Computerexperimenten lernt man, die realen Vorgänge besser zu verstehen, man kann möglicherweise bereits eine Theorie aufstellen, und man braucht sicherlich durch die umfangreichen Voruntersuchungen sehr viel weniger reale Experimente um z. B. die Herstellung von reinen Kristallen in den Griff zu bekommen.

Es ist — unter einigen Zusatzannahmen — möglich, viele kombinatorische Optimierungsprobleme als Realisierung physikalischer Systeme und die Optimierungsvorschrift als Energieminimierung anzusehen. Daher sind einige Physiker, insbesondere angeregt durch den Aufsatz

Kirkpatrick et al. (1983).

auf die Idee gekommen, kombinatorische Optimierungsprobleme mit den relativ ausgereiften Simulationstechniken der statischen Mechanik zu behandeln. Dies ist der Grund dafür, dass man bei den Darstellungen dieser Methoden viel physikalisches Vokabular findet, hinter dem man i. a. mehr vermutet, als einige recht simple heuristische Ideen.

Sorgfältige Implementierungen zeigen, dass Simulated Annealing bei einigen interessanten Problemen relativ konsistent sehr gute heuristische Lösungen erzeugt. Übereinstimmende Erfahrung ist jedoch, dass Simulated Annealing viel Parameter- und Feintuning erfordert und bis zur Generierung guter Lösungen sehr hohen Zeitaufwand erfordert. Allein aus dem letzten Grunde ist Simulated Annealing für viele (speziell große) praktische Probleme nicht sonderlich geeignet ist.

Nach der langen Vorrede nun eine kurze Beschreibung des Verfahrens — ohne physikalisches Brimborium.

(10.7) Simulated-Annealing-Verfahren (Überblick). *Wir nehmen an, dass ein kombinatorisches Minimierungsproblem $(E, \mathcal{I}, c)$ vorliegt.*

(1) Wähle mehrere Startheuristiken zur approximativen Lösung von $(E, \mathcal{I}, c)$. Seien $I_1, \ldots, I_k$ die hierbei gefundenen Lösungen. Sei I_0 die beste dieser Lösungen.

(2) Wähle ein (oder mehrere) Verbesserungsverfahren.

(3) Initialisiere zwei Listen L und L' von jeweils $k \geq 1$ Lösungen von $(E, \mathcal{I}, c)$. Setze $I_1, \ldots, I_k$ auf Liste L.

(4) Für alle Lösungen I auf Liste L führe aus:

(a) Wende eines oder alle Verbesserungsverfahren auf I an (oder wähle eines der Verfahren zufällig oder wähle eine aus I zufällig erzeugte neue Lösung).

(b) Wird eine Lösung I' gefunden, deren Wert besser als der von I ist, setze I' auf L'. Gilt $c(I') < c(I_0)$, setze $I_0 := I'$.

(c) Wird keine Lösung produziert, die besser als I ist, setze die beste produzierte Lösung I' mit Wahrscheinlichkeit p auf L'. (Wichtig: p sollte während des Verfahrens variieren, und die Wahrscheinlichkeit des Akzeptierens schlechter Lösungen sollte mit zunehmender Ausführungsdauer abnehmen!)

(5) Ist L' leer, STOP und gib I_0 aus, andernfalls setze $L := L'$, $L' := \emptyset$ und gehe zu (4).

Das oben beschriebene Verfahren hat sehr viele offene Parameter, von deren Wahl der Erfolg stark abhängt. Leider ist es nicht einfach, herauszufinden, wann welche Strategie bei welchen Problemen zu einer guten Heuristik führt.

Die Idee hinter dem Verfahren ist einfach erklärt. Man bearbeitet parallel mehrere Lösungen, um auf verschiedenen Wegen Fortschritte zu erzielen und so u. U. in mehrere "lokale Minima" hineinzulaufen, von denen vielleicht eines das globale Minimum ist. (Manche Verfahren des Simulated Annealing arbeiten jedoch nur mit $k = 1$, also einer gegenwärtigen Lösung.) Der meiner Meinung nach wichtigste Aspekt besteht darin, dass man gewillt ist, gelegentlich auch Verschlechterungen hinzunehmen. Hinter diesem Schritt steckt die folgende Erfahrung. Verbesserungsverfahren laufen relativ schnell in Fallen (bzw. lokale Minima), aus denen die Verbesserungsvorschriften nicht herausführen. Geht man jedoch auf eine schlechtere Lösung zurück, so kann man durch einen erneuten und weitere Austausche u. U. zu einer wesentlich besseren Lösung gelangen. Hierzu liefert Abbildung 9.7 eine suggestive Anschauung.

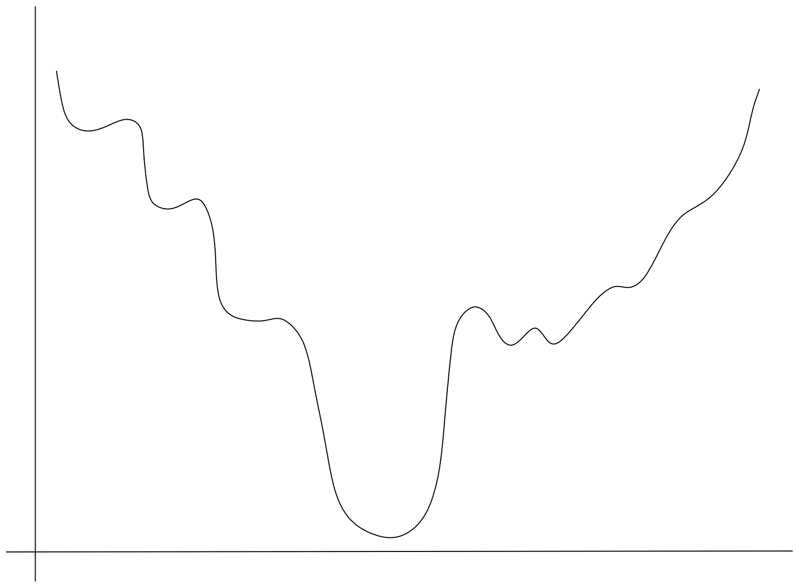


Abb. 9.7

Trägt man auf der x-Achse die Lösungen auf und auf der y-Achse deren Wert und betrachtet man die Methode, in der linken oberen Ecke einen Ball laufen zu lassen, als Verbesserungsheuristik, so ist klar, dass der Ball in Abbildung 9.7 häufig in den Nebentälern (lokale Minima) hängen bleibt. Stößt man den Ball jedoch zwischendurch gelegentlich an, so kann er — abhängig von der Stoßrichtung und -stärke — aus dem Tal herauskatapultiert werden und in ein tieferes Tal rollen. Dies "zeigt", dass gelegentliches Verschlechtern langfristig zu besseren Lösungen führen kann.

All diese Überlegungen sind jedoch nur überzeugende Prinzipien, bei denen nicht direkt klar ist, ob sie sich auch in effiziente heuristische Verfahren übersetzen lassen. Methoden des Typs (9.7) sind sehr nützlich in Bezug auf kombinatorische Optimierungsprobleme mit wenig untersuchten Strukturen und Nebenbedingungen. Nach meinen Erfahrungen sind Verfahren dieser Art (insbesondere wegen der hohen Laufzeiten) in der Regel den recht sophistifizierten Heuristiken für intensiv studierte Probleme wie das TSP unterlegen sind. Inzwischen gibt es eine Vielzahl von Aufsätzen zu Simulated Annealing und Varianten davon. Man schaue z.B. nur das Journal of Heuristics durch.

Zwei gute "Computational Studies", die über Erfahrungen mit Simulated Annealing berichten, sind Johnson et al. (1989) und Johnson et al. (1991).

In diesen Papern werden einige konkrete Implementierungen des allgemeinen Verfahrens (9.7) angegeben, bezüglich verschiedener kombinatorischer Optimierungsprobleme (wie das weiter oben vorgestellte Leiterplatten-Plazierungsproblem, das Subset-Sum-Problem, das Knotenfärbungsproblem für Graphen und das TSP) getestet und mit bekannten Heuristiken für diese Probleme verglichen. Soweit mir bekannt ist, sind Die Erfahrungen sind "durchwachsen".

Ein Buch, das Simulated Annealing im Detail beschreibt und insbesondere den stochastischen Hintergrund erklärt, ist van Laarhoven and Aarts (1987).

Weitere Verfahren, die häufig "Meta-Heuristiken" genannt werden, wurden in der Vorlesung skizziert:

- Tabu-Search
- Genetische Algorithmen
- Neuronale Netze
- Ameisensysteme
- Space-Filling Curves

Eine Ausarbeitung ist nicht erfolgt.

Literaturhinweise hierzu findet man u.a. in der bereits zitierten TSPBIB Homepage

`http://www.densis.fee.unicamp.br/~moscato/TSPBIB_home.html`

Ein gutes Buch mit Übersichtsartikeln zu den verschiedenen heuristischen Techniken ist Aarts and Lenstra (1997).

9.3 Färbungsprobleme

Die in den beiden vorangegangenen Abschnitten skizzierten Eröffnungs- und Verbesserungsheuristiken lassen sich auch sehr schön am Beispiel des Knotenfärbungsproblems darstellen. Hierüber wurde in der Vorlesung ein Überblick gegeben, aber nicht schriftlich ausgearbeitet. Der Überblick hat sich an dem bereits zitierten Aufsatz von Johnson et al. (1991) orientiert sowie an dem Übersichtsartikel von Culberson and Luo (1996).

Literaturverzeichnis

Aarts, E. and Lenstra, J. K. (1997). *Local Search in Combinatorial Optimization*. Wiley, Chichester, first edition.

Culberson, J. C. and Luo, F. (1996). Exploring the k-colorable Landscape with Iterated Greedy. In Johnson, D. S. and Trick, M. A., editors, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 26, pages 245–284, und Kapitel 5 des Buches West (1996). American Mathematical Society, Providence.

Johnson, D. S., Aragon, C. R., McGeoch, L. A., and Schevon, C. (1989). Optimization by Simulated Annealing: An Experimental Evaluation; Part I, Graph Partitioning. *Operations Research*, 37:865–892.

Johnson, D. S., Aragon, C. R., McGeoch, L. A., and Schevon, C. (1991). Optimization by Simulated Annealing: An Experimental Evaluation; Part II, Graph Coloring and Number Partitioning. *Operations Research*, 39(3):378–406.

Johnson, D. S. and Papadimitriou, C. H. (1985). Performance guarantees for heuristics. In Lawler, E. L., Lenstra, J. K., Kan, A. R., and Shmoys, D., editors, *The Travelling Salesman Problem*, pages 145–180. Wiley & Sons, Inc., New York.

Kirkpatrick, S., Jr., C. D. G., and Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, 220:671–680.

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

Reinelt, G. (1994). *The Traveling Salesman, Computational Solutions for TSP Applications*. Springer.

van Laarhoven, P. J. M. and Aarts, E. H. L. (1987). *Simulated Annealing:Theory and Applications*. D. Reidel Publishing Co., Dordrecht.

Kapitel 10

Gütemaße für Heuristiken

In diesem Kapitel wollen wir die Gütekriterien zur Bestimmung des Approximationsverhaltens von Heuristiken verfeinern.

Im vorhergehenden Kapitel haben wir Beispiele für heuristische Verfahren angegeben, für die gewisse Gütegarantien bewiesen werden können. Wir haben auch gesehen, dass es bei manchen Problemen überhaupt nicht möglich ist, mit polynomialem Rechenaufwand zu garantieren, dass eine fast optimale Lösung gefunden werden kann (falls $\mathcal{P} \neq \mathcal{NP}$). Ferner gibt es Probleme, bei denen bestmögliche Worst-Case Schranken existieren, die mit polynomialen Verfahren erreicht aber nicht verbessert werden können. Um diese (und andere) Fälle besser klassifizieren zu können, hat sich die folgende Terminologie gebildet.

(10.1) Definition. *Sei Π ein kombinatorisches Optimierungsproblem, und es sei $\Pi' := \{P \in \Pi \mid P \text{ hat eine zulässige Lösung}\}$. Für jedes Problembeispiel $P \in \Pi'$ bezeichne $c_{\text{opt}}(P)$ den Wert einer Optimallösung von P. (Um einige Fallunterscheidungen zu vermeiden, setzen wir im weiteren voraus, daß $c_{\text{opt}}(P) > 0$ gilt.) Ferner sei A ein Algorithmus, der für jedes Problembeispiel $P \in \Pi'$ eine zulässige Lösung liefert. Den Wert dieser Lösung bezeichnen wir mit $c_A(P)$. Für alle Problembeispiele $P \in \Pi \setminus \Pi'$ setzen wir $c_{\text{opt}}(P) := c_A(P) := 1$.*

(a) *Sei $\varepsilon > 0$ eine fest vorgegebene Zahl. Falls Π ein Maximierungsproblem ist, gelte zusätzlich $\varepsilon \leq 1$. Gilt für jedes Problembeispiel $P \in \Pi$*

$$R_A(P) := \frac{|c_A(P) - c_{\text{opt}}(P)|}{c_{\text{opt}}(P)} \leq \varepsilon\,,$$

so heißt A **ε-approximativer Algorithmus***, und die Zahl ε heißt* **Gütegarantie** *von A.*

(b) *Sei A ein ε-approximativer Algorithmus für Π.*

(b_1) *Ist* Π *ein Maximierungsproblem, heißt* $1 - \varepsilon$ *die* **Worst-Case-Schranke** *von* A *(denn dann gilt* $c_A(P) \geq (1 - \varepsilon)c_{\text{opt}}(P)$*).*

(b_2) *Ist* Π *ein Minimierungsproblem, heißt* $1 + \varepsilon$ *die* **Worst-Case-Schranke** *von* A *(denn dann gilt* $c_A(P) \leq (1 + \varepsilon)c_{\text{opt}}(P)$*).*

(c) Ein **Approximationsschema** *(***AS***) für* Π *ist ein Algorithmus* A*, der zwei Inputs hat, nämlich ein Problembeispiel* $P \in \Pi$ *und eine rationale Zahl* $\varepsilon > 0$*, und der für jedes* $P \in \Pi$ *und jedes* $\varepsilon > 0$ *eine Lösung produziert, die* $R_A(P) \leq \varepsilon$ *erfüllt.*

(d) Ein Approximationsschema A *für* Π *heißt* **polynomiales Approximationsschema** *(***PAS***), wenn die Laufzeit von* A *polynomial ist in der Inputlänge von* $P \in \Pi$.

(e) Ein Approximationsschema A *für* Π *heißt* **voll-polynomiales Approximationsschema** *(***FPAS***), wenn die Laufzeit von* A *polynomial in* $\langle P \rangle + \frac{1}{\varepsilon}$ *ist, also polynomial in der Inputlänge von* $P \in \Pi$ *und polynomial in* $\frac{1}{\varepsilon}$ *ist.*

□

Die Unterscheidung zwischen “Gütegarantie” und “Worst-Case- Schranke” ist etwas künstlich und häufig werden — auch von mir — die beiden Begriffe durcheinander gebracht.

(10.2) Beispiele.

(a) Der Greedy-Algorithmus, siehe (5.18), liefert die Gütegarantie q und ist somit ein Algorithmus mit Worst-Case-Schranke $(1 - q)$ für Maximierungsprobleme über allgemeine Unabhängigkeitssysteme (E, J, c), wobei q der Rangquotient ist.

(b) Die Christofides-Heuristik (9.5) (f) ist ein $\frac{1}{2}$-approximatives Verfahren für das euklidische symmetrische TSP.

(c) Ist $P \in \Pi$, hat P die Inputlänge n und hat ein Algorithmus A z. B. die Laufzeit $O\left(n^{\frac{1}{\varepsilon}}\right)$, so ist A ein polynomiales Approximationsschema. Hat A z. B. die Laufzeit $O\left(n^k \cdot \left(\frac{1}{\varepsilon}\right)^l\right)$, (mit Konstanten k und l) so ist A ein voll-polynomiales Approximationsschema.

□

Keine der Heuristiken, die wir bisher kennengelernt haben, ist ein polynomiales oder gar voll-polynomiales Approximationsschema. Die Gütegarantien gelten nur für ein bestimmtes ε (und alle größeren Werte).

Es stellt sich natürlich sofort die Frage, ob es für alle schwierigen Optimierungsprobleme möglich ist, ein polynomiales bzw. voll-polynomiales AS zu entwickeln. Wir werden dies im folgenden theoretisch untersuchen.

Man beachte, dass ein FPAS nicht polynomial im Gesamtinput ist. Nach Definition ist es zwar polynomial in der Inputlänge von $P \in \Pi$ und in $\frac{1}{\varepsilon}$, jedoch benötigt man zur Darstellung der rationalen Zahl $\varepsilon = \frac{p}{q}$ (bzw. $\frac{1}{\varepsilon}$) nur $\langle\varepsilon\rangle \approx \log p + \log q$ und nicht $p + q$ Speicherplätze.

(10.3) Satz. *Sei Π ein kombinatorisches Optimierungsproblem. Gibt es für Π ein FPAS, das auch polynomial in $\langle\varepsilon\rangle$ ist, so gibt es einen polynomialen Algorithmus zur Lösung aller Problembeispiele in Π.*

Beweis : Gibt es überhaupt ein FPAS, sagen wir A, so sind die Kodierungslängen sowohl des Wertes der Lösung $c_A(P)$ von A als auch des Optimalwertes $c_{\text{opt}}(P)$ polynomial in der Inputlänge von P. Ist A polynomial in $\langle\varepsilon\rangle$, so können wir einen polynomialen Algorithmus B für Π wie folgt konstruieren.

1. Setze $\varepsilon := \frac{1}{2}$ und wende A auf $P \in \Pi$ an. Wir erhalten einen Wert $c_{A_\varepsilon}(P)$.
2. Setze
   im Maximierungsfall $\quad \delta := \frac{1}{2c_{A_\varepsilon}(P)+1}$,
   im Minimierungsfall $\quad \delta := \frac{1}{c_{A_\varepsilon}(P)+1}$.
3. Wende A auf P mit Genauigkeitsparameter δ an. Wir erhalten eine Lösung $c_{A_\delta}(P)$.

Sei B der durch 1., 2. und 3. beschriebene Algorithmus, so ist die Laufzeit von B offensichtlich polynomial in der Inputlänge von P, denn $\varepsilon = \frac{1}{2}$ ist eine Konstante, somit sind $c_{A_\varepsilon}(P)$ und δ Größen deren Kodierungslängen polynomial in der Inputlänge von P sind. Folglich ist auch Schritt 3 polynomial in der Inputlänge von P.

Wir können o. B. d. A. annehmen, dass die Zielfunktion ganzzahlig ist. Somit sind auch die Werte der optimalen Lösungen und der durch B erzeugten Lösung ganzzahlig. Wir zeigen nun, dass $c_{\text{opt}}(P) = c_{A_\delta}(P)$ gilt.

Ist Π ein Maximierungsproblem, so gilt:

$$\begin{aligned} c_{A_\varepsilon}(P) \geq \tfrac{1}{2}c_{\text{opt}}(P) \quad &\Rightarrow 2c_{A_\varepsilon}(P) + 1 > c_{\text{opt}}(P) \\ c_{A_\delta}(P) \geq (1-\delta)c_{\text{opt}}(P) \quad &= \left(1 - \tfrac{1}{2c_{A_\varepsilon}(P)+1}\right)c_{\text{opt}}(P) \\ &> \left(1 - \tfrac{1}{c_{\text{opt}}(P)}\right)c_{\text{opt}}(P) \\ &= c_{\text{opt}} - 1\,. \end{aligned}$$

Da $c_{A_\delta}(P)$ ganzzahlig und echt größer als $c_{\text{opt}} - 1$ ist, folgt $c_{A_\delta}(P) = c_{\text{opt}}(P)$.

Ist Π ein Minimierungsproblem, so erhalten wir:

$$c_{\text{opt}}(P) < c_{A_\varepsilon}(P) + 1 \qquad \Rightarrow \frac{1}{c_{\text{opt}}(P)} > \frac{1}{c_{A_\varepsilon}(P)+1}$$

$$
\begin{aligned}
c_{\text{opt}}(P) \le c_{A_\delta}(P) \le (1+\delta)c_{\text{opt}}(P) &= \left(1 + \tfrac{1}{c_{A_\varepsilon}(P)+1}\right)c_{\text{opt}}(P) \\
&< \left(1 + \tfrac{1}{c_{\text{opt}}(P)}\right)c_{\text{opt}}(P) \\
&= c_{\text{opt}}(P) + 1\,.
\end{aligned}
$$

Aufgrund der Ganzzahligkeit folgt daraus die Behauptung.

□

In einem gewissen Sinne ist also ein FPAS das beste, was man als appoximatives Verfahren für ein $\mathcal{NP}$-vollständiges Problem erwarten kann.

Wir führen nun weitere Maßzahlen ein, die im Zusammenhang mit der Analyse approximativer Algorithmen interessant sind. (Die Definition dieser Zahlen ist in der Literatur nicht einheitlich.) Wir hatten in (10.1) (a) die Größe

$$R_A(P) := \frac{|c_A(P) - c_{\text{opt}}(P)|}{c_{\text{opt}}(P)}$$

definiert. Wir setzen nun:

(10.4) Definition. *Sei Π ein Optimierungsproblem.*

(a) Ist A ein approximativer Algorithmus für Π, dann heißt

$$R_A := \inf\{\varepsilon > 0 \mid R_A(P) \le \varepsilon, \quad \text{für alle } P \in \Pi\}$$

die **absolute Gütegarantie von** A, *und*

$$
\begin{aligned}
R_A^\infty := \inf\{\varepsilon > 0 \mid &\text{ Es gibt ein } n_0 \in \mathbb{N}, \text{ so dass } R_A(P) \le \varepsilon \\
&\text{ für alle } P \in \Pi \text{ mit } c_{\text{opt}} \ge n_0\},
\end{aligned}
$$

heißt die **asymptotische Gütegarantie von** A. *(Bei Maximierungsproblemen Π geschieht die Infimumsbildung über alle ε mit $0 < \varepsilon < 1$.)*

(b) $R_{\min}(\Pi) := \inf\{\varepsilon > 0$ (bzw. $0 < \varepsilon < 1$ bei Maximierungsproblemen) $\mid$ Es gibt einen polynomialen Approximationsalgorithmus A mit $R_A^\infty = \varepsilon\}$, heißt die **bestmögliche asymptotische Gütegarantie von** Π. *Es ist $R_{\min}(\Pi) = \infty$, falls kein solches ε existiert.*

□

(10.5) Beispiel. Die FIRST FIT Heuristik FF, siehe (11.14), für das Bin-Packing-Problem hat eine asymptotische Gütegarantie von $R^{\infty}_{FF} = \frac{7}{10}$. Die absolute Gütegarantie ist nicht $\frac{17}{10}$ wegen des konstanten Terms "2" in Satz (11.16).

□

$R_{\min}(\Pi)$ ist natürlich eine der wichtigsten Größen dieser Analysetechniken, da durch $R_{\min}(\Pi)$ die bestmögliche Approximierbarkeit beschrieben wird.

(10.6) Bemerkung. Für $R_{\min}(\Pi)$ sind drei Fälle von Bedeutung.

(a) $R_{\min}(\Pi) = \infty$,
d. h. für Maximierungsprobleme, dass für kein $0 < \varepsilon < 1$ ein polynomialer ε-approximativer Algorithmus existiert, und für Minimierungsprobleme, dass für kein $\varepsilon > 0$ ein polynomialer ε-approximativer Algorithmus existiert.

(b) $R_{\min}(\Pi) > 0$,
d. h. es gibt eine Schranke für die Gütegarantie, die mit polynomialem Aufwand nicht überwunden werden kann.

(c) $R_{\min}(\Pi) = 0$, d. h. das Optimum kann durch polynomiale Algorithmen beliebig approximiert werden. Offensichtlich kann nur ein Problem mit $R_{\min}(\Pi) = 0$ ein PAS oder ein FPAS haben. Zum Entwurf von PAS bzw. FPAS für ein Problem Π müssen wir also zunächst die asymptotische Gütegarantie kennen. Hier gibt es vier Unterfälle:

- (c_1) Es gibt ein PAS für Π.
- (c_2) Es gibt ein FPAS für Π.
- (c_3) Es gibt einen polynomialen Approximationsalgorithmus A mit $R^{\infty}_{A} = 0$.
- (c_4) Es gibt einen polynomialen Approximationsalgorithmus A mit

$$|c_A(P) - c_{\text{opt}}(P)| \leq K \quad \text{für ein festes } K \text{ und alle } P \in \Pi.$$

Eine solche Garantie nennt man auch **Differenzgarantie**.

□

Offensichtlich ist die beste aller Gütegarantien, die man erwarten darf, durch (c_4) beschrieben.

Der Fall (10.6) (a) trifft auf das symmetrische Travelling-Salesman-Problem zu, wie Satz (9.7) zeigt. Wir wollen nun ein Problem einführen, eine bestmögliche Heuristik angeben und zeigen, dass für dieses Problem (10.6) (b) zutrifft. Den Fall (10.6) (c) behandeln wir im nächsten Kapitel.

(10.7) Das k-Zentrumsproblem.

Gegeben sei ein vollständiger Graph $K_n = (V, E)$ mit Kantengewichten c_e für alle $e \in E$. Gesucht ist eine Knotenmenge (die Zentren) $S \subseteq V$ mit höchstens k Elementen, so dass

$$c(S) := \max_{i \in V} \min_{j \in S} c_{ij}$$

minimal ist.

□

Anwendungen dieses Standortproblems sind offensichtlich. Man möchte z. B. höchstens k Feuerwehrdepots so verteilen, dass die maximale Entfernung irgendeines möglichen Brandherdes zu einem Depot möglichst klein wird. Damit soll z. B. gewährleistet werden, dass jeder Brandherd in einer bestimmten maximalen Zeit erreichbar ist.

Dieses Problem ist $\mathcal{NP}$-schwer, selbst dann, wenn die Dreiecksungleichung $c_{ij} + c_{jk} \geq c_{ik}$ für alle $i, j, k \in V$ gilt. Man kann sogar zeigen:

(10.8) Satz. *Es gibt ein ε mit $0 < \varepsilon < 1$ und einen polynomialen Algorithmus A, der für jedes k-Zentrumsproblem mit Dreiecksungleichung eine Lösung S_A liefert, die bezüglich der Optimallösung S_{opt} folgende Gütegarantie hat*

$$c(S_{\text{opt}}) \leq c(S_A) \leq (1 + \varepsilon)c(S_{\text{opt}}),$$

genau dann, wenn $\mathcal{P}$=$\mathcal{NP}$ gilt.

Beweis : Siehe Hsu and Nemhauser (1979). □

Mit anderen Worten: Das ε-Approximationsproblem für das k-Zentrumsproblem mit Dreiecksungleichung ist für $0 < \varepsilon < 1$ $\mathcal{NP}$-vollständig. Bemerkenswert ist nun, dass man die Gütegarantie $\varepsilon = 1$ in polynomialer Zeit tatsächlich erreichen kann. Wir betrachten dazu den folgenden Algorithmus.

(10.9) 1-approximative Heuristik für das k-Zentrumsproblem.

Input: *Vollständiger Graph $K_n = (V, E)$, mit Gewichten c_e für alle $e \in E$.*

Output: *Eine Menge $S \subseteq V$ mit $|S| \leq k$.*

1. *Wir ordnen die Kanten so, dass $c_{e_1} \leq c_{e_2} \leq \ldots \leq c_{e_m}$ gilt und setzen $k := 0$.*
2. *Für jeden Knoten $v \in V$ legen wir eine Adjazenzliste $\text{ADJ}(v)$ an, auf der die Nachbarn u von v in nicht-absteigender Reihenfolge bezüglich der Kantengewichte c_{uv} auftreten. Mit $G_i = (V, E_i)$ bezeichnen wir den Untergraphen von K_n mit Kantenmenge $E_i = \{e_1, e_2, \ldots, e_i\}$.*

3. *Falls $k = |V|$, dann gib V aus und STOP.*
4. *LOW := 1 (bedeutet: S kann die Menge V sein).*
   HIGH := m (bedeutet: S besteht nur aus einem Element).
5. *(Binäre Suche)*
   Ist HIGH = LOW + 1 gehe zu 9.
6. *Setze $i := \lfloor \frac{HIGH+LOW}{2} \rfloor$ und konstruiere $G_i = (V, E_i)$ und die zugehörigen (sortierten) Adjazenzlisten $\mathrm{ADJ}_i(v)$. Setze $S := \emptyset$ und $T := V$.*
7. *Ist $T \neq 0$, so wähle einen Knoten $v \in T$ und führe aus:*

$$\begin{aligned} S: &= S \cup \{v\}, \\ T: &= T \setminus (\{v\} \cup \{w \mid w \text{ Nachbar von } v \text{ in } G_i\}) . \end{aligned}$$

   und gehe zu 7.
8. *Falls $|S| \leq k$, dann setze HIGH := i und $S' := S$, andernfalls LOW := i.*
   Gehe zu 5.
9. *Gib S' aus.*

□

Hochbaum and B.Shmoys (1985) haben gezeigt:

(10.10) Satz. *Der Algorithmus (10.10) liefert für k-Zentrumsprobleme, bei denen die Dreiecksungleichung gilt, eine Knotenmenge $S \subseteq V$, $|S| \leq k$, mit $c(S) \leq 2c(S_{\text{opt}})$ in $O(|E| \log |E|)$ Zeit.*

□

Dieser Algorithmus — eine Mischung aus Greedy und binärer Suche — ist also in der Tat bestmöglich unter allen approximativen Algorithmen für das k-Zentrumsproblem mit Dreiecksungleichung. Aus (10.10) und (10.8) folgt:

(10.11) Folgerung. *Sei Π das k-Zentrumsproblem mit Dreiecksungleichung, dann gilt*

$$R_{\min}(\Pi) = 1 .$$

□

Literaturverzeichnis

Hochbaum, D. S. and B.Shmoys, D. (1985). A best possible heuristic for the k-center problem. *Mathematics of Operations Research*, 10:180–184.

Hsu, W. L. and Nemhauser, G. L. (1979). Easy and hard bottleneck location problems. *Discrete Applied Mathematics*, 1:209–216.

Kapitel 11

Weitere Heuristiken

Wir beschäftigen uns in diesem Kapitel vornehmlich mit schwierigen (d. h. $\mathcal{NP}$-vollständigen) kombinatorischen Optimierungsproblemen. Da — wenn man der Komplexitätstheorie glauben darf — wohl niemals effiziente Lösungsverfahren für derartige Probleme gefunden werden dürften, ist es besonders wichtig, Algorithmen zu entwerfen, die zulässige Lösungen generieren, deren Werte "nahe" beim Optimalwert liegen, und die in der Praxis schnell arbeiten. Verfahren, die zulässige, aber nicht notwendig optimale Lösungen liefern nennt man **Heuristiken**, gelegentlich spricht man auch von **Approximationsverfahren** (besonders dann, wenn man genaue Informationen über die Qualität der Lösungen hat, die das Verfahren produziert). In diesem Kapitel wollen wir (relativ unsystematisch) einige Beispiele für Heuristiken angeben und sie analysieren. Hierbei soll die Technik der Analyse von Heuristiken geübt werden, und gleichzeitig werden auch einige weitere kombinatorische Optimierungsprobleme vorgestellt.

Heuristiken liefern bei Maximierungsproblemen (bzw. Minimierungsproblemen) untere (bzw. obere) Schranken für den Wert einer Optimallösung. Um die Güte einer heuristischen Lösung abschätzen zu können, bedient man sich häufig weiterer Heuristiken, die dann obere (bzw. untere) Schranken für den Optimalwert liefern. Diese Heuristiken nennt man **duale Heuristiken** und spricht dann auch von **primalen Heuristiken**, um die dualen Heuristiken von den oben eingeführten Verfahren zu unterscheiden. Das Wort "primal" in diesem Zusammenhang kommt daher, dass diese Verfahren zulässige Lösungen für das Ausgangsproblem, also in der LP-Theorie das primale Problem liefern. Die Lösungen sind also primal zulässig.

11.1 Maschinenbelegung mit unabhängigen Aufgaben

Wir betrachten das folgende Problem der Maschinenbelegungsplanung:

11.1 Parallel-Shop-Problem.

(a) Gegeben seien m identische **Maschinen** oder Prozessoren
$$M_1, M_2, \dots, M_m$$
(z. B. Drucker eines Computers, Offset-Drucker, Walzstraßen, Pressen, Stanzen).

(b) Gegeben seien n **Aufgaben** (oder Aufträge oder Operationen oder Jobs)
$$T_1, T_2, \dots, T_n$$
(z. B. Druckjobs, -aufträge, zu walzende Profile, Press- und Stanzaufträge).

(c) Jeder Auftrag hat eine **Ausführungszeit** (oder Bearbeitungszeit)
$$t_1, t_2, \dots, t_n, \quad \text{(in Zeiteinheiten)},$$
und jeder Auftrag benötigt nur eine Maschine zu seiner Bearbeitung.

(d) Die Maschinen arbeiten parallel, und jede kann nur eine Aufgabe gleichzeitig erledigen. Hat eine Maschine die Ausführung einer Aufgabe begonnen, so führt sie diese bis zum Ende der Bearbeitungszeit durch.

□

Ein Maschinenbelegungsproblem dieser Art kann man einfach durch die Angabe einer Zahlenfolge auf folgende Weise beschreiben:

$$m, n, t_1, t_2, \dots, t_n.$$

(11.2) Beispiel. Wir betrachten die Zahlenfolge

$$3,\ 11,\ 2,\ 4,\ 3,\ 4,\ 4,\ 5,\ 2,\ 1,\ 4,\ 3,\ 2\,.$$

Das heißt, wir haben $m = 3$ Maschinen und $n = 11$ Aufträge: T_1 T_2 T_3 T_4 T_5 T_6 T_7 T_8 T_9 T_{10} T_{11} zu bearbeiten. Ein möglicher *Belegungsplan* der drei Maschinen wäre der folgende:

$$\begin{array}{lll} M_1 & \text{bearbeitet} & T_1, T_2, T_3, T_4\,, \\ M_2 & \text{bearbeitet} & T_5, T_6, T_9\,, \\ M_3 & \text{bearbeitet} & T_7, T_8, T_{10}, T_{11}\,. \end{array}$$

Einen solchen Belegungsplan kann man bequem in einem Balkendiagramm wie folgt darstellen.

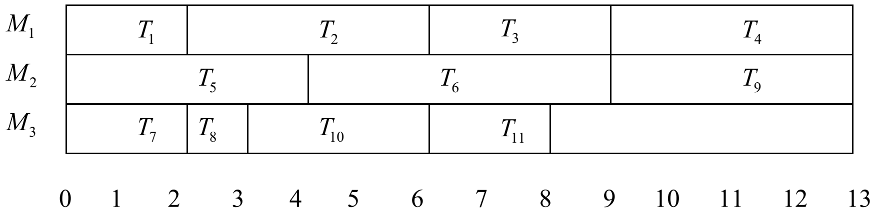


Abb. 11.1

Insgesamt benötigt man also bei diesem Belegungsplan zur Bearbeitung aller Aufträge 13 Zeiteinheiten, wobei jedoch die dritte Maschine 5 Zeiteinheiten leer steht.

$\square$

Ein sinnvolles und für die Praxis relevantes Optimierungsproblem besteht nun darin, einen Belegungsplan zu finden, so dass der Gesamtauftrag so früh wie möglich fertiggestellt ist. Zur vollständigen Formulierung von (9.1) fahren wir also wie folgt fort:

(11.1) (Fortsetzung)

(e) Ist ein Belegungsplan gegeben, so bezeichnen wir mit
$$C_1, C_2, \ldots, C_m$$
die **Fertigstellungsdauer** (-zeit) der auf den Maschinen $M_1, \ldots, M_m$ ausgeführten Aufträge.

(f) **Optimierungsziel:** Finde einen Belegungsplan, so dass die größte Fertigstellungsdauer so klein wie möglich ist. Dafür schreiben wir:
$$\min_{\text{Belegungspl.}} \max_{i=1,\ldots,m} C_i$$
oder kurz $\min C_{\max}$ mit $C_{\max} = \max\{C_i \mid i = 1, \ldots, m\}$.

$\square$

Der Belegungsplan in Abb. 11.2 für das Beispiel (11.2) ergibt eine bessere Lösung als der in Abb. 11.1 angegebene.

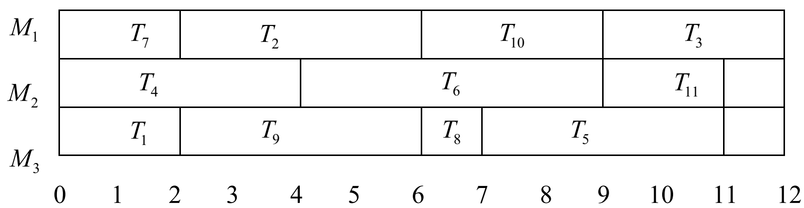


Abb. 11.2

In Abb. 11.2 werden insgesamt nur 12 Zeiteinheiten benötigt. Dieser Belegungsplan ist sogar optimal, denn insgesamt sind 34 Zeiteinheiten auf 3 Maschinen zu verteilen, und es ist offenbar unmöglich, 34 Zeiteinheiten auf 3 Maschinen zu jeweils 11 Zeiteinheiten Bearbeitungsdauer aufzuteilen. Das hier vorgestellte Maschinenbelegungsproblem (11.1) ist schwierig im Sinne der Komplexitätstheorie. Selbst das Problem mit 2 parallelen identischen Maschinen ist bereits $\mathcal{NP}$-vollständig.

Es ist daher sinnvoll, nach heuristischen Verfahren zu suchen, die schnell arbeiten und einen gewissen Grad an Genauigkeit garantieren. Wir wollen zunächst die simpelste Heuristik untersuchen, die man für dieses Problem angeben kann.

(11.3) Die LIST-Heuristik. *Gegeben sei ein Parallel Shop Problem durch*

$$m, n, t_1, t_2, \ldots, t_n.$$

1. **Initialisierung:** *Belege die Maschinen* $M_1, M_2, \ldots, M_m$ *mit den Aufgaben* $T_1, T_2, \ldots, T_m$. *Setze* $i := m + 1$.
2. *Hat eine der Maschinen* M_j *ihre gegenwärtige Aufgabe erledigt, so belege* M_j *mit der Aufgabe* T_i. *Setze* $i := i + 1$ *und wiederhole 2.*

□

Für unser Beispiel (11.2) ergibt die LIST-Heuristik (11.3) den in Abb. 11.3 angegebenen Belegungsplan, der offenbar wiederum optimal ist.

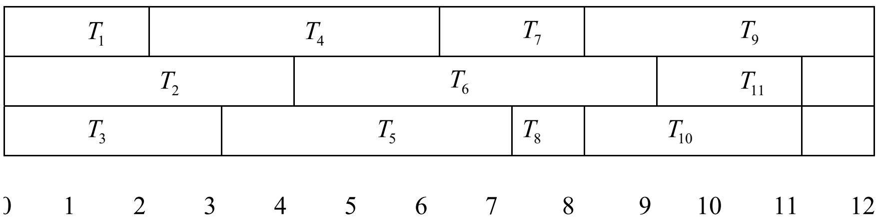


Abb. 11.3

Das günstige Ergebnis bei Beispiel (11.2) sollte jedoch nicht zu Fehlschlüssen führen. Es gibt schlechte Beispiele, die zu recht ungünstigen Belegungsbeispielen führen.

(11.4) Beispiel.

$$m = 6,\ n = 11,\ \underbrace{5, 5, 5, 5, 5}_{5\times},\ \underbrace{1, 1, 1, 1, 1}_{5\times},\ 6\,.$$

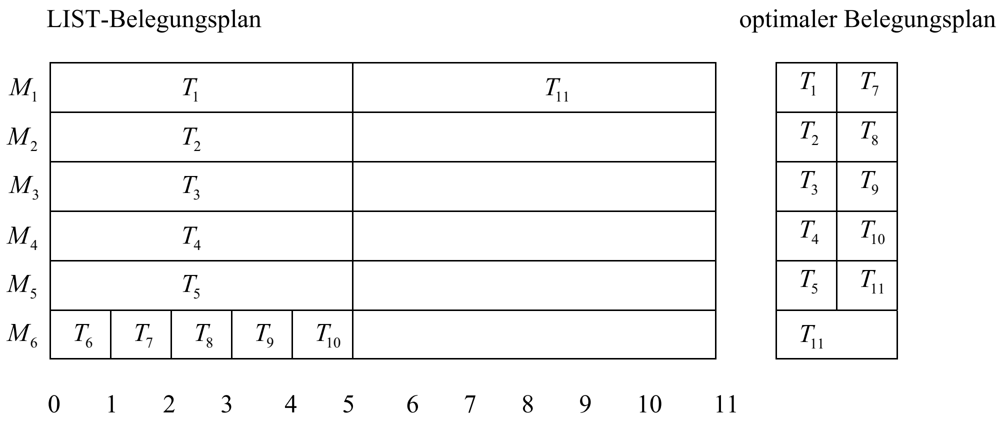


Abb. 11.4

□

Ganz allgemein kann man Beispiele angeben, mit $n = 2m - 1$, bei denen die Fertigstellungszeit C_L der LIST-Heuristik gleich $2m - 1$ ist, während die optimale Fertigstellungszeit C_{opt} gleich m ist. (Simple Modifikation von Beispiel (11.4).) Dies ist jedoch der schlechtest mögliche Fall wie der folgende Satz zeigt.

(11.5) Satz. *Gegeben sei ein Parallel-Shop-Problem mit m Maschinen und n Aufträgen. Sei C_L die Fertigstellungszeit der LIST-Heuristik und C_{opt} die optimale Fertigstellungszeit, dann gilt*

$$C_L \leq \left(2 - \tfrac{1}{m}\right) C_{\text{opt}}.$$

□

Der Beweis von Satz (11.5) folgt direkt aus dem nachfolgenden Lemma (11.6) zusammen mit der Überlegung, dass $C_{\text{opt}} \geq t$, wobei t die Bearbeitungszeit der Aufgabe ist, die als letzte im LIST-Belegungsplan ausgeführt wird.

(11.6) Lemma. *Ist t die Bearbeitungszeit der Aufgabe, die als letzte im LIST-Belegungsplan ausgeführt wird, dann gilt*

$$C_L \leq C_{\text{opt}} \left(1 + \tfrac{(m-1)t}{m C_{\text{opt}}}\right).$$

Beweis : Zum Zeitpunkt $C_L - t$ ist aufgrund der Definition der LIST-Heuristik keine Maschine leer. Eine beendet gerade ihre gegenwärtige Aufgabe und wird mit der Aufgabe der Länge t belegt. Es gilt also:

$$\begin{aligned}
mC_{\text{opt}} &\geq \textstyle\sum_{i=1}^{n} t_i \\
\sum_{i=1}^{n} t_i - t &\geq m(C_L - t) \\
\Longrightarrow \; mC_{\text{opt}} - t &\geq mC_L - mt \Longrightarrow C_L \leq C_{\text{opt}} + \frac{m-1}{m}t \\
&\Longrightarrow C_L \leq C_{\text{opt}}\Big(1 + \frac{(m-1)t}{mC_{\text{opt}}}\Big)
\end{aligned}$$

□

Wir betrachten nun eine geringfügig modifizierte Version von LIST:

(11.7) LIST DECREASING (LD). *Gegeben sei ein Parallel-Shop Problem*

$$m, n, t_1, t_2, \ldots, t_n.$$

1. *Ordne die Aufgaben, so dass gilt* $t_1 \geq t_2 \geq \ldots \geq t_m$.
2. *Wende LIST an.*

□

Ein Problem, bei dem LIST DECREASING ein relativ schlechtes Ergebnis liefert, ist das folgende.

(11.8) Beispiel.

$$m = 6, \; n = 13, 11, 11, 10, 10, 9, 9, 8, 8, 7, 7, 6, 6, 6$$

LD ergibt hier einen Belegungsplan mit $C_{LD} = 23$, während das Optimum 18 beträgt.

Optimaler Belegungsplan:
$$\begin{aligned}
M_1 &: \quad T_1, T_{10} \\
M_2 &: \quad T_2, T_9 \\
M_3 &: \quad T_3, T_8 \\
M_4 &: \quad T_4, T_7 \\
M_5 &: \quad T_5, T_6 \\
M_6 &: \quad T_{11}, T_{12}, T_{13}
\end{aligned}$$

□

I. a. gibt es Beispiele mit $n = 2m + 1$, $C_{LD} = 4m - 1$ und $C_{\text{opt}} = 3m$. Jedoch sind diese die schlechtest möglichen Beispiele. Um dies zu zeigen, beginnen wir mit einem Hilfssatz.

(11.9) Lemma. *Gilt $t_1 \geq t_2 \geq \dots t_n > \frac{C_{\text{opt}}}{3}$, dann ist $C_{LD} = C_{\text{opt}}$.*

Beweis : Offenbar bearbeitet im optimalen Belegungsplan jede Maschine höchstens zwei Aufgaben (andernfalls gilt $3t_n > C_{\text{opt}}$). Zur Vereinfachung der Beweisführung führen wir $2m - n$ Aufgaben mit Bearbeitungszeit Null ein, so dass jede Maschine 2 Aufgaben erledigt, wobei wir die Bearbeitungsdauern der Aufgaben auf M_i mit a_i und b_i bezeichnen wollen und $a_i \geq b_i$ gelten soll. Also gilt

$$a_1 \geq a_2 \geq \dots \geq a_m$$

$$a_1 \geq b_1, \dots, a_m \geq b_m .$$

Daraus folgt natürlich $a_1 = t_1$. Falls $a_i = t_i$ für $i = 1, \dots, k-1$ aber $a_k < t_k$, dann gilt natürlich $t_k = b_i$ für ein $i \leq k-1$, denn $t_k > a_k \geq a_{k+1} \dots \geq a_m$ und $a_j \geq b_j$ für $j = k, \dots, n$. Tauschen wir nun b_i und a_k aus, so erhalten wir wiederum einen optimalen Belegungsplan, denn $C_{\text{opt}} \geq a_i + b_i > a_i + a_k \geq b_i + b_k$. Nach höchstens m Austauschoperationen dieser Art erhalten wir einen optimalen Belegungsplan mit

$$a_1 = t_1,\ a_2 = t_2, \dots,\ a_m = t_m.$$

Analog können wir durch Austauschen erreichen, dass in einer optimalen Lösung $b_m = t_{m+1}, b_{m-1} = t_{m+2}, \dots, b_1 = t_{2m}$ gilt. Dies ist aber eine Lösung, die LD liefert, und wir sind fertig. □

(11.10) Satz.

$$C_{LD} \leq \tfrac{1}{3}(4 - \tfrac{1}{m})C_{\text{opt}} .$$

Beweis : Angenommen, die Behauptung stimmt nicht, dann gibt es ein Gegenbeispiel mit kleinstmöglichem n. Es gilt $t_1 \geq t_2 \geq \dots \geq t_n$. Wir behaupten, dass T_n die letzte erledigte Aufgabe im LD-Belegplan ist. Wäre dies nicht so, dann könnten wir T_n aus unserem Beispiel entfernen, wobei die LD-Lösung des neuen Beispiels unverändert bliebe, die Optimallösung jedoch nicht schlechter wäre. Dies wäre ein Gegenbespiel zur Behauptung mit $n-1$ Aufgaben. Wenden wir nun Lemma (11.6) (mit $t = t_n$) an und benutzen wir die Annahme, dass wir mit einem Gegenbeispiel arbeiten, so erhalten wir:

$$\begin{aligned}
\frac{1}{3}(4 - \frac{1}{m})C_{\text{opt}} \quad &< \quad C_{LD} \leq \Big(1 + \frac{(m-1)t_n}{mC_{\text{opt}}}\Big)C_{\text{opt}} \\
\Longrightarrow \frac{4}{3} - \frac{1}{3m} \quad &< \quad \frac{C_LD}{C_{\text{opt}}} \leq 1 + \frac{m-1}{m}\frac{t_n}{C_{\text{opt}}} \\
\Longrightarrow \frac{1}{3}\frac{4m-1}{m} \quad &< \quad \frac{(m-1)t_n}{m}\frac{1}{C_{\text{opt}}} \\
\frac{1}{3}C_{\text{opt}} \quad &< \quad \frac{(m-1)t_n \cdot m}{(4m-1)m} \leq t_n .
\end{aligned}$$

Aus Lemma (11.9) folgt nun aber, dass $C_{LD} = C_{\text{opt}}$ gelten muss, ein Widerspruch. □

11.2 Maschinenbelegung mit abhängigen Aufgaben

Wir betrachten wiederum ein Parallel-Shop Problem wie in Abschnitt 11.1, das durch die Anzahl m der Maschinen, die Anzahl n der Aufgaben $T_1, \dots, T_n$ und die Bearbeitungszeiten $t_1, \dots, t_n$ beschrieben ist:

$$n, m, t_1, \dots, t_n.$$

Eine zusätzliche Komplikation wird dadurch erzeugt, dass gewisse Aufgaben nicht unabhängig voneinander sind, d. h. dass eine Aufgabe T_i erst in Angriff genommen werden kann, wenn eine andere Aufgabe T_j vollständig erledigt ist. Solche Beschränkungen nennt man **Reihenfolgebedingungen**.

Schreiben wir symbolisch $T_j \longrightarrow T_i$ dafür, dass Aufgabe T_i erst beginnen kann, wenn Aufgabe T_j beendet ist, so kann man alle Reihenfolgebedingungen offenbar in einem **Reihenfolgedigraphen** darstellen. Ein solcher ist in Abbildung 11.5 angegeben.

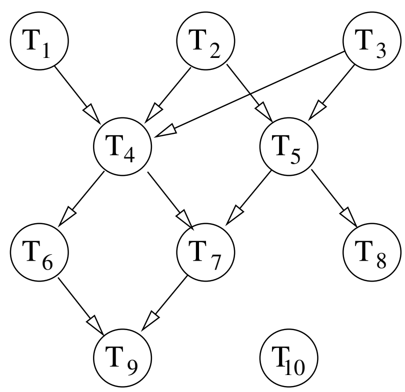


Abb. 11.5

Bei einem Reihenfolgedigraphen sind nur unmittelbare Reihenfolgebedingungen dargestellt, z. B. $T_1 \longrightarrow T_4 \longrightarrow T_6 \longrightarrow T_9$. Daraus folgt natürlich, dass $T_1 \longrightarrow T_9$ gilt, obwohl dies im Diagramm nicht dargestellt ist.

Fügen wir zu einem Reihenfolgedigraphen (bzw. zu einem allgemeinen Digraphen) alle implizierten Reihenfolgebedingungen hinzu, so nennt man diesen Digraphen den **transitiven Abschluss** des Reihenfolgedigraphen (bzw. Digraphen).

Natürlich ist das Parallel-Shop Problem mit Reihenfolgebedingungen ebenfalls $\mathcal{NP}$-schwer, und wir zeigen nun, dass die LIST-Heuristik auch für diesen Problemtyp gute Ergebnisse liefert.

(11.11) Die LIST-Heuristik bei Reihenfolgebedingungen. *Gegeben sei ein Parallel-*

Shop Problem durch

$$m, n, t_1, \dots, t_n$$

und durch einen Reihenfolgedigraphen D.

1. *Belege die Maschine M_1 mit der ersten verfügbaren Aufgabe, sagen wir T_1.*

2. *Hat eine Maschine ihre gegenwärtige Aufgabe erledigt, so weise ihr die nächste verfügbare Aufgabe zu. Falls aufgrund der Reihenfolgebedingungen keine Aufgabe zur Verfügung steht, bleibt die Maschine leer, bis die nächste Aufgabe freigegeben wird.*

$\square$

(11.12) Beispiel. Betrachten wir das Parallel-Shop Problem

$$3, 10, 1, 2, 1, 2, 1, 1, 3, 6, 2, 1$$

mit dem Reihenfolgedigraphen aus Abbildung 11.5, so erhalten wir als LIST-Lösung den in Abbildung 11.6 angegebenen Belegungsplan mit $C_L = 9$.

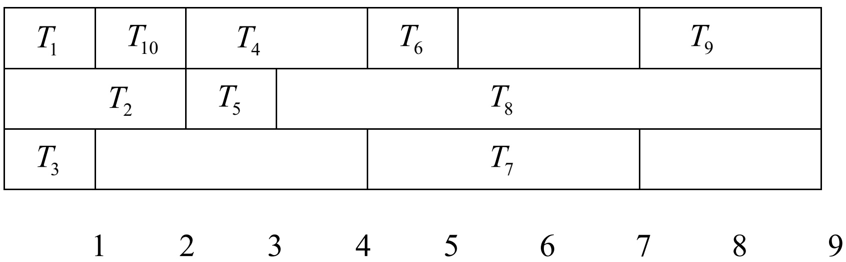


Abb. 11.6

$\square$

Es ist erstaunlich, dass die Schranke aus Satz (11.5) auch für diesen komplizierteren Fall gilt.

(11.13) Satz. *Gegeben sei ein Parallel-Shop Problem mit Reihenfolgebedingungen. Sei C_L die Fertigstellungszeit der LIST-Heuristik und C_{opt} die optimale Fertigstellungszeit. Dann gilt*

$$C_L \le (2 - \tfrac{1}{m}) C_{\text{opt}}.$$

Beweis : Wir bezeichnen mit T_1^* eine Aufgabe, die im LIST-Belegplan als letzte endet. Sei k die größte ganze Zahl, für die es eine Menge von Aufgaben

$$T_k^* \longrightarrow T_{k-1}^* \longrightarrow \ldots \longrightarrow T_1^*$$

gibt, so dass die folgende Eigenschaft erfüllt ist:

> Falls zu irgendeinem Zeitpunkt zwischen der Startzeit von T_k^* und der Endzeit C_L von T_1^* eine Maschine leer ist, dann wird irgendeine der Aufgaben T_i^* bearbeitet.

Es ist offensichtlich, dass es überhaupt so eine Folge $T_k^* \longrightarrow \ldots \longrightarrow T_1^*$ gibt. Der Hauptschritt des Beweises ist der Beweis der folgenden Behauptung.

> Zu jedem Zeitpunkt vor dem Beginn der Ausführung von T_k^* sind alle m Maschinen belegt.

Nehmen wir an, dass diese Behauptung falsch ist. Dann finden wir einen letzten Zeitpunkt vor dem Beginn von T_k^*, zu dem eine Maschine leer steht. Warum bearbeitet diese Maschine nicht T_k^*? Die einzige mögliche Antwort ist, dass es eine Aufgabe T_j gibt mit $T_j \longrightarrow T_k^*$, die noch nicht beendet ist. Setzten wir nun $T_{k+1}^* := T_j$, so haben wir eine längere Kette von Aufgaben mit den gewünschten Eigenschaften. Ein Widerspruch zur Maximalität von k.

Sei nun w_i die gesamte Leerzeit der Maschine M_i, für $i = 1, \ldots, m$ dann gilt

$$\sum_{j=1}^{n} t_j + \sum_{i=1}^{m} w_i = mC_L.$$

Sei t die Gesamtausführungszeit der Aufgaben $T_k^*, \ldots, T_1^*$, dann implizieren die beiden Eigenschaften der Kette $T_k^* \longrightarrow \ldots \longrightarrow T_1^*$

$$\sum_{i=1}^{m} w_i \leq (m-1)t\,.$$

(Die Leerzeiten fallen erst nach Beginn von T_k^* an, dann ist aber immer eine Maschine beschäftigt.) Also erhalten wir

$$\begin{array}{rl} mC_L & \leq \sum_{j=1}^{n} t_j + (m-1)t \leq mC_{\text{opt}} + (m-1)C_{\text{opt}} \\ \Longrightarrow\ C_L & \qquad\qquad\qquad \leq \big(1 + \frac{m-1}{m}\big)C_{\text{opt}}. \end{array}$$

□

Leider ist kein zu Satz (11.10) analoges Resultat für Probleme mit Reihenfolgebedingungen bekannt. **Forschungsproblem**: Welche Gütegarantie hat die LIST DECREASING Heuristik für das Parallel-Shop-Problem mit Reihenfolgebedingungen?

11.3 Das Packen von Kisten (Bin-Packing)

Das Problem, das wir nun behandeln wollen, ist in gewissem Sinne dual zum Parallel-Shop-Problem mit unabhängigen Aufgaben. Bei diesem haben wir Aufgaben und Maschinen gegeben und suchen eine möglichst kurze Gesamtfertigstellungszeit. Beim **Bin-Packing-Problem** hat man dagegen eine Anzahl von Aufgaben und eine späteste Fertigstellungszeit vorgegeben. Gesucht ist eine minimale Zahl von Maschinen, um die Aufgaben in der vorgeschriebenen Zeit zu erledigen.

Es ist üblich, dieses Problem als Kistenpackungsproblem zu beschreiben. Jede Kiste habe eine Höhe d und jeder Gegenstand eine Höhe t_i (natürlich kann man auch Gewichte etc. wählen). Das Ziel ist, eine minimale Zahl von Kisten zu finden, die alle Gegenstände aufnehmen. Für dieses Problem betrachten wir die folgende Heuristik.

(11.14) FIRST-FIT (FF). *Gegeben sei ein Bin-Packing-Problem durch die Folge $d, t_1, \ldots, t_n$. Die Gegenstände werden in der Reihenfolge, wie sie in der Liste vorkommen, behandelt. Jeder Gegenstand wird in die erste Kiste, in die er passt, gelegt.*

□

(11.15) Beispiel. Gegeben sei ein Bin-Packing-Problem durch die Kistenhöhe $d = 101$ und die folgenden 37 Gegenstände: 6 (7mal), 10 (7mal), 16 (3mal), 34 (10mal), 51 (10mal). Die FF-Lösung ist in Abbildung 11.7 dargestellt.

	6 (7 x)	10 (5 x) = 92
	10 (2 x)	16 (3 x) = 68
5 x {	34 (7 x)	
10 x {	51 (1 x)	

Abb. 11.7

Das heißt, bei der FF-Lösung werden 17 Kisten benötigt. Die optimale Lösung benötigt nur 10 Kisten, wie Abbildung 11.8 zeigt.

3 x	51, 34, 16,
7 x	51, 34, 10, 6

Abb. 11.8

□

Daraus folgt, dass $\frac{m_{FF}}{m_{\text{opt}}}$ (m_{FF} ist die durch FF bestimmte Kistenzahl und m_{opt} die optimale Kistenzahl m) durchaus $\frac{17}{10}$ sein kann. Die folgende Abschätzung

$$m_{FF} \leq 1 + 2m_{\text{opt}}$$

ist trivial, denn die FF-Lösung füllt jede benutzte Kiste (außer vielleicht der letzten) im Durchschnitt mindestens bis zur Hälfte, also gilt:

$$\tfrac{d}{2}(m_{FF} - 1) \leq \sum_{j=1}^{n} t_j \leq m_{\text{opt}} \cdot d.$$

Es ist erheblich schwieriger zu zeigen, dass der im Beispiel (11.15) gefundene Bruch $\frac{17}{10}$ asymptotisch der beste ist.

(11.16) Satz. *Gegeben sei ein Bin-Packing-Problem. m_{FF} sei die durch FIRST FIT gefundene Kistenzahl, m_{opt} die optimale, dann gilt*

$$m_{FF} < \tfrac{17}{10} m_{\text{opt}} + 2.$$

□

Im Beweis von Satz (11.16) benutzen wir die Existenz einer Funktion

$$w : [0,1] \longrightarrow [0,1],$$

die die folgenden Eigenschaften hat:

Eigenschaft 1:

$$\tfrac{1}{2} < t \leq 1 \quad \Longrightarrow \quad w(t) = 1\,.$$

Eigenschaft 2:

$$\left.\begin{array}{rcl} k & \geq & 2 \\ t_1 \geq t_2 \geq \ldots & \geq & t_k \\ \sum_{i=1}^{k} t_i & \leq & 1 \\ \sum_{i=1}^{k} w(t_i) & = & 1 - s,\ s > 0 \end{array}\right\} \quad \Longrightarrow \quad \sum_{i=1}^{k} t_i \leq 1 - t_k - \tfrac{5}{9}s\,.$$

Eigenschaft 3:

$$\sum_{i=1}^{k} t_i \leq 1 \qquad \Longrightarrow \quad \sum_{i=1}^{k} w(t_i) \leq \tfrac{17}{10}\,.$$

Wir werden später die Existenz einer solchen Funktion beweisen und nehmen zum Beweis von (11.16) zunächst die Existenz von w an.

Beweis von Satz (11.16). Wir nehmen an, dass alle Kisten die Höhe 1 und die Gegenstände eine Höhe von $t'_i = \frac{t_i}{d}$ haben ($1 \leq i \leq n$). Wir nehmen zusätzlich an, dass jeder Gegenstand i ein **Gewicht** $w(t'_i)$ hat. Das Gewicht einer Kiste sei die Summe der Gewichte der in sie gelegten Gegenstände.

Betrachten wir die FF-Lösung, so bezeichnen wir mit $B^*_1, \ldots, B^*_m$ diejenigen Kisten (in ihrer Originalreihenfolge), die ein Gewicht kleiner als 1 haben, und zwar habe B^*_i das Gewicht $1 - s_i$, $s_i > 0$. Die ungenutzte Höhe von B^*_i bezeichnen wir mit c_i. Wir setzen $c_0 = 0$ und behaupten

$$\text{(1)} \qquad c_i \geq \tfrac{5}{9} s_i + c_{i-1} \quad \text{für} \quad i = 1, 2, \ldots, m-1\,.$$

Um (1) zu zeigen, stellen wir zunächst fest, dass jeder Gegenstand x, der in einer der Kisten B^*_i liegt höchstens eine Höhe von $\frac{1}{2}$ hat, denn andernfalls wäre nach Eigenschaft 1 sein Gewicht 1 und somit hätte B^*_i ein Gewicht nicht kleiner als 1.

Daraus folgt $c_i < \frac{1}{2}$ für $i = 1, \ldots, m-1$, denn andernfalls hätte ein Gegenstand aus B^*_m, der auch höchstens die Höhe $\frac{1}{2}$ hat, in die Kiste B^*_i gelegt werden können.

Daraus können wir schließen, dass jede Kiste B^*_i mindestens 2 Gegenstände enthält, denn es gilt $c_i < \frac{1}{2}$, und jeder Gegenstand in B^*_i hat höchstens die Höhe $\frac{1}{2}$.

Sei nun B^*_i irgendeine Kiste mit $1 \leq i \leq m-1$, und nehmen wir an, dass sie mit Objekten der Höhe $x_1 \geq x_2 \geq \ldots \geq x_k$ gefüllt ist. Wir wissen bereits, dass $k \geq 2$ gilt, und somit folgt aus Eigenschaft 2:

$$\begin{aligned} 1 - c_i = \sum_{i=1}^{k} x_i \quad &\leq 1 - x_k - \tfrac{5}{9} s_i \\ \Longrightarrow \quad c_i \quad &\geq \tfrac{5}{9} s_i + x_k\,. \end{aligned}$$

Da $x_k > c_{i-1}$ (andernfalls wäre x_k in der Kiste B^*_{i-1}), gilt (1). Nun gilt aufgrund von (1)

$$\sum_{i=1}^{m-1} s_i \leq \tfrac{9}{5} \sum_{i=1}^{m-1} (c_i - c_{i-1}) = \tfrac{9}{5} \overbrace{c_{m-1}}^{<\frac{1}{2}} < \tfrac{9}{10}.$$

Es folgt $\sum_{i=1}^{m} s_i \leq \frac{9}{5} + \overbrace{s_m}^{\leq 1} < 2$. Die letzte Gleichung impliziert

$$\begin{aligned}
\sum_{j=1}^{n} w(t'_j) &= \sum_{i=1}^{m} \sum_{j \in B_i^*} w(t'_j) + \sum_{j \notin B_i^*} w(t'_j) \\
&\geq \sum_{i=1}^{m} (1 - s_i) + m' \\
&= m + m' - \sum_{i=1}^{m} s_i \\
&> m_{FF} - 2\,.
\end{aligned}$$

Andererseits impliziert Eigenschaft 3

$$\sum_{j=1}^{n} w(t'_j) = \sum_{i=1}^{m_{\text{opt}}} \sum_{j \in B_i^{\text{opt}}} w(t'_j) \leq m_{\text{opt}} \tfrac{17}{10}\,.$$

Es gilt $\sum_{j \in B_i^{\text{opt}}} t_j \leq 1 \implies \sum_{j \in B_i^{\text{opt}}} w(t'_j) \leq \frac{17}{10}$. Insgesamt gilt also $m_{FF} \leq \sum_{j=1}^{n} w(t'_j) + 2 \leq 2 + \frac{17}{10} m_{\text{opt}}$. □

Um den Beweis von Satz (11.16) zu vervollständigen, müssen wir noch die Existenz der Funktion mit den angegebenen Eigenschaften nachweisen. Wir geben diese Funktion an:

$$w(x) := \begin{cases} \frac{6}{5}x & \text{falls } 0 \leq x < \frac{1}{6} \\ \frac{9}{5}x - \frac{1}{10} & \text{falls } \frac{1}{6} \leq x < \frac{1}{3} \\ \frac{6}{5}x + \frac{1}{10} & \text{falls } \frac{1}{3} \leq x \leq \frac{1}{2} \\ 1 & \text{falls } \frac{1}{2} < x \leq 1 \end{cases}$$

Graph von $w(x)$

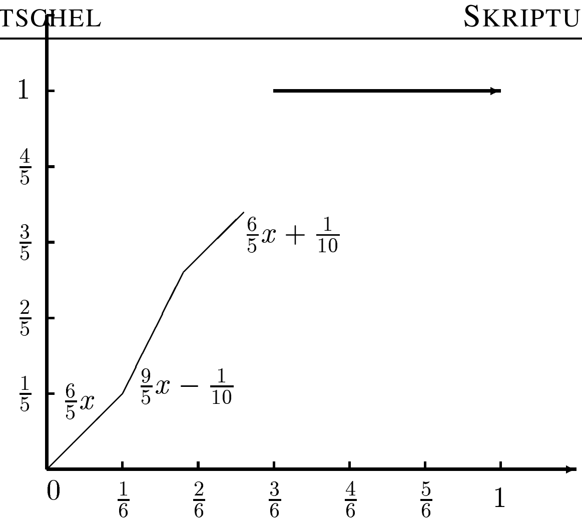


Abb. 11.9

w hat offenbar die Eigenschaft 1. Der Beweis der übrigen Eigenschaften ist recht kompliziert, wir spalten die Kette der Argumente in mehrere Behauptungen auf.

$$\begin{array}{lll}
\textbf{Behauptung1}: & \left(\frac{1}{3} \leq x \leq \frac{1}{2}\right) \implies w\left(\frac{1}{3}\right) + w\left(x - \frac{1}{3}\right) = w(x), & \textit{trivial} \\
\textbf{Behauptung2}: & \left(0 \leq x, y \leq \frac{1}{6}\right) \implies w(x) + w(y) \leq w(x+y), & \textit{trivial} \\
\textbf{Behauptung3}: & \left(0 \leq x \leq \frac{1}{2}\right) \implies \frac{6}{5}x \leq w(x) \leq \frac{3}{2}x, & \textit{trivial} \\
\textbf{Behauptung4}: & \left(x \leq x^* \leq \frac{1}{2}\right) \implies w^* - w(x) \leq \frac{9}{5}(x^* - x), & \textit{trivial.} \\
\textbf{Behauptung5}: & \left(\begin{array}{c} k \geq 2 \\ x_1 \geq x_2 \geq \ldots \geq x_k > 0 \\ 1 - x_k \leq \sum_{i=1}^k x_i \leq 1 \end{array}\right) \implies \sum_{i=1}^k w(x_j) \geq 1\,. &
\end{array}$$

Beweis : Wir können folgendes annehmen:

$x_1 \leq \frac{1}{2}$, denn sonst wäre $w(x_1) = 1$.

$x_2 < \frac{1}{3}$, denn sonst wäre $w(x_1) + w(x_2) \geq 2(w(\frac{1}{3})) = 1$.

$x_k > \frac{1}{6}$, denn sonst wäre $\sum_{i=1}^k x_i \geq \frac{5}{6}$, und aus Behauptung 3 würde folgen

$$\sum w(x_i) \geq \sum \tfrac{6}{5} x_i = \frac{6}{5} \sum x_i \geq 1\,.$$

Wir betrachten zunächst den Fall $k = 2$. $x_1 + x_2 \geq 1 - x_2 \implies x_1 \geq 1 - 2x_2 \implies x_1 \geq \frac{1}{3}, x_2 \geq \frac{1-x_1}{2}$. Und somit

$$w(x_1) + w(x_2) = \underbrace{\tfrac{6}{5}x_1 + \tfrac{1}{10}}_{w(x_1)} + \underbrace{\tfrac{9}{5}x_2 - \tfrac{1}{10}}_{w(x_2)} \geq \tfrac{6}{5}x_1 + \tfrac{9}{10}(1 - x_1) \geq 1\,.$$

$k \geq 3$.

Falls $x_1 \geq \frac{1}{3}$, dann gilt

$$\sum_{i=1}^{k} w(x_i) \geq \overbrace{\tfrac{6}{5}x_1 + \tfrac{1}{10}}^{w(x_1)} + \overbrace{\tfrac{9}{5}x_2 - \tfrac{1}{10}}^{w(x_2)} + \overbrace{\tfrac{6}{5}\sum_{i=3}^{k} x_i}^{\text{Abschätzung nach Beh. 3}} = \tfrac{6}{5}\sum_{i=1}^{k} x_i + \tfrac{3}{5}x_2$$
$$\geq \tfrac{6}{5}(1 - x_k) + \tfrac{3}{5}x_k = \tfrac{6}{5} - \tfrac{3}{5}\underbrace{x_k}_{<\frac{1}{3}} \geq 1\,.$$

Falls $x_1 < \frac{1}{3}$, dann gilt

$$\sum_{i=1}^{k} w(x_i) \geq \overbrace{\tfrac{9}{5}(x_1 + x_2) - \tfrac{1}{5}}^{w(x_1)+w(x_2)} + \overbrace{\tfrac{6}{5}(x_3 + \ldots + x_k)}^{\text{Absch. nach Beh. 3}}$$
$$= \tfrac{6}{5}\sum_{i=1}^{k} x_i + \tfrac{3}{5}(x_1 + x_2) - \tfrac{1}{5} \geq \tfrac{6}{5}(1 - x_k) + \tfrac{6}{5}x_k - \tfrac{1}{5} = 1.$$

$\square$

Behauptung 6:

$$\text{(Eigenschaft 2)} \quad \left.\begin{array}{c} k \geq 2 \\ x_1 \geq x_2 \geq \ldots \geq x_k > 0 \\ \sum_{i=1}^{k} x_i \leq 1 \\ \sum_{i=1}^{k} w(x_i) = 1 - s,\ s > 0 \end{array}\right\} \Longrightarrow \sum_{i=1}^{k} x_i \leq 1 - x_k - \tfrac{5}{9}s\,.$$

Beweis : Wir verneinen Behauptung 5. Ist also $\sum w(x_i) < 1$, so muss eine der vier Voraussetzungen von Behauptung 5 nicht gelten. Da wir in Behauptung 6 drei der vier Voraussetzungen fordern, muss $1 - x_k \leq \sum x_i$ verletzt sein. Wir definieren $t := 1 - x_k - \sum_{i=1}^{k} x_i$, und nach obiger Überlegung gilt $t > 0$. Folglich gilt $x_1 + x_2 + t \leq 1 - x_k < 1$ und deshalb gibt es Zahlen x_1^*, x_2^*, so dass

$$x_1 \leq x_1^* \leq \tfrac{1}{2}, \quad x_2 \leq x_2^* \leq \tfrac{1}{2}, \quad x_1^* + x_2^* = x_1 + x_2 + t.$$

Setzen wir $x_i^* = x_i$, $i = 3, \ldots, k$, so sind die Voraussetzungen von Behauptung 5 erfüllt, und wir erhalten $\sum_{i=1}^{k} w(x_i^*) \geq 1$. Also gilt

$$w(x_1^*) + w(x_2^*) \geq w(x_1) + w(x_2) + s.$$

Behauptung 4 impliziert nun

$$\begin{aligned} s &\le w(x_1^*) - w(x_1) + w(x_2^*) - w(x_2) \\ &\le \tfrac{9}{5}(x_1^* - x_1) + \tfrac{9}{5}(x_2^* - x_2) = \tfrac{9}{5}t\,. \end{aligned}$$

Also folgt $t \ge \frac{5}{9}s$, und damit gilt

$$\sum x_i = 1 - x_k - t \le 1 - x_k - \tfrac{5}{9}s\,.$$

□

Behauptung 7:

$$\sum_{i=1}^{k} x_i \le \tfrac{1}{2} \implies \sum_{i=1}^{k} w(x_i) \le \tfrac{7}{10}\,.$$

Beweis : Aufgrund von Behauptung 1 können wir annehmen, dass $x_i \le \frac{1}{3}$ gilt für $i = 1, \dots, k$ (andernfalls könnten wir jedes $x_i > \frac{1}{3}$ ersetzen durch $x' = \frac{1}{3}$ und $x_i'' = x_i - \frac{1}{3}$).

Aufgrund von Behauptung 2 können wir annehmen, dass höchstens ein x_i kleiner gleich $\frac{1}{6}$ ist (andernfalls könnten wir $x_i, x_j \le \frac{1}{6}$ durch $x_{ij} := x_i + x_j$ ersetzen). Wir unterscheiden vier Fälle:

i) $k = 1$ und $x_1 \le \frac{1}{3}$, dann gilt: $w(x_1) \le w(\frac{1}{3}) = \frac{1}{2} < \frac{7}{10}$.

ii) $k = 2$, $\frac{1}{6} \le x_2 \le x_1 \le \frac{1}{3}$, dann gilt: $w(x_1) + w(x_2) = \frac{9}{5}(x_1 + x_2) - \frac{2}{10} \le \frac{7}{10}$.

iii) $k = 2$, $x_2 \le \frac{1}{6} \le x_1 \le \frac{1}{3}$, dann gilt: $w(x_1) + w(x_2) \le w\frac{1}{3} + w\frac{1}{6} = \frac{7}{10}$.

iv) $k = 3$, $x_3 \le \frac{1}{6} \le x_2 \le x_1 \le \frac{1}{3}$, dann gilt: $w(x_1) + w(x_2) + w(x_3) = \frac{9}{5}(x_1 + x_2) - \frac{2}{10} + \frac{6}{5}x_3 \le \frac{9}{5}(x_2 + x_2 + x_3) - \frac{2}{10} \le \frac{7}{10}$.

□

Behauptung 8: (Eigenschaft 3)

$$\sum_{i=1}^{k} x_i \le 1 \quad \implies \sum_{i=1}^{k} w(x_i) \le \tfrac{17}{10}\,.$$

Beweis : Falls $x_i \le \frac{1}{2}$ für alle i, dann impliziert Behauptung 3

$$\sum w(x_i) \le \tfrac{3}{2} \sum x_i \le \tfrac{3}{2} < \tfrac{17}{10}\,.$$

Ist eines der x_i größer als $\frac{1}{2}$, sagen wir $x_1 > \frac{1}{2}$, dann folgt aus Behauptung 7 ($\sum_{i=2}^{k} x_i \leq \frac{1}{2}$)

$$\sum_{i=2}^{k} w(x_i) \leq \tfrac{7}{10}\,,$$

und somit $w(x_1) + \sum_{i=2}^{k} w(x_i) \leq \frac{17}{10}$. □

Damit ist unsere Analyse der FIRST FIT Heuristik beendet. Eine mögliche Verbesserung der Methode liegt auf der Hand.

(11.17) FIRST FIT DECREASING (FFD) *Gegeben sei ein Bin-Packing-Problem durch* $d, t_1, \ldots, t_n$.

1. *Ordne zunächst alle* t_i*, so dass gilt* $t_1 \geq t_2 \geq \ldots \geq t_n$.
2. *Wende FF an.*

□

Wir betrachten das folgende Beispiel.

(11.18) Beispiel. Sei

$d = 60,$ 31 (6mal), 17 (6mal), 16 (6mal), 13 (12mal),

so benötigt FFD 11 Kisten, während in der optimalen Lösung nur 9 Kisten benutzt werden.

$$\begin{array}{llcll} 6\times & 31,17 & & 6\times & 31,16,13, \\ 2\times & 16,16,16 & & \multicolumn{2}{l}{\underbrace{3\times \quad 17,17,13,13}_{\text{Optimum}}} \\ & \multicolumn{3}{c}{\underbrace{3\times \quad 13,13,13,13}_{\text{FFD}}} & \end{array}$$

□

Ist m_{FFD} die Kistenzahl der durch FFD gefundenen Lösungen und m_{opt} die optimale Kistenzahl, so kann gezeigt werden:

(11.19) Satz.

$$m_{FFD} < 4 + \tfrac{11}{9} m_{\text{opt}}\,.$$

□

Der Beweis ist in seiner Struktur ähnlich wie der von Satz (11.16). Es treten jedoch erheblich kompliziertere technische Detailprobleme auf. Der erste Beweis für Satz (11.19) benötigte rund 70 Seiten; ein kürzerer ist zu finden in: Baker (1985): "A new proof for the First-Fit Decreasing bin-packing algorithm", Journal of Algorithms 6, pp. 47–70.

Zum Abschluss möchten wir noch bemerken, dass FFD (wie auch einige andere Heuristiken) eine paradoxe Eigenschaft hat. Nämlich, die Entfernung eines Gegenstandes kann die Kistenzahl erhöhen.

(11.20) Beispiel. Gegeben sei ein Bin Packing Problem durch:

$d = 396,$ 285, 188 (6mal), 126 (18mal), 115 (3mal), 112 (3mal), 75, 60, 51,
12 (3mal), 10 (6mal), 9(12mal).

Es werden bei der FFD Lösung 12 Kisten benötigt. Entfernen wir den Gegenstand mit Höhe 75, so gilt $m_{FFD} = 13$.

□

Zwischen FF und FFD besteht (für praktische Zwecke) ein fundamentaler Unterschied. Um FFD anwenden zu können, müssen vor Beginn der Festlegung des Belegplans alle Zahlen t_i bekannt sein. Das ist bei FF nicht der Fall, hier wird jede gerade verfügbare Zahl verarbeitet, und für den Algorithmus ist es uninteressant, welche und wieviele Zahlen t_i später noch erscheinen.

Das Bin Packing Problem ist sicherlich nicht das anwendungsreichste kombinatorische Optimierungsproblem, aber es ist die beliebteste "Spielwiese" der Heuristik-Designer. Die hier angegebenen Heuristiken FF und FFD sind nicht die besten bezüglich (des beweisbaren) Worst-Case-Verhaltens. Es gibt polynomiale Verfahren, die für jedes feste ε eine Lösung garantieren, die nicht schlechter als das $(1 + \varepsilon)$-fache des Wertes der Optimallösung ist. Diese Verfahren sind allerdings recht kompliziert und basieren auf der Ellipsoidmethode. Die Literatur zum Bin-Packing-Thema ist außerordentlich umfangreich. Der Aufsatz

E. G. Coffman, M. R. Garey & D. S. Johnson (1983) *Approximation algorithms for bin-packing: an updated survey* Preprint, Bell Laboratories, Murray Hill, 1983

beschreibt den gegenwärtigen Stand der Forschung und gibt einen Überblick über die bisherige Entwicklung.

Literaturverzeichnis

Baker, B. S. (1985). A new proof for the First-Fit Decreasing bin-packing algorithm. *Journal of Algorithms*, 6:47–70.

Kapitel 12

Das Rucksackproblem

Die in Kapitel 10 bereitgestellte Maschinerie von Begriffen wollen wir anhand eines kombinatorischen Optimierungsproblems ausloten, um zu zeigen, mit welchen "Tricks" man gewisse Gütegarantien erreichen kann, bzw. mit welchen Beweismethoden die Existenz polynomialer Algorithmen mit bestimmten Gütegarantien ausgeschlossen werden kann.

Dieses Kapitel dient auch zur Vorbereitung auf die Vorlesung ADM II, die sich mit linearer und ganzzahliger Optimierung beschäftigt. 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 haben wir in Kapitel 1 (siehe (1.10) und (1.11)) bereits zwei Varianten kennengelernt. Hier formulieren wir einige weitere.

(12.1) Definition. *Seien $a_i, c_j \in \mathbb{Z}, j = 1, 2, \dots, n$ und $b \in \mathbb{N}$.*

(a) Das Problem

(KP)

$$\begin{aligned} \max \sum_{j=1}^{n} c_j x_j & \\ \sum_{j=1}^{n} a_j x_j &\leq b \\ x_j &\in \mathbb{Z}_+ \end{aligned}$$

heißt **(allgemeines) Knapsack-Problem** *oder* **Rucksackproblem**.

(b) *Fordern wir zusätzlich, dass* $x_j \in \{0,1\}$, $j = 1, \ldots, n$ *gilt, so heißt (KP)* **binäres Knapsack-Problem** *oder* **0/1-Knapsack-Problem**.

(c) *Das Problem*

$$\text{(GKP)} \qquad \begin{aligned} \max \sum_{j=1}^{n} c_j x_j & \\ \sum_{j=1}^{n} a_j x_j = b & \\ x_j \in \mathbb{Z}_+ & \end{aligned}$$

heißt **Gleichungs-Knapsack-Problem**. *Mit der Zusatzforderung* $x_j \in \{0,1\}$, $j = 1, \ldots, n$, *heißt dieses Problem* **binäres** *oder* **0/1 Gleichungs-Knapsack-Problem**.

(d) *Ein binäres Knapsack-Problem, das die folgende Bedingung erfüllt*

$$\frac{c_1}{a_1} = \frac{c_j}{a_j}, \quad j = 2, \ldots, n,$$

heißt **wertunabhängiges Knapsack-Problem**.

(e) *Ein wertunabhängiges Knapsack-Problem mit*

$$c_j = a_j, \quad j = 1, \ldots, n,$$

heißt **Subset-Sum-Problem**. *Dieses kann wie folgt dargestellt werden:*

$$\begin{aligned} \max \sum_{j=1}^{n} a_j x_j & \\ \sum_{j=1}^{n} a_j x_j & \leq b \\ x_j & \in \{0,1\}, \ j = 1, \ldots, n. \end{aligned}$$

(f) **Zulässigkeitstest:** *Gibt es einen Vektor* $x \in \{0,1\}^n$ *mit*

$$a^T x = b?$$

(Gibt es ein $x \in \mathbb{Z}_+^n$ *mit* $a^T x = b$*?)*

(g) *Sei* A *eine* (m,n)*-Matrix mit* $a_{ij} \in \mathbb{Z}_+$ *und* $b \in \mathbb{N}^m$, $c \in \mathbb{N}^n$, *dann heißt*

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

$$x \quad \in \mathbb{Z}_+^n,$$

mehrdimensionales Knapsack-Problem. *Falls* $x \in \{0,1\}^n$ *gefordert wird, so heißt dieses Problem* **mehrdimensionales 0/1-Knapsack-Problem**.

(h) **Multiple-Choice Knapsack-Problem***:*
Gegeben $n, m \in \mathbb{N}$, $b \in \mathbb{N}$, $a_{ij}, c_{ij} \in \mathbb{Z}_+$, $1 \leq i \leq m$, $1 \leq j \leq n$. *Gesucht ist ein ganzzahliger oder* $0/1$*-Vektor der Länge* $m \cdot n$, $x = (x_{11}, \ldots, x_{mn})$ *mit*

(h₁) $\sum_{i=1}^m \sum_{j=1}^n a_{ij} x_{ij} \leq b$.

(h₂) *Für jedes* $i = 1, \ldots, m$ *gibt es höchstens ein* $j \in \{1, \ldots, n\}$ *mit* $x_{ij} > 0$.

(h₃) $\sum_{i=1}^m \sum_{j=1}^n c_{ij} x_{ij}$ *ist maximal.*

□

Wir wollen nun einfache Heuristiken für verschiedene Typen von Knapsack-Problemen darstellen und ihr Verhalten analysieren. Zunächst wollen wir zeigen, dass weder für das allgemeine noch für das 0/1-Knapsack-Problem eine Differenzgarantie, siehe (10.6) (c_4), gegeben werden kann.

(12.2) Satz. *Es gibt einen polynomialen Algorithmus A für das allgemeine oder binäre Knapsack-Problem, dessen Wert höchstens um eine Konstante vom Optimalwert differiert, genau dann, wenn $\mathcal{P} = \mathcal{NP}$ gilt.*

Beweis : "$\Longrightarrow$" Gegeben sei ein Knapsackproblem (KP)

$$\begin{aligned} \max \sum_{j=1}^n c_j x_j & \\ \sum_{j=1}^n a_j x_j & \leq b \\ x_j & \in \mathbb{Z}_+ \text{ oder } x \in \{0,1\}^n \end{aligned}$$

Wir bezeichnen den Optimalwert von (KP) mit c^*. Sei A ein polynomialer Algorithmus, der eine Lösung von (KP) mit Wert c_A liefert. Wir nehmen an, dass A eine Differenzgarantie für alle Knapsackprobleme gibt, d. h. $\exists K \in \mathbb{Z}_+$ mit

$$c^* - c_A \leq K \qquad \text{für alle Knapsackprobleme}$$

Wir zeigen nun, dass unter dieser Annahme ein polynomialer Algorithmus für (KP) existiert.

Aus (KP) konstruieren wir ein neues Knapsackproblem ($(K+1)$KP), indem wir $\overline{c} = (K+1)c$ als neue Zielfunktion wählen. Offenbar ist eine Lösung x für (KP) optimal genau dann, wenn x für ($(K+1)$KP) optimal ist.

Wenden wir den Algorithmus A auf $((K+1)\text{KP})$ an, so erhalten wir eine Lösung mit Wert $\overline{c}_A$, so dass gilt

$$\overline{c}^* - \overline{c}_A \leq K.$$

Bezüglich (KP) gilt offenbar $c^* = \dfrac{\overline{c}^*}{K+1}$, und die Lösung von A hat bezüglich (KP) den Wert $c_A = \dfrac{\overline{c}_A}{K+1}$. Daraus folgt

$$c^* - c_A = \frac{1}{K+1}(\overline{c}^* - \overline{c}_A) \leq \frac{K}{K+1} < 1.$$

Da sowohl c^* als auch c_A ganzzahlig sind, muss $c^* = c_A$ gelten. Folglich ist A ein Algorithmus, der in polynomialer Zeit eine optimale Lösung von (KP) liefert. Da (KP) $\mathcal{NP}$-vollständig ist, folgt hieraus $\mathcal{P} = \mathcal{NP}$.

"$\Longleftarrow$", trivial, da dann (KP) polynomial lösbar ist. □

Die Technik des hier gegebenen Beweises ist sehr allgemein und kann für viele andere Probleme benutzt werden, um zu zeigen, dass für sie keine polynomialen Algorithmen mit Differenzgarantie gefunden werden können, siehe (10.6) (c_4). Wir müssen uns nur überlegen, dass die Multiplikation der Zielfunktion des Problems mit einer beliebigen Konstanten die optimale Lösung nicht ändert. Daraus folgt, dass für kein kombinatorisches Optimierungsproblem mit linearer Zielfunktion eine Differenzgarantie gegeben werden kann. Dies wollen wir als wichtige Nebenbemerkung formulieren.

(12.3) Satz. *Ist Π ein kombinatorisches Optimierungsproblem mit linearer Zielfunktion, d. h. jedes Problembeispiel in Π ist von der folgenden Form*

> *Gegeben eine endliche Menge $E, \mathcal{I} \subseteq \mathcal{P}(E)$ und $c : E \to \mathbb{Z}$,*
>
> *gesucht $I \in \mathcal{I}$ mit $c(I) := \sum_{e \in I} c_e$ maximal (minimal),*

dann gibt es einen polynomialen Approximationsalgorithmus für Π mit Differenzgarantie genau dann, wenn $\mathcal{P} = \mathcal{NP}$ gilt.

□

Wir untersuchen nun zwei Versionen des Greedy-Algorithmus für das Knapsack-Problem.

(12.4) Zwei Greedy-Algorithmen für das allgemeine bzw.

binäre Knapsack-Problem

Input: $c_j, a_j \in \mathbb{N}$, $j = 1, \dots, n$ *und* $b \in \mathbb{N}$.

Output: *Eine zulässige (approximative) Lösung für*

((KP) bzw. (0/1-KP))
$$\max \sum_{j=1}^{n} c_j x_j$$
$$\sum_{j=1}^{n} a_j x_j \leq b$$
$$x_j \in \mathbb{Z}_+ \text{ oder } \quad x_j \in \{0,1\}, j = 1, \ldots, n.$$

1. **Zielfunktionsgreedy:**

   Ordne die Indizes, so dass $c_1 \geq c_2 \geq \ldots \geq c_n$ *gilt.*

1'. **Gewichtsdichtengreedy:**

   Berechne die Gewichtsdichten $\rho_j := \frac{c_j}{a_j}, j = 1, 2, \ldots, n$, *und ordne die Indizes so dass* $\rho_1 \geq \rho_2 \geq \ldots \geq \rho_n$ *gilt.*

2. *DO j=1 TO n:*

   (allgemeiner Knapsack)

   Setze $x_j := \lfloor \frac{b}{a_j} \rfloor$ *und* $b := b - \lfloor \frac{b}{a_j} \rfloor$.

   (0/1-Knapsack)

   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 von der in (12.4) beschriebenen Algorithmen ist offensichtlich $O(n \log(n))$, wobei das Sortieren die Hauptarbeit erfordert.

(12.5) Bemerkung Der Zielfunktionsgreedy kann für das allgemeine und auch für das binäre Knapsack-Problem beliebig schlechte Lösungen liefern, d. h. $R_{\text{greedy}} = \infty$.

Beweis : (a) allgemeines Knapsack-Problem. Wir betrachten das folgende Beispiel mit $\alpha \geq 2$:

$$\begin{array}{rrrl} \max & \alpha x_1 + & (\alpha - 1) x_2 & \\ & \alpha x_1 + & x_2 & \leq \alpha \\ & & x_1, x_2 & \in \mathbb{Z}_+ \end{array}$$

Offenbar gilt $c_{\text{greedy}} = \alpha$ und $c_{\text{opt}} = \alpha(\alpha - 1)$. Daraus folgt

$$\frac{c_{\text{opt}} - c_{\text{greedy}}}{c_{\text{opt}}} = \frac{\alpha(\alpha-2)}{\alpha(\alpha-1)} = \frac{\alpha-2}{\alpha-1} \to 1.$$

Mithin gilt $R_{\text{greedy}} = \infty$.

(b) 0/1-Knapsack-Problem. Wir betrachten

$$\begin{array}{rrcl} \max & nx_1 + (n-1)x_2 + \ldots + (n-1)x_n & & \\ & nx_1 + \quad x_2 + \ldots + \quad x_n & \leq & n \\ & x_j & \in & \{0,1\}. \end{array}$$

Trivialerweise gilt $c_{\text{greedy}} = n$ und $c_{\text{opt}} = n(n-1)$, also

$$\frac{c_{\text{opt}} - c_{\text{greedy}}}{c_{\text{opt}}} = \frac{n(n-1)-n}{n(n-1)} = \frac{n-2}{n-1} \to 1.$$

$\square$

Wir wollen uns hier noch einmal an Kapitel 5 erinnern. Für $a \in \mathbb{N}^n, b \in \mathbb{N}$ sei $E = \{1,2,\ldots,n\}, L = \{x \in \{0,1\}^n \mid a^T x \leq b\}$ und

$$\mathcal{I} := \{I \subseteq E \mid \exists x \in L \text{ mit } \chi^I = x\},$$

dann ist $\mathcal{I}$ eine mengentheoretische Darstellung aller Lösungen L des durch a und b gegebenen 0/1-Knapsack-Problems.

Das Mengensystem $\mathcal{I} \subseteq 2^E$ ist offenbar ein Unabhängigkeitssystem. Bemerkung (12.5) zeigt, dass der Zielfunktionsgreedy bezüglich $\max\{c(I) \mid I \in \mathcal{I}\}$ beliebig schlecht werden kann. Ist dies nicht ein Widerspruch zu Satz (5.14), in dem eine universelle Schranke für die Güte des Greedyalgorithmus angegeben wurde? Natürlich nicht! (12.5) zeigt lediglich, dass der Rangquotient q für das Unabhängigkeitssystem $\mathcal{I}$ mit wachsendem n beliebig klein werden kann.

(12.6) Satz. *Der Gewichtsdichten-Greedyalgorithmus ist für das allgemeine Knapsack-Problem ein $\frac{1}{2}$-approximativer Algorithmus, und es gilt $R_{\text{Ggreedy}} = \frac{1}{2}$.*

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

$$c_{\text{Ggreedy}} \geq c_1 x_1 = c_1 \lfloor \frac{b}{a_1} \rfloor,$$

und ebenso

$$c_{\text{opt}} \leq c_1 \frac{b}{a_1} \leq c_1 (\lfloor \frac{b}{a_1} \rfloor + 1) \leq 2c_1 \lfloor \frac{b}{a_1} \rfloor \leq 2c_{\text{Ggreedy}}$$

und somit

$$c_{\text{Ggreedy}} \geq \frac{1}{2} c_{\text{opt}}.$$

Wir zeigen nun, dass diese Schranke tatsächlich asymptotisch angenommen wird. Dazu betrachten wir das folgende Knapsack-Problem:

$$\begin{array}{rrcl} \max & 2\alpha x_1 + 2(\alpha-1)x_2 & & \\ & \alpha x_1 + \ (\alpha-1)x_2 & \leq & 2(\alpha-1) \\ & x_1, x_2 & \in & \mathbb{Z}_+ \end{array}$$

Offensichtlich gilt $\rho_1 \geq \rho_2$, $c_{\text{Ggreedy}} = 2\alpha$, $c_{\text{opt}} = 4(\alpha-1)$ und somit

$$\frac{c_{\text{opt}} - c_{\text{Ggreedy}}}{c_{\text{opt}}} = \frac{2\alpha-4}{4\alpha-4} \to \frac{1}{2}.$$

□

Um lästige Trivialbemerkungen zu vermeiden, nehmen wir im weiteren an, dass die Indizes so geordnet sind, dass $\rho_1 \geq \rho_2 \geq \ldots \geq \rho_n$ gilt und dass $a_j \leq b$ gilt für $j = 1, \ldots, n$. Leider gilt die schöne Schranke aus (12.6) nicht für das 0/1-Knapsack-Problem.

(12.7) Bemerkung. *Gegeben sei ein 0/1-Knapsack-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-Knapsack-Problems und $k \in \{0, 1, \ldots, n-1\}$. Dann gilt

$$\begin{aligned} c_{\text{opt}} = \sum_{j=1}^{n} c_j x_j^* \leq \sum_{j=1}^{k} c_j x_j^* + \sum_{j=k+1}^{n} \frac{a_j c_{k+1}}{a_{k+1}} x_j^* \quad &= \frac{c_{k+1}}{a_{k+1}} \sum_{j=1}^{n} a_j x_j^* + \sum_{j=1}^{k} \Big(c_j - \frac{c_{k+1}}{a_{k+1}} a_j\Big) x_j^* \\ &\leq \frac{c_{k+1}}{a_{k+1}} b + \sum_{j=1}^{k} \Big(c_j - \frac{c_{k+1}}{a_{k+1}} a_j\Big) \\ &= \sum_{j=1}^{k} c_j + \frac{c_{k+1}}{a_{k+1}} \Big(b - \sum_{j=1}^{k} a_j\Big). \end{aligned}$$

(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. $\square$

(12.8) Bemerkung.

(a) Der Gewichtsdichten-Greedyalgorithmus kann im Falle des 0/1-Knapsack-Problems beliebig schlechte Lösungen liefern, d. h. $R_{\text{Ggreedy}} = \infty$.

(b) Führen wir sowohl den Gewichtsdichten- als auch den Zielfunktions-Greedyalgorithmus für ein $0/1$-Knapsack-Problem aus, so ist dieses kombinierte Verfahren ein $\frac{1}{2}$-approximativer Algorithmus.

Beweis : (a) Wir betrachten das folgende Beispiel mit $\rho_1 \geq \rho_2$:

$$\begin{array}{rrcl} \max & x_1 + \alpha x_2 & & \\ & x_1 + \alpha x_2 & \leq & \alpha \\ & x_1, x_2 & \in & \{0,1\}. \end{array}$$

Es gilt $c_{\text{opt}} = \alpha$ und $c_{\text{Ggreedy}} = 1$, also

$$\frac{c_{\text{opt}} - c_{\text{Ggreedy}}}{c_{\text{opt}}} = \frac{\alpha - 1}{\alpha} \to 1.$$

Beweis : (b) Gilt nach Ausführung des Gewichtsdichten-Greedyalgorithmus $c_{\text{Ggreedy}} \geq \frac{1}{2} c_{\text{opt}}$, so sind wir fertig. Andernfalls sei

$$c_k = max\{c_j | \quad j = 1, \ldots, n\}$$

und mit (12.7) (a) 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. $\square$

Für praktische Anwendungen sind die in der nachfolgenden Bemerkung angegebenen Schranken sicherlich auch recht interessant.

Bevor wir uns dem Entwurf eines FPAS für das 0/1-Knapsack-Problem zuwenden, wollen wir zunächst ein PAS für das Subset-Sum-Problem darstellen, das besonders einfach ist und das eine der wesentlichen Ideen zur Konstruktion solcher Algorithmen deutlich macht, nämlich die Kombination von teilweiser Enumeration und dem Greedy-Algorithmus.

(12.9) Ein PAS für das Subset-Sum-Problem.

Input: $a_j \in \mathbb{Z}_+, j = 1, \dots, n$, $b \in \mathbb{Z}_+$ *und* $0 < \varepsilon = \frac{p}{q} < 1$ *(gewünschter Approximationsgrad).*

Output: *Zulässige Lösung* $\overline{x}$ *von*

(SSP)
$$\begin{aligned} &\max \sum_{j=1}^{n} a_j x_j \\ &\sum_{j=1}^{n} a_j x_j \le b \\ &x_j \in \{0,1\},\ j = 1, \dots, n. \end{aligned}$$

so dass $a^T \overline{x} \ge (1-\varepsilon) a^T x^*$ *ist, wobei* x^* *eine Optimallösung ist.*

1. *Setze* $K := \left\lceil \frac{q}{p} \right\rceil = \left\lceil \frac{1}{\varepsilon} \right\rceil$.
2. *Teile die Indexmenge* $N = \{1, \dots, n\}$ *in "große"* (L) *und "kleine"* (S) *Indizes auf:*
$$\begin{aligned} L :=&\ \ \{j \in N \mid a_j \ge \tfrac{b}{K}\}, \\ S' := S :=&\ \ \{j \in N \mid a_j < \tfrac{b}{K}\}. \end{aligned}$$
3. *Löse das Subset-Sum-Problem*

(SSP')
$$\begin{aligned} &\max \sum_{j \in L} a_j x_j \\ &\sum_{j \in L} a_j x_j \le b \\ &x_j \in \{0,1\},\ j \in L \end{aligned}$$

optimal (z. B. durch Enumeration aller Lösungen). Sei $\overline{x}_j, j \in L$, *eine optimale Lösung von (SSP'), und sei*
$$\begin{aligned} \overline{L} := L' &:= \{j \in L \mid \overline{x}_j = 1\} \\ b' &:= \sum_{j \in L'} a_j \overline{x}_j. \end{aligned}$$

4. *Falls für alle $j \in S'$ gilt $b' + a_j > b$, so kann kein $x_j, j \in S$, zusätzlich auf 1 gesetzt werden. Setze $\overline{x}_j := 0, j \in S'$, und STOP.*

5. *Andernfalls finde ein $j \in S'$, so dass $b - (b' + a_j) \geq 0$ und diese Differenz minimal ist. Setze*

$$\begin{aligned} L' &:= L' \cup \{j\} \\ S' &:= S' \setminus \{j\} \\ b' &:= b' + a_j \end{aligned}$$

und gehe zu 4.

□

Algorithmus (12.9) besteht also aus einem Enumerationsverfahren, das für die wichtigsten Indizes eine Optimallösung liefert, und einem Greedy-Teil (Schritte 4 und 5), der die restlichen Variablen festlegt.

(12.10) Satz. *Das in (12.9) beschriebene Verfahren ist ein PAS für das Subset-Sum-Problem. Löst man Schritt 3 durch vollständige Enumeration aller höchstens K-elementigen Teilmengen von L, so ist die Laufzeit dieses Algorithmus $O(n^K)$.*

Beweis : Gilt am Ende des Verfahrens $S \subseteq L'$, d. h. gilt in der gefundenen Lösung $\overline{x}$, $\overline{x}_j = 1$ für alle $j \in S$, so ist $\overline{x}$ optimal für (SSP). Denn angenommen, es gäbe eine bessere Lösung x' für (SSP), so gilt aufgrund der Optimalität in Schritt 3 für (SSP′) $\sum_{j \in L} a_j \overline{x}_j \geq \sum_{j \in L} a_j x^*$ und somit $a^T x^* = \sum_{j \in L} a_j x_j^* + \sum_{j \in S} a_j x_j^* \leq \sum_{j \in L} a_j \overline{x}_j + \sum_{j \in S} a_j \overline{x}_j$.

Nehmen wir an, dass mindestens für ein $j_0 \in S, \overline{x}_{j_0} = 0$ gilt und dass b^* den Optimalwert bezeichnet, so erhalten wir

$$\sum_{j \in N} a_j \overline{x}_j = \sum_{j \in L'} a_j = b' > b - a_{j_0} > b - \frac{b}{K} = b\big(1 - \frac{1}{K}\big) \geq b^* \big(1 - \frac{1}{K}\big)$$

(wobei $b' > b - a_{j_0}$ aufgrund von Schritt 5 gilt) und somit $b' > (1 - \varepsilon) b^*$.

Für Schritt 2 benötigen wir $O(n)$ Schritte, für Schritt 3 $O(\binom{n}{K} \cdot K)$ Schritte (wenn wir alle höchstens K-elementigen Teilmengen von N enumerieren) und für die Schritte 4 und 5 nochmals $O(n)$ Schritte. Die Gesamtlaufzeit ist dann $O(\binom{n}{K} \cdot K)$, also $O(n^K)$ bei **festem** K (denn $\binom{n}{K} = \frac{n!}{K!(n-K)!} = \frac{1}{K!} n(n-1) \dots (n-k+1) = O(n^K)$). □

Nachdem wir bisher den Greedy-Algorithmus für das Knapsack-Problem und ein PAS für das Subset-Sum-Problem kennengelernt haben, wollen wir nun ein FPAS für das 0/1-Knapsack-Problem entwickeln.

Gegeben seien $c_i, a_i \in \mathbb{N}, i = 1, \ldots, n$ und $b \in \mathbb{N}$. Wir betrachten

$$\begin{aligned} \max \sum_{j=1}^{n} c_j x_j & \\ \sum_{j=1}^{n} a_j x_j \leq b & \\ & x_j \in \{0,1\},\ j = 1, \ldots, n, \end{aligned}$$

wobei wir wie immer annehmen, dass $\rho_1 = \frac{c_1}{a_1} \geq \rho_2 \geq \ldots \geq \rho_n = \frac{c_n}{a_n}$ und $a_i \leq b$, $i = 1, \ldots, n$. Zunächst zeigen wir, dass der Wert der Optimallösung einfach abgeschätzt werden kann.

(12.11) Lemma. *Sei k der größte Index, so dass $a_1 + \ldots + a_k \leq b$. Setze $c_{\text{est}} := c_1 + \ldots + c_k + c_{k+1}$. Dann gilt*

$$c_{\text{opt}} \leq c_{\text{est}} \leq 2c_{\text{opt}}.$$

Beweis : Da $\overline{x}_i = 1, i = 1, \ldots, k, \overline{x}_i = 0$, sonst, eine zulässige Lösung ist, gilt $c_1 + \ldots + c_k \leq c_{\text{opt}}$ und offenbar $c_{k+1} \leq c_{\text{opt}}$, also $c_{\text{est}} \leq 2c_{\text{opt}}$. Nach Bemerkung (12.7) (a) gilt

$$c_{\text{opt}} \leq \sum_{j=1}^{k} c_j + \frac{c_{k+1}}{a_{k+1}} \underbrace{\Big(b - \sum_{j=1}^{k} a_j\Big)}_{\leq a_{k+1}} < \sum_{j=1}^{k+1} c_j = c_{\text{est}}.$$

$\square$

Die zweite Grundlage des FPAS, neben einer geeigneten Aufteilung der Variablen und einer geschickten Parameterwahl, ist die Lösung eines speziellen Gleichheits-Knapsack-Problems mit Hilfe der dynamischen Programmierung. Dieses (SGKP) hat die folgende Form

$$(SGKP_d) \qquad \begin{aligned} \min \sum_{j=1}^{m} a_j x_j & \\ \sum_{j=1}^{m} w_j x_j = d & \\ x_j \in \{0,1\},\ j = 1, \ldots, m, & \end{aligned}$$

wobei $a_j, w_j \in \mathbb{Z}_+, j = 1, \ldots, m$ gilt und d eine ganze Zahl zwischen 0 und einer vorgegebenen Zahl N ist.

Wir sind nicht nur an einer Lösung für ein spezielles d interessiert, sondern an Optimallösungen von (SGKP$_d$) für $d = 0, \ldots, N$. Der nachfolgende Algorithmus beschreibt, wie die Lösungen von (SGKP$_d$) rekursiv konstruiert werden können. Sei

$$(SGKP)_{r,d}) \qquad \begin{aligned} \min \ \textstyle\sum_{j=1}^r a_j x_j \\ \textstyle\sum_{j=1}^r w_j x_j = d \\ x_j \in \{0,1\}, \ j = 1, \ldots, r, \end{aligned}$$

$1 \leq r \leq m, 0 \leq d \leq N$ und $f(r,d)$ der Optimalwert von (SGKP$_{r,d}$).

(12.12) Lemma. *Für die Funktion $f(r,d), 1 \leq r \leq m, 0 \leq d \leq N$ gilt*

$$\begin{aligned}
(1) \quad f(r,0) &= 0, \quad r = 1, \ldots, m \\
(2) \quad f(1,d) &= \begin{cases} 0 \textit{falls} & d = 0 \\ a_1 \textit{falls} & d = w_1 \\ \infty \textit{andernfalls} \end{cases} \\
(3) \quad f(r,d) &= \min \begin{cases} f(r-1,d) \\ \\ f(r-1,d-w_r) + a_r \end{cases} \qquad r \geq 2, d \geq 1
\end{aligned}$$

(falls $d - w_r < 0$, dann $f(r-1, d-w_r) = \infty$).

Beweis : Die ersten beiden Beziehungen sind trivial. Eine Optimallösung $x_1^*, \ldots, x_r^*$ von (SGKP$_{r,d}$) erfüllt entweder $x_r^* = 0$ oder $x_r^* = 1$.
Ist $x_r^* = 0$, so ist $x_1^*, \ldots, x_{r-1}^*$ eine Optimallösung von (SGKP$_{r-1,d}$), also $f(r,d) = f(r-1,d)$.
Ist $x_r^* = 1$, so ist $x_1^*, \ldots, x_{r-1}^*$ eine Optimallösung von (SGKP$_{r-1,d-1}$), also $f(r,d) = f(r-1,d-w_r) + a_r$. □

Die Funktion $f(r,d)$ kann man sehr einfach in Tabellenform speichern.

(12.13) Lemma.

(a) *Eine Optimallösung von (SGKP$_{r,d}$) kann wie folgt aus der Funktionstabelle von f konstruiert werden.*

DO $j = r$ TO 2 BY -1

Falls $f(j,d) = \infty$, dann gibt es keine Lösung. STOP.

Falls $f(j,d) < \infty$, dann führe aus

Falls $f(j,d) = f(j-1,d)$, setze $x_j^d := 0$.

Falls $f(j,d) = f(j-1,d-w_j) + a_j$, setze $x_j^d := 1$ und

$d := d - w_j$.

END.

Falls $d = 0$, setze $x_1^d = 0$, andernfalls $x_1^d = 1$.

(b) *Die Berechnung von f kann so gestaltet werden, daß insgesamt nur jeweils zwei Spalten der Funktionstabelle gespeichert werden. Die Optimallösung kann auch hierbei konstruiert werden, wenn jeweils nach Berechnung einer Spalte r die Optimallösungen von (SGKP$_{r,d}$) gespeichert werden.*

(c) *Der Aufwand zur Berechnung von $f(m,N)$ beträgt $O(mN)$ Rechenschritte. Der Algorithmus ist polynomial!*

Beweis : Klar. □

Wir führen ein Beispiel zur Berechnung von $f(r,d)$ vor.

(12.14) Beispiel. Wir betrachten das folgende 0/1-Knapsack-Problem

$$\begin{array}{rll} \min & 31x_1 + 22x_2 + 50x_3 & \\ & 3x_1 + \ \ 2x_2 + \ \ 5x_3 & = d \\ & x_1, x_2, x_3 & \in \{0,1\} \end{array}$$

$N = 11$

				Output der Optimallösungen		
d,r	1	2	3	1	2	3
0	0	0	0	0	0	0
1	∞	∞	∞	0	0	0
2	∞	22	22	–	1	0
3	31	31	31	1	0	0
4	∞	∞	∞	–	–	–
5	∞	53	50	–	1	1
6	∞	∞	∞	–	–	–
7	∞	∞	72	–	–	1
8	∞	∞	81	–	–	1
9	∞	∞	∞	–	–	–
10	∞	∞	103	–	–	1
11	∞	∞	∞	–	–	–

Die letzte Spalte (für $r = 3$) enthält den bezüglich des jeweiligen d optimalen Lösungswert. □

Das FPAS für das 0/1-Knapsack-Problem basiert nun darauf, dass eine Folge von (SGKP)'s gelöst wird, deren Anzahl polynomial im Input des 0/1-Knapsack-Problems ist und deren Lösungsaufwand ebenfalls polynomial im Input ist.

Wir beschreiben nun das FPAS von Ibarra und Kim, wobei wir zunächst noch die geeignete Wahl der Parameter offen lassen.

(12.15) FPAS von Ibarra und Kim für das 0/1-Knapsack-Problem.

Input: $a_j, c_j \in \mathbb{Z}_+, j = 1, \ldots, n$, $b \in \mathbb{Z}_+$ *und zwei Parameter* s, t.

Output: *Approximative Lösung von*

$$\text{(KP)} \qquad \begin{aligned} &\max \sum_{j=1}^{n} c_j x_j \\ &\sum_{j=1}^{n} a_j x_j \le b \\ &x_j \in \{0,1\},\ j = 1, \ldots, n, \end{aligned}$$

(Der Parameter s ist ein Skalierungsparameter, der Parameter t wird benutzt, um die Variablen in zwei Klassen zu zerlegen.)

1. *Abschätzung für den Optimalwert:*

   Sei k der größte Index, so dass $\sum_{j=1}^{k} a_j \le b$. Setze

   $$c_{\text{est}} = \sum_{j=1}^{k+1} c_j$$

   (siehe Lemma (12.11)).

2. *Zerlegung der Indexmenge:*

   $$\begin{aligned} L &:= \{j \in \{1, \ldots, n\} \mid c_j \ge t\} \qquad &\text{(large indices)} \\ S &:= \{j \in \{1, \ldots, n\} \mid c_j < t\} \qquad &\text{(small indices).} \end{aligned}$$

3. *Löse die speziellen Gleichheits-Knapsack-Probleme*

$$(SGKP_d) \qquad \begin{aligned} &\min \sum_{j \in L} a_j x_j \\ &\sum_{j \in L} \lfloor \frac{c_j}{s} \rfloor x_j = d \\ &x_j \in \{0,1\},\ j \in L, \end{aligned}$$

für $d = 0, 1, \ldots, \lfloor \frac{c_{\text{est}}}{s} \rfloor$. (Hierdurch wird die wahre Optimallösung approximativ erreicht.)

4. *Für $d = 0, 1, \ldots, \lfloor \frac{c_{\text{est}}}{s} \rfloor$ führe aus:*

   (a) *Ist $x_j^d, j \in L$ die Optimallösung von (SGKP$_d$) und gilt $\sum_{j \in L} a_j x_j^d \leq b$, dann wende den Gewichtsdichten-Greedyalgorithmus (12.4) auf das folgende Knapsack-Problem an.*

$$(SKP_d) \qquad \begin{aligned} &\max \sum_{j \in S} c_j x_j \\ &\sum_{j \in S} a_j x_j \leq b - \sum_{j \in L} a_j x_j^d \ (=: b_d) \\ &x_j \in \{0,1\},\ j \in S, \end{aligned}$$

   (b) *Sei $x_j^d, j \in S$ die Greedy-Lösung von (SKP$_d$), dann ist $x_j^d, j \in \{1, \ldots, n\}$ eine Lösung von (KP).*

5. *Wähle die beste der $\lfloor \frac{c_{\text{est}}}{s} \rfloor + 1$ gefundenen Lösungen von (KP).*

$\square$

Idee des Algorithmus:

– Man nehme die größten Zielfunktionswerte ($c_j, j \in L$).

– Versuche in Schritt 3 ungefähr den optimalen Zielfunktionswert zu erreichen, wobei durch Abrunden Fehler in Kauf genommen werden.

– Unter allen möglichen Approximationen wähle man in Schritt 3 jene, die am wenigsten vom Rucksack verbraucht.

– Am Schluss fülle man den Rucksack mit dem verbleibenden “Kleinkram” auf (Greedy).

(12.16) Beispiel.

$$\begin{array}{lrl} \max & 930x_1 + 858x_2 + 846x_3 + 674x_4 + 962x_5 + 860x_6 & \\ & 92x_1 + \;\; 85x_2 + \;\; 84x_3 + \;\; 67x_4 + \;\; 96x_5 + \;\; 86x_6 & \leq 250 \\ & x_i & \in \{0,1\}. \end{array}$$

Wir setzen $s = 250$ und $t = 860$.

1. $c_{\text{est}} = 930 + 858 + 846 = 2634$
2. $L := \{j \mid c_j \geq 860\} = \{1,5,6\}$

   $S := \{j \mid c_j < 860\} = \{2,3,4\}$
3. Für $d = 0,1,\ldots,\lfloor\frac{c_{\text{est}}}{s}\rfloor = 10$ ist zu lösen

$$\min \sum_{j\in L} a_j x_j = 92x_1 + 96x_5 + 86x_6$$

(SGKP$_d$) $$juhu \sum_{j\in L} \lfloor\frac{c_j}{s}\rfloor x_j = 3x_1 + 3x_5 + 3x_6$$

$$x_1, x_5, x_6 \in \{0,1\}$$

Wir erhalten hierfür

	x_1	x_1, x_5	x_1, x_5, x_6
d, r	1	2	3
0	0	0	0
1	∞	∞	∞
2	∞	∞	∞
3	92	92	86
4	∞	∞	∞
5	∞	∞	∞
6	∞	188	178
7	∞	∞	∞
8	∞	∞	∞
9	∞	∞	274
10	∞	∞	∞

Optimallösungen existieren für:

$$\begin{array}{llllll} d=0, & x_1^0 = 0, & x_5^0 = 0, & x_6^0 = 0 & \text{Wert} & 0 \\ d=3, & x_1^3 = 0, & x_5^3 = 0, & x_6^3 = 1 & \text{Wert} & 860 \end{array}$$

$$\begin{array}{lllll} d=6, & x_1^6=1, & x_5^6=0, & x_6^6=1 & \text{Wert } 1790 \\ d=9, & x_1^9=0, & x_5^9=0, & x_6^9=1 & \text{Wert } 2752 \end{array}$$

4. Die Knapsack-Probleme haben die folgende Form:

$$\text{(SKP}_d\text{)} \qquad \begin{aligned} \max \sum_{j\in S} c_j x_j &= 858x_2 + 846x_3 + 674x_4 \\ \sum_{j\in S} a_j x_j &= 85x_2 + 84x_3 + 67x_4 \le b_d \\ & x_2, x_3, x_4 \in \{0,1\}, \end{aligned}$$

   wobei $b_0 = 250$, $b_3 = 164$ und $b_6 = 72$. (b_9 kommt nicht in Betracht, da $\sum_{j\in L} a_j x_j^9 = 274 > b = 250$. Wir haben also die drei Knapsackprobleme (SKP$_0$), (SKP$_3$) und (SKP$_6$) mit dem Gewichtsdichten-Greedy-Algorithmus zu lösen.

   Es ergeben sich die folgenden Lösungen

$$\begin{array}{lllll} (\text{SKP}_0) & x_2^0=1, & x_3^0=1, & x_4^0=1 & \text{Wert } 2378 \\ (\text{SKP}_3) & x_2^3=1, & x_3^3=0, & x_4^3=1 & \text{Wert } 1532 \\ (\text{SKP}_6) & x_2^6=0, & x_3^6=0, & x_4^6=1 & \text{Wert } 674 \end{array}$$

5. Wir erhalten drei Lösungen von (KP) mit den Werten 2378, 2392 und 2464. Also hat die Lösung von (SGKP$_6$) zusammen mit der Lösung von (SKP$_6$) die beste Lösung

$$x_1 = 1, \quad x_2 = 0, \quad x_3 = 0, \quad x_4 = 1, \quad x_5 = 0, \quad x_6 = 1$$

   ergeben mit Wert 2464.

□

(12.17) Satz. *Sei c_{IK} der durch Algorithmus (12.15) gefundene Lösungswert, dann gilt*

$$c_{IK} \ge c_{\text{opt}} - (\frac{s}{t}\, c_{\text{opt}} + t).$$

Beweis : Sei $x_1^*, \ldots, x_n^*$ eine optimale Lösung von (KP). Setze

$$d := \sum_{j\in L} \lfloor \frac{c_j}{s} \rfloor x_j^*,$$

dann gilt

$$d \le \Big\lfloor \frac{1}{s} \sum_{j\in L} c_j x_j^* \Big\rfloor \le \lfloor \frac{1}{s}\, c_{\text{opt}} \rfloor \le \lfloor \frac{1}{s}\, c_{\text{est}} \rfloor .$$

Also ist im Algorithmus (12.15) eine Kandidatenlösung $\overline{x}_1, \dots, \overline{x}_n$ mit

$$\sum_{j\in L} \lfloor \frac{c_j}{s} \rfloor \overline{x}_j = d$$

betrachtet worden, da in diesem Fall die Lösungsmenge nicht leer ist. Daraus folgt

$$c_{IK} \geq \sum_{j=1}^{n} c_j \overline{x}_j = c_{\text{opt}} - \Big(\sum_{j\in L} c_j x_j^* - \sum_{j\in L} c_j \overline{x}_j + \sum_{j\in S} c_j x_j^* - \sum_{j\in S} c_j \overline{x}_j \Big).$$

Wir erhalten

$$\begin{aligned}
\sum_{j\in L} c_j x_j^* - \sum_{j\in L} c_j \overline{x}_j \quad &\leq s \overbrace{\sum_{j\in L} \lfloor \frac{c_j}{s} \rfloor x_j^*}^{=d} + \textstyle\sum_{j\in L} \Big(c_j - s \lfloor \frac{c_j}{s} \rfloor \Big) x_j^* - s \overbrace{\sum_{j\in L} \lfloor \frac{c_j}{s} \rfloor \overline{x}_j}^{=d} \\
&= \textstyle\sum_{j\in L} \underbrace{(c_j - s \lfloor \frac{c_j}{s} \rfloor)}_{\leq s} x_j^* \\
&\leq s \textstyle\sum_{j\in L} x_j^* \leq \frac{s}{t} \sum_{j\in L} c_j x_j^* \leq \frac{s}{t}\, c_{\text{opt}}.
\end{aligned}$$

Wenden wir Bemerkung (12.8) (b) auf (SKP$_d$) an, so erhalten wir

$$\begin{aligned}
c_{\text{Ggreedy}}^{(\text{SKP}_d)} \quad &= \sum_{j\in S} c_j \overline{x}_j > c_{\text{opt}}^{(\text{SKP}_d)} - \max\{c_j \mid j \in S\} \geq c_{\text{opt}}^{(\text{SKP}_d)} - t \\
&\Rightarrow t > c_{\text{opt}}^{(\text{SKP}_d)} - c_{\text{Ggreedy}}^{(\text{SKP}_d)} \geq \sum_{j\in S} c_j x_j^* - \sum_{j\in S} c_j \overline{x}_j,
\end{aligned}$$

also

$$c_{IK} \geq c_{\text{opt}} - (\frac{s}{t} c_{\text{opt}} + t).$$

□

(12.18) Satz *Sei $\varepsilon > 0$, setze $s := (\frac{\varepsilon}{3})^2 c_{\text{est}}$ und $t = \frac{\varepsilon}{3} c_{\text{est}}$ und wende den Algorithmus (12.15) an. Dann gilt*

(a) Die Laufzeit von (12.15) ist $O(n \log n) + O(n(\frac{1}{\varepsilon})^2)$ (d. h. (12.15) ist ein FPAS).

(b) Für den durch (12.15) gefundenen Lösungswert c_{IK} gilt

$$c_{IK} \geq (1 - \varepsilon) c_{\text{opt}}.$$

Beweis : (a) Für die Anwendung des Gewichtsdichten-Greedyalgorithmus müssen wir die ρ_j sortieren, was $O(n \log n)$ Zeit benötigt.

Die Berechnung von s und t erfordert $O(\log \varepsilon + n)$ Operationen. Die Lösung der Probleme (SGKP$_d$) in Schritt 3 benötigt $O(n \lfloor \frac{c_{\text{est}}}{s} \rfloor) = O(\frac{n}{\varepsilon^2})$ Operationen, (aufgrund der Rekursivität braucht nicht jedes (SGKP$_d$) neu berechnet zu werden) (Lemma (12.13) (c)). Also wird Schritt 3 in $O(\frac{n}{\varepsilon^2})$ Schritten erledigt. In Schritt 4 wird $O(\frac{1}{\varepsilon^2})$ mal der Greedyalgorithmus mit jeweils $O(n)$ Operationen angewendet, also ist der Aufwand in Schritt 4 ebenfalls $O(\frac{n}{\varepsilon^2})$, wodurch wir die Gesamtlaufzeit erhalten.

(b) Aufgrund von Lemma (12.11) gilt $t \leq \frac{2\varepsilon}{3} c_{\text{opt}}$. Also folgt aus Satz (12.17)

$$c_{IK} \geq c_{\text{opt}} - \left(\frac{\varepsilon}{3} c_{\text{opt}} + \frac{2\varepsilon}{3} c_{\text{opt}}\right) = (1 - \varepsilon)\, c_{\text{opt}}.$$

$\square$

Wir werden uns nun noch überlegen, wie man den Algorithmus von Ibarra und Kim so umformen kann, dass damit auch das (allgemeine) Knapsack-Problem gelöst werden kann. Wir betrachten also

$$\begin{array}{lll} \max \sum_{j=1}^{n} c_j x_j & & \\ \sum_{j=1}^{n} a_j x_j & \leq b & \\ \qquad x_j & \in \mathbb{N}, j = 1, \dots, n, & \end{array}$$

wobei alle Zahlen $c_j, a_j, b \in \mathbb{N}$ sind. Wir nehmen wie üblich an, dass gilt

$$\begin{array}{ccll} a_j & \leq & b & j = 1, \dots, n \\ \frac{c_1}{a_1} & \geq & \frac{c_2}{a_2} \geq \dots \frac{c_n}{a_n}. & \end{array}$$

Den Gewichtsdichten-Greedyalgorithmus haben wir in (12.4) dargestellt und in Satz (12.6) haben wir gezeigt, dass er ein $\frac{1}{2}$-approximativer Algorithmus ist.

Um Ibarra und Kim anwenden zu können, benötigen wir eine Schätzung c_{est} und Parameter s, t zur Definition von S und L.

1. Setze $c_{\text{est}} = c_1 \lfloor \frac{b}{a_1} \rfloor$.

   (dann gilt $c_{\text{est}} \leq c_{\text{Ggreedy}} \leq c_{\text{opt}} \leq c_1 + c_{\text{est}} \leq 2c_{\text{est}}$).

2. Definiere L und S wie in (12.15), wobei s und t gegeben sind.

3. Die Gleichheits-Knapsack-Probleme sind die gleichen wie in (12.15), es wird lediglich die Bedingung $x_j \in \{0,1\}$ durch $x_j \in \mathbb{N}$ ersetzt. Wir lösen diese ebenfalls für $d = 0, 1, \ldots, \lfloor \frac{c_{\text{est}}}{s} \rfloor$.

Zur Lösung dieser Probleme verwenden wir die folgende Rekursion

$$
\begin{aligned}
f(r,0) &= \qquad 0, \quad \text{für alle } d \\
f(1,d) &= \begin{cases} \frac{a_1 d}{w_1} & \text{falls } w_1 \text{ ein Teiler von } d \\ \infty & \text{sonst} \end{cases} \\
f(r,d) &= \quad \min \begin{cases} f(r-1,d) \\ f(r,d-w_r)+a_r \end{cases}
\end{aligned}
$$

Der Rest verläuft genauso wie bei (12.15). Insbesondere gilt Satz (12.18) wörtlich ebenfalls für das allgemeine Knapsack-Problem wie für das 0/1-Knapsack-Problem.

Vollpolynomiale Approximationsschemata sind auch für das mehrdimensionale Knapsack-Problem (12.1) (g) und das Multiple-Choice-Knapsack-Problem (12.1) (h) gefunden worden. FPAS sind ansonsten nur noch für einige Scheduling-Probleme bekannt. Man kann zeigen, dass für die meisten kombinatorischen Optimierungsprobleme keine FPAS und keine PAS existieren.