



Algorithmische Graphentheorie

von
Volker Turau

2., überarbeitete Auflage

Oldenbourg Verlag München Wien

Volker Turau ist seit 2002 Professor im Bereich Telematik an der Technischen Universität Hamburg-Harburg. Von 1977 bis 1983 studierte und promovierte er an der Johannes Gutenberg Universität in Mainz. Anschließend hatte er eine Postdoktorantenstelle an der Universität Manchester in England inne und war als wissenschaftlicher Mitarbeiter an der Universität Karlsruhe tätig. Im Oktober 1987 wechselte er in das Institut für Publikations- und Informationssysteme (IPSI) der Gesellschaft für Mathematik und Datenverarbeitung in Darmstadt, wo er bis 1992 im Bereich objektorientierter und multimedialer Datenbanken arbeitete. Zudem hatte Volker Turau zwei längere Forschungsaufenthalte am New Jersey Institute of Technology und am Hewlett-Packard Labor in Palo Alto, Kalifornien. Von 1992 bis 2002 war er Professor für Informatik an den Fachhochschulen Gießen-Friedberg und Wiesbaden für das Fachgebiet Web-basierte Systeme und hatte drei längere Forschungsaufenthalte am International Computer Science Institute Berkely. Volker Turau ist Mitglied in mehreren Programm-Komitees (u.a. ACM Symposium on Applied Computing, W3C Konferenz) sowie von ACM, IEEE und der Gesellschaft für Informatik GI. Seine Forschungsinteressen umfassen verteilte Informationssysteme, Sensornetzwerke, E-Learning-Umgebungen sowie die Integration unternehmensweiter Anwendungen.

Bibliografische Information Der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <<http://dnb.ddb.de>> abrufbar.

© 2004 Oldenbourg Wissenschaftsverlag GmbH
Rosenheimer Straße 145, D-81671 München
Telefon: (089) 45051-0
www.oldenbourg-verlag.de

Das Werk einschließlich aller Abbildungen ist urheberrechtlich geschützt. Jede Verwertung außerhalb der Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlages unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Bearbeitung in elektronischen Systemen.

Lektorat: Kathrin Mönch
Herstellung: Rainer Hartl
Umschlagkonzeption: Kraxenberger Kommunikationshaus, München
Gedruckt auf säure- und chlорfreiem Papier
Druck: R. Oldenbourg Graphische Betriebe Druckerei GmbH

ISBN 3-486-20038-0

Vorwort

Die erfreuliche Nachfrage nach dem Buch war Anlaß, eine in vielen Teilen erweiterte und überarbeitete Neuauflage zu verfassen. In fast allen Kapiteln kamen neue Beispiele hinzu oder es wurden vorhandene Beispiele erweitert bzw. verbessert. Viele dieser Änderungen gehen auf Anregungen von Lesern zurück, denen an dieser Stelle herzlich gedankt sei. Auf vielfachen Wunsch sind im Anhang für alle Aufgaben Lösungen angegeben. Mein besonderer Dank gilt folgenden Lesern: Elke Hocke, Thomas Emden-Weinert, Holger Dörnemann, Philipp Flach, Bernhard Schiefer, Ralf Kastner, Andreas Görz und Felix Yu. Dem Oldenbourg Verlag möchte ich für die angenehme Zusammenarbeit danken.

Hamburg

Volker Turau

Vorwort zur 1. Auflage

Graphen sind die in der Informatik am häufigsten verwendete Abstraktion. Jedes System, welches aus diskreten Zuständen oder Objekten und Beziehungen zwischen diesen besteht, kann als Graph modelliert werden. Viele Anwendungen erfordern effiziente Algorithmen zur Verarbeitung von Graphen. Dieses Lehrbuch ist eine Einführung in die algorithmische Graphentheorie. Die Algorithmen sind in kompakter Form in einer programmiersprachennahen Notation dargestellt. Eine Übertragung in eine konkrete Programmiersprache wie C++ oder Pascal ist ohne Probleme durchzuführen. Die meisten der behandelten Algorithmen sind in der dargestellten Form im Rahmen meiner Lehrveranstaltungen implementiert und getestet worden. Die praktische Relevanz der vorgestellten Algorithmen wird in vielen Anwendungen aus Gebieten wie Compilerbau, Betriebssysteme, künstliche Intelligenz, Computernetzwerke und Operations Research demonstriert.

Dieses Buch ist an alle jene gerichtet, die sich mit Problemen der algorithmischen Graphentheorie beschäftigen. Es richtet sich insbesondere an Studenten der Informatik und Mathematik im Grund- als auch im Hauptstudium.

Die neun Kapitel decken die wichtigsten Teilgebiete der algorithmischen Graphentheorie ab, ohne einen Anspruch auf Vollständigkeit zu erheben. Die Auswahl der Algorithmen

erfolgte nach den folgenden beiden Gesichtspunkten: Zum einen sind nur solche Algorithmen berücksichtigt, die sich einfach und klar darstellen lassen und ohne großen Aufwand zu implementieren sind. Der zweite Aspekt betrifft die Bedeutung für die algorithmische Graphentheorie an sich. Bevorzugt wurden solche Algorithmen, welche entweder Grundlagen für viele andere Verfahren sind oder zentrale Probleme der Graphentheorie lösen.

Unter den Algorithmen, welche diese Kriterien erfüllten, wurden die effizientesten hinsichtlich Speicherplatz und Laufzeit dargestellt. Letztlich war die Auswahl natürlich oft eine persönliche Entscheidung. Aus den genannten Gründen wurde auf die Darstellung von Algorithmen mit kompliziertem Aufbau oder auf solche, die sich auf komplexe Datenstrukturen stützen, verzichtet. Es werden nur sequentielle Algorithmen behandelt. Eine Berücksichtigung von parallelen oder verteilten Graphalgorithmen würde den Umfang dieses Lehrbuchs sprengen.

Das erste Kapitel gibt anhand von mehreren praktischen Anwendungen eine Motivation für die Notwendigkeit von effizienten Graphalgorithmen. Das zweite Kapitel führt in die Grundbegriffe der Graphentheorie ein. Als Einstieg in die algorithmische Graphentheorie wird der Algorithmus zur Bestimmung des transitiven Abschlusses eines Graphen diskutiert und analysiert. Das zweite Kapitel stellt außerdem Mittel zur Verfügung, um Zeit- und Platzbedarf von Algorithmen abzuschätzen und zu vergleichen.

Kapitel 3 beschreibt Anwendungen von Bäumen und ihre effiziente Darstellung. Dabei werden mehrere auf Bäumen basierende Algorithmen präsentiert. Kapitel 4 behandelt Suchstrategien für Graphen. Ausführlich werden Tiefen- und Breitensuche, sowie verschiedene Realisierungen dieser Techniken diskutiert. Zahlreiche Anwendungen auf gerichtete und ungerichtete Graphen zeigen die Bedeutung dieser beiden Verfahren.

Kapitel 5 diskutiert Algorithmen zur Bestimmung von minimalen Färbungen. Für allgemeine Graphen wird mit dem Backtracking-Algorithmus ein Verfahren vorgestellt, welches sich auch auf viele andere Probleme anwenden lässt. Für planare und transitiv orientierbare Graphen werden effiziente Algorithmen zur Bestimmung von minimalen Färbungen dargestellt.

Die beiden Kapitel 6 und 7 behandeln Flüsse in Netzwerken. Zunächst werden zwei Algorithmen zur Bestimmung von maximalen Flüssen vorgestellt. Der erste basiert auf Erweiterungswegen minimaler Länge, der zweite verwendet die Technik der blockierenden Flüsse. Im Mittelpunkt von Kapitel 7 stehen Anwendungen dieser Verfahren: Bestimmung von maximalen Zuordnungen in bipartiten Graphen, Bestimmung der Kanten- und Eckenzusammenhangszahl eines ungerichteten Graphen und Bestimmung von minimalen Schnitten.

Kapitel 8 betrachtet verschiedene Varianten des Problems der kürzesten Wege in kantenbewerteten Graphen. Es werden auch Algorithmen zur Bestimmung von kürzesten Wegen diskutiert, wie sie in der künstlichen Intelligenz Anwendung finden.

Kapitel 9 gibt eine Einführung in approximative Algorithmen. Unter der Voraussetzung $\mathcal{P} \neq \mathcal{NP}$ wird gezeigt, daß die meisten \mathcal{NP} -vollständigen Probleme keine approximativen Algorithmen mit beschränktem absoluten Fehler besitzen und daß sich die Probleme aus \mathcal{NPC} bezüglich der Approximierbarkeit mit beschränktem relativen Fehler sehr unterschiedlich verhalten. Breiten Raum nehmen approximative Algorithmen

für das Färbungsproblem und das Traveling-Salesman Problem ein. Schließlich werden Abschätzungen für den Wirkungsgrad dieser Algorithmen untersucht.

Ein unerfahrener Leser sollte zumindest die ersten vier Kapitel sequentiell lesen. Die restlichen Kapitel sind relativ unabhängig voneinander (mit Ausnahme von Kapitel 7, welches auf Kapitel 6 aufbaut). Das Kapitel über approximative Algorithmen enthält viele neuere Forschungsergebnisse und ist aus diesem Grund das umfangreichste. Damit wird auch der Hauptrichtung der aktuellen Forschung der algorithmischen Graphentheorie Rechnung getragen.

Jedes Kapitel hat am Ende einen Abschnitt mit Übungsaufgaben; insgesamt sind es etwa 250. Der Schwierigkeitsgrad ist dabei sehr unterschiedlich. Einige Aufgaben dienen nur zur Überprüfung des Verständnisses der Verfahren. Mit * bzw. ** gekennzeichnete Aufgaben erfordern eine intensivere Beschäftigung mit der Aufgabenstellung und sind für fortgeschrittene Studenten geeignet.

Mein Dank gilt allen, die mich bei der Erstellung dieses Buches unterstützt haben. Großen Anteil an der Umsetzung des Manuskriptes in L^AT_EX hatten Thomas Erik Schmidt und Tim Simon. Einen wertvollen Beitrag leisteten auch die Studentinnen und Studenten mit ihren kritischen Anmerkungen zu dem Stoff in meinen Vorlesungen und Seminaren. Ein besonderer Dank gilt meiner Schwester Christa Teusch für die kritische Durchsicht des Manuskriptes. Den beiden Referenten Professor Dr. A. Beutelspacher und Professor Dr. P. Widmayer danke ich für ihre wertvollen Verbesserungsvorschläge zu diesem Buch.

Wiesbaden, im Februar 1996

Volker Turau

Inhaltsverzeichnis

1	Einleitung	1
1.1	Verletzlichkeit von Kommunikationsnetzen	1
1.2	Wegplanung für Roboter.....	3
1.3	Optimale Umrüstzeiten für Fertigungszellen.....	5
1.4	Objektorientierte Programmiersprachen	6
1.5	Suchmaschinen	10
1.6	Literatur	13
1.7	Aufgaben	13
2	Einführung	17
2.1	Grundlegende Definitionen.....	17
2.2	Spezielle Graphen	21
2.3	Graphalgorithmen	24
2.4	Datenstrukturen für Graphen	24
2.4.1	Adjazenzmatrix	25
2.4.2	Adjazenzliste	26
2.4.3	Kantenliste.....	27
2.4.4	Bewertete Graphen	28
2.4.5	Implizite Darstellung	28
2.5	Darstellung der Algorithmen.....	29
2.6	Der transitive Abschluß eines Graphen	30
2.7	Vergleichskriterien für Algorithmen	35
2.8	Greedy-Algorithmen	40
2.9	Zufällige Graphen	44
2.10	Literatur	45
2.11	Aufgaben	45

3	Bäume	51
3.1	Einführung	51
3.2	Anwendungen	54
3.2.1	Hierarchische Dateisysteme	54
3.2.2	Ableitungsbäume	54
3.2.3	Suchbäume	55
3.2.4	Datenkompression	59
3.3	Datenstrukturen für Bäume	64
3.3.1	Darstellung mit Feldern	64
3.3.2	Darstellung mit Adjazenzlisten	65
3.4	Sortieren mit Bäumen	67
3.5	Vorrang-Warteschlangen	71
3.6	Minimal aufspannende Bäume	74
3.6.1	Der Algorithmus von Kruskal	75
3.6.2	Der Algorithmus von Prim	80
3.7	Literatur	82
3.8	Aufgaben	82
4	Suchverfahren in Graphen	87
4.1	Einleitung	88
4.2	Tiefensuche	88
4.3	Anwendung der Tiefensuche auf gerichtete Graphen	92
4.4	Kreisfreie Graphen und topologische Sortierung	94
4.4.1	Rekursion in Programmiersprachen	95
4.4.2	Topologische Sortierung	96
4.5	Starke Zusammenhangskomponenten	98
4.6	Transitiver Abschluß und transitive Reduktion	103
4.7	Anwendung der Tiefensuche auf ungerichtete Graphen	106
4.8	Anwendung der Tiefensuche in der Bildverarbeitung	108
4.9	Blöcke eines ungerichteten Graphen	110
4.10	Breitensuche	116
4.11	Beschränkte Tiefensuche	121
4.12	Literatur	124
4.13	Aufgaben	124

5	Färbung von Graphen	131
5.1	Einführung	131
5.2	Anwendungen von Färbungen	138
5.2.1	Maschinenbelegungen	138
5.2.2	Registerzuordnung in Compilern	139
5.2.3	Public-Key Kryptosysteme	140
5.3	Backtracking-Verfahren	141
5.4	Das Vier-Farben-Problem	145
5.5	Transitiv orientierbare Graphen	148
5.6	Literatur	156
5.7	Aufgaben	157
6	Flüsse in Netzwerken	165
6.1	Einleitung	165
6.2	Der Satz von Ford und Fulkerson	170
6.3	Bestimmung von Erweiterungswegen	172
6.4	Der Algorithmus von Dinic	181
6.5	0-1-Netzwerke	190
6.6	Kostenminimale Flüsse	194
6.7	Literatur	196
6.8	Aufgaben	197
7	Anwendungen von Netzwerkalgorithmen	203
7.1	Maximale Zuordnungen	203
7.2	Netzwerke mit oberen und unteren Kapazitäten	209
7.3	Eckenzusammenhang in ungerichteten Graphen	214
7.4	Kantenzusammenhang in ungerichteten Graphen	222
7.5	Minimale Schnitte	225
7.6	Literatur	233
7.7	Aufgaben	234
8	Kürzeste Wege	241
8.1	Einleitung	241
8.2	Das Optimalitätsprinzip	244

8.3	Der Algorithmus von Moore und Ford	248
8.4	Anwendungen auf spezielle Graphen	252
8.4.1	Graphen mit konstanter Kantenbewertung	252
8.4.2	Graphen ohne geschlossene Wege	252
8.4.3	Graphen mit nichtnegativen Kantenbewertungen	253
8.5	Routingverfahren in Kommunikationsnetzen	256
8.6	Kürzeste-Wege-Probleme in der künstlichen Intelligenz	258
8.6.1	Der iterative A^* -Algorithmus	264
8.6.2	Umkreissuche	267
8.7	Kürzeste Wege zwischen allen Paaren von Ecken	271
8.8	Der Algorithmus von Floyd	276
8.9	Steiner Bäume	278
8.10	Literatur	282
8.11	Aufgaben	282
9	Approximative Algorithmen	289
9.1	Die Komplexitätsklassen \mathcal{P} , \mathcal{NP} und \mathcal{NPC}	290
9.2	Einführung in approximative Algorithmen	295
9.3	Absolute Qualitätsgarantien	297
9.4	Relative Qualitätsgarantien	299
9.5	Approximative Färbungsalgorithmen	305
9.6	Das Problem des Handlungsreisenden	315
9.7	Literatur	325
9.8	Aufgaben	326
A	Angaben zu den Graphen an den Kapitelanfängen	337
B	Lösungen der Übungsaufgaben	341
B.1	Kapitel 1	341
B.2	Kapitel 2	342
B.3	Kapitel 3	349
B.4	Kapitel 4	357
B.5	Kapitel 5	365
B.6	Kapitel 6	372
B.7	Kapitel 7	380

B.8	Kapitel 8	389
B.9	Kapitel 9	398

Literaturverzeichnis	415
-----------------------------	------------

Index	423
--------------	------------

Inhaltsverzeichnis

1	Einleitung	1
1.1	Verletzlichkeit von Kommunikationsnetzen	1
1.2	Wegplanung für Roboter.....	3
1.3	Optimale Umrüstzeiten für Fertigungszellen.....	5
1.4	Objektorientierte Programmiersprachen	6
1.5	Suchmaschinen	10
1.6	Literatur	13
1.7	Aufgaben	13
2	Einführung	17
2.1	Grundlegende Definitionen.....	17
2.2	Spezielle Graphen	21
2.3	Graphalgorithmen	24
2.4	Datenstrukturen für Graphen	24
2.4.1	Adjazenzmatrix	25
2.4.2	Adjazenzliste	26
2.4.3	Kantenliste.....	27
2.4.4	Bewertete Graphen	28
2.4.5	Implizite Darstellung	28
2.5	Darstellung der Algorithmen.....	29
2.6	Der transitive Abschluß eines Graphen	30
2.7	Vergleichskriterien für Algorithmen	35
2.8	Greedy-Algorithmen	40
2.9	Zufällige Graphen	44
2.10	Literatur	45
2.11	Aufgaben	45

3	Bäume	51
3.1	Einführung	51
3.2	Anwendungen	54
3.2.1	Hierarchische Dateisysteme	54
3.2.2	Ableitungsbäume	54
3.2.3	Suchbäume	55
3.2.4	Datenkompression	59
3.3	Datenstrukturen für Bäume	64
3.3.1	Darstellung mit Feldern	64
3.3.2	Darstellung mit Adjazenzlisten	65
3.4	Sortieren mit Bäumen	67
3.5	Vorrang-Warteschlangen	71
3.6	Minimal aufspannende Bäume	74
3.6.1	Der Algorithmus von Kruskal	75
3.6.2	Der Algorithmus von Prim	80
3.7	Literatur	82
3.8	Aufgaben	82
4	Suchverfahren in Graphen	87
4.1	Einleitung	88
4.2	Tiefensuche	88
4.3	Anwendung der Tiefensuche auf gerichtete Graphen	92
4.4	Kreisfreie Graphen und topologische Sortierung	94
4.4.1	Rekursion in Programmiersprachen	95
4.4.2	Topologische Sortierung	96
4.5	Starke Zusammenhangskomponenten	98
4.6	Transitiver Abschluß und transitive Reduktion	103
4.7	Anwendung der Tiefensuche auf ungerichtete Graphen	106
4.8	Anwendung der Tiefensuche in der Bildverarbeitung	108
4.9	Blöcke eines ungerichteten Graphen	110
4.10	Breitensuche	116
4.11	Beschränkte Tiefensuche	121
4.12	Literatur	124
4.13	Aufgaben	124

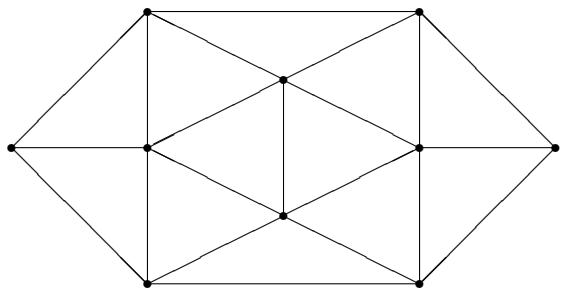
5	Färbung von Graphen	131
5.1	Einführung	131
5.2	Anwendungen von Färbungen	138
5.2.1	Maschinenbelegungen	138
5.2.2	Registerzuordnung in Compilern	139
5.2.3	Public-Key Kryptosysteme	140
5.3	Backtracking-Verfahren	141
5.4	Das Vier-Farben-Problem	145
5.5	Transitiv orientierbare Graphen	148
5.6	Literatur	156
5.7	Aufgaben	157
6	Flüsse in Netzwerken	165
6.1	Einleitung	165
6.2	Der Satz von Ford und Fulkerson	170
6.3	Bestimmung von Erweiterungswegen	172
6.4	Der Algorithmus von Dinic	181
6.5	0-1-Netzwerke	190
6.6	Kostenminimale Flüsse	194
6.7	Literatur	196
6.8	Aufgaben	197
7	Anwendungen von Netzwerkalgorithmen	203
7.1	Maximale Zuordnungen	203
7.2	Netzwerke mit oberen und unteren Kapazitäten	209
7.3	Eckenzusammenhang in ungerichteten Graphen	214
7.4	Kantenzusammenhang in ungerichteten Graphen	222
7.5	Minimale Schnitte	225
7.6	Literatur	233
7.7	Aufgaben	234
8	Kürzeste Wege	241
8.1	Einleitung	241
8.2	Das Optimalitätsprinzip	244

8.3	Der Algorithmus von Moore und Ford	248
8.4	Anwendungen auf spezielle Graphen	252
8.4.1	Graphen mit konstanter Kantenbewertung	252
8.4.2	Graphen ohne geschlossene Wege	252
8.4.3	Graphen mit nichtnegativen Kantenbewertungen	253
8.5	Routingverfahren in Kommunikationsnetzen	256
8.6	Kürzeste-Wege-Probleme in der künstlichen Intelligenz	258
8.6.1	Der iterative A^* -Algorithmus	264
8.6.2	Umkreissuche	267
8.7	Kürzeste Wege zwischen allen Paaren von Ecken	271
8.8	Der Algorithmus von Floyd	276
8.9	Steiner Bäume	278
8.10	Literatur	282
8.11	Aufgaben	282
9	Approximative Algorithmen	289
9.1	Die Komplexitätsklassen \mathcal{P} , \mathcal{NP} und \mathcal{NPC}	290
9.2	Einführung in approximative Algorithmen	295
9.3	Absolute Qualitätsgarantien	297
9.4	Relative Qualitätsgarantien	299
9.5	Approximative Färbungsalgorithmen	305
9.6	Das Problem des Handlungsreisenden	315
9.7	Literatur	325
9.8	Aufgaben	326
A	Angaben zu den Graphen an den Kapitelanfängen	337
B	Lösungen der Übungsaufgaben	341
B.1	Kapitel 1	341
B.2	Kapitel 2	342
B.3	Kapitel 3	349
B.4	Kapitel 4	357
B.5	Kapitel 5	365
B.6	Kapitel 6	372
B.7	Kapitel 7	380

B.8	Kapitel 8	389
B.9	Kapitel 9	398
Literaturverzeichnis		415
Index		423

Kapitel 1

Einleitung



In vielen praktischen und theoretischen Anwendungen treten Situationen auf, die durch ein System von Objekten und Beziehungen zwischen diesen Objekten charakterisiert werden können. Die Graphentheorie stellt zur Beschreibung von solchen Systemen ein Modell zur Verfügung: den Graphen. Die problemunabhängige Beschreibung mittels eines Graphen lässt die Gemeinsamkeit von Problemen aus den verschiedensten Anwendungsbereichen erkennen. Die Graphentheorie ermöglicht somit die Lösung vieler Aufgaben, welche aus dem Blickwinkel der Anwendung keine Gemeinsamkeiten haben. Die algorithmische Graphentheorie stellt zu diesem Zweck Verfahren zur Verfügung, die problemunabhängig formuliert werden können. Ferner erlauben Graphen eine anschauliche Darstellung, welche die Lösung von Problemen häufig erleichtert.

Im folgenden werden vier verschiedene Anwendungen diskutiert. Eine genaue Betrachtung der Aufgabenstellungen führt ganz natürlich auf eine graphische Beschreibung und damit auch auf den Begriff des Graphen; eine Definition wird bewußt erst im nächsten Kapitel vorgenommen. Die Beispiele dienen als Motivation für die Definition eines Graphen; sie sollen ferner einen Eindruck von der Vielfalt der zu lösenden Aufgaben geben und die Notwendigkeit von effizienten Algorithmen vor Augen führen.

1.1 Verletzlichkeit von Kommunikationsnetzen

Ein Kommunikationsnetz ist ein durch Datenübertragungswege realisierter Verband mehrerer Rechner. Es unterstützt den Informationsaustausch zwischen Benutzern an

verschiedenen Orten. Die *Verletzlichkeit* eines Kommunikationsnetzes ist durch die Anzahl von Leitungen oder Rechnern gekennzeichnet, die ausfallen müssen, damit die Verbindung zwischen zwei beliebigen Benutzern nicht mehr möglich ist. Häufig ist eine Verbindung zwischen zwei Benutzern über mehrere Wege möglich. Somit ist beim Ausfall einer Leitung oder einer Station die Verbindung nicht notwendigerweise unterbrochen. Ein Netzwerk, bei dem schon der Ausfall einer einzigen Leitung oder Station gewisse Verbindungen unmöglich macht ist verletzlicher, als ein solches, wo dies nur beim Ausfall von mehreren Leitungen oder Stationen möglich ist.

Die minimale Anzahl von Leitungen und Stationen, deren Ausfall die Funktion des Netzwerkes beeinträchtigt, hängt sehr stark von der Beschaffenheit des Netzwerkes ab. Netzwerke lassen sich graphisch durch Knoten und Verbindungslien zwischen den Knoten darstellen: Die Knoten entsprechen den Stationen, die Verbindungslien den Datenübertragungswegen. In Abbildung 1.1 sind vier Grundformen gebräuchlicher Netzwerke dargestellt.

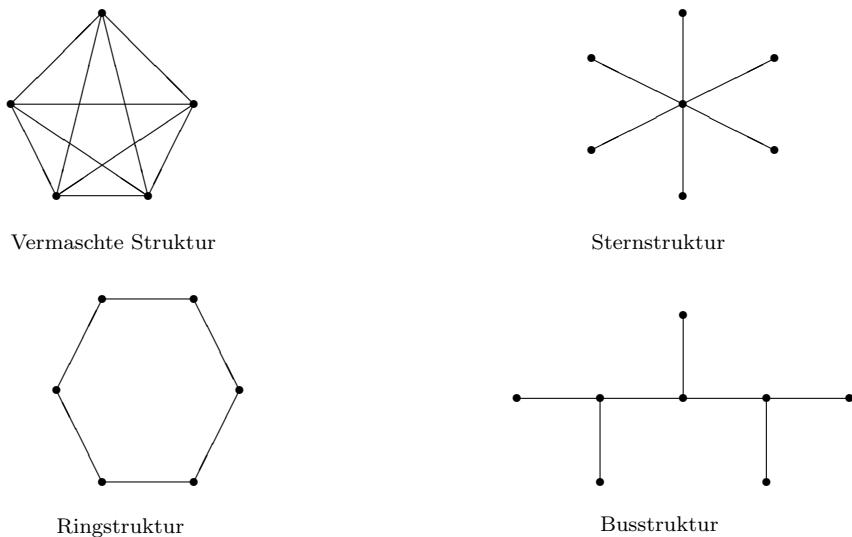


Abbildung 1.1: Netzwerktopologien

Bei der vermaschten Struktur ist beim Ausfall einer Station die Kommunikation zwischen den restlichen Stationen weiter möglich. Sogar der Ausfall von bis zu sechs Datenübertragungswegen muß noch nicht zur Unterbrechung führen. Um die Kommunikation mit einem Benutzer zu unterbrechen, müssen mindestens vier Datenübertragungswege ausfallen. Bei der Sternstruktur ist nach dem Ausfall der zentralen Station keine Kommunikation mehr möglich. Hingegen führt in diesem Fall der Ausfall einer Leitung nur zur Abkopplung eines Benutzers.

Eine Menge von Datenübertragungswegen in einem Kommunikationsnetzwerk heißt *Schnitt*, falls ihr Ausfall die Kommunikation zwischen irgendwelchen Stationen unter-

bricht. Ein Schnitt mit der Eigenschaft, daß keine echte Teilmenge ebenfalls ein Schnitt ist, nennt man *minimaler Schnitt*. Die Anzahl der Verbindungslien in dem minimalen Schnitt mit den wenigsten Verbindungslien nennt man *Verbindungs zusammenhang* oder auch die *Kohäsion* des Netzwerkes. Sie charakterisiert die Verletzlichkeit eines Netzwerkes. Die Kohäsion von Bus- und Sternstruktur ist gleich 1, die der Ringstruktur gleich 2, und die vermaschte Struktur hat die Kohäsion 4.

Analog kann man auch den Begriff *Knotenzusammenhang* eines Netzwerkes definieren. Er gibt die minimale Anzahl von Stationen an, deren Ausfall die Kommunikation der restlichen Stationen untereinander unterbrechen würde. Bei der Bus- und Sternstruktur ist diese Zahl gleich 1 und bei der Ringstruktur gleich 2. Fällt bei der vermaschten Struktur eine Station aus, so bilden die verbleibenden Stationen immer noch eine vermaschte Struktur. Somit ist bei dieser Struktur beim Ausfall von beliebig vielen Stationen die Kommunikation der restlichen Stationen gesichert.

Verfahren zur Bestimmung von Verbindungs zusammenhang und Knotenzusammenhang eines Netzwerkes werden in Kapitel 7 behandelt.

1.2 Wegplanung für Roboter

Ein grundlegendes Problem auf dem Gebiet der Robotik ist die Planung von kollisionsfreien Wegen für Roboter in ihrem Einsatzgebiet. Von besonderem Interesse sind dabei die kürzesten Wege, auf denen der Roboter mit keinem Hindernis in Kontakt kommt. Zum Auffinden dieser Wege muß eine Beschreibung der Geometrie des Roboters und des Einsatzgebietes vorliegen. Ohne Einschränkungen der Freiheitsgrade des Roboters und der Komplexität des Einsatzgebietes ist dieses Problem praktisch nicht lösbar.

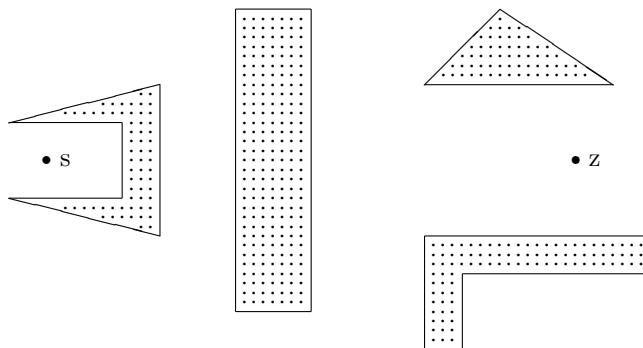


Abbildung 1.2: Hindernisse für einen Roboter

Eine stark idealisierte Version dieses komplizierten Problems erhält man, indem man die Bewegung auf eine Ebene beschränkt und den Roboter auf einen Punkt reduziert. Diese Ausgangslage kann auch in vielen Anwendungen durch eine geeignete Transformation geschaffen werden. Das Problem stellt sich nun folgendermaßen dar: Für eine Menge

von Polygonen in der Ebene, einen Startpunkt s und einen Zielpunkt z ist der kürzeste Weg von s nach z gesucht, welcher die Polygone nicht schneidet; Abbildung 1.2 zeigt eine solche Situation.

Zunächst wird der kürzeste Weg analysiert, um dann später ein Verfahren zu seiner Bestimmung zu entwickeln. Dazu stellt man sich den kürzesten Weg durch ein straff gespanntes Seil vor. Es ergibt sich sofort, daß der Weg eine Folge von Geradensegmenten der folgenden Art ist:

- a) Geradensegmente von s oder z zu einer konvexen Ecke eines Hindernisses, welche keine anderen Hindernisse schneiden
- b) Kanten von Hindernissen zwischen konvexen Ecken
- c) Geradensegmente zwischen konvexen Ecken von Hindernissen, welche keine anderen Hindernisse schneiden
- d) das Geradensegment von s nach z , sofern kein Hindernis davon geschnitten wird

Für jede Wahl von s und z ist der kürzeste Weg eine Kombination von Geradensegmenten der oben beschriebenen vier Typen. Der letzte Typ kommt nur dann vor, wenn die direkte Verbindung von s und z kein Hindernis schneidet; in diesem Fall ist dies auch der kürzeste Weg. Abbildung 1.3 zeigt alle möglichen Geradensegmente für die Situation aus Abbildung 1.2.

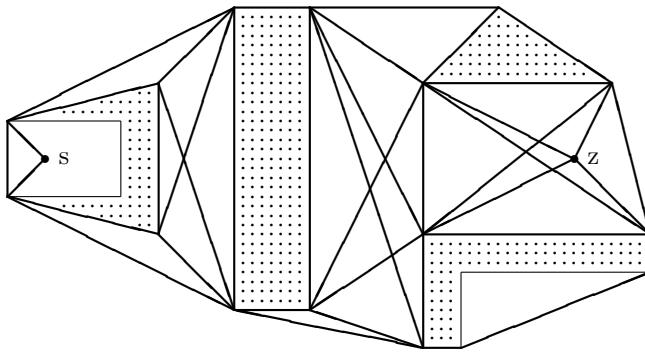


Abbildung 1.3: Geradensegmente für den kürzesten Weg

Um einen kürzesten Weg von s nach z zu finden, müssen im Prinzip alle Wege von s nach z , welche nur die angegebenen Segmente verwenden, betrachtet werden. Für jeden Weg ist dann die Länge zu bestimmen, und unter allen Wegen wird der kürzeste ausgewählt. Das Problem lässt sich also auf ein System von Geradensegmenten mit entsprechenden Längen reduzieren. In diesem ist dann ein kürzester Weg von s nach z zu finden.

Die Geradensegmente, die zu diesem System gehören, sind in Abbildung 1.3 fett gezeichnet. Abbildung 1.4 zeigt den kürzesten Weg von s nach z . Verfahren zur Bestimmung von kürzesten Wegen in Graphen werden in Kapitel 8 behandelt.

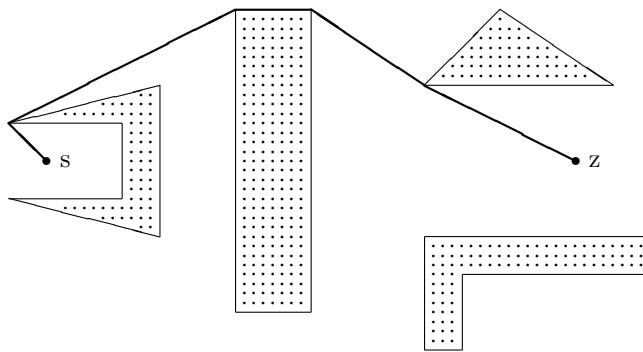


Abbildung 1.4: Der kürzeste Weg von s nach z

1.3 Optimale Umrüstzeiten für Fertigungszellen

Die Maschinen einer Fertigungszelle in einer Produktionsanlage müssen für unterschiedliche Produktionsaufträge verschieden ausgerüstet und eingestellt werden. Während dieser Umrüstzeit kann die Fertigungszelle nicht produktiv genutzt werden. Aus diesem Grund wird eine möglichst geringe Umrüstzeit angestrebt. Die Umrüstzeiten sind abhängig von dem Ist- und dem Sollzustand der Fertigungszelle. Liegen mehrere Produktionsaufträge vor, deren Reihenfolge beliebig ist, und sind die entsprechenden Umrüstzeiten bekannt, so kann nach einer Reihenfolge der Aufträge gesucht werden, deren Gesamtumrüstzeit minimal ist. Hierbei werden Aufträge, welche keine Umrüstzeiten erfordern, zu einem Auftrag zusammengefaßt. Das Problem läßt sich also folgendermaßen beschreiben: Gegeben sind n Produktionsaufträge P_1, \dots, P_n . Die Umrüstzeit der Fertigungszelle nach Produktionsablauf P_i für den Produktionsablauf P_j wird mit T_{ij} bezeichnet. Im allgemeinen ist $T_{ij} \neq T_{ji}$, d.h. die Umrüstzeiten sind nicht symmetrisch. Eine optimale Reihenfolge hat nun die Eigenschaft, daß unter allen Reihenfolgen P_{i_1}, \dots, P_{i_n} die Summe

$$\sum_{j=1}^{n-1} T_{i_j i_{j+1}}$$

minimal ist. Diese Summe stellt nämlich die Gesamtumrüstzeit dar.

Das Problem läßt sich auch graphisch interpretieren: Jeder Produktionsablauf wird durch einen Punkt in der Ebene und jeder mögliche Umrütvorgang durch einen Pfeil vom Anfangs- zum Zielzustand dargestellt. Die Pfeile werden mit den entsprechenden Umrüstzeiten markiert. Abbildung 1.5 zeigt ein Beispiel für vier Produktionsaufträge.

Eine Reihenfolge der Umrütvorgänge entspricht einer Folge von Pfeilen in den vorgegebenen Richtungen. Hierbei kommt man an jedem Punkt genau einmal vorbei. Die Gesamtumrüstzeit entspricht der Summe der Markierungen dieser Pfeile; umgekehrt entspricht jeder Weg mit der Eigenschaft, daß er an jedem Punkt genau einmal vorbeikommt, einer möglichen Reihenfolge. Bezieht man noch den Ausgangszustand in diese

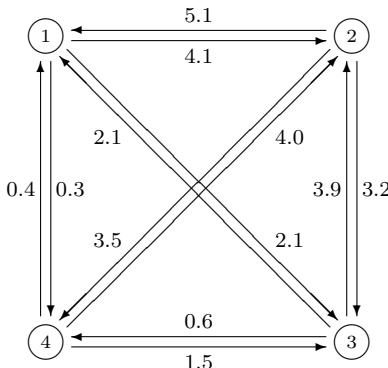


Abbildung 1.5: Umrüstzeiten für vier Produktionsaufträge

Darstellung ein und gibt ihm die Nummer 1, so müssen alle Wege bei Punkt 1 beginnen. Für n Produktionsaufträge gibt es somit $n + 1$ Punkte. Es gibt dann insgesamt $n!$ verschiedene Möglichkeiten, den optimalen Weg zu finden, bestehend darin, die Gesamtrüstzeiten für alle $n!$ Wege zu berechnen und dann den Weg mit der minimalen Zeit zu bestimmen. Interpretiert man die Darstellung aus Abbildung 1.5 in dieser Art (d.h. drei Produktionsaufträge und der Ausgangszustand), so müssen insgesamt sechs verschiedene Wege betrachtet werden. Man sieht leicht, daß der Weg 1–4–3–2 mit einer Umrüstzeit von 5.7 am günstigsten ist. Die Anzahl der zu untersuchenden Wege steigt jedoch sehr schnell an. Bei zehn Produktionsaufträgen müssen bereits 3628800 Wege betrachtet werden. Bei größeren n stößt man bald an Zeitgrenzen. Für $n = 15$ müßten mehr als $1.3 \cdot 10^{12}$ Wege betrachtet werden. Bei einer Rechenzeit von einer Sekunde für 1 Million Wege würden mehr als 15 Tage benötigt. Um zu vertretbaren Rechenzeiten zu kommen, müssen andere Verfahren angewendet werden.

In Kapitel 9 werden Verfahren vorgestellt, mit denen man in einer annehmbaren Zeit zu einem Resultat gelangt, welches relativ nahe an die optimale Lösung herankommt.

1.4 Objektorientierte Programmiersprachen

Eine der wichtigsten Entwicklungen der letzten Jahre auf dem Gebiet der Programmiersprachen sind die objektorientierten Sprachen. Smalltalk und C++ sind Beispiele für solche Sprachen. In traditionellen Programmiersprachen sind Daten und Prozeduren separate Konzepte. Der Programmierer ist dafür verantwortlich, beim Aufruf einer Prozedur diese mit aktuellen Parametern vom vereinbarten Typ zu versorgen. In objektorientierten Programmiersprachen steht das Konzept der Objekte im Mittelpunkt. Objekte haben einen internen Zustand, welcher nur durch entsprechende Methoden verändert werden kann. Methoden entsprechen Prozeduren in traditionellen Programmiersprachen, und das Konzept einer Klasse ist die Verallgemeinerung des Typkonzeptes. Jedes Objekt ist Instanz einer Klasse; diese definiert die Struktur und die Methoden zum Verändern des Zustandes ihrer Instanzen.

Ein Hauptziel der objektorientierten Programmierung ist es, eine gute Strukturierung und eine hohe Wiederverwendbarkeit von Software zu erzielen. Dazu werden Klassen in Hierarchien angeordnet; man spricht dann von Ober- und Unterklassen. Jede Methode einer Klasse ist auch auf die Instanzen aller Unterklassen anwendbar. Man sagt, eine Klasse vererbt ihre Methoden rekursiv an ihre Unterklassen. Der Programmierer hat aber auch die Möglichkeit, die Implementierung einer geerbten Methode zu überschreiben. Die Methode behält dabei ihren Namen; es wird nur die Implementierung geändert. Enthält ein Programm den Aufruf einer Methode für ein Objekt, so kann in vielen Fällen während der Übersetzungszeit nicht entschieden werden, zu welcher Klasse dieses Objekt gehört. Somit kann auch erst zur Laufzeit des Programms die korrekte Implementierung einer Methode ausgewählt werden (*late binding*). Der Grund hierfür liegt hauptsächlich darin, daß der Wert einer Variablen der Klasse C auch Instanz einer Unterklasse von C sein kann.

Die Auswahl der Implementierung einer Methode zur Laufzeit nennt man *Dispatching*. Konzeptionell geht man dabei wie folgt vor:

- Bestimmung der Klasse, zu der das Objekt gehört.
- Falls diese Klasse eine Implementierung für diese Methode zur Verfügung stellt, wird diese ausgeführt.
- Andernfalls werden in aufsteigender Reihenfolge die Oberklassen durchsucht und wie oben verfahren.
- Wird keine Implementierung gefunden, so liegt ein Fehler vor.

Im folgenden wird nur der Fall betrachtet, daß jede Klasse maximal eine Oberklasse hat (*single inheritance*). Hat eine Klasse mehrere Oberklassen (*multiple inheritance*), so muß die Reihenfolge, in der in Schritt c) die Oberklassen durchsucht werden, festgelegt werden.

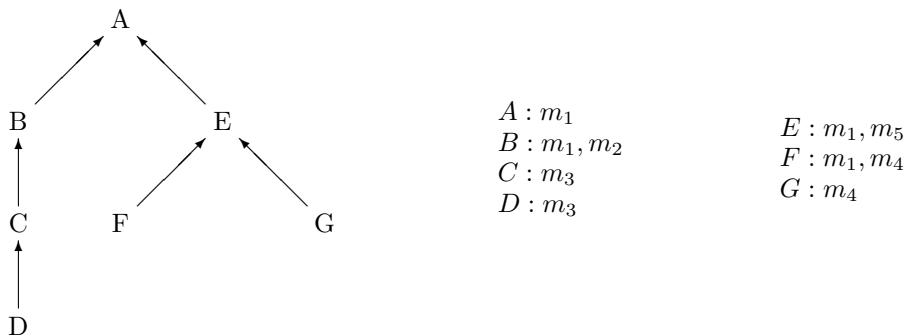


Abbildung 1.6: Eine Klassenhierarchie

Abbildung 1.6 zeigt eine Klassenhierarchie bestehend aus sieben Klassen A, B, \dots, G . Die Unterklassenrelation ist durch einen Pfeil von einer Klasse zu ihrer Oberklasse

gekennzeichnet. Ferner ist angegeben, welche Klassen welche Methoden implementieren. Beispielsweise kann die Methode m_1 auf jedes Objekt angewendet werden, aber es gibt vier verschiedene Implementierungen von m_1 .

Welche Objekte verwenden welche Implementierung von m_1 ? Die folgende Tabelle gibt dazu einen Überblick; hierbei wird die Adresse einer Methode m , welche durch die Klasse K implementiert wird, durch $m|K$ dargestellt. Wird z.B. die Methode m_1 für eine Instanz der Klasse G aufgerufen, so ergibt sich direkt, daß $m_1|E$ die zugehörige Implementierung ist.

Klasse	A	B	C	D	E	F	G
Implementierung	$m_1 A$	$m_1 B$	$m_1 B$	$m_1 B$	$m_1 E$	$m_1 F$	$m_1 E$

Ein Programm einer objektorientierten Sprache besteht im wesentlichen aus Methodenaufrufen. Untersuchungen haben gezeigt, daß in manchen objektorientierten Sprachen bis zu 20% der Laufzeit für Dispatching verwendet wird. Aus diesem Grund besteht ein hohes Interesse an schnellen Dispatchverfahren. Die effizienteste Möglichkeit, Dispatching durchzuführen, besteht darin, zur Übersetzungszeit eine globale *Dispatchtabelle* zu erzeugen. Diese Tabelle enthält für jede Klasse eine Spalte und für jede Methode eine Zeile. Ist m der Name einer Methode und C eine Klasse, so enthält der entsprechende Eintrag der Dispatchtabelle die Adresse der Implementierung der Methode m , welche für Instanzen der Klasse C aufgerufen wird. Ist eine Methode auf die Instanzen einer Klasse nicht anwendbar, so bleibt der entsprechende Eintrag in der Dispatchtabelle leer. Diese Organisation garantiert, daß die Implementierung einer Methode in konstanter Zeit gefunden wird; es ist dabei genau ein Zugriff auf die Dispatchtabelle notwendig. Abbildung 1.7 zeigt die Dispatchtabelle für das oben angegebene Beispiel.

	A	B	C	D	E	F	G
m_1	$m_1 A$	$m_1 B$	$m_1 B$	$m_1 B$	$m_1 E$	$m_1 F$	$m_1 E$
m_2		$m_2 B$	$m_2 B$	$m_2 B$			
m_3			$m_3 C$	$m_3 D$			
m_4						$m_4 F$	$m_4 G$
m_5					$m_5 E$	$m_5 E$	$m_5 E$

Abbildung 1.7: Eine Dispatchtabelle

Der große Nachteil einer solchen Dispatchtabelle ist der Platzverbrauch. Für die komplette Klassenhierarchie eines Smalltalk Systems mit 766 Klassen und 4500 verschiedenen Methoden ergibt sich ein Platzverbrauch von über 13 MBytes. Hierbei benötigt eine Adresse 4 Bytes. Aus diesem Grund sind Dispatchtabellen in dieser Form praktisch nicht verwendbar.

Um den Speicheraufwand zu senken, macht man sich zunutze, daß die meisten Einträge der Dispatchtabelle nicht besetzt sind. Der Speicheraufwand kann reduziert werden,

indem man für zwei Methoden, welche sich nicht beeinflussen, eine einzige Zeile verwendet. Zwei Methoden beeinflussen sich nicht, wenn es kein Objekt gibt, auf welches beide Methoden anwendbar sind. Im obigen Beispiel beeinflussen sich die Methoden m_3 und m_5 nicht; somit können beide Methoden in der Dispatchtabelle eine gemeinsame Zeile verwenden. Um eine optimale Kompression zu finden, wird zunächst untersucht, welche Methoden sich nicht beeinflussen. Dazu wird ein sogenannter *Konfliktgraph* gebildet. In diesem werden Methoden, welche sich beeinflussen, durch eine Kante verbunden. Abbildung 1.8 zeigt den Konfliktgraphen für das obige Beispiel.

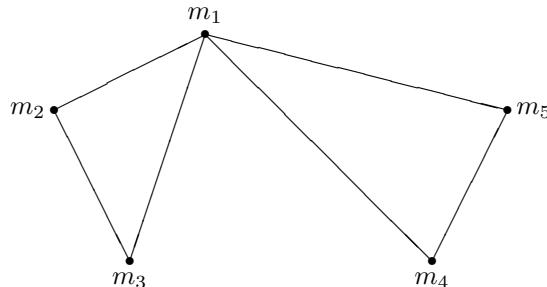


Abbildung 1.8: Ein Konfliktgraph

Um die minimale Anzahl von Zeilen zu finden, wird eine Aufteilung der Menge der Methoden in eine minimale Anzahl von Teilmengen vorgenommen, so daß die Methoden in jeder Teilmenge nicht durch Kanten verbunden sind. Jede Teilmenge entspricht dann einer Zeile. Für das obige Beispiel ist $\{m_1\}$, $\{m_2, m_5\}$ und $\{m_3, m_4\}$ eine solche minimale Aufteilung. Die Dispatchtabelle wird nun mittels zweier Tabellen dargestellt. Die erste Tabelle gibt für jede Methode die entsprechende Zeile in der zweiten Tabelle an. Abbildung 1.9 zeigt diese Tabellen für das obige Beispiel.

Methode	m_1	m_2	m_3	m_4	m_5
Zeile	1	2	3	3	2

	A	B	C	D	E	F	G
1	$m_1 A$	$m_1 B$	$m_1 B$	$m_1 B$	$m_1 E$	$m_1 F$	$m_1 E$
2		$m_2 B$	$m_2 B$	$m_2 B$	$m_5 E$	$m_5 E$	$m_5 E$
3			$m_3 C$	$m_3 D$		$m_4 F$	$m_4 G$

Abbildung 1.9: Komprimierte Dispatchtabelle

Die Auswahl einer entsprechenden Methode erfordert nun zwei Tabellenzugriffe. Die Bestimmung der zugehörigen Zeilen kann dabei schon zur Übersetzungszeit erfolgen;

somit ist zur Laufzeit weiterhin nur ein Tabellenzugriff erforderlich. Der Platzbedarf wird durch dieses Verfahren häufig erheblich reduziert. Wendet man dieses Verfahren auf das oben angegebene Smalltalk System an, so reduziert sich der Speicheraufwand auf etwas mehr als 1 MByte. Dies entspricht einer Reduktion von etwa 90%.

Das Problem bei dieser Vorgehensweise ist die Bestimmung einer Aufteilung der Methoden in eine minimale Anzahl von Teilmengen von sich nicht beeinflussenden Methoden. Dieses Problem wird ausführlich in den Kapiteln 5 und 9 behandelt.

1.5 Suchmaschinen

Das World Wide Web ist inzwischen zu einer bedeutenden Informationsquelle von enormer Größe geworden. Die Suche nach Informationen zu einem bestimmten Thema in diesem weltumspannenden Netzwerk ist schwierig. Allein durch manuelles Durchsuchen von Web-Seiten mit einem Browser ist eine umfassende Informationssuche nicht möglich. Man hat eine bessere Chance die gewünschte Information zu finden, wenn man die Dienste von Suchmaschinen in Anspruch nimmt. Diese verwalten in der Regel einen Index, der Stichwörter mit Web-Seiten verbindet. Eine Anfragesprache ermöglicht eine gezielte Suche in diesem Index. Die Indizes der Suchmaschinen werden automatisch durch so genannte Web-Roboter aufgebaut.

Das Konzept der Suchmaschinen basiert auf einer graphentheoretischen Modellierung des Inhaltes des WWW. Die Grundlage für das Auffinden neuer Dokumente bildet die Hypertextstruktur des Web. Diese macht das Web zu einem gerichteten Graphen, wobei die Dokumente die Ecken bilden. Enthält eine Web-Seite einen Verweis (*Link*) auf eine andere Seite, so zeigt eine Kante von der ersten Seite zu der zweiten Seite. Abbildung 1.10 zeigt ein Beispiel mit vier Web-Seiten und fünf Verweisen. Die wahre Größe des WWW ist nicht bekannt, über die Anzahl der verschiedenen Web-Seiten gibt es nur Schätzungen. Die Betreiber von Suchmaschinen veröffentlichen gelegentlich Angaben über die Anzahl der durch sie indizierten Web-Seiten. Beispielsweise gab der Suchdienstanbieter Overture im August 2003 bekannt, daß sein Web-Roboter 3.151.734.117 Web-Seiten analysiert habe und Overture damit den bisherigen Spaltenreiter Google um etwa 68 Millionen Web-Seiten übertroffen habe.

Um möglichst viele Seiten aufzufinden, ohne dabei Seiten mehrfach zu analysieren, müssen Web-Roboter bei der Suche nach neuen Dokumenten systematisch vorgehen. Häufig werden von einer Web-Seite aus startend rekursiv alle Verweise verfolgt. Ausgehend von einer oder mehreren Startadressen durchsuchen Web-Roboter das Web und extrahieren nach verschiedenen Verfahren Wörter und fügen diese in die Indizes ein. Zur Verwaltung der Indizes werden spezielle Datenbanksysteme verwendet. Dabei werden sowohl statistische Methoden wie Worthäufigkeiten und inverse Dokumentenhäufigkeiten als auch probabilistische Methoden eingesetzt. Viele Roboter untersuchen die Dokumente vollständig, andere nur Titel und Überschriften. Zur Unterstützung komplexer Anfragen werden neben Indizes auch die Anfänge der Dokumente abgespeichert. Bei der Analyse von Web-Seiten kommen HTML-Parser zum Einsatz, die je nach Aufgabenstellung den Quelltext mehr oder weniger detailliert analysieren.

Anwender nutzen den Dienst einer Suchmaschine, indem sie ihre Suchbegriffe in eine ent-

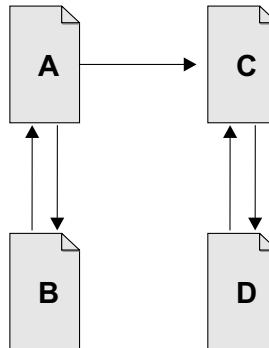


Abbildung 1.10: Vier Web-Seiten mit ihren Verweisen

sprechende Suchmaske eingeben. Die Suchmaschine durchsucht daraufhin ihren Index nach dem Vorkommen der Suchbegriffe und bestimmt die Treffermenge. Die Größe des Web bedingt, daß die Anzahl der gefundenen Dokumente sehr groß sein kann. Darüber hinaus haben viele der gefundenen Dokumente oft nur eine geringe oder sogar keine Relevanz für die Anfrage. Im Laufe der Zeit wurden deshalb verschiedene Verfahren zur Bewertung von Webseiten mit dem Ziel der Relevanzbeurteilung durch Suchmaschinen entwickelt. Die gefundenen Seiten werden dann nach ihrer vermeindlichen Relevanz sortiert. Dies erleichtert Anwendern die Sichtung des Suchergebnisses.

Ein aus unmittelbar nachvollziehbaren Gründen auch heute immer noch von praktisch allen Suchmaschinen genutzter Maßstab ist das Vorkommen eines Suchbegriffs im Text einer Webseite. Dieses Vorkommen wird nach verschiedensten Kriterien wie etwa der relativen Häufigkeit und der Position des Vorkommens oder auch der strukturellen Platzierung des Suchbegriffs im Dokument gewichtet.

Im Zuge der wachsenden wirtschaftlichen Bedeutung von Suchmaschinen versuchten Autoren ihre Web-Seiten so zu gestalten, daß sie bei der Relevanzbeurteilung durch Suchmaschinen gut abschnitten. Beispielsweise wurden zugkräftige Wörter mehrfach an wichtigen Stellen in den Dokumenten plaziert (sogar in Kommentaren). Um einem solchen Mißbrauch entgegenzuwirken, entwickelten Suchmaschinenbetreiber weitere Relevanzkriterien. Ein einfaches Verfahren besteht darin, die Anzahl der Verweise auf ein Dokument als ein grundsätzliches Kriterium für die Bedeutung einzubeziehen. Ein Dokument ist dabei um so wichtiger, je mehr Dokumente auf es verweisen. Es zeigte sich aber bald, daß auch dieses Konzept schnell von Webmastern unterlaufen werden konnte, indem sie automatisch Web-Seiten mit einer Vielzahl von Verweisen auf ihre eigentlichen Seiten generierten.

Ein vom Suchdienst Google zum ersten Mal eingesetztes Konzept heißt *PageRank*. Es basiert auf der Idee, daß ein Dokument zwar bedeutsam ist, wenn andere Dokumente Verweise auf es enthalten, nicht jedes verweisende Dokument ist jedoch gleichwertig. Vielmehr wird einem Dokument unabhängig von seinem Inhalten ein hoher Rang zugeschrieben, wenn anderen bedeutende Dokumente auf es verweisen. Die Wichtigkeit eines

Dokuments bestimmt sich beim PageRank-Konzept aus der Bedeutsamkeit der darauf verweisenden Dokumente, d.h., die Bedeutung eines Dokuments definiert sich rekursiv aus der Bedeutung anderer Dokumente. Somit hat im Prinzip der Rang jedes Dokuments eine Auswirkung auf den Rang aller anderen Dokumente, die Verweisstruktur des gesamten Web wird also in die Relevanzbewertung eines Dokumentes einbezogen.

Der von Lawrence Page und Sergey Brin entwickelte PageRank-Algorithmus ist sehr einfach nachvollziehbar. Für eine Web-Seite W bezeichnet $LC(W)$ die Anzahl der Verweise auf Seite W und $PL(W)$ die Menge der Web-Seiten, welche einen Verweis auf Seite W enthalten. Der PageRank $PR(W)$ einer Seite W berechnet sich wie folgt:

$$PR(W) = (1 - d) + d \sum_{D \in PL(W)} \frac{PR(D)}{LC(D)}$$

Hierbei ist $d \in [0, 1]$ ein einstellbarer Dämpfungsfaktor. Der PageRank einer Seite W bestimmt sich dabei rekursiv aus dem PageRank derjenigen Seiten, die einen Verweis auf Seite W enthalten. Der PageRank der Seiten $D \in PL(W)$ fließt nicht gleichmäßig in den PageRank von W ein. Der PageRank einer Seite wird stets anhand der Anzahl der von der Seite ausgehenden Links gewichtet. Das bedeutet, daß je mehr ausgehende Links eine Seite D hat, um so weniger PageRank gibt sie an W weiter. Der gewichtete PageRank der Seiten $D \in PL(W)$ wird addiert. Dies hat zur Folge, daß jeder zusätzliche eingehende Link auf W stets den PageRank dieser Seite erhöht. Schließlich wird die Summe der gewichteten PageRanks der Seiten $D \in PL(W)$ mit dem Dämpfungsfaktor d multipliziert. Hierdurch wird das Ausmaß der Weitergabe des PageRanks von einer Seite auf eine andere verringert.

Die Eigenschaften des PageRank werden jetzt anhand des in Abbildung 1.10 dargestellten Beispieles veranschaulicht. Der Dämpfungsfaktor d wird Angaben von Page und Brin zufolge für tatsächliche Berechnungen oft auf 0.85 gesetzt. Der Einfachheit halber wird d an dieser Stelle ein Wert von 0.5 zugewiesen, wobei der Wert von d zwar Auswirkungen auf den PageRank hat, das Prinzip jedoch nicht beeinflußt. Für obiges Beispiel ergeben sich folgende Gleichungen:

$$\begin{aligned} PR(A) &= 0.5 + 0.5 PR(B)/2 \\ PR(B) &= 0.5 + 0.5 PR(A) \\ PR(C) &= 0.5 + 0.5 (PR(B)/2 + PR(D)) \\ PR(D) &= 0.5 + 0.5 PR(C) \end{aligned}$$

Dieses Gleichungssystem läßt sich sehr einfach lösen. Es ergibt sich folgende Lösung:

$$PR(A) = 0.71428 \quad PR(B) = 0.85714 \quad PR(C) = 1.2857 \quad PR(D) = 1.1428$$

Somit hat Seite C den höchsten und A den niedrigsten PageRank.

Es stellt sich die Frage, wie der PageRank von Web-Seiten berechnet werden kann. Eine ganzheitliche Betrachtung des WWW ist sicherlich ausgeschlossen, deshalb erfolgt in der Praxis eine näherungsweise, iterative Berechnung des PageRank. Dazu wird zunächst jeder Seite ein PageRank mit Wert 1 zugewiesen, und anschließend wird der PageRank

aller Seiten in mehreren Iterationen ermittelt. Abbildung 1.11 zeigt das Ergebnis der ersten fünf Iterationen für das betrachtete Beispiel. Bereits nach sehr wenigen Iterationen wird eine sehr gute Näherung an die tatsächlichen Werte erreicht. Für die Berechnung des PageRanks für das komplette Web werden von Page und Brin etwa 100 Iterationen als hinreichend genannt.

Iteration	$PR(A)$	$PR(B)$	$PR(C)$	$PR(D)$
0	1	1	1	1
1	0.75	0.875	1.21875	1.109375
2	0.71875	0.859375	1.2695312	1.1347656
3	0.71484375	0.8574219	1.2817383	1.1408691
4	0.71435547	0.85717773	1.284729	1.1423645
5	0.71429443	0.8571472	1.285469	1.1427345

Abbildung 1.11: Die iterative Berechnung des PageRank

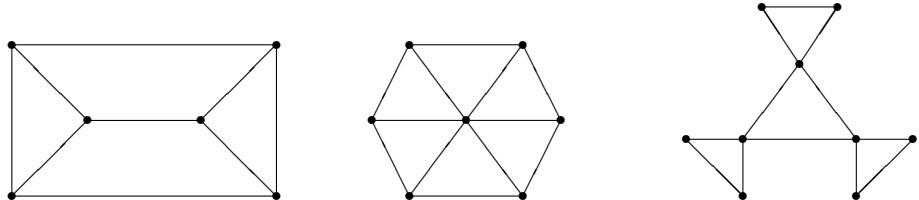
Die bei der systematischen Suche nach neuen Web-Seiten von Web-Robotern eingesetzten Verfahren wie Tiefen- und Breitensuche werden in Kapitel 4 ausführlich vorgestellt.

1.6 Literatur

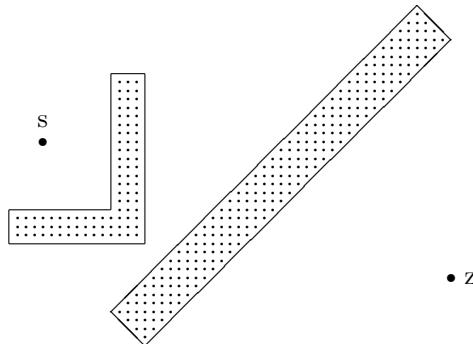
Verletzlichkeit von Kommunikationsnetzen ist ein Teilgebiet der *Zuverlässigkeitstheorie*. Eine ausführliche Darstellung der graphentheoretischen Aspekte findet man in [80]. Eine gute Übersicht über *Wegeplanungsverfahren* in der Robotik findet man in [18]. Das Problem der optimalen Umrüstzeiten ist eng verwandt mit dem *Traveling-Salesman Problem*. Eine ausführliche Darstellung findet man in [87]. Verfahren zur Optimierung von Methodendispatching in objektorientierten Programmiersprachen sind in [6] beschrieben. Weitere Information zur Bestimmung des PageRank einer Web-Seite findet man in [16, 102]. Weitere praktische Anwendungen der Graphentheorie sind in [62, 63] beschrieben.

1.7 Aufgaben

- Bestimmen Sie Verbindungs- und Knotenzusammenhang der folgenden drei Netzwerke.



2. Bestimmen Sie für die im folgenden dargestellte Menge von Hindernissen das System aller Geradensegmente analog zu Abbildung 1.3. Bestimmen Sie den kürzesten Weg von s nach z .

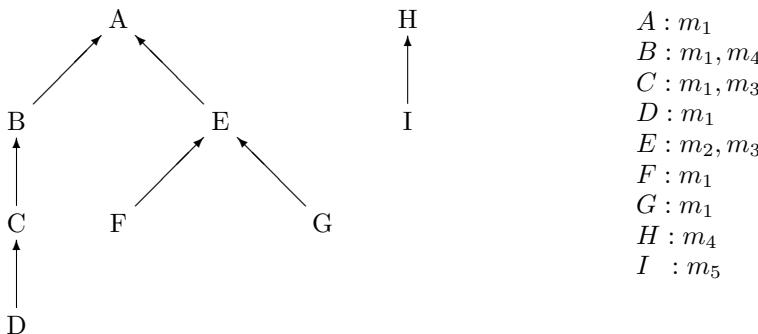


3. Im folgenden sind die Umrüstzeiten T_{ij} für vier Produktionsaufträge für eine Fertigungszelle in Form einer Matrix gegeben.

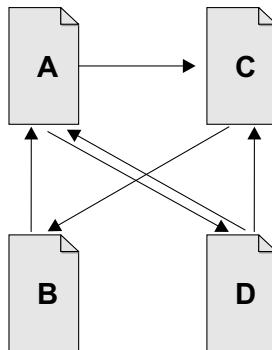
$$\begin{pmatrix} 0 & 1.2 & 2 & 2.9 \\ 0.8 & 0 & 1.0 & 2.3 \\ 2 & 1.2 & 0 & 3.4 \\ 5.1 & 1.9 & 2.1 & 0 \end{pmatrix}$$

Bestimmen Sie die optimale Reihenfolge der Produktionsaufträge, um die geringste Gesamtumrüstzeit zu erzielen. Die Fertigungszelle befindet sich anfangs im Zustand 1.

4. Bestimmen Sie für die folgende Klassenhierarchie und die angegebenen Methoden den Konfliktgraphen und geben Sie eine optimal komprimierte Dispatchtabelle an.



5. Bestimmen Sie den PageRank aller Seiten der folgenden Hypertextstruktur ($d = 0.5$).



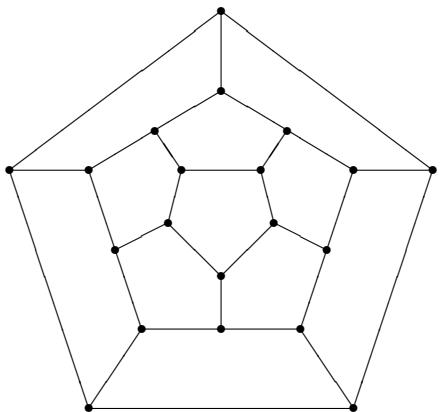
6. Es sei M eine Menge von Web-Seiten mit folgenden beiden Eigenschaften:

- Jede Seite in M enthält mindestens einen Verweis und
- außerhalb von M gibt es keine Seiten, welche auf Seiten in M verweisen.

Beweisen Sie, daß die Summe der Werte des PageRank aller Seiten aus M gleich der Anzahl der Seiten in M ist.

Kapitel 2

Einführung



Dieses Kapitel führt in die Grundbegriffe der Graphentheorie ein und beschreibt wichtige Klassen von Graphen. Es werden verschiedene Datenstrukturen für Graphen betrachtet und deren Speicherverbrauch bestimmt. Als Einführung in die algorithmische Graphentheorie wird der Algorithmus zur Bestimmung des transitiven Abschlusses eines Graphen diskutiert. Ein weiterer Abschnitt stellt Mittel zur Verfügung, um Zeit- und Platzbedarf von Algorithmen abzuschätzen und zu vergleichen. Schließlich behandelt der letzte Abschnitt dieses Kapitels Klassen von Zufallsgraphen, mit denen die Algorithmen getestet und die Laufzeit verschiedener Algorithmen verglichen werden können.

Die nachfolgenden Kapitel beschäftigen sich dann ausführlich mit weiterführenden Konzepten.

2.1 Grundlegende Definitionen

Ein *ungerichteter Graph* besteht aus einer Menge E von *Ecken* und einer Menge K von *Kanten*. Eine Kante ist gegeben durch ein ungeordnetes Paar von Ecken, den Endecken

der Kante. Die Ecken werden oft auch Knoten oder Punkte genannt. Ein *gerichteter Graph* besteht aus einer Menge E von Ecken und einer Menge K von *gerichteten Kanten*, die durch geordnete Paare von Ecken, den Anfangsecken und den Endencken, bestimmt sind. Eine anschauliche Vorstellung von Graphen vermittelt ihre geometrische Darstellung durch Diagramme in der Euklidischen Ebene. Die Ecken werden dabei als Punkte und die Kanten als Verbindungsstrecken der Punkte repräsentiert. Die Richtungen der Kanten in gerichteten Graphen werden durch Pfeile dargestellt.

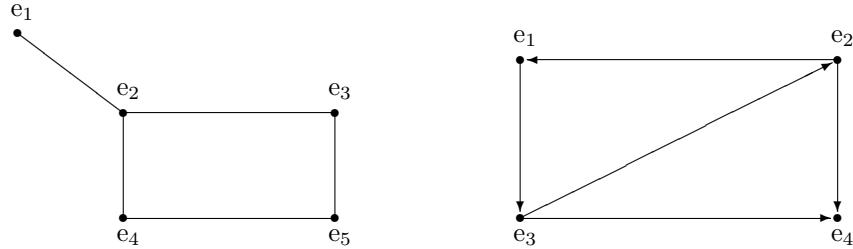


Abbildung 2.1: Diagramme von Graphen

Abbildung 2.1 zeigt zwei Diagramme, die einen gerichteten und einen ungerichteten Graphen repräsentieren. Die Ecken- bzw. Kantenmenge des ungerichteten Graphen ist:

$$\begin{aligned} E &= \{e_1, e_2, e_3, e_4, e_5\} \\ K &= \{(e_1, e_2), (e_2, e_3), (e_3, e_5), (e_5, e_4), (e_2, e_4)\} \end{aligned}$$

und die des gerichteten Graphen ist:

$$\begin{aligned} E &= \{e_1, e_2, e_3, e_4\} \\ K &= \{(e_1, e_3), (e_2, e_1), (e_2, e_4), (e_3, e_2), (e_3, e_4)\} \end{aligned}$$

Eine geometrische Repräsentation bestimmt eindeutig einen Graphen, aber ein Graph kann auf viele verschiedene Arten durch ein Diagramm dargestellt werden. Die geometrische Lage der Ecken in der Ebene trägt keinerlei Bedeutung. Die beiden Diagramme in Abbildung 2.2 repräsentieren den gleichen Graphen, denn die entsprechenden Mengen E und K sind identisch. Man beachte, daß die bei der geometrischen Darstellung entstehenden Schnittpunkte zweier Kanten nicht unbedingt Ecken entsprechen müssen. Ferner müssen die Kanten auch nicht durch Strecken dargestellt werden, sondern es können beliebige Kurven verwendet werden.

Die getroffene Definition eines Graphen schließt nicht aus, daß es zwischen zwei Ecken mehr als eine Kante gibt und daß es Kanten gibt, deren Anfangs- und Endecke identisch sind. Graphen, bei denen eine solche Situation nicht vorkommt, nennt man *schlicht*. Bei schlichten Graphen gibt es somit keine Kanten, deren Anfangs- und Endecke gleich sind, und keine Mehrfachkanten zwischen zwei Ecken. Sofern es nicht ausdrücklich vermerkt ist, werden in diesem Buch nur schlichte Graphen behandelt. Dies schließt aber nicht aus, daß es bei gerichteten Graphen zwischen zwei Ecken zwei Kanten mit entgegengesetzten Richtungen geben kann.

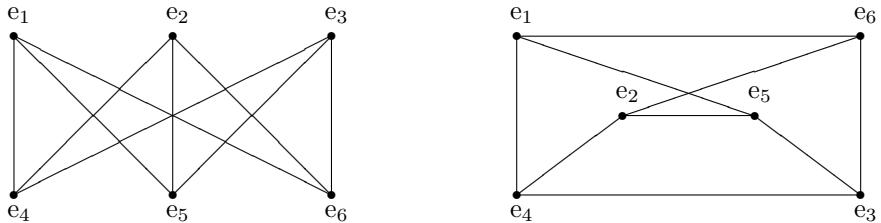


Abbildung 2.2: Verschiedene Diagramme für den gleichen Graphen

Die Struktur eines Graphen ist dadurch bestimmt, welche Ecke mit welcher verbunden ist, die Bezeichnung der Ecken trägt keine Bedeutung. Deshalb kann sie auch bei der geometrischen Repräsentation weggelassen werden. Häufig werden die Ecken mit den Zahlen von 1 bis n durchnumeriert. Im folgenden wird die Anzahl der Ecken immer mit n und die der Kanten immer mit m bezeichnet. Abbildung 2.3 zeigt alle elf ungerichteten Graphen, die genau vier Ecken haben.

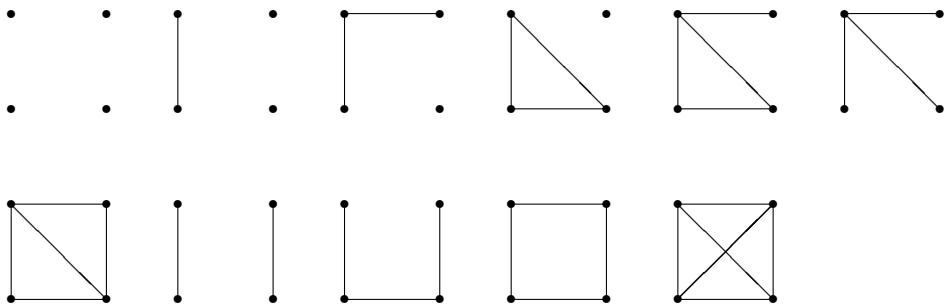


Abbildung 2.3: Die ungerichteten Graphen mit genau vier Ecken

Zwei Ecken e und f eines ungerichteten Graphen heißen *benachbart*, wenn es eine Kante von e nach f gibt. Die *Nachbarn* $N(e)$ einer Ecke e sind die Ecken, die zu e benachbart sind. Eine Ecke e eines ungerichteten Graphen ist *inzident* mit einer Kante k des Graphen, wenn e eine Endecke der Kante k ist.

Der *Grad* $g(e)$ einer Ecke e in einem ungerichteten Graphen G ist die Anzahl der Kanten, die mit e inzident sind. Mit $\Delta(G)$ (oder kurz Δ) wird der maximale und mit $\delta(G)$ (oder kurz δ) der minimale Eckengrad von G bezeichnet.

In einem gerichteten Graphen unterscheidet man zwischen *Ausgangsgrad* $g^+(e)$ einer Ecke e und dem *Eingangsgrad* $g^-(e)$. Der Ausgangsgrad ist gleich der Anzahl der Kanten, deren Anfangsseite e ist, und der Eingangsgrad ist gleich der Anzahl der Kanten mit Endecke e . Abbildung 2.4 illustriert diese Begriffe an zwei Beispielen. Eine Ecke e mit $g^-(e) = g^+(e) = 0$ (bzw. $g(e) = 0$) heißt *isolierte Ecke*. Der erste Graph in Abbildung

2.3 besteht aus vier isolierten Ecken, d.h. seine Kantenmenge ist leer.

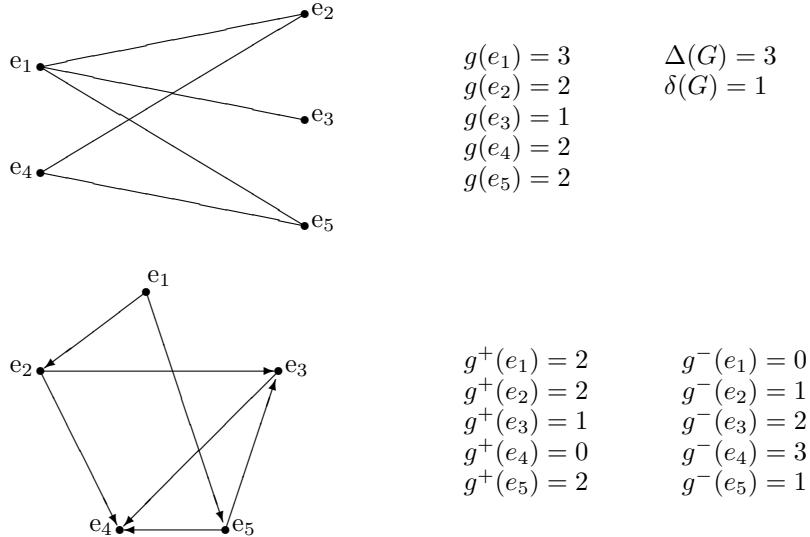


Abbildung 2.4: Die Grade von Ecken

In einem ungerichteten Graphen ist die Summe der Grade aller Ecken gleich der zweifachen Anzahl der Kanten:

$$\sum_{i=1}^n g(i) = 2m.$$

Dies folgt daraus, daß jede Kante zum Eckengrad von zwei Ecken beiträgt. Aus der letzten Gleichung kann eine nützliche Folgerung gezogen werden: Die Anzahl der Ecken mit ungeradem Eckengrad in einem ungerichteten Graphen ist eine gerade Zahl.

Eine Folge von Kanten k_1, k_2, \dots, k_s eines Graphen heißt *Kantenzug*, falls es eine Folge von Ecken e_0, e_1, \dots, e_s des Graphen gibt, so daß für jedes i zwischen 1 und s die Kante k_i gleich (e_{i-1}, e_i) ist. Ein Kantenzug heißt *Weg*, wenn alle verwendeten Kanten verschieden sind. Der Begriff Weg entspricht dem anschaulichen Wegbegriff in der geometrischen Darstellung des Graphen. Man nennt e_0 die *Anfangsecke* und e_s die *Endecke* des Weges. Ist $e_0 = e_s$, so heißt der Weg *geschlossen*, andernfalls heißt er *offen*. Der gerichtete Graph aus Abbildung 2.1 enthält einen Weg von e_1 nach e_4 bestehend aus der Folge der Kanten $(e_1, e_3), (e_3, e_2)$ und (e_2, e_4) , aber es gibt keinen Weg von e_4 nach e_1 . Man beachte, daß es auch in ungerichteten Graphen Ecken geben kann, zwischen denen kein Weg existiert. Man vergleiche dazu die Graphen in Abbildung 2.3. Ein Weg heißt *einfacher Weg*, falls alle verwendeten Ecken (bis auf Anfangs- und Endecke) verschieden sind.

Der *Abstand* $d(e, f)$ zweier Ecken e, f eines Graphen ist gleich der minimalen Anzahl von Kanten eines Weges mit Anfangsecke e und Endecke f . Gibt es keinen solchen Weg,

so ist $d(e, f) = \infty$. Es ist $d(e, e) = 0$ für jede Ecke e eines Graphen. Für den ersten Graphen aus Abbildung 2.4 gilt $d(e_1, e_4) = 2$.

Eine Ecke e kann man von der Ecke f erreichen, falls es einen Weg von e nach f gibt. Die Menge aller Ecken in einem gerichteten Graphen, die man von einer Ecke e aus über genau eine Kante erreichen kann, nennt man die *Nachfolger* von e , und die Menge aller Ecken, von denen man e über genau eine Kante erreichen kann, nennt man die *Vorgänger* von e .

Das *Komplement* eines ungerichteten Graphen G ist ein ungerichteter Graph \bar{G} mit der gleichen Eckenmenge. Zwei Ecken sind in \bar{G} dann und nur dann benachbart, wenn Sie nicht in G benachbart sind. Abbildung 2.5 zeigt einen ungerichteten Graphen und sein Komplement.



Abbildung 2.5: Ein ungerichteter Graph und sein Komplement

Sei G ein Graph mit Eckenmenge E und Kantenmenge K . Jeden Graphen, dessen Eckenmenge eine Teilmenge von E und dessen Kantenmenge eine Teilmenge von K ist, nennt man *Untergraph* von G . Ist D eine Teilmenge von E , so nennt man den Graphen mit Eckenmenge D und der Kantenmenge

$$\{(e, f) \mid e, f \in D \text{ und } (e, f) \in K\}$$

den von D induzierten *Untergraphen* von G . Der induzierte Untergraph besteht aus genau den Kanten von G , deren Ecken in D liegen. Abbildung 2.6 zeigt einen Graphen mit sechs Ecken und den von der Menge $D = \{1, 2, 3, 5, 6\}$ induzierten Untergraphen. Es sei J eine Teilmenge der Kanten, dann nennt man den Graphen mit Kantenmenge J und Eckenmenge $\{e \mid e \text{ ist Endecke einer Kante aus } J\}$ den von J induzierten Untergraphen.

2.2 Spezielle Graphen

Ein ungerichteter Graph heißt *zusammenhängend*, falls es für jedes Paar e, f von Ecken einen Weg von e nach f gibt. Die Eckenmenge E eines ungerichteten Graphen kann in disjunkte Teilmengen E_1, \dots, E_s zerlegt werden, so daß die von E_i induzierten Untergraphen zusammenhängend sind. Dazu bilde man folgende Relation: $e, f \in E$ sind

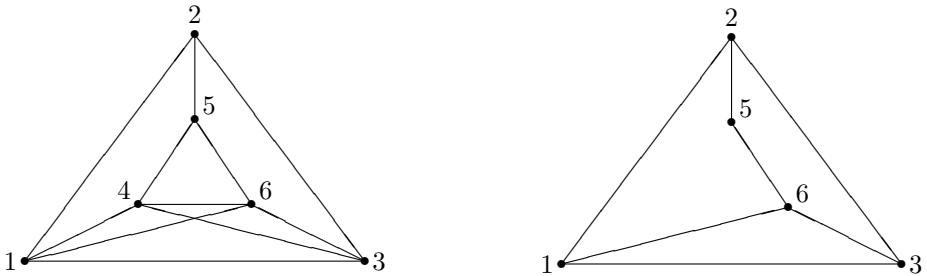


Abbildung 2.6: Ein Graph und ein induzierter Untergraph

äquivalent, falls es einen Weg von e nach f gibt. Dies ist eine Äquivalenzrelation. Die von den Äquivalenzklassen dieser Relation induzierten Untergraphen sind zusammenhängend. Man nennt sie die *Zusammenhangskomponenten*. Ein zusammenhängender Graph besteht also nur aus einer Zusammenhangskomponente. Der erste Graph aus Abbildung 2.3 besteht aus vier und der zweite Graph aus drei Zusammenhangskomponenten.

Ein ungerichteter Graph heißt *vollständig*, wenn je zwei Ecken durch eine Kante verbunden sind. Der vollständige Graph mit n Ecken wird mit K_n bezeichnet. Der letzte Graph in Abbildung 2.3 ist der Graph K_4 . Ein Graph heißt *bipartit*, wenn sich die Menge E der Ecken in zwei disjunkte Teilmengen E_1 und E_2 zerlegen lässt, so daß weder Ecken aus E_1 , noch Ecken aus E_2 untereinander benachbart sind. Der obere Graph aus Abbildung 2.4 ist bipartit. Hierbei ist $E_1 = \{e_1, e_4\}$ und $E_2 = \{e_2, e_3, e_5\}$. Ein Graph heißt *vollständig bipartit*, falls er bipartit ist und jede Ecke aus E_1 zu jeder Ecke aus E_2 benachbart ist. Besteht E_1 aus m Ecken und E_2 aus n Ecken, so wird der vollständig bipartite Graph mit $K_{m,n}$ bezeichnet. Der Graph aus Abbildung 2.2 ist der Graph $K_{3,3}$.

Ein Graph heißt *zyklisch*, falls er aus einem einfachen geschlossenen Weg besteht. Der zyklische Graph mit n Ecken wird mit C_n bezeichnet. Der vorletzte Graph in der Abbildung 2.3 ist der Graph C_4 . Ein Graph, in dem jede Ecke den gleichen Grad hat, heißt *regulär*. Die Graphen C_n , K_n und $K_{n,n}$ sind regulär. Die Ecken des Graphen aus Abbildung 2.7 haben alle den Grad 3. Dieser Graph ist nach dem Graphentheoretiker J. Petersen benannt.

Ein ungerichteter, zusammenhängender Graph heißt *Baum*, falls er keinen geschlossenen Weg enthält. In Bäumen gibt es zwischen je zwei Ecken des Graphen genau einen Weg. Ein *Wurzelbaum* ist ein Baum, bei dem eine Ecke als Wurzel ausgezeichnet ist. Von der Wurzel verschiedene Ecken mit Grad 1 nennt man Blätter.

Ein gerichteter Graph B heißt *Baum*, wenn der zugrundeliegende ungerichtete Graph ein Wurzelbaum ist und wenn es in B von der Wurzel aus genau einen Weg zu jeder Ecke gibt. In einem gerichteten Baum hat jede Ecke außer der Wurzel den Eingrad 1. Abbildung 2.8 zeigt einen gerichteten Baum, bei dem der Ausgrad jeder Ecke höchstens zwei ist. Solche Bäume nennt man *Binärbäume*. Binärbäume finden große Anwendung in der Informatik. Ecken mit Ausgrad 0 nennt man Blätter. Die anderen Ecken nennt man *innere Ecken*.

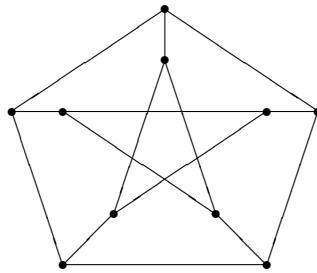


Abbildung 2.7: Der Petersen Graph

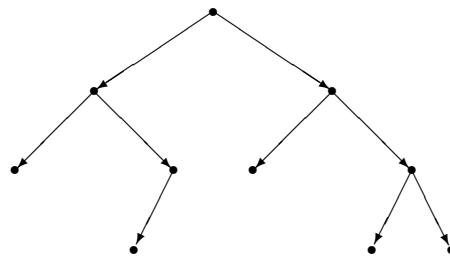


Abbildung 2.8: Ein Binärbaum

Ein Graph heißt *planar* oder *plättbar*, falls es eine Darstellung in der Ebene gibt, bei der sich die Kanten höchstens in ihren Endpunkten schneiden. Abbildung 2.9 zeigt links einen planaren Graphen und rechts eine kreuzungsfreie Darstellung dieses Graphen.

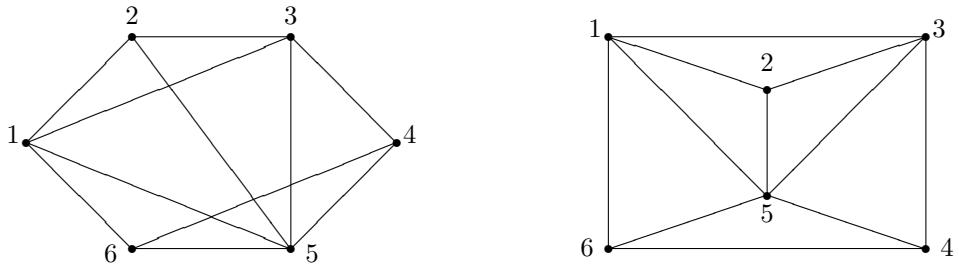


Abbildung 2.9: Ein planarer Graph und eine kreuzungsfreie Darstellung

Die Graphen $K_{3,3}$ und K_5 sind Beispiele für nicht planare Graphen. Sie besitzen keine kreuzungsfreien Darstellungen in der Ebene (man vergleiche dazu Abbildung 2.2). Sie spielen bei der Charakterisierung von planaren Graphen eine wichtige Rolle, denn sie sind in gewisser Weise die „kleinsten“ nicht planaren Graphen. In Abschnitt 5.4 wird bewiesen, daß sie nicht planar sind.

2.3 Graphalgorithmen

Es gibt keine Patentrezepte, um effiziente Algorithmen für Graphen zu entwerfen. Fast jede Problemstellung erfordert eine andere Vorgehensweise. Trotzdem kann man drei verschiedene Aspekte des Entwurfs von Graphalgorithmen unterscheiden. Diese wiederholen sich mehr oder weniger deutlich in jedem Algorithmusentwurf. Der erste Schritt besteht meist in einer mathematischen Analyse des Problems und führt häufig zu einer Charakterisierung der Lösung. Die dabei verwendeten Hilfsmittel aus der diskreten Mathematik sind meist relativ einfach. Nur selten werden tieferliegende mathematische Hilfsmittel herangezogen. Dieser erste Schritt führt meist zu Algorithmen, die nicht sehr effizient sind. Der zweite Schritt besteht darin, den Algorithmus mittels einiger Standardtechniken zu verbessern. Dies sind Techniken, die bei Algorithmen auf den verschiedensten Gebieten erfolgreich eingesetzt werden. Beispiele hierfür sind *dynamisches Programmieren*, *greedy-Techniken* und *divide and conquer*. Diese Techniken führen häufig zu erheblichen Verbesserungen der Effizienz der Algorithmen. Der dritte Schritt besteht in dem Entwurf einer Datenstruktur, welche die Basisoperationen, die der Algorithmus verwendet, effizient unterstützt. Dies führt häufig zu einer weiteren Verbesserung der Laufzeit oder zu einer Senkung des Specheraufwandes. Meistens geht dieser Effizienzgewinn mit einem erhöhten Implementierungsaufwand einher.

Nicht immer werden diese drei Schritte in der angegebenen Reihenfolge angewendet. Manchmal werden auch einige Schritte wiederholt. Häufig führt eine mathematische Analyse der verwendeten Datenstrukturen zu einer Verbesserung. In diesem Buch wird versucht, die grundlegenden Techniken von Graphalgorithmen verständlich darzustellen. Es ist nicht das Ziel, den aktuellen Stand der Forschung zu präsentieren, sondern dem Leser einen Zugang zu Graphalgorithmen zu eröffnen, indem die wichtigsten Techniken erklärt werden. Am Ende jedes Kapitels werden Literaturhinweise auf aktuelle Entwicklungen gegeben.

In den folgenden beiden Abschnitten werden grundlegende Voraussetzungen für Graphalgorithmen bereitgestellt. Zum einen sind dies Datenstrukturen für Graphen und zum anderen Hilfsmittel, um die Effizienz von Algorithmen zu beschreiben.

2.4 Datenstrukturen für Graphen

Bei der Lösung eines Problems ist es notwendig, eine Abstraktion der Wirklichkeit zu wählen. Danach erfolgt die Wahl der Darstellung dieser Abstraktion. Diese Wahl muß immer im Bezug auf die mit den Daten durchzuführenden Operationen erfolgen. Diese ergeben sich aus dem Algorithmus für das vorliegende Problem und sind somit problemabhängig. In diesem Zusammenhang wird auch die Bedeutung der Programmiersprache ersichtlich. Die Datentypen, die die Programmiersprache zur Verfügung stellt, beeinflussen die Art der Darstellung wesentlich, so kann man etwa ohne dynamische Datenstrukturen gewisse Darstellungen nur schwer oder überhaupt nicht unterstützen. Die wichtigsten Operationen für Graphen sind die folgenden:

1. Feststellen, ob ein Paar von Ecken benachbart ist.
2. Bestimmung der Nachbarn einer Ecke.

3. Bestimmung der Nachfolger und Vorgänger einer Ecke in einem gerichteten Graphen.

Neben der Effizienz der Operationen ist auch der Verbrauch an Speicherplatz ein wichtiges Kriterium für die Wahl einer Datenstruktur. Häufig stellt man hierbei eine Wechselwirkung zwischen dem Speicherplatzverbrauch und der Effizienz der Operationen fest. Eine sehr effiziente Ausführung der Operationen geht häufig zu Lasten des Speicherplatzes.

Weiterhin muß beachtet werden, ob es sich um eine statische oder um eine dynamische Anwendung handelt. Bei dynamischen Anwendungen wird der Graph durch den Algorithmus verändert. In diesem Fall sind dann Operationen zum Löschen und Einfügen von Ecken bzw. Kanten notwendig. Häufig sollen auch nur Graphen mit einer gewissen Eigenschaft (z.B. ohne geschlossene Wege) dargestellt werden. In diesen Fällen führt die Ausnutzung der speziellen Struktur häufig zu effizienteren Datenstrukturen. Im folgenden werden nun die wichtigsten Datenstrukturen für Graphen vorgestellt. Datenstrukturen für spezielle Klassen von Graphen werden in den entsprechenden Kapiteln diskutiert.

2.4.1 Adjazenzmatrix

Die naheliegendste Art für die Darstellung eines schlichten Graphen G ist eine $n \times n$ Matrix $A(G)$. Hierbei ist der Eintrag a_{ij} von $A(G)$ gleich 1, falls es eine Kante von i nach j gibt, andernfalls gleich 0. Da die Matrix $A(G)$ (im folgenden nur noch mit A bezeichnet) direkt die Nachbarschaft der Ecken widerspiegelt, nennt man sie *Adjazenzmatrix*. Da ihre Einträge nur die beiden Werte 0 und 1 annehmen, kann $A(G)$ als Boolesche Matrix dargestellt werden. Es lassen sich sowohl gerichtete als auch ungerichtete Graphen darstellen, wobei im letzten Fall die Adjazenzmatrizen symmetrisch sind. Abbildung 2.10 zeigt einen gerichteten Graphen und seine Adjazenzmatrix.

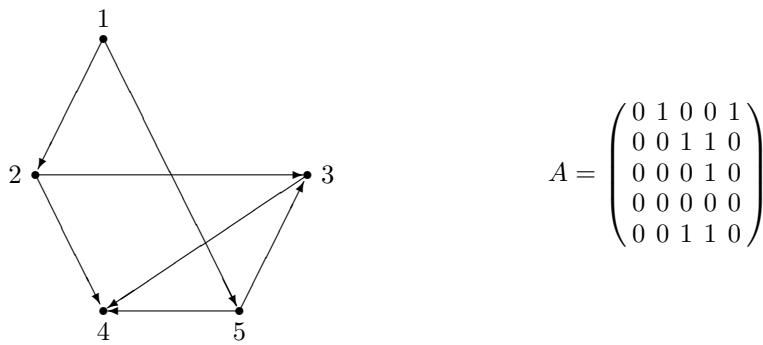


Abbildung 2.10: Ein gerichteter Graph und seine Adjazenzmatrix

Ist ein Diagonaleintrag der Adjazenzmatrix gleich 1, so bedeutet dies, daß eine Schlinge vorhanden ist. Um festzustellen, ob es eine Kante von i nach j gibt, muß lediglich der Eintrag a_{ij} betrachtet werden. Somit kann diese Operation unabhängig von der

Größe des Graphen in einem Schritt durchgeführt werden. Um die Nachbarn einer Ecke i in einem ungerichteten Graphen festzustellen, müssen alle Einträge der i -ten Zeile oder der i -ten Spalte durchsucht werden. Diese Operation ist somit unabhängig von der Anzahl der Nachbarn. Auch wenn eine Ecke keine Nachbarn hat, sind insgesamt n Überprüfungen notwendig. Für das Auffinden von Vorgängern und Nachfolgern in gerichteten Graphen gilt eine analoge Aussage. Im ersten Fall wird die entsprechende Spalte und im zweiten Fall die entsprechende Zeile durchsucht.

Für die Adjazenzmatrix werden unabhängig von der Anzahl der Kanten immer n^2 Speicherplätze benötigt. Für Graphen mit weniger Kanten sind dabei die meisten Einträge gleich 0. Da die Adjazenzmatrix eines ungerichteten Graphen symmetrisch ist, genügt es in diesem Fall, die Einträge oberhalb der Diagonale zu speichern. Dann benötigt man nur $n(n - 1)/2$ Speicherplätze (vergleichen Sie Aufgabe 15).

2.4.2 Adjazenzliste

Eine zweite Art zur Darstellung von Graphen sind Adjazenzlisten. Der Graph wird dabei dadurch dargestellt, daß für jede Ecke die Liste der Nachbarn abgespeichert wird. Der Speicheraufwand ist in diesem Fall direkt abhängig von der Anzahl der Kanten. Es lassen sich gerichtete und ungerichtete Graphen darstellen. Für die Realisierung der Adjazenzliste bieten sich zwei Möglichkeiten an. Bei der ersten Möglichkeit werden die Listen mit Zeigern realisiert. Dabei wird der Graph durch ein Feld A der Länge n dargestellt. Der i -te Eintrag enthält einen Zeiger auf die Liste der Nachbarn von i . Die Nachbarn selber werden als verkettete Listen dargestellt. Abbildung 2.11 zeigt links die entsprechende Darstellung des Graphen aus Abbildung 2.10.

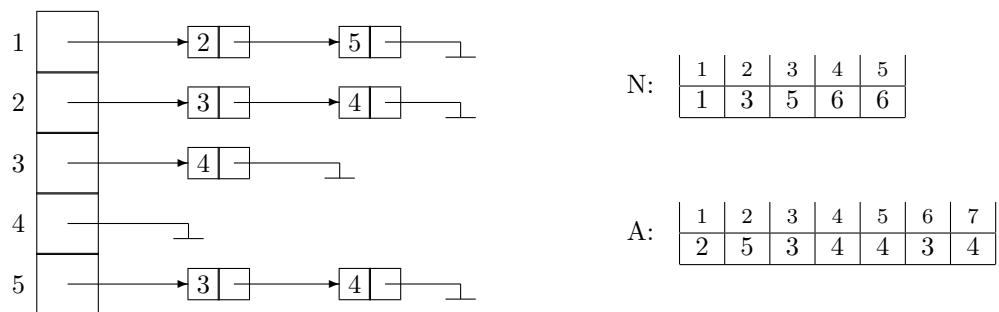


Abbildung 2.11: Adjazenzliste mit Zeigern und mit Feldern

Bei der zweiten Möglichkeit verwendet man ein Feld A , um die Nachbarn abzuspeichern. In diesem Feld werden die Nachbarn lückenlos abgelegt: zuerst die Nachbarn der Ecke 1, dann die Nachbarn der Ecke 2 etc. Für gerichtete Graphen hat das Feld A die Länge m und für ungerichtete Graphen die Länge $2m$. Um festzustellen, wo die Nachbarliste für eine bestimmte Ecke beginnt, wird ein zweites Feld N der Länge n verwaltet. Dieses Feld enthält die Indizes der Anfänge der Nachbarlisten; d.h. eine Ecke i hat genau dann

Nachbarn, falls $N[i + 1] > N[i]$ gilt. Die Nachbarn sind dann in den Komponenten $A[N[i]]$ bis $A[N[i + 1] - 1]$ gespeichert. Eine Ausnahmebehandlung erfordert die n -te Ecke (vergleichen Sie Aufgabe 14). Abbildung 2.11 zeigt rechts die entsprechende Darstellung des Graphen aus Abbildung 2.10.

Um festzustellen, ob es eine Kante von i nach j gibt, muß die Nachbarschaftsliste von Ecke i durchsucht werden. Falls es keine solche Kante gibt, muß die gesamte Liste durchsucht werden. Diese kann bis zu $n - 1$ Einträge haben. Diese Operation ist somit nicht mehr unabhängig von der Größe des Graphen. Einfach ist die Bestimmung der Nachbarn einer Ecke; die Liste liegt direkt vor. Aufwendiger ist die Bestimmung der Vorgänger einer Ecke in einem gerichteten Graphen. Dazu müssen alle Nachbarlisten durchsucht werden. Wird diese Operation häufig durchgeführt, lohnt es sich, zusätzliche redundante Information zu speichern. Mit Hilfe von zwei weiteren Feldern werden die Vorgängerlisten abgespeichert. Mit ihnen können auch die Vorgänger einer Ecke direkt bestimmt werden. Die Adjazenzliste benötigt für einen gerichteten Graphen $n + m$ und für einen ungerichteten Graphen $n + 2m$ Speicherplätze. Für Graphen, deren Kantenzahl m viel kleiner als die maximale Anzahl ist, verbraucht die Adjazenzliste weniger Platz als die Adjazenzmatrix. Bei dynamischen Anwendungen ist die Adjazenzliste basierend auf Zeigern den anderen Darstellungen vorzuziehen.

2.4.3 Kantenliste

Bei der Kantenliste werden die Kanten explizit in einer Liste gespeichert. Dabei wird eine Kante als Paar von Ecken repräsentiert. Wie bei der Adjazenzliste bieten sich zwei Arten der Realisierung an: mittels Zeigern oder mittels Feldern. Im folgenden gehen wir kurz auf die zweite Möglichkeit ein. Die Kantenliste wird mittels eines $2 \times m$ Feldes K realisiert. Hierbei ist $(K[1, i], K[2, i])$ die i -te Kante. Es werden $2m$ Speicherplätze benötigt. Abbildung 2.12 zeigt die Kantenliste des Graphen aus Abbildung 2.10.

1	2	3	4	5	6	7
1	1	2	2	3	5	5
2	5	3	4	4	4	3

Abbildung 2.12: Die Kantenliste eines gerichteten Graphen

Um bei einem ungerichteten Graphen festzustellen, ob es eine Kante von i nach j gibt, muß die ganze Kantenliste durchsucht werden. Bei gerichteten Graphen ist es vorteilhaft, die Kanten lexikographisch zu sortieren. In diesem Falle kann man mittels binärer Suche schneller feststellen, ob es eine Kante von i nach j gibt.

Für ungerichtete Graphen kann die Kantenliste erweitert werden, so daß jede Kante zweimal vertreten ist, einmal in jeder Richtung. Mit diesem erhöhten Speicheraufwand läßt sich die Nachbarschaft zweier Ecken ebenfalls mittels binärer Suche feststellen. Für die Bestimmung aller Nachbarn einer Ecke in einem ungerichteten Graphen bzw. für die Bestimmung von Vorgängern und Nachfolgern in gerichteten Graphen gelten ähnliche Aussagen. Für dynamische Anwendungen ist auch hier eine mittels Zeigern realisierte Kantenliste vorzuziehen.

2.4.4 Bewertete Graphen

In vielen praktischen Anwendungen sind die Ecken oder Kanten eines Graphen mit Werten versehen. Solche Graphen werden *ecken-* bzw. *kantenbewertet* genannt. Die Werte kommen dabei aus einer festen Wertemenge W . In den Anwendungen ist W häufig eine Teilmenge von \mathbb{R} oder eine Menge von Wörtern. Abbildung 2.13 zeigt einen kantenbewerteten Graphen mit Wertemenge $W = \mathbb{N}$. Die Bewertungen der Kanten sind fett dargestellt.

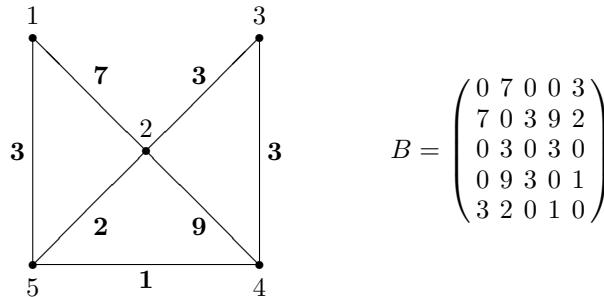


Abbildung 2.13: Ein kantenbewerteter Graph mit seiner bewerteten Adjazenzmatrix

Eine Möglichkeit, bewertete Graphen darzustellen, besteht darin, die Bewertungen in einem zusätzlichen Feld abzuspeichern. Jede der oben angegebenen Darstellungsarten kann so erweitert werden. Bei eckenbewerteten Graphen stehen die Bewertungen in einem Feld der Länge n , dessen Typ sich nach dem Charakter der Wertemenge richtet.

Für kantenbewertete Graphen können einige der oben erwähnten Darstellungsarten direkt erweitert werden. Es kann zum Beispiel die *bewertete Adjazenzmatrix* B gebildet werden. Hierbei ist $B[i, j]$ die Bewertung der Kante von i nach j . Gibt es keine Kante von i nach j , so wird ein Wert, welcher nicht in W liegt, an der Stelle $B[i, j]$ abgelegt. Für numerische Wertemengen kann dies z.B. die Zahl 0 sein, sofern $0 \notin W$. Abbildung 2.13 zeigt die bewertete Adjazenzmatrix des dargestellten Graphen. Bei der Darstellung mittels Adjazenzlisten lässt sich die Liste der Nachbarn erweitern, indem man zu jedem Nachbarn noch zusätzlich die Bewertung der entsprechenden Kante abspeichert.

2.4.5 Implizite Darstellung

In manchen Fällen besitzen Graphen eine spezielle Struktur, die zu einer kompakten Speicherung führt. Dies gilt zum Beispiel für sogenannte Intervallgraphen. Es sei $I = \{I_1, \dots, I_n\}$ eine Menge von reellen Intervallen der Form $I_i = [a_i, b_i]$. Die Menge I bestimmt einen ungerichteten Graphen G_I mit Eckenmenge $E = \{1, \dots, n\}$. Zwei Ecken i, j sind genau dann benachbart, wenn die zugehörigen Intervalle sich überschneiden (d.h. $I_i \cap I_j \neq \emptyset$). Solche Graphen nennt man *Intervallgraphen*. Eine mögliche Darstellung besteht darin, die Intervallgrenzen in einer $2 \times n$ Matrix abzuspeichern. Dazu werden lediglich $2n$ Speicherplätze benötigt. Um festzustellen, ob es eine Kante von i nach j gibt, muß lediglich überprüft werden, ob sich die entsprechenden Intervalle

überschneiden. Um die Nachbarn einer Ecke i zu bestimmen, muß ein Intervall mit allen anderen verglichen werden. Durch den Einsatz von Baumstrukturen, wie sie in Kapitel 3 dargestellt werden, erreicht man eine sehr kompakte Darstellung von Intervallgraphen, welche auch Nachbarschaftsabfragen effizient unterstützt. Die Bestimmung der Nachbarn einer Ecke erfordert dabei nicht die Betrachtung aller Intervalle. Abbildung 2.14 zeigt eine Menge von fünf Intervallen und den zugehörigen Graphen.

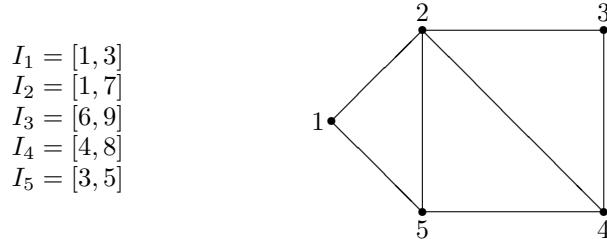


Abbildung 2.14: Ein Intervallgraph

2.5 Darstellung der Algorithmen

Für die Darstellung der Algorithmen wird im folgenden eine Pascal-ähnliche Notation verwendet. Eine Übertragung in eine höhere Programmiersprache ist ohne großen Aufwand durchzuführen. Um die Beschreibungen der Algorithmen unabhängig von einer speziellen Datenstruktur für Graphen zu halten, werden im folgenden einige abstrakte Datentypen für Graphen verwendet.

Graph	für beliebige Graphen;
G-Graph	für gerichtete Graphen;
K-G-Graph	für kreisfreie, gerichtete Graphen;
P-Graph	für planare Graphen;
Netzwerk	für Netzwerke;
0-1-Netzwerk	für Netzwerke, deren Kapazitäten 0 oder 1 sind;
Baum	für Bäume.

Die Datentypen für Graphen unterstützen einige Iteratoren:

```

for jede Ecke i do
for jede Kante k do
for jeden Nachbar j von i do

```

Bei den ersten beiden Iterationen durchläuft die Variable i nacheinander alle Ecken bzw. Kanten des Graphen. Der letzte Iterator steht nur für ungerichtete Graphen zur Verfügung. Hierbei repräsentiert i eine Ecke des Graphen, und die Variable j durchläuft

nacheinander alle Ecken, welche zu i benachbart sind. Für gerichtete Graphen gibt es dafür die Iteratoren:

```
for jeden Vorgänger j von i do
  for jeden Nachfolger j von i do
```

Diese Iteratoren lassen sich leicht für jede der im letzten Abschnitt vorgestellten Datenstrukturen realisieren. Mit n wird immer die Anzahl der Ecken und mit m die Anzahl der Kanten bezeichnet.

Des weiteren werden noch weitere Datenstrukturen wie Liste, Stapel, Warteschlange und Menge verwendet. Für eine Beschreibung der für diese Datenstrukturen zur Verfügung stehenden Operationen und entsprechende Implementierungen sei auf die in Abschnitt 2.10 angegebene Literatur verwiesen. Die Bezeichnung der Operationen ist selbsterklärend. Im folgenden sind beispielhaft vier Stapeloperationen angegeben:

```
S.einfügen(i)
S.enthalten(i)
S.entfernen
S.kopf
```

Hierbei bezeichnet S eine Variable vom Typ **stapel of T** und i ein Objekt vom Typ T . Die erste Operation fügt i in S ein. Die zweite Operation überprüft, ob i in S enthalten ist und gibt **true** oder **false** zurück. Die dritte Operation entfernt das oberste Objekt aus dem Stapel und gibt es zurück. Die letzte Operation gibt eine Kopie des obersten Objektes zurück, ohne es zu entfernen.

Ferner werden noch folgende Anweisungen verwendet:

```
initialisiere f mit 0
exit('Text')
```

Die erste Anweisung wird sowohl zum Initialisieren von einfachen Variablen als auch von Feldern verwendet. Mittels der Anweisung `exit('Text')` wird eine Prozedur oder ein Programm direkt verlassen, und `Text` wird ausgegeben. Die Konstante `MAX` wird bei der Vereinbarung von Feldern verwendet und bezeichnet meistens die maximale Anzahl der zur Verfügung stehenden Speicherplätze für die Ecken eines Graphen.

2.6 Der transitive Abschluß eines Graphen

Viele Programmiersprachen bieten die Möglichkeit, große Programme auf mehrere Dateien zu verteilen. Dadurch erreicht man eine bessere Organisation der Programme und kann z.B. Schnittstellen und Implementierungen in verschiedenen Dateien ablegen. Mit Hilfe des *include-Mechanismus* kann der Inhalt einer Datei in eine andere Datei hineinkopiert werden. Dies nimmt der Compiler bei der Übersetzung vor. Am Anfang einer Datei steht dabei eine Liste von Dateinamen, die in die Datei eingefügt werden sollen.

Diese Dateien können wiederum auf andere Dateien verweisen; d.h. eine Datei kann indirekt in eine andere Datei eingefügt werden. Bei großen Programmen ist es wichtig zu wissen, welche Datei in eine gegebene Datei direkt oder indirekt eingefügt wird. Diese Situation kann leicht durch einen gerichteten Graphen dargestellt werden. Die Dateien bilden die Ecken, und falls eine Datei direkt in eine andere eingefügt wird, wird dies durch eine gerichtete Kante zwischen den Ecken, die der Datei und der eingefügten Datei entsprechen, dargestellt. Somit erhält man einen gerichteten Graphen. Die Anzahl der Ecken entspricht der Anzahl der Dateien, und die Datei D_i wird direkt oder indirekt in die Datei D_j eingefügt, falls es einen Weg von D_j nach D_i gibt. Alle in diesem Graphen von einer Ecke D_i aus erreichbaren Ecken entsprechen genau den in die Datei D_i eingefügten Dateien.

Das obige Beispiel führt zur Definition des *transitiven Abschlusses* eines gerichteten Graphen. Der transitive Abschluß eines gerichteten Graphen G ist ein gerichteter Graph mit gleicher Eckenmenge wie G , in dem es von der Ecke e eine Kante zur Ecke f gibt, falls es in G einen Weg von e nach f gibt, der aus mindestens einer Kante besteht. Abbildung 2.15 zeigt einen gerichteten Graphen und seinen transitiven Abschluß.

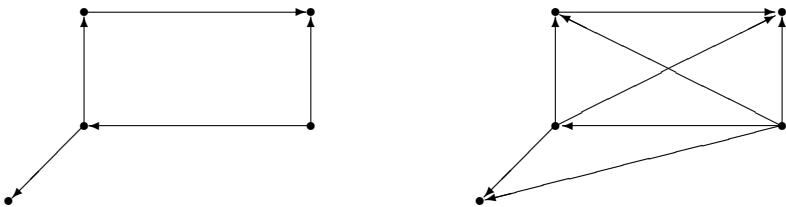


Abbildung 2.15: Ein gerichteter Graph mit seinem transitiven Abschluß

Wie bestimmt man den transitiven Abschluß eines gerichteten Graphen? Zur Vorbereitung eines Verfahrens wird folgendes Lemma benötigt.

Lemma. Es sei G ein gerichteter Graph mit Adjazenzmatrix A . Dann ist der (i, j) -Eintrag von A^s gleich der Anzahl der verschiedenen Kantenzüge mit Anfangsecke i und Endecke j , welche aus s Kanten bestehen.

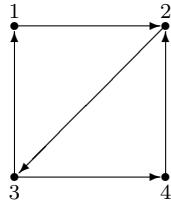
Beweis. Der Beweis erfolgt durch vollständige Induktion nach s . Für $s = 1$ ist die Aussage richtig, denn die Adjazenzmatrix enthält die angegebene Information über Kantenzüge der Länge 1. Die Aussage sei nun für $s > 1$ richtig. Nun sei \tilde{a}_{il} der (i, l) -Eintrag von A^s und a_{lj} der (l, j) -Eintrag von A . Nach Induktionsvoraussetzung ist \tilde{a}_{il} die Anzahl der Kantenzüge mit Anfangsecke i und Endecke l , welche aus s Kanten bestehen. Dann ist $\tilde{a}_{il}a_{lj}$ die Anzahl der Kantenzüge mit Anfangsecke i und Endecke j , welche aus $s + 1$ Kanten bestehen und deren vorletzte Ecke die Nummer l trägt. Da

der (i, j) -Eintrag von A^{s+1} gleich

$$\sum_{l=1}^n \tilde{a}_{il} a_{lj}$$

ist, ist die Behauptung somit bewiesen. ■

Abbildung 2.16 zeigt einen gerichteten Graphen und die ersten 3 Potenzen der zugehörigen Adjazenzmatrix.



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

$$A^2 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad A^3 = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix} \quad E = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Abbildung 2.16: Ein gerichteter Graph und die Potenzen seiner Adjazenzmatrix

Mit Hilfe der Potenzen der Adjazenzmatrix A läßt sich der transitive Abschluß leicht bestimmen. Gibt es in einem Graphen mit n Ecken einen Weg zwischen zwei Ecken, so gibt es auch zwischen diesen beiden Ecken einen Weg, der aus maximal $n - 1$ Kanten besteht. Ist also die Ecke j von der Ecke i erreichbar, so gibt es eine Zahl s mit

$$1 \leq s \leq n - 1,$$

so daß der (i, j) -Eintrag in A^s ungleich 0 ist. Aus der Matrix

$$S = \sum_{s=1}^{n-1} A^s$$

kann nun die Adjazenzmatrix E des transitiven Abschluß gebildet werden:

$$e_{ij} = \begin{cases} 1 & \text{falls } s_{ij} \neq 0; \\ 0 & \text{falls } s_{ij} = 0; \end{cases}$$

Abbildung 2.16 zeigt auch die Matrix E des abgebildeten Graphen. Die Matrix E heißt *Erreichbarkeitsmatrix*. Ist ein Diagonaleintrag von E von 0 verschieden, so bedeutet dies, daß die entsprechende Ecke auf einem geschlossenen Weg liegt; d.h. der transitive Abschluß ist genau dann schlicht, wenn der Graph keinen geschlossenen Weg enthält.

Dieses Verfahren ist konzeptionell sehr einfach, aber aufwendig, da die Berechnung der Potenzen von A rechenintensiv ist. Im folgenden wird deshalb ein zweites Verfahren vorgestellt.

Die Adjazenzmatrix des gerichteten Graphen G wird in n Schritten in die Adjazenzmatrix des transitiven Abschluß überführt. Die zugehörigen Graphen werden mit G_0, G_1, \dots, G_n bezeichnet, wobei $G_0 = G$ ist. Der Übergang von G_l nach G_{l+1} erfolgt, indem für jedes Paar i, j von Ecken, für die es in G_l noch keine Kante gibt, eine Kante von i nach j hinzugefügt wird, falls es in G_l Kanten von i nach l und von l nach j gibt; d.h. es müssen in jedem Schritt alle Paare von Ecken überprüft werden.

Wieso produziert dieses Verfahren den transitiven Abschluß? Der Korrektheitsbeweis wird mit Hilfe des folgenden Lemmas erbracht. Um vollständige Induktion anzuwenden, wird eine etwas stärkere Aussage bewiesen.

Lemma. Für $l = 0, 1, \dots, n$ gilt: In G_l gibt es genau dann eine Kante von i nach j , wenn es in G einen Weg von i nach j gibt, welcher nur Ecken aus der Menge $\{1, \dots, l\}$ verwendet.

Beweis. Der Beweis erfolgt durch vollständige Induktion nach l . Für $l = 0$ ist die Aussage richtig. Die Aussage sei nun für $l > 0$ richtig. Gibt es in G_{l+1} eine Kante von i nach j , so gibt es nach Konstruktion auch in G einen Weg von i nach j , der nur Ecken aus der Menge $\{1, \dots, l+1\}$ verwendet. Umgekehrt ist noch zu beweisen, daß es in G_{l+1} eine Kante von i nach j gibt, falls es in G einen Weg W gibt, welcher nur Ecken aus $\{1, \dots, l+1\}$ verwendet. Verwendet W nur Ecken aus $\{1, \dots, l\}$, so ist die Aussage nach Induktionsvoraussetzung richtig. Verwendet W die Ecke $l+1$, so gibt es auch einen einfachen Weg W' mit dieser Eigenschaft. Es sei W_1 der Teilweg von i nach $l+1$ und W_2 der von $l+1$ nach j . Nach Induktionsvoraussetzung gibt es somit in G_l Kanten von i nach $l+1$ und von $l+1$ nach j . Somit gibt es nach Konstruktion in G_{l+1} eine Kante von i nach j . ■

Die Korrektheit des Verfahrens folgt nun aus diesem Lemma, angewendet für den Fall $l = n$. Wie kann dieses Verfahren in ein Programm umgesetzt werden? Dazu muß erst eine geeignete Darstellung des Graphen gefunden werden. Die Datenstruktur sollte Abfragen nach dem Vorhandensein von Kanten effizient unterstützen, da diese sehr häufig vorkommen. Da die Graphen G_l , welche in den Zwischenschritten auftreten, nicht weiter benötigt werden, sollte die Datenstruktur diesen Übergang erlauben, ohne allzuviel Speicher zu verbrauchen. Die Adjazenzmatrix erfüllt diese Anforderungen. Abbildung 2.17 zeigt die Prozedur `transAbschluss`, welche die Adjazenzmatrix A eines ungerichteten Graphen in die des transitiven Abschlusses überführt. Der Algorithmus wurde unabhängig von S.A. Warshall und von B. Roy entwickelt.

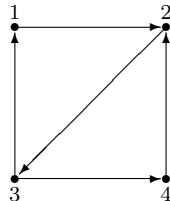
Nach der Initialisierung der Matrix E mit der Adjazenzmatrix von G folgen drei Laufschleifen. Die Äußere realisiert die n Schritte, und mit Hilfe der beiden Inneren werden jeweils alle Einträge dahingehend überprüft, ob eine neue Kante hinzukommt. Abbildung 2.18 zeigt eine Anwendung der Prozedur `transAbschluss` auf den Graphen aus Abbildung 2.16. Die in den entsprechenden Iterationen geänderten Werte sind dabei eingerahmt.

```

var E : array[1..max,1..max] of Integer;
procedure transAbschluss(G : G-Graph);
var
  i,j : Integer;
begin
  Initialisiere E mit A(G);
  for l := 1 to n do
    for i := 1 to n do
      for j := 1 to n do
        if E[i,l] = 1 and E[l,j] = 1 then
          E[i,j] := 1;
end

```

Abbildung 2.17: Die Prozedur `transAbschluss`



$$A(G) = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \quad E_1 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & \boxed{1} & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

$$E_2 = \begin{pmatrix} 0 & 1 & \boxed{1} & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & \boxed{1} & 1 \\ 0 & 1 & \boxed{1} & 0 \end{pmatrix} \quad E_3 = \begin{pmatrix} \boxed{1} & 1 & 1 & \boxed{1} \\ \boxed{1} & \boxed{1} & 1 & \boxed{1} \\ 1 & 1 & 1 & 1 \\ \boxed{1} & 1 & 1 & \boxed{1} \end{pmatrix} \quad E_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Abbildung 2.18: Eine Anwendung der Prozedur `transAbschluss`

Die Adjazenzmatrizen der Graphen G_1, \dots, G_4 sind dabei mit E_1, \dots, E_4 bezeichnet. Von 0 verschiedene Diagonaleinträge zeigen an, daß die entsprechenden Ecken auf geschlossenen Wegen liegen. Somit kann mit diesem Verfahren auch die Existenz geschlossener Wege überprüft werden. In Kapitel 4 werden für dieses Problem einfachere Verfahren vorgestellt.

Welches der beiden diskutierten Verfahren ist vorzuziehen? Dazu könnte man beide Verfahren implementieren und beide auf die gleichen Graphen anwenden; dabei müßte man den Verbrauch an Speicherplatz und CPU-Zeit messen und vergleichen. Im nächsten Abschnitt werden Hilfsmittel zur Verfügung gestellt, um Algorithmen bezüglich Speicherplatz- und Zeitverbrauch zu vergleichen, ohne diese zu realisieren.

2.7 Vergleichskriterien für Algorithmen

Zur Lösung eines Problems stehen häufig mehrere Algorithmen zur Verfügung, und es stellt sich die Frage, nach welchen Kriterien die Auswahl erfolgen soll. Ähnlich sieht es bei der Entwicklung eines Algorithmus für ein Problem aus: Welche Aspekte haben die größte Bedeutung?

Vom Gesichtspunkt der Umsetzung des Algorithmus in ein ablauffähiges Programm ist ein leicht zu verstehender Algorithmus, der einfache Datentypen verwendet, vorzuziehen. Dies erleichtert die Verifikation des Programms und auch die spätere Wartung. Zu beachten ist auch die Verwendung schon existierender Software. Der Anwender eines Algorithmus ist vor allem an der Effizienz interessiert; dies beinhaltet neben dem zeitlichen Aspekt auch den beanspruchten Speicherplatz. Die Entscheidung wird wesentlich davon geprägt, wie oft das zu schreibende Programm ausgeführt werden soll. Falls es sich um ein Programm handelt, welches sehr häufig benutzt wird, lohnt sich der Aufwand, einen komplizierten Algorithmus zu realisieren, wenn dies zu Laufzeitverbesserungen oder Speicherplatzersparnis führt. Andernfalls kann der erzielte Vorteil durch den höheren Entwicklungsaufwand zunichte gemacht werden. Die Entscheidung muß also situationsabhängig erfolgen.

Unter einem Algorithmus verstehen wir hier eine Verarbeitungsvorschrift zur Lösung eines Problems. Diese Verarbeitungsvorschrift kann z.B. durch ein Programm in einer Programmiersprache dargestellt sein. Im folgenden wird deshalb nicht mehr zwischen einem Algorithmus und dem zugehörigen Programm unterschieden.

Um ein Maß für die Effizienz eines Algorithmus zu bekommen, welches unabhängig von einem speziellen Computer ist, wurde die *Komplexitätstheorie* entwickelt. Sie stellt Modelle zur Verfügung, um a priori Aussagen über Laufzeit und Speicheraufwand eines Algorithmus machen zu können. Man ist damit in der Lage, Algorithmen innerhalb dieses Modells zu vergleichen, ohne diese konkret zu realisieren.

Ein ideales Komplexitätsmaß für einen Algorithmus wäre eine Funktion, die jeder Eingabe die Zeit zuordnet, die der Algorithmus zur Berechnung der Ausgabe benötigt. Damit würde der Vergleich von Algorithmen auf den Vergleich der entsprechenden Funktionen reduziert. Um Aussagen zu machen, die unabhängig von der verwendeten Hardware und dem verwendeten Compiler sind, werden keine konkreten Zeitangaben gemacht, sondern es wird die Anzahl von notwendigen Rechenschritten angegeben. Unter einem Rechenschritt versteht man die üblichen arithmetischen Grundoperationen wie Speicherzugriffe etc., von denen man annimmt, daß sie auf einer gegebenen Rechenanlage die gleiche Zeit brauchen. Dadurch wird der Vergleich zumindest theoretisch unabhängig von der verwendeten Hard- und Software. Man muß natürlich beachten, daß in der Realität die einzelnen Operationen auf einer Rechenanlage unterschiedliche Zeiten benötigen.

Um die Eingabe einfacher zu charakterisieren, beschränkt man sich darauf, die Länge der Eingabe anzugeben. Diese ergibt sich häufig aus der Zahl der Zeichen, die die Eingabe umfaßt. Wie schon bei der Zeitangabe wird auch hier keine quantitative Aussage über die Länge eines Zeichens gemacht, sie muß lediglich konstant sein. Das heißt sowohl bei der Eingabe als auch bei dem Komplexitätsmaß bezieht man sich auf relative Größen und kommt dadurch zu Aussagen wie: Bei einer Eingabe mit einer Länge proportional

zu n ist die Laufzeit des Algorithmus proportional zu n^2 . Mit anderen Worten: Wird die Eingabe verdoppelt, so vervierfacht sich die Laufzeit.

In den folgenden Algorithmen wird die Eingabe immer ein Graph sein; die Länge der Eingabe wird hierbei entweder durch die Anzahl der Ecken bzw. Kanten oder von beiden angegeben. Bei bewerteten Graphen müssen auch die Bewertungen miteinbezogen werden. Die Laufzeit bezieht sich dann auf Graphen mit der angegebenen Ecken- bzw. Kantenzahl und ist unabhängig von der Struktur der Graphen. Man unterscheidet deswegen die Komplexität im schlimmsten Fall (im folgenden *worst case Komplexität* genannt) und im Mittel (im folgenden *average case Komplexität* genannt). Die worst case Komplexität bezeichnet die maximale Laufzeit bzw. den maximalen Speicherplatz für eine beliebige Eingabe der Länge n . Somit wird dadurch die Laufzeit bzw. der Speicherplatz für die ungünstigste Eingabe der Länge n beschrieben. Falls der Algorithmus nicht für alle Eingaben der Länge n gleich viel Zeit benötigt, ist der Maximalwert ausschlaggebend.

Die average case Komplexität mittelt über alle Eingaben der Länge n . Diese Größe ist häufig viel aussagekräftiger, da das Maximum meistens für „entartete“ Eingaben angenommen wird, die in der Praxis nur selten oder nie vorkommen. Dies sei an dem Beispiel des internen Sortierverfahrens *Quicksort* dargestellt. Um n Objekte zu sortieren, sind dabei im worst case n^2 Schritte notwendig. Im average case sind dagegen nur $n \log_2 n$ Schritte notwendig. Quicksort gilt aufgrund von vielen Tests als schnellstes internes Sortierverfahren. Leider ist es in der Praxis häufig sehr schwer, die average case Komplexität zu berechnen, da die Analyse in diesem Falle mathematisch oft sehr anspruchsvoll wird. Für viele Algorithmen ist die average case Komplexität gleich der worst case Komplexität. Es gibt viele Algorithmen, für die es bis heute nicht gelungen ist, die average case Komplexität zu bestimmen.

Platzbedarf und Laufzeit werden in der Regel als Funktionen $f : \mathbb{N} \rightarrow \mathbb{R}^+$ angegeben. Um die Bestimmung der Funktionen zu erleichtern, verzichtet man auf die Angabe des genauen Wertes von $f(n)$ und gibt nur den qualitativen Verlauf, d.h. die Größenordnung von f an. Dies geschieht mit dem *Landauschen Symbol O*.

Sind f und g Funktionen von \mathbb{N} nach \mathbb{R}^+ , so sagt man, daß f die *Ordnung* von g hat, wenn es eine Konstante $c > 0$ und $n_0 \in \mathbb{N}$ gibt, so daß

$$f(n) \leq cg(n) \text{ für alle } n \geq n_0.$$

In diesem Falle schreibt man auch $f(n) = O(g(n))$. Ist die Laufzeit $f(n)$ eines Algorithmus gleich $O(n^2)$, so bedeutet dies, daß es Zahlen c und n_0 gibt, so daß $f(n) \leq cn^2$ für alle $n \in \mathbb{N}$ mit $n \geq n_0$. Ist $f(n) = (n+2)^2 \log n$, so ist $f(n) \leq 2n^2 \log n$ für $n \geq 5$, und somit ist die Laufzeit $O(n^2 \log n)$. Natürlich ist $f(n)$ in diesem Fall auch von der Ordnung $O(n^3)$, aber dies ist eine schwächere Aussage, und man versucht natürlich immer, die stärkste Aussage zu machen.

Damit hat man ein Hilfsmittel zur Hand, mit dem man Algorithmen für ein und dasselbe Problem gut vergleichen kann. So hat z.B. der naive Algorithmus zum Sortieren von n Zahlen, bei dem sukzessive immer die kleinste Zahl ausgewählt wird, eine Laufzeit von $O(n^2)$. Allerdings werden wir im Kapitel 3 sehen, daß es für dieses Problem auch Algorithmen mit einer worst case Laufzeit von $O(n \log n)$ gibt.

Durch diese qualitative Bewertung sind nur Aussagen folgender Form möglich: Für große Eingaben ist dieser Algorithmus schneller bzw. platzsparender. Es kann sein, daß ein Algorithmus mit Laufzeit $O(n^3)$ für kleine n einem Algorithmus mit Laufzeit $O(n^2)$ überlegen ist. In einem konkreten Problem muß man diesen Aspekt natürlich berücksichtigen. Mit Hilfe der Laufzeitkomplexität kann man auch abschätzen, für welche Eingaben die Rechenanlage noch in der Lage ist, das Problem in vertretbarer Zeit zu lösen. Für Algorithmen mit exponentieller Laufzeit, d.h. $O(a^n)$ für $a > 1$, wird man sehr schnell an Grenzen kommen. Daran ändert auch ein viel schnellerer Computer nichts. Aus diesem Grund sind Algorithmen mit exponentieller Laufzeit praktisch nicht sinnvoll einsetzbar. Praktisch interessant sind nur Algorithmen mit polynomialem Aufwand, d.h. $O(p(n))$, wobei $p(n)$ ein Polynom ist; solche Algorithmen werden im folgenden *effizient* genannt.

Das Konzept der worst case Komplexität wird nun an einem Beispiel erläutert. Für ein gegebenes Problem stehen fünf verschiedene Algorithmen A_1 bis A_5 zur Verfügung. In der folgenden Tabelle sind die Laufzeiten der fünf Algorithmen für eine Eingabe der Länge n angegeben. Ferner sind auch die worst case Komplexitäten angegeben.

	A_1	A_2	A_3	A_4	A_5
benötigte Zeiteinheiten	2^n	$n^3/3$	$31n \log_2 n$	$2n^2 + 2n$	$90n$
worst case Komplexität	$O(2^n)$	$O(n^3)$	$O(n \log n)$	$O(n^2)$	$O(n)$

Abbildung 2.19 zeigt die Laufzeitkurven der fünf Algorithmen. Obwohl die worst case Komplexität des Algorithmus A_5 am geringsten ist, sind für kleine Werte von n andere Algorithmen vorzuziehen.

Erst wenn die Länge der Eingabe über 44 liegt, ist Algorithmus A_5 am schnellsten. Die folgende Tabelle zeigt für die 5 Algorithmen die maximale Länge einer Eingabe, welche diese innerhalb eines vorgegebenen Zeitlimits verarbeiten können.

Zeitlimit	A_1	A_2	A_3	A_4	A_5
100	6	6	2	6	1
1000	9	14	9	22	11
10000	13	31	55	70	111
100000	16	66	376	223	1111

Bei einem Limit von 100 Zeiteinheiten ist Algorithmus A_1 überlegen. Er kann in dieser Zeit eine Eingabe der Länge 6 verarbeiten und benötigt dazu 64 Zeiteinheiten. Bei einem Limit von 1000 Zeiteinheiten verarbeitet Algorithmus A_4 die längste Eingabe. Der Algorithmus A_5 ist ab einem Limit von 3960 Zeiteinheiten den anderen Algorithmen überlegen. Erst bei einem Limit von 100000 Zeiteinheiten ist Algorithmus A_3 dem Algorithmus A_4 überlegen.

Für Algorithmus A_5 ergibt sich bei einer Verzehnfachung des Zeitlimits (unabhängig vom absoluten Wert des Zeitlimits) immer ein Anstieg der längsten Eingabe von 900%.

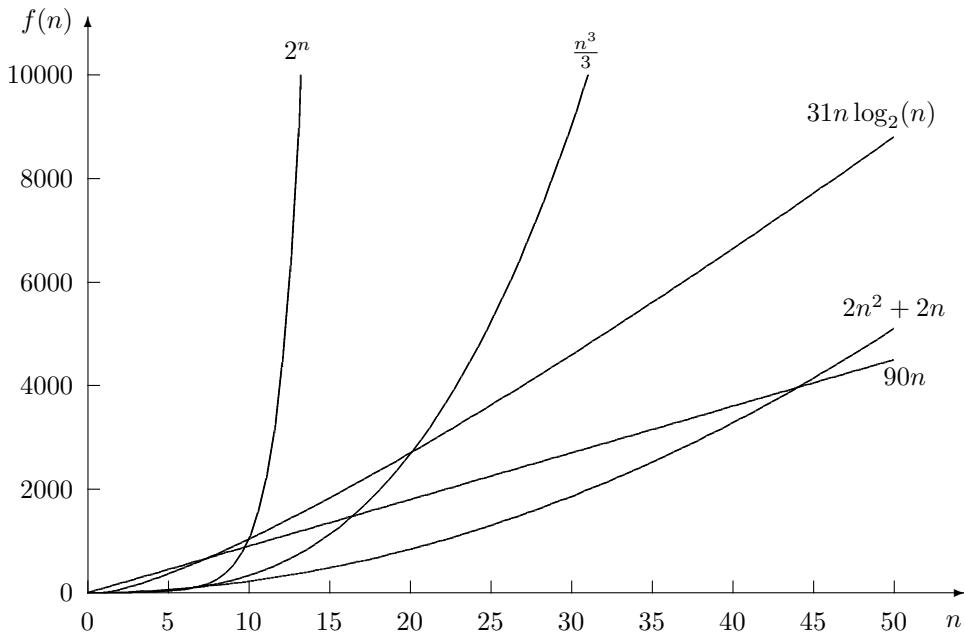


Abbildung 2.19: Die Laufzeit von 5 Algorithmen

Bei Algorithmus A_1 ist dieser Anstieg abhängig vom Zeitlimit. Bei einer Verzehnfachung des Zeitlimits von 10000 auf 100000 ergibt sich ein Anstieg der längsten Eingabe von 23% bei Algorithmus A_1 . Mit steigendem Zeitlimit geht dieser Anstieg gegen 0.

Um die Komplexität eines Algorithmus zu bestimmen, ist es notwendig, die Komplexität der verwendeten Grundoperationen zu kennen. Für Graphen ist dabei die gewählte Darstellungsart von großer Bedeutung. In Abbildung 2.20 sind Speicheraufwand und zeitlicher Aufwand für die Grundoperationen der in Abschnitt 2.4 vorgestellten Datenstrukturen zusammengestellt. Mit Hilfe dieser Angaben lässt sich die Komplexität eines Graphalgorithms bestimmen, indem man zählt, wie oft die einzelnen Operationen durchgeführt werden.

Welche Komplexität haben die im letzten Abschnitt angegebenen Algorithmen zur Berechnung der Erreichbarkeitsmatrix? Der erste Algorithmus bestimmt die Potenzen A^s der Adjazenzmatrix A für $s = 1, \dots, n - 1$ und addiert diese dann. Das normale Verfahren, zwei $n \times n$ -Matrizen zu multiplizieren, hat einen Aufwand von $O(n^3)$ (für jeden Eintrag sind n Multiplikationen und $n - 1$ Additionen notwendig). Somit ergibt sich ein Gesamtaufwand von $O(n^4)$. Es gibt aber effizientere Verfahren zur Multiplikation von Matrizen. Der Aufwand beträgt dabei $O(n^\alpha)$, wobei $\alpha \approx 2.4$ ist. Diese erfordern aber einen hohen Implementierungsaufwand. Mit ihnen ergibt sich ein günstigerer Aufwand für die Bestimmung des transitiven Abschlusses. Es gibt Algorithmen, deren Aufwand unter $O(n^\alpha \log n)$ liegt (vergleichen Sie Aufgabe 21).

Darstellungsart	Speicherplatzbedarf	Feststellung ob es eine Kante von i nach j gibt	Bestimmung der Nachbarn (Nachfolger) von i	Bestimmung der Vorgänger von i (nur für gerichtete Graphen)
Adjazenzmatrix	$O(n^2)$	$O(1)$	$O(n)$	$O(n)$
Adjazenzliste	$O(n+m)$	$O(g(i))$ bzw. $O(g^+(i))$	$O(g(i))$ bzw. $O(g^+(i))$	$O(n+m)$
Adjazenzliste mit invertierter Adjazenzliste (nur gerichtete Graphen)	$O(n+m)$	$O(g^+(i))$	$O(g^+(i))$	$O(g^-(i))$
Kantenliste	$O(m)$	$O(m)$	$O(m)$	$O(m)$
doppelte, lexikographisch sortierte Kantenliste (nur ungerichtete Graphen)	$O(m)$	$O(\log(m))$	$O(\log(m) + g(i))$	—
lexikographisch sortierte Kantenliste mit invertierter Darstellung (nur gerichtete Graphen)	$O(m)$	$O(\log(m))$	$O(\log(m) + g^+(i))$	$O(\log(m) + g^-(i))$

Abbildung 2.20: Übersicht über die Komplexität der Darstellungsarten

Die Analyse des zweiten Algorithmus ist einfach: Innerhalb der Prozedur `transAbschluss` sind drei ineinander geschachtelte Laufschleifen der Länge n . Somit ist der Aufwand gleich $O(n^3)$. Hierbei sind für jeden Graphen mit n Ecken mindestens n^3 Schritte notwendig. Für die beiden Algorithmen stimmen worst case und average case Komplexität überein.

Die eigentliche Arbeit erfolgt in der `if`-Anweisung

```
if E[i,1] = 1 and E[1,j] = 1 then E[i,j] := 1
```

Hierbei wird der Eintrag $E[i,j]$ geändert, falls $E[i,1]$ und $E[1,j]$ beide gleich 1 sind. Diese Anweisung wird n^3 -mal durchgeführt. Diese Anzahl kann gesenkt werden, wenn man beachtet, daß die erste Bedingung unabhängig von j ist. Das heißt, falls $E[i,1]$ gleich 0 ist, wird für keinen Wert von j der Eintrag $E[i,j]$ geändert. Aus diesem Grund wird diese Bedingung vor der letzten Laufschleife getestet. Abbildung 2.21 zeigt die so geänderte Prozedur `transAbschluss`.

```

procedure transAbschluss(G : G-Graph);
var
    l,i,j : Integer;
begin
    Initialisiere E mit A(G);
    for l := 1 to n do
        for i := 1 to n do
            if E[i,l] = 1 then
                for j := 1 to n do
                    if E[l,j] = 1 then
                        E[i,j] := 1;
end

```

Abbildung 2.21: Eine verbesserte Version der Prozedur transAbschluss

Die worst case Komplexität dieser verbesserten Version ist immer noch $O(n^3)$. Dies folgt aus der Anwendung dieser Prozedur auf einen gerichteten Graphen mit $O(n^2)$ Kanten. Die eigentliche Laufzeit wird sich aber in vielen Fällen wesentlich verringern. Dies zeigt die beschränkte Aussagekraft der worst case Komplexität. Nichts desto trotz ist die worst case Komplexität ein wichtiges Hilfsmittel zur Analyse von Algorithmen.

Eine für praktische Anwendungen interessante Beschreibung der worst case Komplexität verwendet Parameter, die neben der Länge der Eingabe auch die Länge der Ausgabe des Algorithmus verwenden. Im vorliegenden Beispiel kann man die Ausgabe durch die Anzahl m_{Ab} der Kanten des transitiven Abschlusses beschreiben. Die innerste Laufschleife wird maximal m_{Ab} -mal durchlaufen. Somit ergibt sich eine Gesamlaufzeit von $O(n^2 + nm_{Ab})$. Ist die Größenordnung von m_{Ab} gleich $O(n)$, so ergibt sich eine Laufzeit $O(n^2)$. Algorithmen, deren Laufzeit auch von der Länge der Ausgabe explizit abhängen, werden in der Literatur als *ausgabesensitive Algorithmen* bezeichnet.

2.8 Greedy-Algorithmen

Eine wichtige Klasse von Algorithmen zur Lösung von Optimierungsproblemen sind sogenannte *Greedy-Algorithmen*. Sie versuchen eine global optimale Lösung zu finden, indem sie iterativ lokal optimale Lösungen erweitern. Bei der Erweiterung einer partiellen Lösung wird nur die aktuelle Situation betrachtet, d.h. es erfolgt keine Analyse der globalen Situation. Ferner werden einmal getroffene Entscheidungen zur Erweiterung einer partiellen Lösung nicht wieder revidiert (daher auch der Name *gierig*). Der große Vorteil von Greedy-Algorithmen ist der, daß sie einfach zu entwerfen und effizient zu implementieren sind. Beim Entwurf eines Algorithmus für ein Optimierungsproblem wird man zunächst den Einsatz eines Greedy-Algorithmus prüfen. Der große Nachteil besteht darin, daß sie, von einigen Ausnahmen abgesehen, oft nur ein lokales Optimum und nicht das eigentlich gewünschte globale Optimum finden. Wie in Kapitel 9 gezeigt werden wird, ist dies aber in vielen Fällen das Beste, was man in annehmbarer Ausführungszeit erwarten kann.

Der Entwurf und die Implementierung eines Greedy-Algorithmus wird im folgenden am Beispiel des Optimierungsproblems der maximalen Clique in ungerichteten Graphen exemplarisch erläutert. Eine Teilmenge C der Eckenmenge eines ungerichteten Graphen G heißt *Clique*, falls der von C induzierte Untergraph von G vollständig ist. Beim Optimierungsproblem der maximalen Clique wird die Clique von G mit den meisten Ecken gesucht. Abbildung 2.22 zeigt einen ungerichteten Graphen mit einer maximalen Clique der Größe 5.

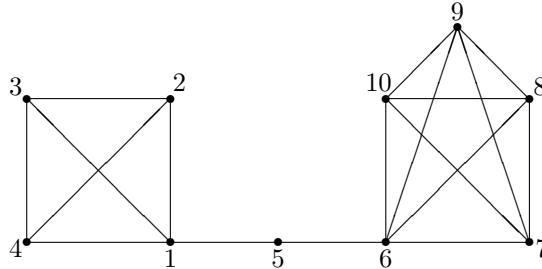


Abbildung 2.22: Ein ungerichteter Graph mit einer maximalen Clique des Größe 5

Der Greedy-Algorithmus startet mit einer leeren Liste `clique` und betrachtet alle Ecken des Graphen der Reihe nach. Ist eine Ecke zu allen Ecken in `clique` benachbart, so wird die Ecke in `clique` eingefügt, andernfalls wird sie ignoriert. Am Ende dieser Iteration enthält `clique` eine Clique. Abbildung 2.23 zeigt eine einfache Umsetzung dieses Verfahrens. Jedes Paar von Ecken wird höchstens einmal auf Nachbarschaft getestet. Kann dieser Test in konstanter Zeit durchgeführt werden, so ist der Aufwand des Algorithmus $O(m + n)$. Wie aus Abbildung 2.20 ersichtlich, ist dies aber nur bei der Darstellungsart Adjazenzmatrix gegeben, diese hat aber einen Speicherbedarf von $O(n^2)$. Eine Realisierung mit linearem Aufwand basiert auf Adjazenzlisten, bei denen die Ecken in den Nachbarschaftslisten nach aufsteigenden Eckennummern sortiert sind. Der Algorithmus durchläuft die Ecken ebenfalls nach aufsteigenden Eckennummern. Für die Auswertung der `if`-Anweisung muß getestet werden, ob die sortierte Liste `clique` in der sortierten Nachbarschaftsliste von i enthalten ist. Mittels zweier Positionszeiger in den beiden Listen kann dies mit Aufwand $O(g(i))$ implementiert werden, über alle Ecken summiert ergibt sich der Aufwand $O(m + n)$.

Betrachten wir nun die Anwendung des Greedy-Algorithmus auf den in Abbildung 2.22 dargestellten Graphen. Werden die Ecken in der angegebenen Reihenfolge bearbeitet, so wird eine Clique der Größe 4 gefunden. Hingegen erhält man die optimale Lösung, wenn die Ecken in absteigender Reihenfolge betrachtet werden. Das schlechteste Ergebnis erhält man, wenn mit Ecke 5 begonnen wird. Dieses Beispiel zeigt, daß das Ergebnis des Greedy-Algorithmus stark von der Wahl der Erweiterung der partiellen Lösung abhängt. In dem betrachteten Beispiel ist die Reihenfolge der Ecken ausschlaggebend. Aus diesem Grund wird die Reihenfolge häufig nicht zufällig gewählt. Oftmals ergibt sich aus der Problemstellung eine vermeintlich günstige Reihenfolge. Da in einer Clique der Größe c der Eckengrad jeder Ecke gleich $c - 1$ ist, bietet es sich bei dem vorliegenden Problem an, die Ecken in der Reihenfolge absteigender Eckengrade zu betrachten. Man

```

function greedy-clique(G : Graph) : Liste of Integer;
var
    i : Integer;
    clique : Liste of Integer;
begin
    clique := ∅;
    for jede Ecke i do
        if alle Ecken aus clique sind zu i benachbart then
            clique.append(i);
        greedy-clique := clique;
    end

```

Abbildung 2.23: Greedy-Algorithmus zur Bestimmung einer maximalen Clique

überzeugt sich schnell, daß auch dies noch keine Garantie für das Auffinden einer optimalen Lösung ist (vergleichen Sie hierzu Aufgabe 7 in Kapitel 9). Auch ohne an dieser Stelle einen formalen Beweis zu erbringen, wird man allerdings von der zweiten Variante bessere Ergebnisse erwarten. Der Greedy-Algorithmus kann nun folgendermaßen formuliert werden:

```

clique := ∅;
while es gibt noch eine Ecke in G do begin
    Füge die Ecke e mit maximalem Eckengrad in G in clique ein;
    Entferne e und alle nicht zu e benachbarten Ecken aus G;
end

```

Man beachte, daß der Eckengrad jeweils im Restgraph bestimmt wird. Eine naive Implementierung kann leicht zu einem Aufwand von $O(n^2)$ führen (beispielsweise bei einer unvorsichtigen Bestimmung der Ecke mit dem jeweils maximalen Eckengrad). Um diese Variante noch mit linearen Aufwand realisieren zu können, müssen einige spezielle Datenstrukturen verwendet werden:

1. EckengradListe

Eine doppelt verkettete Liste mit Zeigern auf den Kopf von Listen vom Typ **EckenListe**, absteigend sortiert nach Eckengraden

2. EckenListe

Eine doppelt verkettete Liste mit Elementen vom Typ **Ecke**, die Ecken in einer **EckenListe** haben alle den gleichen Eckengrad

3. Ecken

Ein Feld mit Zeigern auf Elemente innerhalb von **EckenListen**, jeweils auf die zugehörende Ecke

Weiterhin gibt es noch eine Variable **maxEckengrad**, sie zeigt jeweils auf den Kopf der ersten Liste in **EckengradListe**. Die zur Clique gehörenden Ecken können beispielsweise in einem einfachen Feld verwaltet werden.

In einem ersten Schritt werden diese Datenstrukturen initialisiert. Ausgehend von der Adjazenzliste wird der Eckengrad jeder Ecke bestimmt und in einem linearen Feld der Länge n wird festgehalten, welcher Eckengrad existiert. Dann wird das Feld **EckengradListe** angelegt, für jeden auftretenden Eckengrad wird eine leere Liste vom Typ **EckenListe** erzeugt. Diese Listen werden in absteigender Reihenfolge gespeichert und die Variable **maxEckengrad** zeigt auf die erste Liste. In einem weiteren Feld werden Zeiger auf diese Listen gespeichert, d.h. die i -te Komponente enthält einen Zeiger auf die **Eckenliste** für den Eckengrad i (sofern dieser vorkommt). Nun wird für jede Ecke eine Datenstruktur **Ecke** erzeugt und gemäß Eckengrad an die entsprechende Liste in **EckengradListe** angehängt. Ein Zeiger auf diese Struktur wird in das Feld **Ecken** eingefügt. Mit den zu Verfügung stehenden Strukturen erfordert dies pro Ecke konstanten Aufwand. Die Initialisierung erfordert insgesamt den Aufwand $O(n + m)$.

Die Ecke mit maximalem Eckengrad kann jederzeit mittels der Variablen **maxEckengrad** in konstanter Zeit bestimmt werden. Mit welchem Aufwand kann eine Ecke i aus dem Graphen entfernt werden? Über das Feld **Ecken** erhält man einen Zeiger in die zugehörige Liste **EckenListe**. Da es sich um eine doppelt verkettete Liste handelt, kann die Ecke mit konstantem Aufwand entfernt werden. Wird die entsprechende **EckenListe** leer, so kann diese Liste ebenfalls in konstanter Zeit aus **EckengradListe** entfernt werden. Eventuell muß auch die Variable **maxEckengrad** aktualisiert werden. Damit die Datenstruktur in einem konsistenten Zustand bleibt, müssen noch die Eckengrade der Nachbarn der entfernten Ecke um eins erniedrigt werden. Pro Nachbar erfordern die Veränderungen in den doppelt verketteten Listen einen konstanten Aufwand. Dazu beachte man, daß jede benachbarte Ecke in konstanter Zeit aufgefunden, aus der Eckenliste entfernt und in die nächst niedrige Liste wieder eingefügt werden kann. Gegebenenfalls muß diese Liste erst erzeugt werden, auch die Variable **maxEckengrad** muß eventuell aktualisiert werden. Das Entfernen der Ecke i kann insgesamt mit Aufwand $O(g(i))$ durchgeführt werden.

Bei der Entfernung der nicht benachbarten Ecken wird ebenso vorgegangen. Da jede Ecke einmal entfernt wird, ist der Gesamtaufwand $O(n + m)$. Man beachte, daß dies die Bestimmung der nicht benachbarten Ecken einschließt. Hierzu bezeichne man die ausgewählten Ecken mit maximalem Eckengrad mit e_1, \dots, e_c . Zur Bestimmung der nicht benachbarten Ecken von e_i mit $i > 1$ werden die Nachbarn von e_{i-1} durchlaufen und diejenigen, die nicht zu e_i benachbart sind, werden entfernt. Für $i = 2, \dots, c$ ergibt sich jeweils ein Aufwand von $O(g(e_{i-1}))$. Da die nicht zu e_1 benachbarten Ecken mit Aufwand $O(n)$ bestimmt werden können, ist der Gesamtaufwand $O(n + m)$. Diese Aussagen gelten bei der Verwendung von Adjazenzlisten, bei denen die Ecken in den Nachbarschaftslisten nach aufsteigenden Eckennummern sortiert sind.

Durch den Einsatz von doppelt verketteten Listen kann auch diese Variante des Greedy-Algorithmus in linearer Zeit durchgeführt werden. Dies gelingt aber nicht in allen Fällen, beispielsweise dann nicht, wenn sich die Reihenfolge, in der die Ecken oder Kanten betrachtet werden, aus einem Sortiervorgang ergibt, welcher nicht in linearer Zeit durchzuführen ist. Die in Kapitel 3 vorgestellten Algorithmen zur Bestimmung minimal aufspannender Bäume von Kruskal und Prim sind Greedy-Algorithmen, die nicht in linearer Zeit implementiert werden können. Weitere Beispiele für Greedy-Algorithmen werden in den Kapiteln 9 und 5 behandelt.

2.9 Zufällige Graphen

Um Graphalgorithmen zu testen, benötigt man Graphen. Dazu wurden Sammlungen angelegt; diese enthalten häufig Graphen, für die viele der bekannten Algorithmen ihre maximale Laufzeit benötigen. Die Graphen in einer solchen Sammlung sind in der Regel für ein spezielles Problem zusammengestellt worden. In einigen Kapiteln werden Hinweise auf solche Sammlungen gegeben.

Eine andere Möglichkeit besteht darin, beliebige Graphen mit einer vorgegebenen Eckenzahl zu erzeugen. Um experimentelle Aussagen über die Komplexität im Mittel zu machen, muß man Graphen zufällig aus der Menge aller Graphen mit n Ecken wählen. Eine Möglichkeit besteht darin, daß man mittels eines Zufallsgenerators die Einträge von Adjazenzmatrizen erzeugt. Da man einem Graphen im allgemeinen mehrere Adjazenzmatrizen zuordnen kann (in Abhängigkeit der Numerierung der Ecken), ist dies kein „faires“ Verfahren (unabhängig von der Qualität des Zufallsgenerators); nicht jeder Graph wird dabei mit der gleichen Wahrscheinlichkeit erzeugt. Zum Beispiel gibt es vier verschiedene ungerichtete Graphen mit drei Ecken. Abbildung 2.24 zeigt diese vier Graphen zusammen mit der Anzahl der dazugehörigen verschiedenen Adjazenzmatrizen. Es gibt insgesamt acht verschiedene Adjazenzmatrizen. Unter der Annahme, daß der Zufallsgenerator „fair“ ist, bekommt man den vollständigen Graphen K_3 mit der Wahrscheinlichkeit von $1/8$, den Graphen mit einer Kante und 3 Ecken jedoch mit der Wahrscheinlichkeit von $3/8$.

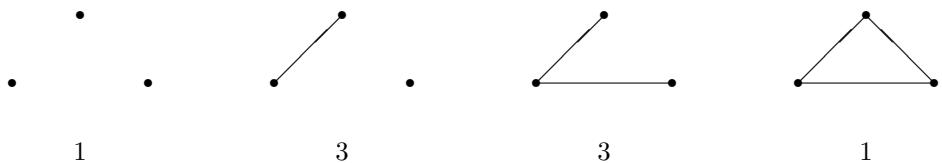


Abbildung 2.24: Die ungerichteten Graphen mit drei Ecken

Algorithmen, um beliebige Graphen „fair“ zu erzeugen, sind für große n sehr aufwendig. Es gibt zwar Verfahren mit einer mittleren Laufzeit von $O(n^2)$, aber diese sind für große n nicht durchführbar, da die beteiligten Konstanten sehr groß sind.

Eine Alternative besteht darin, daß man Klassen von Graphen betrachtet. Ein Beispiel hierfür ist die Klasse $G_{n,p}$, wobei n die Anzahl der Ecken ist und es zwischen je zwei Ecken mit der Wahrscheinlichkeit p eine Kante gibt (d.h. $p \in [0, 1]$). Der Erwartungswert für die Anzahl der Kanten für einen Graphen aus $G_{n,p}$ ist:

$$n(n - 1)p/2$$

2.10 Literatur

Das Standardwerk der theoretischen Graphentheorie ist das Buch von F. Harary [59]. Dort sind viele wichtige Ergebnisse zusammengestellt. Ausgezeichnete Einführungen in Algorithmen enthalten [92] und [24]. Eine Zusammenfassung neuerer Ergebnisse der algorithmischen Graphentheorie findet man in [88]. Eine sehr gute Einführung in Datenstrukturen findet man in [1], [101] und [94]. Für eine theoretische Einführung in die Komplexitätstheorie vergleiche man [103]. Neuere Ergebnisse über Algorithmen zur Bestimmung des transitiven Abschlusses eines gerichteten Graphen sind in [88] enthalten. Den dargestellten Algorithmus findet man in [121, 111]. Allgemeine Überlegungen zu der Qualität von Greedy-Algorithmen auf der Basis von Unabhängigkeitssystemen und Matroiden findet man in [76]. D. Knuth hat eine große Sammlung von Graphen und Programmen zur Erzeugung von Graphen zusammengestellt [79]; die zum Teil aus sehr kuriosen Anwendungen entstandenen Graphen sind auch als Dateien verfügbar.

2.11 Aufgaben

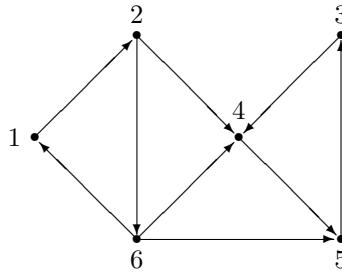
1. Ist die Anzahl der Ecken geraden Grades in einem ungerichteten Graphen stets gerade, stets ungerade oder kann man keine solche Aussage machen?
2. Beweisen Sie folgende Aussage: Haben in einem ungerichteten Graphen alle Ecken den Eckengrad 3, so ist die Anzahl der Ecken gerade.
3. Sei G ein ungerichteter bipartiter Graph mit n Ecken und m Kanten. Beweisen Sie, daß $4m \leq n^2$ gilt.
4. Es sei G ein regulärer, bipartiter Graph mit Eckenmenge $E = E_1 \cup E_2$. Beweisen Sie, daß $|E_1| = |E_2|$ gilt.
5. Es sei G ein ungerichteter Graph mit n Ecken und m Kanten. Es gelte

$$m > (n - 1)(n - 2)/2.$$

Beweisen Sie, daß G zusammenhängend ist! Gibt es Graphen mit $(n - 1)(n - 2)/2$ Kanten, die nicht zusammenhängend sind? Wenn ja, geben Sie Beispiele an.

6. Wie sehen die Komplemente der Graphen $K_{n,m}$ aus?
7. Beweisen Sie folgende Aussage: Für jeden Graphen G mit sechs Ecken enthält G oder \overline{G} einen geschlossenen Weg, der aus drei Kanten besteht.
8. Beweisen Sie, daß ein gerichteter Graph, der einen geschlossenen Weg enthält, auch einen einfachen geschlossenen Weg enthält.
9. Beweisen Sie folgende Aussage: Enthält ein Graph G kein Dreieck (d.h. keinen Untergraph vom Typ C_3), dann gilt $m \leq \Delta(G)(n - \Delta(G))$.
- * 10. Eine Teilmenge H der Eckenmenge E eines gerichteten Graphen heißt *unabhängig*, falls keine zwei Ecken aus H benachbart sind.

- a) Beweisen Sie, daß es in einem gerichteten Graphen ohne geschlossene Wege eine unabhängige Eckenmenge H gibt, so daß man von jeder Ecke aus $E \setminus H$ mit einer Kante eine Ecke in H erreichen kann.
- b) Zeigen Sie anhand eines Beispiels, daß die Aussage in gerichteten Graphen mit geschlossenen Wegen nicht gilt.
- c) Beweisen Sie, daß es in einem gerichteten Graphen eine unabhängige Eckenmenge H gibt, so daß von jeder Ecke aus $E \setminus H$ über einen aus höchstens zwei Kanten bestehenden Weg, eine Ecke aus H erreicht werden kann. H nennt man ein *Herz* des Graphen.
- d) Entwerfen Sie einen Algorithmus zur Bestimmung eines Herzens eines gerichteten Graphen.
11. Eine Kante eines ungerichteten Graphen heißt *Brücke*, falls durch ihre Entfernung die Anzahl der Zusammenhangskomponenten vergrößert wird. Beweisen Sie, daß es in einem Graphen, in dem jede Ecke geraden Eckengrad hat, keine Brücke geben kann.
12. Geben Sie für den folgenden gerichteten Graphen eine Darstellung mit Hilfe der Adjazenzmatrix, der Adjazenzliste und der Kantenliste an.



13. Ein ungerichteter Graph ist durch seine Kantenliste gegeben:

1	2	3	4	5	6	7	8	9
1	2	3	4	6	5	4	2	1
2	3	4	1	5	3	6	6	5

- a) Geben Sie eine geometrische Darstellung des Graphen an!
- b) Zeigen Sie, daß der Graph bipartit ist.
14. Eine Darstellungsart für gerichtete Graphen sind Adjazenzlisten, basierend auf Feldern. Entwerfen Sie Funktionen, um festzustellen, ob zwei gegebene Ecken benachbart sind und zur Bestimmung von Ausgangs- und Eingangsgrad einer Ecke. Erweitern Sie das Feld N um eine Komponente und setzen Sie $N[n+1] = m+1$. Wieso führt dies zu einer Vereinfachung der oben genannten Funktionen?
15. Die Adjazenzmatrix eines ungerichteten Graphen ohne Schlingen ist eine symmetrische Matrix, deren Diagonalelemente gleich 0 sind. In diesem Fall genügt es also,

die $(n^2 - n)/2$ Einträge oberhalb der Diagonalen zu speichern. Dies geschieht am einfachsten mit Hilfe eines Feldes der Länge $(n^2 - n)/2$. Schreiben Sie Prozeduren, die mittels dieser Darstellung feststellen, ob zwei gegebene Ecken benachbart sind und die Anzahl der Nachbarn einer gegebenen Ecke bestimmen.

16. Der Platzbedarf der Adjazenzliste eines ungerichteten Graphen kann dadurch verringert werden, indem zu jeder Ecke i nur die Nachbarecken j mit $j \geq i$ abgespeichert werden. Schreiben Sie Prozeduren, die mittels dieser Darstellung feststellen, ob zwei gegebene Ecken benachbart sind und die die Anzahl der Nachbarn einer gegebenen Ecke bestimmen. In welchem Fall ändert sich die Zeitkomplexität?
17. In der Adjazenzliste eines gerichteten Graphen sind die Nachbarn in der Reihenfolge aufsteigender Eckennummern abgespeichert. Nutzen Sie diese Tatsache beim Entwurf einer Prozedur, welche testet, ob zwei gegebene Ecken benachbart sind. Gehen Sie dabei von der Darstellung mittels Zeigern aus.
18. Die *Inzidenzmatrix* eines gerichteten Graphen G ist eine $n \times m$ -Matrix C mit den Einträgen

$$c_{ij} = \begin{cases} +1, & \text{falls } i \text{ Anfangsseite der Kante } j \text{ ist;} \\ -1, & \text{falls } i \text{ Endseite der Kante } j \text{ ist;} \\ 0, & \text{sonst.} \end{cases}$$

Hierfür sind die Kanten mit $1, \dots, m$ nummeriert. Für ungerichtete Graphen definiert man

$$c_{ij} = \begin{cases} 1, & \text{falls } i \text{ auf der Kante } j \text{ liegt} \\ 0, & \text{sonst.} \end{cases}$$

Geben Sie die Inzidenzmatrix für den Graphen aus Aufgabe 12 an. Entwerfen Sie Prozeduren, um festzustellen, ob zwei gegebene Ecken benachbart sind, und um die Anzahl der Nachbarn einer Ecke zu bestimmen. Unterscheiden Sie dabei die Fälle gerichteter und ungerichteter Graphen.

19. Für welche der in Abschnitt 2.4 diskutierten Datenstrukturen für ungerichtete Graphen kann der Grad einer Ecke in konstanter Zeit bestimmt werden?
20. In Abschnitt 2.6 wurde die Verteilung von Quellcode auf verschiedene Dateien mittels gerichteter Graphen dargestellt. Wie kann man feststellen, ob eine Datei sich selbst einschließt?
21. In diesem Kapitel wurde ein Algorithmus zur Bestimmung des transitiven Abschlusses eines ungerichteten Graphen G angegeben. Dabei wurden die ersten $n-1$ Potenzen der Adjazenzmatrix A aufaddiert und daraus die Erreichbarkeitsmatrix gebildet. Bilden Sie zunächst die Matrix¹

$$S = \prod_{i=0}^{\lfloor \log_2 n \rfloor} (I + A^{2^i}) - I$$

¹Für eine reelle Zahl x bezeichnet $\lfloor x \rfloor$ die größte natürliche Zahl kleiner oder gleich x .

und dann eine Matrix E :

$$e_{ij} = \begin{cases} 1 & \text{falls } s_{ij} \neq 0; \\ 0 & \text{falls } s_{ij} = 0; \end{cases}$$

Hierbei ist I die Einheitsmatrix. Beweisen Sie, daß E die Erreichbarkeitsmatrix von G ist. Wie viele Matrixmultiplikationen sind zur Bestimmung von S notwendig?

22. Beweisen Sie die folgenden Komplexitätsaussagen:

a) $n^2 + 2n + 3 = O(n^2)$

b) $\log n^2 = O(\log n)$

c) $\sum_{i=1}^n i = O(n^2)$

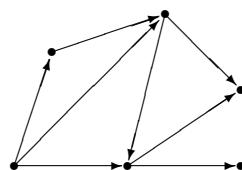
d) $\log n! = O(n \log n)$

- * 23. Eine Ecke e in einem gerichteten Graphen mit n Ecken heißt *Senke*, falls

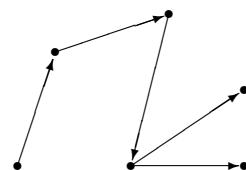
$$g^+(e) = 0 \text{ und } g^-(e) = n - 1.$$

Entwerfen Sie einen Algorithmus, der feststellt, ob ein gerichteter Graph eine Senke besitzt und diese dann bestimmt. Bestimmen Sie die Laufzeit Ihres Algorithmus! Es gibt einen Algorithmus mit Laufzeit $O(n)$.

24. Es sei G ein gerichteter Graph mit Eckenmenge E und Kantenmenge K . Die *transitive Reduktion* von G ist ein gerichteter Graph R mit der gleichen Eckenmenge E wie G und minimaler Kantenanzahl, so daß die Erreichbarkeitsmatrizen von G und R gleich sind. Ferner muß die Kantenmenge von R eine Teilmenge von K sein. Das folgende Beispiel zeigt einen gerichteten Graphen und eine transitive Reduktion.

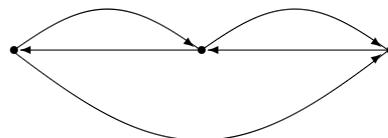


Gerichteter Graph



Transitive Reduktion

- a) Bestimmen Sie zwei verschiedene transitive Reduktionen des folgenden gerichteten Graphen:



- * b) Beweisen Sie, daß die transitive Reduktion eines gerichteten Graphen ohne geschlossene Wege eindeutig bestimmt ist. (Hinweis: Machen Sie einen Widerspruchsbeweis. Nehmen Sie an, es existieren zwei verschiedene transitive Reduktionen R_1 und R_2 . Betrachten Sie nun eine Kante k , die in R_1 enthalten ist, aber nicht in R_2 . Sei e die Anfangsecke und f die Endcke von k . Beweisen Sie, daß es in R_2 einen Weg W von e nach f gibt, der über eine dritte Ecke v führt. Zeigen Sie nun, daß es in R_1 einen Weg von e nach v und einen Weg von v nach f gibt, die beide nicht die Kante k enthalten. Dies ergibt aber einen Widerspruch.)
- c) Geben Sie ein Beispiel für einen gerichteten Graphen ohne geschlossene Wege mit n Ecken, dessen transitive Reduktion $O(n^2)$ Kanten enthält.
- * d) In einem ungerichteten Graphen ohne geschlossene Wege und Eckenmenge E bezeichnet $\text{ maxlen}(e, f)$ die maximale Länge eines Weges von Ecke e nach Ecke f . Es sei

$$K' = \{(e, f) \mid e, f \in E \text{ und } \text{ maxlen}(e, f) = 1\}$$

Beweisen Sie, daß K' die Menge der Kanten der transitiven Reduktion von G ist.

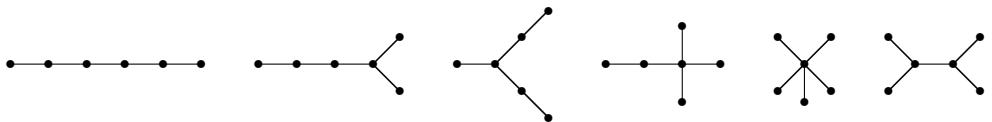
- * 25. Der folgende Algorithmus testet, ob ein ungerichteter Graph einen geschlossenen Weg der Länge 4 enthält. Es wird ein Feld **vierkreis** verwaltet, welches für jedes Paar von Ecken einen Eintrag hat. Die $n(n-1)/2$ Einträge von **vierkreis** werden mit 0 initialisiert. Nun werden die Ecken v des Graphen nacheinander betrachtet. Für jedes Paar (e, f) von zu v benachbarten Ecken e, f wird der entsprechende Eintrag von **vierkreis** um 1 erhöht. Dieser Vorgang wird abgebrochen, sobald ein Eintrag den Wert 2 bekommt. In diesem Fall liegt ein geschlossener Weg der Länge 4 vor, andernfalls gibt es keinen solchen Weg. Im ersten Fall sei (e, f) das zuletzt betrachtete Paar von Nachbarn. Dann liegen sich e und f auf einem geschlossenen Weg der Länge 4 gegenüber. Die anderen beiden Ecken lassen sich dann leicht bestimmen.
 - a) Geben Sie eine Implementierung für diesen Algorithmus an und beweisen Sie seine Korrektheit.
 - b) Zeigen Sie, daß der Algorithmus eine Laufzeit von $O(n^2)$ hat.
 - c) Ein geschlossener Weg der Länge 4 entspricht dem bipartiten Graphen $K_{2,2}$. Wie kann der Algorithmus abgeändert werden, so daß man feststellen kann, ob ein ungerichteter Graph einen Untergraphen vom Typ $K_{s,s}$ enthält? Welche Laufzeit hat dieser Algorithmus?
- * 26. Es sei G ein gerichteter Graph und E die Erreichbarkeitsmatrix von G . Es sei G' der Graph, der entsteht, wenn in G eine zusätzliche Kante eingefügt wird. Erstellen Sie einen Algorithmus, welcher die Erreichbarkeitsmatrix E' von G' unter Verwendung der Matrix E bestimmt. Zeigen Sie, daß der Algorithmus eine Laufzeit von $O(n^2)$ hat. Geben Sie ein Beispiel an, bei dem sich E und E' in $O(n^2)$ Positionen unterscheiden.

27. Geben Sie für den folgenden Algorithmus zur Bestimmung einer maximalen Clique in einem ungerichteten Graphen G eine Implementierung mit Aufwand $O(n + m)$ an.

```
C := G;
while C keine Clique do begin
    Sei e eine Ecke mit minimalem Eckengrad in C;
    Entferne e aus C;
end
```

Kapitel 3

Bäume



Eine für Anwendungen sehr wichtige Klasse von Graphen sind die sogenannten Bäume. In vielen Gebieten wie z.B. beim Compilerbau oder bei Datenbanksystemen werden sie häufig zur Darstellung von hierarchischen Beziehungen verwendet. In diesem Kapitel werden Anwendungen von Bäumen und ihre effiziente Darstellung beschrieben. Es wird ein auf Bäumen basierender Sortieralgorithmus vorgestellt und eine effiziente Realisierung von Vorrang-Warteschlangen diskutiert. Ferner werden zwei Algorithmen zur Bestimmung von minimalen aufspannenden Bäumen und ein Verfahren zur Datenkompression beschrieben.

3.1 Einführung

Ein ungerichteter Graph, der keinen geschlossenen Weg enthält, heißt *Wald*. Die Zusammenhangskomponenten eines Waldes nennt man *Bäume*. Ein Baum ist also ein zusammenhängender ungerichteter Graph, der keinen geschlossenen Weg enthält. Abbildung 3.1 zeigt alle möglichen Bäume mit höchstens fünf Ecken.

Abbildung 3.2 zeigt zwei Graphen, die keine Bäume sind. Der erste Graph enthält einen geschlossenen Weg; der zweite ist nicht zusammenhängend: dieser Graph ist ein Wald.

Für Bäume besteht folgender Zusammenhang zwischen der Anzahl m der Kanten und n der Ecken:

$$m = n - 1.$$

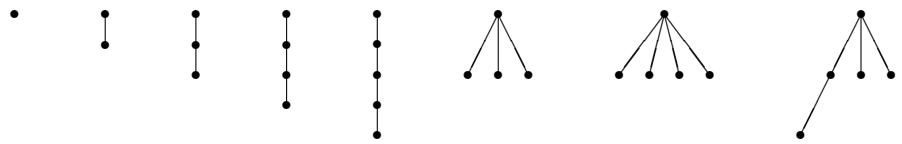


Abbildung 3.1: Alle Bäume mit höchstens fünf Ecken

Diese Aussage beweist man mit vollständiger Induktion nach n . Für $n = 1$ ist die Gleichung erfüllt. Ist die Anzahl der Ecken echt größer 1, so gibt es eine Ecke mit Eckengrad 1. Um sie zu finden, verfolgt man, von einer beliebigen Ecke aus startend, die Kanten des Baumes, ohne eine Kante zweimal zu verwenden. Da es keine geschlossenen Wege gibt, erreicht man keine Ecke zweimal, und der Weg endet an einer Ecke mit Eckengrad 1. Entfernt man diese Ecke und die zugehörige Kante aus dem Graphen, so kann man die Induktionsvoraussetzung anwenden und somit gilt

$$m - 1 = n - 2,$$

woraus sich die obige Gleichung ergibt.



Abbildung 3.2: Zwei Graphen, die keine Bäume sind

Für einen Wald, der aus z Zusammenhangskomponenten besteht, gilt:

$$m = n - z.$$

Diese Gleichung ergibt sich unmittelbar durch Addition der entsprechenden Gleichungen der einzelnen Zusammenhangskomponenten.

Einen Untergraphen eines Graphen G , der dieselbe Eckenmenge wie G hat und ein Baum ist, nennt man einen *aufspannenden Baum* von G . Man sieht leicht, daß jeder zusammenhängende Graph mindestens einen aufspannenden Baum besitzt. Dazu geht man folgendermaßen vor: Solange der Graph noch einen geschlossenen Weg hat, entferne man aus diesem eine Kante. Am Ende liegt ein aufspannender Baum vor. Abbildung 3.3 zeigt einen Graphen und zwei aufspannende Bäume für ihn.

Bäume sind zusammenhängende Graphen, und es gilt $m = n - 1$. Der folgende Satz zeigt, daß Bäume schon durch diese Eigenschaft charakterisiert sind.

Satz. Ein ungerichteter Graph G ist genau dann ein Baum, wenn eine der folgenden drei Bedingungen erfüllt ist:

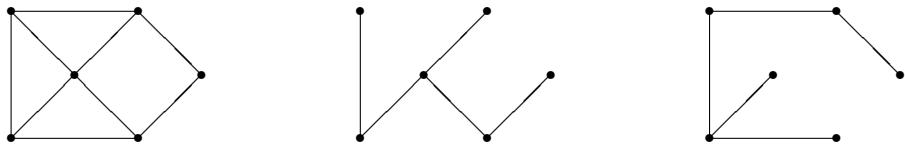


Abbildung 3.3: Ein Graph und zwei aufspannende Bäume

1. G ist zusammenhängend und $m = n - 1$;
2. G enthält keinen geschlossenen Weg und $m = n - 1$;
3. In G gibt es zwischen jedem Paar von Ecken genau einen Weg.

Beweis. Ein Baum hat sicherlich die angegebenen Eigenschaften. Um die umgekehrte Richtung zu beweisen, werden die drei Bedingungen einzeln betrachtet. Ist (1) erfüllt, so wähle man einen aufspannenden Baum B von G . Da B ein Baum mit n Ecken ist, hat B genau $n - 1$ Kanten. Also ist B gleich G , und G ist somit ein Baum. Ist (2) erfüllt, so ist G ein Wald, und aus der oben bewiesenen Gleichung $m = n - z$ folgt $z = 1$; d.h. G ist zusammenhängend und somit ein Baum.

Aus (3) folgt ebenfalls direkt, daß G ein Baum ist, denn würde es einen geschlossenen Weg geben, so gäbe es ein Paar von Ecken, zwischen denen zwei verschiedene Wege existieren. ■

Ein gerichteter Graph B heißt *Baum*, wenn der zugrundeliegende ungerichtete Graph ein Wurzelbaum ist und wenn es in B von der Wurzel aus genau einen Weg zu jeder Ecke gibt. Sind e und f Ecken eines gerichteten Baumes, so daß e von f aus erreichbar ist, so heißt f *Vorgänger* von e und e *Nachfolger* von f . Ein gerichteter Baum besitzt genau eine Wurzel. Abbildung 3.4 zeigt einen gerichteten Baum. Eine Ecke mit Ausgangsgrad 0 nennt man ein *Blatt* des Baumes. Die anderen Ecken nennt man *innere Ecken*.

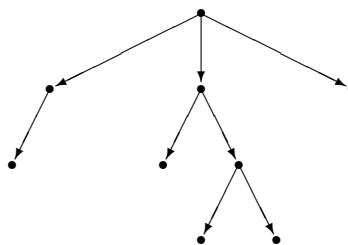


Abbildung 3.4: Ein gerichteter Baum

Wurzelbäume werden häufig als gerichtete Bäume interpretiert. Hierfür werden sie so gezeichnet, daß die Wurzel am höchsten liegt, und die Kanten werden so dargestellt,

daß ihre Richtungen nach unten zeigen. Gerichtete Bäume werden oft als ungerichtete Wurzelbäume dargestellt. Man interpretiert sie dann gemäß der oben angegebenen Konvention und läßt häufig auch die Richtungspfeile an den Kanten weg, da die Richtungen klar sind.

Das *Niveau* einer Ecke in einem Wurzelbaum ist gleich der Anzahl der Kanten des einzigen Weges von der Wurzel zu dieser Ecke. Ist w die Wurzel, so ist das Niveau einer Ecke e gleich $d(w, e)$. Die Wurzel hat das Niveau 0. Die *Höhe* eines Wurzelbaumes ist definiert als das maximale Niveau einer Ecke. Der Wurzelbaum aus Abbildung 3.4 hat die Höhe 3. Die von einer Ecke e eines Wurzelbaumes erreichbaren Ecken bilden den *Teilbaum mit Wurzel* e .

3.2 Anwendungen

Im folgenden werden vier wichtige Anwendungen von Bäumen diskutiert. Zunächst werden zwei Anwendungen aus den Gebieten Betriebssysteme und Compilerbau vorgestellt. Anschließend werden Suchbäume und ein Verfahren zur Datenkompression ausführlicher dargestellt.

3.2.1 Hierarchische Dateisysteme

Das Betriebssystem eines Computers verwaltet alle Dateien des Systems. Die Dateiverwaltung vieler Betriebssysteme organisiert die Dateien nicht einfach als eine unstrukturierte Menge, sondern die Dateien werden in einer Hierarchie angeordnet, die eine Baumstruktur trägt. Man führt dazu einen speziellen Dateityp ein: *Directories* oder *Verzeichnisse*. Sie enthalten Namen von Dateien und Verweise auf diese Dateien, die ihrerseits wieder Verzeichnisse sein können. Jede Ecke in einem Dateibaum entspricht einer Datei; Dateien die keine Verzeichnisse sind, sind Blätter des Baumes. Die Nachfolger eines Verzeichnisses sind die in ihm enthaltenen Dateien. Abbildung 3.5 zeigt ein Beispiel eines Dateisystems unter dem Betriebssystem UNIX. Hierbei sind Ecken, welche Verzeichnisse repräsentieren, als Kreise und reine Dateien als Rechtecke dargestellt.

3.2.2 Ableitungsbäume

Die Übersetzung von höheren Programmiersprachen in maschinenorientierte Sprachen erfolgt durch einen Compiler. Dabei wird ein Quellprogramm in ein semantisch äquivalentes Zielprogramm übersetzt. Ein wichtiger Bestandteil eines Compilers ist ein Parser, dessen Hauptaufgabe es ist, zu untersuchen, ob das Quellprogramm syntaktisch korrekt ist. Dies ist der Fall, wenn es der zugrundeliegenden Grammatik der Programmiersprache entspricht. Ein Parser liest ein Quellprogramm, überprüft die Syntax und meldet Fehler. Für den Parser besteht das Quellprogramm aus einer Folge von Symbolen. Diese werden zu grammatischen Sätzen zusammengefaßt, die der Compiler in den weiteren Phasen der Übersetzung weiterverarbeitet. Diese Sätze werden durch sogenannte *Ableitungsbäume* dargestellt. Dies sind Wurzelbäume, wobei die Ecken mit den Symbolen der Sätze markiert sind. Die Kanten reflektieren den syntaktischen Aufbau des Programms.

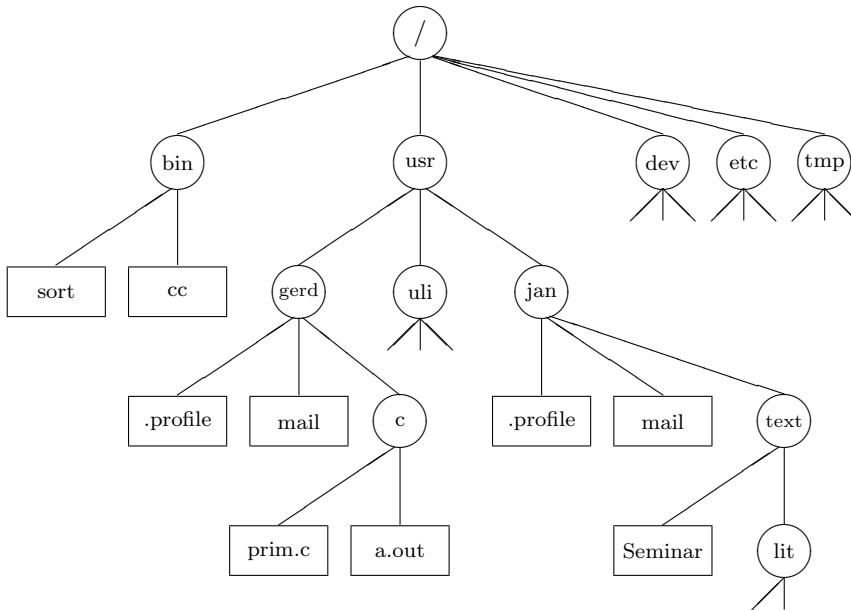


Abbildung 3.5: Verzeichnisstruktur eines Dateisystems

Man unterscheidet zwischen Nichtterminalsymbolen (das sind Zeichen, die grammatischen Konstrukte repräsentieren) und Terminalssymbolen, den elementaren Wörtern der Sprache. Im Ableitungsbaum entsprechen die Terminalssymbole gerade den Blättern. Die Wurzel des Ableitungsbaumes ist mit dem sogenannten Startsymbol markiert. Abbildung 3.6 zeigt den Ableitungsbaum der Wertzuweisung $y := x + 200 * z$, wie sie in einem Pascalprogramm vorkommen kann. Auf die entsprechende Grammatik wird hier nicht näher eingegangen.

Ableitungsbäume sind Beispiele für *geordnete Bäume*. Bei einem geordneten Baum stehen die Nachfolger jeder Ecke in einer festen Reihenfolge. Häufig werden die Nachfolger von links nach rechts gelesen. Bei Ableitungsbäumen ergeben die Markierungen der Blätter von links nach rechts gelesen den Programmtext. In Abbildung 3.6 ergibt sich von links nach rechts: $y := x + 200 * z$. Ableitungsbäume bilden die Grundlage für die semantische Analyse, die z.B. überprüft, ob bei der Wertzuweisung der Typ der Variablen auf der linken Seite mit dem Typ des Ausdrückes auf der rechten Seite übereinstimmt.

3.2.3 Suchbäume

Während der Analyse eines Quellprogramms muß ein Compiler die verwendeten Bezeichner zusammen mit der zugehörigen Information abspeichern. Zum Beispiel wird für jede Variable der Typ abgespeichert. Diese Information wird in der sogenannten *Symboltabelle* verwaltet, die als Suchbaum implementiert werden kann. Im weiteren

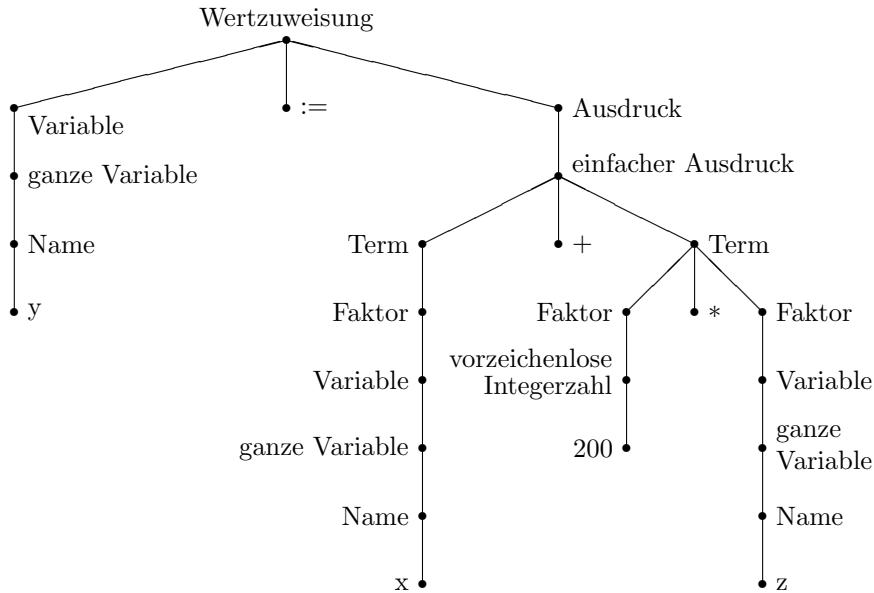


Abbildung 3.6: Der Ableitungsbaum für $y := x + 200 * z$

Verlauf der Analyse wird dann immer wieder auf diese Symboltabelle zugegriffen: Es wird nach Einträgen gesucht und es werden neue Einträge hinzugefügt. Neben Löschen von Einträgen sind dies die Operationen, welche Suchbäume zur Verfügung stellen. Allgemein eignen sich Suchbäume für die Verwaltung von Informationen, die nach einem vorgegebenen Kriterium linear geordnet sind. Bei einer Symboltabelle ist die Ordnung durch die alphabetische Ordnung der Namen (*lexikographische Ordnung*) bestimmt. Zu den einzelnen Objekten werden noch zusätzliche Informationen gespeichert (im obigen Beispiel die Typen der Variablen etc.); deshalb nennt man den Teil der Information, auf dem die Ordnung definiert ist, den *Schlüssel* des Objektes.

Symboltabellen werden häufig durch eine spezielle Form von geordneten Bäumen realisiert, den sogenannten *Binärbäumen*. Ein Binärbaum ist ein geordneter Wurzelbaum, bei dem jede Ecke höchstens zwei Nachfolger hat. Die Ordnung der Nachfolger einer Ecke wird dadurch gekennzeichnet, daß man von einem linken und einem rechten Nachfolger spricht. Ein *binärer Suchbaum* ist ein Binärbaum, bei dem die Ecken mit den Elementen einer geordneten Menge markiert sind und bei dem die folgende *Suchbaumbedingung* erfüllt ist: Für jede Ecke e sind die Schlüssel der Markierungen der Ecken des linken Teilbaumes von e kleiner als der Schlüssel der Markierung von e , und die Schlüssel der Markierungen der Ecken des rechten Teilbaumes von e sind größer als der Schlüssel von e . Diese Bedingung muß für alle Ecken einschließlich der Wurzel des Baumes gelten. Dies setzt voraus, daß die Schlüssel der Markierungen eine lineare Ordnung tragen und daß die Schlüssel aller Objekte verschieden sind.

Abbildung 3.7 zeigt einen binären Suchbaum zur Verwaltung von Typen von Variablen

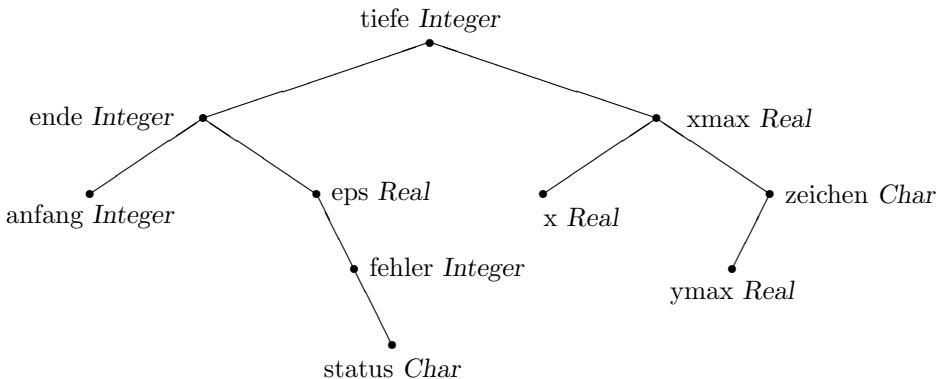


Abbildung 3.7: Ein binärer Suchbaum

während der Übersetzung eines Programms durch einen Compiler. Die Markierungen bestehen in diesem Falle aus den Namen von Variablen und den dazugehörigen Typen. Hierbei bilden die Namen die Schlüssel, und die Ordnung ist wieder die lexikographische Ordnung. Da der Schlüssel der Wurzel *tiefe* ist, sind die Schlüssel der Ecken im linken Teilbaum lexikographisch kleiner und die im rechten Teilbaum größer als *tiefe*.

Wie unterstützen binäre Suchbäume Abfragen nach dem Vorhandensein eines Objekts mit einem bestimmten Schlüssel s ? Man vergleicht zuerst s mit dem Schlüssel w der Wurzel. Falls $s = w$, so ist man fertig, und das gesuchte Objekt ist in der Wurzel gespeichert. Andernfalls ist $s < w$ oder $s > w$. Die Suchbaumbedingung besagt nun, daß das Objekt, falls es vorhanden ist, im ersten Fall im linken Teilbaum der Wurzel und im zweiten Fall im rechten Teilbaum gespeichert ist. Die Suche beschränkt sich also auf einen der zwei Teile und wird genauso fortgesetzt. Falls die Suche bei einer Ecke angelangt ist, die keine Nachfolger hat, so ist das Objekt nicht im Suchbaum vorhanden. Die Suche in binären Suchbäumen ist also ein rekursiver Vorgang: Das Ergebnis des Vergleichs der Schlüssel von Wurzel und Objekt entscheidet, welcher Teilbaum weiter untersucht wird. Dort erfolgt wieder der Schlüsselvergleich mit der Wurzel. Die Suche endet, falls das Objekt gefunden wurde oder falls man bei einem Blatt angelangt ist.

Bevor auf eine konkrete Realisierung des Suchvorgangs eingegangen wird, soll zuerst eine effiziente Darstellung von binären Suchbäumen vorgestellt werden. Diese Darstellung basiert auf Zeigern:

```

Eckentyp = record
  Schlüssel : Schlüsseltyp;
  Daten : Datentyp;
  linkerNachfolger, rechterNachfolger : zeiger Eckentyp;
end
  
```

Der Typ **Eckentyp** beschreibt die Information, die zu einer Ecke gehört: Schlüssel, Daten und die beiden Nachfolger. Die beiden Typen **Schlüsseltyp** und **Datentyp** werden

in Abhängigkeit der Anwendung vereinbart. Der eigentliche Typ, der den binären Suchbaum beschreibt, besteht nur noch aus einem Zeiger auf die Wurzel:

```
Suchbaum = zeiger Eckentyp;
```

Das Suchverfahren läßt sich nun leicht durch die in Abbildung 3.8 dargestellte rekursive Funktion `suchen` realisieren.

```
function suchen(s : Schluesseltyp; b : Suchbaum;
                           var d : Datenyp) : Boolean;
begin
  if b = nil then
    suchen := FALSE
  else if s = b → Schluessel then begin
    d := b → Daten;
    suchen := TRUE;
  end
  else if s < b → Schluessel then
    suchen := suchen(s, b → linkerNachfolger, d)
  else
    suchen := suchen(s, b → rechterNachfolger, d);
end
```

Abbildung 3.8: Die Funktion suchen

Der Rückgabewert von `suchen` ist TRUE, falls der gesuchte Schlüssel s im Suchbaum vorkommt, und in diesem Fall enthält d die Daten des entsprechenden Objektes.

Wie aufwendig ist die Suche in Binäräumen? Im ungünstigsten Fall muß von der Wurzel bis zu einem Blatt gesucht werden. Das heißt, ist h die Höhe des Suchbaums, so sind $O(h)$ Vergleiche durchzuführen. Ein Binärbaum der Höhe h enthält höchstens $2^{h+1} - 1$ Ecken. Somit ist die Höhe eines Binärbaumes mit n Ecken mindestens:

$$\log_2(n + 1) - 1$$

Man beachte aber, daß die Höhe eines Binärbaumes mit n Ecken im ungünstigsten Fall gleich $n - 1$ sein kann. Abbildung 3.9 zeigt einen Binärbaum mit denselben zehn Objekten wie der Binärbaum aus Abbildung 3.7. Im ersten Fall hat der Baum die Höhe 4 und im zweiten Fall die Höhe 9. In diesem Fall entartet der Suchbaum zu einer linearen Liste, und beim Suchen sind $O(n)$ Vergleiche durchzuführen. Der Aufwand der Funktion `suchen` hängt also wesentlich von der Höhe des Suchbaumes ab.

Wie werden Suchbäume aufgebaut, und wie werden neue Objekte in einen Suchbaum eingefügt? Das Einfügen eines Objektes mit Schlüssel s erfolgt analog zur Funktion `suchen`. Ist der Suchbaum noch leer, so wird die Wurzel erzeugt, und in diese werden die Daten eingetragen. Andernfalls wird nach einem Objekt mit dem Schlüssel s gesucht. Findet man ein solches Objekt, so kann das neue Objekt nicht eingefügt werden,

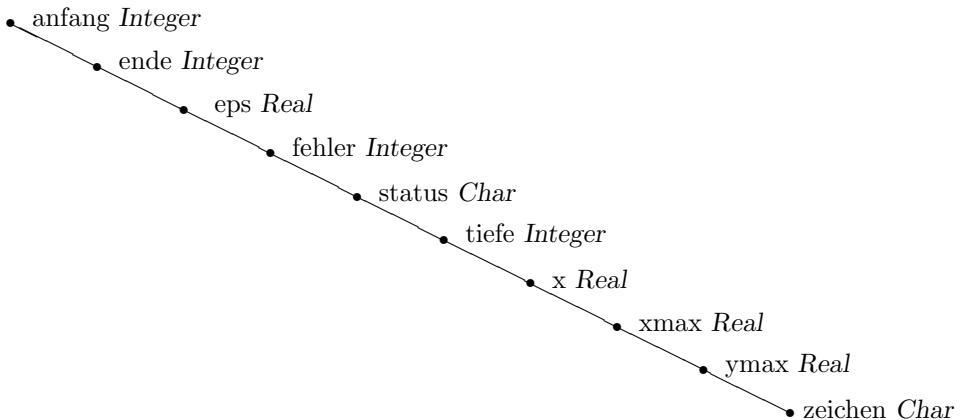


Abbildung 3.9: Ein entarteter binärer Suchbaum

da alle Objekte im Suchbaum verschiedene Schlüssel haben müssen. Ist die Suche erfolglos, so endet sie bei einer Ecke. Entweder ist diese Ecke ein Blatt oder sie hat nur einen Nachfolger. Entsprechend dem Schlüssel dieser Ecke wird ein linker bzw. rechter Nachfolger erzeugt, und in diesen wird das Objekt eingetragen. Das Löschen von Objekten in Suchbäumen ist etwas aufwendiger (vergleichen Sie Übungsaufgabe 10).

Die Algorithmen für Suchen, Einfügen und Löschen von Objekten benötigen im Mittel $O(\log_2 n)$ Schritte, wobei n die Anzahl der Einträge im Suchbaum ist. Die worst case Komplexität ist $O(h)$, wobei h die Höhe des Suchbaumes ist. Bei entarteten Suchbäumen kann dies sogar $O(n)$ sein. Da dies für praktische Anwendungen nicht akzeptabel ist, sind sogenannte *höhenbalancierte Suchbäume* vorzuziehen. Sie garantieren eine Such- und Einfügezeit von $O(\log_2 n)$. Das Einfügen neuer Objekte ist dabei komplizierter, denn es muß auf die Ausgeglichenheit des Baumes geachtet werden und dazu sind zusätzliche Operationen notwendig. Bei höhenbalancierten Suchbäumen ist garantiert, daß die Höhe des völlig ausgeglichenen Gegenstückes nie um mehr als 45% überstiegen wird. Dies gilt unabhängig von der Anzahl der vorhandenen Objekte.

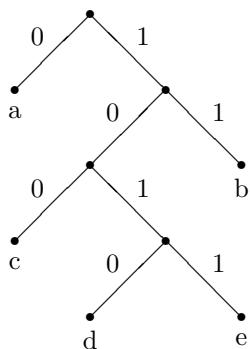
3.2.4 Datenkompression

Ziel der Datenkompression ist eine Reduzierung einer Datenmenge unter Beibehaltung des Informationsinhaltes, um Speicherplatz und Übertragungszeit zu verringern. Die ursprünglichen Daten müssen sich wieder eindeutig aus den komprimierten Daten rekonstruieren lassen. Typische Beispiele für Daten, die komprimiert werden, sind Texte und Binärdateien wie übersetzte Programme oder digitalisierte Bilder. Binärbäume bieten eine einfache Möglichkeit, Datenkompression durchzuführen. Hierbei werden die einzelnen Zeichen der Originaldaten (z.B. Buchstaben in einem Text) nicht durch Codes konstanter Länge dargestellt (wie dies z.B. beim ASCII-Code geschieht), sondern es wird die Häufigkeit der vorhandenen Zeichen berücksichtigt. Es wird die Tatsache

ausgenutzt, daß in vielen Fällen nicht alle Zeichen mit der gleichen Wahrscheinlichkeit vorkommen, sondern manche Zeichen nur selten und manche Zeichen sehr häufig. Häufig vorkommende Zeichen bekommen einen kürzeren Code zugeordnet als weniger häufig vorkommende.

Im folgenden gehen wir immer davon aus, daß die komprimierten Daten in binärer Form dargestellt werden, d.h. als Folge der Zeichen 0 und 1. Das Verfahren basiert auf der Annahme, daß die einzelnen Zeichen mit bekannten Häufigkeiten vorkommen. Die naheliegende Vorgehensweise, die Zeichen nach ihrer Häufigkeit zu sortieren und dann dem ersten Zeichen die 0 zuzuordnen, dem zweiten die 1, dem dritten die 00 etc., stößt auf das Problem, daß die ursprünglichen Daten sich nicht eindeutig rekonstruieren lassen. Es läßt sich nicht entscheiden, ob 00 zweimal das Zeichen repräsentiert, welches durch die 0 dargestellt ist, oder nur ein Zeichen, nämlich das mit der dritthöchsten Wahrscheinlichkeit. Dieses Problem wird durch sogenannte *Präfix-Codes* gelöst. Bei Präfix-Codes unterscheidet sich die Darstellung von je zwei verschiedenen Zeichen schon im Präfix: Die Darstellung eines Zeichens ist niemals der Anfang der Darstellung eines anderen Zeichens. Die oben beschriebene Situation kann also bei Präfix-Codes nicht vorkommen, da 0 und 00 nicht gleichzeitig als Codewörter verwendet werden; bei Präfix-Codes kann also nicht jede Folge der Zeichen 0 und 1 als Codewort verwendet werden.

Präfix-Codes lassen sich durch Binäräbäume darstellen. Hierbei entsprechen die dargestellten Wörter den Blättern des Baumes; die Codierung ergibt sich aus dem eindeutigen Weg von der Wurzel zu dem entsprechenden Blatt. Eine Kante zu einem linken Nachfolger entspricht einer 0 und eine Kante zu einem rechten Nachfolger einer 1. Jeder Code, der auf diese Art aus einem Binärbaum entsteht, ist ein Präfix-Code. Abbildung 3.10 zeigt einen solchen Binärbaum für den Fall, daß die zu komprimierenden Daten aus den fünf Zeichen a, b, c, d und e bestehen.



Codewörter :

a:	0
b:	1 1
c:	1 0 0
d:	1 0 1 0
e:	1 0 1 1

Abbildung 3.10: Binärbaum für einen Präfix-Code

Der Text *bade* wird als 11010101011 dargestellt, und umgekehrt entspricht dem codierten Text 1001011110 der Text *ceba*. Abbildung 3.11 zeigt einen weiteren Präfix-Code für die Zeichen a,b,c,d und e.

Die maximale Länge eines Codewortes ist im ersten Fall 4 und im zweiten Fall 3. Welcher der beiden Codes erzielt eine höhere Kompression? Die maximale Länge eines Codewortes gibt darüber noch keine Auskunft. Vorgegeben seien die folgenden Häufigkeiten der einzelnen Buchstaben.

Zeichen	Häufigkeit
a	51%
b	20%
c	8%
d	15%
e	6%

Ein Maß für die Güte einer Codierung ist die *mittlere Codewortlänge* l der Codierung. Es gilt:

$$l = \sum_{i=1}^n p_i l_i.$$

Hierbei ist l_i die Länge des i-ten Codewortes und p_i die entsprechende Häufigkeit. Für die oben angegebenen Codierungen ergibt sich im ersten Fall eine mittlere Codewortlänge von 1.99 und im zweiten Fall von 2.21. Somit ist bei dieser Häufigkeitsverteilung die erste Codierung der zweiten vorzuziehen. Die Frage ist, ob es für diese Häufigkeitsverteilung eine Codierung mit einer noch kleineren mittleren Codewortlänge gibt.

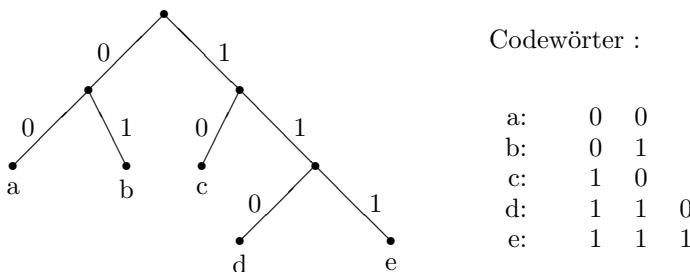


Abbildung 3.11: Binärbaum für einen Präfix-Code

Ein Algorithmus, der zu einer gegebenen Häufigkeitsverteilung einen Präfix-Code mit minimaler mittlerer Codewortlänge konstruiert, stammt von D.A. Huffman. Dieser wird im folgenden beschrieben: Es wird dabei ein Binärbaum konstruiert, aus dem sich dann der Präfix-Code ergibt. Hierbei wird zuerst ein Wald erzeugt und dieser wird dann sukzessive in einen Binärbaum umgewandelt. Im ersten Schritt erzeugt man für jedes Zeichen einen Wurzelbaum mit nur einer Ecke, die mit dem Zeichen und der entsprechenden Häufigkeit markiert ist. Der so entstandene Wald heißt W . In jedem weiteren Schritt werden nun zwei Bäume aus W zu einem einzigen vereinigt, bis W ebenfalls ein Baum ist. Dabei sucht man die beiden Bäume B_1 und B_2 aus W , deren Wurzeln die

kleinsten Markierungen haben. Ist dies auf mehrere Arten möglich, so hat die Auswahl der beiden Bäume keinen Einfluß auf die erzielte mittlere Wortlänge der Codierung. Diese Bäume werden zu einem Baum verschmolzen. Die Wurzel dieses Baumes wird mit der Summe der Markierungen der Wurzeln von B_1 und B_2 markiert. Die Nachfolger der Wurzel sind die Wurzeln von B_1 und B_2 . Auf diese Weise entsteht der gesuchte Binärbaum.

Abbildung 3.12 zeigt die Entstehung des Binärbaumes für das angegebene Beispiel, und Abbildung 3.13 zeigt die erzeugte Codierung. Es ergibt sich eine mittlere Codewortlänge von 1.92, d.h. die erzielte Codierung ist besser als die ersten beiden. Bei dem Verfahren ist es nicht notwendig, alle Ecken zu markieren. Lediglich die Wurzeln und die Blätter bekommen noch die entsprechenden Markierungen.

Der durch den Huffman-Algorithmus erzeugte Binärbaum ist nicht eindeutig, denn es kann passieren, daß die Wurzeln von mehreren Bäumen die gleiche Gewichtung haben. In diesem Falle haben die entstehenden Codes die gleiche mittlere Codewortlänge. Diese ist auch unabhängig davon, welcher der zwei Bäume mit minimalem Gewicht zum linken bzw. rechten Teilbaum des neuen Baumes gemacht wird. Im folgenden wird die Optimalität des Huffman-Algorithmus bewiesen.

Satz. Der Huffman-Algorithmus bestimmt für gegebene Häufigkeiten von Zeichen einen Präfix-Code mit minimaler Wortlänge.

Beweis. Es seien p_1, \dots, p_n die Häufigkeiten der Zeichen in aufsteigender Reihenfolge. Im folgenden werden die Ecken mit ihren Häufigkeiten identifiziert. Zunächst wird gezeigt, daß es einen Präfix-Code minimaler Wortlänge gibt, in dessen Binärbaum es auf dem vorletzten Niveau eine Ecke gibt, deren Nachfolger p_1 und p_2 sind. Dazu betrachte man eine beliebige Ecke x auf dem vorletzten Niveau. Es seien c und d mit $c \leq d$ die Nachfolger von x . Falls $c = p_1$ und $d = p_2$ so ist die Aussage erfüllt. Ist $d > p_2$, so folgt aus der Minimalität des Codes $l_{p_2} \geq l_d$. Da d auf dem untersten Niveau liegt, gilt $l_d \geq l_{p_2}$. Somit ist $l_d = l_{p_2}$ und analog $l_c = l_{p_1}$. Vertauscht man nun p_2 und d bzw. p_1 und c im Binärbaum, so erhält man den gewünschten Präfix-Code.

Die Aussage des Satzes wird nun durch vollständige Induktion nach n bewiesen. Für $n = 1, 2$ ist die Optimalität offensichtlich gegeben. Sei nun $n > 2$ und B ein optimaler Binärbaum für p_1, \dots, p_n , indem p_1 und p_2 Brüder sind. Man entferne die zu p_1 und p_2 gehörenden Blätter und markiere das neu entstandene Blatt mit $p_1 + p_2$. Es sei B' der neue Baum. Dies ist ein Binärbaum für die Häufigkeiten $p_1 + p_2, p_3, \dots, p_n$. Für die mittlere Wortlängen l_B von B und $l_{B'}$ von B' gilt $l_B = l_{B'} + p_1 + p_2$. Sei nun B'_1 ein vom Huffmann-Algorithmus erstellter Binärbaum für die Häufigkeiten $p_1 + p_2, p_3, \dots, p_n$. Dann ist B'_1 nach Induktionsvoraussetzung optimal. Es sei B_1 der Baum, welcher aus B'_1 hervorgeht, in dem die Ecke $p_1 + p_2$ zwei Nachfolger mit den Häufigkeiten p_1 und p_2 bekommt. B_1 ist gerade der vom Huffmann-Algorithmus erstellte Baum für die Häufigkeiten p_1, \dots, p_n . Nun gilt $l_{B_1} = l_{B'_1} + p_1 + p_2$. Nun kann aber l_{B_1} nicht echt größer als l_B sein, denn dies würde $l_{B'_1} > l_{B_1}$ implizieren, was wegen der Optimalität von B'_1 nicht gilt. Somit ist $l_{B_1} = l_B$, d.h. B_1 ist optimal. ■

Der Aufwand des Huffmann-Algorithmus bei einer Eingabe von n Zeichen ist $O(n \log n)$.

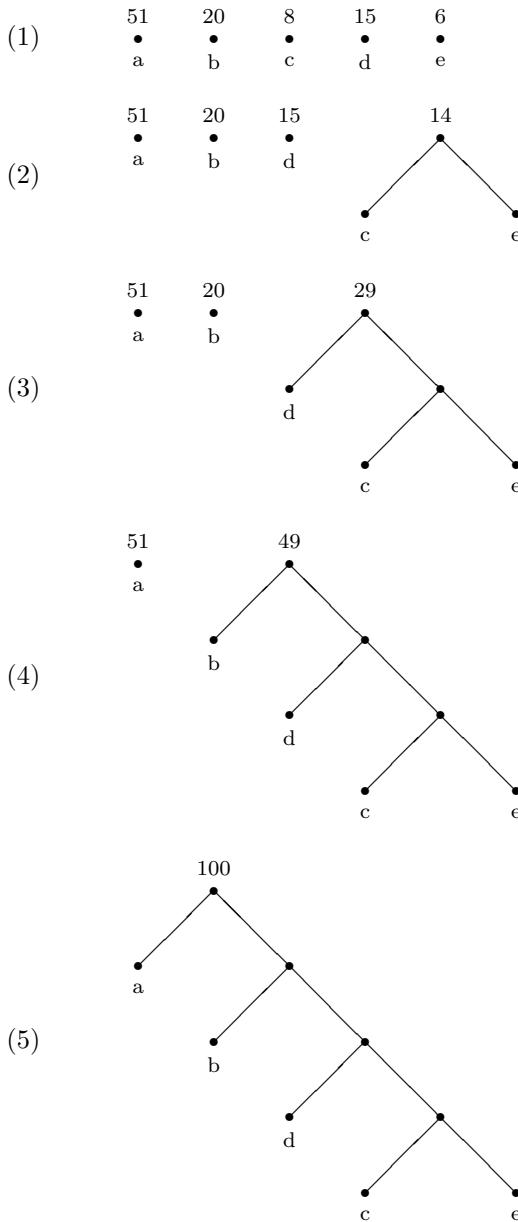


Abbildung 3.12: Eine Anwendung des Algorithmus von Huffman

a:	0
b:	1 0
c:	1 1 1 0
d:	1 1 0
e:	1 1 1 1

Abbildung 3.13: Die vom Algorithmus von Huffman erzeugte Codierung

Der erzeugte Baum hat insgesamt n Blätter, also hat der Baum insgesamt $2n - 1$ Ecken (vergleichen Sie Übungsaufgabe 5); somit sind n Schritte notwendig. Das anfängliche Sortieren der n Bäume hat den Aufwand $O(n \log n)$. Durch eine geschickte Verwaltung der Bäume des Waldes W erreicht man, daß der Gesamtaufwand $O(n \log n)$ ist. Eine dafür geeignete Datenstruktur wird im nächsten Abschnitt vorgestellt.

Einen höheren Komprimierungsgrad als der Huffman-Algorithmus erzielen Verfahren, die nicht einzelne Zeichen, sondern Paare, Tripel oder größere Gruppen von Quellzeichen zusammen codieren.

3.3 Datenstrukturen für Bäume

Die in Abschnitt 2.4 vorgestellten Datenstrukturen für Graphen können natürlich auch für Bäume verwendet werden. Die Eigenschaft, daß es keine geschlossenen Wege gibt, führt aber zu effizienteren Datenstrukturen. Eine Datenstruktur für Binäräbäume wurde schon im letzten Abschnitt vorgestellt.

3.3.1 Darstellung mit Feldern

In einem Wurzelbaum hat jede Ecke außer der Wurzel genau einen Vorgänger, d.h. durch die Vorgänger ist der Baum eindeutig bestimmt. Das führt zur folgenden einfachen Datenstruktur:

```
Wurzelbaum = array[1..max] of Integer;
```

Für einen Wurzelbaum, der durch eine Variable **A** vom Typ **Wurzelbaum** dargestellt ist, bezeichnet **A[i]** den Vorgänger der Ecke *i*. Ist **A[i] = 0**, so bedeutet dies, daß *i* die Wurzel ist. Nicht genutzte Einträge in dem Feld werden speziell gekennzeichnet. Abbildung 3.14 zeigt einen Wurzelbaum und die Darstellung mit Hilfe eines Feldes.

Tragen die Ecken des Wurzelbaumes noch Markierungen, so werden diese in einem getrennten Feld abgespeichert. Der größte Vorteil dieser Datenstruktur ist die Unterstützung der Abfrage nach dem Vorgänger einer Ecke. Dadurch findet man auch sehr effizient den Weg von einer Ecke zu der Wurzel des Baumes. Man muß nur die entsprechenden Indizes in dem Array verfolgen, bis man bei 0 ankommt; das Auffinden der Nachfolger einer Ecke wird nur schlecht unterstützt. Es ist dazu notwendig, das ganze Feld zu durchsuchen. Ferner kann man mit dieser Datenstruktur keine Ordnungsbäume abspeichern, da die Ordnung der Nachfolger einer Ecke durch das Feld

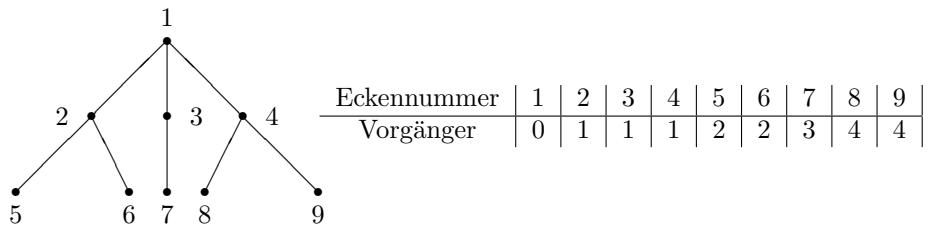


Abbildung 3.14: Ein Wurzelbaum und seine Darstellung mittels eines Feldes

nicht bestimmt ist. Dies kann man nur dadurch erzwingen, daß die Reihenfolge durch die Numerierung implizit gegeben ist. Für Ordnungsbäume sind Datenstrukturen, die auf der Adjazenzliste aufbauen, günstiger.

3.3.2 Darstellung mit Adjazenzlisten

Die in Kapitel 2 diskutierte Adjazenzliste eignet sich natürlich auch für Wurzelbäume. Die Reihenfolge der Nachfolger wird durch die Reihenfolge innerhalb der Nachbarliste berücksichtigt. Somit können auch Ordnungsbäume dargestellt werden. Sollen spezielle Operationen, wie sie z.B. in dem Algorithmus von Huffman vorkommen, unterstützt werden, sind allerdings noch einige Änderungen notwendig. Der Huffman-Algorithmus geht von einem Wald aus, und in jedem Schritt werden zwei Bäume verschmolzen bis nur noch ein Baum übrig bleibt. Die im folgenden diskutierte Datenstruktur unterstützt diese Operation sehr effizient. Abbildung 3.15 zeigt eine Listenrepräsentation für Bäume.

```

GraphIndex = 1..max;
Eckenmenge = array[GraphIndex] of record
    Bewertung : Bewertungstyp;
    linkerNachfolger : GraphIndex;
    rechterNachfolger : GraphIndex;
    Vorgänger : GraphIndex;
end;
Wald = record
    Ecken : Eckenmenge;
    Wurzeln : array[GraphIndex] of GraphIndex;
    frei : GraphIndex;
end

```

Abbildung 3.15: Listenrepräsentation für Bäume

Der Typ **Eckenmenge** ist ein Feld mit einer Komponente für jede Ecke des Baumes; jede

Ecke wird durch eine Struktur mit vier Komponenten dargestellt: eine Komponente für die Bewertung der Ecke, zwei Komponenten für die Indizes des linken und des rechten Nachfolgers und eine Komponente für den Index des Vorgängers. Existiert eine dieser drei Ecken nicht, so wird dies durch einen besonderen Eintrag gekennzeichnet. Der Typ **Wald** ist ein Record bestehend aus drei Komponenten: einem Feld vom Typ **Eckenmenge** für die Ecken, einem Feld für die Wurzeln der Bäume des Waldes und einem Feld für die erste ungenutzte Komponente in **Ecken**. Die ungenutzten Ecken sind über ihre rechten Nachfolger verkettet. Auf diese Art lässt sich der freie Platz leicht verwalten. Kommt eine neue Ecke hinzu, so findet man mit Hilfe der Komponente **frei** einen freien Platz. Anschließend wird die Komponente **frei** aktualisiert. Abbildung 3.16 zeigt einen Wald, wie er im Huffman-Algorithmus vorkommt. Die Numerierung der Ecken gibt ihren Index in dem Feld **Eckenmenge** an.

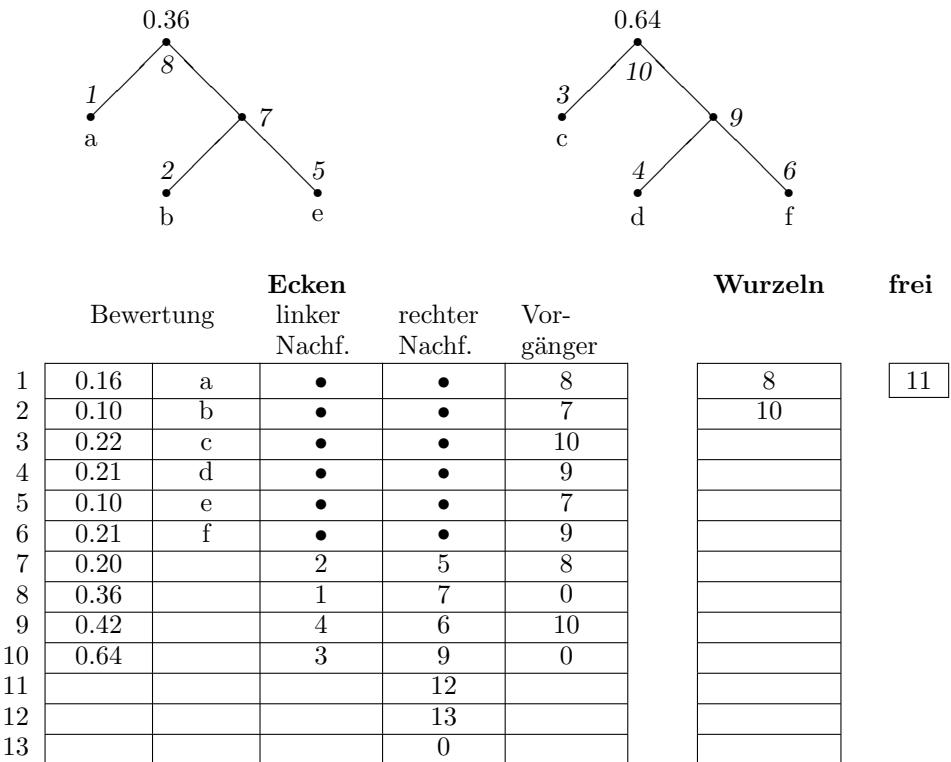


Abbildung 3.16: Datenstruktur für den Algorithmus von Huffman

Das Verschmelzen zweier Bäume als Wald kann nun in konstanter Zeit erfolgen. Zuerst muß eine neue Ecke erzeugt werden, die neue Wurzel. Über die Komponente **frei** wird ein freier Platz in dem Feld **Ecken** gefunden. An dieser Stelle erfolgt nun die Eintragung der neuen Wurzel. Nachfolger sind die alten Wurzeln; bei diesen muß noch der Vorgängereintrag geändert werden. Zuvor muß noch die Komponente **frei** auf den

neuen Stand gebracht werden. Die alten Wurzeln werden aus dem Feld **Wurzeln** entfernt, und die neue Wurzel wird eingetragen.

3.4 Sortieren mit Bäumen

Sortieren ist in der Datenverarbeitung eine häufig vorkommende Aufgabe. Es wurde dafür eine große Vielfalt von Algorithmen entwickelt. Je nach dem Grad der Ordnung in der Menge der zu sortierenden Objekte sind unterschiedliche Algorithmen effizienter. Folgende theoretische Aussage gibt eine untere Schranke für den Aufwand des Sortierens an: Um n Objekte zu sortieren, benötigt ein Algorithmus, der seine Information über die Anordnung der Objekte nur aus Vergleichsoperationen bezieht, im allgemeinen Fall mindestens $O(n \log_2 n)$ Vergleiche. Ein Sortierverfahren, welches im schlechtesten Fall diese Schranke nicht überschreitet, ist *heapsort*. Um eine Folge von Objekten zu sortieren, muß eine totale Ordnung auf der Menge der Objekte definiert sein. In den folgenden Beispielen werden natürliche Zahlen sortiert.

Heapsort verwendet eine spezielle Form von Suchbäumen. Ein *Heap* ist ein binärer, geordneter, eckenbewerteter Wurzelbaum mit folgenden Eigenschaften:

- (1) Die kleinste Bewertung jedes Teilbaumes befindet sich in dessen Wurzel.
- (2) Ist h die Höhe des Heaps, so haben die Ecken der Niveaus 0 bis $h - 2$ alle den Eckengrad 2.
- (3) Ist das letzte Niveau nicht voll besetzt, so sind die Ecken von links nach rechts fortschreitend lückenlos angeordnet.

Eigenschaft (1) impliziert, daß die Wurzel des Baumes die kleinste aller Bewertungen trägt. Heapsort funktioniert so, daß zunächst ein Heap erzeugt wird, dessen Bewertung aus den zu sortierenden Objekten besteht. Nun wird das kleinste Element entfernt; dieses befindet sich in der Wurzel. Die verbleibenden Ecken werden wieder zu einem Heap gemacht, der natürlich eine Ecke weniger hat. Danach steht das zweitkleinste Element in der neuen Wurzel und kann entfernt werden etc. Dadurch werden die Bewertungen der Ecken in aufsteigender Reihenfolge sortiert.

Ein Heap mit n Ecken läßt sich sehr günstig in einem Feld $H[1..n]$ abspeichern. Die Bewertungen der Ecken des i -ten Niveaus stehen in der entsprechenden Reihenfolge in den Komponenten 2^i bis $2^{i+1} - 1$, wobei die Bewertungen des letzten Niveaus h in den Komponenten 2^h bis n stehen. Aus den Bedingungen (1) bis (3) folgt: $H[i] \leq H[2i]$ und $H[i] \leq H[2i+1]$ für alle i mit der Eigenschaft $2i \leq n$ bzw. $2i+1 \leq n$. Hat man umgekehrt ein Feld $H[1..n]$ mit dieser Eigenschaft gegeben, so entspricht dies genau einem Heap: Die Wurzel steht in $H[1]$ und die Nachfolger von Ecke i stehen in $H[2i]$ bzw. $H[2i+1]$, sofern $2i \leq n$ bzw. $2i+1 \leq n$ gilt. Abbildung 3.17 zeigt einen Heap und die entsprechende Darstellung in einem Feld.

Die Realisierung des oben beschriebenen Sortieralgorithmus erfordert zwei Operationen: eine zur Erzeugung eines Heaps und eine zur Korrektur eines Heaps, nachdem

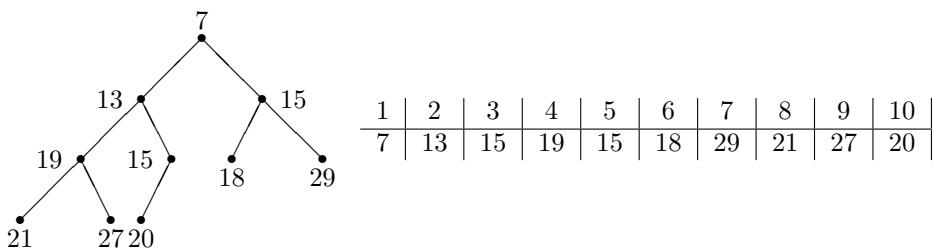


Abbildung 3.17: Ein Heap und die Darstellung in einem Feld

das kleinste Element entfernt worden ist. Für die zweite Operation werden die Komponenten $H[1]$ und $H[n]$ vertauscht. Danach steht das kleinste Element in $H[n]$, und der neue Heap entsteht in den ersten $n-1$ Komponenten. Da H vorher die Heapeigenschaften erfüllte, verletzt lediglich $H[1]$ die Bedingung (1). Um die Heapeigenschaften wieder herzustellen, wird nun $H[1]$ mit $H[2]$ und $H[3]$ verglichen und gegebenenfalls mit dem kleineren der beiden vertauscht. Falls eine Vertauschung notwendig war, wird genauso mit dieser Komponente verfahren; d.h. im allgemeinen wird $H[i]$ mit $H[2i]$ und $H[2i+1]$ verglichen, bis man bei einem Blatt angelangt ist oder keine Vertauschung mehr notwendig ist. Danach ist die Heapeigenschaft wieder hergestellt. Abbildung 3.18 zeigt eine Prozedur **absinken**, welche diese Korrektur durchführt.

Die Prozedur ist etwas allgemeiner gehalten, da sie auch zur Erzeugung eines Heaps benötigt wird: Es wird vorausgesetzt, daß der Heap im Feld H zwischen den Komponenten **anfang** und **ende** abgespeichert ist. Wird die Prozedur dazu verwendet, die Heapeigenschaft wiederherzustellen, so ist **anfang** immer gleich 1. Für den Heap wird folgende Datenstruktur verwendet:

```
Heap = array[1..max] of Inhaltstyp;
```

Der Eintrag, welcher die Heapeigenschaft eventuell verletzt (d.h. $H[\text{anfang}]$), wird in einer Hilfsvariablen **x** abgespeichert. An den Datentyp **Inhaltstyp** wird nur die Anforderung gestellt, daß die Werte sich vergleichen lassen (z.B. alle numerischen Datentypen). In der WHILE-Schleife wird zuerst festgestellt, ob der Eintrag $H[r]$ zwei Nachfolger hat und wenn ja, welcher der kleinere der beiden ist; dieser steht dann in $H[s]$. Ist nun **x** größer als $H[s]$, so wird $H[s]$ in $H[r]$ abgespeichert, und $H[s]$ wird betrachtet. Eine explizite Vertauschung der Einträge ist dabei nicht notwendig, da der ursprüngliche Wert von $H[\text{anfang}]$ immer beteiligt ist. Am Ende wird **x** an der korrekten Position abgelegt. Die WHILE-Schleife kann an zwei Stellen verlassen werden: zum einen, wenn ein Blatt im Heap erreicht wurde und zum anderen, wenn mitten im Heap die Heapeigenschaft erfüllt ist. Der letzte Fall wird mit Hilfe der **break**-Anweisung durchgeführt.

Bei der Heaperzeugung wird der Heap sukzessive von rechts nach links aufgebaut. Da bei einem Heap keine Bedingungen an die Blätter gestellt werden, erfüllt die zweite Hälfte der Komponenten von H trivialerweise die Heapeigenschaften. In jedem Schritt

```

procedure absinken(var H : Heap; anfang, ende : Integer);
var
  r, s : Integer;
  x : Inhaltstyp;
begin
  r := anfang;
  x := H[anfang];
  while 2*r <= ende do begin
    s := 2*r;
    if (s < ende) and (H[s] > H[s+1]) then
      s := s + 1;
    if x > H[s] then begin
      H[r] := H[s];
      r := s;
    end
    else
      break;
  end;
  H[r] := x;
end

```

Abbildung 3.18: Die Prozedur **absinken**

wird nun eine weitere Komponente hinzugenommen, und dann wird mit der Prozedur **absinken** die Heapeigenschaft wiederhergestellt; d.h. die Prozedur **absinken** wird beim Aufbau des Heaps $n/2$ -mal aufgerufen. Abbildung 3.19 zeigt die vollständige Prozedur **heapsort**.

Der Heap wird mit der Laufschleife in Zeile (1) erzeugt. In Zeile (3) wird das kleinste Element mit dem letzten Element des aktuellen Heaps vertauscht, und in Zeile (4) wird die Heapeigenschaft wieder hergestellt.

Abbildung 3.20 zeigt die Veränderungen des Suchbaumes beim Aufruf von **heapsort** am Beispiel der Zahlenfolge 17, 9, 3, 81, 25 und 11. Die Veränderungen erfolgen in den Zeilen (1),(3) und (4) und sind entsprechend markiert.

Abbildung 3.21 zeigt, wie sich diese Veränderungen im Heap H niederschlagen. Die fettgedruckten Zahlen am Ende des Heaps sind dabei schon sortiert.

Wie aufwendig ist **heapsort**? Dazu wird zuerst die Erzeugung des Heaps betrachtet, d.h. die Laufschleife in Zeile (1). Hierzu ist es notwendig, zuerst den Aufwand der Prozedur **absinken** zu bestimmen. Ein Durchlauf der **while**-Schleife hat den konstanten Aufwand $O(1)$. Bezeichne mit $A(i)$ die Anzahl der Durchläufe der **while**-Schleife beim Aufruf von **absinken(H, i, n)**. Dann ist $A(i)$ höchstens gleich der Anzahl der Niveaus, die das Objekt i absinken kann. Es sei h die Höhe des Heaps und s die Anzahl der Objekte in dem letzten Niveau. Die Gesamtzahl der Durchläufe beim Aufbau des Heaps ist

```

procedure heapsort(var H : Heap; n : Integer);
var
    i : Integer;
begin
    for i := (n div 2) downto 1 do (1)
        absinken(H,i, n);
    for i := n downto 2 do begin      (2)
        vertausche(H[1],H[i]);
        absinken(H,1,i - 1);          (3)
    end
end

```

Abbildung 3.19: Die Prozedur heapsort

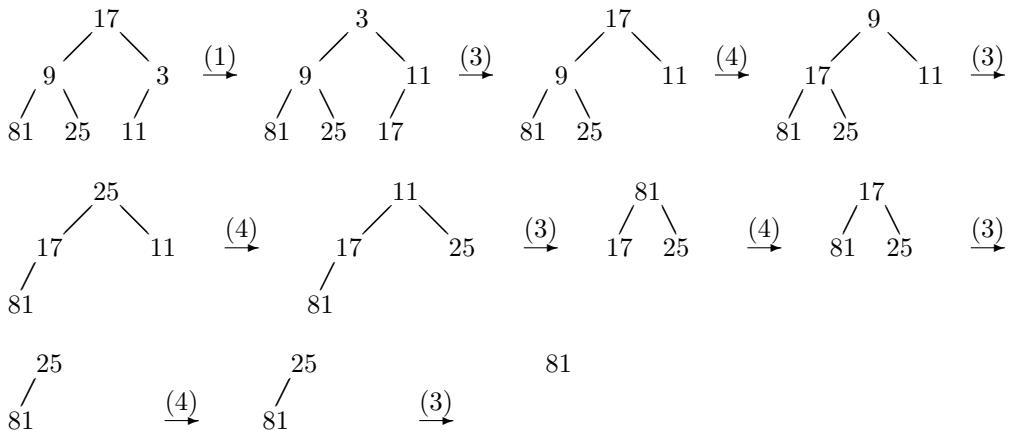


Abbildung 3.20: Veränderung des Suchbaumes beim Aufruf von heapsort

höchstens

$$\sum_{i=1}^{n-s} A(i),$$

denn die s Ecken auf dem letzten Niveau sinken nicht ab.

In den Niveaus $0, \dots, h-1$ sind zusammen $2^h - 1$ Objekte. Ferner gilt $0 \leq s \leq 2^h$. Das Objekt in der Wurzel kann maximal h Niveaus absinken, und die Objekte auf dem ersten Niveau können maximal $h-1$ Niveaus absinken. Allgemein gilt für $l = 0, 1, \dots, h$, daß die Ecken im l -ten Niveau maximal $h-l$ Niveaus absinken können. Im l -ten Niveau befinden sich die 2^l Objekte $2^l + t$ für $t = 0, 1, \dots, 2^l - 1$. Somit ist $A(2^l + t) \leq h-l$ für $l = 0, 1, \dots, h$ und $t = 0, 1, \dots, 2^l - 1$. Das heißt, alle Objekte auf dem l -ten Niveau zusammen können maximal $2^l(h-l)$ Niveaus absinken.

	1	2	3	4	5	6
	17	9	3	81	25	11
(1)	3	9	11	81	25	17
(3)	17	9	11	81	25	3
(4)	9	17	11	81	25	3
(3)	25	17	11	81	9	3
(4)	11	17	25	81	9	3
(3)	81	17	25	11	9	3
(4)	17	81	25	11	9	3
(3)	25	81	17	11	9	3
(4)	25	81	17	11	9	3
(3)	81	25	17	11	9	3

Abbildung 3.21: Die Änderung des Heaps H

Daraus ergibt sich

$$\sum_{i=1}^{n-s} A(i) \leq \sum_{l=0}^{h-1} 2^l (h-l) = \sum_{l=1}^h 2^{h-l} l = 2^h \sum_{l=1}^h \frac{l}{2^l} \leq 2^h d \leq nd.$$

Die Konstante d ist der Grenzwert der konvergenten Reihe

$$\sum_{l=1}^{\infty} \frac{l}{2^l}.$$

Der Aufwand zur Erzeugung des Heaps ist somit $O(n)$. Um den Aufwand von `heapsort` zu bestimmen, muß man noch die Laufschleife (2) betrachten, die $(n-1)$ -mal durchlaufen wird. Dabei erfolgt der Aufruf `absinken(H, 1, i-1)`. Aus dem Obigen ergibt sich, daß das erste Objekt maximal h Niveaus absinken kann. Da $h \leq \log_2 n$ ist, hat jeder Durchlauf den Aufwand $O(\log_2 n)$. Insgesamt ergibt sich somit ein Aufwand von $O(n + n \log_2 n) = O(n \log_2 n)$.

Mit `heapsort` hat man einen Sortieralgorithmus, der die theoretische untere Schranke von $\tilde{O}(n \log n)$ Schritten im ungünstigsten Fall nicht überschreitet. Für praktische Zwecke sind unter Umständen Algorithmen vorzuziehen, die im schlechtesten Fall $O(n^2)$ Schritte benötigen, aber eine bessere mittlere Laufzeit haben. Interessant ist `heapsort` für den Fall, daß nur die kleinsten k Elemente sortiert werden sollen. In diesem Fall braucht man die Laufschleife in Zeile (2) nur k -mal zu durchlaufen. Es ergibt sich eine Laufzeit von $O(n + k \log n)$. Falls $k \leq n / \log n$ ist, so ist die Laufzeit des Algorithmus sogar linear.

3.5 Vorrang-Warteschlangen

Bei einer Rechenanlage, die im Mehrprogrammbetrieb arbeitet, wird die Reihenfolge des Zugangs zum Prozessor durch *Prioritäten* festgelegt. Jeder Auftrag bekommt je nach

Dringlichkeit eine Priorität zugeordnet. Hohe Priorität bedeutet hohe Dringlichkeit und niedrige Priorität niedrige Dringlichkeit. Aufträge mit hoher Priorität werden bevorzugt ausgeführt. Der *Scheduler* ist ein Teil des Betriebssystems, welcher den Zugang von Aufträgen zum Prozessor regelt. Die Vergabe von Prioritäten kann extern (d.h. vom Benutzer) oder intern (d.h. vom Betriebssystem) vorgenommen werden. Im letzten Fall werden unter anderem folgende Größen berücksichtigt:

- Größe der Speicheranforderung,
- Anzahl der offenen Files,
- Verhältnis Rechenzeit zu Ein-/Ausgabezeit.

Alle Aufträge befinden sich in einer Warteschlange, die nach Prioritäten geordnet ist. Der Auftrag in der Warteschlange mit der höchsten Priorität wird aus der Warteschlange entfernt und dem Prozessor zur Ausführung zugeführt. Viele Betriebssysteme entziehen nach einer gewissen vorgegebenen Zeitspanne dem Auftrag wieder den Prozessor (*pre-emptive scheduling*). Wurde der Auftrag in dieser Zeit nicht abgeschlossen, so wird eine neue Priorität für ihn bestimmt, und der Auftrag wird wieder in die Warteschlange eingefügt. Die neue Priorität richtet sich nach dem Verhalten des Auftrages während der vergangenen Zeitspanne und der alten Priorität. Um einen hohen Durchsatz von Aufträgen zu erzielen, ist es notwendig, daß der Verwaltungsaufwand gering gehalten wird. Dazu wird eine Datenstruktur für die Warteschlange benötigt, welche die verwendeten Operationen effizient unterstützt. Dies sind Auswählen und Entfernen des Auftrags mit höchster Priorität, Einfügen eines Auftrages mit gegebener Priorität, Ändern der Priorität eines Auftrages und Entfernen eines beliebigen Auftrages. Eine Datenstruktur mit diesen Operationen nennt man *Vorrang-Warteschlange*. Eine Realisierung von Vorrang-Warteschlangen basiert auf dem im letzten Abschnitt vorgestellten Suchbaum, dem Heap. Dazu wird folgende Datenstruktur vereinbart.

```
VorrangWarteschlange = record
    inhalt : Heap;
    ende : 1..max;
end
```

Der Typ **Inhaltstyp** innerhalb von **Heap** beschreibt die Daten der Elemente in der Warteschlange. Seine Struktur ist für das Folgende nicht von Bedeutung. Es wird lediglich vorausgesetzt, daß auf seinen Werten eine lineare Ordnung definiert ist. Im obigen Beispiel enthält **Inhaltstyp** neben auftragsspezifischen Informationen auch eine Komponente, nach der der Vorrang geregelt wird. Es ist dabei nicht relevant, ob die Einträge mit aufsteigender oder absteigender Priorität behandelt werden. Um die im letzten Abschnitt angegebenen Prozeduren verwenden zu können, wird im folgenden von absteigenden Prioritäten ausgegangen. Eine Realisierung der Vorrang-Warteschlange für das oben angegebene Beispiel des Schedulers läßt sich daraus leicht ableiten.

Das Element mit der niedrigsten Priorität steht immer in der ersten Komponente des Heaps. Eine Funktion zum Entfernen des Elements mit der niedrigsten Priorität unter

Beibehaltung der Heapeigenschaft ergibt sich direkt aus der im letzten Abschnitt angegebenen Prozedur `absinken`: Das erste Element wird entfernt, und der letzte Eintrag wird in die erste Komponente gespeichert. Dann wird das Feld `ende` um 1 erniedrigt, und die Prozedur `absinken(v.inhalt,1,ende)` wird aufgerufen. Abbildung 3.22 zeigt eine Realisierung der Funktion `entfernen`.

```
function entfernen(var v : VorrangWarteschlange) : Inhaltstyp;
var
    i : Integer;
begin
    if v.ende = 0 then
        exit('Die Vorrang-Warteschlange ist leer')
    else begin
        entfernen := v.inhalt[1];
        v.inhalt[1] := v.inhalt[v.ende];
        v.ende := v.ende - 1;
        absinken(v.inhalt,1,v.ende);
    end
end
```

Abbildung 3.22: Die Funktion entfernen

Die Prozedur `einfügen` fügt ein neues Element unter Beibehaltung der Heapeigenschaft in die Vorrang-Warteschlange ein. Das neue Element wird, sofern noch Platz vorhanden ist, zunächst am Ende des Feldes `inhalt` eingefügt. Die Priorität des Elementes wird dann mit der Priorität des Vorgängers im Heap verglichen. Hat der Vorgänger eine höhere Priorität, so werden die beiden Einträge vertauscht, und ein weiterer Vergleich erfolgt. Die Prozedur endet damit, daß ein Vorgänger eine niedrigere Priorität besitzt oder das neue Element in der Wurzel steht, d.h. die insgesamt niedrigste Priorität besitzt. Abbildung 3.23 zeigt eine Realisierung der Prozedur `einfügen`.

Eine Prozedur zur Änderung der Priorität eines Elements der Vorrang-Warteschlange benötigt die Position `k` des Elementes im Heap. Die Positionen der Elemente im Heap lassen sich mittels eines zusätzlichen Feldes verwalten. Wird die Priorität erhöht, so wird das Element mittels `absinken(v.inhalt,k,ende)` nach unten bewegt. Wird die Priorität erniedrigt, so muß das Element im Heap nach oben bewegt werden. Dies erfolgt ganz analog zu der Prozedur `einfügen`.

Eine Prozedur, welche ein beliebiges Element aus einer Vorrang-Warteschlange löscht, läßt sich ganz ähnlich realisieren. Dazu muß allerdings die Position `k` des Elementes im Heap bekannt sein. Zunächst wird der letzte Eintrag in die `k`-te Komponente des Feldes gespeichert, und das Feld `ende` wird um 1 erniedrigt. Analog zur letzten Prozedur wird dann die Heapeigenschaft wieder hergestellt.

Aus der im letzten Abschnitt durchgeföhrten Analyse folgt sofort, daß die worst case Komplexität aller Operationen für eine Vorrang-Warteschlange mit n Elementen $O(\log n)$ ist. Der im folgenden beschriebene Algorithmus von Prim zeigt eine Anwendung von Vorrang-Warteschlangen.

```

procedure einfügen(var v : VorrangWarteschlange;
                    x : Inhaltstyp);
var
    i : Integer;
begin
    if v.ende = max then
        exit('Die Vorrang-Warteschlange ist voll')
    else begin
        v.ende := v.ende + 1;
        i := v.ende;
        while i > 1 and x < v.inhalt[i div 2] do begin
            v.inhalt[i] := v.inhalt[i div 2];
            i := i div 2;
        end;
        v.inhalt[i] := x;
    end
end

```

Abbildung 3.23: Die Prozedur einfügen

3.6 Minimal aufspannende Bäume

Es sei G ein kantenbewerteter, zusammenhängender, ungerichteter Graph. Die Summe der Bewertungen der Kanten eines aufspannenden Baumes B nennt man die *Kosten* von B . Ein aufspannender Baum B von G heißt *minimal aufspannender Baum* von G , falls kein anderer aufspannender Baum B' von G existiert, dessen Kosten niedriger sind. Abbildung 3.24 zeigt einen kantenbewerteten Graphen und einen zugehörigen minimal aufspannenden Baum mit Kosten 10. Ein Graph kann mehrere minimal aufspannende Bäume haben. In Abbildung 3.27 ist ein weiterer minimal aufspannender Baum für den Graphen aus Abbildung 3.24 dargestellt.

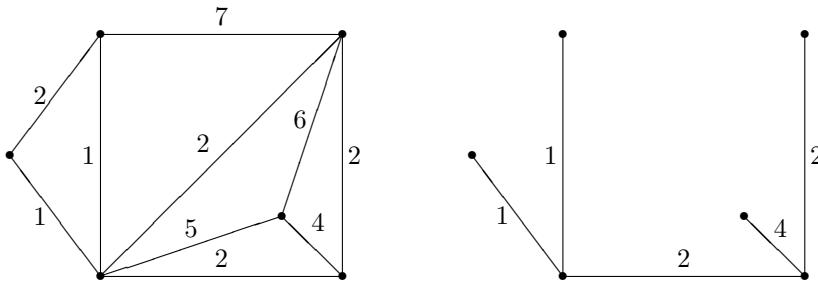


Abbildung 3.24: Ein Graph und ein zugehöriger minimal aufspannender Baum

Ziel dieses Abschnittes ist die Darstellung von effizienten Algorithmen zur Bestimmung minimal aufspannender Bäume. Die Korrektheit dieser Algorithmen beruht wesentlich

auf folgendem Satz:

Satz. Es sei G ein kantenbewerteter zusammenhängender Graph mit Eckenmenge E . Ferner sei U eine Teilmenge von E und (u, v) eine Kante mit minimalen Kosten mit $u \in U$ und $v \in E \setminus U$. Dann existiert ein minimal aufspannender Baum von G , der die Kante (u, v) enthält.

Beweis. Angenommen (u, v) liegt in keinem minimal aufspannenden Baum von G . Sei B irgendein minimal aufspannender Baum von G . Fügt man (u, v) in B ein, so erhält man einen geschlossenen Weg W in B , der (u, v) enthält. Da W Ecken aus U und $E \setminus U$ verwendet, muß es außer (u, v) noch eine Kante (u', v') in W geben, so daß $u' \in U$ und $v' \in E \setminus U$ ist. Dies folgt aus der Geschlossenheit von W . Entfernt man (u', v') aus B , so ist B wieder ein aufspannender Baum, der die Kante (u, v) enthält. Da dies nach Annahme kein minimal aufspannender Baum sein kann, muß die Bewertung von (u, v) echt größer sein als die von (u', v') . Dies widerspricht aber der Wahl von (u, v) . ■

Eine typische Anwendung, in der minimal aufspannende Bäume auftreten, ist die Planung von Kommunikationsnetzen. Zwischen n Orten ist ein Kommunikationsnetz so zu planen, daß je zwei verschiedene Orte – entweder direkt oder über andere Orte – durch Leitungen miteinander verbunden sind und sich Verzweigungspunkte des Netzes nur in den Orten befinden. Die Baukosten für eine Direktleitung zwischen zwei Orten, falls eine solche Leitung überhaupt möglich ist, seien bekannt. Gesucht ist das Leitungsnetz, dessen Gesamtkosten minimal sind. Die Ecken des Graphen repräsentieren die Orte, die Kanten die möglichen Verbindungen mit den entsprechenden Kosten. Ein minimal aufspannender Baum repräsentiert ein Netzwerk mit geringsten Kosten.

3.6.1 Der Algorithmus von Kruskal

Der Algorithmus von *J.B. Kruskal* erstellt für einen kantenbewerteten, zusammenhängenden Graphen G sukzessive einen minimal aufspannenden Baum B . Er gehört zu der Gruppe der Greedy-Algorithmen. Am Anfang ist B ein Wald mit der gleichen Eckenmenge wie G , der keine Kanten besitzt; d.h. B besteht nur aus isolierten Ecken. In jedem Schritt wird nun eine Kante in B eingefügt; diese vereinigt zwei Bäume aus B . Somit wird die Anzahl der Zusammenhangskomponenten von B in jedem Schritt um 1 verringert. Dazu werden die Kanten von G den Bewertungen nach sortiert. Diese Liste wird dann in aufsteigender Reihenfolge abgearbeitet. Verbindet eine Kante zwei Ecken aus verschiedenen Zusammenhangskomponenten von B , so wird sie in B eingefügt; ansonsten wird die Kante nicht verwendet. Dadurch wird sichergestellt, daß kein geschlossener Weg entsteht. Der Algorithmus endet, wenn B zusammenhängend ist; dann ist B ein Baum.

Abbildung 3.25 zeigt das Entstehen eines minimal aufspannenden Baumes für den Graphen aus Abbildung 3.24. Zuerst werden die zwei Kanten mit Bewertung 1 eingefügt. Von den Kanten mit Bewertung 2 können nur zwei eingefügt werden, denn sonst würde ein geschlossener Weg entstehen, da Anfangs- und Endknoten in der gleichen Zusammenhangskomponente liegen. Danach wird noch die Kante mit Bewertung 4 eingefügt.

Es bleibt, die Korrektheit des Algorithmus zu beweisen und eine Analyse der Laufzeit

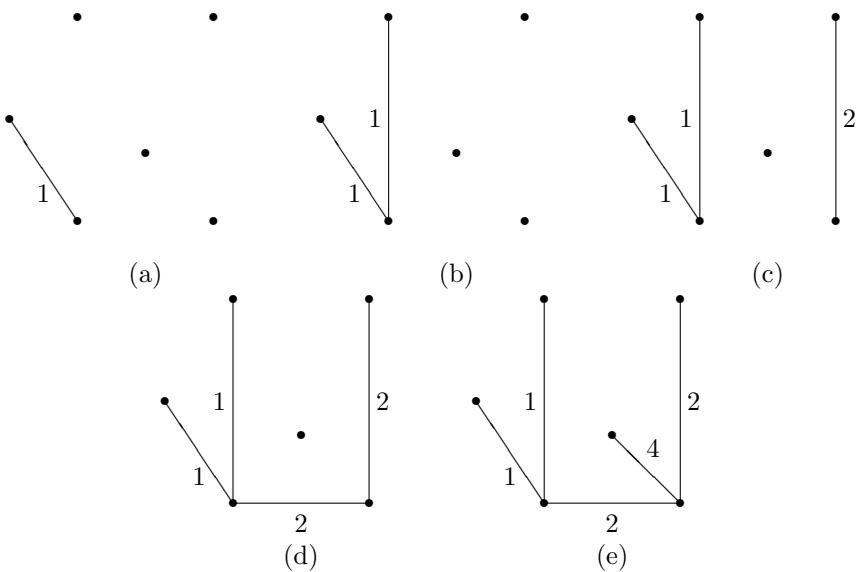


Abbildung 3.25: Eine Anwendung des Algorithmus von Kruskal

vorzunehmen. Die Korrektheit wird in folgendem Satz bewiesen:

Satz. Der Algorithmus von Kruskal bestimmt für einen zusammenhängenden kantenbewerteten Graphen einen minimal aufspannenden Baum.

Beweis. Der Beweis wird mittels vollständiger Induktion geführt. Der Algorithmus produziert sicherlich einen aufspannenden Baum B . Es wird nun gezeigt, daß B minimal ist. Die eingefügten Kanten seien mit k_1, k_2, \dots, k_{n-1} bezeichnet. Per Induktion wird nun gezeigt, daß es für jedes $j \in \{1, \dots, n-1\}$ einen minimal aufspannenden Baum T_j gibt, der die Kanten k_1, k_2, \dots, k_j enthält. Dann ist $T_{n-1} = B$, und B ist somit minimal. Die Aussage für $j = 1$ folgt direkt aus dem letzten Satz. Sei also $j > 1$ und T_j ein minimal aufspannender Baum, der k_1, \dots, k_j enthält. Falls k_{j+1} auch in T_j ist, so ist $T_{j+1} = T_j$. Falls k_{j+1} nicht in T_j liegt, so füge man k_{j+1} in T_j ein. Dadurch entsteht in T_j ein geschlossener Weg W . Da B ein Baum ist, sind nicht alle Kanten von W in B enthalten. Sei k eine solche Kante. Sei T_{j+1} der Baum, der aus T_j hervor geht, indem man k entfernt und k_{j+1} einfügt. Für T_{j+1} gilt:

$$\text{Kosten}(T_{j+1}) = \text{Kosten}(T_j) - \text{Bewertung}(k) + \text{Bewertung}(k_{j+1}).$$

Wäre die Bewertung von k echt kleiner als die Bewertung von k_{j+1} , so wäre k vor k_{j+1} eingefügt worden. Da dies nicht der Fall ist, gilt

$$\text{Bewertung}(k) \geq \text{Bewertung}(k_{j+1}).$$

Da die Kosten von T_j minimal sind, folgt aus obiger Gleichung, daß die Kosten von T_{j+1} gleich den Kosten von T_j sind. Somit ist T_{j+1} der gesuchte minimal aufspannende Baum, der k_1, \dots, k_{j+1} enthält. ■

Welchen Aufwand hat der Algorithmus von *Kruskal*? Obwohl nur $n - 1$ Kanten in B eingefügt werden, kann es sein, daß alle m Kanten von G betrachtet werden müssen. Dies ist der Fall, wenn die Kante mit der höchsten Bewertung zu dem minimal aufspannenden Baum gehört (vergleichen Sie Übungsaufgabe 21).

Der Aufwand, die m Kanten zu sortieren, ist, $O(m \log m)$. In jedem Schritt wird getestet, ob die Endpunkte in verschiedenen Zusammenhangskomponenten liegen. Dazu müssen die Zusammenhangskomponenten verwaltet werden. Eine einfache Möglichkeit dafür bildet ein Feld Z der Länge n , wobei n die Anzahl der Ecken von G ist. In diesem Feld wird für jede Ecke die Nummer einer Zusammenhangskomponente abgespeichert. Am Anfang ist $Z[i] = i$ für $i = 1, \dots, n$, d.h. alle Ecken sind isoliert. Der Test, ob zwei Ecken zur gleichen Zusammenhangskomponente gehören, besteht darin, ihre Werte in Z zu vergleichen; dies geschieht in konstanter Zeit. Wird eine Kante in B eingefügt, so müssen die beiden Zusammenhangskomponenten vereinigt werden. In Z werden dazu die Werte der Ecken der ersten Zusammenhangskomponente auf die Nummer der zweiten gesetzt. Im ungünstigsten Fall muß dazu das ganze Feld durchsucht werden. Das bedeutet pro eingefügter Kante einen Aufwand von $O(n)$. Da $n - 1$ Kanten eingefügt werden müssen, ergibt dies einen Aufwand von $O(n^2)$. Insgesamt ergibt sich ein Aufwand von $O(m \log m + n^2)$. Da ein Graph mit n Ecken höchstens $n(n - 1)/2$ Kanten hat, ist $O(\log m) = O(\log n)$. Somit ist der Aufwand gleich $O(m \log n + n^2)$.

Im folgenden wird eine effizientere Implementierung mit Aufwand $O(m \log n)$ vorgestellt. Dazu werden die zeitkritischen Stellen analysiert und geeignete Datenstrukturen verwendet. Die zeitkritischen Stellen sind zum einen das Sortieren der Kanten und zum anderen das Testen auf Kreisfreiheit. Für das Sortieren bieten sich die im Abschnitt 3.4 behandelten Heaps an, denn in vielen Fällen wird es nicht notwendig sein, alle Kanten zu betrachten. In diesen Fällen ist eine vollständige Sortierung nicht notwendig. Die Kanten werden in einem Heap abgelegt, und in jedem Durchlauf wird die Kante mit kleinstem Wert entfernt und die Heapeigenschaft wieder hergestellt. Das Erzeugen des Heaps hat den Aufwand $O(m)$ und das Wiederherstellen der Heapeigenschaft $O(\log m)$. Werden also l Durchläufe gemacht, so bedeutet dies einen Aufwand von $O(m + l \log m) = O(m + l \log n)$. Ist l von der Größenordnung $O(n)$, so bedeutet dies einen Aufwand von $O(m + n \log n)$, und ist l von der Größenordnung $O(m)$, so ergibt sich ein Aufwand von $O(m \log n)$.

Im folgenden wird nun eine Darstellung für Zusammenhangskomponenten von B vorgestellt, mit der sich effizient geschlossene Wege auffinden lassen. Zusätzlich zu dem Feld Z werden noch zwei weitere Felder A und L der Länge n angelegt. In dem Feld A wird die momentane Anzahl der Ecken der entsprechenden Zusammenhangskomponente abgespeichert. $A[i]$ ist die Anzahl der Ecken der Zusammenhangskomponente mit

der Nummer i , und $A[Z[i]]$ ist die Anzahl der Ecken der Zusammenhangskomponente, die die Ecke i enthält. Das zweite Feld L dient dazu, die Ecken einer Zusammenhangskomponente schneller aufzufinden. Dazu wird eine *zyklische Adreßkette* gebildet. Der Eintrag $L[i]$ gibt die Nummer einer Ecke an, die in derselben Zusammenhangskomponente wie i liegt. Dadurch kann man sich, von i startend die Adreßkette durchlaufend, alle Ecken der Zusammenhangskomponente von i beschaffen. Bilden zum Beispiel die Ecken 2, 7, 9 und 14 eine Zusammenhangskomponente, so ist

$$L[2] = 7, \quad L[7] = 9, \quad L[9] = 14, \quad L[14] = 2.$$

Besteht eine Zusammenhangskomponente aus z Ecken, so kann man diese auch in z Schritten auffinden. Wird eine neue Kante eingefügt, so müssen die beiden Adreßketten zu einer Adreßkette vereinigt werden. Sind i und j die Enden der neuen Kante, so müssen die Werte von $L[i]$ und $L[j]$ vertauscht werden. Somit erfordert das Verschmelzen von zwei Zusammenhangskomponenten nur konstante Zeit. Es muß noch das Feld Z geändert werden, um weiterhin die schnelle Abfrage auf geschlossene Wege zu unterstützen. Dazu müssen die Einträge der Ecken einer Zusammenhangskomponente alle geändert werden. Dies ist sehr einfach: Mit Hilfe des Feldes A kann die kleinere der beiden Zusammenhangskomponenten ermittelt werden, und mit Hilfe des Feldes L findet man direkt alle beteiligten Ecken. Hat die kleinere der beiden Zusammenhangskomponenten z Ecken, so ist der Aufwand nur noch $O(z)$ im Vergleich zu $O(n)$ in der einfachen Version.

Wie wirken sich diese neuen Felder auf den Gesamtaufwand aus? Es sei s eine natürliche Zahl mit

$$2^{s-1} \leq n < 2^s.$$

Dann gilt $s \leq \log_2 n + 1$. Beim Einfügen der letzten Kante hat die kleinste Zusammenhangskomponente höchstens $n/2$ Ecken. Dies ist der Fall, wenn die beiden letzten Komponenten gleichviele Ecken haben. Diese beiden Komponenten bestehen im ungünstigsten Fall wiederum aus jeweils zwei gleichgroßen Komponenten; d.h. für deren Entstehen müßten maximal $n/4 + n/4 = n/2$ Einträge geändert werden. Somit werden für die letzte Kante maximal $n/2$ Einträge in Z geändert, für die beiden Kanten davor ebenfalls $n/2$ Änderungen, für die vier Kanten davor ebenfalls $n/2$ Änderungen etc. Wegen

$$n \leq 2^s - 1 = 2^0 + 2^1 + \cdots + 2^{s-1}$$

müssen insgesamt maximal $(n/2)s \leq (n/2)(\log_2 n + 1)$ Einträge in Z geändert werden. Dies ergibt einen Gesamtaufwand von $O(n \log n)$.

Diese Realisierung des Algorithmus von Kruskal hat also im ungünstigsten Fall den Aufwand $O(m \log n + n \log n) = O((m+n) \log n)$. Da die Anzahl m der Kanten in zusammenhängenden Graphen mindestens $n-1$ ist, ist der Aufwand gleich $O(m \log n)$.

Abbildung 3.26 zeigt die Prozedur `kruskal`, welche den Algorithmus von Kruskal realisiert. Der Funktionsaufruf `B.initBaum(G)` erzeugt einen Baum `B` ohne Kanten, der die gleichen Eckenmenge wie `G` hat. Mit `B.einfügen(a, e)` wird die Kante `(a, e)` in `B` eingefügt. Der Funktionsaufruf `H.erzeugeHeap(G)` erzeugt einen Heap, welcher die Kanten von `G` mit ihren Bewertungen enthält. Mit `(a, e) := H.entfernen` wird die Kante

```

procedure kruskal(G : Graph; var b : Baum);
var
  A, L, Z : array[1..max] of Integer;
  H : heap of Kanten;
  u, e, i : Integer;
begin
  B.initBaum(G);
  for jede Ecke i do begin
    A[i] := 1; L[i] := i; Z[i] := i
  end;
  H.erzeugeHeap(G);
  repeat
    (u,e) := H.entfernen;
    if Z[u] ≠ Z[e] then
      begin
        B.einfügen(u,e);
        if A[Z[u]] < A[Z[e]] then begin
          min := u; max := e;
        end
        else begin
          min := e; max := u;
        end;
        A[Z[max]] := A[Z[max]] + A[Z[min]];
        i := min;
        repeat
          Z[i] := Z[max];
          i := L[i]
        until i = min;
        vertausche(L[u],L[e]);
      end
    until B.kantenzahl = n - 1;
end

```

Abbildung 3.26: Der Algorithmus von Kruskal

(a,e) mit der niedrigsten Bewertung aus dem Heap entfernt, und die Heapeigenschaft wird wieder hergestellt.

Es gibt noch effizientere Darstellungen für die Zusammenhangskomponenten, die auf speziellen Bäumen basieren. Die Implementierung solcher Bäume und die Analyse der Laufzeit ist jedoch sehr aufwendig.

3.6.2 Der Algorithmus von Prim

Der Algorithmus von Kruskal baut einen minimal aufspannenden Baum dadurch auf, daß die Zusammenhangskomponenten eines Waldes nacheinander vereinigt werden. Im Algorithmus von *R. Prim* bilden die ausgewählten Kanten zu jedem Zeitpunkt einen Baum B . Man beginnt mit einer beliebigen Ecke des Graphen, und durch Hinzufügen von Kanten entsteht dann der gesuchte Baum. Bei der Auswahl der Kanten verfährt man wie folgt: Sei U die Menge der Ecken des Baumes B und E die Menge der Ecken des Graphen. Man wähle unter den Kanten (u, v) , deren Anfangsseite u in U und deren Endecke v in $E \setminus U$ liegen, diejenige mit der kleinsten Bewertung aus. Diese wird dann in B eingefügt. Ferner wird v in U eingefügt. Dieser Schritt wird solange wiederholt, bis $U = E$ ist.

Der Algorithmus von Prim vermeidet das Sortieren der Kanten. Allerdings ist die Auswahl der Kanten komplizierter. Abbildung 3.27 zeigt das Entstehen des minimal aufspannenden Baumes aus Abbildung 3.24.

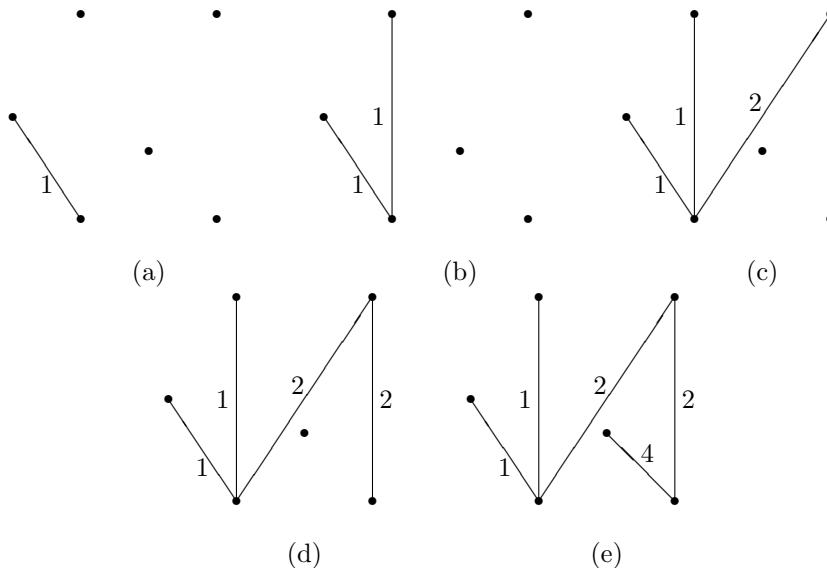


Abbildung 3.27: Eine Anwendung des Algorithmus von Prim

Abbildung 3.28 gibt die Grobstruktur des Algorithmus von Prim an. Die Verwaltung der Menge U und die Auswahl der Kanten kann mit Hilfe eines Feldes erfolgen. In diesem Feld speichert man für jede Ecke $e \in E \setminus U$ die kleinste Bewertung unter den Kanten (e, u) mit $u \in U$. Ferner speichert man auch noch die Endecke dieser Kante ab. Die Ecken aus U werden in diesem Feld besonders markiert. Die Auswahl der Kanten erfolgt nun leicht mittels dieses Feldes. Nachdem eine Kante (e, u) ausgewählt wurde, wird die Ecke e als zu U gehörend markiert. Ferner wird für jeden Nachbarn v von e mit $v \in E \setminus U$ überprüft, ob die Bewertung der Kante (v, e) kleiner ist als der im Feld vermerkte Wert. Ist dies der Fall, so wird das Feld entsprechend geändert.

```

procedure prim(G : Graph; var B : Baum);
var
    U : set of Integer;
    u, e : Integer;
begin
    B.initBaum(G);
    U := {1};
    while U.anzahl ≠ n do begin
        Sei (e,u) die Kante aus G mit der kleinsten Bewertung,
        so daß u ∈ U und e ∈ E\U
        B.einfügen(e,u);
        U.einfügen(e);
    end
end

```

Abbildung 3.28: Der Algorithmus von Prim

Der Korrektheitsbeweis des Algorithmus von Prim erfolgt analog zum Beweis der Korrektheit des Algorithmus von Kruskal. Die Laufzeit der Prozedur `prim` lässt sich leicht angeben. Es werden insgesamt $n - 1$ Iterationen durchgeführt, denn es wird jeweils eine Ecke in U eingefügt. Die Auswahl der Kante und das Aktualisieren des Feldes haben zusammen einen Aufwand von $O(n)$. Somit ergibt sich ein Gesamtaufwand von $O(n^2)$. Die Menge U kann auch durch eine Vorrang-Warteschlange dargestellt werden. Dann sind insgesamt $O(m)$ Operationen notwendig. Die Gesamlaufzeit beträgt dann $O(m \log n)$. Unter Verwendung von speziellen Datenstrukturen, sogenannten *Fibonacci-Heaps*, kann der Gesamtaufwand verringert werden. Es ergibt sich dann eine Laufzeit von $O(m + n \log n)$.

Ein Vergleich der beiden vorgestellten Algorithmen ist schwierig, da diese sehr stark von der gewählten Datenstruktur abhängen. Der Algorithmus von Prim, basierend auf Feldern, ist vorzuziehen, falls die Anzahl m der Kanten etwa die Größenordnung von n^2 hat, d.h. es liegt ein „dichter“ Graph vor.

Der Algorithmus von Fredman und Tarjan zur Bestimmung minimal aufspannender Bäume basiert ebenfalls auf Fibonacci-Heaps. Die erzielte Laufzeit ist fast linear in m , der Anzahl der Kanten. Aber bis heute ist kein Algorithmus für dieses Problem mit einer Laufzeit von $O(m)$ bekannt.

3.7 Literatur

Die Verwendung von Bäumen im Compilerbau ist ausführlich in dem Standardwerk [2] beschrieben. Eine detaillierte Behandlung von Suchbäumen findet man in [126]. Der Algorithmus von Huffman wurde zum erstenmal 1952 veröffentlicht [70]. Eine ausführliche Darstellung von Verfahren zur Kompression von Texten findet man in [11]. Der Heapsort-Algorithmus wurde von Williams entwickelt [124] und von Floyd [40] verbessert. Eine detaillierte Analyse von Vorrang-Warteschlangen findet man in [78]. Die Originalarbeiten zu den beiden Algorithmen zur Bestimmung minimal aufspannender Bäume sind [84] und [106]. Der Algorithmus von Fredman und Tarjan ist in [44] beschrieben. Einen Vergleich von Algorithmen zur Bestimmung minimal aufspannender Bäume für verschiedene Klassen von Graphen findet man in [4].

3.8 Aufgaben

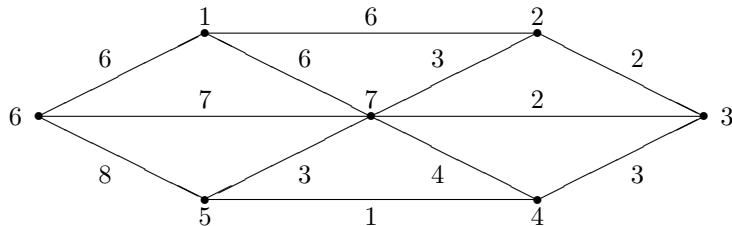
1. Beweisen Sie folgende Aussagen:
 - a) Ein zusammenhängender Graph ist genau dann ein Baum, wenn $n > m$ gilt.
 - b) Ein zusammenhängender Graph enthält genau dann einen einzigen geschlossenen Weg, wenn $n = m$ gilt.
- ** 2. Erstellen Sie eine Charakterisierung von ungerichteten Graphen mit der Eigenschaft, daß je zwei Ecken genau einen gemeinsamen Nachbarn haben. Was kann man über die Anzahl der Ecken bzw. Kanten dieser Graphen sagen?
3. Es sei d_1, d_2, \dots, d_n eine Folge von n positiven natürlichen Zahlen, deren Summe $2n - 2$ ist. Zeigen Sie, daß es einen Baum mit n Ecken gibt, so daß die d_i die Eckengrade sind.
4. Es sei B ein Baum, dessen Komplement \overline{B} nicht zusammenhängend ist. Beweisen Sie folgende Aussagen:
 - a) \overline{B} besteht aus genau zwei Zusammenhangskomponenten.
 - b) Eine der Zusammenhangskomponenten ist ein vollständiger Graph.

Bestimmen Sie die Struktur der zweiten Zusammenhangskomponente und die von B !

5. Bestimmen Sie die Anzahl der Ecken eines Binärbaumes mit b Blättern, bei dem jede Ecke den Ausgangsgrad 2 oder 0 hat.
6. Es sei G ein Wurzelbaum mit Eckenmenge E . Ferner sei B die Menge der Blätter und $anz(e)$ die Anzahl der Nachfolger der Ecke e . Beweisen Sie folgende Gleichung:

$$|B| = \sum_{e \in E \setminus B} (anz(e) - 1) + 1.$$

7. Beweisen Sie die folgende Aussage: Ein Binärbaum der Höhe h hat höchstens 2^h Blätter.
8. Ein gerichteter Graph G heißt *quasi stark zusammenhängend*, wenn es zu je zwei Ecken e, f von G eine Ecke v gibt, so daß e und f von v aus erreichbar sind. Beweisen Sie folgende Aussage: Ein gerichteter Graph ist genau dann quasi stark zusammenhängend, wenn er eine Wurzel besitzt.
9. Entwerfen Sie eine Datenstruktur für ein hierarchisches Dateisystem. Schreiben Sie eine Prozedur, welche die Namen aller Dateien im angegebenen Verzeichnis und allen darunterliegenden Unterverzeichnissen ausgibt.
10. Schreiben Sie einen Algorithmus für das Löschen von Einträgen in binären Suchbäumen.
11. Schreiben Sie eine Prozedur, die alle Objekte eines Suchbaumes in aufsteigender Reihenfolge ausgibt.
12. Finden Sie einen Binärbaum mit zehn Ecken, dessen Höhe minimal bzw. maximal ist.
- ** 13. Es sei B ein Huffman-Baum. Es seien a und b Symbole, so daß a auf einem höheren Niveau liegt als b . Beweisen Sie, daß die Wahrscheinlichkeit von b kleiner gleich der von a ist.
14. Bestimmen Sie einen Präfix-Code für die sechs Zeichen a, b, c, d, e und f mit den Häufigkeiten 12%, 32%, 4%, 20%, 16% und 16%. Bestimmen Sie die mittlere Codewortlänge.
15. In Abschnitt 3.3 wurde eine Datenstruktur vorgestellt, welche speziell für Binärbäume, wie sie im Huffmann-Algorithmus vorkommen, geeignet ist. Entwerfen Sie einen Algorithmus, der auf dieser Datenstruktur basiert und für einen gegebenen Binärbaum den entsprechenden Präfix-Code erzeugt.
16. Finden Sie für den untenstehenden ungerichteten, bewerteten Graphen zwei verschiedene minimal aufspannende Bäume. Verwenden Sie dazu den Algorithmus von Kruskal! Geben Sie die Kosten eines minimal aufspannenden Baumes an!



- * 17. Entwerfen Sie einen Algorithmus für das folgende Problem: Gegeben ist ein kan-tenbewerteter, zusammenhängender, ungerichteter Graph G und eine Kante k von G . Unter den aufspannenden Bäumen von G , die k enthalten, ist derjenige mit den geringsten Kosten zu bestimmen.

18. Schreiben Sie einen Algorithmus zur Bestimmung eines maximal aufspannenden Baumes eines ungerichteten kantenbewerteten Graphen.
- * 19. Es sei G ein ungerichteter kantenbewerteter Graph und k_1, k_2, \dots, k_s ein Weg W in G . Ist k_i die Kante von W mit der kleinsten Bewertung, so nennt man die Bewertung von k_i den *Querschnitt* von W . Für ein Paar e, f von Ecken von G ist der *Durchsatz* gleich dem größten Querschnitt eines Weges zwischen e und f . Das heißt, ist D der Durchsatz der Ecken e und f , so gibt es einen Weg W zwischen e und f mit folgender Eigenschaft: Die Bewertung jeder Kante von W ist mindestens gleich D . Entwerfen Sie einen Algorithmus zur Bestimmung des Durchsatzes aller Paare von Ecken eines ungerichteten kantenbewerteten Graphen (Hinweis: Es sei B ein maximal aufspannender Baum von G und W ein Weg in B mit Anfangscke e und Endcke f . Beweisen Sie, daß der Durchsatz von e, f in G gleich dem Querschnitt von W ist).
20. Beweisen Sie, daß folgender Algorithmus einen minimal aufspannenden Baum B für einen zusammenhängenden ungerichteten Graphen mit Kanten k_1, \dots, k_m bestimmt.

```

Initialisiere B mit {k1};
for i := 2 to m do begin
    B := B.einfügen(ki);
    if B enthält einen geschlossenen Weg W then begin
        sei k die Kante auf W mit der höchsten Bewertung;
        B.entfernen(k);
    end
end

```

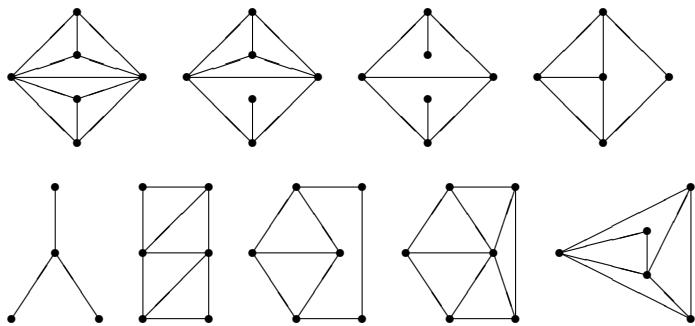
21. Konstruieren Sie einen ungerichteten bewerteten Graphen, in dem die Kante mit der höchsten Bewertung in jedem minimal aufspannenden Baum liegt.
22. Es sei G ein zusammenhängender bewerteter Graph, dessen Kantenbewertungen alle verschieden sind. Zeigen Sie, daß G einen eindeutigen minimal aufspannenden Baum besitzt.
23. Vergleichen Sie die Laufzeiten der Algorithmen von Prim und Kruskal zur Bestimmung von minimal aufspannenden Bäumen für Graphen aus der Klasse $G_{n,p}$ mit zufällig erzeugten Kantenbewertungen. Variieren Sie dabei sowohl n als auch p . Für welche Werte von p ist der Algorithmus von Prim vorzuziehen?
24. Betrachten Sie die Klasse der ungerichteten bewerteten Graphen, deren Kantenbewertungen ganze Zahlen aus der Menge $\{1, 2, \dots, C\}$ sind. Geben Sie einen effizienten Algorithmus zur Bestimmung von minimal aufspannenden Bäumen für diese Graphen an. Bestimmen Sie die Laufzeit Ihres Algorithmus in Abhängigkeit von n, m und C .
- * 25. Es sei G ein ungerichteter bewerteter Graph und B ein minimal aufspannender Baum von G mit Kosten C . Beweisen Sie folgende Aussage: Addiert man eine

beliebige reelle Zahl b zu der Bewertung jeder Kante von G , so ist B immer noch ein minimal aufspannender Baum, und die Kosten sind $C + (n - 1)b$. Diese Tatsache führt zu einem Algorithmus zur Bestimmung eines minimal aufspannenden Baumes. Man wählt eine Kante $k = (e, f)$ mit der kleinsten Bewertung b und subtrahiert b von jeder Kantens Bewertung. Danach hat die Kante k die Bewertung 0 und gehört zu dem minimal aufspannenden Baum. Die Ecken e und f werden zu einer Ecke verschmolzen. Die Nachbarn dieser neuen Ecken sind die Nachbarn der Ecken e und f . Entstehen dabei Doppelkanten, so wählt man die mit der geringsten Bewertung. Danach wird wieder eine Kante mit der kleinsten Bewertung ausgewählt etc. Beweisen Sie, daß auf diese Art ein minimal aufspannender Baum entsteht. (Hinweis: Vergleichen Sie diesen Algorithmus mit dem von Kruskal.)

- * 26. Es sei G ein ungerichteter bewerteter Graph und B ein minimal aufspannender Baum von G . In G wird eine neue Ecke und zu dieser inzidente Kanten eingefügt. Geben Sie einen effizienten Algorithmus an, welcher einen minimal aufspannenden Baum für den neuen Graphen bestimmt.

Kapitel 4

Suchverfahren in Graphen



In diesem Kapitel werden Suchstrategien für Graphen behandelt. Sie bilden die Grundlage für viele graphentheoretische Algorithmen, in denen es notwendig ist, alle Ecken oder Kanten eines Graphen systematisch zu durchlaufen. In Kapitel 3 wurde dies bereits für Suchbäume abgehandelt. Die beiden hier vorgestellten Suchstrategien *Tiefensuche* und *Breitensuche* verallgemeinern diese Techniken, so daß sie auf beliebige Graphen anwendbar sind. Es wird zunächst das Grundprinzip der Tiefensuche vorgestellt. Danach werden Anwendungen der Tiefensuche auf gerichtete Graphen diskutiert: Topologische Sortierungen, Bestimmung der starken Zusammenhangskomponenten und Bestimmung des transitiven Abschluß und der transitiven Reduktion.

Daran schließen sich Anwendungen auf ungerichtete Graphen an: Bestimmung der Zusammenhangskomponenten und der Blöcke eines Graphen. Ferner wird eine Anwendung der Tiefensuche in der Bildverarbeitung diskutiert. Danach wird die Breitensuche mit einer Anwendung auf ungerichtete Graphen vorgestellt: die Erkennung von bipartiten Graphen. Im letzten Abschnitt wird eine Variante der Tiefensuche vorgestellt, welche nur wenig Speicherplatz verbraucht und auf implizite Graphen anwendbar ist.

4.1 Einleitung

Bei vielen graphentheoretischen Problemen ist es notwendig, alle Ecken oder Kanten eines Graphen zu durchsuchen. Beispiele hierfür sind die Fragen, ob ein ungerichteter Graph zusammenhängend ist oder ob ein gerichteter Graph einen geschlossenen Weg enthält. Für eine systematische Vorgehensweise gibt es mehrere Alternativen. Die in diesem Kapitel vorgestellten Verfahren durchsuchen einen Graphen, ausgehend von einer Startecke, und besuchen alle von dieser Startecke aus erreichbaren Ecken. Der Verlauf der Suche spiegelt sich im sogenannten *Erreichbarkeitsbaum* wider. Dies ist ein Wurzelbaum, dessen Wurzel die Startecke ist. Der Erreichbarkeitsbaum besteht aus den von der Startecke aus erreichbaren Ecken zusammen mit den Kanten, welche zu diesen Ecken führten (bei ungerichteten Graphen werden die Kanten hin zur neu besuchten Ecke gerichtet). Ein Graph kann für eine gegebene Startecke verschiedene Erreichbarkeitsbäume besitzen. Alle haben die gleiche Eckenmenge, sie unterscheiden sich jedoch in der Kantenmenge.

Die Grundlage aller Suchverfahren bildet der im folgenden beschriebene *Markierungsalgorithmus*. Die Suche startet bei der Startecke und bewegt sich über die Kanten in den Graphen. Bei gerichteten Graphen müssen dabei die durch die Kanten vorgegebenen Richtungen eingehalten werden. Alle besuchten Ecken werden markiert. Damit die Suche nicht in Schleifen gerät, werden nur unmarkierte Ecken besucht. Die Grundstruktur des Markierungsalgorithmus sieht wie folgt aus:

1. Markiere die Startecke.
2. Solange es noch Kanten von markierten zu unmarkierten Ecken gibt, wähle eine solche Kante und markiere die Endecke dieser Kante.

Die verwendeten Kanten (mit entsprechender Richtung) bilden den Erreichbarkeitsbaum. Da in jedem Schritt eine Ecke markiert wird, terminiert der Algorithmus für endliche Graphen nach endlich vielen Schritten. Es folgt direkt, daß am Ende die Menge der markierten Ecken mit der Menge der erreichbaren Ecken übereinstimmt. Bei entsprechender Realisierung des Auswahlschrittes ergibt sich eine Laufzeit von $O(n + m)$.

Die aus dieser Grundstruktur abgeleiteten Suchverfahren unterscheiden sich in der Art und Weise, wie im zweiten Schritt die Kanten ausgewählt werden. Hierfür gibt es im wesentlichen zwei Varianten. Diese führen zur Tiefen- bzw. zur Breitensuche. Beide Verfahren lassen sich sowohl auf gerichtete als auch auf ungerichtete Graphen anwenden.

4.2 Tiefensuche

Die Tiefensuche (*depth-first-search*) versucht, die am weitesten von der Startecke entfernten Ecken so früh wie möglich zu besuchen. Das Verfahren heißt Tiefensuche, da man jeweils so tief wie möglich in den Graphen hineingeht. Dazu wählt man im zweiten Schritt eine Kante aus, deren Anfangsseite die zuletzt markierte Ecke ist. Führen die Kanten der zuletzt markierten Ecke alle zu schon markierten Ecken, so betrachtet man die zuletzt markierten Ecken in umgekehrter Reihenfolge, bis eine entsprechende Kante

gefunden wird. Dieses Auswahlprinzip der Kanten läßt sich auf zwei verschiedene Arten realisieren.

Da die gerade besuchte Ecke wieder Startpunkt für die weitere Suche ist, bietet sich eine Realisierung mit Hilfe einer rekursiven Prozedur an. Die Reihenfolge, in der die markierten Ecken nach verwendbaren Kanten durchsucht werden, entspricht der Reihenfolge, in der die Ecken markiert werden; d.h. die zuletzt markierte Ecke wird immer als erstes verwendet. Diese Reihenfolge kann mittels eines Stapels realisiert werden: Die oberste Ecke wird jeweils nach verwendbaren Kanten untersucht, die noch unmarkierten Nachbarn dieser Ecke werden markiert und oben auf den Stapel abgelegt.

Zunächst wird die rekursive Realisierung vorgestellt. Für die Markierung einer Ecke reicht eine Boolesche Variable aus. Für viele Anwendungen ist die Reihenfolge, in der die Ecken besucht werden, wichtig. Aus diesem Grund werden im folgenden die besuchten Ecke mit aufsteigenden Nummern (*Tiefensuchenummern*) markiert. Die Nummer 0 bedeutet, daß die Ecke noch nicht besucht worden ist. Die Nummern werden in einem Feld abgespeichert und spiegeln die Reihenfolge wider, in der die Ecken besucht werden.

Die Startecke bekommt die Tiefensuchenummer 1, die nächste besuchte Ecke die 2 usw. Von der Startecke geht man zu einer benachbarten unmarkierten Ecke und markiert diese. Von dort aus wird das gleiche Verfahren fortgesetzt, bis man zu einer Ecke kommt, deren Nachbarn alle schon markiert sind. In diesem Fall geht man auf dem Weg, der zu dieser Ecke führte, zurück, bis man zu einer Ecke gelangt, die noch einen unmarkierten Nachbarn hat, und startet von neuem. Das Verfahren endet damit, daß alle von der Startecke aus erreichbaren Ecken markiert sind. Dies müssen natürlich nicht alle Ecken des Graphen sein.

Sollen alle Ecken eines Graphen besucht werden, so sind weitere Tiefensuchedurchgänge notwendig. Dabei wird eine noch unmarkierte Ecke zur neuen Startecke. In der oben beschriebenen Form numeriert das Verfahren lediglich die Ecken des Graphen. Die Numerierung hängt dabei von der Reihenfolge ab, in der die Nachbarn besucht werden. Ferner ergeben auch verschiedene Startecken verschiedene Numerierungen. Im folgenden wird immer die Ecke 1 als Startecke verwendet.

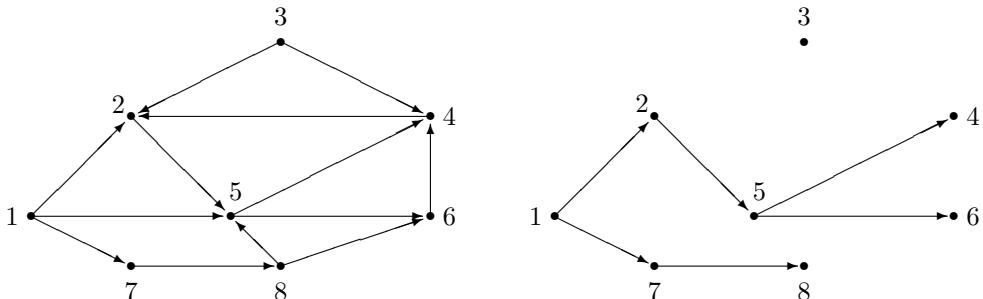


Abbildung 4.1: Ein gerichteter Graph mit einem Tiefensuchewald

Die Ecken des Graphen aus Abbildung 4.1 werden bei der Tiefensuche mit Startecke 1 in der Reihenfolge 1, 2, 5, 4, 6, 7, 8 besucht. Hierbei werden die Nachbarn einer Ecke in aufsteigender Reihenfolge ihrer Eckenummern besucht. Diese Reihenfolge wird auch

bei den weiteren Beispielen in diesem und den folgenden Kapiteln angewendet. Die Ecke 3 ist von Ecke 1 aus nicht erreichbar.

Die im folgenden vorgestellte Grundversion der Tiefensuche bewirkt die oben beschriebene Numerierung der Ecken. Die Nummern werden in einem Feld `TSNummer` abgespeichert. In der Prozedur `suchen` erfolgt zunächst die Initialisierung dieses Feldes, dann der Aufruf der Prozedur `tiefensuche` für die Startecke und danach ein weiterer Aufruf von `tiefensuche` für jede noch nicht besuchte Ecke. Die Eintragung einer Tiefensuchenummer durch die Prozedur `tiefensuche` in das globale Feld `TSNummer` dient gleichzeitig als Markierung einer Ecke. Eine Ecke ist genau dann noch unbesucht, wenn der entsprechende Eintrag in `TSNummer` gleich 0 ist. Abbildung 4.2 zeigt die Prozedur `suchen`.

```

var  TSNummer : array[1..max] of Integer;
      zähler : Integer;
procedure suchen(G : Graph);
var
      i : Integer;
begin
  Initialisiere TSNummer und zähler mit 0;
  for jede Ecke i do
    if TSNummer[i] = 0 then
      tiefensuche(i);
end

```

Abbildung 4.2: Die Prozedur `suchen`

Abbildung 4.3 zeigt eine Realisierung der Prozedur `tiefensuche`. Die Vergabe der Nummern erfolgt über eine globale Variable `zähler`, die mit 0 initialisiert wird.

```

procedure tiefensuche(i : Integer);
var
      j : Integer;
begin
  zähler := zähler + 1;
  TSNummer[i] := zähler;
  for jeden Nachbar j von i do
    if TSNummer[j] = 0 then
      tiefensuche(j);
end

```

Abbildung 4.3: Die Prozedur `tiefensuche`

Wendet man die Prozedur `suchen` auf den Graphen aus Abbildung 4.1 an, so ergibt sich folgende Aufrufhierarchie:

```

tiefensuche(1)
    tiefensuche(2)
        tiefensuche(5)
            tiefensuche(4)
            tiefensuche(6)
        tiefensuche(7)
            tiefensuche(8)
    tiefensuche(3)

```

Nach dem Aufruf von `suchen` für den Graphen aus Abbildung 4.1 sieht das Feld `TSNummer` wie folgt aus:

Eckennummer	1	2	3	4	5	6	7	8
TSNummer	1	2	8	4	3	5	6	7

Die Komplexitätsanalyse der Tiefensuche hängt stark von der gewählten Speicherungsart ab. Man beachte, daß die Prozedur `tiefensuche` genau einmal für jede Ecke aufgerufen wird, denn jeder Aufruf von `tiefensuche(i)` verändert als erstes `TSNummer[i]`, und ein Aufruf erfolgt nie für eine Ecke j , für die `TSNummer[j]` schon verändert wurde. Bei jedem Aufruf von `tiefensuche` werden alle erreichbaren Ecken untersucht. Bei einem gerichteten Graphen wird dadurch jede Kante maximal einmal und bei einem ungerichteten Graphen maximal zweimal bearbeitet. Somit ist die Komplexität dieses Teils $O(m)$. Da alle n Einträge des Feldes `TSNummer` geändert werden, ergibt sich insgesamt die Komplexität $O(n + m)$. Diese Analyse setzt voraus, daß man auf die Nachbarn $N(i)$ einer Ecke i in der Zeit $O(g(i))$ zugreifen kann. Dies ist zum Beispiel bei der Adjazenzliste der Fall. Verwendet man hingegen die Adjazenzmatrix, muß man jeweils alle Einträge einer Zeile untersuchen, um alle Nachbarn zu bestimmen. In diesem Falle ergibt sich die Komplexität $O(n^2)$.

Eine nichtrekursive Realisierung der Tiefensuche basiert auf einem Stapel. Abbildung 4.4 zeigt eine solche Realisierung. Bei einem Stapel werden neue Einträge oben eingefügt und auch von dort wieder entfernt. Jede neu markierte Ecke wird oben auf dem Stapel abgelegt. Am Anfang enthält der Stapel nur die Startecke. In jedem Durchgang der `while`-Schleife wird die oberste Ecke vom Stapel entfernt, und die unbesuchten Nachbarn dieser Ecke werden oben auf dem Stapel abgelegt. Die Tiefensuchenummern können nicht als Markierungen verwendet werden, da diese erst beim Entfernen der Ecken vom Stapel vergeben werden. In der dargestellten Variante werden die Markierungen in einem Booleschen Feld verwaltet.

Man beachte, daß die Ecken in den beiden Realisierungen nicht in der gleichen Reihenfolge besucht werden. Besucht man in der nichtrekursiven Variante die Nachbarn nicht in der Reihenfolge, wie sie in der Adjazenzliste stehen, sondern genau in umgekehrter Reihenfolge, so ergibt sich die gleiche Numerierung der besuchten Ecken. Die dargestellte nichtrekursive Version der Tiefensuche besucht nur die von der Startecke aus erreichbaren Ecken. Durch eine kleine Erweiterung erreicht man, daß alle Ecken besucht werden. Im letzten Abschnitt dieses Kapitels werden noch zwei weitere Varianten der Tiefensuche vorgestellt.

```

var TSNr : array[1..max] of Integer;
    Besucht : array[1..max] of Boolean;
procedure tiefensuche(G : Graph; startecke : Integer);
var
    i, j, zähler : Integer;
    S : stapel of Integer;
begin
    Initialisiere Besucht mit false und zähler mit 0;
    Besucht[startecke] := true;
    S.einfügen(startecke)
    while S ≠ ∅ do begin
        i := S.entfernen;
        zähler := zähler + 1;
        TSNr[i] := zähler;
        for jeden Nachbar j von i do
            if Besucht[j] = false then begin
                Besucht[j] := true;
                S.einfügen(j);
            end
        end
    end
end

```

Abbildung 4.4: Die Prozedur tiefensuche

4.3 Anwendung der Tiefensuche auf gerichtete Graphen

Die Kanten des gerichteten Graphen, die bei der Tiefensuche zu noch nicht besuchten Ecken führen, nennt man *Baumkanten*. Die Baumkanten bilden eine Menge von Erreichbarkeitsbäumen, den sogenannten *Tiefensuchewald*. Diejenigen Ecken, für die die Prozedur **tiefensuche** direkt von der Prozedur **suchen** aufgerufen wird, bilden die Wurzeln. Es ist leicht ersichtlich, daß die Baumkanten keinen geschlossenen Weg bilden können, denn sonst würde eine Baumkante zu einer schon besuchten Ecke führen.

Die Tiefensuche unterteilt die Kanten eines gerichteten Graphen in zwei Mengen: Baumkanten und Kanten, die zu schon besuchten Ecken führen. Die letzteren werden nochmal in drei Gruppen aufgeteilt. Eine Kante, die von einer Ecke e zu einer schon markierten Ecke f führt, heißt:

Rückwärtskante, falls f im Tiefensuchewald ein Vorgänger von e ist;

Vorwärtskante, falls f im Tiefensuchewald ein Nachfolger von e ist und

Querkante, falls f weder Nachfolger noch Vorgänger von e im Tiefensuchewald ist.

Abbildung 4.5 zeigt den Tiefensuchebaum des Graphen aus Abbildung 4.1 und jeweils

eine Rückwärtskante $(4, 2)$, eine Vorwärtskante $(1, 5)$ und eine Querkante $(3, 4)$. Die letzten drei Kanten sind gestrichelt dargestellt.

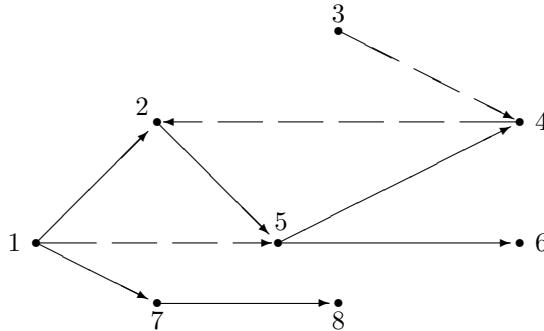


Abbildung 4.5: Rückwärts-, Vorwärts- und Querkante in einem gerichteten Graphen

Die Tiefensuchenummern der Ecken geben Aufschluß über die Art einer Kante. Es sei (e, f) eine Kante eines gerichteten Graphen. Ist $\text{TSNummer}[e] < \text{TSNummer}[f]$, so bedeutet dies, daß e vor f besucht wurde. In der Zeit zwischen dem Aufruf von `tiefensuche(e)` und dem Ende von diesem Aufruf werden alle von e aus erreichbaren unmarkierten Ecken besucht. Diese werden Nachfolger von e im Tiefensuchebaum. Also ist auch f ein Nachfolger von e . Somit ist (e, f) eine Baumkante, falls der erste Besuch von f über die Kante (e, f) erfolgte, und andernfalls eine Vorwärtskante. Ist $\text{TSNummer}[e] > \text{TSNummer}[f]$, so wurde f vor e besucht. In diesem Fall ist (e, f) eine Rückwärts- oder Querkante, je nachdem, ob f ein Vorgänger von e im Tiefensuchewald ist oder nicht. Somit gilt folgendes Lemma:

Lemma. Es sei G ein gerichteter Graph, auf den die Tiefensuche angewendet wurde. Für eine Kante $k = (e, f)$ von G gilt:

1. $\text{TSNummer}[e] < \text{TSNummer}[f]$ genau dann, wenn k eine Baum- oder eine Vorwärtskante ist.
2. $\text{TSNummer}[e] > \text{TSNummer}[f]$ genau dann, wenn k eine Rückwärts- oder eine Querkante ist.

Die Prozedur `tiefensuche` kann leicht abgeändert werden, so daß auch zwischen Rückwärts- und Querkanten unterschieden wird (vergleichen Sie Aufgabe 3).

Bei der Tiefensuche werden alle von der Startecke aus erreichbaren Ecken besucht. Somit kann mittels wiederholter Tiefensuche der transitive Abschluß eines gerichteten Graphen bestimmt werden. Die Tiefensuche wird dazu n -mal durchgeführt. Bei der Verwendung der Adjazenzliste ergibt sich ein Aufwand von $O(n(n + m))$.

Der in Kapitel 2 vorgestellte Algorithmus hatte einen Aufwand von $O(n^3)$. Für Graphen mit wenigen Kanten (d.h. $m \ll n^2$) ist somit der auf der Tiefensuche basierende Algorithmus bei Verwendung der Adjazenzliste effizienter. In den folgenden Abschnitten

werden Anwendungen der Tiefensuche zur Lösung von graphentheoretischen Problemen vorgestellt.

4.4 Kreisfreie Graphen und topologische Sortierung

Ein gerichteter Graph heißt *kreisfrei (azyklisch)*, falls er keinen geschlossenen Weg enthält. Einen solchen Graphen nennt man auch *DAG (directed acyclic graph)*. Jeder gerichtete Baum ist kreisfrei, aber die Umkehrung gilt nicht. Abbildung 4.6 zeigt links einen kreisfreien Graphen, welcher kein gerichteter Baum ist, und rechts einen gerichteten Graphen, der nicht kreisfrei ist.

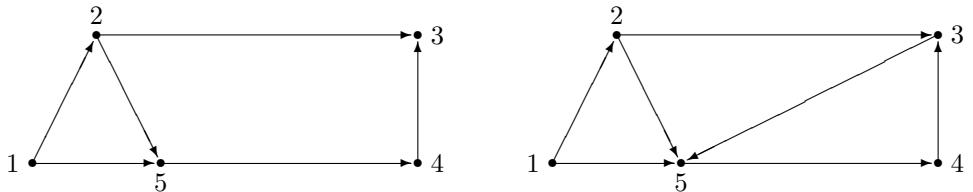


Abbildung 4.6: Ein kreisfreier und ein nicht kreisfreier Graph

Mit Hilfe der Tiefensuche kann man leicht feststellen, ob ein gerichteter Graph kreisfrei ist. Findet man eine Rückwärtskante, so liegt sicherlich ein geschlossener Weg vor. Auch die Umkehrung gilt: Gibt es einen geschlossenen Weg, so existiert auch eine Rückwärtskante. Dazu betrachte man einen gerichteten Graphen mit einem geschlossenen Weg. Es sei f die Ecke auf diesem geschlossenen Weg, auf die die Tiefensuche zuerst trifft. Von f aus werden nun alle erreichbaren Ecken besucht; somit auch die Vorgängerecke e von f auf dem geschlossenen Weg. Die Kante (e, f) ist also eine Rückwärtskante.

Wie stellt man fest, ob eine Rückwärtskante existiert? Nach dem obigen Lemma gilt $\text{TSNummer}[e] > \text{TSNummer}[f]$ für eine Rückwärtskante (e, f) . Neben dieser Bedingung muß noch sichergestellt sein, daß f im Tiefensuchewald ein Vorgänger von e ist. Dies bedeutet, daß f während des Aufrufs `tiefensuche(e)` besucht wurde und nicht später. Im letzten Fall ist (e, f) eine Querkante. Dazu verwaltet der Tiefensuchealgoritmus ein Boolesches Feld `Verlassen` der Länge n . Das Feld wird im Hauptprogramm mit `false` initialisiert. Ist die Tiefensuche für eine Ecke e beendet, so wird `Verlassen[e]` auf `true` gesetzt. In der Prozedur `tiefensuche` in Abbildung 4.3 muß dazu lediglich am Ende die Zeile

```
Verlassen[i] := true;
```

eingefügt werden. Stößt man nun während des Aufrufs `tiefensuche(e)` auf einen schon besuchten Nachbarn f , und hat `Verlassen[f]` noch den Wert `false`, so ist (e, f) ei-

ne Rückwärtskante. Da die Tiefensuchenummern in diesem Fall nicht mehr explizit benötigt werden, wird das Feld `TsNummer` durch das Boolesche Feld `Besucht` ersetzt. In diesem wird nur noch vermerkt, ob eine Ecke schon besucht wurde. Im Hauptprogramm werden die Felder `Besucht` und `Verlassen` mit `false` initialisiert. Die Prozedur `kreisfrei` entspricht ansonsten der Prozedur `suchen` aus dem letzten Abschnitt. Die Prozedur `tiefensuche` wurde durch die Prozedur `kreisfrei` ersetzt. Die beiden Prozeduren sind in Abbildung 4.7 dargestellt.

```

var Besucht, Verlassen : array[1..max] of Boolean;
procedure kreisfrei(G : G-Graph);
var
    i : Integer;
begin
    Initialisiere Besucht und Verlassen mit false;
    for jede Ecke i do
        if Besucht[i] = false then
            kreissuchen(i);
            exit('Kreisfrei');
    end

    procedure kreissuchen(i : Integer);
    var
        j : Integer;
    begin
        Besucht[i] := true;
        for jeden Nachfolger j von i do
            if Besucht[j] = false then
                kreissuchen(j)
            else
                if Verlassen[j] = false then
                    exit('Nicht kreisfrei');
                Verlassen[i] := true;
    end

```

Abbildung 4.7: Die Prozeduren `kreisfrei` und `kreissuchen`

Wendet man das Verfahren auf den rechten Graphen aus Abbildung 4.6 an, so stellt man fest, daß die Kante (4,3) eine Rückwärtskante ist. Die Zeitkomplexität für den Test auf Kreisfreiheit ist die gleiche wie für die allgemeine Tiefensuche. Im folgenden wird eine Anwendung von kreisfreien Graphen im Compilerbau beschrieben.

4.4.1 Rekursion in Programmiersprachen

Rekursion ist ein wichtiges Konzept in Programmiersprachen. Aber nicht alle Programmiersprachen unterstützen rekursive Definitionen. In Fortran 77 und Cobol sind z.B. rekursive Prozeduren nicht möglich. Auch rekursive Datenstrukturen werden nicht von

allen Programmiersprachen unterstützt. In diesen Fällen muß der Compiler überprüfen, ob die Definitionen rekursionsfrei sind. Bei Prozeduren unterscheidet man zwischen *direkter Rekursion* (eine Prozedur ruft sich selbst auf) und *indirekter Rekursion* (eine Prozedur P_1 ruft die Prozedur P_2 auf, P_2 ruft P_3 auf etc. und P_n ruft P_1 auf). Um indirekte Rekursion festzustellen, wird ein gerichteter Graph vom Compiler gebildet. Die Ecken dieses Graphen sind die Prozeduren, und eine Kante von P_i nach P_j bedeutet, daß P_j direkt von P_i aufgerufen wird. Die Kreisfreiheit dieses Graphen ist gleichbedeutend mit der Abwesenheit von indirekter Rekursion. Eine Schlinge in diesem Graphen zeigt eine direkte Rekursion an.

4.4.2 Topologische Sortierung

Mit kreisfreien Graphen lassen sich hierarchische Strukturen darstellen. Als Beispiel sei die Planung der Fertigung eines komplexen Werkstückes beschrieben. Die gesamte Fertigung besteht aus n Teilarbeiten T_i . Für den Beginn jeder Teilarbeit ist der Abschluß einiger anderer Teilarbeiten Voraussetzung. Dies kann mittels eines kreisfreien Graphen dargestellt werden. In diesem Graphen bilden die T_i die Ecken, und eine Kante von T_i nach T_j bedeutet, daß T_i vor Beginn von T_j beendet sein muß. Dieser Graph ist kreisfrei, ansonsten kann das Werkstück nicht gefertigt werden. Gesucht ist nun eine Reihenfolge der n Teilarbeiten unter Beachtung der Nebenbedingungen. Eine solche Reihenfolge nennt man eine *topologische Sortierung*.

Eine topologische Sortierung eines kreisfreien gerichteten Graphen ist eine Numerierung der Ecken mit Nummern von 1 bis n , so daß für alle Kanten die Nummer der Anfangsseite kleiner ist als die der Endseite. Die folgende Tabelle zeigt die Numerierung einer topologischen Sortierung des in Abbildung 4.6 links dargestellten Graphen:

Eckennummer	1	2	3	4	5
Sortierungsnummer	1	2	5	4	3

Es kann für einen kreisfreien gerichteten Graphen verschiedene topologische Sortierungen geben. Der im folgenden beschriebene Algorithmus beweist, daß jeder kreisfreie gerichtete Graph eine topologische Sortierung besitzt. Der Algorithmus basiert auf der Tiefensuche. Die Sortierungsnummern werden dabei in absteigender Form vergeben, d.h. in der Reihenfolge $n, n - 1, \dots, 1$. Die Vorgehensweise ist die, daß eine Ecke eine Sortierungsnummer bekommt, falls alle ihre Nachfolger schon eine haben. Dadurch wird die Eigenschaft, daß Kanten immer in Richtung höherer Nummern zeigen, gewahrt. Nach dem Aufruf von `tiefensuche(i)` werden alle von i aus erreichbaren Ecken besucht, d.h. am Ende des Aufrufs kann i eine Sortierungsnummer bekommen. Die Sortierungsnummern werden in einem Feld `ToSoNummer` der Länge n abgespeichert. Dieses Feld wird zuvor mit 0 initialisiert. Abbildung 4.8 zeigt eine Realisierung der Prozedur `tsprozedur`. Die Variable `nummer` enthält immer die als nächstes zu vergebende Sortierungsnummer.

Die Prozedur `tsprozedur` wird in der Prozedur `topsort` für noch unbesuchte Ecken aufgerufen. Abbildung 4.9 zeigt die Prozedur `topsort`. Hier erfolgt auch die Initialisierung der Variablen.

```

procedure tsprozedur(i : Integer);
var
  j : Integer;
begin
  Besucht[i] := true;
  for jeden Nachfolger j von i do
    if Besucht[j] = false then
      tsprozedur(j);
  ToSoNummer[i] := nummer;
  nummer := nummer - 1;
end

```

Abbildung 4.8: Die Prozedur tsprozedur

```

var ToSoNummer : array[1..max] of Integer;
  Besucht : array[1..max] of Boolean;
  nummer : Integer;
procedure topsort(G : G-Graph);
var
  i : Integer;
begin
  Initialisiere Besucht mit false und ToSoNummer mit 0;
  nummer := n;
  for jede Ecke i do
    if Besucht[i] = false then
      tsprozedur(i);
end

```

Abbildung 4.9: Die Prozedur topsort

Die Prozedur `topsort` erzeugt eine topologische Sortierung, falls sie auf kreisfreie gerichtete Graphen angewendet wird. Der Grund hierfür liegt darin, daß es in kreisfreien Graphen keine Rückwärtskanten gibt. Trifft man innerhalb von `tsprozedur(i)` auf eine schon besuchte Ecke j , so ist die entsprechende Kante eine Vorwärts- oder Querkante. In jedem Fall ist der Aufruf von `tsprozedur(j)` aber schon abgeschlossen. Somit hat j schon eine topologische Sortierungsnummer, und diese ist höher als die, die i bekommen wird. Wendet man die Prozedur `topsort` auf den in Abbildung 4.6 links dargestellten Graphen an, so ergibt sich folgende Aufrufhierarchie für die Prozedur `tsprozedur`:

```

tsprozedur(1)
  tsprozedur(2)
    tsprozedur(3)
    tsprozedur(5)
      tsprozedur(4)

```

Was passiert, wenn die Prozedur `topsort` auf Graphen angewendet wird, welche nicht kreisfrei sind? Es wird ebenfalls eine Numerierung produziert, welche aber nicht einer topologischen Sortierung entspricht. Die Prozedur `tsprozedur` läßt sich leicht dahingehend erweitern, daß geschlossene Wege erkannt werden. Dies ist der Fall, wenn man auf eine Rückwärtskante trifft. Diese führt zu einer Ecke j , welche schon besucht wurde (d.h. `TSNummer[j] = true`) und die noch keine topologische Sortierungsnummer hat (d.h. `ToSoNummer[j] = 0`). Somit muß nur die IF-Anweisung in der Prozedur `tsprozedur` durch folgende Anweisung ersetzt werden:

```

if Besucht[j] = false then
    tsprozedur(j)
else
    if ToSoNummer[j] = 0 then
        exit('Nicht kreisfrei');
    
```

Mit dieser Änderung kann die Prozedur `topsort` auf beliebige gerichtete Graphen angewendet werden. Enthält der Graph einen geschlossenen Weg, so wird dieser erkannt und eine entsprechende Ausgabe erzeugt. Andernfalls wird eine topologische Sortierung erzeugt. Verwendet man die Adjazenzlistendarstellung für den gerichteten Graphen, so ist die Komplexität von `topsort` gleich $O(n + m)$.

4.5 Starke Zusammenhangskomponenten

Ein gerichteter Graph heißt *stark zusammenhängend*, falls es für jedes Paar e, f von Ecken sowohl einen Weg von e nach f als auch von f nach e gibt. Die Eckenmenge E eines gerichteten Graphen kann in disjunkte Teilmengen E_1, \dots, E_s zerlegt werden, so daß die von den E_i induzierten Untergraphen stark zusammenhängend sind. Dazu bilde man folgende Relation: $e, f \in E$ sind äquivalent, falls es einen Weg von e nach f und einen Weg von f nach e gibt. Diese Relation ist symmetrisch und transitiv. Die von den Äquivalenzklassen dieser Relation induzierten Untergraphen sind stark zusammenhängend. Man nennt sie die *starken Zusammenhangskomponenten*. Ein stark zusammenhängender Graph besteht also nur aus einer starken Zusammenhangskomponente. Abbildung 4.10 zeigt links einen gerichteten Graphen und rechts die drei starken Zusammenhangskomponenten dieses Graphen.

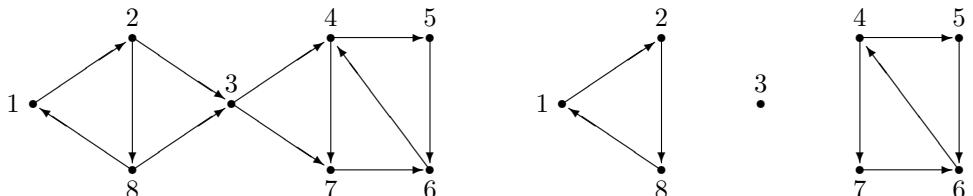


Abbildung 4.10: Ein gerichteter Graph und seine starken Zusammenhangskomponenten

Jede Ecke eines gerichteten Graphen gehört somit zu genau einer starken Zusammenhangskomponente. Es kann aber Kanten geben, die zu keiner Komponente gehören. Diese verbinden Ecken in verschiedenen Zusammenhangskomponenten. Das Auffinden der starken Zusammenhangskomponenten kann mit Hilfe der Tiefensuche vorgenommen werden. Die Ecken werden dabei in der Reihenfolge der Tiefensuche besucht und auf einem Stapel abgelegt. Die Ablage auf dem Stapel erfolgt sofort, wenn eine Ecke zum ersten Mal besucht wird. Nachdem die Tiefensuche eine Ecke w verläßt, sind die Ecken, die oberhalb von w auf dem Stapel liegen, gerade die Ecken, die von w aus neu besucht wurden. Diejenige Ecke einer starken Zusammenhangskomponente, welche während der Tiefensuche zuerst besucht wurde, nennt man die *Wurzel* der starken Zusammenhangskomponente. Im Beispiel aus Abbildung 4.10 sind dies die Ecken 1, 3 und 4, falls man die Tiefensuche bei Ecke 1 startet.

Von der Wurzel einer starken Zusammenhangskomponente sind alle Ecken der Zusammenhangskomponente erreichbar. Somit liegen nach dem Verlassen einer Wurzel w alle Ecken der gleichen Zusammenhangskomponente oberhalb von w auf dem Stapel. Oberhalb von w können aber auch noch andere Ecken liegen, nämlich die Ecken der starken Zusammenhangskomponenten, die von w aus erreichbar sind. In dem Graphen aus Abbildung 4.10 sind z.B. alle Ecken von der Ecke 1 aus erreichbar, d.h. beim Verlassen der Ecke 1 liegen die Ecken aller drei starken Zusammenhangskomponenten auf dem Stapel. Die Tiefensuche ist allerdings in diesem Fall schon für die Wurzeln von zwei der drei starken Zusammenhangskomponenten beendet. Man geht deshalb folgendermaßen vor: Jedesmal beim Verlassen einer Wurzel w werden alle Ecken oberhalb von w und einschließlich w von dem Stapel entfernt. Dadurch erreicht man, daß beim Verlassen einer Wurzel einer starken Zusammenhangskomponente die zugehörigen Ecken oben auf dem Stapel liegen.

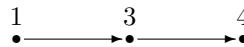


Abbildung 4.11: Der Strukturgraph des Graphen aus Abbildung 4.10

Dazu betrachte man folgenden *Strukturgraphen* \hat{G} von G . Die Eckenmenge \hat{E} von \hat{G} ist die Menge aller starken Zusammenhangskomponenten von G . Sind $E_i, E_j \in \hat{E}$, so gibt es eine Kante von E_i nach E_j , falls es Ecken $e \in E_i$ und $f \in E_j$ gibt, so daß in G eine Kante von e nach f existiert. Der Graph \hat{G} ist kreisfrei, denn die starken Zusammenhangskomponenten, die auf einem geschlossenen Weg lägen, würden eine einzige starke Zusammenhangskomponente bilden. Abbildung 4.11 zeigt den Strukturgraph des Graphen aus Abbildung 4.10. Dabei sind die Ecken mit den Nummern der entsprechenden Wurzeln markiert.

Da der Strukturgraph kreisfrei ist, besitzt er eine topologische Sortierung. Die starken Zusammenhangskomponenten erscheinen in umgekehrter Reihenfolge ihrer topologischen Sortierungsnummern vollständig oben auf dem Stapel und können dann entfernt werden. Von einer starken Zusammenhangskomponente kann es nur Kanten zu starken Zusammenhangskomponenten mit höheren topologischen Sortierungsnummern geben. Dies nutzt man aus, um beim Verlassen einer Ecke zu erkennen, ob es sich um die

Wurzel einer starken Zusammenhangskomponente handelt. Im folgenden wird gezeigt, daß eine Ecke w genau dann die Wurzel einer starken Zusammenhangskomponente ist, wenn weder w noch eine Ecke oberhalb von w eine Quer- oder Rückwärtskante zu einer Ecke v besitzt, die unter w auf dem Stapel liegt. Abbildung 4.12 zeigt die Grobstruktur einer entsprechenden rekursiven Prozedur.

```

procedure sZhKprozedur(i : Integer);
begin
    Lege i auf dem Stapel ab;
    for alle unbesuchten Nachbarn j von i do
        sZhKprozedur(j);
    if weder Ecke i noch eine Ecke oberhalb von i besitzt eine
        Rückwärts- oder Querkante zu einer Ecke unterhalb
        von i in dem Stapel then
        Entferne i und alle oberhalb von i liegenden Ecken aus
            dem Stapel;
end

```

Abbildung 4.12: Die Grobstruktur des Algorithmus zur Bestimmung der starken Zusammenhangskomponenten

Lemma. Die Prozedur `sZhKprozedur` bestimmt alle von i aus erreichbaren starken Zusammenhangskomponenten.

Beweis. Der Beweis erfolgt durch vollständige Induktion nach l , der Anzahl der Ecken in dem zugehörigen Strukturgraphen. Zunächst wird der Fall $l = 1$ betrachtet. Dann ist G stark zusammenhängend, und zwischen je zwei Ecken existiert ein Weg. Es sei e die erste Ecke, welche die in der Prozedur angegebene Bedingung der IF-Anweisung erfüllt; d.h. weder e noch eine Ecke oberhalb von e besitzt eine Quer- oder Rückwärtskante zu einer Ecke, die unter e auf dem Stapel liegt. Angenommen $e \neq i$. Da G stark zusammenhängend ist, gibt es einen Weg W von e nach i . Da die Prozedur `sZhKprozedur` den Graphen gemäß der Tiefensuche durchsucht, sind in diesem Moment alle von e aus erreichbaren Ecken im Stapel. Insbesondere sind also alle Ecken von W im Stapel. Sei nun w die erste Ecke auf W , welche nicht oberhalb von e im Stapel liegt (eventuell ist $w = i$), und v der Vorgänger von w auf diesem Weg. Somit wurde w vor v besucht. Nach dem im Abschnitt 4.3 bewiesenen Lemma ist (v, w) eine Rückwärts- oder eine Querkante. Da die Bedingung der IF-Anweisung für e erfüllt ist, ergibt dies einen Widerspruch. Somit ist $e = i$, und es werden alle Ecken ausgegeben, d.h. das Lemma ist für $l = 1$ bewiesen.

Sei nun $l > 1$. Es sei u die erste Ecke innerhalb eines Blattes des Strukturgraphen, welche die Prozedur `sZhKprozedur` erreicht. Da der Strukturgraph kreisfrei ist, wird immer ein Blatt erreicht. Es sei e wieder die erste Ecke, welche die in der Prozedur angegebene Bedingung der IF-Anweisung erfüllt. Wieder sind in diesem Moment alle von e aus erreichbaren Ecken im Stapel. Da von jeder Ecke des Graphen die Ecken von mindestens einem Blatt im Strukturgraphen erreichbar sind, liegt auch u im Stapel. Angenommen

$e \neq u$. Oberhalb von u liegen alle Ecken der starken Zusammenhangskomponente, welche u enthält. Von dieser kann keine andere Zusammenhangskomponente erreicht werden.

Analog zum ersten Teil beweist man, daß e nicht oberhalb von u im Stapel liegen kann. Somit liegt e unterhalb von u und die Prozedur **sZhKprozedur** hat schon u verlassen. Also gibt es eine Rückwärts- oder eine Querkante k von einer Ecke v oberhalb von u zu einer Ecke w unterhalb von u . Wäre dies eine Rückwärtskante, so würden w und v zur gleichen starken Zusammenhangskomponente gehören. Dies widerspricht der Wahl von u . Somit ist k eine Querkante, d.h. die Prozedur **sZhKprozedur** hat w schon verlassen. Da w noch auf dem Stapel liegt, gibt es eine Quer- oder Rückwärtskante von w oder einer Ecke oberhalb von w im Stapel zu einer Ecke w_1 unterhalb von w und oberhalb von e im Stapel. Da von dort die Ecke v erreichbar ist, kann dies wiederum keine Rückwärtskante sein. Somit ist auch die Tiefensuche für w_1 schon abgeschlossen, und die gleiche Argumentation läßt sich wiederholen. Da nur endlich viele Ecken oberhalb von e auf dem Stapel liegen, muß sich dabei irgendwann eine Rückwärtskante ergeben. Dies führt dann wieder zu einem Widerspruch. Somit ist $e = u$, und die Ecken, welche zu dem entsprechenden Blatt gehören, werden ausgegeben. Von nun an verhält sich die Prozedur so als wären diese Ecken nie vorhanden gewesen, d.h. die Behauptung folgt nun aus der Induktionsvoraussetzung. Damit ist der Beweis vollständig. ■

Wie kann die in der Prozedur angegebene Eigenschaft geprüft werden? Hierbei nützt man die durch die Tiefensuche erzeugte Numerierung der Ecken aus. Für jede Ecke wird die niedrigste Tiefensuchenummer unter den Ecken gespeichert, die man über eine Quer- oder Rückwärtskante von einem Nachfolger aus erreichen kann. Hierbei werden nur Ecken berücksichtigt, die noch im Stapel sind. Dieser Vorgang erfolgt während der Tiefensuche. Vor dem Ende der Tiefensuche für eine Ecke kann man mit diesem Wert entscheiden, ob die Wurzel einer starken Zusammenhangskomponente vorliegt. Dies ist genau dann der Fall, wenn dieser Wert nicht kleiner als die Tiefensuchenummer der Ecke ist.

Es werden zwei Felder verwaltet: **TSNummer** und **MinNummer**. Beim Besuch einer Ecke w wird zuerst **TSNummer[w]** belegt, und **MinNummer[w]** bekommt den gleichen Wert. Danach wird jeder Nachfolger v von w bearbeitet und anschließend der Wert von **MinNummer[w]** folgendermaßen geändert. Ist $w \rightarrow v$ eine Baumkante:

MinNummer[w] := min(MinNummer[w], MinNummer[v]);

und ist $w \rightarrow v$ eine Quer- oder Rückwärtskante und ist v noch im Stapel:

MinNummer[w] := min(MinNummer[w], TSNummer[v]);

Am Ende bilden jeweils die Ecken, deren Wert im Feld **MinNummer** gleich ist, eine starke Zusammenhangskomponente. Die Nummer ist dabei die Tiefensuchenummer der Wurzel der Zusammenhangskomponente. Abbildung 4.13 zeigt eine Realisierung dieser Anwendung der Tiefensuche. Die Korrektheit der Prozedur **starkeZhkomponente** ergibt sich aus dem obigen Lemma.

Abbildung 4.14 zeigt eine Anwendung des Algorithmus zur Bestimmung der starken Zusammenhangskomponenten auf den Graphen aus Abbildung 4.10. Teil (a) und (b) zeigen den Graphen und die Aufrufhierarchie der Prozedur **sZhKprozedur**. Teil (c) zeigt die Beladung des Feldes **MinNummer** am Ende des Algorithmus. Teil (d) zeigt die Veränderung

```

var MinNummer, TsNummer : array[1..max] of Integer;
    S : stapel of Integer;
    zähler : Integer;

procedure starkeZhKomponente(G : G-Graph);
var
    i : Integer;
begin
    Initialisiere TSNummer und zähler mit 0;
    for jede Ecke i do
        if TSNummer[i] = 0 then
            sZhKprozedur(i);
end

procedure sZhKprozedur(i : Integer);
var
    j : Integer;
begin
    zähler := zähler + 1;
    TSNummer[i] := MinNummer[i] := zähler;
    S.einfügen(i);
    for jeden Nachfolger j von i do begin
        if TSNummer[j] = 0 then begin
            sZhKprozedur(j);
            MinNummer[i] := min(MinNummer[i],MinNummer[j]);
        end
        else
            if S.enthalten(j)=true and
                TSNummer[i] > TSNummer[j] then
                    MinNummer[i] := min(MinNummer[i],TSNummer[j]);
    end;
    if TSNummer[i] = MinNummer[i] then
        while S.enthalten(i) = true do
            S.entfernen;
end

```

Abbildung 4.13: Prozeduren zur Bestimmung der starken Zusammenhangskomponenten

des Stapels gegen Ende des Aufrufs von (i) `sZhKprozedur(4)`, (ii) `sZhKprozedur(3)` und (iii) `sZhKprozedur(1)` und die ausgegebenen starken Zusammenhangskomponenten.

Unter Verwendung der Adjazenzliste ist der zeitliche Aufwand für die Bestimmung der starken Zusammenhangskomponenten $O(n + m)$, da die einzelnen Stapeloperationen in konstanter Zeit realisiert werden können. In der gleichen Zeit kann auch der Strukturgraph bestimmt werden.

4.6 Transitiver Abschluß und transitive Reduktion

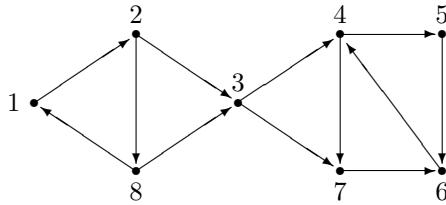
In vielen Anwendungen, in denen gerichtete Graphen verwendet werden, ist man vorrangig daran interessiert, ob es zwischen zwei gegebenen Ecken einen Weg gibt. Anfragen dieser Art können mittels aus dem Ausgangsgraphen abgeleiteter Graphen beantwortet werden: dem *transitiven Abschluß* und der *transitiven Reduktion*. Der transitive Abschluß eines gerichteten Graphen G ist ein gerichteter Graph mit gleicher Eckenmenge wie G , in dem es von der Ecke e eine Kante zur Ecke f gibt, falls es in G einen Weg von e nach f gibt, der aus mindestens einer Kante besteht. Liegt die Adjazenzmatrix des transitiven Abschlusses vor, so kann eine Abfrage auf die Existenz eines Weges in konstanter Zeit beantwortet werden. Dies erfordert natürlich einen Speicherplatz von $O(n^2)$. Die transitive Reduktion eines gerichteten Graphen G ist ein gerichteter Graph mit gleicher Eckenmenge wie G und einer minimalen Anzahl von Kanten des Graphen G , so daß beide Graphen den gleichen transitiven Abschluß haben. Die transitive Reduktion hat minimalen Speicheraufwand, beantwortet eine Abfrage aber auch nicht in konstanter Zeit. Man beachte, daß die transitive Reduktion nur für kreisfreie gerichtete Graphen eindeutig ist (vergleichen Sie dazu Übungsaufgabe 24 aus Kapitel 2). In vielen Fällen wird der Speicheraufwand für die transitive Reduktion wesentlich geringer sein als für den Ausgangsgraphen.

In diesem Abschnitt wird ein weiterer Algorithmus zur Bestimmung des transitiven Abschlusses eines gerichteten Graphen angegeben. Dieser Algorithmus hat in vielen Fällen eine geringere Laufzeit als die in den Abschnitten 2.6 und 4.3 angegebenen Algorithmen. Daneben wird auch ein Algorithmus zur Bestimmung der transitiven Reduktion eines kreisfreien gerichteten Graphen vorgestellt.

Zunächst wird gezeigt, daß man sich bei der Bestimmung des transitiven Abschlusses auf kreisfreie gerichtete Graphen beschränken kann. Der entscheidende Punkt hierbei ist, daß von jeder Ecke einer starken Zusammenhangskomponente die gleichen Ecken erreicht werden können. Somit genügt es, für je eine Ecke jeder starken Zusammenhangskomponente die Menge der erreichbaren Ecken zu bestimmen. Dazu wird zu einem gegebenen Graphen G der zugehörige Strukturgraph \hat{G} betrachtet. Wie im letzten Abschnitt gezeigt wurde, ist \hat{G} kreisfrei. Kennt man den transitiven Abschluß des Strukturgraphen \hat{G} , so kann daraus in linearer Zeit $O(n + m)$ der transitive Abschluß des Ursprungsgraphen bestimmt werden.

Es sei E die Eckenmenge von G , und für $e \in E$ sei E_e die starke Zusammenhangskom-

(a)



(b)

sZhKprozedur(1)
 sZhKprozedur(2)
 sZhKprozedur(3)
 sZhKprozedur(4)
 sZhKprozedur(5)
 sZhKprozedur(6)
 sZhKprozedur(7)
 sZhKprozedur(8)

(c)

Eckennummer	1	2	3	4	5	6	7	8
MinNummer	1	1	3	4	4	4	4	1

(d)

Stapel

starke Zusammenhangskomponente

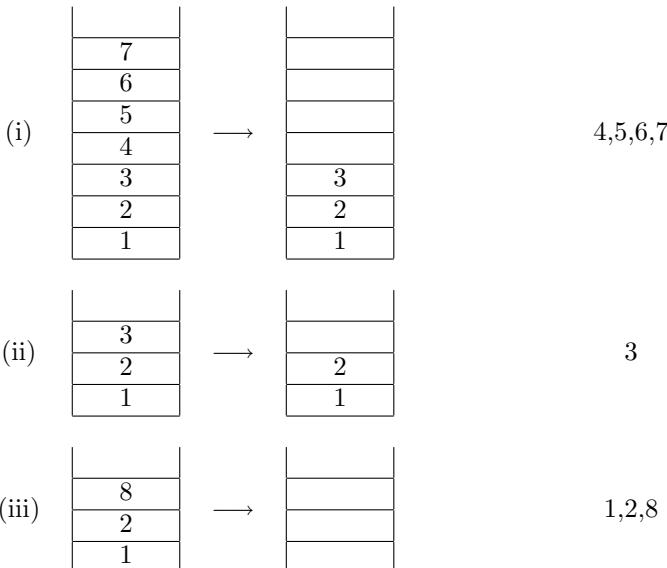


Abbildung 4.14: Bestimmung der starken Zusammenhangskomponenten

ponente von G , welche e enthält. Dann bildet die Menge

$$K = \left\{ (e, f) \mid e, f \in E, E_e = E_f \text{ und } |E_e| > 1 \text{ oder in } \hat{G} \text{ ist } E_f \text{ von } E_e \text{ erreichbar} \right\}$$

die Kantenmenge des transitiven Abschlusses von G . Man beachte, daß der Strukturgraph \hat{G} ebenfalls in linearer Zeit $O(n + m)$ bestimmt werden kann.

Im folgenden werden nun kreisfreie gerichtete Graphen betrachtet. In Übungsaufgabe 24 aus Kapitel 2 wurde bereits gezeigt, daß für kreisfreie Graphen G die transitive Reduktion eindeutig ist. Ferner wurde bewiesen, daß eine Kante (e, f) von G genau dann zur transitiven Reduktion gehört, wenn es keinen Weg von e nach f gibt, welcher aus mehr als einer Kante besteht. Im folgenden Lemma wird ein Zusammenhang zwischen transitiver Reduktion und transitivem Abschluß bewiesen. Dazu wird angenommen, daß eine topologische Sortierung des Graphen vorliegt.

Lemma. Es sei G ein kreisfreier gerichteter Graph mit Eckenmenge E . Die Nachbarn einer Ecke $e \in E$ seien in aufsteigender Reihenfolge einer topologischen Sortierung: f_1, \dots, f_s . Dann ist (e, f_i) genau dann eine Kante der transitiven Reduktion von G , wenn f_i von keiner der Ecken f_1, \dots, f_{i-1} aus erreichbar ist.

Beweis. a) Es sei (e, f_i) eine Kante der transitiven Reduktion von G . Angenommen, es gibt eine Ecke f_j mit $j < i$, von der aus f_i erreichbar ist. Dann gibt es einen Weg von e über f_j nach f_i . Dies widerspricht aber dem Resultat aus der oben angegebenen Übungsaufgabe.

b) Es sei f_i eine Ecke, von der aus keine der Ecken f_1, \dots, f_{i-1} erreichbar ist. Angenommen, (e, f_i) ist keine Kante der transitiven Reduktion von G . Dann gibt es einen Weg von e nach f_i , welcher aus mehr als einer Kante besteht. Es sei f der Nachfolger von e auf diesem Weg. Da die Nachfolger von e bezüglich einer topologischen Sortierung geordnet sind, ist $f \in \{f_1, \dots, f_{i-1}\}$. Dieser Widerspruch beweist das Lemma. ■

Aus dem letzten Lemma ergibt sich ein Algorithmus, welcher simultan den transitiven Abschluß und die transitive Reduktion eines kreisfreien gerichteten Graphen bestimmt, d.h. beide Probleme werden mit gleicher Laufzeit gelöst. Abbildung 4.15 zeigt eine entsprechende Prozedur `transAbschluss`. Hierbei wird vorausgesetzt, daß der Graph mit Hilfe seiner Adjazenzliste gegeben ist, wobei die Nachbarn jeder Ecke in aufsteigender Reihenfolge gemäß einer topologischen Sortierung geordnet sind. Der transitive Abschluß als auch die transitive Reduktion werden durch ihre Adjazenzmatrizen E_{Ab} bzw. E_{Red} dargestellt. Die Prozedur `transAbschluss` läßt sich auch so realisieren, daß die Darstellung mittels Adjazenzlisten erfolgt.

Die Korrektheit der Prozedur `transAbschluss` folgt direkt aus dem letzten Lemma und folgender Beobachtung: Zu jedem Zeitpunkt gilt für alle i : Ist ein Nachfolger j von i schon in E_{Ab} eingetragen, so sind alle von j aus erreichbaren Ecken ebenfalls schon in E_{Ab} eingetragen.

Die **if**-Anweisung wird m -mal durchgeführt, jedoch nur in m_{Red} Fällen müssen die entsprechenden Anweisungen durchgeführt werden. Hierbei bezeichnet m_{Red} die Anzahl

```

var EAb, ERed : array[1..max,1..max] of Integer;
procedure transAbschluss(G : K-G-Graph);
var
    i,j,k : Integer;
begin
    Initialisiere EAb und ERed mit 0;
    for i := n downto 1 do begin
        EAb[i,i] := 1;
        for jeden Nachbar j von i in aufsteigender Reihenfolge do
            if EAb[i,j] = 0 then
                for k := j to n do
                    if EAb[i,k] = 0 then
                        EAb[i,k] := EAb[j,k];
                    ERed[i,j] := 1;
            end
    end
end

```

Abbildung 4.15: Die Prozedur transAbschluss

der Kanten in der transitiven Reduktion. Der Gesamtaufwand aller **if**-Anweisungen zusammen ist somit $O(m + nm_{Red})$. Da auch die restlichen Anweisungen nicht mehr Zeit benötigen, ist dies auch schon der Gesamtaufwand von **transAbschluss**.

Es sei noch bemerkt, daß die Prozedur den sogenannten reflexiven transitiven Abschluß bestimmt, d.h. jede Ecke ist immer von sich aus erreichbar. Da der Ausgangsgraph kreisfrei ist, läßt sich dieser Nachteil beheben, indem in der Adjazenzmatrix E_{Ab} die Diagonaleinträge auf 0 gesetzt werden.

Aus den Ausführungen zu Beginn dieses Abschnittes folgt die Gültigkeit des folgenden Satzes.

Satz. Der transitive Abschluß eines gerichteten Graphen G kann mit dem Aufwand $O(m + nm_{Red})$ bestimmt werden. Hierbei bezeichnet m_{Red} die Anzahl der Kanten der transitiven Reduktion des Strukturgraphen von G.

4.7 Anwendung der Tiefensuche auf ungerichtete Graphen

Die Tiefensuche kann auch auf ungerichtete Graphen angewendet werden. Kanten, die zu unbesuchten Ecken führen, werden *Baumkanten* genannt. Baumkanten sind gerichtete Kanten und zeigen in die Richtung der neu besuchten Ecke. Tiefensuchebäume für ungerichtete Graphen sind also ebenfalls Wurzelbäume, wobei die Startecke die Wurzel ist. Bei der Tiefensuche wird immer eine vollständige Zusammenhangskomponente

durchlaufen. Somit ist die Anzahl der Tiefensuchebäume im Tiefensuchewald gleich der Anzahl der Zusammenhangskomponenten des Graphen. Daraus folgt auch, daß es in ungerichteten Graphen keine Querkanten geben kann. Da jede Kante in jeder Richtung durchlaufen wird, entfällt auch die Unterscheidung in Rückwärts- und Vorwärtskanten. Man unterteilt die Kanten eines ungerichteten Graphen in zwei Teilmengen. Eine Kante (e, f) heißt

Baumkante, falls der Aufruf `tiefensuche(e)` direkt `tiefensuche(f)` aufruft oder umgekehrt, und

Rückwärtskante, falls weder `tiefensuche(e)` direkt `tiefensuche(f)` aufruft noch umgekehrt, sondern einer der beiden Aufrufe indirekt durch den anderen erfolgt.

Es gilt folgendes Lemma:

Lemma. Ist (e, f) eine Rückwärtskante eines ungerichteten Graphen, so ist entweder e ein Vorgänger oder ein Nachfolger von f im Tiefensuchewald.

Beweis. Es wird nur der Fall betrachtet, bei dem die Tiefensuchenummer von e kleiner ist als die von f . Der andere Fall kann analog behandelt werden. Der Aufruf von `tiefensuche(e)` erfolgt also vor dem von `tiefensuche(f)`. Der Aufruf von `tiefensuche(e)` bewirkt, daß die Tiefensuche für alle Ecken, die von e aus erreichbar sind, durchgeführt wird. Da (e, f) eine Kante des Graphen ist, ist f von e aus erreichbar. Somit erfolgt der Aufruf von `tiefensuche(f)` direkt oder indirekt von `tiefensuche(e)`. Somit muß f ein Nachfolger von e im Tiefensuchebaum sein. ■

```

var ZKNummer : array[1..max] of Integer;
    zähler : Integer;
procedure zusammenhangskomponenten(G : Graph);
var
    i : Integer;
begin
    Initialisiere ZKNummer und zähler mit 0;
    for jede Ecke i do
        if ZKNummer[i] = 0 then begin
            zähler := zähler + 1;
            zusammenhang(i);
        end
    end
end

```

Abbildung 4.16: Die Prozedur zusammenhangskomponenten

Im folgenden wird die Tiefensuche so abgeändert, daß die Zusammenhangskomponenten eines ungerichteten Graphen bestimmt werden. Die Tiefensuche durchläuft immer alle

Ecken, welche von der Startecke erreichbar sind, d.h. eine komplette Zusammenhangskomponente. Der in Abbildung 4.16 dargestellte Algorithmus vergibt für jede Zusammenhangskomponente eine Nummer. Die Ecken werden mit dieser Nummer markiert. Am Ende bilden dann die Ecken mit gleicher Markierung eine vollständige Zusammenhangskomponente. Die Markierungen werden in dem Feld **ZKNummer** abgespeichert. Dieses Feld wird mit 0 initialisiert. Es wird auch dazu verwendet die besuchten Ecken zu verwalten; es ersetzt somit auch das Feld **TSNummer**.

Der Unterschied zur Tiefensuche ist der, daß die Variable **zähler** nicht mehr innerhalb der rekursiven Prozedur, sondern in der Prozedur **zusammenhangskomponenten** steht. Abbildung 4.17 zeigt die Prozedur **zusammenhang**.

```
procedure zusammenhang(i : Integer);
var
    j : Integer;
begin
    ZKNummer[i] := zähler;
    for jeden Nachbar j von i do
        if ZKNummer[j] = 0 then
            zusammenhang(j);
end
```

Abbildung 4.17: Die Prozedur **zusammenhang**

Die Komplexität der Prozedur **zusammenhangskomponenten** ist die gleiche wie die der Tiefensuche, bei Verwendung der Adjazenzliste $O(n + m)$ und bei Verwendung der Adjazenzmatrix $O(n^2)$. Abbildung 4.18 zeigt einen ungerichteten Graphen mit seiner Adjazenzliste. Ferner ist die Aufrufhierarchie der Prozedur **zusammenhang** und die Belegung des Feldes **ZKNummer** angegeben. Es gibt die beiden Zusammenhangskomponenten $\{1, 5, 6\}$ und $\{2, 3, 4\}$.

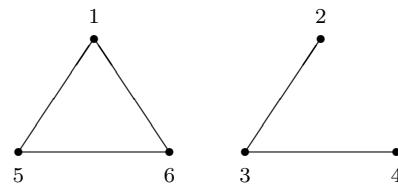
4.8 Anwendung der Tiefensuche in der Bildverarbeitung

Bildverarbeitung bzw. Bildverständen sind wichtige Teilgebiete der künstlichen Intelligenz. Um den Inhalt eines Bildes zu verstehen, müssen die einzelnen Bildpunkte zu Gebilden zusammengefaßt und als Objekte erkannt werden. Ausgangspunkt ist das digitalisierte Bild, welches aus Grauwerten besteht. Dieses wird in ein binäres, d.h. schwarz-weißes Bild überführt. Dazu wird ein Schwellwert ausgewählt und alle darunterliegenden Bildpunkte auf 0 und die übrigen auf 1 abgebildet. Die Bildpunkte mit Wert 1 werden nun als Objekte interpretiert, während solche mit Wert 0 als Hintergrund angesehen werden.

Um Objekte zu erkennen, wird der Begriff des *8-Zusammenhangs* von Bildpunkten definiert. Dazu wird ein *Nachbarschaftsgraph* aus dem binären Bild konstruiert: Die Bildpunkte mit Wert 1 bilden die Ecken, und zwischen zwei Ecken gibt es eine Kante, falls

Adjazenzliste

1	→	5,6
2	→	3
3	→	2,4
4	→	3
5	→	1,6
6	→	1,5



```

zähler = 1
zusammenhang(1)
zusammenhang(5)
zusammenhang(6)

zähler = 2
zusammenhang(2)
zusammenhang(3)
zusammenhang(4)

```

Eckennummer	1	2	3	4	5	6
ZKNummer	1	2	2	2	1	1

Abbildung 4.18: Bestimmung der Zusammenhangskomponenten

die entsprechenden Bildpunkte eine gemeinsame Kante oder Ecke haben. Die Zusammenhangskomponenten des Nachbarschaftsgraphen bilden die 8-zusammenhängenden Komponenten. Abbildung 4.19 zeigt ein binäres Bild zusammen mit dem entsprechenden Nachbarschaftsgraphen. Die 8-zusammenhängenden Komponenten werden in einer der ersten Phasen des Bildverständens bestimmt und bilden dann die Grundlage weiterer Untersuchungen.

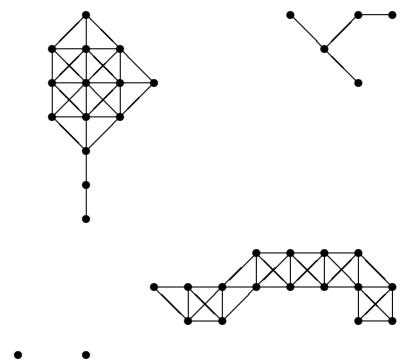
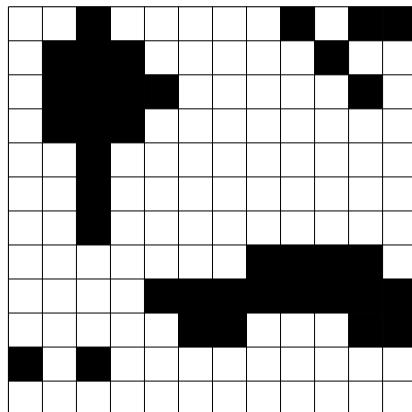


Abbildung 4.19: Ein binäres Bild und der zugehörige Nachbarschaftsgraph

Der im letzten Abschnitt diskutierte Algorithmus zur Bestimmung der Zusammenhangskomponenten kann natürlich für die Ermittlung der 8-zusammenhängenden Komponenten verwendet werden. Dazu ist es aber notwendig, den Nachbarschaftsgraphen aufzu-

bauen. Die Tiefensuche lässt sich aber auch direkt auf die Matrix B des binären Bildes anwenden. Dadurch erspart man sich den Aufbau des Graphen.

Abbildung 4.20 zeigt eine auf die spezielle Situation abgestimmte Version der Tiefensuche. Hierbei ist B eine $n \times n$ Matrix und $B[i,j] = \text{true}$, falls der entsprechende Bildpunkt den Wert 1 hat. Ein Eintrag (i,j) hat dabei maximal acht Nachbarn. Die Anzahl reduziert sich auf drei bzw. fünf, falls der Bildpunkt in der Ecke bzw. am Rande liegt. In der Matrix $Zk\text{Nummer}$ haben zwei Bildpunkte genau dann den gleichen Eintrag, wenn sie zur gleichen 8-Zusammenhangskomponente gehören. Ist der Eintrag gleich 0, so hatte der Bildpunkt den Wert 0 und gehört somit zum Hintergrund.

```

var ZkNummer : array[1..n,1..n] of Integer;
      zähler : Integer;

procedure 8-zuhaKomponenten(B : array[1..n,1..n] of Boolean);
var
      i, j : Integer;
begin
    Initialisiere ZkNummer und zähler mit 0;
    for i := 1 to n do
        for j := 1 to n do
            if (B[i,j] = true and
                ZkNummer[i,j] = 0) then begin
                ZkNummer[i,j] := zähler;
                zähler := zähler + 1;
                8-zusammenhang(i,j);
            end;
    end

procedure 8-zusammenhang(i, j : Integer);
var
      k, l : Integer;
begin
    ZkNummer[i,j] := zähler;
    for jeden Nachbar (k,l) von (i,j) do
        if B[k,l] = true and ZkNummer[k,l] = 0 then
            8-zusammenhang(k,l);
    end

```

Abbildung 4.20: Die Bestimmung der 8-zusammenhängenden Komponenten

4.9 Blöcke eines ungerichteten Graphen

Eine Ecke eines ungerichteten Graphen heißt *trennende Ecke*, wenn deren Wegfall die Anzahl der Zusammenhangskomponenten erhöht. In Abbildung 4.21 sind x und y die einzigen trennenden Ecken des Graphen G . Ein Graph ohne trennende Ecken heißt *zweifach zusammenhängend*. Diese Graphen kann man leicht charakterisieren.

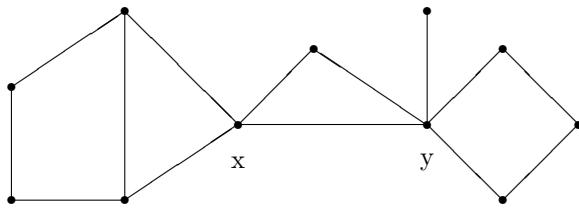


Abbildung 4.21: Ein Graph G mit seinen trennenden Ecken

Lemma. Ein zusammenhängender ungerichteter Graph mit mindestens drei Ecken ist genau dann zweifach zusammenhängend, wenn je zwei Ecken gemeinsam auf einem einfachen geschlossenen Weg liegen.

Beweis. Es sei zunächst G ein Graph mit der angegebenen Eigenschaft. Angenommen, G ist nicht zweifach zusammenhängend. Dann besitzt G eine trennende Ecke e . Somit gibt es Ecken x und y , die in G durch einen Weg verbunden sind und nach dem Entfernen von e in verschiedenen Zusammenhangskomponenten liegen. Somit können x und y nicht auf einem einfachen geschlossenen Weg liegen. Dieser Widerspruch zeigt, daß G zweifach zusammenhängend ist.

Es seien x und y zwei beliebige Ecken eines zweifach zusammenhängenden Graphen G . Es muß nun gezeigt werden, daß x und y auf einem gemeinsamen einfachen geschlossenen Weg liegen. Dies geschieht durch vollständige Induktion nach dem Abstand $d(x, y)$. Hierbei ist $d(x, y)$ die minimale Anzahl von Kanten von einem Weg von x nach y . Ist $d(x, y) = 1$, so existiert eine Kante zwischen x und y . Da x keine trennende Ecke ist, gibt es in G einen Weg von x nach y , der nicht diese Kante verwendet. Somit liegen x und y in diesem Fall auf einem einfachen geschlossenen Weg. Sei nun $d(x, y) > 1$.

Da G zusammenhängend ist, gibt es einen einfachen Weg W von x nach y . Sei z der Vorgänger von y auf diesem Weg. Da $d(x, z) < d(x, y)$ ist, liegen x und z nach Induktionsannahme auf einem einfachen geschlossenen Weg W_1 . Abbildung 4.22 zeigt diese Situation. Da G zweifach zusammenhängend ist, gibt es einen einfachen Weg W_2 von x nach y , der z nicht enthält. Unter den Ecken, die W_1 und W_2 gemeinsam haben, sei l die Ecke auf W_2 , die am nächsten zu y liegt. Eine solche Ecke muß es geben, da z.B. x auf W_1 und W_2 liegt. Nun kann man leicht einen einfachen geschlossenen Weg angeben, auf dem x und y liegen: Von x startend folge man W_1 bis zu l auf dem Teil, der nicht über z führt, von l auf W_2 zu y , von y zu z , und dann auf dem Teil von W_1 , der nicht über l führt, zurück nach x . In Abbildung 4.22 ist dieser einfache geschlossene Weg fett gezeichnet. ■

Die Kantenmenge K eines ungerichteten Graphen kann in disjunkte Teilmengen K_1, \dots, K_s zerlegt werden, so daß die von den K_i gebildeten Untergraphen zweifach zusammenhängend

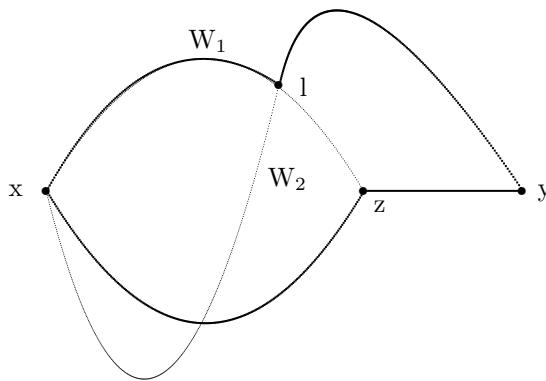


Abbildung 4.22: Ein einfacher geschlossener Weg in einem zweifach zusammenhängenden Graphen

sind. Dazu bilde man folgende symmetrische Relation über der Menge K der Kanten: $k_1, k_2 \in K$ sind äquivalent, falls es einen einfachen geschlossenen Weg gibt, der sowohl k_1 als auch k_2 enthält.

Im folgenden wird zunächst gezeigt, daß diese Relation transitiv ist. Es seien $k_1 = (u_1, v_1)$, $k_2 = (u_2, v_2)$ und $k_3 = (u_3, v_3)$ Kanten und W_1 bzw. W_2 einfache geschlossene Wege, welche die Kanten k_1, k_2 bzw. k_2, k_3 enthalten. Die Bezeichnung der Ecken sei so gewählt, daß u_1, u_2, v_2, v_1 in dieser Reihenfolge auf W_1 erscheinen. Es sei e die erste Ecke auf dem Teilweg W_1 von u_1 nach u_2 , welche auch auf W_2 liegt. Da u_2 auf W_2 liegt, muß es eine solche Ecke geben. Ferner sei f die erste Ecke auf dem Teilweg von W_1 von v_1 nach v_2 , welche auch auf W_2 liegt. Da v_2 auf W_2 liegt, muß es auch eine solche Ecke geben. Da W_1 ein einfacher Weg ist, gilt $e \neq f$. Es sei W der Teilweg von W_1 von e nach f , welcher die Kante k_1 enthält, und W' sei der Teilweg von W_2 von e nach f , welcher die Kante k_3 enthält. Diese beiden Wege bilden zusammen einen einfachen geschlossenen Weg, der die Kanten k_1, k_3 enthält. Somit hat man gezeigt, daß die angegebene Relation transitiv ist.

Die von den Äquivalenzklassen dieser Relation gebildeten Untergraphen sind entweder zweifach zusammenhängend oder bestehen aus genau einer Kante. Dies folgt aus dem obigen Lemma. Die so gebildeten Untergraphen nennt man die *Blöcke* des Graphen. Abbildung 4.23 zeigt die Blöcke des Graphen aus Abbildung 4.21.

Zwei Blöcke haben maximal eine gemeinsame Ecke; dies ist dann eine trennende Ecke. Jede trennende Ecke gehört zu mindestens zwei verschiedenen Blöcken (in Abbildung 4.23 gehört y zu den Blöcken B_2, B_3 und B_4). Eine Ecke, die weder isoliert, noch eine trennende Ecke ist, gehört genau zu einem Block. Jede Kante gehört zu genau einem Block.

Für jeden Graphen G kann man den sogenannten *Blockgraph* G_B bilden. Die Ecken von

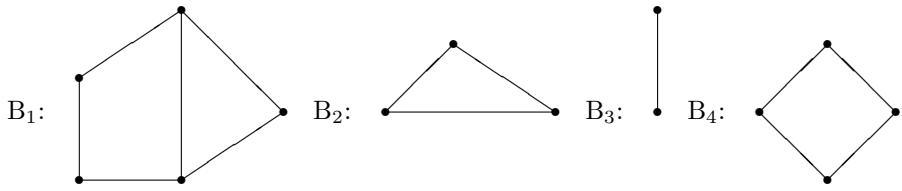


Abbildung 4.23: Die Blöcke des Graphen aus Abbildung 4.21

G_B sind die Blöcke und die trennenden Ecken von G . Ein Block B und eine trennende Ecke e sind genau dann durch eine Kante verbunden, falls e in B liegt. Abbildung 4.24 zeigt den Blockgraph G_B für den Graphen aus Abbildung 4.21.

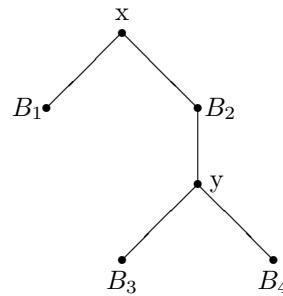


Abbildung 4.24: Der Blockgraph G_B zu dem Graphen G aus Abbildung 4.21

Ist G ein zweifach zusammenhängender Graph, so besteht G_B genau aus einer Ecke. Wenn G zusammenhängend ist, so ist G_B ein Baum. Dies folgt direkt aus dem letzten Lemma.

Das Auffinden der Blöcke eines ungerichteten Graphen kann mit Hilfe der Tiefensuche erfolgen. Das Verfahren ist ähnlich dem der Bestimmung der starken Zusammenhangskomponente eines gerichteten Graphen. Das folgende Lemma ergibt sich sofort aus den Eigenschaften des Tiefensuchebaumes für ungerichtete Graphen.

Lemma. Es sei B der Tiefensuchebaum eines ungerichteten zusammenhängenden Graphen. Genau dann ist die Wurzel e von B eine trennende Ecke, falls e mehrere Nachfolger in B hat. Eine andere Ecke e ist genau dann eine trennende Ecke, falls die folgenden beiden Bedingungen erfüllt sind:

1. e hat in B einen Nachfolger;

2. weder e noch ein Nachfolger von e in B ist mittels einer Rückwärtskante mit einem Vorgänger von e in B verbunden.

Um die trennenden Ecken während der Tiefensuche zu erkennen, werden wieder zwei Felder verwaltet: **TSNummer** und **MinNummer**. Das erste Feld hat die gleiche Bedeutung wie bei der Tiefensuche, und das zweite Feld enthält für jede Ecke w den Wert

$$\min\{\text{TSNummer}[v] \mid v \text{ ist Nachfolger von } w \text{ im Tiefensuchewald und } (v, w) \text{ ist eine Rückwärtskante}\}$$

Somit ist **MinNummer**[w] die Tiefensuchenummer des ersten Nachfolgers v von w im Tiefensuchewald T , der durch einen Weg in T , gefolgt von einer einzigen Rückwärtskante (v, w) erreicht werden kann.

Beim Besuch einer Ecke w wird zuerst **TSNummer**[w] belegt, und **MinNummer**[w] bekommt den gleichen Wert. Danach wird jeder Nachbar v von w bearbeitet und anschließend der Wert von **MinNummer**[w] folgendermaßen geändert. Wurde v bisher noch nicht besucht:

$$\text{MinNummer}[w] := \min(\text{MinNummer}[w], \text{MinNummer}[v]);$$

Wurde v schon besucht und ist v nicht der Vorgänger von w im Tiefensuchebaum:

$$\text{MinNummer}[w] := \min(\text{MinNummer}[w], \text{TSNummer}[v]);$$

Im letzten Fall ist (w, v) eine Rückwärtskante. Falls eine Ecke w eine trennende Ecke ist, so hat w einen Nachbarn v , so daß **MinNummer**[v] \geq **TSNummer**[w], oder w ist die Startecke der Tiefensuche. Somit lassen sich die trennenden Ecken während der Tiefensuche erkennen. Die notwendigen Prozeduren sind in Abbildung 4.25 dargestellt.

Die Prozedur **blöcke** verwendet einen Stapel, auf dem die Kanten abgelegt werden. Sobald ein Block gefunden wurde, werden die entsprechenden Kanten wieder entfernt. Für Nachbarn j der Startecke 1 gilt:

$$\text{MinNummer}[j] \geq \text{TSNummer}[1] = 1$$

Führt eine Baumkante von 1 nach j , so wird dadurch nach dem Aufruf von **block-proz(j)** der Block, welcher die Kante $(1, j)$ enthält, ausgegeben. Somit wird der Fall, daß die Startecke eine trennende Ecke ist, korrekt behandelt. Gibt es eine Baumkante von i nach j und ist **MinNummer**[j] \geq **TSNummer**[i], so muß noch gezeigt werden, daß die Kanten oberhalb von (i, j) einschließlich (i, j) in S einen Block bilden. Der Beweis erfolgt durch vollständige Induktion nach der Anzahl b der Blöcke. Ist $b = 1$, so ist der Graph zweifach zusammenhängend. Dann hat die Ecke 1 nur einen Nachfolger j im Tiefensuchebaum, und die Korrektheit des Algorithmus folgt sofort. Es sei nun $b > 1$ und j die erste Ecke, für die **MinNummer**[j] \geq **TSNummer**[i] gilt. Bis zu diesem Zeitpunkt wurden noch keine Kanten von S entfernt, und alle Kanten oberhalb von (i, j) sind inzident zu einem Nachfolger von j im Tiefensuchebaum. Da j eine trennende Ecke ist,

```

var TSNr, MinNr, Vorgänger : array[1..max] of Integer;
    zähler : Integer;
    S : stapel of Kanten;
procedure blöcke(G : Graph);
var
    i : Integer;
begin
    Initialisiere TSNr und zähler mit 0;
    for jede Ecke i do
        if TSNr[i] = 0 then
            blockproz(i);
end

procedure blockproz(i : Integer);
var
    j : Integer;
begin
    zähler := zähler + 1;
    TSNr[i] := MinNr[i] := zähler;
    for jeden Nachbar j von i do begin
        Falls (i,j) noch nicht in S war, S.einfügen((i,j));
        if TSNr[j] = 0 then begin
            Vorgänger[j] := i;
            blockproz(j);
            if MinNr[j] >= TSNr[i] then
                while S.enthalten((i,j)) = true do
                    S.entfernen;
            MinNr[i] := min(MinNr[i],MinNr[j]);
        end
        else
            if j ≠ Vorgänger[i] then
                MinNr[i] := min(MinNr[i],TSNr[j]);
    end
end

```

Abbildung 4.25: Prozeduren zur Bestimmung der Blöcke

bilden die Kanten oberhalb von (i, j) auf S genau den Block, welcher die Kante (i, j) enthält. Es sei G' der Graph, der entsteht, wenn man aus G genau diese Kanten und die entsprechenden Ecken (außer der Ecke j) entfernt. G' besteht aus $b - 1$ Blöcken. Nach Induktionsannahme arbeitet der Algorithmus für G' korrekt. Somit arbeitet er aber auch für G korrekt.

Abbildung 4.26 zeigt eine Anwendung des Algorithmus zur Bestimmung der Blöcke eines ungerichteten Graphen. Teil (a) zeigt den Graphen und die Aufrufhierarchie der Prozedur `blockproz`. Teil (b) zeigt die Belegung der Felder `TSNummer` und `MinNummer` am Ende des Algorithmus. Teil (c) zeigt die Veränderung des Stapels nach dem Aufruf von (i) `blockproz(3)`, (ii) `blockproz(5)` und (iii) `blockproz(2)` und die ausgegebenen Blöcke.

Der zeitliche Aufwand zur Bestimmung der Blöcke ist $O(n + m)$. Gegenüber der Tiefensuche sind zusätzlich nur die Stapeloperationen zu berücksichtigen. Das einzige Problem besteht darin, zu vermeiden, daß eine Kante zweimal auf den Stapel kommt. Dies läßt sich aber leicht feststellen. Eine Kante (i, j) ist schon auf dem Stapel gewesen, falls (j, i) eine Baumkante im Tiefensuchebaum ist, oder falls $\text{TSNummer}[j] > \text{TSNummer}[i]$ und (i, j) keine Baumkante ist. Baumkanten können dabei mittels des Feldes `Vorgänger` ermittelt werden. Somit ist der Gesamtaufwand für die Stapeloperationen $O(m)$.

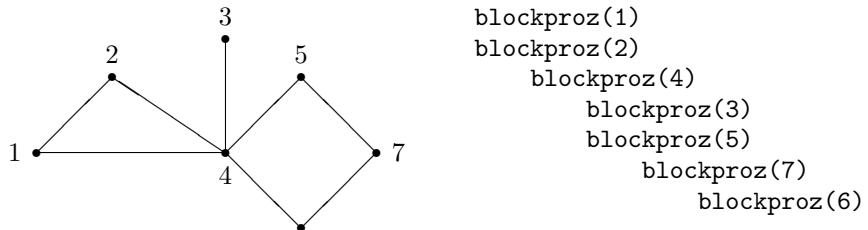
4.10 Breitensuche

Eine Alternative zur Tiefensuche bildet die *Breitensuche* (*breadth-first-search*), welche ebenfalls auf dem im Abschnitt 4.1 diskutierten Markierungsalgorithmus beruht. Im Unterschied zur Tiefensuche wird bei der Breitensuche die Suche so breit wie möglich angelegt; für jede besuchte Ecke werden zunächst alle Nachbarn besucht. Dazu wählt man im zweiten Schritt des Markierungsalgorithmus jeweils eine Kante, deren Anfangsecke die am längsten markierte Ecke ist. Bei der Breitensuche werden die Ecken beim Besuch numeriert (*Breitensuchenummern*). Im Gegensatz zur Tiefensuche bekommt nicht jede Ecke eine andere Nummer. Ecken mit gleicher Breitensuchenummer bilden ein *Niveau*. Die Startecke e bekommt die Nummer $\text{Niv}(e) = 0$ und wird als besucht markiert; sie bildet das Niveau 0. Die Ecken des Niveaus $i + 1$ sind die noch unbesuchten Nachbarn der Ecken aus Niveau i . Die Breitensuche endet damit, daß alle von der Startecke aus erreichbaren Ecken markiert sind. Die Kanten, die zu unbesuchten Ecken führen, nennt man Baumkanten. Die Baumkanten bilden den *Breitensuchebaum*.

Die Breitensuche kann sowohl auf gerichtete als auch auf ungerichtete Graphen angewendet werden. Abbildung 4.27 zeigt einen gerichteten Graphen mit seinem Breitensuchebaum für die Startecke 1. Man vergleiche dazu den Tiefensuchebaum dieses Graphen, der in Abbildung 4.1 dargestellt ist. Es gibt drei Niveaus: Ecke 1 auf Niveau 0, die Ecken 2, 5 und 7 auf Niveau 1 und die Ecken 4, 6, 8 auf Niveau 2. Die Ecke 3 ist nicht von der Startecke aus erreichbar.

Abbildung 4.28 zeigt eine Realisierung der Breitensuche. Dabei werden alle von der Startecke aus erreichbaren Ecken besucht und in Niveaus aufgeteilt. Sind nicht alle

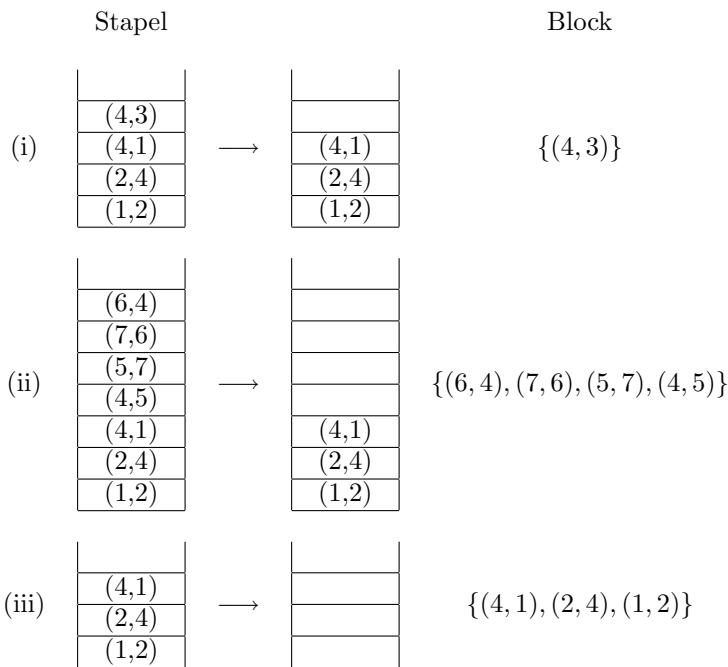
(a)



(b)

Eckennummer	1	2	3	4	5	6	7
TSNummer	1	2	4	3	5	7	6
MinNummer	1	1	4	1	3	3	3

(c)

*Abbildung 4.26: Bestimmung der Blöcke*

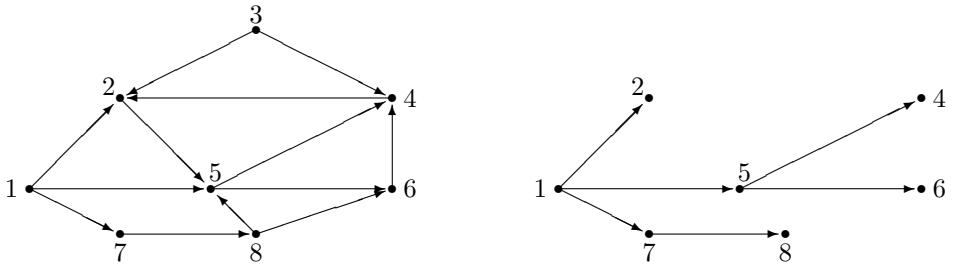


Abbildung 4.27: Ein gerichteter Graph mit seinem Breitensuchebaum

Ecken des Graphen von der Startecke aus erreichbar, so sind mehrere Breitensuchen notwendig, um alle Ecken zu besuchen. In dem Feld `niveau` wird für jede Ecke die Niveaunummer abgespeichert. In dem Feld `niveauListe` werden die Ecken in der Reihenfolge, in der sie besucht werden, abgelegt. In der ersten Komponente steht die Startecke. Die Indizes, bei denen ein neues Niveau beginnt, stehen in dem Feld `niveauBeginn`; d.h. die Ecken auf dem Niveau s findet man in den Komponenten

```
niveauListe[niveauBeginn[s]], ..., niveauListe[niveauBeginn[s+1]-1].
```

Die Variablen `start` und `ende` kennzeichnen immer den Bereich des letzten Niveaus, und die Variable `nächstes` gibt den Index des nächsten freien Platzes in `niveauListe` an. Die `repeat`-Schleife wird für jedes Niveau einmal durchgeführt. Jede Kante wird bei gerichteten Graphen einmal und bei ungerichteten Graphen zweimal durchlaufen. Daraus folgt, daß die Breitensuche die gleiche Komplexität wie die Tiefensuche hat. Bei der Verwendung der Adjazenzliste zur Darstellung des Graphen ergibt sich die Komplexität $O(n + m)$, wobei n die Anzahl der Ecken und m die Anzahl der Kanten ist.

Die Eigenschaften der Breitensuche sind im folgenden Lemma zusammengefaßt.

Lemma. Es sei G ein gerichteter oder ein ungerichteter Graph.

- Für jede Ecke gibt $Niv(e)$ die Anzahl der Kanten des kürzesten Weges von der Startecke zur Ecke e an.
- Ist G ein zusammenhängender ungerichteter Graph, so ist der Breitensuchebaum ein aufspannender Baum.
- Ist (e, f) eine Baumkante, so gilt $|Niv(e) - Niv(f)| = 1$. Ist der Graph gerichtet, so ist $Niv(f) = Niv(e) + 1$.
- Ist der Graph ungerichtet, so gilt $|Niv(e) - Niv(f)| \leq 1$ für jede Kante (e, f) . Ist der Graph gerichtet, so ist $Niv(f) \leq Niv(e) + 1$.

Beweis. a) Der Beweis erfolgt durch vollständige Induktion nach dem Niveau i . Die Ecken mit $i = 1$ sind direkt mit der Startecke verbunden. Sei nun e eine Ecke mit

```

var niveau, niveauListe : array[1..max] of Integer;
    niveauBeginn : array[0..max] of Integer;
procedure breitensuche(G : Graph; startecke : Integer);
var
    start, ende, nächstes, i, j : Integer;
    niveauNummer, niveauAnzahl : Integer;
begin
    Initialisiere niveau mit -1 und niveauNummer mit 0;
    niveau[startecke] := niveauNummer;
    niveauBeginn[0] := 1;
    niveauListe[1] := startecke;
    start := ende := 1;
    nächstes := 2;
repeat
    niveauNummer := niveauNummer + 1;
    niveauBeginn[niveauNummer] := nächstes;
    niveauAnzahl := 0;
    for i := start to Ende do
        for jeden Nachbar j von niveauListe[i] do
            if niveau[j] = -1 then begin
                niveau[j] := niveauNummer;
                niveauListe[nächstes] := j;
                nächstes := nächstes + 1;
                niveauAnzahl := niveauAnzahl + 1;
            end;
            start := Ende + 1;
            Ende := Ende + niveauAnzahl;
        until niveauAnzahl = 0;
    end;

```

Abbildung 4.28: Die Prozedur breitensuche

$Niv(e) = i+1$. Sei f eine Ecke aus Niveau i , welche zu e benachbart ist. Nach Induktion gibt es einen Weg, bestehend aus i Kanten von der Startecke zur Ecke f . Somit gibt es auch einen Weg von der Startecke zur Ecke e , der aus $i+1$ Kanten besteht. Gäbe es nun umgekehrt einen Weg von der Startecke zur Ecke e mit weniger als $i+1$ Kanten, so würde daraus folgen, daß e in einem Niveau j mit $j < i+1$ wäre. Dieser Widerspruch zeigt die Behauptung.

- b) Da Baumkanten immer zu unbesuchten Kanten führen, kann kein geschlossener Weg auftreten. Da G zusammenhängend ist, wird auch jede Ecke erreicht.
- c) Eine Baumkante verbindet immer Ecken aus zwei aufeinanderfolgenden Niveaus. Bei gerichteten Graphen führen Baumkanten immer zu Ecken im nächst höheren Niveau.
- d) Für Baumkanten folgt die Aussage aus c). Es sei zunächst G ein ungerichteter

Graph. Sei nun (e, f) keine Baumkante. Dann ist f entweder im gleichen Niveau wie e , ein Niveau höher oder ein Niveau tiefer. Auf jeden Fall gilt $|Niv(e) - Niv(f)| \leq 1$. Sei nun G ein gerichteter Graph und (e, f) keine Baumkante. Dann wurde f vorher schon besucht, d.h. das Niveau von f ist maximal $Niv(e) + 1$. ■

Die vorgestellte Realisierung der Breitensuche läßt die Gemeinsamkeit mit der Tiefensuche nicht gut erkennen. Eine Realisierung mittels einer Warteschlange offenbart aber die Ähnlichkeit dieser Verfahren. Ersetzt man in der nichtrekursiven Realisierung der Tiefensuche den Stapel durch eine Warteschlange, so bewirkt dies, daß die Reihenfolge der besuchten Ecken der der Breitensuche entspricht. In die Warteschlange kommen die Ecken, deren Nachbarn noch besucht werden müssen. Die Breitensuche entfernt nun immer eine Ecke vom Anfang der Warteschlange und fügt die noch nicht besuchten Nachbarn dieser Ecke am Ende in die Warteschlange ein. Die Breitensuche ist beendet, wenn die Warteschlange leer ist. Abbildung 4.29 zeigt die Prozedur **breitensuche**.

```

var niveau : array[1..max] of Integer;
procedure breitensuche(G : Graph; startecke : Integer);
var
    i, j : Integer;
    W : warteschlange of Integer;
begin
    Initialisiere niveau mit -1;
    niveau[startecke] := 0;
    W.einfügen(startecke);
    while W ≠ ∅ do begin
        i := W.entfernen();
        for jeden Nachbar j von i do
            if niveau[j] = -1 then begin
                niveau[j] := niveau[i] + 1;
                W.einfügen(j);
            end
    end
end

```

Abbildung 4.29: Die Prozedur **breitensuche**

Mit Hilfe der Breitensuche kann man leicht feststellen, ob ein ungerichteter Graph bipartit ist. Wendet man die Breitensuche auf einen bipartiten Graphen mit der Eckenmenge $E_1 \cup E_2$ an, so liegen die Niveaus abwechselnd in E_1 und E_2 . Liegt die Startecke in E_1 , so gilt $Niv(e) \equiv 0(2)$ für jede Ecke e aus E_1 und $Niv(e) \equiv 1(2)$ für jede Ecke e aus E_2 . Daraus folgt, daß für jede Kante (e, f) eines bipartiten Graphen $Niv(e) + Niv(f) \equiv 1(2)$ ist. Die Umkehrung dieser Aussage gilt ebenfalls.

Lemma. Es sei G ein ungerichteter Graph auf den die Breitensuche angewendet wird. Genau dann ist G bipartit, wenn für jede Kante (e, f) von G gilt:

$$Niv(e) + Niv(f) \equiv 1(2).$$

Eine weitere Anwendungen der Breitensuche findet man in Kapitel 6.

4.11 Beschränkte Tiefensuche

Viele Techniken der künstlichen Intelligenz stützen sich auf Suchverfahren auf Graphen. Das besondere dabei ist, daß die Graphen sehr groß sind und daß sie meistens nur implizit vorliegen. Dies sei am Beispiel des 8-Puzzles erläutert: Ein quadratisches Brett ist in neun Quadrate aufgeteilt. Eines dieser Quadrate ist leer, auf den anderen befinden sich acht Plättchen mit den Nummern von 1 bis 8. Ziel ist es, durch Verschieben der Plättchen die Zielstellung zu erreichen. Abbildung 4.30 zeigt eine Start- und eine Zielstellung für dieses Puzzle.

7	3	1
2	8	
4	6	5

Startstellung:

1	2	3
4	5	6
7	8	

Zielstellung:

Abbildung 4.30: Start- und Zielstellung eines 8-Puzzles

Das Problem läßt sich leicht durch einen gerichteten Graphen darstellen: Jedem Brettzustand wird eine Ecke zugeordnet, und zwei Ecken sind durch eine Kante verbunden, wenn die zugehörigen Zustände durch eine Verschiebung des Plättchens ineinander überführt werden können. Gesucht ist ein Weg von der Start- zur Zielecke. Man sieht sofort, daß der Graph sehr groß ist. Es gibt 9! Ecken. Der Eckengrad jeder Ecke ist gleich 2, 3 oder 4, je nach der Position des freien Quadrats. Das Erzeugen des Graphen ist sehr zeit- und speicherintensiv.

Die zu untersuchenden Graphen liegen meistens in impliziter Form vor, d.h. es gibt eine Funktion, welche zu einer gegebenen Ecke die Nachfolger erzeugt. Für das oben angegebene Beispiel des 8-Puzzles läßt sich eine solche Funktion leicht angeben. Im Prinzip können sowohl Tiefen- als auch Breitensuche zur Lösung dieses Problems angewendet werden. Die Breitensuche liefert sogar den Weg mit den wenigsten Kanten.

Die Größe des Graphen erschwert allerdings die Verwendung der in diesem Kapitel dargestellten Realisierungen dieser Suchverfahren. Die Verwaltung aller Ecken in einem Feld erfordert sehr viel Speicherplatz. Die nächst größere Variante dieses Problems, das 15-Puzzle, liegt jenseits aller möglichen Hauptspeichergrößen. Aus diesem Grund muß auf eine Verwaltung der schon besuchten Ecken verzichtet werden.

Die Tiefensuche läßt sich so erweitern, daß diese Information nicht benötigt wird. Der Preis dafür ist, daß Ecken eventuell mehrmals besucht werden. Damit die Suche sich trotzdem nicht in geschlossenen Wegen *verirrt*, muß zumindest der Weg von der Startecke zur aktuellen Ecke verwaltet werden. Bevor eine Ecke besucht wird, wird überprüft, ob sie nicht schon auf dem aktuellen Weg vorkommt. Auf diese Weise erreicht

man, daß die Suche den Graphen vollständig durchsucht. Dabei wird jeder Weg von der Startecke zu einer beliebigen Ecke durchlaufen, d.h. eine Ecke kann mehrmals besucht werden.

Die in Abbildung 4.4 dargestellte Version der Tiefensuche eignet sich für diese Vorgehensweise, denn zu jedem Zeitpunkt befinden sich die Ecken des Weges von der Startecke zur aktuellen Ecke im Stapel. Allerdings befinden sich dort noch mehr Ecken, welche eventuell nie gebraucht werden. Der Grund hierfür ist, daß immer alle Nachfolger der aktuellen Ecke auf dem Stapel abgelegt werden. Eine alternative Vorgehensweise besteht darin, immer nur einen Nachfolger zu erzeugen und diesen auf dem Stapel abzulegen. Statt die Ecke bei der nächsten Gelegenheit zu entfernen, wird zunächst ein weiterer Nachfolger erzeugt und auf dem Stapel abgelegt. Erst wenn alle Nachfolger erzeugt wurden, wird die Ecke wieder aus dem Stapel entfernt. Der Vorteil ist eine weitere Platzersparnis, und die Ecken im Stapel beschreiben genau den Weg von der Startecke zur Zielecke. Der maximale Speicheraufwand ist in diesem Fall proportional zu der maximalen Länge eines Weges.

```

function b-tiefensuche(G : G-Graph; start, T : Integer) : Boolean;
var
    q : Boolean;
    i, j : Integer;
    S : stapel of Integer;
begin
    S.einfügen(start);
    q := ziel(start);
    while S ≠ ∅ and q = false do begin
        i := S.kopf;
        if es gibt noch einen Nachfolger j von i then begin
            q := ziel(j);
            if S.enthalten(j) = false and S.tiefe < T or q = true then
                S.einfügen(j);
        end
        else
            S.entfernen;
    end;
    if q = true then
        S.ausgabe;
    b-tiefensuche := q;
end

```

Abbildung 4.31: Eine Realisierung der beschränkten Tiefensuche

Auf diese Weise hat man zwar das Speicherplatzproblem beseitigt, aber das Verhalten der Tiefensuche ist im Vergleich zu der Breitensuche immer noch nicht zufriedenstellend. Es kann passieren, daß die Tiefensuche sehr tief in den Graphen vordringt ohne die Lösung zu finden, obwohl es Lösungen gibt welche nur aus wenigen Kanten bestehen. Noch gravierender ist das Problem bei unendlichen Graphen. Dort kann es geschehen,

daß die Suche keine Lösung findet, da sie in der *falschen Richtung* sucht. Die Breitensuche findet immer den Weg mit den wenigsten Kanten zuerst. Dabei kann es allerdings passieren, daß die Länge der verwendeten Warteschlange exponentiell wächst, was wiederum zu Speicherplatzproblemen führt.

Um das Problem zu lösen, muß die Tiefensuche dahingehend erweitert werden, daß die Suchtiefe beschränkt wird; d.h. hat der Stapel eine gewisse Tiefe T erreicht, so werden keine neuen Ecken mehr aufgelegt. Diese Art der Tiefensuche durchläuft somit alle Ecken, welche maximal den Abstand T von der Startecke haben. Diese Variante der Tiefensuche nennt man *beschränkte Tiefensuche*. Kann man eine obere Grenze T für die Länge des gesuchten Weges angeben, so findet die beschränkte Tiefensuche (mit Tiefenschranke $T > 0$) immer einen Weg zur Zielecke, sofern es einen gibt. Abbildung 4.31 zeigt eine Realisierung der beschränkten Tiefensuche. Dabei wird die Funktion `ziel` verwendet, welche `true` zurückliefert, falls die Ecke eine Zielecke ist. Dadurch ist es möglich, einen Weg zu einer Ecke aus einer Menge von Zielecken zu suchen.

Die beschränkte Tiefensuche findet nicht immer den kürzesten Weg von der Start- zur Zielecke. Um das zu erreichen, müßte man die Länge des kürzesten Weges zu einer Zielecke im voraus kennen. In diesem Fall würde die beschränkte Tiefensuche mit dem entsprechenden Parameter den kürzesten Weg finden. Um in jedem Fall den kürzesten Weg zu finden, kann man die beschränkte Tiefensuche wiederholt mit aufsteigenden Tiefenschranken aufrufen. Diese Form der Tiefensuche wird auch *iterative Tiefensuche* genannt. Abbildung 4.32 zeigt eine Realisierung der iterativen Tiefensuche.

```

procedure it-tiefensuche(G : G-Graph; start, T : Integer);
var
    t : Integer;
    q : Boolean;
begin
    Initialisiere t mit 1;
    repeat
        q := b-tiefensuche(G,start,t);
        t := t+1;
    until q = true or t > T;
    if q = true then
        exit('Ziel gefunden')
    else
        exit('Ziel nicht gefunden');
end

```

Abbildung 4.32: Eine Realisierung der iterativen Tiefensuche

Die iterative Tiefensuche hat natürlich den Nachteil, daß in jedem neuen Durchgang die im vorhergehenden Durchgang besuchten Ecken wieder besucht werden. Dies ist der Preis dafür, daß zum einen der kürzeste Weg gefunden wird, und zum anderen, daß die Suche mit einem Speicher der Größe $O(T)$ auskommt, wobei T die Länge des kürzesten Weges von der Start- zu einer Zielecke ist.

Im folgenden wird die Laufzeit der iterativen Tiefensuche für einen wichtigen Spezialfall untersucht. Der zu untersuchende Graph sei ein Wurzelbaum, bei dem der Ausgrad jeder Ecke gleich der Konstanten b ist. Die Wurzel sei die Startecke, und die Zielecke liege im T -ten Niveau. Die Ecken in dem Niveau $T - 1$ werden zweimal erzeugt: im letzten und im vorletzten Durchgang; die Ecken in dem Niveau $T - 2$ werden dreimal erzeugt etc. Die Gesamtanzahl der erzeugten Ecken ist somit

$$b^T + 2b^{T-1} + 3b^{T-2} + \dots + Tb.$$

Mit Hilfe einer einfachen Reihenentwicklung zeigt man, daß für $b > 1$ die Gesamtanzahl aller erzeugten Ecken durch

$$b^T \left(\frac{b}{b-1} \right)^2$$

beschränkt ist. Da b eine Konstante ist, die unabhängig von T ist, ist die Gesamtanzahl gleich $O(b^T)$. Da die Anzahl der Ecken auf dem T -ten Niveau gleich b^T ist, ist die iterative Tiefensuche für diese Graphen optimal.

4.12 Literatur

Breiten- und Tiefensuche sind als Suchverfahren in Graphen bereits im letzten Jahrhundert bekannt gewesen. Die hier verwendete Darstellungsform der Tiefensuche geht auf R.E. Tarjan [117] zurück. Dort sind auch die Algorithmen zur Bestimmung der starken Zusammenhangskomponenten eines gerichteten Graphen und der Blöcke eines ungerichteten Graphen beschrieben. Verbesserte Versionen dieser Algorithmen findet man in [100]. Ein effizientes Verfahren zur Bestimmung des transitiven Abschlusses basierend auf den starken Zusammenhangskomponenten, ist in [99] beschrieben. Es gibt noch weitere effiziente Algorithmen, die auf der Tiefensuche basieren. Zum Beispiel entwickelten J.E. Hopcroft und R.E. Tarjan einen Algorithmus mit Laufzeit $O(n)$, welcher testet, ob ein Graph planar ist [67]. Eine Beschreibung der Breitensuche findet man in [98]. Innerhalb der künstlichen Intelligenz wurden viele Suchverfahren für Graphen entwickelt. Eine genaue Untersuchung der beschränkten und der iterativen Tiefensuche findet man in [81].

4.13 Aufgaben

- Gegeben sind zwei ungerichtete Graphen durch ihre Adjazenzlisten. Wenden Sie den Algorithmus zur Tiefensuche auf diese beiden Graphen an! Verwenden Sie die Ecke 1 als Startecke. Numerieren Sie die Ecken in der Reihenfolge, in der sie besucht werden! Geben Sie den Tiefensuche-Wald an.

a) 1 → 2,4,8	2 → 1,3,4,5	3 → 2,4,7	4 → 1,2,3
5 → 2	6 → 7	7 → 3,6,8	8 → 1,7
b) 1 → 5,9	2 → 4,7,8	3 → 5,6,9	4 → 2,7,8

$$\begin{array}{llll} 5 \longrightarrow 1,3,6 & 6 \longrightarrow 3,5,9 & 7 \longrightarrow 2,4 & 8 \longrightarrow 2,4 \\ 9 \longrightarrow 1,3,6 & & & \end{array}$$

2. Gegeben sind zwei gerichtete Graphen durch ihre Adjazenzlisten. Wenden Sie den Algorithmus zur Tiefensuche auf diese beiden Graphen an! Verwenden Sie die Ecke 1 als Startecke. Numerieren Sie die Ecken in der Reihenfolge, in der sie besucht werden! Geben Sie den Tiefensuche-Wald an.

$$\begin{array}{llll} \text{a)} \quad 1 \longrightarrow 2,6,4 & 2 \longrightarrow 6,3 & 3 \longrightarrow 4 & 4 \longrightarrow 2 \\ \quad 5 \longrightarrow 4,6 & \quad 6 \longrightarrow 4 & & \\ \text{b)} \quad 1 \longrightarrow 2,4 & 2 \longrightarrow 3 & 3 \longrightarrow 4 & 4 \longrightarrow 2 \\ \quad 5 \longrightarrow 4,6 & \quad 6 \longrightarrow 4 & & \end{array}$$

3. Ändern Sie die Realisierung der Tiefensuche wie folgt ab: Jede Ecke wird mit zwei Zeitstempeln versehen; einen zu Beginn und einen am Ende des entsprechenden Tiefensucheauftrufs. Dazu wird das Feld **TSNummer** durch die Felder **TSB** und **TSE** ersetzt, welche beide mit 0 initialisiert werden. Die Prozedur **tiefensuche** wird wie folgt geändert:

```

procedure tiefensuche(i : Integer);
var
    j : Integer;
begin
    zähler := zähler + 1;
    TSB[i] := zähler;
    for jeden Nachbar j von i do
        if TSNummer[j] = 0 then
            tiefensuche(j);
    zähler := zähler + 1;
    TSE[i] := zähler;
end

```

Beweisen Sie, daß für diese Variante der Tiefensuche folgende Aussagen gelten:

- a) Sind e, f Ecken eines ungerichteten Graphen, so gilt genau eine der folgenden Bedingungen:
 - (i) Die Intervalle $[TSB[e], TSE[e]]$ und $[TSB[f], TSE[f]]$ sind disjunkt.
 - (ii) Das Intervall $[TSB[e], TSE[e]]$ ist vollständig in dem Intervall $[TSB[f], TSE[f]]$ enthalten.
 - (iii) Das Intervall $[TSB[f], TSE[f]]$ ist vollständig in dem Intervall $[TSB[e], TSE[e]]$ enthalten.
- b) Ist $k = (e, f)$ eine Kante eines gerichteten Graphen, so gilt
 - (i) k ist genau dann eine Baum- oder eine Vorwärtskante, wenn $TSB[e] < TSB[f] < TSE[f] < TSE[e]$ gilt;

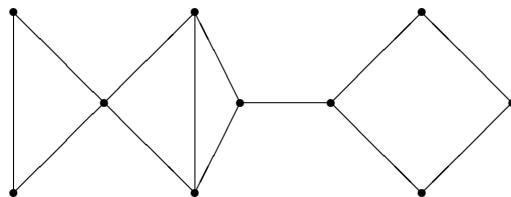
- (ii) k ist genau dann eine Rückwärtskante, wenn $\text{TSB}[f] < \text{TSB}[e] < \text{TSE}[e] < \text{TSE}[f]$ gilt;
- (iii) k ist genau dann eine Querkante, wenn $\text{TSB}[f] < \text{TSE}[f] < \text{TSB}[e] < \text{TSE}[e]$ gilt.
4. Es seien e, f Ecken in einem gerichteten Graphen G , so daß f von e aus erreichbar ist. Beweisen oder widerlegen Sie die folgende Behauptung: Falls $\text{TSNummer}[e] < \text{TSNummer}[f]$, so ist f im Tiefensuchewald von G von e aus erreichbar.
5. Ist die Numerierung des folgenden Digraphen eine topologische Sortierung?
- | | | | | | |
|-------------|-------------|-------------|-------------|-------------|-----------------|
| 1 → 2 | 2 → 4,5 | 3 → - | 4 → 3 | 5 → 3 | (Adjazenzliste) |
| 1: 1 | 2: 2 | 3: 4 | 4: 5 | 5: 3 | (Numerierung) |
6. Im folgenden ist die Adjazenzliste eines Digraphen gegeben. Geben Sie eine topologische Sortierung an! Wenden Sie den auf der Tiefensuche basierenden Algorithmus an!
- | | | | | |
|-----------|---------|---------|-------|---------|
| 1 → 3,5,2 | 2 → 6,7 | 3 → 4,5 | 4 → 7 | 5 → 4,6 |
| 6 → 4,7 | 7 → - | | | |
7. Untersuchen Sie die Graphen aus Aufgabe 2 daraufhin, ob sie eine topologische Sortierung besitzen. Wenn die topologischen Sortierungen existieren, so geben Sie diese an!
8. Stellen Sie fest, ob die folgenden gerichteten Graphen kreisfrei sind (gegeben sind die Adjazenzlisten). Wenden Sie dazu die Prozedur `topsort` an. Geben Sie die Aufrufhierarchie von `topsort` an. Für kreisfreie Graphen geben Sie zusätzlich eine topologische Sortierung an!
- 1 → 2,3 2 → - 3 → 2,4 4 → 1
 - 1 → 3,4,5,6 2 → 3,4,5,6 3 → 4 4 → 5 5 → 6 6 → -
 - 1 → 2,3 2 → 4 3 → 4 4 → 5,6 5 → 6 6 → -
9. Die Ecken eines gerichteten kreisfreien Graphen G seien so numeriert, daß die Numerierung eine topologische Sortierung bildet. Welche Eigenschaft hat die Adjazenzmatrix von G ?
10. Die von der Prozedur `tsprozedur` aus Abbildung 4.9 auf Seite 97 erzeugte topologische Sortierung hängt davon ab, in welcher Reihenfolge die Nachbarn der Ecken besucht werden. Gibt es für jede topologische Sortierung eines gerichteten Graphen eine Reihenfolge der Nachbarn jeder Ecke, so daß Prozedur `tsprozedur` diese topologische Sortierung erzeugt?
11. Entwerfen Sie einen Algorithmus, basierend auf der Tiefensuche, zur Bestimmung der Höhe eines Wurzelbaumes.

12. Bestimmen Sie die starken Zusammenhangskomponenten des Graphen aus Abbildung 4.1.
13. Beweisen Sie, daß der im folgenden beschriebene Algorithmus die starken Zusammenhangskomponenten eines gerichteten Graphen G in linearer Zeit $O(n + m)$ bestimmt.
- Wenden Sie die in Aufgabe 3 beschriebene Variante der Tiefensuche auf G an.
 - Bestimmen Sie den *invertierten Graphen* G' von G , indem Sie die Richtungen aller Kanten umdrehen.
 - Wenden Sie die Tiefensuche auf den Graphen G' an, wobei die Ecken in der Reihenfolge mit absteigenden Werten im Feld TSE (aus dem ersten Schritt) betrachtet werden.
 - Die beim letzten Schritt entstehenden Tiefensuchebäume entsprechen den starken Zusammenhangskomponenten von G .
14. Beweisen Sie folgende Aussage: In einem DAG gibt es eine Ecke, deren Ausgangsgrad gleich 0 ist, und eine Ecke, deren Eingangsgrad gleich 0 ist.
Entwerfen Sie einen Algorithmus zur Bestimmung einer topologischen Sortierung in einem DAG, welcher auf dieser Aussage basiert. Bestimmen Sie die Komplexität dieses Algorithmus und vergleichen Sie diese mit der des Algorithmus, welcher auf der Tiefensuche beruht.
15. Es sei E die Erreichbarkeitsmatrix eines gerichteten Graphen und d_{ii} der i -te Diagonaleintrag der Matrix E^2 . Beweisen Sie folgende Aussagen: Ist $d_{ii} = 0$, so bildet die Ecke i eine starke Zusammenhangskomponente und ist $d_{ii} \neq 0$, so ist d_{ii} gleich der Anzahl der Ecken der starken Zusammenhangskomponente, welche die Ecke i enthält.
16. Entwerfen Sie einen Algorithmus mit Laufzeit $O(n)$, welcher testet, ob ein ungerichteter Graph einen geschlossenen Weg enthält. Die Laufzeit soll unabhängig von der Anzahl der Kanten sein.
17. Im folgenden ist die Adjazenzliste eines ungerichteten Graphen in Form einer Tabelle gegeben. Finden Sie mit Hilfe der Tiefensuche die Zusammenhangskomponenten!

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	3	0	1	11	0	11	0	1	10	9	4	1	2	4
5	8	5	12	14	2	14	5		12	12	6	3	5	6
7		13			7		13			14	9	7	11	
						13					10			

18. Es sei G ein gerichteter Graph und a und b Ecken von G . Entwerfen Sie einen Algorithmus, welcher in linearer Zeit alle Ecken e und die zu ihnen inzidenten Kanten aus G entfernt, die die folgende Eigenschaft haben: Die Ecke b ist nicht von e aus erreichbar, oder die Ecke e ist nicht von a aus erreichbar.

- * 19. Ändern Sie den Algorithmus zur Bestimmung der Blöcke eines ungerichteten Graphen ab, so daß die trennenden Ecken bestimmt werden.
- 20. Bestimmen Sie den Blockgraphen des folgenden Graphen:



- 21. Es sei G ein ungerichteter Graph. Die Anzahl der Blöcke von G sei b und die der trennenden Ecken sei t . Beweisen Sie, daß folgende Ungleichung gilt: $b \geq t + 1$.
- 22. Beweisen Sie, daß jeder zusammenhängende ungerichtete Graph mindestens zwei Ecken besitzt, welche keine trennenden Ecken sind, sofern er insgesamt mindestens zwei Ecken besitzt.
- 23. Geben Sie die Breitensuche-Wälder für die ungerichteten Graphen aus Aufgabe 1 an!
- 24. Es sei G ein gerichteter Graph. Entwerfen Sie ein Verfahren, welches in linearer Zeit alle Kanten von G bestimmt, welche nicht auf einem geschlossenen Weg von G liegen.
- * 25. Entwerfen Sie einen Algorithmus zur Bestimmung des transitiven Abschlusses eines gerichteten Graphen auf der Basis der Breitensuche. Bestimmen Sie die Zeitkomplexität des Algorithmus!
- 26. Entwerfen Sie einen Algorithmus zur Bestimmung der Länge des kürzesten geschlossenen Weges in einem ungerichteten Graphen auf der Basis der Breitensuche. Bestimmen Sie die Zeitkomplexität des Algorithmus!
- 27. Entwerfen Sie einen effizienten Algorithmus zur Bestimmung des transitiven Abschlusses eines gerichteten kreisfreien Graphen.
- 28. Geben Sie Beispiele für ungerichtete Graphen mit n Ecken, deren Tiefensuchebäume die Höhe $n - 1$ bzw. 1 haben. Wie sehen die Breitensuchebäume dieser Graphen aus?
- 29. Entwerfen Sie einen Algorithmus, basierend auf der Breitensuche, zur Bestimmung der Zusammenhangskomponenten eines ungerichteten Graphen.
- 30. Die Wege in einem Breitensuchebaum für einen gerichteten Graphen von der Startecke zu den anderen Ecken sind kürzeste Wege, d.h. sie haben die minimale Anzahl von Kanten. Der Breitensuchebaum enthält für jede Ecke aber nur einen solchen Weg. Es kann aber zu einer Ecke von der Startecke aus mehrere verschiedene Wege mit der minimalen Anzahl von Kanten geben. Verändern Sie die Prozedur **breitensuche**, so daß ein Graph entsteht, der gerade aus allen kürzesten Wegen

von der Startecke zu allen anderen Ecken besteht. Dieser Graph ist die „Vereinigung“ aller möglichen Breitensuchebäume mit der gleichen Startecke. Welche Laufzeit hat die neue Prozedur?

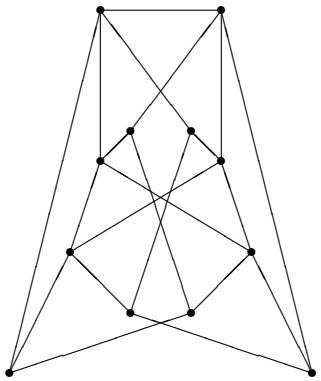
- * 31. Es sei G ein zusammenhängender ungerichteter Graph und B ein aufspannender Baum von G . Ferner sei e eine beliebige Ecke von G , und alle Kanten von B seien so gerichtet, daß e eine Wurzel von B ist. Beweisen Sie folgende Aussagen:
 - a) B ist ein Tiefensuchebaum mit Startecke e , falls für jede Kante (u, v) von G entweder u Nachfolger von v oder v Nachfolger von u in B ist.
 - b) B ist ein Breitensuchebaum mit Startecke e , falls für jede Kante (u, v) von G die Abstände $d(e, u)$ und $d(e, v)$ in B sich maximal um 1 unterscheiden.
- * 32. Betrachten Sie einen Wurzelbaum B , bei dem der Ausgrad jeder Ecke gleich der Konstanten b ist. Die Höhe von B sei gleich C , und z sei eine Ecke mit Abstand T von der Wurzel. Gesucht ist der kürzeste Weg von der Wurzel zur Ecke z . Vergleichen Sie den Speicheraufwand und die Laufzeit von Breitensuche, Tiefensuche und iterativer Tiefensuche für dieses Problem.
- 33. Es sei T ein Tiefensuchebaum eines ungerichteten Graphen. Beweisen Sie, daß die Menge der Blätter von T eine unabhängige Menge ist.
- 34. Ein boolescher Schaltkreis kann als ein gerichteter, azyklischer Graph modelliert werden. Es gibt mehrere Arten von Ecken:

Eckenart	Eingrad	Ausgrad	Anzahl	Wert
Eingabeecke	1	≥ 1	≥ 1	Wert der Variablen
Ausgabeecke	1	1	1	Eingehender Wert
Verarbeitungsecke	1 oder 2	1	≥ 1	Logische Verknüpfung der eingehenden Werte

Es gibt drei Arten von Verarbeitungsecken: Konjunktion, Disjunktion und Negation. Die Eingabeecken sind mit Variablen x_1, \dots, x_k markiert. Für jede Belegung $(w_1, \dots, w_k) \in \{0, 1\}^k$ der Variablen kann jeder Ecke ein Wert zugewiesen werden, dieser berechnet sich wie oben dargestellt aus den Werten der Vorgängerecken. Der Wert des Schaltkreises ist gleich dem Wert der Ausgabeecke. Entwerfen Sie einen Algorithmus, der in linearer Zeit den Wert eines Schaltkreises für eine gegebene Belegung der Variablen bestimmt.

Kapitel 5

Färbung von Graphen



Am Anfang der Graphentheorie standen spezifische Probleme, die leicht, d.h. ohne großen Formalismus, zu beschreiben waren. Beispiele hierfür sind das *Königsberger Brückenproblem* und das *Vier-Farben-Problem*. Während das erste Problem sich leicht lösen ließ, wurde das zweite Problem erst in jüngster Vergangenheit nach etlichen vergeblichen Anläufen gelöst. In diesem Kapitel werden Algorithmen zur Bestimmung von minimalen Färbungen vorgestellt. Für allgemeine Graphen wird mit dem Backtracking-Algorithmus ein Verfahren vorgestellt, welches sich auch auf viele andere Probleme anwenden lässt. Dieses Verfahren hat allerdings eine exponentielle Laufzeit. Für einige spezielle Klassen von Graphen existieren effiziente Algorithmen. In diesem Kapitel werden solche Algorithmen für planare Graphen und transitiv orientierbare Graphen diskutiert. Im letzten Abschnitt werden Färbungen von Permutationsgraphen behandelt.

5.1 Einführung

Eine *Färbung* eines Graphen ist eine Belegung der Ecken mit Farben, so daß je zwei benachbarte Ecken verschiedene Farben erhalten. Eine c -*Färbung* ist eine Färbung, die c verschiedene Farben verwendet, d.h. es ist eine Abbildung f von der Menge der Ecken

des Graphen in die Menge $\{1, 2, \dots, c\}$, so daß benachbarte Ecken verschiedene Zahlen zugeordnet bekommen (d.h. die Farben werden durch Zahlen repräsentiert). Für einen Graphen gibt es im allgemeinen verschiedene Färbungen mit unterschiedlichen Anzahlen von Farben. Die *chromatische Zahl* $\chi(G)$ eines Graphen G ist die kleinste Zahl c , für welche der Graph eine c -Färbung besitzt. Abbildung 5.1 zeigt einen Graphen, der die chromatische Zahl 4 hat und eine entsprechende Färbung. Die Markierungen an den Ecken repräsentieren die Farben. Man überzeugt sich leicht, daß der Graph keine 3-Färbung besitzt. Eine Färbung, die genau $\chi(G)$ Farben verwendet, nennt man eine *minimale Färbung*.

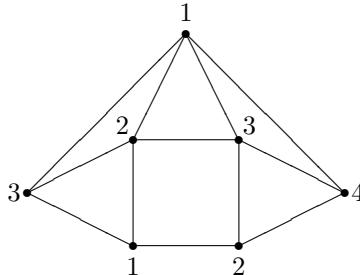


Abbildung 5.1: Ein Graph mit chromatischer Zahl 4

Für einen Graphen G mit $\chi(G) = c$ kann es mehrere verschiedene c -Färbungen geben. Viele c -Färbungen lassen sich durch Vertauschung der Farben ineinander überführen. Die Bestimmung der chromatischen Zahl eines Graphen ist eine schwierige Aufgabe. Es gilt $\chi(K_n) = n$, $\chi(C_{2n}) = 2$ und $\chi(C_{2n+1}) = 3$. Die Graphen mit $\chi(G) = 2$ sind gerade die bipartiten Graphen. Ein Algorithmus, der Graphen mit chromatischer Zahl 2 erkennt, wurde schon im Kapitel 4 vorgestellt. Er basiert auf der Breitensuche. Es ergibt sich sofort folgende Charakterisierung von Graphen mit chromatischer Zahl 2:

Lemma. Ein ungerichteter Graph G hat genau dann eine 2-Färbung, wenn er keinen geschlossenen Weg mit einer ungeraden Kantenzahl enthält.

Beweis. Nach dem im Abschnitt 4.10 bewiesenen Lemma ist G genau dann bipartit, wenn für jede Kante (e, f) von G gilt: $Niv(e) + Niv(f) \equiv 1(2)$. Daraus folgt, daß die Graphen C_{2n+1} nicht bipartit sind. Somit kann es in einem bipartiten Graphen auch keine geschlossenen Wege mit einer ungeraden Kantenzahl geben. Es sei nun G ein Graph, der keinen geschlossenen Weg mit einer ungeraden Kantenzahl enthält. Ist (e, f) eine Baumkante, so liegen e und f in benachbarten Niveaus. Andernfalls wird durch (e, f) in dem Breitensuchebaum ein geschlossener Weg gebildet; dieser hat eine gerade Anzahl von Kanten. Somit liegen auch in diesem Fall e und f in benachbarten Niveaus. Daraus folgt $Niv(e) + Niv(f) \equiv 1(2)$. Somit ist G bipartit und $\chi(G) = 2$. ■

Das Erkennen von Graphen mit chromatischer Zahl 3 ist dagegen viel schwieriger. Bisher gibt es weder eine einfache Charakterisierung dieser Graphen, noch ist ein effizienter Algorithmus zum Erkennen dieser Graphen bekannt. In Kapitel 9 wird gezeigt, daß es sogar viele Anzeichen dafür gibt, daß es keinen effizienten Algorithmus für dieses

Problem geben kann.

Im folgenden werden zunächst obere und untere Schranken für die chromatische Zahl eines Graphen angegeben. Eine triviale obere Schranke für $\chi(G)$ ist die Anzahl der Ecken, d.h. für einen Graphen mit n Ecken gilt $\chi(G) \leq n$. Eine n -Färbung erhält man, indem man jeder Ecke eine andere Farbe zuordnet.

Gibt es in einem Graphen eine Menge C von Ecken, so daß der von C induzierte Untergraph der vollständige Graph K_c ist, so gilt $\chi(G) \geq c$. Umgekehrt muß aber ein Graph mit $\chi(G) = c$ keinen vollständigen Graphen mit c Ecken enthalten. Der Graph in Abbildung 5.1 hat die chromatische Zahl 4, enthält aber nicht den Graphen K_4 .

Eine Teilmenge C der Eckenmenge eines ungerichteten Graphen G heißt *Clique*, falls der von C induzierte Untergraph von G vollständig ist. Die *Cliquenzahl* $\omega(G)$ ist die Mächtigkeit einer Clique von G mit den meisten Ecken. Der Graph in Abbildung 5.1 hat die Cliquenzahl 3. Es gilt $\omega(K_n) = n$, $\omega(C_3) = 3$ und $\omega(C_n) = 2$ falls $n = 2$ oder $n \geq 4$. Die Ecken einer Clique müssen bei einer Färbung des Graphen alle verschiedene Farben erhalten. Somit gilt

$$\chi(G) \geq \omega(G).$$

Der Graph aus Abbildung 5.1 zeigt, daß $\chi(G)$ echt größer als $\omega(G)$ sein kann. J. Mycielski hat eine Serie von Graphen G_c mit $c \in \mathbb{N}$ konstruiert, so daß $\omega(G_c) = 2$ und $\chi(G_c) = c$ ist (vergleichen Sie Aufgabe 27). Somit besteht im allgemeinen kein direkter Zusammenhang zwischen der Cliquenzahl und der chromatischen Zahl eines Graphen. Hinzu kommt, daß bis heute kein effizienter Algorithmus zur Bestimmung der Cliquenzahl eines Graphen bekannt ist.

Ist Δ der maximale Grad einer Ecke von G , so gilt

$$\chi(G) \leq 1 + \Delta.$$

Diese Ungleichung kann mit folgendem einfachen Verfahren bewiesen werden: Die Ecken werden in irgendeiner Reihenfolge betrachtet, und jeder Ecke wird die kleinste Farbe zugeordnet, die verschieden zu den Farben der schon gefärbten Nachbarn ist. Auf diese Art erzeugt man eine Färbung, welche maximal $\Delta + 1$ Farben verwendet, denn für jede Ecke können maximal Δ Farben verboten sein.

Die Realisierung dieses Algorithmus ist einfach. Die Ecken werden in irgendeiner Reihenfolge durchlaufen, für jede Ecke wird die Menge der schon gefärbten Nachbarn betrachtet und die kleinste nicht vorkommende Farbnummer bestimmt. Diese ist dann die Farbnummer der aktuellen Ecke. Bei unvorsichtiger Realisierung kann sich eine Laufzeit von $O(n^2)$ ergeben. Der entscheidende Punkt, um eine lineare Laufzeit $O(n + m)$ zu erzielen, ist die Bestimmung der kleinsten unbenutzten Farbnummer unter den schon gefärbten Nachbarn einer Ecke i .

Im folgenden wird gezeigt, daß dieser Schritt mit Aufwand $O(g(i))$ realisiert werden kann. Die Grundidee ist die Verwaltung eines Feldes *vergeben* der Länge $\Delta + 1$. In diesem Feld werden in jedem Durchgang die schon an die Nachbarn einer Ecke vergebenen Farben eingetragen. Hierbei bedeutet der Eintrag 1, daß die Farbe schon vergeben

ist. Der entscheidende Punkt ist der, daß dieses Feld nicht in jedem Durchgang pauschal auf 0 gesetzt wird. Dies würde nämlich zu einem Aufwand von $O(\Delta n)$ führen. In jedem Durchgang werden zunächst die vergebenen Farben markiert und später wird diese Markierung wieder rückgängig gemacht. Somit kann dieser Schritt mit Aufwand $O(g(i))$ realisiert werden. Die Bestimmung der kleinsten nicht vergebenen Farbnummer (d.h. die Suche nach der ersten 0 im Feld `vergeben`) kann mit dem gleichen Aufwand realisiert werden, denn spätestens der Eintrag mit der Nummer $g(i) + 1$ ist ungleich 1. Insgesamt ergibt dies den Aufwand $O(n + m)$. Abbildung 5.2 zeigt eine Realisierung dieses Algorithmus.

```

var f : array[1..max] of Integer;

function greedy-färbung(G : Graph) : Integer;
var
    i, j, k, χ : Integer;
    vergeben array[1..Δ + 1] of Integer;
begin
    Initialisiere f, vergeben und χ mit 0;
    f[1] := 1;
    for i := 2 to n do begin
        for jeden Nachbar j von i do
            if f[j] > 0 then
                vergeben[f[j]] := 1;
        k := 1;
        while vergeben[k] = 1 do
            k := k + 1;
        f[i] := k;
        if k > χ then
            χ := k;
        for jeden Nachbar j von i do
            if f[j] > 0 then
                vergeben[f[j]] := 0;
    end;
    greedy-färbung := χ;
end

```

Abbildung 5.2: Die Funktion `greedy-färbung`

Die Funktion `greedy-färbung` produziert eine Färbung f eines ungerichteten Graphen. Wie bei allen Greedy-Algorithmen wird eine einmal getroffene Entscheidung über die Farbe einer Ecke nicht mehr revidiert. Wendet man die Funktion `greedy-färbung` auf den Graphen G aus Abbildung 5.3 an, so wird eine Färbung mit vier Farben erzeugt. Da der Graph G bipartit ist, gilt aber $\chi(G) = 2$. Somit liefert `greedy-färbung` zwar eine Färbung, aber nicht unbedingt eine minimale Färbung.

Das Ergebnis ist stark geprägt von der Reihenfolge, in der die Ecken durchsucht werden. Vertauscht man die Nummern der Ecken 3 und 4, so liefert der Greedy-Algorithmus

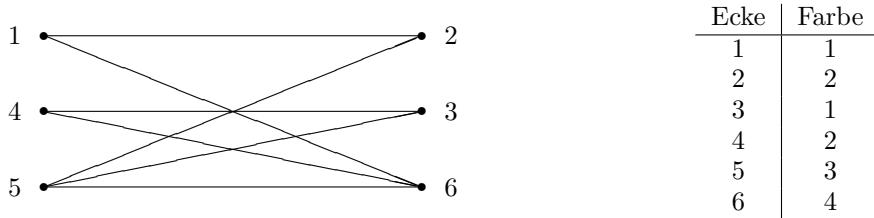


Abbildung 5.3: Ein bipartiter Graph für den der Greedy-Algorithmus eine 4-Färbung produziert

eine 2-Färbung, d.h. die optimale Lösung. Man kann sogar zeigen, daß es für jeden Graphen Reihenfolgen gibt, für die der Greedy-Algorithmus eine minimale Färbung erzeugt. Allgemein gilt der folgende Satz:

Satz. Für jeden Graphen G mit $\chi(G) = c$ gibt es eine Numerierung der Ecken, so daß der Greedy-Algorithmus eine c -Färbung liefert.

Beweis. Man betrachte eine c -Färbung von G . Die Ecken von G werden nun aufsteigend numeriert; zuerst die Ecken mit Farbe 1, dann die mit Farbe 2 etc. Die Reihenfolge innerhalb einer Farbe ist beliebig. Für diese Reihenfolge liefert der Greedy-Algorithmus eine c -Färbung. ■

Das Verhalten des Greedy-Algorithmus wird in Kapitel 9 näher untersucht. Dort wird unter anderem eine obere Grenze für die Anzahl der vergebenen Farben in Abhängigkeit der Anzahl der Kanten angegeben. Der Greedy-Algorithmus beweist, daß $\chi(G) \leq 1 + \Delta$ ist. Die zyklischen Graphen C_n mit ungeradem n werden durch den Greedy-Algorithmus unabhängig von der Reihenfolge der Ecken immer optimal gefärbt (man beachte $\chi(C_n) = 3$ und $\Delta(C_n) = 2$ für ungerades n).

Für zusammenhängende Graphen ist $\chi(G) = 1 + \Delta$ nur dann, wenn G vollständig oder $G = C_{2n+1}$ ist. Im letzten Fall ist $\Delta = 2$. Dies wird in dem folgenden Satz bewiesen.

Satz (BROOKS). Es sei G ein ungerichteter Graph mit n Ecken. Ist $\Delta(G) \geq 3$ und enthält G nicht den vollständigen Graphen $K_{\Delta+1}$, so ist $\chi(G) \leq \Delta$.

Beweis. Der Beweis erfolgt durch vollständige Induktion nach n . Wegen $\Delta \geq 3$ muß $n \geq 4$ sein. Für $n = 4$ erfüllen nur die in Abbildung 5.4 dargestellten drei Graphen die Voraussetzungen des Satzes. Die chromatischen Zahlen dieser Graphen sind 2, 3 und 3. Somit gilt der Satz für $n = 4$.

Es sei nun G ein Graph mit $n > 4$ Ecken, der die Voraussetzungen des Satzes erfüllt. Es kann angenommen werden, daß G zusammenhängend ist. Gibt es in G eine Ecke e , deren Eckengrad kleiner als Δ ist, so ist $\chi(G \setminus \{e\}) \leq \Delta$ nach Induktionsannahme. Daraus folgt, daß auch $\chi(G) \leq \Delta$ ist. Somit kann weiter angenommen werden, daß der Eckengrad jeder Ecke gleich Δ ist. Da G nicht vollständig ist, gilt $\Delta < n - 1$.

Angenommen es gibt in G eine Ecke e , so daß $G \setminus \{e\}$ in zwei Graphen G_1 und G_2 zerfällt, d.h., es gibt keine Kante von G_1 nach G_2 . Nach Induktionsvoraussetzung ist $\chi(G_1 \cup \{e\}) \leq \Delta$ und $\chi(G_2 \cup \{e\}) \leq \Delta$. Daraus folgt, daß auch $\chi(G) \leq \Delta$ ist (man muß eventuell einige Farben vertauschen). Somit kann es keine solche Ecke in G geben.

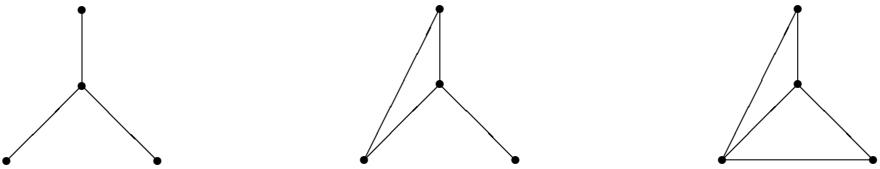


Abbildung 5.4: Die Graphen mit vier Ecken und maximalem Eckengrad 3

Im folgenden wird nun der Fall betrachtet, daß es eine Ecke e mit folgender Eigenschaft gibt: Unter den Nachbarn von e gibt es zwei nichtbenachbarte Ecken a und b , so daß $G \setminus \{a, b\}$ zusammenhängend ist. In diesem Fall wendet man die Tiefensuche mit Startecke e auf den Graphen $G \setminus \{a, b\}$ an. Die Ecken von $G \setminus \{a, b\}$ werden mit ihren Tiefensuchenummern numeriert. Die Ecken von $G \setminus \{a, b\}$ haben folgende Bezeichnung $e_1 = e, e_2, \dots, e_{n-2}$. Es läßt sich nun eine Δ -Färbung von G angeben, indem die Ecken in umgekehrter Reihenfolge gefärbt werden. Zuvor bekommen die Ecken a und b die Farbe 1. Jede Ecke $e_i \neq e_1$ ist mindestens zu einer Ecke e_j mit $j < i$ benachbart. Wegen $g(e_i) \leq \Delta$ haben höchstens $\Delta - 1$ der Nachbarn von e_i schon eine Farbe. Somit verbleibt eine Farbe zur Färbung von e_i . Die Ecke e_1 ist zu a und b benachbart. Da beide die Farbe 1 haben, verwenden die Nachbarn von e_1 maximal $\Delta - 1$ Farben. Somit kann auch e_1 gefärbt werden, und man hat gezeigt, daß $\chi(G) \leq \Delta$ ist.

Es bleibt noch, die Existenz von e, a und b zu zeigen. Da G nicht der vollständige Graph ist, gibt es Ecken x, c und d , so daß c zu x und d benachbart ist, und d und x untereinander nicht benachbart sind. Ist $G \setminus \{x\}$ ein zweifach zusammenhängender Graph, so ist $G \setminus \{x, d\}$ zusammenhängend. Nun wähle man $a = x, b = d$ und $e = c$. Es bleibt noch der Fall, daß $G \setminus \{x\}$ nicht zweifach zusammenhängend ist. Dazu betrachte man in dem Blockbaum von $G \setminus \{x\}$ zwei Blätter. Es seien A und B die zugehörigen Blöcke von $G \setminus \{x\}$. Da G zweifach zusammenhängend ist, gibt es nichtbenachbarte Ecken $a \in A$ und $b \in B$, die zu x benachbart sind. Wegen $g(x) = \Delta \geq 3$ ist $G \setminus \{a, b\}$ zusammenhängend. Mit $e = x$ hat man die gewünschten Ecken gefunden. Damit ist der Beweis vollständig.

Der maximale Eckengrad eines ungerichteten Graphen beschränkt somit die chromatische Zahl. Auf der anderen Seite gibt es zu je zwei Zahlen Δ, c mit $\Delta \geq c \geq 2$ einen Graphen G mit maximalem Eckengrad Δ und $\chi(G) = c$. Dazu geht man von dem vollständigen Graphen K_c und einem sternförmigen Graphen S mit $\Delta - 1$ Ecken aus. Mit einer zusätzlichen Kante zwischen einer beliebigen Ecke von K_c und dem Zentrum von S entsteht ein Graph mit der gewünschten Eigenschaft. Abbildung 5.5 zeigt diese Konstruktion.

Um zu einer weiteren Abschätzung der chromatischen Zahl eines Graphen zu kommen, wird der Begriff der unabhängigen Menge benötigt. Eine Teilmenge U der Eckenmenge

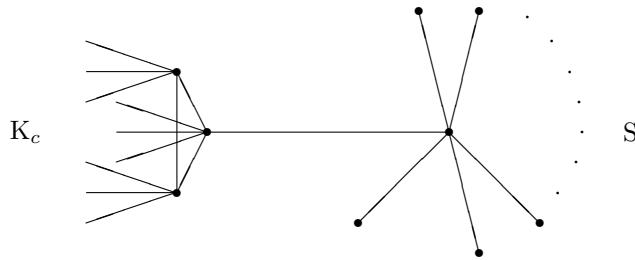


Abbildung 5.5: Ein Graph mit chromatischer Zahl c und beliebig hohem maximalem Eckengrad

eines ungerichteten Graphen G heißt *unabhängig*, falls keine zwei Ecken aus U benachbart sind. Die *Unabhängigkeitszahl* $\alpha(G)$ von G ist die maximale Mächtigkeit einer unabhängigen Menge.

Für den Graphen aus Abbildung 5.6 bilden die Ecken 2, 4 und 6 eine unabhängige Menge. Man sieht sofort, daß es keine unabhängige Menge mit vier Ecken gibt, und somit ist $\alpha(G) = 3$. Es gilt $\alpha(K_n) = 1$, $\alpha(C_{2n+1}) = n$, $\alpha(C_{2n}) = n$ und $\alpha(G) = \omega(\bar{G})$.

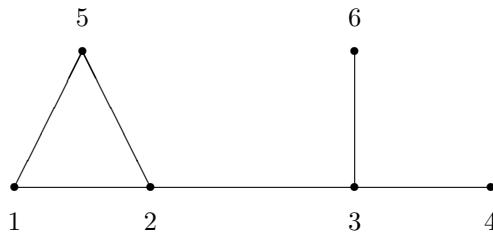


Abbildung 5.6: Ein Graph G mit $\alpha(G) = 3$ und $\chi(G) = 3$

Mit Hilfe von $\alpha(G)$ kann man folgende Abschätzung für $\chi(G)$ beweisen.

Lemma. Für einen ungerichteten Graphen G gilt

$$\frac{n}{\alpha(G)} \leq \chi(G) \leq n - \alpha(G) + 1.$$

Beweis. Es sei f eine Färbung von G , welche $\chi(G)$ Farben verwendet. Für $i = 1, \dots, \chi(G)$ sei E_i die Menge der Ecken von G mit Farbe i . Die E_i sind unabhängige Mengen und $|E_i| \leq \alpha(G)$. Hieraus folgt

$$n = \sum_{i=1}^{\chi(G)} |E_i| \leq \chi(G)\alpha(G).$$

Somit ist die untere Abschätzung bewiesen.

Sei nun U eine unabhängige Menge von G mit $|U| = \alpha(G)$. Sei G' der von den nicht in U liegenden Ecken induzierte Untergraph. Sicherlich ist $\chi(G') \geq \chi(G) - 1$ und $\chi(G') \leq |G'| = n - \alpha(G)$. Daraus folgt $\chi(G) \leq \chi(G') + 1 \leq n - \alpha(G) + 1$. ■

Die folgenden beiden Beispiele zeigen, daß die bewiesenen Ungleichungen beliebig schlecht werden können. Verbindet man jede Ecke des vollständigen Graphen K_s mit einer weiteren Ecke, so ergibt sich ein Graph G mit $2s$ Ecken und $\alpha(G) = s$. Somit ist $n/\alpha(G) = 2$, während $\chi(G) = s$ ist. Für den vollständig bipartiten Graphen $K_{s,s}$ gilt $n - \alpha(K_{s,s}) + 1 = s + 1$ und $\chi(K_{s,s}) = 2$.

Algorithmen zur Bestimmung der chromatischen Zahl eines Graphen und einer minimalen Färbung sind sehr zeitaufwendig. Alle bisher bekannten Algorithmen besitzen eine Laufzeit, die exponentiell von der Anzahl der Ecken des Graphen abhängt. Somit sind diese Algorithmen für praktische Probleme nur begrenzt brauchbar. Alle bekannten Algorithmen zur Feststellung, ob ein Graph eine c -Färbung besitzt, probieren im Prinzip alle verschiedenen Zuordnungen von Farben zu Ecken durch, bis eine c -Färbung gefunden wurde. Es wird vermutet, daß es keinen Algorithmus gibt, der substantiell schneller ist. Von praktischer Bedeutung sind deshalb Algorithmen, die Färbungen produzieren, die nicht unbedingt minimal sind; diese benötigen dann mehr als $\chi(G)$ Farben. Allgemein werden Algorithmen, die nicht notwendigerweise eine optimale Lösung produzieren, *Heuristiken* oder *approximative Algorithmen* genannt. Approximative Algorithmen zum Färben von Graphen werden in Kapitel 9 diskutiert. Klassen von Graphen, für die effiziente Algorithmen zur Bestimmung minimaler Färbungen bekannt sind, werden später in diesem Kapitel vorgestellt.

5.2 Anwendungen von Färbungen

Im folgenden werden aus verschiedenen Bereichen Anwendungen dargestellt, die auf ein Färbungsproblem zurückgeführt werden können. In Kapitel 1 wurde bereits eine Anwendung aus dem Bereich von objektorientierten Programmiersprachen diskutiert.

5.2.1 Maschinenbelegungen

Zur Durchführung einer Menge von Aufgaben A_1, \dots, A_n werden verschiedene Maschinen eingesetzt. Die Durchführung der Aufgaben A_i erfordert jeweils die gleiche Zeit T . Die einzelnen Aufgaben können gleichzeitig durchgeführt werden, sofern nicht die selben Maschinen benötigt werden. Ziel der Maschinenbelegung ist es, eine Reihenfolge der Abarbeitung der Aufgaben zu finden, die diese Einschränkung beachtet und zeitoptimal ist; d.h. es wird ein hoher Grad an Parallelität angestrebt.

Die Ausschlußbedingungen zwischen den Aufgaben lassen sich als Graph, dem sogenannten *Konfliktgraphen*, interpretieren: Jede Ecke entspricht einer Aufgabe, und zwei Ecken sind benachbart, falls die Aufgaben nicht gleichzeitig durchgeführt werden können. Eine zulässige Reihenfolge der Aufgaben impliziert eine Färbung des Konfliktgraphen, indem Ecken, deren Aufgaben gleichzeitig durchgeführt werden, die gleiche Farbe zugeordnet

bekommen. Umgekehrt impliziert jede Färbung des Konfliktgraphen eine zulässige Reihenfolge der Aufgaben, indem man eine beliebige Reihenfolge der Farben wählt und Aufgaben mit gleicher Farbe gleichzeitig durchführt. Eine optimale Lösung wird somit erreicht, wenn eine Färbung mit minimaler Farbenzahl gefunden wird. Die minimale Durchführungszeit für alle Aufgaben ist somit gleich dem Produkt von T und der chromatischen Zahl des Konfliktgraphen. Abbildung 5.7 zeigt einen Konfliktgraphen mit einer optimalen Färbung und die entsprechende Maschinenbelegung.

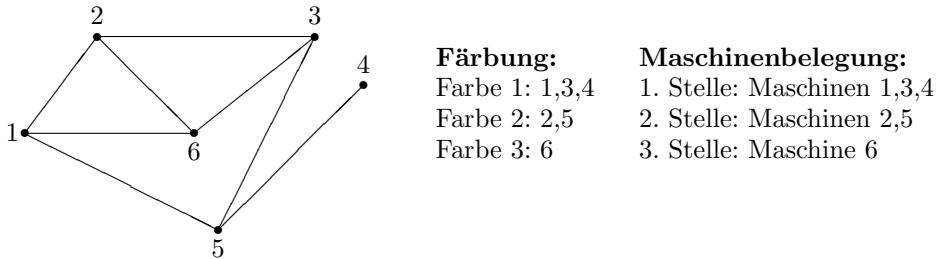


Abbildung 5.7: Ein Konfliktgraph mit einer Färbung

5.2.2 Registerzuordnung in Compilern

Die Phase der Registervergabe eines Compilers ist zwischen der Optimierungsphase und der eigentlichen Codeerzeugungsphase angesiedelt. Das Quellprogramm wurde in eine sogenannte Zwischensprache transformiert, die auf der Annahme basiert, daß eine unendliche Anzahl von symbolischen Registern zur Verfügung steht. Während der Registervergabe werden den symbolischen Registern physikalische Register zugewiesen. Dazu wird ein *Registerinterferenzgraph* erzeugt. In diesem stellen die Ecken symbolische Register dar, und eine Kante verbindet zwei Ecken, falls das eine Register an dem Punkt aktiv ist, an dem das andere benötigt wird. Ist k die Anzahl der zur Verfügung stehenden Register, so wird versucht, den Registerinterferenzgraph mit k Farben zu färben. Jede Farbe stellt ein Register dar, und die Färbung gewährleistet, daß kein Paar von symbolischen Registern, die aufeinandertreffen können, dem gleichen physikalischen Register zugewiesen werden. In vielen Fällen wird es aber keine k -Färbung geben. Da Algorithmen, die entscheiden, ob ein Graph eine k -Färbung besitzt, zu zeitaufwendig für den Einsatz in Compilern sind, werden Heuristiken verwendet, um zu Lösungen zu gelangen. Eine solche Heuristik liefert die folgende rekursive Vorgehensweise.

Man wähle eine Ecke e aus dem Registerinterferenzgraphen G mit $g(e) < k$, entferne e und alle zu e inzidenten Kanten aus G und färbe den so entstandenen Graphen. Danach wähle man eine Farbe, die nicht unter den Nachbarn von e vorkommt (eine solche Farbe existiert, da $g(e) < k$) und färbe damit die Ecke e . Gilt zu einem Zeitpunkt, daß alle Ecken mindestens den Eckengrad k haben, so muß der Registerinterferenzgraph verändert werden. Dies geschieht dadurch, daß ein Register ausgewählt wird und zusätzlicher Code eingefügt wird, der den Inhalt aus diesem Register in den Speicher transportiert und an den entsprechenden Stellen wieder lädt. Danach wird der Regi-

sterinterferenzgraph diesem neuen Code angepaßt, und die entsprechende Ecke wird mit ihren Kanten entfernt. Dies hat zur Folge, daß die Eckengrade der Nachbarn der entfernten Ecke sich um 1 erniedrigen. Zur Auswahl der Ecken werden wiederum Heuristiken verwendet, die die zusätzliche Ausführungszeit abschätzen und versuchen, diese zu minimieren. Dieser Prozeß wird so lange wiederholt, bis eine Ecke e mit $g(e) < k$ existiert. Danach kann der Färbungsprozeß wieder aufgenommen werden.

5.2.3 Public-Key Kryptosysteme

Wird über ein Computernetzwerk Geschäftsverkehr abgewickelt, so muß sich jeder Benutzer eindeutig identifizieren können, und jeder muß vor Fälschung seiner Unterschrift sicher sein. Diese Forderungen können mit sogenannten Public-Key Kryptosystemen erfüllt werden. Diese Systeme beruhen auf sogenannten *Einwegfunktionen*, deren Funktionswerte leicht zu bestimmen sind, deren Umkehroperationen aber einen solchen Aufwand benötigen, daß sie praktisch nicht durchführbar sind. Das bekannteste Public-Key Kryptosystem ist das RSA-System. Dies beruht auf der Tatsache, daß es zur Zeit praktisch nahezu unmöglich ist, sehr große natürliche Zahlen in ihre Primfaktoren zu zerlegen. An dieser Stelle kann man sich zu nutze machen, daß alle bekannten Algorithmen zur Färbung von Graphen einen exponentiellen Aufwand haben, und damit für große Graphen sehr aufwendig sind. Im folgenden wird ein theoretisches Kryptosystem, basierend auf Graphfärbungen, beschrieben.

Zur Identifikation von Benutzern in einem Computersystem wählt sich jeder Benutzer einen Graphen mit chromatischer Zahl 3. Dieser Graph wird in eine Liste zusammen mit dem Namen des Benutzers eingetragen. Diese Liste braucht nicht geheim zu bleiben, lediglich die explizite Färbung muß der Benutzer geheim halten. Zur Identifikation eines Benutzers gibt dieser seinen Namen bekannt und beweist seine Identität, in dem er zeigt, daß er eine 3-Färbung des entsprechenden Graphen kennt. Da es auch für Graphen mit chromatischer Zahl 3 praktisch sehr aufwendig ist, eine 3-Färbung zu finden, ist diese Methode so gut wie fälschungssicher. Auf der anderen Seite ist es sehr einfach, Graphen zu konstruieren, die eine 3-Färbung besitzen und diese Färbung auch anzugeben. Dabei geht man induktiv vor. Ist ein Graph mit einer 3-Färbung gegeben, so erzeugt man eine neue Ecke und gibt dieser eine beliebige der drei Farben. Anschließend verbindet man diese Ecke mit einer beliebigen Teilmenge der existierenden Ecken, die eine der anderen beiden Farben haben. Diesen Prozeß wiederholt man so lange, bis der Graph genügend Ecken hat.

Es bleibt noch zu zeigen, wie man beweisen kann, daß man eine 3-Färbung eines Graphen kennt, ohne dabei diese explizit bekannt zu geben. Explizites Bekanntmachen der 3-Färbung würde nämlich eine Schwachstelle darstellen, und man wäre gezwungen, jedesmal einen neuen Graphen zu verwenden. Um dies zu vermeiden, verwendet man die folgende Prozedur, mit der man mit beliebig hoher Wahrscheinlichkeit beweisen kann, daß man eine 3-Färbung kennt, ohne aber diese bekannt zu geben. Man übergibt dem System eine 3-Färbung der Ecken. Dem System ist es aber nur erlaubt, sich die Farben von zwei benachbarten Ecken anzuschauen und zu überprüfen, ob die beiden Farben verschieden sind. Dieser Schritt wird nun mehrmals wiederholt. Damit die Färbung nicht sukzessive anhand dieser Tests von dem System herausgefunden werden kann, übergibt man in jedem Schritt eine andere 3-Färbung des Graphen. Solche erhält man,

indem man die drei Farben auf eine der sechs möglichen Arten permutiert. Dies macht es dem System fast unmöglich, aus den einzelnen Informationen auf eine 3-Färbung zu schließen.

Mit der Anzahl der erfolgreichen Testabfragen bei einer Überprüfung steigt die Wahrscheinlichkeit sehr schnell, daß der Benutzer wirklich eine 3-Färbung kennt. Hat der Benutzer eine Zuordnung von Farben zu Ecken, die keine 3-Färbung ist und bei der die Enden von k der m Kanten die gleiche Farbe haben, so ist die Wahrscheinlichkeit, daß das System diesen Fehler findet, gleich k/m . Nach l erfolgreichen Tests ist die Wahrscheinlichkeit, daß dieser Benutzer nicht entdeckt wird, gleich $(1 - k/m)^l$, und dieser Ausdruck strebt für wachsendes l schnell gegen Null. Das Erstaunliche bei diesem Verfahren ist, daß kein Wissen über die 3-Färbung bekannt gegeben wurde.

Verfahren zur Verwaltung von Geheimwissen, bei denen das System keine Verwaltungsinformation geheim halten muß, nennt man auch *zero-knowledge Protokolle*.

5.3 Backtracking-Verfahren

Das Backtracking-Verfahren ist ein Lösungsverfahren, welches sich auf viele Probleme anwenden läßt. Man versucht hierbei, sukzessive eine Lösung zu finden, indem man Teillösungen erweitert und dabei systematisch alle Möglichkeiten für Lösungen durchprobiert. Ist zu einem Zeitpunkt ein weiterer Ausbau einer vorliegenden Teillösung nicht mehr möglich, so wird der letzte Teilschritt rückgängig gemacht, um die so reduzierte Teillösung auf einem anderen Weg auszubauen. Dieses Vorgehen wiederholt man so lange, bis eine Lösung gefunden wird oder bis man erkennt, daß das Problem keine Lösung besitzt.

Mit Hilfe des Backtracking-Verfahrens kann man entscheiden, ob ein gegebener Graph eine c -Färbung besitzt; es wird also nicht direkt die chromatische Zahl bestimmt. Dabei versucht man, in jedem Schritt eine neue Ecke so zu färben, daß die Farbe mit der Färbung der vorhergehenden Ecken verträglich ist. Hat man eine Färbung für die Ecken $1, 2, \dots, i-1$ schon gefunden, so vergibt man eine Farbe für die Ecke i und überprüft, ob die Farben der schon gefärbten Nachbarn der Ecke i verschieden von dieser Farbe sind. Ist dies nicht der Fall, so wird eine neue Farbe für die Ecke i gewählt und der Test wiederholt. Sind irgendwann alle c Farben vergeben, so geht man einen Schritt zurück, ändert die Farbe der Ecke $i-1$, und fährt so fort.

Die Vergabe der Farben erfolgt systematisch nach aufsteigenden Nummern. Bei der erstmaligen Wahl der Farbe einer Ecke wählt man immer die Farbe 1; ansonsten wird die Nummer der Farbe um 1 erhöht bis zur Farbe mit der Nummer c . Das Verfahren endet entweder, wenn die letzte Ecke eine zulässige Farbe bekommen hat, mit einer c -Färbung, oder es endet mit der Feststellung, daß es keine c -Färbung gibt. Der letzte Fall tritt ein, wenn die Farben 1 bis c für die erste Ecke zu keinem Erfolg geführt haben.

Die in Abbildung 5.8 dargestellte Funktion `färbung` gibt genau `true` zurück, falls der Graph eine c -Färbung besitzt. Die Farben der Ecken werden in dem Feld f verwaltet. Falls eine Ecke noch keine Farbe zugeordnet bekommen hat, ist der entsprechende Eintrag gleich 0. Die Vergabe der Farben erfolgt mittels der rekursiven Prozedur

```

var f : array[1..max] of Integer;
function färbung(G : Graph; c : Integer) : Boolean;
var
  q : Boolean;
begin
  Initialisiere f mit 0;
  versuche(1,c,q);
  färbung := q;
end

procedure versuche(i, c : Integer; var q : Boolean);
var
  farbe : Integer;
begin
  Initialisiere farbe mit 0;
  repeat
    farbe := farbe + 1;
    q := false;
    if möglich(i,farbe) then begin
      f[i] := farbe;
      if i < n then begin
        versuche(i+1,c,q);
        if q = false then
          f[i] := 0;
      end
      else
        q := true;
    end;
    until q = true or farbe = c;
end

function möglich(i,farbe : Integer) : Boolean;
var
  j : Integer;
begin
  möglich := true;
  for jeden Nachbar j von i do
    if f[j] = farbe then
      möglich := false;
end

```

Abbildung 5.8: Backtracking-Verfahren zur Bestimmung einer c -Färbung

versuche. Hierbei wird versucht, eine Färbung der Ecken $1, \dots, i - 1$ zu erweitern, indem der Ecke i der Reihe nach die Farben $c = 1, 2, \dots$ zugeordnet werden. Die Funktion möglich überprüft, ob eine solche Zuordnung zulässig ist. Das Backtracking-Verfahren kann leicht abgeändert werden, damit es alle c -Färbungen findet oder die chromatische Zahl bestimmt (vergleichen Sie Aufgabe 22). Durch wiederholte Anwendung der Funktion `färbung` kann die chromatische Zahl eines Graphen bestimmt werden.

Das Backtracking-Verfahren durchsucht systematisch alle möglichen Farbzuordnungen, bis eine zulässige Färbung gefunden wird. Diesen Prozeß kann man mit Hilfe eines Suchbaumes darstellen. Für jede Ecke in dem Graphen gibt es genau ein Niveau im Suchbaum, und die Markierung einer Ecke gibt die zugeordnete Farbe an. Jeder Pfad von der Wurzel zu einem Blatt gibt eine Farbzuordnung der entsprechenden Ecken an. Die Zeichen + und – an den Blättern deuten an, ob die Farbzuordnung zulässig ist. Ein + auf dem untersten Niveau bedeutet, daß es eine c -Färbung gibt. Abbildung 5.9 zeigt einen Graphen und die entsprechenden Suchbäume für die Fälle $c = 3$ und $c = 2$. Der linke Suchbaum zeigt, daß der Graph eine 3-Färbung besitzt. Die Kanten des Weges, welcher zu einer 3-Färbung führt, sind fett gezeichnet. Dabei bekommen die Ecken 1 und 3 die Farbe 1, die Ecken 2 und 5 die Farbe 2 und die Ecke 4 die Farbe 3. Der rechte Suchbaum zeigt, daß der Graph keine 2-Färbung besitzt. Somit ist die chromatische Zahl gleich 3.

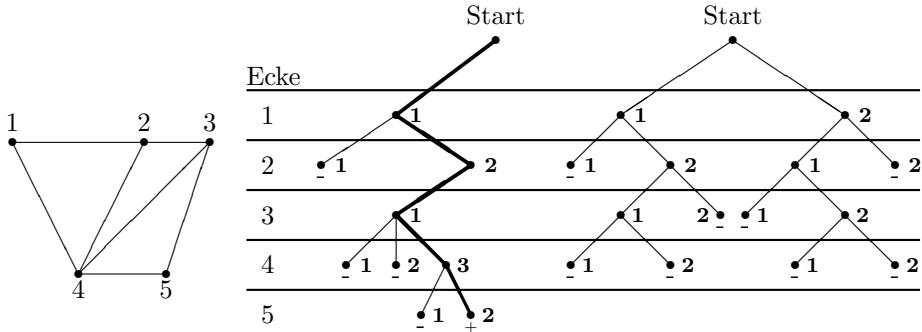


Abbildung 5.9: Suchbäume des Backtrackingverfahrens

Die Anzahl der Ecken im Suchbaum ist ein gutes Maß für den Aufwand des Backtracking-Verfahrens. In diesem Suchbaum hat jede Ecke maximal c Nachfolger. Es lassen sich leicht Graphen konstruieren, welche zu einem Suchbaum mit einer exponentiellen Anzahl von Ecken führen (vergleichen Sie Aufgabe 19). Somit ist das Backtracking-Verfahren zur Bestimmung der chromatischen Zahl für Graphen mit einer großen Eckenanzahl nur bedingt geeignet.

In der Praxis läßt sich die Laufzeit des Backtracking-Verfahrens aber noch erheblich verbessern. Betrachten Sie dazu noch einmal den in Abbildung 5.9 rechts dargestellten Suchbaum. Der linke Teilbaum stellt den Versuch dar, eine 2-Färbung zu finden, bei der Ecke 1 die Farbe 1 hat. Nachdem dieser Versuch gescheitert ist, kann direkt gefolgert werden, daß der Graph keine 2-Färbung besitzt. Es kann keine 2-Färbung geben, bei

der Ecke 1 die Farbe 2 hat: durch Vertauschung der Farben 1 und 2 würde man aus einer solchen Färbung eine Färbung erhalten, bei der Ecke 1 doch die Farbe 1 hat. Somit braucht der zweite Teilbaum nicht erzeugt zu werden. Diese Beobachtung kann noch verallgemeinert werden.

Es sei F die höchste Farbnummer, welche für die Färbung der Ecken $1, \dots, i$ vergeben wurde. Für die Färbung der Ecke $i + 1$ brauchen nur die Farben mit den Nummern $1, 2, \dots, F, F + 1$ getestet zu werden. Führt keine dieser Farben zu einer gültigen c -Färbung, so gelingt dies auch nicht, wenn Ecke $i + 1$ mit einer der Farben $F + 2, \dots, c$ gefärbt wird (gleiches Argument wie oben). Dieser Umstand kann leicht in der Prozedur **versuche** berücksichtigt werden. Man übergibt dieser Prozedur die höchste Nummer, welche diese als Farbe vergeben soll (Parameter: `letzteFarbe`). Innerhalb von **versuche** werden nun nicht mehr die Farben 1 bis c durchprobiert, sondern nur noch 1 bis `letzteFarbe`. Dazu wird die Abbruchbedingung der **repeat**-Schleife erweitert. Nun ist nur noch der rekursive Aufruf von **versuche** zu beachten: je nachdem, ob die Farbe mit der Nummer `letzteFarbe` vergeben worden ist oder nicht, wird `letzteFarbe` + 1 oder `letzteFarbe` übergeben. Abbildung 5.10 zeigt diese Variante der Prozedur **versuche**. Der Aufruf der Prozedur **versuche** in der Funktion **färbung** lautet nun `versuche(1, c, 1, q)`, d.h. für die Ecke 1 wird nur die Farbe 1 verwendet. Eine weitere Verbesserung des Backtracking-Verfahrens findet man in Aufgabe 20.

```

procedure versuche(i, c, letzteFarbe : Integer; var q : Boolean);
var
    farbe : Integer;
begin
    Initialisiere farbe mit 0;
    repeat
        farbe := farbe + 1;
        q := false;
        if möglich(i,farbe) then begin
            f[i] := farbe;
            if i < n then begin
                if farbe = letzteFarbe then
                    versuche(i+1,c,letzteFarbe + 1,q)
                else
                    versuche(i+1,c,letzteFarbe,q);
                if q = false then
                    f[i] := 0;
            end
            else
                q := true;
        end;
        until q = true or farbe = c or farbe = letzteFarbe;
end

```

Abbildung 5.10: Verbesserte Realisierung der Prozedur **versuche**

5.4 Das Vier-Farben-Problem

Eines der ältesten graphentheoretischen Probleme ist das bekannte Vier-Farben-Problem. Es handelt sich dabei um die Vermutung, daß sich die Länder auf einer beliebigen Landkarte mit höchstens vier Farben so färben lassen, daß Länder, die eine gemeinsame Grenze haben, verschiedene Farben bekommen. In die Terminologie der Graphentheorie übersetzt lautet die Frage: Ist die chromatische Zahl jedes planaren Graphen kleiner gleich 4? In dem zugehörigen Graphen bilden die Länder die Ecken, und jede Kante entspricht einer gemeinsamen Grenze. Die Vermutung tauchte zum erstenmal im Jahre 1850 auf, als ein Student von De Morgan diese Frage aufwarf. Das Problem blieb über 125 Jahre ungelöst, obwohl es immer wieder Leute gab, welche behaupteten, einen Beweis gefunden zu haben (unter anderem A.B. Kempe und P.G. Tait). Diese Beweise konnten aber einer genaueren Überprüfung nie standhalten. Das Problem wurde erst 1976 von K. Appel und W. Haken gelöst. Sie verwendeten dabei Techniken, die auf den Mathematiker H. Heesch zurückgehen. Der Beweis brachte keine größere Einsicht in die Problemstellung, und wesentliche Teile wurden mit der Unterstützung von Computern durchgeführt.

In dem Beweis zeigte man, daß es genügt, 4-Färbungen für eine Menge von etwa 1950 Arten von Graphen zu finden. Jeder planare Graph kann auf einen dieser Fälle zurückgeführt werden. Ein Computerprogramm zeigte die Existenz der 4-Färbungen dieser Graphen. Der Beweis von Appel und Haken leidet in der Ansicht vieler Wissenschaftler unter zwei Makel. Zum einen kann der durch das Computerprogramm erbrachte Beweisteil nicht per Hand nachvollzogen werden und zum anderen ist der restliche Beweisteil sehr kompliziert und bisher nur von sehr wenigen Leuten verifiziert worden.

In letzter Zeit wurden durch N. Robertson, D. Sanders, P. Seymour und R. Thomas einige Vereinfachungen an dem Beweis vorgenommen. Dadurch reduzierte sich die Anzahl der zu überprüfenden Fälle auf 633. Ferner wurde auch die Struktur des Beweises vereinfacht, ohne jedoch einen grundsätzlich neuen Weg zu beschreiben. Das verwendete Computerprogramm ist heute frei verfügbar, so daß jeder das Verfahren verifizieren kann und die Berechnungen sich nachvollziehen lassen. Die Laufzeit des Programmes liegt unter vier Stunden auf einer SUN Sparc 20 Workstation. Bis heute gibt es aber keinen Beweis, der ohne die Hilfe eines Computers auskommt.

Einfacher hingegen ist es, zu beweisen, daß jeder planare Graph eine 5-Färbung hat. Der Beweis beruht auf der im folgenden bewiesenen Eigenschaft, daß jeder planare Graph eine Ecke besitzt, deren Grad kleiner oder gleich 5 ist. Dies ist eine Folgerung aus der sogenannten *Eulerschen Polyederformel*. Sie gibt einen Zusammenhang zwischen der Anzahl der Flächen, Kanten und Ecken eines Polyeders an. Die Kanten und Ecken eines Polyeders bilden einen planaren Graphen. Dies sieht man leicht, wenn man eine entsprechende Projektion der Kanten des Polyeders in die Ebene vornimmt.

Die Einbettung eines planaren Graphen in eine Ebene bewirkt eine Unterteilung der Ebene in Gebiete. Hierbei ist auch das Außengebiet zu berücksichtigen. Es gilt dabei folgender Zusammenhang zwischen dem Graphen und der Anzahl der entstehenden Gebiete:

Eulersche Polyederformel. Es sei G ein zusammenhängender planarer Graph mit n

Ecken und m Kanten. Die Einbettung von G in die Ebene unterteile die Ebene in g Gebiete. Dann gilt: $n - m + g = 2$.

Beweis. Der Beweis erfolgt durch vollständige Induktion nach der Anzahl m der Kanten. Ist $m = 0$, so besteht G nur aus einer Ecke, und g ist gleich 1. Also stimmt die Polyederformel für $m = 0$.

Die Aussage sei für $m > 0$ bewiesen. Es sei G ein zusammenhängender planarer Graph mit $m + 1$ Kanten. Enthält G keinen geschlossenen Weg, so ist G ein Baum. Dann ist $g = 1$ und $m = n - 1$. Also stimmt die Polyederformel auch in diesem Fall. Bleibt noch der Fall, daß G einen geschlossenen Weg enthält. Entfernt man aus diesem Weg eine Kante, so verschmelzen zwei Gebiete zu einem. Das heißt, dieser Graph hat n Ecken und m Kanten, und er unterteilt die Ebene in $g - 1$ Gebiete. Da für diesen Graphen nach Induktionsvoraussetzung die Aussage wahr ist, ergibt sich $n - m + g - 1 = 2$. Also gilt $n - (m + 1) + g = 2$, und der Beweis ist vollständig. ■

Aus der Eulerschen Polyederformel ergibt sich der folgende nützliche Satz:

Satz. Für einen planaren Graphen G mit n Ecken und $m > 1$ Kanten gilt $m \leq 3n - 6$. Ist G planar und bipartit, so gilt sogar $m \leq 2n - 4$.

Beweis. Es genügt, den Satz für zusammenhängende planare Graphen zu beweisen. Der allgemeine Fall ergibt sich dann durch Summation der einzelnen Gleichungen. Es sei g die Anzahl der Gebiete einer Einbettung des Graphen in die Ebene. Jede Kante trägt zur Begrenzung von genau zwei Gebieten bei. Ferner besteht die Begrenzung von jedem Gebiet aus mindestens drei Kanten. Somit gilt:

$$3g \leq 2m.$$

Nach Eulerscher Polyederformel gilt $g = 2 + m - n$. Somit gilt:

$$6 + 3m - 3n \leq 2m.$$

Daraus ergibt sich sofort das gewünschte Resultat. Ist G bipartit, so kann es kein Gebiet geben, welches von genau drei Kanten begrenzt ist (C_3 ist nicht bipartit). Somit gilt in diesem Fall sogar

$$4g \leq 2m.$$

Hieraus folgt mit Hilfe der Eulerschen Polyederformel die zweite Behauptung. ■

Die Graphen K_3 und $K_{2,2}$ zeigen, daß die angegebenen oberen Grenzen für die Kantenanzahl nicht verschärft werden können. Aus dem letzten Satz folgt sofort, daß der vollständige Graph K_5 nicht planar ist, denn für ihn gilt $n = 5$ und $m = 10$. Ferner ist auch der vollständig bipartite Graph $K_{3,3}$ nicht planar, denn für ihn gilt $n = 6$ und $m = 9$. Das folgende Korollar ist eine einfache Folgerung aus diesem Satz.

Korollar. In einem planaren Graphen gibt es eine Ecke, deren Eckengrad kleiner gleich 5 ist.

Beweis. Gibt es nur eine Kante, so ist die Aussage klar. Angenommen, alle Ecken hätten mindestens den Grad 6. Daraus folgt, daß $6n \leq 2m$ ist. Dies steht aber im Widerspruch zu dem obigen Satz. ■

Im Jahre 1879 veröffentlichte A. Kempe einen „Beweis“ für das Vier-Farben-Problem. Erst elf Jahre später entdeckte P. Heawood einen Fehler in dieser Arbeit. Er bemerkte dabei, daß der Fehler für den Fall von fünf Farben nicht auftritt. Dieser Beweis ist im folgenden dargestellt. Er stützt sich auf die folgende einfache Beobachtung, die von A. Kempe stammt. Es sei G ein ungerichteter Graph mit Eckenmenge E und f eine Färbung von G . Für zwei Farben i, j sei G_{ij} der von

$$\{e \in E \mid f(e) = i \text{ oder } f(e) = j\}$$

induzierte Untergraph von G und Z eine Zusammenhangskomponente von G_{ij} . Ändert man die Färbung f für alle Ecken e von Z derart, daß $f(e) = i$ gilt, falls vorher $f(e) = j$ war und umgekehrt, so ist f weiterhin eine Färbung von G , welche genauso viele Farben verwendet.

Satz. Die chromatische Zahl jedes planaren Graphen ist kleiner oder gleich fünf.

Beweis. Der Beweis erfolgt durch Induktion nach n . Ist $n = 1$, so ist der Satz wahr. Sei G ein planarer Graph mit $n + 1$ Ecken, und e sei eine Ecke, deren Grad kleiner oder gleich 5 ist. Sei G' der Graph, der aus G entsteht, wenn man die Ecke e samt den mit ihr inzidenten Kanten entfernt. Nach Induktionsvoraussetzung besitzt G' eine 5-Färbung. Verwenden in dieser Färbung die Nachbarn von e nicht alle 5 Farben, so erhält man daraus sofort eine 5-Färbung für G . Im anderen Fall läßt sich dies durch eine Änderung der Farben erreichen. Die Ecke e hat dann den Grad 5. Die Nachbarn von e seien so numeriert, daß es eine kreuzungsfreie Darstellung gibt, in der die Ecken in der Reihenfolge e_1, e_2, \dots, e_5 um e angeordnet sind. Man betrachte die Graphen G_{ij} mit $1 \leq i, j \leq 5$, die von den Ecken mit den Farben i und j induziert sind. Hierbei tragen die Ecken e_i jeweils die Farbe i .

Es werden nun zwei Fälle unterschieden, je nachdem, ob es in G_{13} einen Weg von e_1 nach e_3 gibt oder nicht. Im zweiten Fall liegt e_1 in einer Zusammenhangskomponente Z von G_{13} , welche e_3 nicht enthält. Vertauscht man nun die Farben der Ecken in Z , so erhält man wieder eine 5-Färbung von G' . In dieser haben e_1 und e_3 die gleiche Farbe, und es ergibt sich sofort eine 5-Färbung für G . Bleibt noch der Fall, daß es einen Weg von e_1 nach e_3 in G_{13} gibt. Zusammen mit e ergibt sich daraus ein geschlossener Weg W in G . Bedingt durch die Lage der e_i folgt, daß e_2 im Inneren dieses Weges und e_4 im Außengebiet liegt. Somit muß jeder Weg in G' von e_2 nach e_4 eine Ecke, die auf W liegt, benutzen. Da diese Ecken aber die Farben 1 und 3 haben, liegen e_2 und e_4 in verschiedenen Zusammenhangskomponenten von G_{24} . Wie oben ergibt sich auch hier eine 5-Färbung von G . ■

Der Beweis des letzten Satzes führt direkt zu einem Algorithmus zur Erzeugung einer 5-

Färbung für einen planaren Graphen. Abbildung 5.11 zeigt eine entsprechende Prozedur 5-färbung. Der Induktionsbeweis wird dabei in eine rekursive Prozedur umgesetzt.

```

var f : array[1..max] of Integer;
procedure 5-färbung(G : P-Graph);
var
    i,j,u,v : Integer;
    F,S : set of Integer;
begin
    if Eckengrad aller Ecken von G maximal 4 then
        greedy-färbung(G)
    else begin
        Sei v eine Ecke von G mit g(v) ≤ 5 und S die Menge der
        Nachbarn von v in G;
        5-färbung(G\{v\});
        F := {f[u] | u ∈ S};
        if |F| = 5 then begin
            Unter den Teilmengen {i,j} von S bestimme diejenige,
            für die die Ecke i in dem von
            {u | f[u] = f[i] oder f[u] = f[j]}
            induzierten Untergraph H nicht von j aus erreichbar ist;
            Sei C die Zusammenhangskomponente von H, welche i
            enthält;
            Vertausche die Farben aller Ecken aus C;
            F := {f[u] | u ∈ S};
        end;
        f[v] := min {i ≥ 1 | i ∉ F};
    end
end

```

Abbildung 5.11: Die Prozedur 5-färbung

Für die Bestimmung der Laufzeit beachte man, daß für zusammenhängende planare Graphen $O(m) = O(n)$ gilt. Der Aufruf der Prozedur **greedy-färbung** hat somit einen Aufwand von $O(n)$. Man beachte, daß maximal zehn Teilmengen der Form $\{i, j\}$ von S zu untersuchen sind. Die Bestimmung der Zusammenhangskomponenten erfolgt mit der Tiefensuche. Somit hat dieser Teil einen Aufwand von $O(n')$, wobei n' die Anzahl der Ecken des Graphen ist, für den die Zusammenhangskomponenten bestimmt werden. Im ungünstigsten Fall ist n' gleich $n-1, n-2, n-3, \dots$. Daraus folgt, daß der Gesamtaufwand $O(n^2)$ ist.

5.5 Transitiv orientierbare Graphen

In diesem Abschnitt wird eine weitere Klasse von Graphen vorgestellt, für die sich minimale Färbungen effizient bestimmen lassen. Eine *Orientierung* eines ungerichteten

Graphen G ist ein gerichteter Graph, welcher aus G hervorgeht, in dem jede Kante von G gerichtet wird. Eine Orientierung heißt *kreisfrei*, wenn der gerichtete Graph kreisfrei ist. Jeder ungerichtete Graph besitzt mindestens eine kreisfreie Orientierung. Man erhält z.B. eine kreisfreie Orientierung, indem man die Ecken von $1, \dots, n$ numeriert und die Kanten in Richtung höherer Eckenzahlen orientiert. Abbildung 5.12 zeigt einen ungerichteten Graphen und eine so entstandene Orientierung des Graphen.

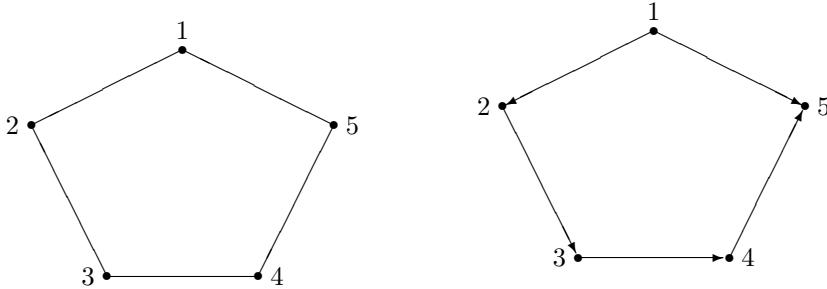


Abbildung 5.12: Ein ungerichteter Graph und eine kreisfreie Orientierung

Im folgenden wird der Begriff der Höhe einer Ecke benötigt. Es sei G ein kreisfreier gerichteter Graph. Für jede Ecke e von G wird die *Höhe* $h(e)$ wie folgt definiert:

$$h(e) = \begin{cases} 0, & \text{falls } g^+(e) = 0; \\ \text{Länge des längsten Weges in } G \text{ mit Startecke } e, & \text{sonst.} \end{cases}$$

Hier ist die Länge eines Weges gleich der Anzahl der Kanten. Die Höhen der Ecken eines kreisfreien Graphen lassen sich effizient bestimmen.

Lemma. Es sei G ein kreisfreier gerichteter Graph. Dann gilt für jede Ecke e von G :

$$h(e) = \begin{cases} 0, & \text{falls } g^+(e) = 0; \\ \max \{h(f) \mid (e, f) \text{ Kante in } G\} + 1, & \text{sonst.} \end{cases}$$

Beweis. Es sei e eine Ecke von G mit $g^+(e) > 0$ und W ein Weg mit Startecke e , welcher aus $h(e)$ Kanten besteht. W verweise die Kante $k = (e, f)$, und W_1 sei der Teil des Weges W , welcher bei f startet. Dann besteht W_1 aus $h(e) - 1$ Kanten, und somit ist $h(f) \geq h(e) - 1$. Wäre $h(f) > h(e) - 1$, so gäbe es einen Weg \bar{W} mit Startecke f , welcher aus mindestens $h(e)$ Kanten besteht. Da G kreisfrei ist, kann \bar{W} mit Hilfe von k zu einem Weg mit Startecke e verlängert werden. Dieser neue Weg besteht aus mindestens $h(e) + 1$ Kanten. Dieser Widerspruch zeigt, daß $h(e) = h(f) + 1$ ist. ■

Mit Hilfe dieses Lemmas läßt sich leicht ein effizienter Algorithmus zur Bestimmung der Höhen der Ecken eines kreisfreien Graphen angeben. Man bestimme zunächst eine topologische Sortierung des Graphen, betrachte die Ecken dann in der Reihenfolge absteigender topologischer Sortierungsnummern und bestimme die Höhen mittels der

im Lemma angegebenen Gleichung. Dazu beachte man, daß zu dem Zeitpunkt der Berechnung von $h(e)$ die Werte $h(f)$ für alle Nachfolger f von e schon berechnet wurden. Somit gilt:

Lemma. Die Höhen der Ecken eines kreisfreien gerichteten Graphen lassen sich mit Aufwand $O(n + m)$ bestimmen.

Mit Hilfe einer kreisfreien Orientierung eines ungerichteten Graphen kann eine Färbung des Graphen konstruiert werden.

Lemma. Die Höhen der Ecken einer kreisfreien Orientierung eines ungerichteten Graphen definieren eine Färbung.

Beweis. Es sei $k = (e, f)$ eine Kante von G . Je nach der Orientierung von k gilt $h(e) \geq h(f) + 1$ oder $h(f) \geq h(e) + 1$. Somit gilt $h(e) \neq h(f)$, und die Höhen definieren eine Färbung. ■

Der Graph in Abbildung 5.12 zeigt, daß die so entstehenden Färbungen nicht notwendigerweise minimal sind. Die Ecke 1 hat die Höhe 4, d.h. es werden fünf Farben verwendet, die chromatische Zahl ist aber gleich 3. Jeder ungerichtete Graph G besitzt eine kreisfreie Orientierung, so daß die entstehende Färbung minimal ist. Dazu numeriert man die Ecken mit $1, \dots, \chi(G)$ entsprechend einer minimalen Färbung und orientiert die Kanten wieder in Richtung höherer Nummern. Minimale Färbungen erhält man durch eine zusätzliche Forderung an die Orientierung.

Ein gerichteter Graph G heißt *transitiv*, falls er mit seinem transitiven Abschluß übereinstimmt, d.h. sind (e, f) und (f, g) Kanten von G , dann ist auch (e, g) eine Kante von G . Ein ungerichteter Graph G heißt *transitiv orientierbar*, falls er eine kreisfreie transitive Orientierung besitzt. Abbildung 5.13 zeigt einen ungerichteten, transitiv orientierbaren Graphen G und einen entsprechend gerichteten Graphen G_T .

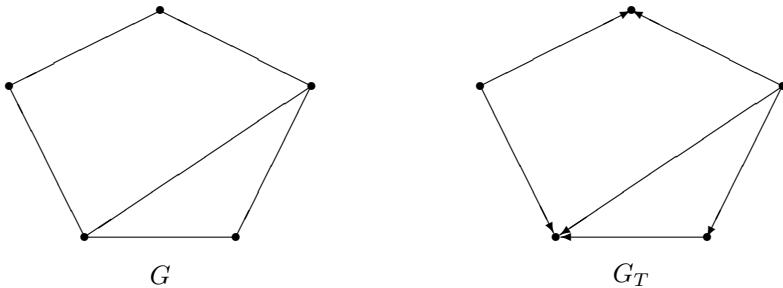


Abbildung 5.13: Ein transitiv orientierbarer Graph

Es lassen sich leicht Beispiele für nicht transitiv orientierbare Graphen angeben. Man zeigt sofort, daß die Graphen C_{2n+1} mit $n > 1$ nicht transitiv orientierbar sind. Der in Abbildung 5.13 dargestellte Graph ist gerade C_5 mit einer zusätzlichen Kante, welche zwei Ecken mit Abstand 2 verbindet. Eine solche Kante nennt man *Dreieckssehne*.

Allgemein haben P. Gilmore und A. Hoffmann gezeigt, daß ein ungerichteter Graph G genau dann transitiv orientierbar ist, wenn in G alle geschlossenen Wege mit ungerader Kantenzahl eine Dreieckssehne besitzen. Dieses Resultat wird aber im folgenden nicht benötigt.

Transitiv orientierbare Graphen treten in vielen Situationen auf. Es sei M eine Menge, auf der eine partielle Ordnung „ \leq “ definiert ist. Es sei G der Graph mit Eckenmenge M und Kantenmenge

$$K = \{(x, y) \mid x \neq y \in M \text{ mit } x \leq y \text{ oder } y \leq x\}.$$

Dann ist G transitiv orientierbar, indem jede Kante in Richtung des größeren Elements ausgerichtet wird. Ist die partielle Ordnung eine totale Ordnung, so ist G der vollständige Graph.

Satz. Es sei G ein ungerichteter, transitiv orientierbarer Graph mit Eckenmenge E , der eine kreisfreie transitive Orientierung G_T besitzt. Dann gilt:

- a) Die Höhen der Ecken von G_T definieren eine minimale Färbung von G , und es gilt $\omega(G) = \chi(G)$.
- b) Für jede Teilmenge $A \subseteq E$ gilt $\omega(G_A) = \chi(G_A)$, wobei G_A der von A induzierte Untergraph von G ist.

Beweis. a) Es sei e_0 eine Ecke von G mit maximaler Höhe c . Es gilt $\chi(G) \leq c + 1$, und es gibt einen Weg $e_0, e_1, e_2, \dots, e_c$ in G_T . Da G_T transitiv ist, gibt es in G_T für $0 \leq i < j \leq c$ eine Kante von e_i nach e_j . Somit bilden die Ecken $\{e_0, \dots, e_c\}$ eine Clique in G . Daraus folgt

$$c + 1 \leq \omega(G) \leq \chi(G) \leq c + 1,$$

und damit $\omega(G) = \chi(G)$. Dies bedeutet, daß die durch die Höhen definierte Färbung minimal ist.

b) Jeder Graph G_A ist ebenfalls transitiv orientierbar. Somit gilt nach dem ersten Teil $\omega(G_A) = \chi(G_A)$. ■

Zunächst wird ein Beispiel betrachtet: Abbildung 5.14 zeigt einen ungerichteten Graphen G , welcher die Voraussetzungen des letzten Satzes erfüllt, und einen entsprechend gerichteten Graphen G_T . Die Ecken sind mit den von den Höhen implizierten Farbnummern markiert. Es ergibt sich $\chi(G) = 4$.

Abbildung 5.15 zeigt eine Realisierung des Algorithmus zur Bestimmung einer minimalen Färbung eines transitiv orientierbaren Graphen. Die Funktion `trans-färbung` hat als Eingabeparameter einen gerichteten, kreisfreien und transitiven Graphen, bestimmt eine minimale Färbung und gibt die chromatische Zahl zurück. Sie verwendet die rekursive Prozedur `tfprozedur`, welche eine topologische Sortierung bestimmt und dabei gleichzeitig die Höhen der Ecken bestimmt. Sowohl `trans-färbung` als auch

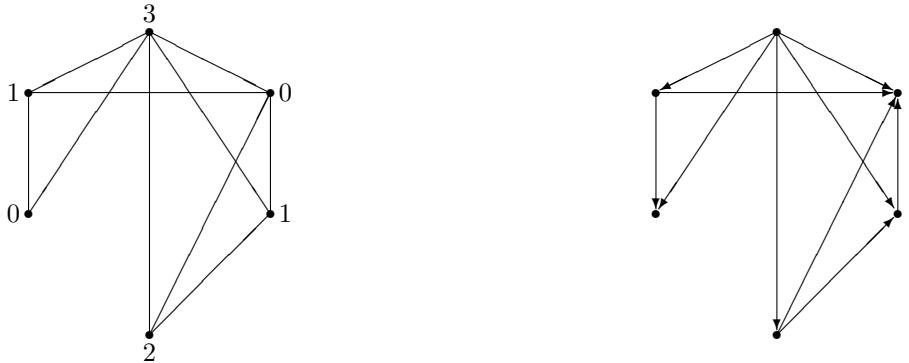


Abbildung 5.14: Ein transitiv orientierbarer Graph

tfprozedur gehen aus den Prozeduren `topsort` bzw. `tsprozedur` aus Kapitel 4 hervor. Eine minimale Färbung eines transitiv orientierbaren Graphen kann mit Aufwand $O(n + m)$ bestimmt werden.

Graphen, die die Voraussetzung des letzten Satzes erfüllen, sind Spezialfälle von sogenannten perfekten Graphen. Ein ungerichteter Graph G mit Eckenmenge E heißt *perfekt*, falls $\omega(G_A) = \chi(G_A)$ für jeden induzierten Untergraphen G_A mit $A \subseteq E$ gilt. Perfekte Graphen wurden innerhalb der Graphentheorie intensiv untersucht, und es wurden einige Charakterisierungen angegeben. Die bipartiten Graphen sind Beispiele für perfekte Graphen. Im folgenden wird noch eine weitere Klasse von perfekten Graphen betrachtet: sogenannte Permutationsgraphen. Für sie lässt sich eine Färbung mit minimaler Farbenzahl mit Aufwand $O(n \log n)$ bestimmen. Eine weitere Klasse von perfekten Graphen wird in den Aufgaben 13 und 38 behandelt.

Es sei π eine Permutation der Menge $\{1, \dots, n\}$. Der zu π gehörende *Permutationsgraph* G_π ist ein Graph mit n Ecken, und die Ecken i, j sind benachbart, falls die Reihenfolge von i und j durch π^{-1} vertauscht wird; d.h. entweder gilt $i < j$ und $\pi^{-1}(i) > \pi^{-1}(j)$ oder $i > j$ und $\pi^{-1}(i) < \pi^{-1}(j)$. Die Definition von G_π lässt sich gut durch das sogenannte *Permutationsdiagramm* veranschaulichen. Dazu werden die Zahlen $1, \dots, n$ auf zwei parallelen Geraden angeordnet: auf der oberen Gerade in der Reihenfolge $1, 2, \dots, n$ und auf der unteren in der Reihenfolge $\pi(1), \pi(2), \dots, \pi(n)$. Anschließend werden gleiche Zahlen durch Liniensegmente verbunden. Zwei Ecken sind in G_π genau dann benachbart, wenn sich die dazugehörigen Liniensegmente schneiden. In Abbildung 5.16 ist das Permutationsdiagramm und der Permutationsgraph für die Permutation $\pi = [5, 3, 1, 6, 7, 4, 9, 10, 8, 2]$ abgebildet.

In dem folgenden Satz wird eine wichtige Eigenschaft von Permutationsgraphen bewiesen.

Satz. Ein Permutationsgraph G_π besitzt eine kreisfreie transitive Orientierung.

Beweis. Jede Kante von G_π wird in Richtung der höheren Zahl gerichtet. Dadurch können keine geschlossenen Wege entstehen. Es bleibt noch zu zeigen, daß die Transi-

```

var f : array[1..max] of Integer;
    χ : Integer;
procedure tfprozedur(i : Integer);
var
    j : Integer;
begin
    Besucht[i] := true;
    for jeden Nachfolger j von i do begin
        if Besucht[j] = false then
            tfprozedur(j);
        f[i] := max{f[i],f[j] + 1};
        χ := max{χ,f[i]};
    end
end

function trans-färbung(G : G-Graph) : Integer;
var
    besucht : array[1..max] of Boolean;
    i : Integer;
begin
    Initialisiere besucht mit false;
    Initialisiere f und χ mit 0;
    for jede Ecke i do
        if Besucht[i] = false then
            tfprozedur(i);
        trans-färbung := χ + 1;
    end
end

```

Abbildung 5.15: Die Funktion `trans-färbung`

tivität erfüllt ist. Es seien (i, j) und (j, k) Kanten von G_π mit $i < j < k$. Es genügt, zu zeigen, daß (i, k) eine Kante von G_π ist. Da (i, j) und (j, k) Kanten in G_π sind, gilt: $\pi^{-1}(i) > \pi^{-1}(j)$ und $\pi^{-1}(j) > \pi^{-1}(k)$. Somit gilt auch $\pi^{-1}(i) > \pi^{-1}(k)$. ■

Somit kann für Permutationsgraphen eine optimale Färbung mit Aufwand $O(n + m)$ bestimmt werden. Für das obige Beispiel ergibt sich folgende Färbung:

Ecke	1	2	3	4	5	6	7	8	9	10
Farbe	2	2	1	1	0	0	0	1	0	0

Somit ist $\chi(G_\pi) = 3$. Für Permutationsgraphen läßt sich ein weiterer Algorithmus zur Bestimmung der chromatischen Zahl angeben, welcher für dichte Graphen effizienter ist. Dazu wird direkt mit der Permutation π gearbeitet.

Es sei $C = \{n_1, \dots, n_s\}$ eine Clique von G_π . Die Elemente von C seien absteigend geordnet, d.h. $n_1 > n_2 > \dots > n_s$. Da C eine Clique ist, muß gelten: $\pi^{-1}(n_1) <$

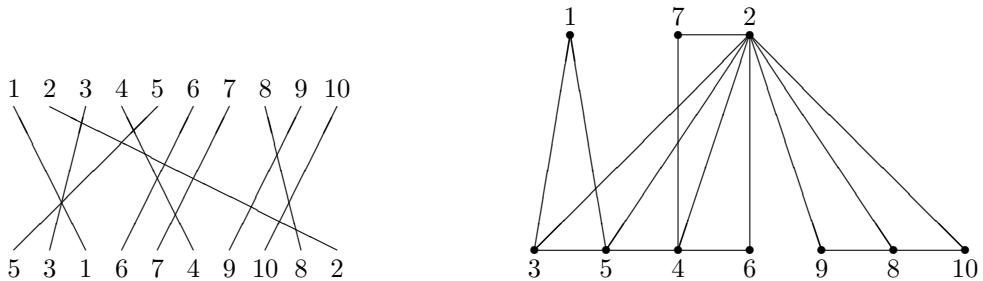


Abbildung 5.16: Das Permutationsdiagramm und der Permutationsgraph für π

$\pi^{-1}(n_2) < \dots < \pi^{-1}(n_s)$; d.h. die Zahlen n_1, n_2, \dots, n_s treten auch in dieser Reihenfolge in der Darstellung von π auf (eventuell liegen andere Zahlen dazwischen). Umgekehrt entspricht jede absteigende Teilsequenz von π genau einer Clique von G_π . Für die oben angegebene Permutation π sind z.B. $\{5, 3, 1\}$ oder $\{5, 4, 2\}$ absteigende Teilsequenzen bzw. Cliques von G_π . Auf die gleiche Weise zeigt man, daß die aufsteigenden Teilsequenzen von π den unabhängigen Mengen von G_π entsprechen.

Lemma. Für jeden Permutationsgraphen G_π gilt:

- Die absteigenden Teilsequenzen von π entsprechen genau den Cliques von G_π . Die Länge der längsten absteigenden Teilsequenz von π ist gleich $\chi(G)$.
- Die aufsteigenden Teilsequenzen von π entsprechen genau den unabhängigen Mengen von G_π . Die minimale Anzahl einer Partition von π in aufsteigenden Teilsequenzen ist gleich $\chi(G)$.

Beweis. a) Es sei C die längste absteigende Teilsequenz von π . Dann ist C eine Clique von G_π , und es gilt $|C| = \omega(G_\pi)$. Aus dem letzten Satz folgt, daß G_π transitiv orientierbar ist, und somit besteht C aus $\chi(G_\pi)$ Ecken.

b) Die Partition der Eckenmenge in unabhängige Teilmengen entspricht genau einer Färbung des Graphen. Damit ist das Lemma bewiesen. ■

Mit Hilfe dieses Ergebnisses kann nun ein Algorithmus zur Bestimmung der chromatischen Zahl und einer minimalen Färbung angegeben werden. Dazu zerlegt man die Permutation in eine minimale Anzahl von aufsteigenden Teilfolgen. Die Ecken jeder Teilfolge bekommen die gleiche Farbe. Die einzelnen Teilfolgen werden dabei parallel aufgebaut. Die Permutation wird von links nach rechts abgearbeitet, und die $\pi(j)$ werden auf die Teilfolgen aufgeteilt. Die Bearbeitung von $\pi(j)$ sieht folgendermaßen aus: Unter den schon bestehenden Teilfolgen wird die erste ausgewählt, deren größtes Element noch kleiner ist als $\pi(j)$. An diese Teilfolge wird $\pi(j)$ angehängt. Gibt es keine solche Teilfolge, so bildet $\pi(j)$ eine neue Teilfolge. Abbildung 5.17 zeigt den Graphen G_π

für die Permutation $\pi = [5, 4, 1, 3, 2]$ und eine entsprechende Zerlegung in aufsteigende Teilfolgen. Hieraus folgt $\chi(G_\pi) = 4$.

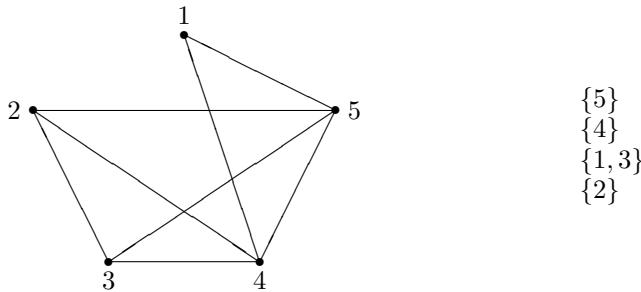


Abbildung 5.17: Ein Permutationsgraph G_π und eine minimale Zerlegung von π in aufsteigenden Teilfolgen

Es bleibt noch zu beweisen, daß die so erstellte Zerlegung $\{U_1, \dots, U_s\}$ von π minimal ist. Nach dem letzten Lemma genügt es, eine absteigende Teilfolge C von π anzugeben, welche aus s Elementen besteht. Sei $\pi(i_s)$ das erste Element von U_s . Ferner sei $\pi[i_{s-1}]$ das Element aus U_{s-1} , welches in dem Moment, als $\pi[i_s]$ in U_s eingefügt wurde, an letzter Stelle von U_{s-1} stand. Wählen Sie auf diese Art rückwärtsschreitend Elemente $\pi[i_{s-j}]$ aus U_{s-j} für $j = 1, \dots, s-1$. Dann gilt für $j = 0, \dots, s-1$

$$\pi[i_{s-j}] < \pi[i_{s-j+1}].$$

Ferner wurden die Elemente auch in dieser Reihenfolge eingefügt. Somit ist

$$\{\pi[i_s], \pi[i_{s-1}], \dots, \pi[i_1]\}$$

eine absteigende Folge der Länge s aus π und $s = \chi(G_\pi)$. Abbildung 5.18 zeigt eine Realisierung dieses Algorithmus. Die einzelnen Teilfolgen werden dabei nicht vollständig abgespeichert, sondern jeweils nur der letzte Eintrag. Wird ein $\pi[j]$ in die i -te Teilequenz eingefügt, so wird Ecke j mit Farbe i gefärbt. Die Funktion `permutations-färbung` liefert die chromatische Zahl zurück.

Die Komplexitätsanalyse von `permutations-färbung` ist einfach. Der aufwendigste Teil ist die Bestimmung von i als Minimum aller l mit `ende[1] < pi[j]`. Eine naive Suche ergibt den Aufwand $O(\chi(G_\pi))$ und insgesamt $O(n \chi(G_\pi))$. Dies kann noch verbessert werden. Dazu beachte man, daß das Feld zu jedem Zeitpunkt absteigend sortiert ist. Dies beweist man leicht durch Induktion nach j . Für $j = 0$ ist dies trivialerweise erfüllt. Am Anfang des j -ten Durchlaufes der FOR-Schleife gilt nach Induktionsannahme

$$\text{ende}[1] > \text{ende}[2] > \dots > \text{ende}[\chi] > 0.$$

Nach der Bestimmung von i gilt:

```

var f : array[1..max] of Integer;
function permutations-färbung(Gπ : Perm-Graph) : Integer;
var
    ende : array[1..max] of Integer;
    χ, i, j, l : Integer;
begin
    Initialisiere Ende, χ und f mit 0;
    for jede Ecke j do begin
        i := min {l | Ende[l] < π[j]};
        f[j] := i;
        Ende[i] := π[j];
        χ := max{χ, i};
    end;
    permutations-färbung := χ;
end

```

Abbildung 5.18: Die Funktion permutations-färbung

$$\text{ende}[1] > \dots > \text{ende}[i-1] > \pi[j] > \text{ende}[i] > \dots > \text{ende}[χ] > 0.$$

Hieraus folgt die Behauptung für j , denn am Ende des j -ten Durchlaufes gilt: $\text{ende}[i] = \pi[j]$. Die Bestimmung von i kann also durch eine binäre Suche erfolgen. Dadurch erhält man einen worst-case-Aufwand von $O(n \log n)$. Mit Hilfe von speziellen Datenstrukturen kann der Aufwand noch auf $O(n \log \log n)$ reduziert werden.

Satz. Die Bestimmung einer minimalen Färbung für einen Permutationsgraph G_π kann mit Aufwand $O(n \log n)$ durchgeführt werden.

5.6 Literatur

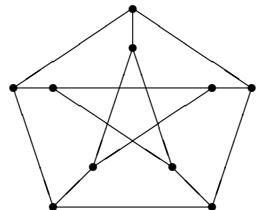
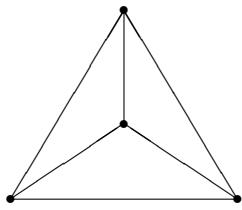
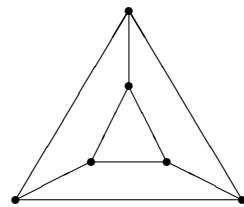
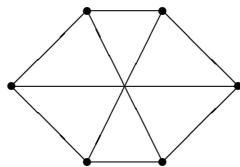
Die Färbbarkeit von Graphen wurde schon im 19. Jahrhundert untersucht, und die Bemühungen, das Vier-Farben-Problem zu lösen, gaben der Graphentheorie sehr viele Impulse. Einen guten Überblick über die verschiedenen Methoden, die zum Beweis der Vier-Farben-Vermutung entwickelt wurden, findet man in [112]. Dort ist auch die letztlich erfolgreiche Methode von Appel und Haken dargestellt. Die Orginalarbeiten von Appel und Haken sind [7] und [8] und die wichtige Arbeit von Heesch ist in [64] zusammengefaßt. Einen Überblick über die neueren Arbeiten von Robertson, Sanders, Seymour und Thomas gibt [109]. Dort findet man auch Hinweise zur Beschaffung der Computerprogramme. Eine kurze Zusammenfassung der Arbeiten zum Vier-Farben-Problem bis 1998 enthält der Aufsatz von Robin Thomas [118].

Der angegebene Beweis des Satzes von Brooks [17] stammt von Lovasz und ist in [112] enthalten. Ein Algorithmus zur exakten Bestimmung der chromatischen Zahl eines Graphen stammt von M. Kubale und B. Jackowski [85]. Das Buch von Jensen und Toft

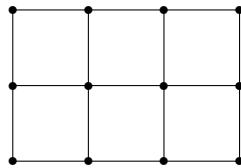
befaßt sich ausschließlich mit Färbungsproblemen für Graphen und enthält eine umfangreiche Sammlung offener Probleme [71]. Der Färbung von Graphen als Technik zur Registervergabe folgt Chaitin [19]. Die Anwendung von Färbungen für Public-Key-Kryptosysteme ist detailliert in [52] beschrieben. Ein Algorithmus mit linearer Laufzeit zur Erzeugung einer 5-Färbung für einen planaren Graphen ist in [22] beschrieben. Eine Charakterisierung von transitiv orientierbaren Graphen findet man in [47]. Einen Überblick über Algorithmen für perfekte Graphen bietet [114]. Die Definition von Permutationsgraphen stammt von S. Even, A. Pnueli und A. Lempel [36, 105]. Für die Bestimmung von minimalen Färbungen für Permutationsgraphen mit Aufwand $O(n \log \log n)$ vergleiche man [108]. Die Konstruktion aus Aufgabe 27 stammt von Mycielski [95]. Hinweise auf Anwendungen von Färbungen und Sammlungen von Graphen, für die bis heute noch keine minimalen Färbungen bekannt sind, findet man in [30]. Dort sind auch entsprechende Graphgeneratoren beschrieben.

5.7 Aufgaben

1. Im folgenden sind vier reguläre Graphen abgebildet. Der maximale Eckengrad ist jeweils 3. Bestimmen Sie die chromatische Zahl.



2. Für jedes Paar z, s von positiven natürlichen Zahlen wird der Gittergraph $L_{z,s}$ wie folgt definiert. Die $n = zs$ Ecken werden in z Zeilen und s Spalten angeordnet und untereinander bzw. nebeneinander liegende Ecken werden durch Kanten verbunden (es gibt also $2sz - (s + z)$ Kanten). Der Gittergraph $L_{3,4}$ sieht wie folgt aus.



Bestimmen Sie $\chi(L_{z,s})$ und $\omega(L_{z,s})$.

3. Es sei $d > 1$ eine natürliche Zahl und $E = \{0,1\}^d$ die Menge der 0-1-Vektoren der Länge d . Der d -dimensionale Hyperwürfel ist ein Graph H_d mit Eckenmenge E , wobei zwei Vektoren aus E genau dann durch eine Kante verbunden sind, wenn sie sich in genau einer Position unterscheiden. Bestimmen Sie $\chi(H_d)$ und $\omega(H_d)$.
4. Es sei m die Länge des längsten Weges in einem ungerichteten Graphen G , dann ist $\chi(G) \leq m + 1$ (Hinweis: Tiefensuche).
5. Beweisen Sie, daß jeder Baum mit mindestens zwei Ecken die chromatische Zahl 2 hat.
6. Sind B_1, \dots, B_s die Blöcke eines ungerichteten Graphen, so gilt

$$\chi(G) = \max \{\chi(B_i) \mid i = 1, \dots, s\}.$$

Beweisen Sie diese Aussage. Wie kann man mit dieser Eigenschaft Algorithmen zur Bestimmung der chromatischen Zahl verbessern?

- * 7. Es sei G ein ungerichteter Graph. Beweisen Sie folgende Aussage.

$$\chi(G) \leq 1 + \max \{\delta(U) \mid U \text{ ist induzierter Untergraph von } G\},$$

wobei $\delta(U)$ den kleinsten Eckengrad von U bezeichnet.

8. Beweisen Sie daß jeder Graph G mindestens $\chi(G)$ Ecken enthält, deren Eckengrad mindestens $\chi(G) - 1$ ist.
9. Beweisen Sie folgende Aussage: Enthält ein Graph G kein Dreieck (d.h. keinen Untergraph vom Typ C_3), dann ist $\alpha(G) > \sqrt{n} - 1$.
10. Ein Graph G heißt *kritisch*, wenn $\chi(G \setminus \{e\}) < \chi(G)$ für alle Ecken e von G gilt. Ist $\chi(G) = c$, dann heißt G in diesem Fall *c-kritisch*.
 - a) Zeigen Sie: Ist G kritisch, so gilt $\chi(G - \{e\}) = \chi(G) - 1$ für jede Ecke e von G .
 - b) Welche der Graphen C_n sind kritisch?
 - c) Geben Sie alle 2- und 3-kritischen Graphen an.
 - d) Beweisen Sie, daß ein kritischer Graph zweifach zusammenhängend ist.

11. Beweisen Sie, daß für einen ungerichteten Graphen G mit n Ecken und m Kanten folgende Ungleichung gilt:

$$\alpha(G) \leq \lfloor \sqrt{n^2 - 2m} \rfloor.$$

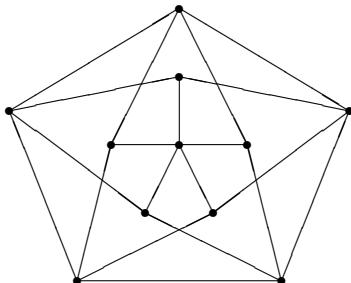
Geben Sie einen Graphen G an, für den $\alpha(G) = \sqrt{n^2 - 2m}$ gilt.

12. Beweisen Sie, daß für einen ungerichteten Graphen G mit m Kanten folgende Ungleichung gilt:

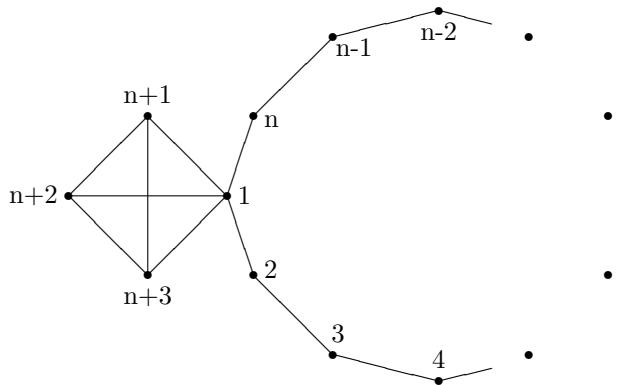
$$\chi(G) \leq \lfloor 1/2 + \sqrt{2m + 1/4} \rfloor.$$

- * 13. Gegeben sind n Zeitintervalle $T_i = [a_i, b_i]$. Die Zeitintervalle sind Start- und Endzeitpunkte von Arbeitsaufträgen. Ein Arbeiter kann nicht gleichzeitig an verschiedenen Arbeitsaufträgen arbeiten. Das Problem besteht darin, die minimale Anzahl von Arbeitern zu finden, welche man benötigt, um alle Aufträge auszuführen. Zur Lösung bildet man den *Schnittgraphen* G der Intervalle T_i . Die Intervalle T_i bilden die Ecken, und zwischen zwei Ecken gibt es eine Kante, falls die beiden Intervalle sich überschneiden.

- a) Zeigen Sie, daß $\chi(G)$ Arbeiter alle Aufträge ausführen können.
 b) Zeigen Sie, daß der folgende Algorithmus eine minimale Färbung des Schnittgraphen liefert.
- (i) Es sei $T_i = [a_i, b_i]$ das Intervall, für welches a_i minimal ist. Die entsprechende Ecke wird mit Farbe 1 gefärbt. Unter den verbleibenden Intervallen wird das Intervall $T_j = [a_j, b_j]$ gewählt, für welches $a_j > b_i$ und a_j minimal ist. Die entsprechende Ecke bekommt wieder die Farbe 1. Dieser Vorgang wird wiederholt, bis es keine Intervalle mehr gibt, die diese Bedingung erfüllen.
 - (ii) Das obige Verfahren wird nun mit den noch ungefärbten Ecken wiederholt. Dabei werden die Farben 2,3,... verwendet.
14. Bestimmen Sie $\chi(G)$ und $\omega(G)$ für den folgenden Graphen G .



15. Eine *Kantenfärbung* eines Graphen ist eine Belegung der Kanten mit Farben, so daß je zwei inzidente Kanten verschiedene Farben erhalten. Die *kantenchromatische Zahl* $\chi'(G)$ eines Graphen G ist die kleinste Zahl c , für welche der Graph eine c -Kantenfärbung besitzt. Bestimmen Sie die kantenchromatischen Zahlen der vier Graphen aus Übungsaufgabe 1.
16. Beweisen Sie, daß $\chi'(G)$ mindestens so groß ist wie der größte Eckengrad von G .
- ** 17. Beweisen Sie, daß die kantenchromatische Zahl eines bipartiten Graphen gleich dem maximalen Eckengrad ist (Hinweis: Vollständige Induktion nach m).
- ** 18. An einem Sportwettbewerb nehmen n Mannschaften teil und jede Mannschaft soll dabei gegen jede spielen. Jede Mannschaft darf an einem Tag nur einmal spielen. Formulieren Sie dies als ein Färbungsproblem. Wieviele Tage dauert der Wettbewerb bei drei, vier bzw. fünf Mannschaften? Formulieren Sie eine Vermutung für beliebiges n und beweisen Sie diese Vermutung.
19. Für jede natürliche Zahl $n \geq 3$ ist G_n der folgende ungerichtete Graph mit $n+3$ Ecken und $n+6$ Kanten.



Bestimmen Sie für jedes n die chromatische Zahl von G_n . Zeigen Sie, daß der Suchbaum, der beim Aufruf von `färbung(G_n , 3)` durchlaufen wird, $O(2^n)$ Ecken hat.

20. Nehmen Sie folgende Verbesserung an der Prozedur `färbung` vor: Von dem Moment an, in dem die Farbe F vergeben wird, werden alle Ecken übergangen, deren Grad echt kleiner F ist. Diese können am Ende leicht mit dem Greedy-Algorithmus gefärbt werden.
21. Bestimmen Sie die Anzahl der Ecken des Suchbaumes, der beim Aufruf von `färbung($K_n, n - 1$)` entsteht. Verwenden Sie dabei in Abschnitt 5.3 angegebenen Varianten der Prozedur `versuche`. Wie groß ist die Ersparnis?
22. Entwerfen Sie mit Hilfe des Backtracking-Verfahrens einen Algorithmus zur Bestimmung der chromatischen Zahl eines ungerichteten Graphen (Hinweis: Verwenden Sie die Funktion `färbung` und gehen Sie wie bei der binären Suche vor).

23. Es sei G ein ungerichteter Graph mit Eckenmenge E , $e \in E$ und $N(e)$ die Menge der Nachbarn von e in G .

a) Beweisen Sie folgende Aussage: Es sei I_1 eine maximale unabhängige Menge des von $E \setminus \{e\}$ induzierten Untergraphen und I_2 eine maximale unabhängige Menge des von $E \setminus (\{e\} \cup N(e))$ induzierten Untergraphen von G . Dann ist die größere der beiden Mengen I_1 und I_2 eine maximale unabhängige Menge von G .

b) Geben Sie einen linearen Algorithmus an, der in einem Graph G mit

$$\Delta(G) \leq 2$$

eine maximale unabhängige Menge bestimmt.

c) Beweisen Sie, daß die folgende Funktion `unabhängig` eine maximale unabhängige Menge von G bestimmt.

```
function unabhängig(G : Graph) : Set of Ecken;
begin
  if  $\Delta(G) \leq 2$  then
    unabhängig := unabhängige Menge von G;
  else begin
    Wähle Ecke e von G mit  $g(e) \geq 3$ ;
     $I_1 := \text{unabhängig}(G \setminus \{e\})$ ;
     $I_2 := \text{unabhängig}(G \setminus (\{e\} \cup N(e)))$ ;
    if  $I_1 > I_2$  then
      unabhängig :=  $I_1$ ;
    else
      unabhängig :=  $I_2 \cup \{e\}$ ;
  end
end
```

* d) Bestimmen Sie die Laufzeit der Funktion `unabhängig`.

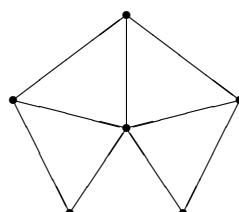
- * 24. Beweisen Sie, daß für einen ungerichteten Graphen G mit n Ecken die Ungleichung

$$2\sqrt{n} \leq \chi(G) + \chi(\bar{G}) \leq n + 1$$

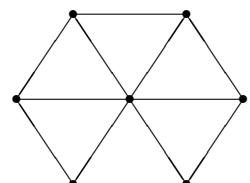
gilt. Hierbei ist \bar{G} das Komplement zu G . Geben Sie Beispiele an, in denen Gleichheit bzw. Ungleichheit gilt.

25. Für jede natürliche Zahl $n \geq 3$ wird der Graph R_n wie folgt definiert: R_n entsteht aus dem Graphen C_n , indem eine zusätzliche Ecke hinzugefügt wird, welche zu jeder Ecke von C_n inzident ist. Die folgende Abbildung zeigt die beiden Graphen R_5 und R_6 . Bestimmen Sie $\chi(R_n)$ in Abhängigkeit von n .

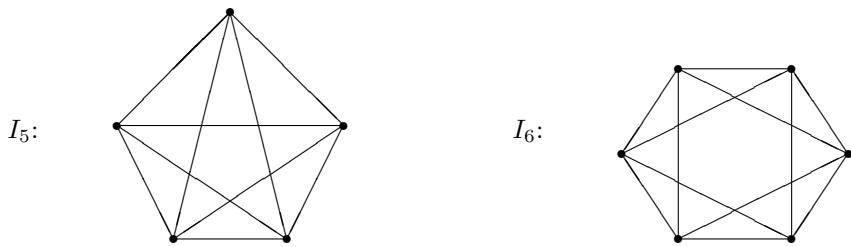
$R_5:$



$R_6:$



26. Für jede natürliche Zahl $n \geq 5$ wird der Graph I_n wie folgt definiert: I_n entsteht aus dem Graphen C_n , indem n Kanten zusätzlich eingefügt werden. Sind die Ecken von C_n fortlaufend von 1 bis n numeriert, so wird eine Kante zwischen der Ecke i und $i+2$ eingefügt. Hierbei werden die Nummern modulo n betrachtet. Somit werden alle Paare von Ecken mit Abstand 2 durch Kanten verbunden. Die folgende Abbildung zeigt die beiden Graphen I_5 und I_6 . Bestimmen Sie $\chi(I_n)$ in Abhängigkeit von n .

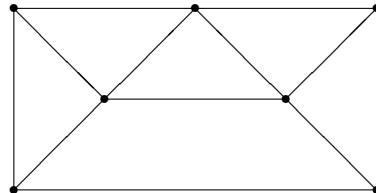


- * 27. Für jede natürliche Zahl $n \geq 3$ wird der Graph G_n wie folgt definiert: Es sei $G_3 = C_5$, und für $n \geq 3$ sei G_n schon konstruiert. G_{n+1} entsteht aus G_n , indem neue Ecken und Kanten hinzugefügt werden. Zu jeder Ecke e von G_n wird eine neue Ecke e' in G_{n+1} eingefügt. Diese ist zu den Nachbarn von e benachbart. Ferner wird noch eine Ecke f in G_{n+1} eingefügt, und diese ist zu allen Ecken e' inzident; d.h. bezeichnet man die Anzahl der Ecken von G_n mit e_n und die der Kanten mit k_n , so gilt $e_{n+1} = 2e_n + 1$ und $k_{n+1} = 3k_n + e_n$. Beweisen Sie, daß $\chi(G_n) = n$ und $\omega(G_n) = 2$ ist (Vergleichen Sie hierzu auch den in Aufgabe 14 dargestellten Graphen).
28. Bestimmen Sie die kantenchromatischen Zahlen der Graphen R_n und I_n aus den Aufgaben 25 und 26.
29. Gegeben sei ein Baum B mit n Ecken. Von den n Ecken sind bereits s Ecken, welche paarweise nicht benachbart sind, mit der Farbe 1 gefärbt. Wie viele Farben werden maximal benötigt, um daraus eine Färbung für B zu erzeugen? Geben Sie einen effizienten Algorithmus für dieses Problem an!
30. In einem Land soll eine begrenzte Anzahl von Sendefrequenzen an lokale Radiostationen vergeben werden. Dabei müssen zwei Radiostationen verschiedene Sendefrequenzen zugeordnet bekommen, wenn ihre geographische Entfernung einen gewissen Wert unterschreitet. Geben Sie eine graphentheoretische Beschreibung dieses Problemes an!
31. Die Korrektheit der Prozedur **5-färbung** stützt sich wesentlich darauf, daß es in einem planaren Graphen eine Ecke e mit $g(e) \leq 5$ gibt. Es sei nun G ein gerichteter Graph mit Eckenmenge $\{e_1, \dots, e_n\}$, und für alle $j = 1, \dots, n$ sei G_j der von $\{e_1, \dots, e_j\}$ induzierte Untergraph von G . Für jedes j habe e_j in G_j höchstens den Eckengrad 5. Liefert die Prozedur **5-färbung** auf jeden Fall eine Färbung für G , welche maximal fünf Farben verwendet? Falls nein, so geben Sie ein Gegenbeispiel mit $\chi(G) > 5$ an.

32. Es sei π eine Permutation der Menge $1, \dots, n$. Es sei G^π ein Graph mit n Ecken, in dem die Ecken i, j benachbart sind, falls die Reihenfolge von i und j durch π vertauscht wird. In welchem Zusammenhang stehen G^π und G_π ?
33. Es sei G ein Permutationsgraph. Beweisen Sie, daß G oder das Komplement \overline{G} eine Clique C mit $|C| \geq \sqrt{n}$ enthält.
34. Liefert der Greedy-Algorithmus für Permutationsgraphen immer eine minimale Färbung?
35. Es sei G_π ein Permutationsgraph. Beweisen Sie, daß auch das Komplement von G_π ein Permutationsgraph ist, und geben Sie eine entsprechende Permutation an.
36. Bestimmen Sie für den Permutationsgraphen G_π mit $\pi = [5, 3, 1, 6, 4, 2]$ eine minimale Färbung.
37. Es sei π eine Permutation. Beweisen Sie folgende Aussagen über die Eckengrade von $\pi(1)$ und $\pi(n)$ in G_π :

$$g(\pi(1)) = \pi(1) - 1 \text{ und } g(\pi(n)) = n - \pi(n).$$

38. Beweisen Sie, daß das Komplement eines Intervallgraphen transitiv orientierbar ist.
39. Es sei G ein ungerichteter Graph mit mehr als zehn Ecken. Beweisen Sie, daß G oder \overline{G} nicht planar ist.
40. Für welches n ist der folgende Graph n -kritisch?



41. Eine Färbung eines ungerichteten Graphen heißt *nicht trivial*, wenn es zu je zwei Farben eine Kante gibt, deren Enden mit diesen Farben gefärbt sind; d.h. es gibt kein Paar von Farbklassen, welche zusammengefaßt werden können. Es sei $\chi_n(G)$ die maximale Anzahl von Farben in einer nicht trivialen Färbung von G . Beweisen Sie folgende Ungleichung:

$$\frac{n - \chi(G)}{n - \chi_n(G)} \leq 2.$$

Entwerfen Sie einen Algorithmus zur Bestimmung einer nicht trivialen Färbung.

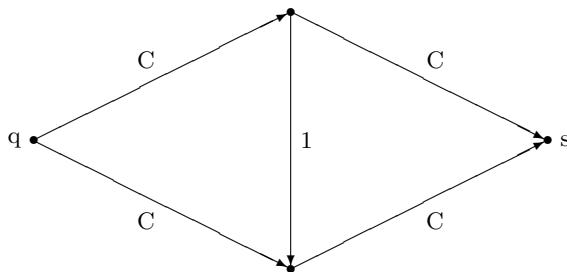
42. Es sei G ein ungerichteter Graph mit Eckenmenge $E = \{1, \dots, n\}$. Es sei G_T der gerichtete Graph, welcher aus G entsteht, indem jede Kante in Richtung der höheren Eckenzahl orientiert wird. Es sei h die maximale Höhe einer Ecke in G_T . Beweisen Sie, daß der Greedy-Algorithmus für G maximal $h + 1$ Farben vergibt.
43. Es sei G ein ungerichteter Graph und a, b nichtbenachbarte Ecken von G . Es sei G^+ der Graph, welcher aus G entsteht, indem die beiden Ecken a und b zu einer Ecke verschmolzen werden. Ferner sei G^- der Graph, welcher aus G entsteht, indem eine Kante zwischen a und b eingefügt wird. Beweisen Sie, daß

$$\chi(G) = \min(\chi(G^+), \chi(G^-))$$

gilt. Entwerfen Sie einen rekursiven Algorithmus zur Bestimmung einer minimalen Färbung eines ungerichteten Graphen, welcher auf dieser Gleichung aufbaut.

Kapitel 6

Flüsse in Netzwerken



Dieses Kapitel behandelt Verfahren zur Bestimmung von maximalen Flüssen in Netzwerken. Die betrachteten Netzwerke haben obere Kapazitätsbeschränkungen. Zunächst werden die grundlegenden Resultate von Ford und Fulkerson bewiesen. Danach werden zwei Algorithmen zur Bestimmung von maximalen Flüssen vorgestellt. Der erste basiert auf Erweiterungswegen minimaler Länge, der zweite verwendet die Technik der blockierenden Flüsse. Für den Fall, daß die Kapazitäten 0 oder 1 sind, werden schärfere Komplexitätsabschätzungen angegeben. Anwendungen von Netzwerkalgorithmen werden im nächsten Kapitel behandelt.

6.1 Einleitung

In diesem Kapitel werden kantenbewertete gerichtete Graphen betrachtet. Die Bewertungen der Kanten sind nichtnegative reelle Zahlen und werden im folgenden *Kapazitäten* genannt. Es seien q und s zwei ausgezeichnete Ecken; q heißt *Quelle* und s heißt *Senke*. Einen solchen Graphen nennt man ein *q - s -Netzwerk*. Mit ihnen können viele Anwendungen modelliert werden. Abbildung 6.1 zeigt ein q - s -Netzwerk. Es repräsentiert ein System von Pipelines, welches Rohöl von einem Ort q zu einem anderen Ort s transportieren soll. Die Bewertungen der Kanten sind die maximalen Transportkapazitäten pro Zeiteinheit der einzelnen Pipelines. Die von q und s verschiedenen Ecken

sind Zwischenstationen. Aus einer solchen Zwischenstation muß pro Zeiteinheit die gleiche Menge abfließen wie hineinfliessen. Die Aufgabe besteht nun in der Bestimmung der maximal von q nach s transportierbaren Menge von Rohöl pro Zeiteinheit. Ferner ist die entsprechende Auslastung der einzelnen Teilstücke zu bestimmen.

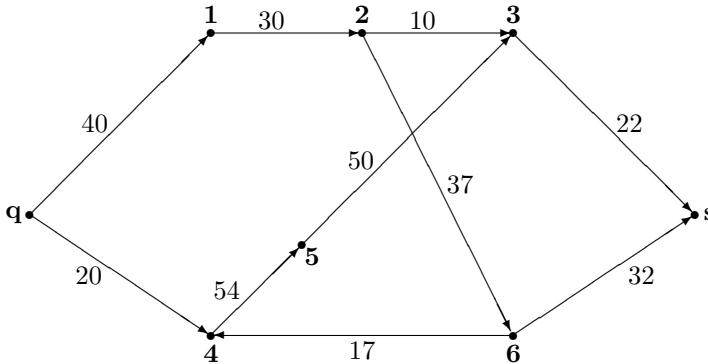


Abbildung 6.1: Ein q - s -Netzwerk

Ein kantenbewerteter gerichteter Graph G heißt q - s -Netzwerk, falls folgende zwei Bedingungen erfüllt sind:

- a) Die Bewertungen $\kappa(k)$ der Kanten sind nichtnegative reelle Zahlen.
- b) q und s sind Ecken von G mit $g^-(q) = g^+(s) = 0$.

Um das obige Transportproblem vollständig zu beschreiben, wird noch der Begriff des q - s -Flusses benötigt. Eine Funktion f , welche jeder Kante eines q - s -Netzwerkes G eine nichtnegative reelle Zahl zuordnet, heißt q - s -Fluß, falls folgende Bedingungen erfüllt sind:

- a) Für jede Kante k von G gilt: $0 \leq f(k) \leq \kappa(k)$.
- b) Für jede Ecke $e \neq q, s$ von G gilt:

$$\sum_{k=(j,e) \in K} f(k) = \sum_{k=(e,j) \in K} f(k),$$

wobei K die Menge der Kanten von G ist. Die erste Bedingung garantiert, daß der Fluß durch eine Kante die Kapazität nicht übersteigt. Die zweite Bedingung kann als Erhaltungsbedingung interpretiert werden: Der Fluß in eine Ecke e muß gleich dem Fluß aus dieser Ecke sein. Dies gilt für alle Ecken e außer der Quelle und der Senke. Falls klar ist, welche Ecken Quelle und Senke sind, wird im folgenden nur noch von einem

Netzwerk bzw. einem *Fluß* gesprochen. Der Fluß, der auf jeder Kante den Wert 0 hat, heißt *trivialer* Fluß.

Der *Wert* eines Flusses f ist gleich dem Gesamtfluß aus der Quelle. Er wird mit $|f|$ bezeichnet:

$$|f| = \sum_{k=(q,i) \in K} f(k).$$

Aus den Eigenschaften eines Flusses folgt sofort, daß auch

$$|f| = \sum_{k=(i,s) \in K} f(k)$$

gilt; d.h. was aus der Quelle hinausfließt, fließt in die Senke hinein. Ein Fluß f heißt *maximal*, wenn $|f| \geq |f'|$ für alle Flüsse f' von G gilt. Gibt es keinen Weg von q nach s , so ist der triviale Fluß maximal.

In der Literatur findet man leicht unterschiedliche Definitionen für q - s -Netzwerke. Gelegentlich wird die zweite Bedingung weggelassen; d.h. Kanten mit Anfangsseite s oder Endseite q sind erlaubt. Dies ist keine wesentliche Erweiterung, denn diese Kanten tragen nichts zum Gesamtfluß eines Netzwerkes bei (vergleichen Sie Aufgabe 1). Die in diesem Kapitel angegebenen Algorithmen arbeiten auch unter dieser Voraussetzung. In diesem Fall ist der Wert eines Flusses f wie folgt definiert:

$$|f| = \sum_{k=(q,i) \in K} f(k) - \sum_{k=(i,q) \in K} f(k).$$

Im folgenden ist ein Fluß f für das Netzwerk aus Abbildung 6.1 angegeben:

$$\begin{array}{lll} f(q, 1) = 30 & f(2, 6) = 20 & f(5, 3) = 0 \\ f(q, 4) = 0 & f(3, s) = 10 & f(6, 4) = 0 \\ f(1, 2) = 30 & f(4, 5) = 0 & f(6, s) = 20 \\ f(2, 3) = 10 & & \end{array}$$

Der Wert des Flusses f ist 30. Das zentrale Problem der Netzwerktheorie ist die Bestimmung maximaler Flüsse. Da die Anzahl der verschiedenen Flüsse auf einem Netzwerk im allgemeinen unendlich ist, ist die Existenz eines maximalen Flusses nicht direkt ersichtlich (Man beachte dazu, daß der Fluß durch eine Kante keine ganze Zahl sein muß).

Um eine obere Schranke für den Wert eines Flusses anzugeben, wird der Begriff des q - s -Schnittes benötigt. Es sei G ein q - s -Netzwerk mit Eckenmenge E . Ein q - s -*Schnitt* (oder kurz *Schnitt*) von G ist eine Partition¹ von E in die Mengen X und \bar{X} , so daß $q \in X$ und $s \in \bar{X}$. Jedem Schnitt (X, \bar{X}) wird die Menge $\{k \in K \mid k = (i, j) \text{ mit } i \in X, j \in \bar{X}\}$ von Kanten zugeordnet. Manchmal wird auch diese Menge als Schnitt bezeichnet. Die

¹Eine *Partition* einer Menge M ist eine Zerlegung von M in disjunkte Teilmengen, deren Vereinigung gleich M ist. Ist $X \subseteq M$, so ist $\bar{X} = M \setminus X$.

Kapazität eines Schnittes (X, \overline{X}) ist gleich der Summe der Kapazitäten aller Kanten mit Anfangsseite in X und Endseite in \overline{X} . Sie wird mit $\kappa(X, \overline{X})$ bezeichnet:

$$\kappa(X, \overline{X}) = \sum_{\substack{k=(i,j) \in K \\ i \in X, j \in \overline{X}}} \kappa(k).$$

Ist ein Fluß f auf dem Netzwerk gegeben, so definiert man analog dazu den Fluß eines Schnittes:

$$f(X, \overline{X}) = \sum_{\substack{k=(i,j) \in K \\ i \in X, j \in \overline{X}}} f(k).$$

Auf die gleiche Art kann auch $f(\overline{X}, X)$ definiert werden. Für das Netzwerk aus Abbildung 6.1 ist mit $X = \{q, 1, 2, 4\}$ und $\overline{X} = \{3, 5, 6, s\}$ ein Schnitt mit Kapazität 101 gegeben. Für den oben angegebenen Fluß f ist $f(\overline{X}, X) = 30$. Das folgende Lemma gibt eine obere Schranke für den Wert eines maximalen Flusses in einem Netzwerkes an.

Lemma. Es sei f ein Fluß eines Netzwerkes G . Dann gilt für jeden Schnitt (X, \overline{X}) von G

$$|f| = f(X, \overline{X}) - f(\overline{X}, X)$$

und

$$|f| \leq \min\{\kappa(X, \overline{X}) \mid (X, \overline{X}) \text{ ist ein Schnitt von } G\}.$$

Beweis. Nach der Erhaltungsbedingung gilt:

$$\begin{aligned} |f| &= \sum_{i \in X} \left(\sum_{k=(i,j) \in K} f(k) - \sum_{k=(j,i) \in K} f(k) \right) \\ &= \sum_{\substack{k=(i,j) \in K \\ i \in X, j \in \overline{X}}} f(k) + \sum_{\substack{k=(i,j) \in K \\ i, j \in X}} f(k) - \sum_{\substack{k=(j,i) \in K \\ i, j \in X}} f(k) - \sum_{\substack{k=(j,i) \in K \\ i \in X, j \in \overline{X}}} f(k). \end{aligned}$$

Da sich die beiden mittleren Summen wegheben, gilt

$$|f| = f(X, \overline{X}) - f(\overline{X}, X).$$

Weil $0 \leq f(k) \leq \kappa(k)$ für alle Kanten k von G gilt, folgt

$$|f| = f(X, \overline{X}) - f(\overline{X}, X) \leq f(X, \overline{X}) \leq \kappa(X, \overline{X}).$$

Hieraus ergibt sich die Aussage des Lemmas. ■

Als Folgerung ergibt sich sofort, daß der Wert eines maximalen Flusses durch ein Netzwerk G kleiner oder gleich der minimalen Kapazität eines Schnittes von G ist. Mit anderen Worten: Durch ein Netzwerk kann höchstens so viel hindurchfließen, wie dessen „engste Stelle“ durchläßt. Im folgenden wird sogar bewiesen, daß diese beiden Größen übereinstimmen. Daraus folgt dann auch die Existenz eines Flusses mit maximalem Wert in einem beliebigen Netzwerk. Um diese Aussage zu beweisen, wird der Begriff des Erweiterungsweges benötigt.

Es sei G ein Netzwerk mit Fluß f . Mit G^u wird der G zugrundeliegende ungerichtete Graph bezeichnet. Ist k eine Kante von G , so bezeichnet k^u die k entsprechende ungerichtete Kante von G^u . Eine Folge W von Kanten k_1, k_2, \dots, k_s von G heißt *Erweiterungsweg* bezüglich f , falls die Folge $k_1^u, k_2^u, \dots, k_s^u$ ein Weg W^u in G^u ist, und falls für jede Kante k_i folgende Bedingung erfüllt ist:

- a) Stimmt die Richtung, in der k_i^u auf W^u durchlaufen wird, mit der von k_i überein, so ist $f(k) < \kappa(k)$.
- b) Stimmt die Richtung, in der k_i^u auf W^u durchlaufen wird, nicht mit der von k_i überein, so ist $f(k) > 0$.

Gibt es in G zwei Ecken, die durch zwei Kanten unterschiedlicher Richtung verbunden sind, so gibt es in dem ungerichteten Graphen G^u zwischen diesen Ecken zwei Kanten; d.h. auch wenn G schlicht ist, muß G^u nicht schlicht sein.

Ein Erweiterungsweg ist also streng genommen kein Weg in G . Kanten eines Erweiterungsweges nennt man dabei *Vorwärtskanten*, falls ihre Richtung mit der des Wegs W^u übereinstimmen, und andernfalls *Rückwärtskanten*. Von besonderem Interesse sind Erweiterungswege mit Startecke q und Endecke s . Sofern Start- und Endecke eines Erweiterungsweges nicht explizit angegeben sind, ist q die Startecke und s die Endecke. Für den oben angegebenen Fluß f für das Netzwerk aus Abbildung 6.1 ist $q, 4, 5, 3, s$ ein Erweiterungsweg, welcher nur aus Vorwärtskanten besteht; $q, 4, 5, 3, 2, 6, s$ hingegen ist ein Erweiterungsweg, welcher auch eine Rückwärtskante enthält. Dagegen ist $q, 1, 2, 3, s$ kein Erweiterungsweg, denn die Kanten $(1, 2)$ und $(2, 3)$ erfüllen nicht die angegebenen Bedingungen. Man beachte, daß die erste Kante in einem Erweiterungsweg stets eine Vorwärtskante ist.

Mit Hilfe eines Erweiterungsweges eines Flusses kann ein neuer Fluß mit einem höheren Wert konstruiert werden. Die Grundidee ist, daß die Vorwärtskanten eines Erweiterungsweges noch nicht voll ausgenutzt sind, während der Fluß der Rückwärtskanten noch vermindert werden kann. Letzteres ist notwendig, um der Flußerhaltungsbedingung zu genügen. Um wieviel kann der Wert des Flusses auf diese Art erhöht werden? Dies hängt von der „engsten Stelle“ des Erweiterungsweges ab. Dazu werden folgende Größen definiert:

$$\begin{aligned} f_v &= \min \{\kappa(k) - f(k) \mid k \text{ ist Vorwärtskante auf dem Erweiterungsweg}\}, \\ f_r &= \min \{f(k) \mid k \text{ ist Rückwärtskante auf dem Erweiterungsweg}\}. \end{aligned}$$

f_v ist der größte Wert, um den man den Fluß durch alle Vorwärtskanten des Erweiterungsweges erhöhen kann. Analog ist f_r der größte Wert, um den man den Fluß durch

alle Rückwärtskanten vermindern kann. Enthält der Erweiterungsweg keine Rückwärtskante, so setzt man $f_r = \infty$. Der Fluß f kann nun um

$$f_\Delta = \min\{f_v, f_r\}$$

erhöht werden. Man beachte, daß stets $f_\Delta > 0$ ist. Es wird nun ein neuer Fluß f' konstruiert. Dieser entsteht aus f , indem der Fluß durch jede Vorwärtskante um f_Δ erhöht und durch jede Rückwärtskante um f_Δ erniedrigt wird. f' ist wieder ein Fluß, und es gilt $|f'| = |f| + f_\Delta$. Da $f_\Delta > 0$ ist, hat der neue Fluß f' einen höheren Wert.

Für den Erweiterungsweg $q, 4, 5, 3, 2, 6, s$ für das Netzwerk aus Abbildung 6.1 und dem oben angegebenen Fluß ergibt sich:

$$f_v = 12 \quad f_r = 10 \quad f_\Delta = 10.$$

Der neue Fluß f' , der sich daraus ergibt, ist in Abbildung 6.2 dargestellt. Jede Kante trägt dabei zwei durch ein Komma getrennte Bewertungen: Zuerst wird der Fluß und dann die Kapazität angegeben. Der Wert des Flusses f' ist 40.

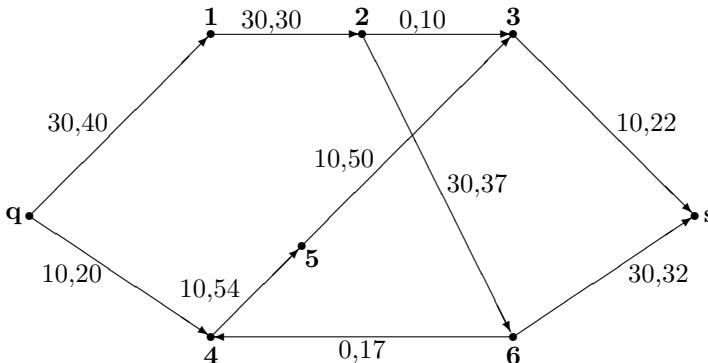


Abbildung 6.2: Der Fluß f'

Das Konzept der Erweiterungswege bildet die Grundlage vieler Algorithmen zur Bestimmung maximaler Flüsse. Hierbei geht man von einem existierenden Fluß f_0 aus (z.B. von dem trivialen Fluß). Mit Hilfe eines Erweiterungsweges für f_0 wird ein zweiter Fluß f_1 mit $|f_1| > |f_0|$ konstruiert. Auf diese Weise bekommt man eine Folge von Flüssen f_0, f_1, f_2, \dots mit $|f_0| < |f_1| < |f_2| < \dots$. Dabei entstehen zwei Probleme: Wie findet man Erweiterungswege, und warum bricht dieses Verfahren irgendwann mit einem Fluß mit maximalem Wert ab? Die grundlegenden Ergebnisse über Erweiterungswege stammen von L.R. Ford und D.R. Fulkerson; sie sind im nächsten Abschnitt zusammengefaßt.

6.2 Der Satz von Ford und Fulkerson

Im letzten Abschnitt wurde gezeigt, daß der Wert eines maximalen Flusses durch das Minimum der Kapazitäten aller Schnitte begrenzt ist. Da es nur endlich viele Schnitte

gibt, existiert daher ein Schnitt, dessen Kapazität κ_0 minimal ist. Im folgenden wird nun gezeigt, daß es einen Fluß f mit $|f| = \kappa_0$ gibt. Daraus folgt die Existenz eines Flusses mit maximalem Wert. Ferner wird auch gezeigt, daß ein Fluß, für den es keinen Erweiterungsweg gibt, maximal ist.

Satz (FORD UND FULKERSON).

- Es sei f ein Fluß eines Netzwerkes. Genau dann ist f ein maximaler Fluß, wenn es keinen Erweiterungsweg bezüglich f gibt.
- Der Wert eines maximalen Flusses in einem Netzwerk ist gleich der minimalen Kapazität eines Schnittes.

Beweis. a) Aus dem letzten Abschnitt folgt, daß ein maximaler Fluß keinen Erweiterungsweg besitzen kann. Es sei nun f ein Fluß, für welchen es keinen Erweiterungsweg gibt. Man bezeichne die Menge aller Ecken e des Netzwerkes, für welche es einen Erweiterungsweg mit Startecke q und Endecke e gibt, mit X . Da es einen Erweiterungsweg von q nach q gibt, ist auch $q \in X$. Dann ist (X, \bar{X}) ein Schnitt. Es sei k eine Kante mit Anfangsseite in X und Endseite in \bar{X} . Aus der Definition von X folgt, daß $f(k) = \kappa(k)$ ist. Somit ist $f(X, \bar{X}) = \kappa(X, \bar{X})$. Ist umgekehrt k eine Kante mit Anfangsseite in \bar{X} und Endseite in X , so ist aus dem gleichen Grund $f(k) = 0$. Somit ist $f(\bar{X}, X) = 0$. Aus dem obigen Lemma folgt nun

$$|f| = f(X, \bar{X}) = \kappa(X, \bar{X}).$$

Da $\kappa(X, \bar{X})$ eine obere Schranke für den Wert eines Flusses ist, ist f ein maximaler Fluß.

b) Es sei f ein maximaler Fluß. Definiere X wie in a). Da der Wert von f maximal ist, ist $s \notin X$. Analog zu a) zeigt man, daß $|f| = \kappa(X, \bar{X})$. Da für jeden Schnitt (Y, \bar{Y}) des Netzwerkes $|f| \leq \kappa(Y, \bar{Y})$ gilt, ist (X, \bar{X}) ein Schnitt mit minimaler Kapazität. ■

Aus diesem Satz folgt noch nicht, daß der im letzten Abschnitt vorgestellte Algorithmus zur Bestimmung eines Flusses mit maximalem Wert terminiert. Zwar existiert zu jedem Fluß, dessen Wert noch nicht maximal ist, ein Erweiterungsweg, welcher zu einem Fluß mit höherem Wert führt. Dies bedeutet aber noch nicht, daß nach endlich vielen Schritten ein Fluß mit maximalem Wert gefunden wird. Ford und Fulkerson haben ein Netzwerk gefunden, für welches die Erhöhungen f_Δ der Flüsse gegen 0 konvergieren, ohne jedoch 0 zu erreichen. Somit terminiert der Algorithmus für dieses Beispiel nicht. Es gilt sogar, daß die Werte der Flüsse nicht gegen den Wert des maximalen Flusses konvergieren, sondern gegen einen kleineren Wert. In diesem Beispiel sind die Kapazitäten natürlich nicht ganzzahlig (vergleichen Sie Aufgabe 2).

Sind alle Kapazitäten ganzzahlig, so ist auch f_Δ ganzzahlig und mindestens gleich 1 (unter der Voraussetzung, daß auch f ganzzahlig ist). In diesem Fall terminiert der Algorithmus nach endlich vielen Schritten. Ferner folgt auch, daß der Wert eines maximalen Flusses ganzzahlig ist. Startet man mit dem trivialen Fluß, so erhält man einen maximalen Fluß f mit der Eigenschaft, daß $f(k)$ für jede Kante k ganzzahlig ist.

Die Anzahl der Schritte, die der Algorithmus benötigt, hängt bei diesem Verfahren nicht nur von der Anzahl der Ecken und Kanten des Netzwerkes ab, sondern auch von den Kapazitäten. Dazu betrachte man das Netzwerk in Abbildung 6.3 mit $C \in \mathbb{N}$. Der Wert eines maximalen Flusses ist $2C$.

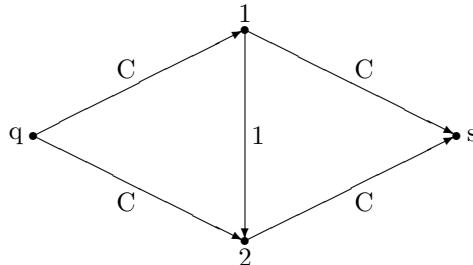


Abbildung 6.3: Ein Netzwerk mit ganzzahligen Kapazitäten

Falls der Algorithmus mit dem trivialen Fluß startet und abwechselnd die beiden Erweiterungswege $q, 1, 2, s$ und $q, 2, 1, s$ verwendet, wird der Wert des Flusses jeweils um 1 erhöht. Es sind somit insgesamt $2C$ Schritte notwendig, um zu dem maximalen Fluß zu gelangen. Dieses Beispiel zeigt, daß man die Erweiterungswege in jedem Schritt sorgfältig auswählen muß. Wählt man im obigen Beispiel direkt die Erweiterungswege $q, 1, s$ und $q, 2, s$, so wird in zwei Schritten ein maximaler Fluß gefunden. Im nächsten Abschnitt werden mehrere Verfahren zur Auswahl von Erweiterungswegen vorgestellt.

6.3 Bestimmung von Erweiterungswegen

Es sei G ein Netzwerk mit Kantenmenge K und einem Fluß f . Dann ist:

$$\overleftarrow{K} = \{(j, i) \mid (i, j) \in K\}$$

Somit geht \overleftarrow{K} aus K hervor, indem die Richtungen der Kanten geändert werden. Ist $k = (j, i) \in K$, so ist $\overleftarrow{k} = (i, j)$. Es sei G_f der gerichtete kantenbewertete Graph mit gleicher Eckenmenge wie G und folgenden Kanten:

- a) $k \in K$ gehört zu G_f , falls $f(k) < \kappa(k)$, und in diesem Fall trägt k die Bewertung $\kappa(k) - f(k)$;
- b) $\overleftarrow{k} \in \overleftarrow{K}$ gehört zu G_f , falls $f(k) > 0$, und in diesem Fall trägt \overleftarrow{k} die Bewertung $f(k)$.

Existieren in G Kanten mit entgegengesetzten Richtungen, so kann dies dazu führen, daß es in G_f Ecken gibt, zwischen denen es zwei Kanten mit der gleichen Richtung gibt. Diese kann man zu einer Kante zusammenfassen und die Bewertungen addieren.

Dadurch wird G_f ein schlichter Graph. Im folgenden wird dies nicht vorgenommen. In den dargestellten Algorithmen wird der Graph G_f nie explizit erzeugt, sondern er dient nur zum besseren Verständnis der Vorgehensweise. Im Rest dieses Kapitels wird folgende Bezeichnungsweise verwendet: Ist k eine Kante eines Netzwerkes G , so bezeichnet \bar{k} die zu k entgegengesetzt gerichtete Kante, die nicht in G liegt.

Die Erweiterungswege von G bezüglich f entsprechen Wegen in G_f und umgekehrt. Abbildung 6.4 zeigt den Graphen G_f zu dem in Abbildung 6.1 dargestellten Netzwerk und dem in Abschnitt 6.1 angegebenen Fluß f .

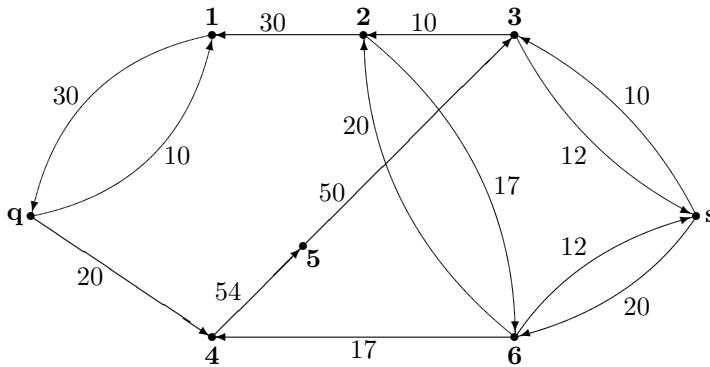


Abbildung 6.4: Der Graph G_f zu dem Netzwerk aus Abbildung 6.1

Die Bestimmung der Erweiterungswege kann nun mit Hilfe von G_f durchgeführt werden. Jeder Weg in G_f von q nach s ist ein Erweiterungsweg, welcher zur Konstruktion eines Flusses mit höherem Wert verwendet werden kann. Solche Wege lassen sich mit den in Kapitel 4 dargestellten Verfahren wie Tiefen- oder Breitensuche leicht finden. Das im letzten Abschnitt erwähnte Beispiel zeigt, daß nicht jede Folge von Erweiterungswegen in endlich vielen Schritten zu einem maximalen Fluß führt. Zwar führt dieses Verfahren bei Netzwerken mit ganzzahligen Kapazitäten zu einem maximalen Fluß, aber die Anzahl der Schritte ist dabei von den Kapazitäten der Kanten abhängig. Das gleiche Argument gilt natürlich auch, wenn die Kapazitäten mit Gleitkommadarstellung und fester Genauigkeit repräsentiert werden, wie dies in Programmiersprachen üblich ist. Es ist aber wünschenswert, ein Verfahren zu haben, dessen Laufzeit unabhängig von den Kapazitäten der Kanten ist.

Für die Auswahl der Erweiterungswege bieten sich zwei Möglichkeiten an:

- Unter allen Erweiterungswegen bezüglich f wird immer derjenige gewählt, für den f_Δ maximal ist.
- Unter allen Erweiterungswegen bezüglich f wird immer derjenige gewählt, welcher aus den wenigsten Kanten besteht.

Man kann zeigen, daß Algorithmen, welche die Erweiterungswege immer nach dem ersten Kriterium auswählen, eine Folge von Flüssen erzeugen, deren Werte gegen den Wert eines maximalen Flusses konvergieren. Die Anzahl der Schritte ist aber weiterhin abhängig von den Kapazitäten der Kanten (vergleichen Sie Aufgabe 11). Verfahren, mit denen Erweiterungswege mit maximalem f_Δ bestimmt werden können, sind in Kapitel 8 beschrieben. Im folgenden wird die zweite Alternative ausführlich betrachtet. Sie führt zu einem effizienten Algorithmus, dessen Laufzeit unabhängig von den Werten der Kapazitäten der Kanten ist.

Die Bestimmung von Wegen mit einer minimalen Anzahl von Kanten erfolgt mit Hilfe der Breitensuche. Diese wird auf den Graphen G_f mit Startecke q angewendet. Sobald die Ecke s erreicht ist, hat man einen geeigneten Erweiterungsweg gefunden. Endet die Breitensuche, bevor die Ecke s erreicht wurde, so ist der vorliegende Fluß bereits maximal. Die Anpassung der Breitensuche an die vorliegende Situation ist einfach. Für jede Ecke e , welche die Breitensuche erreicht, wird der Wert, um den der Fluß entlang des Erweiterungsweges von q nach e maximal vergrößert werden kann, abgespeichert; dazu wird das Feld F_Δ verwaltet.

Um den Erweiterungsweg explizit zur Verfügung zu haben, wird der Breitensuchebaum mit dem in Abschnitt 3.3 vorgestellten Vorgängerfeld abgespeichert. Dies geschieht mit Hilfe des Feldes **vorgänger**. Der Graph G_f wird nicht explizit aufgebaut, sondern die Breitensuche untersucht nicht nur die Nachfolger jeder Ecke im Netzwerk G , sondern auch die Vorgänger.

Abbildung 6.5 zeigt die Funktion **erweiterungsweg**. Sie konstruiert für ein Netzwerk G mit einem Fluß f einen Erweiterungsweg von q nach s mit einer minimalen Anzahl von Kanten. Ihr Rückgabewert ist f_Δ , falls ein Erweiterungsweg von q nach s gefunden wurde, und sonst 0. Mit ihrer Hilfe läßt sich das in Abschnitt 6.1 skizzierte Verfahren zur Bestimmung eines maximalen Flusses leicht realisieren.

Mit Hilfe der in Abbildung 6.6 dargestellten Prozedur **erhöheFluß** kann aus einem gegebenen Fluß f eines Netzwerkes G , einem Erweiterungsweg und dem Wert f_Δ ein neuer Fluß mit dem Wert $|f| + f_\Delta$ konstruiert werden. Das Feld **kantenart** wird dazu verwendet, die Richtung einer Kante im Breitensuchebaum zu vermerken: 1 bedeutet Vorwärtskante und 0 Rückwärtskante.

Das vollständige Verfahren läßt sich nun leicht angeben. Ausgehend von dem trivialen Fluß werden sukzessiv Flüsse mit einem höheren Wert konstruiert. Ein maximaler Fluß ist gefunden, sobald kein Erweiterungsweg mehr existiert. Dies führt zu der in Abbildung 6.7 dargestellten Prozedur **maxFluß**.

Bevor dieser Algorithmus, welcher von J. Edmonds und R. Karp stammt, näher analysiert wird, wird er auf das in Abbildung 6.1 dargestellte Netzwerk angewendet. Startend mit dem trivialen Fluß f_0 wird in vier Schritten der maximale Fluß f_4 gefunden. Die Abbildungen 6.8 bis 6.10 zeigen für jeden einzelnen Schritt das Netzwerk, den Fluß f_i , den gefundenen Erweiterungsweg und den Wert $f_{i\Delta}$. Die gefundenen Erweiterungswege sind jeweils fett gezeichnet. Für den Fluß f_4 wird kein Erweiterungsweg mehr gefunden; d.h. f_4 ist maximal. Die Breitensuche erreicht in G_{f_4} nur die Ecke 1. So mit ist $X = \{q, 1\}$, $\bar{X} = \{2, 3, 4, 5, 6, s\}$ ein Schnitt mit minimaler Kapazität; d.h. $\kappa(X, \bar{X}) = |f_4| = 50$.

```

var vorgänger, kantenart : array[1..max] of Integer;

function erweiterungsweg(G : Netzwerk; f : Fluß) : Real;
var
   $F_{\Delta}$  : array[1..max] of Real;
  besucht : array[1..max] of Integer;
  i, j : Integer;
  W : warteschlange of Integer;
begin
  Initialisiere besucht mit 0;
  vorgänger[G.quelle] := 0;
  besucht[G.quelle] := 1;
   $F_{\Delta}$ [G.quelle] :=  $\infty$ ;
   $F_{\Delta}$ [G.senke] := 0;
  W.einfügen(G.quelle);
repeat
  i := W.entfernen;
  for jeden Nachfolger j von i do
    if besucht[j] = 0 and  $f[i, j] < G.\kappa(i, j)$  then
      begin
        besucht[j] := 1;
        W.einfügen(j);
         $F_{\Delta}[j] := \min(G.\kappa(i, j) - f[i, j], F_{\Delta}[i]);$ 
        vorgänger[j] := i; kantenart[j] := 1;
      end;
  for jeden Vorgänger j von i do
    if besucht[j] = 0 and  $f[j, i] > 0$  then begin
      besucht[j] := 1;
      W.einfügen(j);
       $F_{\Delta}[j] := \min(f[j, i], F_{\Delta}[i]);$ 
      vorgänger[j] := i; kantenart[j] := 0;
    end;
  until  $F_{\Delta}[G.senke] > 0$  or W =  $\emptyset$ ;
  erweiterungsweg :=  $F_{\Delta}[G.senke]$ ;
end

```

Abbildung 6.5: Die Funktion erweiterungsweg

```

procedure erhöheFluß(G : Netzwerk;  $f_\Delta$  : Real; var f : Fluß);
var
    i : Integer;
begin
    i := G.senke;
    while (vorgänger[i] ≠ 0) do begin
        if kantenart[i] = 1 then
            f[vorgänger[i], i] := f[vorgänger[i], i] +  $f_\Delta$ 
        else
            f[i, vorgänger[i]] := f[i, vorgänger[i]] -  $f_\Delta$ ;
        i := vorgänger[i];
    end
end

```

Abbildung 6.6: Die Prozedur `erhöheFluß`

```

procedure maxFluß(G : Netzwerk; var f : Fluß);
var
     $f_\Delta$  : Real;
begin
    Initialisiere f mit 0;
     $f_\Delta$  := erweiterungsweg(G,f);
    while  $f_\Delta \neq 0$  do begin
        erhöheFluß(G, $f_\Delta$ ,f);
         $f_\Delta$  := erweiterungsweg(G,f);
    end
end

```

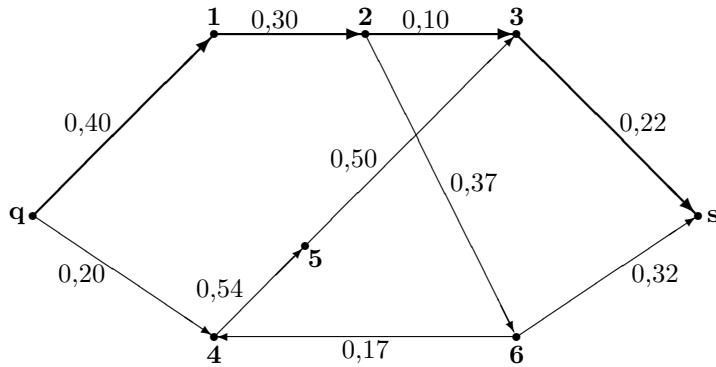
Abbildung 6.7: Die Prozedur `maxFluß`

Eine Kante k eines Netzwerkes G heißt bezüglich eines Flusses f *kritisch*, falls der Fluß durch k gleich 0 oder gleich der Kapazität $\kappa(k)$ der Kante ist. Ist der Fluß durch eine kritische Kante k gleich 0, so kann k nicht als Rückwärtskante in einem Erweiterungsweg auftreten; und ist der Fluß gleich $\kappa(k)$, so kann k nicht als Vorwärtskante in einem Erweiterungsweg auftreten.

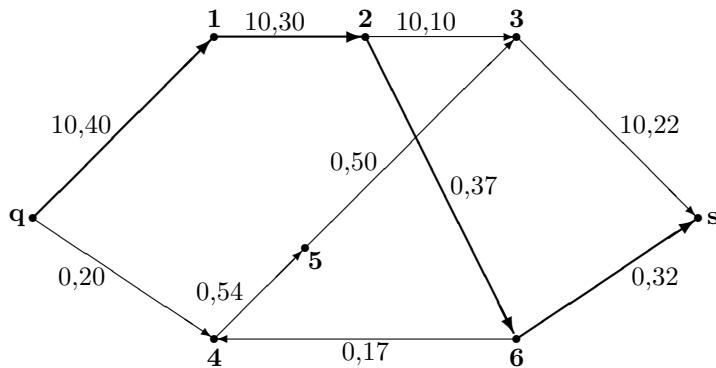
Zur Bestimmung der worst case Komplexität der Prozedur `maxFluß` ist es notwendig, die Anzahl der konstruierten Erweiterungswege abzuschätzen. Dies wird in dem folgenden Satz getan.

Satz. Der Algorithmus von Edmonds und Karp bestimmt in maximal $m n/2$ Schritten einen maximalen Fluß.

Beweis. Es seien f_1, f_2, \dots die durch die Prozedur `erhöheFluß` konstruierten Flüsse und W_1, W_2, \dots die Erweiterungswege, welche zu diesen Flüssen führten. Die Anzahl

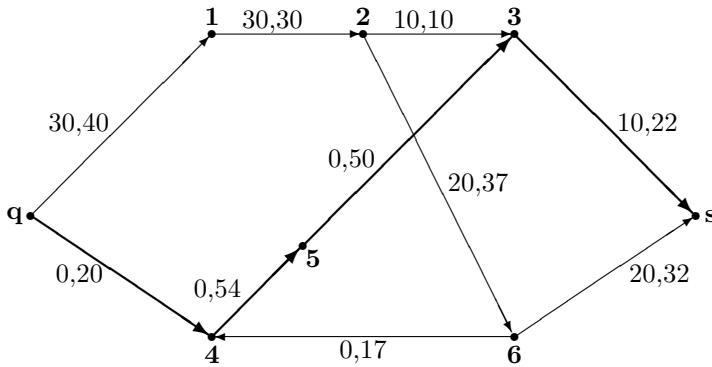


$$|f_0| = 0; \quad \text{Erweiterungsweg : } q, 1, 2, 3, s; \quad f_{0\Delta} = 10$$

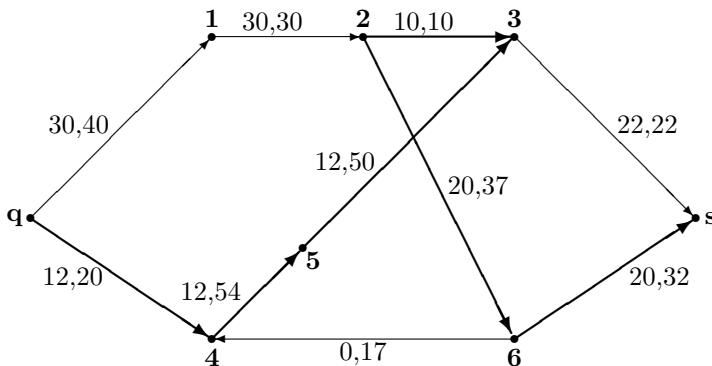


$$|f_1| = 10; \quad \text{Erweiterungsweg : } q, 1, 2, 6, s; \quad f_{1\Delta} = 20$$

Abbildung 6.8: Eine Anwendung des Algorithmus von Edmonds und Karp

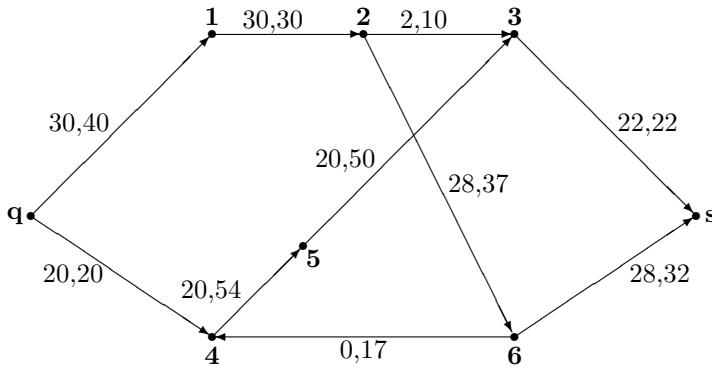


$$|f_2| = 30; \quad \text{Erweiterungsweg : } q, 4, 5, 3, s; \quad f_{2\Delta} = 12$$



$$|f_3| = 42; \quad \text{Erweiterungsweg : } q, 4, 5, 3, 2, 6, s; \quad f_{3\Delta} = 8$$

Abbildung 6.9: Eine Anwendung des Algorithmus von Edmonds und Karp



$$|f_4| = 50; \quad f_4 \text{ ist maximal}$$

Abbildung 6.10: Eine Anwendung des Algorithmus von Edmonds und Karp

der Kanten von W_i wird mit l_i bezeichnet. Ferner sei $d_i(u, v)$ die Anzahl der Kanten des kürzesten Weges zwischen zwei Ecken u und v des Graphen G_{f_i} (f_0 ist der triviale Fluß). Gibt es in G_{f_i} keinen Weg von u nach v , so ist $d_i(u, v) = \infty$. Somit gilt $d_{i-1}(q, s) = l_i$ für alle i . Zunächst wird folgende Aussage bewiesen:

$$(1) \quad d_{i+1}(q, e) \geq d_i(q, e) \text{ für alle Ecken } e \text{ und alle } i \in \mathbb{N}$$

Die Behauptung ist offensichtlich, falls $d_{i+1}(q, e) = \infty$. Der kürzeste Weg von q nach e in $G_{f_{i+1}}$ verwende die Ecken $q = e_0, e_1, \dots, e_h = e$. Somit ist $d_{i+1}(q, e_j) = j$. Zunächst wird gezeigt, daß für $j = 0, \dots, h - 1$ folgende Ungleichung gilt:

$$d_i(q, e_{j+1}) \leq d_i(q, e_j) + 1$$

Dazu werden zwei Fälle betrachtet:

Fall I: Die Kante $k = (e_j, e_{j+1})$ ist in G_{f_i} enthalten. Aus der Eigenschaft der Breitensuche folgt dann sofort $d_i(q, e_{j+1}) \leq d_i(q, e_j) + 1$.

Fall II: Die Kante $k = (e_j, e_{j+1})$ ist nicht in G_{f_i} enthalten. Ist k in $G_{f_{i+1}}$ eine Vorwärtskante, so ist deshalb $f_i(k) = \kappa(k)$, und ist k in $G_{f_{i+1}}$ eine Rückwärtskante, so muß $f_i(k) = 0$ sein. Dies bedeutet, daß der Fluß durch diese Kante im $(i + 1)$ -ten Schritt geändert wurde. Somit ist (e_{j+1}, e_j) eine Kante auf W_{i+1} . Da W_{i+1} ein Weg in dem Breitensuchbaum ist, gilt $d_i(q, e_{j+1}) = d_i(q, e_j) - 1$. Hieraus ergibt sich $d_i(q, e_{j+1}) \leq d_i(q, e_j) + 1$.

Es gilt nun folgende Ungleichung:

$$\begin{aligned}
 d_i(q, e) &= d_i(q, e_h) \\
 &\leq d_i(q, e_{h-1}) + 1 \\
 &\leq d_i(q, e_{h-2}) + 2 \leq \dots \leq \\
 &\leq d_i(q, q) + h \\
 &= d_{i+1}(q, e).
 \end{aligned}$$

Auf die gleiche Weise beweist man auch die folgende Aussage:

$$(2) \quad d_{i+1}(e, s) \geq d_i(e, s) \text{ für alle Ecken } e \text{ und alle } i \in \mathbb{N}.$$

Es sei $k = (u, v)$ eine Kante von G , und W_i, W_j seien Erweiterungswege mit $i < j$, so daß k auf W_i und \overleftarrow{k} auf W_j liegt. Dann gilt:

$$\begin{aligned}
 l_j &= d_{j-1}(q, s) \\
 &= d_{j-1}(q, v) + 1 + d_{j-1}(u, s) \\
 &\geq d_{i-1}(q, v) + 1 + d_{i-1}(u, s) \\
 &= d_{i-1}(q, u) + 1 + 1 + d_{i-1}(v, s) + 1 \\
 &= d_{i-1}(q, s) + 2 \\
 &= l_i + 2.
 \end{aligned}$$

Die erste Ungleichung folgt aus (1) und (2), und die darauffolgende Gleichung ist erfüllt, da (u, v) eine Kante in W_i ist. Somit besitzt W_j mindestens zwei Kanten mehr als W_i .

Jede Erhöhung des Flusses durch einen Erweiterungsweg macht mindestens eine Kante k von G bezüglich des neuen Flusses kritisch. Damit diese Kante k in der gleichen Richtung wieder in einem Erweiterungsweg auftreten kann, muß sie vorher in der entgegengesetzten Richtung in einem Erweiterungsweg vorkommen. Aus dem oben Bewiesenen folgt, daß dieser Erweiterungsweg dann aus mindestens zwei Kanten mehr bestehen muß. Da ein Erweiterungsweg aus maximal $n - 1$ Kanten bestehen kann, kann jede Kante maximal $n/2$ -mal kritisch sein. Somit folgt, daß der Algorithmus maximal $m n/2$ Erweiterungswege konstruiert. Der Algorithmus bestimmt also auch im Fall von irrationalen Kapazitäten in endlich vielen Schritten einen maximalen Fluß. ■

Die Suche nach Erweiterungswegen ist im wesentlichen mit der Breitensuche identisch und hat somit einen Aufwand $O(m)$. Daraus ergibt sich folgender Satz:

Satz. Der Algorithmus von Edmonds und Karp zur Bestimmung eines maximalen Flusses hat eine worst case Komplexität von $O(n m^2)$.

6.4 Der Algorithmus von Dinic

Für dichte Netzwerke (d.h. $m = O(n^2)$) hat der Algorithmus von Edmonds und Karp eine worst case Laufzeit von $O(n^5)$. Für solche Graphen arbeitet ein von E.A. Dinic entwickelter Algorithmus viel effizienter. Durch eine andere Art der Vergrößerung von Flüssen und durch die Betrachtung geeigneter Hilfsnetzwerke gelang es, einen Algorithmus mit einer worst case Laufzeit von $O(n^3)$ zu entwickeln. Die Flußverhöhung erfolgen nicht mehr nur entlang einzelner Wege, sondern auf dem gesamten Netzwerk.

Der ursprünglich von Dinic angegebene Algorithmus hatte eine Laufzeit von $O(n^2m)$. In diesem Abschnitt wird eine Weiterentwicklung dieses Algorithmus vorgestellt, welche eine Laufzeit von $O(n^3)$ hat. Diese Weiterentwicklung stammt von V.M. Malhotra, M. Pramodh Kumar und S.N. Maheswari.

Ausgangspunkt für den Algorithmus von Dinic ist die Beobachtung, daß bei der Konstruktion eines Erweiterungsweges für ein Netzwerk G mit Fluß f viele der betrachteten Kanten nicht in Frage kommen. Die Erweiterungswege erhält man mit Hilfe der Breitensuche und dem Graphen G_f . Der Graph G_f wurde nicht explizit konstruiert, da er nur für einen Breitensuchedurchgang verwendet wurde. Im schlimmsten Fall mußten $m n/2$ Erweiterungswege gesucht werden. Der Algorithmus von Dinic kommt mit maximal n Durchgängen aus. Diese Reduktion wird dadurch erzielt, daß man in jedem Durchgang ein Hilfsnetzwerk aufbaut und in diesem einen Fluß bestimmt. Mit diesem wird dann der Fluß auf dem eigentlichen Netzwerk erhöht.

Das Hilfsnetzwerk entsteht aus G_f , indem dort „überflüssige“ Kanten und Ecken entfernt werden. Auf dem Hilfsnetzwerk wird ein Fluß konstruiert, der zwar nicht maximal sein muß, für den es aber keinen Erweiterungsweg gibt, der nur aus Vorwärtskanten besteht. Einen solchen Fluß nennt man einen *blockierenden Fluß*. Der Fluß f_3 auf dem Netzwerk G aus Abbildung 6.9 ist ein blockierender Fluß. Dieses Beispiel zeigt auch, daß ein blockierender Fluß nicht notwendigerweise ein Fluß mit maximalem Wert ist. Umgekehrt ist natürlich jeder Fluß mit einem maximalen Wert ein blockierender Fluß. Die Grundidee des Algorithmus ist, mittels Hilfsnetzwerken eine Folge von blockierenden Flüssen zu finden, um so zu einem Fluß mit maximalem Wert zu gelangen. Im folgenden wird zunächst die Konstruktion der Hilfsnetzwerke diskutiert.

Im Algorithmus von Edmonds und Karp wurde ein Erweiterungsweg mittels der Breitensuche mit Startecke q auf dem Graphen G_f gefunden. Der Weg von q nach s in dem Breitensuchebaum von G_f ist ein Erweiterungsweg. Kanten, die nicht im Breitensuchebaum enthalten sind, können somit nicht in einem Erweiterungsweg vorkommen. Allerdings ist der Breitensuchebaum mit Startecke q nicht eindeutig bestimmt. In Abhängigkeit der Auswahl der Reihenfolge der Nachfolgeecken ergeben sich verschiedene Breitensuchebäume. Unabhängig von der Reihenfolge der Nachfolger ist die Einteilung der Ecken in die verschiedenen Niveaus. Ist $k = (e, f)$ eine Kante aus irgendeinem Breitensuchebaum, so gilt $\text{Niv}(e) + 1 = \text{Niv}(f)$. Dies führt zur Definition des *geschichteten Hilfsnetzwerkes* G'_f eines Netzwerkes G mit einem Fluß f . Die Eckenmenge E' von G'_f besteht neben der Senke s aus allen Ecken e von G_f mit $\text{Niv}(e) < \text{Niv}(s)$. G'_f enthält genau die Kanten $k = (e, f)$ von G_f , für die $e, f \in E'$ und $\text{Niv}(e) + 1 = \text{Niv}(f)$ gilt. Die Breitensuchenniveaus bilden also die „Schichten“ des Hilfsnetzwerkes. Somit besteht G'_f aus allen kürzesten Wegen in G_f von der Quelle q zur Senke s . Die Kapazitäten

der Kanten von G'_f entsprechen den Bewertungen der Kanten in G_f . G'_f ist somit ein Netzwerk mit gleicher Quelle und Senke wie G . Man beachte, daß es in G_f Kanten mit Endecke q oder Startecke s geben kann, in G'_f aber gilt $g^-(q) = g^+(s) = 0$.

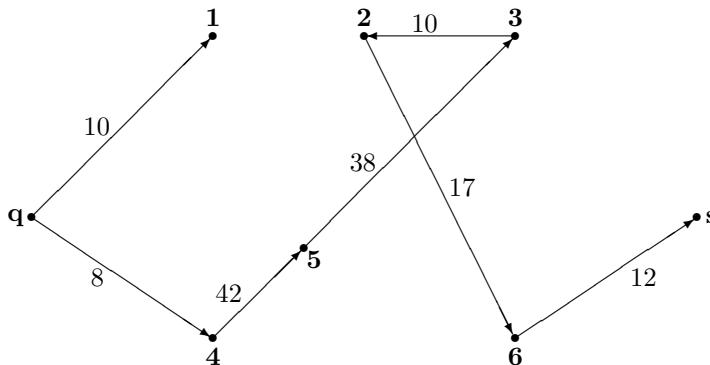
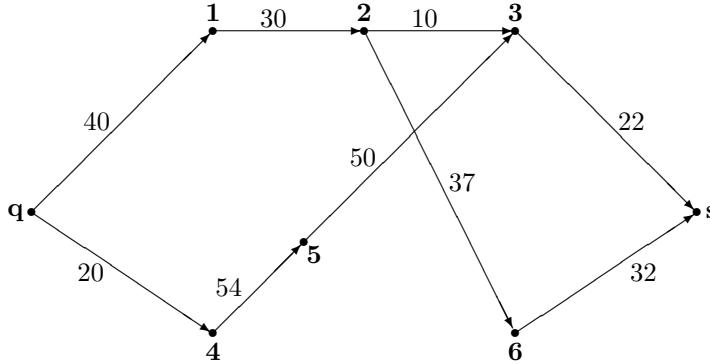


Abbildung 6.11: Die geschichteten Hilfsnetzwerke des Netzwerkes aus Abbildung 6.8, bezüglich der Flüsse f_0 und f_3

Abbildung 6.11 zeigt die geschichteten Hilfsnetzwerke für das Netzwerk aus Abbildung 6.8, bezüglich der dort angegebenen Flüsse f_0 (oben) und f_3 (unten). Angegeben sind jeweils die Kapazitäten der Kanten. Bezuglich f_0 liegt s in Niveau 4 und bezüglich f_3 in Niveau 6. Dies deutet schon die Grundidee des Verfahrens an. Man kann beweisen, daß das Niveau der Senke s in jedem der im Algorithmus von Dinic konstruierten geschichteten Hilfsnetzwerke mindestens um 1 ansteigt. Da s mindestens das Niveau 1 und höchstens das Niveau $n - 1$ hat, folgt daraus, daß man mit maximal $n - 2$ geschichteten Hilfsnetzwerken zu einem maximalen Fluß gelangt. Die Anhebung des Niveaus der Senke beruht darauf, daß zur Konstruktion der geschichteten Hilfsnetzwerke blockierende Flüsse verwendet werden. Mit ihrer Hilfe wird zunächst der Fluß auf dem eigentlichen Netzwerk vergrößert. Dies führt dann zu einem neuen geschichteten Hilfsnetzwerk. Wie

kann man nun mit einem Fluß auf dem geschichteten Hilfsnetzwerk den Fluß auf dem eigentlichen Netzwerk erhöhen? Dazu das folgende Lemma:

Lemma. Es sei G ein Netzwerk mit Fluß f und G'_f das zugehörige geschichtete Hilfsnetzwerk. Ist f' ein Fluß auf G'_f , so gibt es einen Fluß f'' auf G mit $|f''| = |f| + |f'|$.

Beweis. Es sei k eine Kante aus G . Ist k bzw. \overleftarrow{k} keine Kante in G'_f , so sei

$$f'(k) = 0 \text{ bzw. } f'(\overleftarrow{k}) = 0.$$

Somit ist f' für jede Kante von G definiert. Für jede Kante k von G setze man

$$f''(k) = f(k) + f'(k) - f'(\overleftarrow{k}).$$

Existieren in G Kanten mit entgegengesetzten Richtungen, so muß beachtet werden, welche davon eine Kante in G'_f induzierte. Für die andere Kante k gilt dann

$$f'(k) = f'(\overleftarrow{k}) = 0.$$

Man zeigt nun leicht, daß f'' wirklich ein Fluß auf G mit dem angegebenen Wert ist.

■

Abgesehen von der Bestimmung von geschichteten Hilfsnetzwerken und den entsprechenden blockierenden Flüssen, kann der Algorithmus von Dinic jetzt schon angegeben werden. Abbildung 6.12 zeigt die Grobstruktur dieses Algorithmus. Die Erhöhung eines Flusses mittels eines blockierenden Flusses erfolgt gemäß dem letzten Lemma.

```

Initialisiere f mit 0;
Konstruiere G'_f;
while (in G'_f ist s von q aus erreichbar) do begin
    Finde einen blockierenden Fluß h auf G'_f;
    Erhöhe den Fluß f mittels h;
    Konstruiere G'_f;
end

```

Abbildung 6.12: Die Grobstruktur des Algorithmus von Dinic

Im folgenden werden die beiden noch offenen Teilprobleme gelöst. Die Bestimmung eines geschichteten Hilfsnetzwerkes erfolgt ganz analog zu der Funktion `erweiterungsweg`. Die dort verwendete Version der Breitensuche muß allerdings leicht abgeändert werden. Eine Ecke wird besucht, falls sie noch nicht besucht wurde oder falls sie in dem im Aufbau befindlichen Niveau liegt. Eine Ecke kann also somit mehrmals besucht werden. Sie wird aber nur einmal in die Warteschlange eingefügt. Das geschichtete Hilfsnetzwerk wird schichtenweise aufgebaut. Der Algorithmus startet mit einem Hilfsnetzwerk,

welches nur aus der Quelle q besteht. Im Verlauf werden dann die Ecken der einzelnen Niveaus und die Kanten zwischen ihnen eingefügt. Der Algorithmus endet, falls die Senke s in G_f nicht von q aus erreicht werden kann, oder falls das Niveau, welches die Senke enthält, bearbeitet wurde.

Abbildung 6.13 zeigt eine Realisierung der Funktion `hilfsnetzwerk`, welche das geschichtete Hilfsnetzwerk G'_f für ein Netzwerk G mit Fluß f bestimmt. Um alle Kanten zwischen den Niveaus zu finden, wird das Feld `besucht` nicht mit 0, sondern mit n initialisiert. Ist $\text{besucht}[j] > \text{besucht}[i]$ für eine untersuchte Kante (i, j) , so bedeutet dies, daß die Ecke j sich in dem aktuellen Niveau befindet, oder daß j noch nicht besucht wurde. Der Rückgabewert ist `true`, falls s in G_f von q aus erreichbar ist; sonst ist der Rückgabewert `false`. Im ersten Fall ist G'_f das geschichtete Netzwerk.

Wendet man die Funktion `hilfsnetzwerk` auf das Netzwerk aus Abbildung 6.1 an, so entsteht das in Abbildung 6.11 oben dargestellte geschichtete Hilfsnetzwerk. Die Funktion `hilfsnetzwerk` stimmt im wesentlichen noch mit der Breitensuche überein und hat somit eine worst case Komplexität von $O(n + m)$. Aus dem letzten Niveau werden alle Ecken bis auf die Senke entfernt. Dies wird später bei der Konstruktion von blockierenden Flüssen benötigt. Der folgende Satz zeigt, daß der in Abbildung 6.12 angegebene Algorithmus von Dinic maximal $n - 1$ geschichtete Hilfsnetzwerke aufbauen muß. Im letzten ist dann die Senke nicht mehr von der Quelle aus erreichbar, und der Fluß hat somit einen maximalen Wert.

Satz. Nach maximal $n - 2$ Erhöhungen durch blockierende Flüsse liegt ein maximaler Fluß vor.

Beweis. Die Grundidee des Beweises ist, zu zeigen, daß die kürzesten Wege von der Quelle q zur Senke s in den geschichteten Hilfsnetzwerken aus immer mehr Kanten bestehen. Dazu wird gezeigt, daß diese Wege in jedem Schritt mindestens eine Kante mehr enthalten. Da ein kürzester Weg mindestens eine Kante und maximal $n - 1$ Kanten enthält, wird ein Fluß mit maximalem Wert nach spätestens $n - 2$ Schritten erreicht.

Es sei nun G ein Netzwerk mit Fluß f , G'_f das zugehörige geschichtete Hilfsnetzwerk und h ein blockierender Fluß auf G'_f . Für jede Ecke e von G'_f bezeichne man mit $d(e)$ die Anzahl der Kanten des kürzesten Weges von der Quelle q zu e . Ferner sei g der durch die Erhöhung von f durch h entstandene Fluß. Die Anzahl der Kanten des kürzesten Weges von q nach e in G'_g wird mit $d'(e)$ bezeichnet. Falls e nicht in G'_g ist, so sei $d'(e) = \infty$. Zu zeigen ist also $d'(s) > d(s)$. Ist $d'(s) = \infty$, so ist f bereits ein maximaler Fluß. Andernfalls liegt s in G'_g ; d.h. es gibt einen Weg W von q nach s in G'_g . Dieser Weg verwende die Ecken

$$q = e_0, e_1, \dots, e_l = s.$$

Somit gilt $d'(e_i) = i$ für $i = 0, \dots, l$. Es werden nun zwei Fälle betrachtet:

Fall I: Alle Ecken e_i existieren in G'_f . Mittels vollständiger Induktion nach i wird gezeigt, daß $d'(e_i) \geq d(e_i)$ für $i = 0, \dots, l$. Für $i = 0$ ist die Aussage wahr. Angenommen, es gilt $d'(e_i) \geq d(e_i)$. Es wird nun e_{i+1} betrachtet. Falls $d(e_{i+1}) \leq d'(e_i) + 1$ gilt, so

```

function hilfsnetzwerk(G : Netzwerk; f: Fluß;
                        var G' : Netzwerk) : Boolean;
var
    besucht : array[1..max] of Integer;
    i, j : Integer;
    W : warteschlange of Integer;
begin
    Initialisiere besucht mit n und hilfsnetzwerk mit true;
    besucht[G.quelle] := 0;
    Füge G.quelle in G'_ ein;
    W.einfügen(G.quelle);
    repeat
        i := W.entfernen;
        for jeden Nachfolger j von i do
            if besucht[j] > besucht[i] and
                f[i, j] < G.κ(i, j) then begin
                    if besucht[j] = n then begin
                        W.einfügen(j);
                        besucht[j] := besucht[i] + 1;
                        Füge j in G'_ ein;
                    end;
                    Füge Kante (i, j) mit Kapazität G.κ(i, j) - f[i,j]
                    in G'_ ein;
                end;
                for jeden Vorgänger j von i do
                    if besucht[j] > besucht[i] and f[j, i] > 0 then begin
                        if besucht[j] = n then begin
                            W.einfügen(j);
                            besucht[j] := besucht[i] + 1;
                            Füge j in G'_ ein;
                        end;
                        Füge Kante (i,j) mit Kapazität f[j, i] in G'_ ein;
                    end;
                until W = ∅ or besucht[G.senke] <= besucht[W.kopf];
                if besucht[G.senke] < n then begin
                    Entferne bis auf G.senke alle Ecken aus G'_ mit
                    besucht[i] = besucht[G.senke];
                    Entferne alle Kanten, welche zu den gerade entfernten Ecken führen;
                    G'_ .quelle := G.quelle;
                    G'_ .senke := G.senke;
                    hilfsnetzwerk := true;
                end
            end
        end

```

Abbildung 6.13: Die Funktion hilfsnetzwerk

folgt die Behauptung wegen $d'(e_{i+1}) = d'(e_i) + 1$. Es bleibt der Fall $d(e_{i+1}) > d'(e_i) + 1$. Aus der Induktionsannahme folgt dann $d(e_{i+1}) > d(e_i) + 1$. Somit liegen e_i und e_{i+1} nicht in benachbarten Niveaus von G_f . Also kann es in G_f keine Kante von e_i nach e_{i+1} bzw. von e_{i+1} nach e_i geben. Daraus folgt, daß der blockierende Fluß h auch keine solche Kante enthalten kann. Aus der im letzten Lemma beschriebenen Konstruktion zur Bildung von g folgt, daß der Fluß zwischen e_i und e_{i+1} in f und g identisch ist. Somit kann es in G'_g keine Kante von e_i nach e_{i+1} geben. Dieser Widerspruch beendet den Induktionsbeweis. Nun gilt $d'(s) \geq d(s)$. Die Gleichheit würde bedeuten, daß W bezüglich h ein Erweiterungsweg von G'_f ist. Wäre eine der Kanten $k = (e_i, e_{i+1})$ dabei eine Rückwärtskante, so wäre $h(k) = \kappa(k)$. Da aber k auch eine Kante in G'_g ist, kann dies nicht sein. Somit besteht W nur aus Vorwärtskanten. Dies ist aber unmöglich, da h ein blockierender Fluß ist. Hieraus folgt, daß $d'(s) > d(s)$ gilt.

Fall II: Es gibt eine Ecke e_j , die nicht in G'_f ist. Dann ist $e_j \neq s$. Es sei e_j die erste Ecke auf W , welche nicht in G'_f ist. Analog zu Fall I zeigt man, daß $d'(e_i) \geq d(e_i)$ für $i = 0, \dots, j-1$ gilt. Da $k = (e_{j-1}, e_j)$ eine Kante in G'_g ist, muß k auch eine Kante von G_f sein, denn der Fluß durch k wurde durch g nicht geändert. Da e_j nicht G'_f ist, muß somit e_j im gleichen Niveau wie s liegen (in G'_f sind alle Ecken von q aus erreichbar). Somit gilt $d(e_j) = d(s)$. Daraus folgt

$$d'(s) > d'(e_j) = d'(e_{j-1}) + 1 \geq d(e_{j-1}) + 1 \geq d(e_j) = d(s).$$

In jedem Fall gilt $d'(s) > d(s)$, und der Beweis ist vollständig. ■

Um den Algorithmus von Dinic zu vervollständigen, müssen noch die blockierenden Flüsse bestimmt werden. Das ursprünglich von Dinic angegebene Verfahren hatte eine Komplexität von $O(nm)$. Daraus ergab sich für die Bestimmung eines maximalen Flusses ein Algorithmus mit Komplexität $O(n^2m)$. In diesem Abschnitt wird eine Konstruktion von blockierenden Flüssen mit Komplexität $O(n^2)$ vorgestellt. Sie stammt von Malhotra, Pramodh Kumar und Maheswari. Damit ergibt sich eine Gesamtkomplexität von $O(n^3)$. Das Verfahren verzichtet auf die Bestimmung von Erweiterungswegen. Zunächst wird der *Durchsatz* einer Ecke definiert.

Es sei G ein geschichtetes Hilfsnetzwerk mit Eckenmenge E und Kantenmenge K . Für jede Ecke $e \in E$ mit $e \neq q, s$ bezeichnet

$$D(e) = \min \left\{ \sum_{k=(e,f) \in K} \kappa(k), \sum_{k=(f,e) \in K} \kappa(k) \right\}$$

den Durchsatz der Ecke e . Den Durchsatz der Quelle bzw. Senke definiert man als die Summe der Kapazitäten der Kanten, die in der Quelle starten bzw. in der Senke enden. $D(e)$ ist die maximale Flußmenge, die durch die Ecke e fließen kann. Für das in Abbildung 6.11 oben dargestellte geschichtete Hilfsnetzwerk ergibt sich folgender Durchsatz:

Ecke	q	1	2	3	4	5	6	s
Durchsatz	60	30	30	22	20	50	32	54

Die Idee des Verfahrens besteht darin, in jedem Schritt den Fluß auf dem geschichteten Hilfsnetzwerk derart zu erhöhen, daß der Fluß durch die Ecke e mit minimalem Durchsatz gerade $D(e)$ ist. Danach können e und die mit e inzidenten Kanten aus dem Netzwerk entfernt werden. Der Durchsatz aller Ecken muß ebenfalls neu bestimmt werden. Er kann sich auf zwei verschiedene Arten geändert haben. Die Kapazitäten der Kanten werden um den entsprechenden Fluß vermindert. Dadurch kann der Durchsatz sinken. Ferner können Ecken mit Durchsatz 0 entstehen (z.B. die Ecke mit minimalem Durchsatz). Diese werden mit ihren Kanten entfernt und der Durchsatz der Nachbarn abgeändert. Dieser Vorgang wird solange wiederholt, bis der Durchsatz jeder Ecke positiv ist. Die Änderung des Durchsatzes erfolgt mittels der beiden Felder D^+ und D^- . Hierbei ist

$$D^+[e] = \sum_{k=(e,f) \in K} \kappa(k) \quad \text{und} \quad D^-[e] = \sum_{k=(f,e) \in K} \kappa(k).$$

Die Erhöhung des Flusses bis zu dem Punkt, an dem die Ecke mit minimalem Durchfluß gesättigt ist, erfolgt mit den Prozeduren **erweitererückwärts** und **erweiterevorwärts**. Diese verteilen den Fluß $D[e]$, von e aus startend, rückwärts bis zur Quelle bzw. vorwärts bis zur Senke. Dabei werden die beiden Felder D^- und D^+ abgeändert. Kanten, deren Kapazitäten voll ausgenutzt werden, werden entfernt. Die Ecken, deren Durchsatz auf 0 absinkt, werden gesammelt und in der Prozedur **blockfluß** rekursiv entfernt. Abbildung 6.14 zeigt zunächst die Prozedur **blockfluß**.

Wendet man die Prozedur **blockfluß** auf das in Abbildung 6.11 oben dargestellte Hilfsnetzwerk an, so sind die Ecken 4, 3, 1 in dieser Reihenfolge die Ecken mit minimalem Durchsatz 20, 2, 28. Der bestimmte blockierende Fluß ist maximal und ist gleich dem in Abbildung 6.9 angegebenen Fluß f_4 .

Die Prozeduren **erweitererückwärts** und **erweiterevorwärts** arbeiten nach dem gleichen Prinzip. Sie starten bei der Ecke mit minimalem Durchsatz und verteilen diesen Fluß rückwärts zur Quelle bzw. vorwärts zur Senke unter Beibehaltung der Flußerhaltungsbedingung für alle besuchten Ecken. Dies ist möglich, da der Durchsatz der anderen Ecken mindestens genauso hoch ist. Eine Verteilung dieses Flusses rückwärts von der Senke startend würde nicht das gleiche Ziel erreichen. In diesem Fall wäre nicht gesichert, daß die Ecke mit minimalem Durchsatz gesättigt wäre und deshalb entfernt werden könnte. Abbildung 6.15 zeigt die Prozedur **erweitererückwärts**. Analog ist die Prozedur **erweiterevorwärts** zu realisieren.

Im folgenden Satz wird die Korrektheit der Prozedur **blockfluß** bewiesen.

Satz. Die Prozedur **blockfluß** bestimmt einen blockierenden Fluß h auf G .

Beweis. Zunächst wird gezeigt, daß h ein Fluß ist. Die Prozedur **blockfluß** startet mit dem trivialen Fluß. Der Fluß wird nun bei jedem Durchlauf der **repeat**-Schleife geändert. Dies geschieht in den beiden Prozeduren **erweiterevorwärts** und **erweitererückwärts**. Die erste startet einen Breitensuchlauf in der Ecke mit dem geringsten Durchsatz. Das Feld **durchfluß** verwaltet den Fluß, der in eine Ecke hineinfließt muß. Wird der Fluß durch eine Kante $k = (e, f)$ erhöht, so wird **durchfluß[f]**

```

var D, D+, D- : array[1..max] of Real;
S,E : set of Integer;
K : set of Kanten;
procedure blockfluf(G : Netzwerk; var h : Fluß);
var
    i, j, l : Integer;
begin
    Initialisiere h mit 0;
    Initialisiere E mit der Menge der Ecken und K mit der Menge
    der Kanten von G;
    for jede Ecke i do
        Bestimme D+[i] und D-[i] wobei
            D-[G.quelle]=D+[G.senke]=∞;
    repeat
        for jede Ecke i ∈ E do
            D[i] := min(D+[i], D-[i]);
        Wähle j ∈ E so, daß D[j] minimal ist;
        S := {j};
        erweitererückwärts(G,j,h);
        erweitervorwärts(G,j,h);
        while S ≠ ∅ do begin
            i := S.entfernen;
            E.entfernen(i);
            for jede Kante k = (i,l) ∈ K do begin
                D-[l] := D-[l] - G.κ(i,l);
                if D-[l] = 0 then
                    S.einfügen(l);
                    K.entfernen(k);
                end;
            for jede Kante k = (l,i) ∈ K do begin
                D+[l] := D+[l] - G.κ(i,l);
                if D+[l] = 0 then
                    S.einfügen(l);
                    K.entfernen(k);
                end
            end
        until G.senke ∉ E or G.quelle ∉ E;
    end

```

Abbildung 6.14: Die Prozedur blockfluf

```

procedure erweitererückwärts(var G : Netzwerk; j : Integer;
                             var h : Fluß);
var
  durchfluß : array[1..max] of Real;
  W : warteschlange of Integer;
  i,e : Integer;
  m : Real;
begin
  Initialisiere durchfluss mit 0;
  durchfluß[j] := D[j];
  W.einfügen(j);
repeat
  i := W.entfernen;
  while durchfluß[i] ≠ 0 and es gibt
    eine Kante k=(e,i) ∈ E do begin
    m := min(G.κ(e,i), durchfluß[i]);
    h(k) := h(k) + m;
    G.κ(e,i) := G.κ(e,i) - m;
    if G.κ(e,i) = 0 then
      K.entfernen(k)
    D⁻[i] := D⁻[i] - m;
    if D⁻[i] = 0 then
      S.einfügen(i);
    D⁺[e] := D⁺[e] - m;
    if D⁺[e] = 0 then
      S.einfügen(e);
    W.einfügen(e);
    durchfluß[e] := durchfluß[e] + m;
    durchfluß[i] := durchfluß[i] - m;
  end;
  until W = ∅;
end

```

Abbildung 6.15: Die Prozedur erweitererückwärts

entsprechend vermindert und `durchfluß[e]` erhöht. Nachdem eine Ecke i der Warteschlange abgearbeitet wurde, ist `durchfluß[i] = 0`. Die einzige Ausnahme bildet die Senke s , da in einem Netzwerk $g^+(s) = 0$ gilt.

Die Abarbeitung bewirkt, daß der vorhandene Fluß auf die verschiedenen eingehenden Kanten verteilt wird. Am Ende ist somit für jede besuchte Ecke außer der Senke und der Ecke j mit dem minimalen Durchsatz die Flußerhaltungsbedingung erfüllt. Ferner sind die Flüsse durch die Kanten immer kleiner oder gleich den entsprechenden Kapazitäten. Analoges gilt für die Prozedur `erweiterevorwärts`. Somit erfüllen am Ende alle Ecken bis auf Quelle und Senke die Flußerhaltungsbedingung, und dadurch ist h ein Fluß auf G .

Es bleibt noch zu zeigen, daß h ein blockierender Fluß ist. Während der Prozedur `blockfluß` werden an zwei Stellen Kanten entfernt. Zum einen werden Kanten mit maximalem Fluß in den Prozeduren `erweiterevorwärts` und `erweitererückwärts` entfernt. Diese können somit nicht mehr in einem Erweiterungsweg für h vorkommen. In der Prozedur `blockfluß` selber werden Kanten entfernt, falls sie zu Ecken inzident sind, deren Durchsatz schon erschöpft ist. Auch diese Kanten können nicht mehr in einem Erweiterungsweg für h vorkommen. Da es am Ende keinen Weg mehr von der Quelle zur Senke gibt, muß h somit ein blockierender Fluß sein. ■

Es bleibt noch, die Aussage über die Laufzeit des Algorithmus von Dinic zu beweisen.

Satz. Der Algorithmus von Dinic bestimmt einen maximalen Fluß mit dem Zeitaufwand $O(n^3)$.

Beweis. Wie früher schon bewiesen wurde, wird die Prozedur `blockfluß` maximal $(n - 1)$ -mal aufgerufen. Es genügt also, zu zeigen, daß die Prozedur `blockfluß` eine Laufzeit von $O(n^2)$ hat. Da in jeder Iteration der `repeat`-Schleife mindestens eine Ecke entfernt wird, sind maximal $n - 1$ Iterationen notwendig. Zunächst wird die `while`-Schleife für jeden Aufruf der Prozedur `erweitererückwärts` betrachtet. In jedem Durchlauf wird dabei eine Kante entfernt, oder der Durchfluß der Ecke sinkt auf 0; d.h. alle `while`-Schleifen zusammen haben eine Komplexität von $O(m + n)$. Der Rest der Prozedur `erweitererückwärts` hat eine Komplexität von $O(n)$. Die gleiche Analyse kann für die Prozedur `erweiterevorwärts` vorgenommen werden. Da die beiden Prozeduren maximal n -mal aufgerufen werden, ergibt sich ein Aufwand von $O(n^2 + m)$. Die gleiche Argumentation gilt für die `while`-Schleife in `blockfluß`. Somit ergibt sich für die Prozedur `blockfluß` ein Gesamtaufwand von $O(n^2)$. ■

6.5 0-1-Netzwerke

In diesem Abschnitt wird ein wichtiger Spezialfall von allgemeinen Netzwerken betrachtet: Netzwerke, in denen jede Kante die Kapazität 0 oder 1 hat. Solche Netzwerke nennt man *0-1-Netzwerke*. Sie treten in vielen Anwendungen auf. Auf 0-1-Netzwerke existieren maximale Füsse mit speziellen Eigenschaften.

Satz. Es sei G ein 0-1-Netzwerk. Dann existiert ein maximaler Fluß f , welcher auf jeder Kante den Wert 1 oder 0 hat. Ferner gibt es $|f|$ Wege von q nach s , welche paarweise keine Kante gemeinsam haben. Die Kanten dieser Wege haben alle den Fluß 1.

Beweis. Die in diesem Kapitel beschriebenen inkrementellen Algorithmen liefern für Netzwerke mit ganzzahligen Kapazitäten maximale Flüsse, welche nur ganzzahlige Werte annehmen, sofern mit einem ganzzahligen Fluß gestartet wird (z.B. mit dem trivialen Fluß). Somit sind auf 0-1-Netzwerken die Werte dieser maximalen Flüsse auf allen Kanten gleich 0 oder 1. Es sei nun f ein maximaler Fluß von G , welcher auf allen Kanten nur die Werte 0 und 1 annimmt. Dann gibt es einen Weg von q nach s , welcher nur Kanten mit Fluß 1 verwendet. Ändert man den Fluß auf diesen Kanten auf 0, so wird der Wert von f um 1 erniedrigt. Nun muß es wieder einen Weg von q nach s geben, welcher nur Kanten mit Fluß 1 verwendet. Die beiden Wege haben keine Kante gemeinsam. Fährt man auf diese Weise fort, so erhält man $|f|$ kantendisjunkte Wege, auf denen f den Wert 1 hat. ■

Ein Fluß heißt *binär*, wenn er auf jeder Kante den Wert 0 oder 1 hat. Man beachte, daß nicht jeder maximale Fluß auf einem 0-1-Netzwerk ein binärer Fluß ist. Die in den letzten Abschnitten entwickelten Algorithmen lassen sich für 0-1-Netzwerk noch verbessern, so daß man zu effizienteren Verfahren gelangt. Die Grundlage für diesen Abschnitt bildet der Algorithmus von Dinic. In beliebigen Netzwerken können die dazugehörigen geschichteten Hilfsnetzwerke aus bis zu $n - 1$ Niveaus bestehen. Für die Anzahl der Niveaus in einem geschichteten Hilfsnetzwerk eines 0-1-Netzwerkes kann eine bessere obere Schranke angegeben werden. Das folgende Lemma gilt auch für Netzwerke, die nicht schlicht sind.

Lemma. Es sei N ein 0-1-Netzwerk mit der Eigenschaft, daß es zwischen je zwei Ecken maximal zwei parallele Kanten gibt. Ist f_0 der triviale Fluß auf N und M der Wert eines maximalen Flusses, so besteht das geschichtete Hilfsnetzwerk G'_{f_0} aus maximal $2^{2/3}n/\sqrt{M}$ Niveaus.

Beweis. Es sei d die Anzahl der Niveaus in G'_{f_0} . Für $i = 0, \dots, d-1$ sei X_i die Menge der Ecken e aus G'_{f_0} mit $\text{Niv}(e) \leq i$ und \overline{X}_i die Menge der Ecken e aus G'_{f_0} mit $\text{Niv}(e) > i$. Für alle i ist (X_i, \overline{X}_i) ein Schnitt. Da alle Kapazitäten gleich 0 oder 1 sind, ist die Kapazität von (X_i, \overline{X}_i) gleich der Anzahl k_i der Kanten zwischen den Niveaus i und $i + 1$. Da der Wert jedes Flusses kleiner gleich der Kapazität jedes Schnittes ist, gilt

$$M \leq k_i.$$

Somit liegen in einem der beiden Niveaus i bzw. $i + 1$ mindestens $\sqrt{M/2}$ Ecken. Daraus ergibt sich eine untere Grenze für die Anzahl der Ecken in G'_{f_0} , indem man über alle Niveaus summiert:

$$\sqrt{M/2} \frac{d}{2} \leq \text{Anzahl der Ecken in } G'_{f_0} \leq n.$$

Daraus folgt sofort $d \leq 2^{3/2}n/\sqrt{M}$. ■

Mit Hilfe dieses Ergebnisses kann man nun zeigen, daß die worst case Laufzeit des Algorithmus von Dinic für 0-1-Netzwerke geringer als die für allgemeine Netzwerke ist. Eine weitere Verbesserung wird dadurch erreicht, daß das Verfahren zum Auffinden von blockierenden Flüssen auf die speziellen Eigenschaften von 0-1-Netzwerken abgestimmt wird.

Ein Fluß ist ein blockierender Fluß, falls es keinen Erweiterungsweg gibt, der nur aus Vorwärtskanten besteht. Für 0-1-Netzwerke bedeutet dies, daß ein binärer Fluß blockierend ist, falls es keinen Weg von der Quelle zur Senke gibt, der ausschließlich aus Kanten mit Fluß 0 besteht. Dazu werden Wege von der Quelle zur Senke bestimmt und deren Kanten dann aus dem Netzwerk entfernt. Diese Kanten tragen den Fluß 1. Das folgende Verfahren basiert auf der Tiefensuche. Hierbei wird jede Kante nur einmal betrachtet und anschließend entfernt. Abbildung 6.16 zeigt die rekursive Funktion `finde`, welche Wege von der Quelle zur Senke sucht und alle betrachteten Kanten entfernt.

```

function finde(var G : 0-1-Netzwerk; i : Integer;
                var h : Fluß) : Integer;
begin
    if i hat keinen Nachfolger then
        finde := 0
    else
        if G.senke ist Nachfolger von i then begin
            h[i,G.senke] := 1;
            entferne die Kante (i,G.senke) aus G;
            finde := 1;
        end
        else
            while (es gibt noch einen Nachfolger j von i) do begin
                entferne die Kante (i,j) aus G;
                finde := finde(G,j,h);
                if finde = 1 then begin
                    h[i,j] := 1;
                    break;
                end
            end
    end

```

Abbildung 6.16: Die Funktion finde

Jeder Aufruf von `finde` erweitert den Fluß h , soweit h noch nicht blockierend ist, und entfernt die besuchten Kanten aus G . Abbildung 6.17 zeigt die Prozedur `blockfluß`. Sie ruft solange `finde` auf, bis kein Weg mehr von der Quelle zur Senke existiert. Am Ende ist h ein binärer blockierender Fluß.

Die Komplexität der Prozedur `blockfluß` läßt sich leicht bestimmen. Jede Kante wird maximal einmal betrachtet; somit ergibt sich $O(m)$ als worst case Komplexität. Das im letzten Abschnitt vorgestellte Verfahren zur Bestimmung eines blockierenden Flusses in

```

procedure blockfluß(G : 0-1-Netzwerk; var h : Fluß);
begin
    while finde(G,G.quelle,h) = 1 do
        ;
end

```

Abbildung 6.17: Die Prozedur blockfluß

einem beliebigen Netzwerk hatte dagegen eine Komplexität von $O(n^2)$. Mit Hilfe dieser Vorbereitungen kann nun folgender Satz bewiesen werden.

Satz. Ein binärer maximaler Fluß für 0-1-Netzwerke kann mit dem Algorithmus von Dinic mit Komplexität $O(n^{2/3}m)$ bestimmt werden.

Beweis. Da die Prozedur `blockfluß` die Komplexität $O(m)$ hat, genügt es, zu zeigen, daß maximal $O(n^{2/3})$ blockierende Flüsse bestimmt werden müssen. Dazu wird das oben bewiesene Resultat über die Anzahl der Niveaus in geschichteten Hilfsnetzwerken verwendet. Es sei L die Anzahl der notwendigen Flußerhöhungen und M der Wert eines maximalen Flusses. Da der Fluß jeweils um mindestens 1 erhöht wird, ist $L \leq M$. Ist $M \leq n^{2/3}$, so ist die Behauptung bewiesen. Ist $M > n^{2/3}$, so betrachtet man den Fluß h , dessen Wert nach der Erhöhung mittels eines blockierenden Flusses erstmals $M - n^{2/3}$ übersteigt (d.h. $|h| \leq M - n^{2/3}$). Das geschichtete Hilfsnetzwerk G'_h hat die Eigenschaft, daß es zwischen je zwei Ecken höchstens zwei parallele Kanten gibt. Aus dem obigen Lemma folgt, daß G'_h aus maximal $2^{2/3}n/\sqrt{M'}$ Niveaus besteht. Hierbei ist M' der Wert eines maximalen Flusses auf G'_h . Nach Wahl von h ist $M' \geq n^{2/3}$. Somit sind zur Konstruktion von h maximal $2^{2/3}n/\sqrt{M'} \leq 2^{2/3}n/n^{1/3} = 2^{2/3}n^{2/3}$ Flußerhöhungen notwendig. Da bei jeder weiteren Flußerhöhung der Wert des Flusses um mindestens 1 erhöht wird, sind somit maximal $n^{2/3}$ weitere Erhöhungen notwendig. Somit gilt:

$$L \leq 2^{2/3}n^{2/3} + n^{2/3} \leq 4n^{2/3}$$

Damit ist der Beweis vollständig. ■

Für eine wichtige Klasse von Anwendungen kann man zeigen, daß die Laufzeit sogar durch $O(\sqrt{nm})$ beschränkt ist. Dies wird in dem folgenden Satz bewiesen.

Satz. Es sei G ein 0-1-Netzwerk, so daß $g^-(e) \leq 1$ oder $g^+(e) \leq 1$ für jede Ecke e von G gilt. Dann läßt sich ein binärer maximaler Fluß mit der Komplexität $O(\sqrt{nm})$ bestimmen.

Beweis. Es sei d die Anzahl der Niveaus in G'_{f_0} für den trivialen Fluß f_0 und E_i die Menge der Ecken in $Niv(i)$ mit $1 \leq i < d$. Ferner sei M der Wert eines maximalen Flusses f auf G . Der in die Ecken von E_i eintretende und wieder austretende Fluß ist gleich M . Da nach Voraussetzung $g^-(e) \leq 1$ oder $g^+(e) \geq 1$ für alle Ecken e aus E_i gilt, muß E_i mindestens M Ecken enthalten. Summiert man über alle Niveaus, so erhält

man

$$n > (d - 1)M.$$

Somit gilt $d < n/M + 1$. Man beachte, daß alle geschichteten Hilfsnetzwerke, die im Verlauf des Algorithmus von Dinic gebildet werden, die Voraussetzungen des Satzes ebenfalls erfüllen. Analog zum Beweis des letzten Satzes zeigt man nun, daß maximal $2\sqrt{n} + 1$ Flußerhöhungen notwendig sind. Damit ergibt sich insgesamt eine worst case Komplexität von $O(\sqrt{nm})$. ■

6.6 Kostenminimale Flüsse

In praktischen Anwendungen treten Netzwerke häufig in einer Variante auf, bei der den Kanten neben Kapazitäten auch noch Kosten zugeordnet sind. Es sei G ein Netzwerk mit oberen Kapazitätsbeschränkungen. Jeder Kante k in G sind Kosten $c(k) \geq 0$ zugeordnet. Hierbei sind $c(k)$ die Kosten, die beim Transport einer Flußeinheit durch die Kante k entstehen. Die Kosten eines Flusses f von G sind gleich

$$\sum_{k \in E} f(k)c(k).$$

In der Praxis interessiert man sich für folgende Fragestellung: Unter den Flüssen mit Wert w ist derjenige mit minimalen Kosten gesucht. Einen solchen Fluß nennt man *kostenminimal*. Insbesondere sucht man nach einem maximalen Fluß mit minimalen Kosten. Es sei nun f ein Fluß und W ein Erweiterungsweg für f . Die Kosten von W sind gleich der Summe der Kosten der Vorwärtskanten minus der Summe der Kosten der Rückwärtskanten bezüglich f . Ein geschlossener Weg in G_f heißt *Erweiterungskreis*. Die Kosten eines Erweiterungskreises berechnen sich genauso wie für einen Erweiterungsweg. Die wichtigsten Eigenschaften von kostenminimalen Flüssen sind in folgendem Satz zusammengefaßt.

Satz. Es sei G ein Netzwerk mit oberen Kapazitätsbeschränkungen und einer Bewertung der Kanten mit nicht negativen Kosten. Dann gelten folgende Aussagen:

- a) Ein Fluß f mit Wert w hat genau dann minimale Kosten, wenn der Graph G_f keinen Erweiterungskreis mit negativen Kosten besitzt.
- b) Es sei f ein kostenminimaler Fluß mit Wert w und W ein Erweiterungsweg mit den kleinsten Kosten für f . Der aus f und W gebildete neue Fluß f' ist wieder ein kostenminimaler Fluß und hat den Wert $|f| + f_\Delta$.
- c) Sind alle Kapazitäten ganze Zahlen, so gibt es einen kostenminimalen maximalen Fluß, dessen Werte ganzzahlig sind.

Beweis. a) Erhöht man den Fluß durch die Kanten eines Erweiterungskreises wie in Abschnitt 6.1 beschrieben, so bleibt die Flußerhaltungsbedingung für alle Ecken erfüllt

und der Wert des Flusses ändert sich nicht. Mit Hilfe eines Erweiterungskreises mit negativen Kosten können die Kosten eines Flusses gesenkt werden, ohne daß sich der Wert des Flusses ändert. Somit besitzt ein Fluß mit minimalen Kosten keinen Erweiterungskreis mit negativen Kosten.

Es sei f ein Fluß ohne Erweiterungskreis mit negativen Kosten. Angenommen es gibt einen Fluß f' mit $|f| = |f'|$, dessen Kosten niedriger sind als die von f . Bilde nun ein neues Netzwerk $G_{f'-f}$: Ist $k = (e, f)$ eine Kante in G mit $f'(k) \geq f(k)$, so ist auch k eine Kante in $G_{f'-f}$ mit Bewertung $f'(k) - f(k)$ und Kosten $\text{kosten}(k)$, ist $f'(k) < f(k)$, so ist (f, e) eine Kante in $G_{f'-f}$ mit Bewertung $f(k) - f'(k)$ und Kosten $-\text{kosten}(k)$. Jede Kante von $G_{f'-f}$ ist auch eine Kante von G_f und für jede Ecke von G gilt in $G_{f'-f}$ die Flußerhaltungsbedingung. Da die Kosten von f' niedriger als die von f sind, muß es in $G_{f'-f}$ einen geschlossenen Weg mit negativen Kosten geben. Somit gibt es in G_f einen Erweiterungskreis mit negativen Kosten. Dieser Widerspruch zeigt, daß die Kosten von f minimal sind.

b) Angenommen der neue Fluß f' ist nicht kostenminimal. Nach dem ersten Teil existiert in $G_{f'}$ ein Erweiterungskreis C mit negativen Kosten. Mit Hilfe von C und W wird ein neuer Erweiterungsweg für f konstruiert, dessen Kosten geringer als die von W sind. Dieser Widerspruch zeigt, daß f' kostenminimal ist. Kanten von C , die es in G_f noch nicht gab, sind durch die Flußerhöhung mittels W erst entstanden, d.h. sie müssen in umgekehrter Richtung auf W vorkommen. Für eine solche Kante k gilt: $\text{kosten}(k) < \text{kosten}(C \setminus \{k\})$. Betrachte nun die Menge M bestehend aus den Kanten aus C , die nicht mit gegensätzlicher Richtung in W vorkommen und den Kanten aus W , die nicht mit gegensätzlicher Richtung in C vorkommen. Diese Kanten liegen in G_f und die Gesamtkosten der Kanten aus C sind echt kleiner als die Gesamtkosten der Kanten aus W . In G_f muß es nach Konstruktion einen Weg W' von der Quelle zur Senke geben, der aus Kanten aus M besteht (C ist ein geschlossener Weg). Nach Konstruktion von M gilt $\text{kosten}(W') < \text{kosten}(W)$.

c) Startend mit dem trivialen Fluß wird mit Erweiterungswegen von minimalen Kosten eine Folge von kostenminimalen Flüssen konstruiert (dies folgt aus Teil b)). Da die Kapazitäten ganzzahlig sind, sind auch die einzelnen Erhöhungen ganzzahlig und mindestens gleich 1. Somit erreicht man nach endlich vielen Schritten einen kostenminimalen maximalen Fluß, dessen Werte ganzzahlig sind. ■

Einen Algorithmus mit Laufzeit $O(|f_{max}|nm)$ zur Bestimmung eines maximalen kostenminimalen Flusses wird in Aufgabe 33 in Kapitel 8 auf Seite 288 behandelt. Zum Abschluß dieses Kapitels wird noch das sogenannte *Transportproblem* behandelt.

Gegeben sind w Warenlager, die je a_1, \dots, a_w Exemplare einer und derselben Ware lagern. Weiterhin gibt es k Kunden, welche je n_1, \dots, n_k Exemplare dieser Ware benötigen. Es wird angenommen, daß das Angebot die Nachfrage übersteigt, d.h., es gilt

$$\sum_{i=1}^w a_i \geq \sum_{j=1}^k n_j.$$

Die Kosten für den Transport eines Exemplares der Ware von Warenlager i zu Kunde j betragen $c_{ij} \geq 0$. Unter allen Zuordnungen von Waren zu Kunden, die die Nachfragen

aller Kunden erfüllen, ist die kostengünstigste Zuordnung gesucht. Eine Warenzuordnung wird durch eine Matrix $Z = (z_{ij})$ beschrieben. Hierbei ist $z_{ij} \geq 0$ die Anzahl der Waren, welche aus Warenlager i zu Kunde j geliefert wird. D.h., unter den Nebenbedingungen

$$\sum_{j=1}^w z_{ij} \leq a_i, \quad 1 \leq i \leq k$$

$$\sum_{i=1}^k z_{ij} = n_j, \quad 1 \leq j \leq w$$

ist die Summe

$$\sum_{i=1}^k \sum_{j=1}^w c_{ij} z_{ij}$$

mit $z_{ij} \geq 0$ für $1 \leq i \leq k$, $1 \leq j \leq w$ zu minimieren. Das Transportproblem kann auf die Bestimmung eines kostenminimalen Flusses reduziert werden. Hierzu wird ein Graph G mit $w + k + 2$ Ecken definiert: Neben der Quelle q und der Senke s gibt es für jedes Warenlager und jeden Kunden eine Ecke. Es gibt folgende Kanten in G :

- Eine Kante mit Kapazität a_i und Kosten 0 von der Quelle zum i -ten Warenlager.
- Eine Kante k_{ij} von Warenlager i zu Kunde j ohne Kapazitätsbeschränkung und Kosten c_{ij} .
- Eine Kante mit Kapazität n_j und Kosten 0 vom j -ten Kunden zur Senke.

Es ist sofort ersichtlich, daß die kostenminimalen maximalen Flüsse von G den kostenminimalen Zuordnungen des Transportproblems entsprechen. Es sei f ein kostenminimaler maximaler Fluß von G . Aus der Konstruktion des Graphen folgt $|f| = \sum_{j=1}^k n_j$. Dann ist $z_{ij} = f(k_{ij})$ eine kostengünstigste Zuordnung, umgekehrt lässt sich aus einer kostengünstigsten Zuordnung ein kostenminimaler maximaler Fluß herleiten. Interessanterweise kann auch die Bestimmung eines kostenminimalen Flusses auf ein Transportproblem reduziert werden.

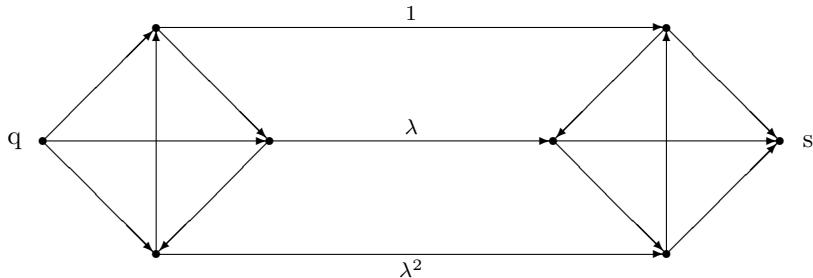
6.7 Literatur

Die Grundlagen von Netzwerken und Flüssen sind in dem Buch von Ford und Fulkerson [43] beschrieben. Dort findet man auch viele Anwendungen von Netzwerken. Das Konzept der Erweiterungswege stammt auch von Ford und Fulkerson. Edmonds und Karp haben bewiesen, daß bei Verwendung von kürzesten Erweiterungswegen maximal $nm/2$ Iterationen notwendig sind [35]. Das Konzept der blockierenden Flüsse und ein darauf aufbauender Algorithmus mit Laufzeit $O(n^2m)$ stammt von Dinic [31]. Den Algorithmus zur Bestimmung von blockierenden Flüssen mit Aufwand $O(n^2)$ findet man in [91]. Ein

Verfahren zur Bestimmung von maximalen Flüssen, welches vollkommen unabhängig von dem Konzept des Erweiterungsweges ist, wurde 1985 von Goldberg entwickelt. Der sogenannte *Preflow-Push-Algorithmus* hat eine Laufzeit von $O(n^3)$ [48]. Mittels spezieller Datenstrukturen erreicht man sogar eine Laufzeit von $O(mn \log(n^2/m))$ [49]. Dies ist der zur Zeit schnellste Algorithmus zur Bestimmung von maximalen Flüssen. Cherkassky und Goldberg haben einen experimentellen Vergleich von Algorithmen zur Bestimmung von maximalen Flüssen erstellt [21]. Die verwendeten Netzwerke sind in [74] beschrieben und können auch in Form von Dateien bezogen werden. Ergebnisse über 0-1-Netzwerke findet man in [37]. Einen guten Überblick über Netzwerkalgorithmen enthält [50]. Aufgabe 2 stammt aus [83].

6.8 Aufgaben

1. Es sei G ein gerichteter kantenbewerteter Graph und q, s Ecken von G . Ferner sei H die Menge der Kanten von G mit Anfangsseite s oder Endseite q . Entfernen Sie aus G die Kanten, welche in H liegen, und bezeichnen Sie diesen Graphen mit G' . Beweisen Sie, daß die Werte von maximalen q - s -Flüssen auf G und G' übereinstimmen.
- * 2. Es sei $\lambda > 0$ die reelle Zahl, für die $\lambda^2 + \lambda - 1 = 0$ gilt. Betrachten Sie das folgende Netzwerk G . Alle unmarkierten Kanten haben die Kapazität $\lambda + 2$. Zeigen Sie, daß es einen Fluß mit Wert 2 gibt und daß dieser Wert maximal ist. Geben Sie ferner eine Folge von Erweiterungswegen W_i von G an, so daß die durch W_i gewonnenen Flüsse gegen einen Fluß mit Wert 2 konvergieren, ohne diesen Wert jedoch zu erreichen.



Was passiert, wenn noch eine zusätzliche Kante von q nach s mit Kapazität 1 eingefügt und die gleiche Folge von Erweiterungswegen verwendet wird?

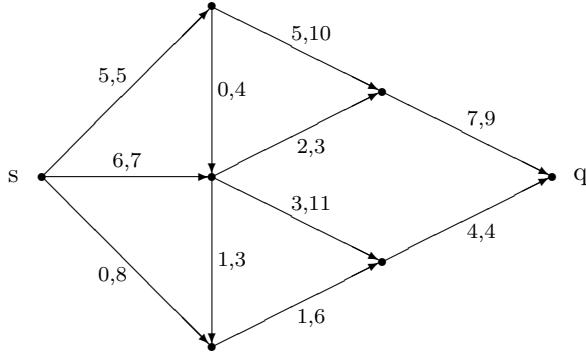
(Hinweis: Beweisen Sie zunächst folgende Gleichungen:

$$\lambda + 2 = \frac{1}{1 - \lambda} = \sum_{i=0}^{\infty} \lambda^i \quad \text{und} \quad \lambda^{i+2} = \lambda^i - \lambda^{i+1} \quad \text{für alle } i \geq 0.$$

Wählen Sie W_0 so, daß von den drei mittleren waagrechten Kanten nur die obere verwendet wird. Die übrigen Erweiterungswege W_i werden nun so gewählt, daß sie

jeweils alle drei mittleren waagrechten Kanten enthalten und den Fluß um jeweils λ^{i+1} erhöhen.)

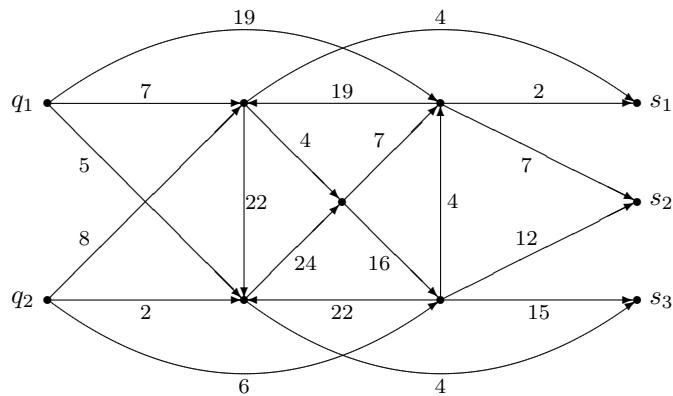
3. Betrachten Sie das folgende Netzwerk G mit dem Fluß f . Die Bewertungen der Kanten geben den Fluß und die Kapazität an.



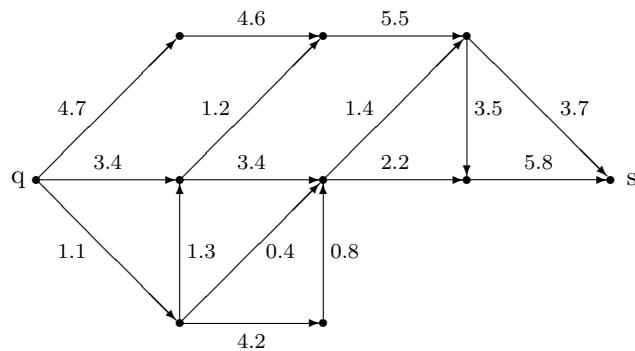
- a) Bestimmen Sie den Wert von f .
- b) Finden Sie einen Erweiterungsweg für f , und bestimmen Sie den erhöhten Fluß.
- c) Bestimmen Sie einen Schnitt mit minimaler Kapazität.
- d) Geben Sie einen maximalen Fluß an.
4. Ändern Sie in dem Netzwerk G aus Abbildung 6.1 auf Seite 166 die Kapazität der Kante $k = (3, s)$ auf 20. Wenden Sie den Algorithmus von Dinic auf dieses Netzwerk an, und bestimmen Sie einen maximalen Fluß. Wie oft muß die Prozedur **blockfluss** aufgerufen werden? Vergleichen Sie das Ergebnis mit der Anwendung des gleichen Algorithmus auf das ursprüngliche Netzwerk.
5. Die in Abschnitt 6.1 angegebene Definition eines Netzwerkes kann leicht verallgemeinert werden, indem mehrere Quellen q_1, \dots, q_r und mehrere Senken s_1, \dots, s_t zugelassen werden. Für diese Ecken gilt dann $g^-(q_i) = g^+(s_j) = 0$. Die Flußberhaltungsbedingung gilt in diesem Fall nur für Ecken, die weder Quellen noch Senken sind. Der Wert eines Flusses f auf einem solchen Netzwerk ist

$$|f| = \sum_{\substack{k=(q_i, e) \in K \\ 1 \leq i \leq r}} f(k),$$

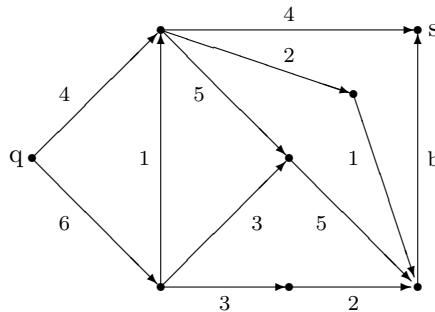
wobei K die Kantenmenge des Netzwerkes ist. Die Bestimmung eines maximalen Flusses auf einem solchen Netzwerk läßt sich leicht auf q-s-Netzwerke zurückführen. Dazu erweitert man das Netzwerk um eine Ecke e_Q (die fiktive Quelle) sowie Kanten (e_Q, q_i) für $1 \leq i \leq r$. Ferner wird noch eine Ecke e_S (die fiktive Senke) mit den entsprechenden Kanten hinzugefügt. Welche Kapazitäten müssen die neuen Kanten tragen? Betrachten Sie das folgende Netzwerk mit den Quellen q_1, q_2 und den Senken s_1, s_2 , und bestimmen Sie einen maximalen Fluß.



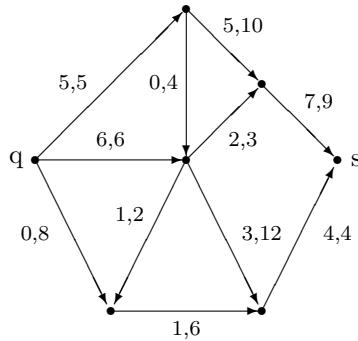
- * 6. Es sei G ein q - s -Netzwerk und $(X_1, \bar{X}_1), (X_2, \bar{X}_2)$ q - s -Schnitte von G mit minimalen Kapazitäten. Beweisen Sie, daß dann auch $(X_1 \cap X_2, \bar{X}_1 \cap \bar{X}_2)$ und $(X_1 \cup X_2, \bar{X}_1 \cup \bar{X}_2)$ q - s -Schnitte mit minimalen Kapazitäten sind.
- 7. Kann es in einem Netzwerk mit maximalem Fluß f eine Kante k geben, so daß $f(k) > |f|$ ist?
- 8. Finden Sie mit Hilfe des Algorithmus von Dinic einen maximalen Fluß für das folgende Netzwerk. Wie viele Erhöhungen durch blockierende Flüsse sind dabei notwendig?



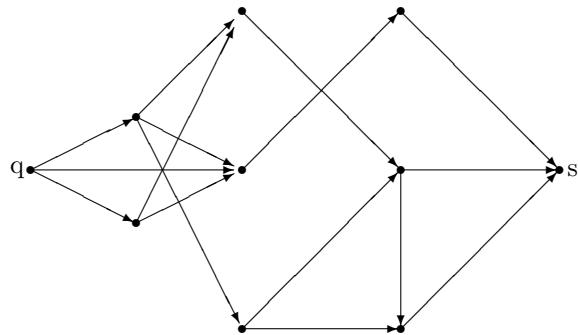
- 9. Finden Sie in dem folgenden Netzwerk einen blockierenden Fluß mit Hilfe der Prozedur `blockfluß` in Abhängigkeit des Wertes von b . Für welche Werte von b ist der Fluß maximal?



10. Beweisen Sie, daß es in jedem Netzwerk eine Folge von höchstens m Erweiterungswegen gibt, welche zu einem maximalen Fluß führen. (Hinweis: Gehen Sie von einem maximalen Fluß aus und erniedrigen Sie diesen so lange durch Erweiterungswege, bis der triviale Fluß vorliegt.) Führt diese Erkenntnis zu einem neuen Algorithmus?
- * 11. Es sei G ein Netzwerk, dessen Kapazitäten ganzzahlig sind. Ein Algorithmus zur Bestimmung eines maximalen Flusses verfährt folgendermaßen: Startend mit dem trivialen Fluß f wird jeweils unter allen Erweiterungswegen derjenige mit maximalem f_Δ ausgewählt. Beweisen Sie, daß nach $O(m \log |f_{max}|)$ Schritten ein maximaler Fluss f_{max} gefunden wird. (Hinweis: Zeigen Sie zunächst, daß es für jeden Fluß f einen Erweiterungsweg mit $f_\Delta \geq (|f_{max}| - |f|)/m$ gibt. Daraus folgt dann, daß nach l Schritten ein Fluß vorliegt, dessen Wert mindestens $|f_{max}|((m-1)/m)^l$ ist. Die Aussage kann nun unter Verwendung der Ungleichung $m(\log m - \log(m-1)) \geq 1$ bewiesen werden.)
12. In einem Netzwerk wird die Kapazität jeder Kante um einen konstanten Wert C erhöht. Um wieviel erhöht sich dann der Wert eines maximalen Flusses? Wie erhöht sich der maximale Fluß, wenn die Kapazität jeder Kante um einen konstanten Faktor C erhöht wird?
13. Beweisen Sie, daß der am Ende von Abschnitt 6.1 diskutierte Algorithmus zur Bestimmung eines maximalen Flusses terminiert, sofern die Kapazitäten der Kanten rationale Zahlen sind.
14. Aus einem ein q - s -Netzwerk G wird eine neuen Netzwerk gebildet, in dem die Richtungen aller Kanten umgedreht und die Kapazitäten beibehalten werden. Das neue Netzwerk hat s als Quelle und q als Senke. Beweisen Sie, daß der Wert eines maximalen Flusses auf beiden Netzwerken gleich ist.
15. Betrachten Sie das folgende Netzwerk mit Fluß f .
- Zeigen Sie, daß f ein blockierender Fluß ist.
 - Konstruieren Sie G_f und G'_f .
 - Finden Sie einen blockierenden Fluß h auf G'_f und erhöhen Sie f damit. Ist der entstehende Fluß maximal?



16. Es sei G ein Netzwerk und f_1, f_2 Flüsse auf G . Für jede Kante k von G setze $f(k) = f_1(k) + f_2(k)$. Unter welcher Voraussetzung ist f ein Fluß auf G ?
17. Es sei G ein Netzwerk, dessen Kapazitäten ganzzahlig sind, und es sei f ein maximaler Fluß auf G . Die Kapazität einer Kante wird um 1 erhöht. Entwerfen Sie einen Algorithmus, welcher in linearer Zeit $O(n + m)$ einen neuen maximalen Fluß bestimmt. Lösen Sie das gleiche Problem für den Fall, daß die Kapazität einer Kante um 1 erniedrigt wird.
- * 18. Es sei G ein Netzwerk und f ein maximaler Fluß. Es sei e eine Ecke von $G \setminus \{q, s\}$ und f_e der Fluß durch die Ecke e . Geben Sie einen Algorithmus an, welcher einen Fluß f' für das Netzwerk $G \setminus \{e\}$ bestimmt, dessen Wert mindestens $|f| - f_e$ ist. Können die Prozeduren **erweitere vorwärts** und **erweitererückwärts** dazu verwendet werden?
19. Es sei G ein Netzwerk und f_1, f_2 Flüsse auf G . Für $\alpha \in [0, 1]$ setze
- $$f_\alpha(k) = \alpha f_1(k) + (1 - \alpha) f_2(k)$$
- für alle Kanten k von G . Zeigen Sie, daß f_α ein Fluß auf G ist, und bestimmen Sie den Wert von f_α .
20. Es sei G ein Netzwerk und f ein maximaler Fluß mit $|f| > 0$. Beweisen Sie, daß es in G eine Kante k gibt, so daß der Wert jedes Flusses auf dem Netzwerk $G' = G \setminus \{k\}$ echt kleiner als $|f|$ ist.
21. Bestimmen Sie einen maximalen Fluß mit Hilfe des Algorithmus von Dinic für das folgende 0-1-Netzwerk, in dem alle Kanten die Kapazität 1 haben.

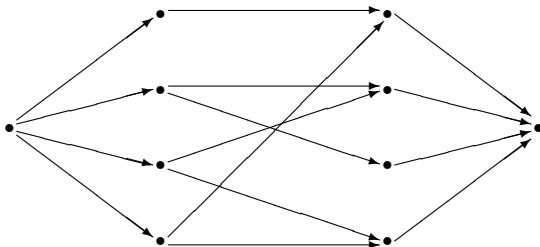


22. Es sei G ein 0-1-Netzwerk mit m Kanten. Beweisen Sie folgende Aussagen:

- Ist M der Wert eines maximalen Flusses und f_0 der triviale Fluß auf G , so besteht das geschichtete Hilfsnetzwerk G'_{f_0} aus maximal m/M Niveaus.
- Der Algorithmus von Dinic bestimmt mit Komplexität $O(m^{3/2})$ einen binären maximalen Fluß für G .

Kapitel 7

Anwendungen von Netzwerkalgorithmen



Viele kombinatorische Probleme lassen sich auf die Bestimmung eines maximalen Flusses auf einem geeigneten Netzwerk zurückführen. Die in diesem Kapitel behandelten Anwendungen zeigen, daß die Netzwerktheorie ein mächtiges Werkzeug zur Lösung von Problemen ist, welche auf den ersten Blick nichts mit Netzwerken zu tun haben. Der erste Schritt besteht aus der Definition eines äquivalenten Netzwerkproblems. Danach können die im letzten Kapitel diskutierten Algorithmen verwendet werden. Mittels einer Rücktransformation kommt man dann zur Lösung des Ausgangsproblems. Im ersten Abschnitt dieses Kapitels wird die Bestimmung von maximalen Zuordnungen in bipartiten Graphen diskutiert. Im zweiten Abschnitt werden Netzwerke mit unteren und oberen Kapazitätsgrenzen betrachtet. Danach stehen Algorithmen zur Bestimmung der Kanten- und Eckenanzahl eines ungerichteten Graphen im Mittelpunkt. Im letzten Abschnitt wird ein Algorithmus zur Bestimmung eines minimalen Schnittes vorgestellt.

7.1 Maximale Zuordnungen

Zur Motivation der in diesem Abschnitt behandelten Probleme wird zunächst ein Beispiel aus dem Bereich *Operations Research* vorgestellt. In einer Produktionsanlage gibt

es n Maschinen M_1, \dots, M_n , welche von m Arbeitern A_1, \dots, A_m bedient werden können. Ein Arbeiter kann nicht gleichzeitig mehrere Maschinen bedienen, und nicht jeder Arbeiter ist ausgebildet, jede der n Maschinen zu bedienen. Um die Maschinen optimal auszunutzen, ist eine Zuordnung von Maschinen und Arbeitern gesucht, bei der möglichst viele Maschinen bedient werden. Dieses Problem kann durch einen ungerichteten bipartiten Graphen mit $n+m$ Ecken dargestellt werden: Die Ecken sind mit M_1, \dots, M_n und A_1, \dots, A_m markiert, und zwischen A_i und M_j gibt es eine Kante, falls der Arbeiter A_i für die Bedienung der Maschine M_j ausgebildet wurde. Abbildung 7.1 zeigt ein Beispiel für einen solchen Graphen.

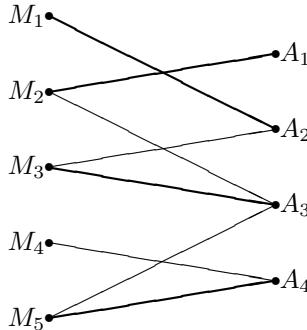


Abbildung 7.1: Ein Graph für das Arbeiter-Maschinen-Zuordnungsproblem

Die fett gezeichneten Kanten zeigen eine Zuordnung, bei der die maximale Anzahl von Maschinen bedient wird. In diesem Abschnitt wird gezeigt, wie sich solche Zuordnungsprobleme mit Hilfe von Netzwerkalgorithmen lösen lassen.

Zunächst werden einige Begriffe eingeführt. Es sei G ein ungerichteter Graph mit Kantenmenge K . Eine Teilmenge Z von K heißt *Zuordnung* (*Matching*) von G , falls die Kanten in Z paarweise keine gemeinsamen Ecken haben. Eine Zuordnung heißt *maximal*, wenn es keine Zuordnung mit mehr Kanten gibt. Die Maximalität von Zuordnungen stützt sich nicht auf die Maximalität von Mengen bezüglich Mengeninklusion, sondern auf die Maximalität der Anzahl der Elemente von Mengen. Eine Zuordnung Z eines Graphen G heißt *nicht erweiterbar*, wenn sie durch keine weitere Kante vergrößert werden kann. In diesem Fall gibt es zu jeder Kante von G eine Kante aus Z , so daß beide Kanten mindestens eine Ecke gemeinsam haben. Nicht erweiterbare Zuordnungen können mit Hilfe eines Greedy-Algorithmus in linearer Zeit bestimmt werden. Die Bestimmung von maximalen Zuordnungen ist dagegen aufwendiger. Abbildung 7.2 zeigt eine nicht erweiterbare Zuordnung (fett gezeichnete Kanten). Diese Zuordnung ist nicht maximal, denn es gibt eine Zuordnung mit drei Kanten.

Eine Zuordnung Z heißt *vollständig*, falls jede Ecke des Graphen mit einer Kante aus Z inkident ist. Jede vollständige Zuordnung ist maximal, aber die Umkehrung gilt nicht. Abbildung 7.3 zeigt einen Graphen mit einer maximalen Zuordnung, die nicht vollständig ist.

Im weiteren beschränken wir uns auf den für viele Anwendungen wichtigen Fall von bipartiten Graphen. Die Eckenmenge E ist dabei immer in die disjunkten Mengen E_1

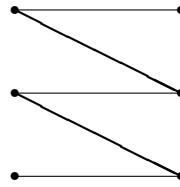


Abbildung 7.2: Eine nicht erweiterbare Zuordnung

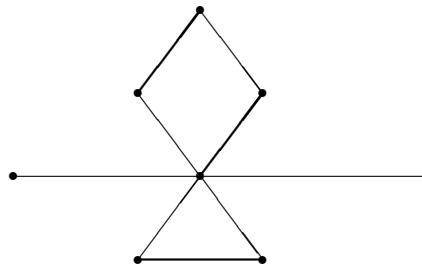
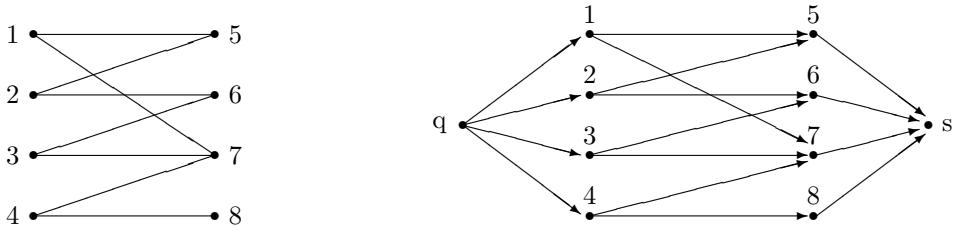


Abbildung 7.3: Eine nicht vollständige maximale Zuordnung

und E_2 aufgeteilt, so daß Anfangs- und Endecke jeder Kante in verschiedenen Mengen liegen. Die Bestimmung einer maximalen Zuordnung erfolgt mittels eines Netzwerkes.

Es sei G ein bipartiter Graph und N_G folgendes 0-1-Netzwerk: Die Eckenmenge E_G von N_G ist gleich $E \cup \{q, s\}$, d.h. es werden zwei neue Ecken eingeführt. Für jede Ecke $e \in E_1$ gibt es in N_G eine Kante von q nach e mit Kapazität 1 und für jede Ecke $e \in E_2$ eine Kante von e nach s mit Kapazität 1. Ferner gibt es für jede Kante (e_1, e_2) von G eine gerichtete Kante von e_1 nach e_2 mit Kapazität 1 ($e_1 \in E_1, e_2 \in E_2$). Abbildung 7.4 zeigt einen bipartiten Graphen G und das dazugehörige Netzwerk N_G .

Abbildung 7.4: Ein bipartiter Graph G und das zugehörige Netzwerk N_G

Der Zusammenhang zwischen einer maximalen Zuordnung eines bipartiten Graphen G und einem maximalen binären Fluß auf N_G wird in folgendem Lemma bewiesen.

Lemma. Die Anzahl der Kanten in einer maximalen Zuordnung eines bipartiten Gra-

phen G ist gleich dem Wert eines maximalen Flusses auf N_G . Ist f ein maximaler binärer Fluß, so bilden die Kanten aus G mit Fluß 1 eine maximale Zuordnung von G .

Beweis. Es sei f ein maximaler binärer Fluß auf dem 0-1-Netzwerk N_G . Ferner sei Z die Menge aller Kanten aus G , für die der Fluß durch die entsprechende Kante in N_G gerade 1 ist. Da jede Ecke aus E_1 in N_G den Eingrad 1 und jede Ecke aus E_2 in N_G den Ausgrad 1 hat, folgt aus der Flußerhaltungsbedingung, daß Z eine Zuordnung ist. Ferner enthält Z genau $|f|$ Kanten.

Sei nun umgekehrt Z eine maximale Zuordnung von G mit z Kanten. Dann läßt sich leicht ein Fluß f mit Wert z auf N_G konstruieren. Für jede Kante $(e_1, e_2) \in Z$ definiert man

$$f(q, e_1) = f(e_1, e_2) = f(e_2, q) = 1$$

und $f(k) = 0$ für alle anderen Kanten k von N_G . Die Flußerhaltungsbedingung ist für f erfüllt, und es gilt $|f| = z$. Damit ist das Lemma bewiesen. ■

Die Ergebnisse aus Kapitel 6 über 0-1-Netzwerke führen zu einem Algorithmus zur Bestimmung einer maximalen Zuordnung eines bipartiten Graphen mit Laufzeit $O(\sqrt{nm})$. Unter Ausnutzung der speziellen Struktur des Netzwerkes N_G ergibt sich sogar folgender Satz.

Satz. Für einen bipartiten Graphen kann eine maximale Zuordnung in der Zeit $O(\sqrt{z}m)$ mittels des Algorithmus von Dinic bestimmt werden. Hierbei bezeichnet z die Anzahl der Kanten in einer maximalen Zuordnung.

Beweis. Es genügt, die Aussage für einen zusammenhängenden bipartiten Graphen zu beweisen. Es sei N_G das zu G gehörende 0-1-Netzwerk und f ein maximaler binärer Fluß auf N_G . Nach dem letzten Lemma gilt $|f| = z$. Zunächst wird ein beliebiger binärer Fluß h auf N_G betrachtet. Es sei N'_G das zu N_G und h gehörende geschichtete Hilfsnetzwerk. Ferner sei W ein Weg von q nach s in N'_G . Die Konstruktion von N_G bedingt, daß die Anzahl der Kanten in W ungerade und mindestens drei ist, somit ist die Anzahl der Niveaus in N'_G gerade. Ferner sind die Kanten von W bis auf die erste und letzte Kante abwechselnd Vorwärts- bzw. Rückwärtskanten. In N'_G gibt es nach den Ergebnissen aus Abschnitt 6.5 genau $z - |h|$ Wege von q nach s , welche paarweise keine gemeinsame Kante haben. Da es maximal $|h|$ Rückwärtskanten gibt, muß es einen Weg von q nach s geben, der maximal $\lfloor |h|/(z - |h|) \rfloor$ Rückwärtskanten enthält. Für die Anzahl der Niveaus d in N'_G ergibt sich folgende Abschätzung

$$d \leq 2 \left\lfloor \frac{|h|}{z - |h|} \right\rfloor + 3.$$

Im Rest des Beweise wird gezeigt, daß der Algorithmus von Dinic maximal $2\lfloor\sqrt{z}\rfloor + 2$ blockierende Flüsse bestimmt. Hieraus ergibt sich dann direkt die Aussage des Satzes.

Es sei h der Fluß, dessen Wert nach der Erhöhung mittels eines blockierenden Flusses erstmals über $r = \lfloor z - \sqrt{z} \rfloor$ liegt. Das zugehörige geschichtete Hilfsnetzwerk besteht aus

maximal $2\lfloor r/(z-r) \rfloor + 3$ Niveaus. Aus dem ersten Teil des Beweises folgt, daß bis zur Konstruktion von h maximal $\lfloor r/(z-r) \rfloor + 1$ Fluß erhöhungen notwendig waren. Man beachte dazu, daß die Niveaus der geschichteten Hilfsnetzwerke sich in jedem Schritt erhöhen und immer gerade sind. Da bei jeder weiteren Fluß erhöhung der Wert des Flusses um mindestens 1 erhöht wird, sind somit maximal $\lfloor \sqrt{z} \rfloor + 1$ weitere Erhöhungen notwendig. Wegen

$$\lfloor r/(z-r) \rfloor + 1 = \left\lfloor \frac{\lfloor z - \sqrt{z} \rfloor}{\lfloor \sqrt{z} \rfloor} \right\rfloor + 1 \leq \lfloor \sqrt{z} \rfloor + 1$$

sind insgesamt $2(\lfloor \sqrt{z} \rfloor + 1)$ Fluß erhöhungen notwendig. ■

Maximale Zuordnungen für Bäume können in linearer Zeit bestimmt werden (vergleichen Sie Aufgabe 31). Verfahren zur Bestimmung von maximalen Zuordnungen in nicht bipartiten Graphen sind komplizierter. Die besten Algorithmen haben erstaunlicherweise die gleiche worst case Laufzeit wie im bipartiten Fall. Dabei wird das Konzept der Erweiterungswege auf dem Netzwerk N_G in ein entsprechendes Konzept für G übertragen (vergleichen Sie hierzu Aufgabe 1). Das größte Problem dabei ist die Bestimmung von Erweiterungswegen in nicht bipartiten Graphen.

Mit Hilfe der Sätze von Ford und Fulkerson läßt sich sich ein Kriterium für die Existenz von vollständigen Zuordnungen in bipartiten Graphen angeben.

Satz (P. HALL). Es sei G ein bipartiter Graph mit Eckenmenge $E = E_1 \cup E_2$. G hat genau dann eine Zuordnung Z mit $|Z| = |E_1|$, wenn für jede Teilmenge T von E_1 gilt: $|N(T)| \geq |T|$.

Beweis. Es sei Z eine Zuordnung von G mit $|Z| = |E_1|$ und $T \subseteq E_1$. Dann ist jede Ecke $v \in T$ zu genau einer Kante $(v, w) \in Z$ inzident. Somit ist $w \in N(T)$, und es folgt sofort $|N(T)| \geq |T|$.

Sei nun umgekehrt $|N(T)| \geq |T|$ für jede Teilmenge T von E_1 . Man betrachte das zugehörige Netzwerk N_G und einen maximalen binären Fluß f . Es sei X die Menge aller Ecken aus N_G , welche bezüglich f durch Erweiterungswege von q aus erreichbar sind. Dann gilt $|f| = \kappa(X, \overline{X})$. Es sei $v \in X \cap E_1$ und $k = (v, w)$ eine Kante in N_G . Angenommen $w \notin X$. Dann ist $f(k) = 1$, sonst wäre $w \in X$. Da $g^-(v) = 1$ ist, ist der Fluß durch die Kante von q nach v gleich 1. Somit wurde v über eine Rückwärtskante erreicht. Das bedeutet aber, daß es zwei Kanten mit Anfangscke v gibt, deren Fluß jeweils 1 ist. Dies widerspricht der Flußbehaltungsbedingung. Somit ist $w \in X$, und es gilt $N(X \cap E_1) \subseteq X$. Ist $k = (e_1, e_2)$ eine Kante mit $e_1 \in X$ und $e_2 \in \overline{X}$, so ist entweder $e_1 = q$ oder $e_2 = s$. Ferner gilt $X \cap E_2 = N(X \cap E_1)$. Daraus ergibt sich

$$|f| = \kappa(X, \overline{X}) = |E_1 \setminus X| + |N(X \cap E_1)| \geq |E_1 \setminus X| + |X \cap E_1| = |E_1| \geq |f|,$$

da $|N(X \cap E_1)| \geq |X \cap E_1|$ nach Voraussetzung gilt. Aus dem letzten Lemma ergibt sich nun, daß G eine Zuordnung mit $|E_1|$ Kanten hat. ■

In Aufgabe 34 wird eine Verallgemeinerung des Satzes von Hall bewiesen. Der Satz von Hall führt zu keinem effizienten Algorithmus, der feststellt, ob ein bipartiter Graph eine

vollständige Zuordnung besitzt. Der Grund hierfür ist die große Anzahl der Teilmengen von E_1 , welche man betrachten müßte. Ist die Anzahl der Ecken von E_1 gleich l , so hat E_1 genau 2^l verschiedene Teilmengen. Für reguläre bipartite Graphen gilt folgender Satz:

Satz. Ein regulärer bipartiter Graph G besitzt eine vollständige Zuordnung.

Beweis. Es sei $E = E_1 \cup E_2$ die Eckenmenge von G und $T \subseteq E_1$. Jede Ecke von G habe den Eckengrad d . Sei K_1 die Menge der Kanten von G , welche zu einer Ecke aus T inzident sind, und K_2 die Menge der Kanten von G , welche zu einer Ecke aus $N(T)$ inzident sind. Es gilt $K_1 \subseteq K_2$ und somit

$$d \cdot |T| = |K_1| \leq |K_2| = d \cdot |N(T)|.$$

Also gilt $|N(T)| \geq |T|$. Auf die gleiche Art zeigt man, daß auch für alle Teilmengen T von E_2 diese Ungleichung gilt. Die Behauptung folgt nun aus dem Satz von Hall. ■

Die Aussage des letzten Satzes kann noch verallgemeinert werden.

Satz. Jeder bipartite Graph G enthält eine Zuordnung, welche alle Ecken mit maximalem Grad zuordnet.

Beweis. Es sei $E = E_1 \cup E_2$ die Eckenmenge von G und U_i die Menge der Ecken aus E_i mit maximalem Grad. Ferner sei G_i der von U_i induzierte Untergraph. Nach dem Satz von Hall gibt es in G_i eine Zuordnung Z_i , welche alle Ecken von U_i zuordnet. Es sei $Z = Z_1 \cap Z_2$. Der von $(Z_1 \cup Z_2) \setminus Z$ induzierte Untergraph Z^* besteht aus disjunkten Kreisen und Pfaden auf den sich die Kanten von Z_1 und Z_2 abwechseln. Jeder Kreis C aus Z^* besteht aus einer geraden Zahl von Kanten. Für jeden solchen Kreis füge die Kanten aus $C \cap Z_1$ in Z ein. Genau eine Endecke jedes Pfades P aus Z^* hat den maximalen Eckengrad. Liegt diese Ecke in E_1 so füge die Kanten aus $P \cap Z_1$ in Z , andernfalls füge die Kanten aus $P \cap Z_2$ in Z ein. Nun ist Z eine Zuordnung von G , welche alle Ecken mit maximalem Grad zuordnet. ■

Folgerung. Die Kanten eines bipartiten Graphen G sind die Vereinigung von $\Delta(G)$ Zuordnungen von G .

Beweis. Nach dem letzten Satz gibt es eine Zuordnung Z von G , welche alle Ecken mit maximalem Grad zuordnet. Entfernt man aus G die in Z liegenden Kanten, so erhält man einen Graphen G' mit $\Delta(G') = \Delta(G) - 1$. Der Beweis der Folgerung wird mittels vollständiger Induktion nach dem maximalen Eckengrad geführt. ■

Zum Abschluß dieses Abschnittes wird noch einmal das zu Beginn betrachtete Beispiel aus dem Gebiet des Operations Research aufgegriffen und leicht verändert. Eine Kante zwischen einem Arbeiter und einer Maschine bedeutet, daß der Arbeiter die damit verbundene Arbeit ausführen muß. Ferner muß jeder Arbeiter alle ihm zugewiesenen Arbeiten in einer beliebigen Reihenfolge durchführen. Ein Arbeiter kann nicht mehrere

Maschinen parallel bedienen, und eine Maschine kann nicht gleichzeitig von mehreren Arbeitern bedient werden. Es wird angenommen, daß alle Arbeiten in der gleichen Zeit T zu bewältigen sind. Die Aussage der Folgerung läßt sich dann so interpretieren: Alle Arbeiten zusammen können in der Zeit $\Delta(G) T$ durchgeführt werden, hierbei ist G der zugehörige bipartite Graph.

7.2 Netzwerke mit oberen und unteren Kapazitäten

Bisher wurden nur Netzwerke betrachtet, in denen der Fluß durch jede Kante nur nach oben beschränkt war. Eine explizite untere Grenze wurde nicht angegeben. Es wurde lediglich verlangt, daß der Fluß nicht negativ ist, d.h. 0 war die untere Grenze für alle Kanten. In vielen Anwendungen sind aber von 0 verschiedene Untergrenzen von Bedeutung. In diesem Abschnitt werden Netzwerke mit oberen und unteren Grenzen für den Fluß durch die Kanten betrachtet. Dazu werden zwei Kapazitätsfunktionen κ_u und κ_o für ein Netzwerk angegeben. Für alle Kanten k des Netzwerkes gilt $\kappa_u(k) \leq \kappa_o(k)$. Der erste Teil der Definition eines Flusses wird folgendermaßen abgeändert:

a) für jede Kante k von G gilt $\kappa_u(k) \leq f(k) \leq \kappa_o(k)$.

Der zweite Teil der Definition bleibt unverändert. Ziel dieses Abschnitts ist die Entwicklung eines Algorithmus zur Bestimmung eines maximalen Flusses in Netzwerken mit oberen und unteren Grenzen. Betrachtet man noch einmal die im letzten Kapitel diskutierten Algorithmen, so haben diese eine Gemeinsamkeit: Ein gegebener Fluß wird schrittweise erhöht, bis er maximal ist. Ausgangspunkt war dabei meist der triviale Fluß. Auf einem Netzwerk mit von 0 verschiedenen Untergrenzen ist der triviale Fluß aber kein zulässiger Fluß. Das erste Problem ist also die Bestimmung eines zulässigen Flusses. Die Anpassung der Algorithmen, um aus einem zulässigen einen maximalen Fluß zu erzeugen, sind leicht vorzunehmen.

Abbildung 7.5 zeigt ein Netzwerk mit unteren und oberen Kapazitätsgrenzen. Die Werte sind dabei durch ein Komma getrennt. Man stellt leicht fest, daß es für dieses Netzwerk überhaupt keinen zulässigen Fluß gibt (aus der Quelle können maximal fünf Einheiten hinausfließen, und in die Senke müssen mindestens sechs Einheiten hineinfließen); d.h. es ist im allgemeinen nicht sicher, daß ein zulässiger Fluß überhaupt existiert.

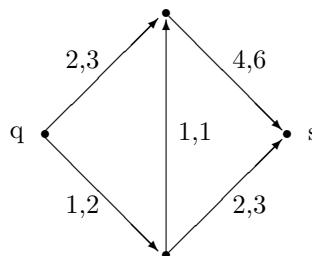


Abbildung 7.5: Ein Netzwerk ohne zulässigen Fluß

Die Bestimmung eines maximalen Flusses auf Netzwerken mit unteren und oberen Grenzen erfolgt in zwei Phasen:

- 1) Überprüfung, ob ein zulässiger Fluß existiert und desselben Bestimmung
- 2) Erhöhung dieses Flusses zu einem maximalen Fluß

L.R. Ford und D.R. Fulkerson haben eine Methode entwickelt, mit der das Problem der ersten Phase auf ein Netzwerkproblem ohne untere Kapazitätsgrenzen zurückgeführt werden kann. Dazu wird ein Hilfsnetzwerk \bar{G} konstruiert.

Es sei G ein Netzwerk mit Kantenmenge K und den Kapazitätsfunktionen κ_u und κ_o . Dann ist \bar{G} ein Netzwerk mit Eckenmenge $\bar{E} = E \cup \{\bar{q}, \bar{s}\}$, wobei E die Eckenmenge von G ist. \bar{q} ist die Quelle und \bar{s} die Senke von \bar{G} . \bar{G} hat die folgenden Kanten mit Kapazität $\bar{\kappa}$:

- a) Für jede Ecke e von G eine Kante $k_e = (e, \bar{s})$ mit Kapazität

$$\bar{\kappa}(k_e) = \sum_{k=(e,w) \in K} \kappa_u(k),$$

d.h. die Kapazität der neuen Kante k_e ist gleich der Summe der unteren Kapazitäten κ_u der Kanten aus G mit Anfangsdecke e . Dies ist genau die Menge, die mindestens aus e hinausfließen muß.

- b) Für jede Ecke e von G eine Kante $k^e = (\bar{q}, e)$ mit Kapazität

$$\bar{\kappa}(k^e) = \sum_{k=(w,e) \in K} \kappa_u(k),$$

d.h. die Kapazität der neuen Kante k^e ist gleich der Summe der unteren Kapazitäten κ_u der Kanten aus G mit Enddecke e . Dies ist genau die Menge, die mindestens in e hineinströmen muß.

- c) Für jede Kante $k \in K$ gibt es auch eine Kante in \bar{G} mit der Kapazität

$$\bar{\kappa}(k) = \kappa_o(k) - \kappa_u(k).$$

- d) Die Kanten (q, s) und (s, q) mit der Kapazität ∞ (d.h. sehr große Werte), wobei q die Quelle und s die Senke von G ist.

Damit ist \bar{G} ein \bar{q} - \bar{s} -Netzwerk, d.h. die Ecken q und s sind normale Ecken in \bar{G} . Abbildung 7.6 zeigt links ein Netzwerk G mit oberen und unteren Kapazitätsgrenzen und rechts das zugehörige Netzwerk \bar{G} . Untere und obere Grenzen sind wieder durch Komma getrennt. Das Netzwerk \bar{G} hat nur obere Kapazitätsgrenzen, d.h. mit den Algorithmen aus dem letzten Kapitel kann ein maximaler Fluß für \bar{G} bestimmt werden.

Die Konstruktion von \bar{G} bedingt, daß folgende Gleichung gilt:

$$\sum_{e \in E} \bar{\kappa}(k_e) = \sum_{e \in E} \bar{\kappa}(k^e)$$

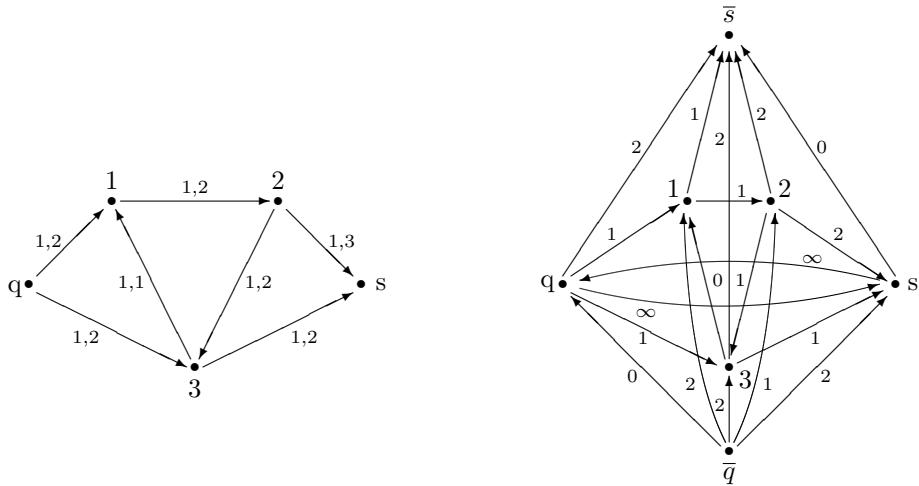


Abbildung 7.6: Ein Netzwerk G mit oberen und unteren Kapazitätsgrenzen und das dazugehörige Netzwerk \bar{G}

Diese Größe wird im folgenden mit S bezeichnet. S ist eine obere Grenze für den Wert eines Flusses auf \bar{G} . Für das Beispiel aus Abbildung 7.6 gilt $S = 7$. Der Wert eines maximalen Flusses auf \bar{G} zeigt an, ob es einen zulässigen Fluß auf G gibt. Das wird in dem folgenden Lemma bewiesen.

Lemma. Genau dann gibt es einen zulässigen Fluß auf dem Netzwerk G , wenn der maximale Fluß auf dem Netzwerk \bar{G} den Wert S hat.

Beweis. Es sei zunächst \bar{f} ein maximaler Fluß auf dem Netzwerk \bar{G} mit Wert S . Für jede Kante k von G setze

$$f(k) = \bar{f}(k) + \kappa_u(k).$$

Im folgenden wird nun gezeigt, daß f ein zulässiger Fluß auf G ist. Da

$$\bar{\kappa}(k) = \kappa_o(k) - \kappa_u(k)$$

ist, gilt

$$0 \leq \bar{f}(k) \leq \bar{\kappa}(k)$$

$$\kappa_u(k) \leq \bar{f}(k) + \kappa_u(k) \leq \bar{\kappa}(k) + \kappa_u(k)$$

$$\kappa_u(k) \leq f(k) \leq \kappa_o(k).$$

Somit ist nur noch die Flußerhaltungsbedingung für alle Ecken $e \neq q, s$ von G nachzuweisen.

Es sei K die Kantenmenge und E die Eckenmenge von G . Nach Voraussetzung ist

$$\sum_{e \in E} \bar{f}(k^e) = |\bar{f}| = S = \sum_{e \in E} \bar{\kappa}(k^e).$$

Da $0 \leq \bar{f}(k^e) \leq \bar{\kappa}(k^e)$ für alle Kanten k^e gilt, ist $\bar{f}(k^e) = \bar{\kappa}(k^e)$. Analog zeigt man $\bar{f}(k_e) = \bar{\kappa}(k_e)$. Da f ein Fluß auf \bar{G} ist, muß \bar{f} die Flußerhaltungsbedingung für die Ecke e erfüllen. Somit gilt:

$$\begin{aligned} \bar{f}(k_e) + \sum_{k=(e,w) \in K} \bar{f}(k) &= \bar{f}(k^e) + \sum_{k=(w,e) \in K} \bar{f}(k) \\ \bar{\kappa}(k_e) + \sum_{k=(e,w) \in K} \bar{f}(k) &= \bar{\kappa}(k^e) + \sum_{k=(w,e) \in K} \bar{f}(k) \\ \sum_{k=(e,w) \in K} \kappa_u(k) + \sum_{k=(e,w) \in K} \bar{f}(k) &= \sum_{k=(w,e) \in K} \kappa_u(k) + \sum_{k=(w,e) \in K} \bar{f}(k) \\ \sum_{k=(e,w) \in K} f(k) &= \sum_{k=(w,e) \in K} f(k) \end{aligned}$$

Daraus folgt, daß f ein zulässiger Fluß für G ist.

Sei nun f ein zulässiger Fluß für G . Im folgenden wird nun ein Fluß \bar{f} auf \bar{G} mit Wert S konstruiert. Für alle $k \in K$ setze

$$\bar{f}(k) = f(k) - \kappa_u(k),$$

und für alle $e \in E$ setze

$$\bar{f}(k_e) = \bar{\kappa}(k_e) \quad \text{und} \quad \bar{f}(k^e) = \bar{\kappa}(k^e).$$

Analog zum ersten Teil zeigt man nun, daß \bar{f} für alle Ecken aus $E \setminus \{q, s\}$ die Flußerhaltungsbedingung erfüllt. Ferner ist $|\bar{f}| = S$. Den Wert von \bar{f} für die Kanten von q nach s und s nach q definiert man so, daß für die Ecken q, s ebenfalls die Flußerhaltungsbedingung erfüllt ist. Da f ein zulässiger Fluß für G ist, gilt auch $0 \leq \bar{f}(k)$ für alle Kanten k von \bar{G} . ■

Für das Netzwerk \bar{G} aus Abbildung 7.6 hat ein maximaler Fluß \bar{f} folgende Werte:

$$\begin{array}{lll} \bar{f}(\bar{q}, 1) = 2 & \bar{f}(1, \bar{s}) = 1 & \bar{f}(s, q) = 2 \\ \bar{f}(\bar{q}, 2) = 1 & \bar{f}(1, 2) = 1 & \bar{f}(\bar{q}, \bar{s}) = 2 \\ \bar{f}(\bar{q}, 3) = 2 & \bar{f}(2, \bar{s}) = 2 & \\ \bar{f}(\bar{q}, s) = 2 & \bar{f}(3, \bar{s}) = 2 & \end{array}$$

Auf allen anderen Kanten hat \bar{f} den Wert 0. Somit hat \bar{f} den Wert 7. Da auch S gleich 7 ist, gibt es einen zulässigen Fluß f auf G . Dieser hat folgende Werte:

$$\begin{array}{lll} f(q, 1) = 1 & f(2, 3) = 1 & f(3, s) = 1 \\ f(q, 3) = 1 & f(2, s) = 1 & \\ f(1, 2) = 2 & f(3, 1) = 1 & \end{array}$$

Nachdem ein zulässiger Fluß gefunden wurde, kann dieser zu einem maximalen Fluß erhöht werden. Dazu muß die Definition eines Erweiterungsweges geändert werden. Für eine *Vorwärtskante* k muß nun

$$f(k) < \kappa_o(k)$$

und für eine *Rückwärtskante*

$$f(k) > \kappa_u(k)$$

gelten. Ferner ist

$$\begin{aligned} f_v &= \min \{ \kappa_o(k) - f(k) \mid k \text{ ist Vorwärtskante auf dem Erweiterungsweg} \} \\ f_r &= \min \{ f(k) - \kappa_u(k) \mid k \text{ ist Rückwärtskante auf dem Erweiterungsweg} \}. \end{aligned}$$

Nimmt man diese Änderungen in dem im Abschnitt 6.3 beschriebenen Algorithmus von Edmonds und Karp vor, so bestimmt dieser ausgehend von einem zulässigen Fluß einen maximalen Fluß. Auch der Satz von Ford und Fulkerson aus Abschnitt 6.2 ist mit diesen Änderungen für Netzwerke mit oberen und unteren Kapazitätsgrenzen korrekt. Hierbei ist zu beachten, daß die Kapazität eines Schnittes (X, \bar{X}) in diesem Falle wie folgt definiert ist:

$$\kappa(X, \bar{X}) = \sum_{\substack{k=(i,j) \in K \\ i \in X, j \in \bar{X}}} \kappa_o(k) - \sum_{\substack{k=(i,j) \in K \\ i \in \bar{X}, j \in X}} \kappa_u(k)$$

Für Netzwerke mit oberen und unteren Kapazitätsgrenzen gilt der folgende Satz:

Satz (FORD UND FULKERSON). Es sei N ein Netzwerk mit oberen und unteren Kapazitätsgrenzen, welches einen zulässigen Fluß besitzt. Der Wert eines maximalen Flusses auf N ist gleich der minimalen Kapazität eines Schnittes von N .

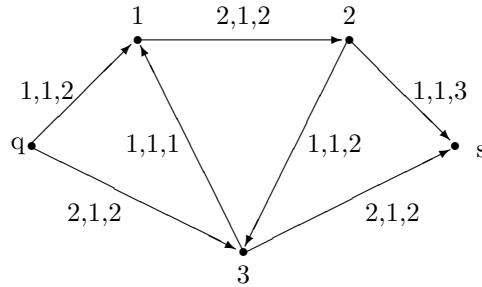


Abbildung 7.7: Maximaler Fluß für das Netzwerk aus Abbildung 7.6

Der Algorithmus von Dinic läßt sich leicht ändern, so daß auch für Netzwerke mit oberen und unteren Kapazitätsgrenzen mit Aufwand $O(n^3)$ entschieden werden kann, ob ein

zulässiger Fluß existiert, und wenn ja, daß dieser auch bestimmt werden kann. Dazu beachte man, daß das Netzwerk G genau $n + 2$ Ecken und $2n + m + 2$ Kanten hat.

Abbildung 7.7 zeigt einen maximalen Fluß f für das Netzwerk aus Abbildung 7.6. Jede Kante k ist mit drei Zahlen markiert: $f(k), \kappa_u(k), \kappa_o(k)$.

Der Wert von f ist 3. Läßt man die unteren Grenzen weg, so hat ein maximaler Fluß den Wert 4; d.h. die unteren Kapazitätsgrenzen können bewirken, daß der Wert eines maximalen Flusses geringer ist als der Wert im gleichen Netzwerk ohne untere Kapazitätsgrenzen.

7.3 Eckenzusammenhang in ungerichteten Graphen

In diesem Abschnitt werden Algorithmen diskutiert, mit denen die in Abschnitt 1.1 beschriebenen Fragen über *Verletzlichkeit* von Kommunikationsnetzen gelöst werden können. Ein Problem in diesem Zusammenhang ist die Bestimmung der minimalen Anzahl von Stationen, deren Ausfall die Kommunikation der verbleibenden Stationen unmöglich machen würde. Dieses Problem führt zu dem Begriff *trennende Eckenmenge*. Eine Menge T von Ecken eines ungerichteten Graphen heißt trennende Eckenmenge für zwei Ecken a und b , falls jeder Weg von a nach b mindestens eine Ecke aus T verwendet. Für Ecken a, b , welche durch eine Kante verbunden sind, gibt es keine trennende Eckenmenge. Für die Ecken 2 und 9 des Graphen aus Abbildung 7.8 ist $\{1, 3, 4, 10\}$ eine trennende Eckenmenge. In den Anwendungen interessiert man sich meistens für trennende Eckenmengen mit möglichst wenig Ecken. Dazu dient folgende Definition:

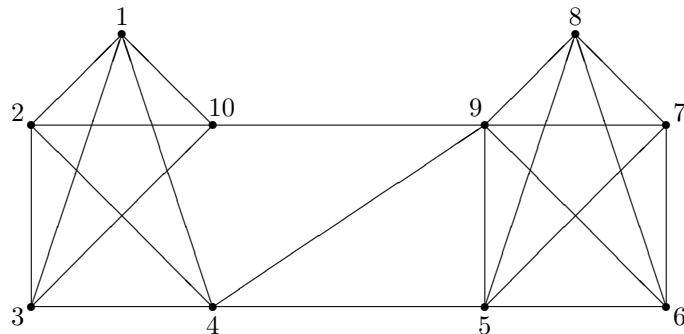


Abbildung 7.8: Ein ungerichteter Graph

Es seien a, b nicht benachbarte Ecken (im folgenden mit $a \not\sim b$ abgekürzt) eines ungerichteten Graphen. Dann ist

$$Z^e(a, b) = \min \{ |T| \mid T \text{ ist trennende Eckenmenge für } a, b \}$$

die minimale Anzahl von Ecken einer trennenden Eckenmenge für a, b . Eine trennende

Eckenmenge T mit $|T| = Z^e(a, b)$ heißt *minimale trennende Eckenmenge* für a, b . Für den Graphen aus Abbildung 7.8 ist $\{4, 10\}$ eine minimale trennende Eckenmenge für die Ecken 2 und 9, somit ist $Z^e(2, 9) = 2$.

Zur Bestimmung von $Z^e(a, b)$ wird eine weitere Definition benötigt. Sind a, b Ecken eines ungerichteten Graphen, so bezeichnet $W^e(a, b)$ die maximale Anzahl von paarweise eckendisjunkten Wegen von a nach b . Dabei sind zwei Wege *eckendisjunkt*, falls sie bis auf Anfangs- und Endecke keine gemeinsamen Ecken verwenden. Es sei T eine minimale trennende Eckenmenge für a, b mit $a \not\sim b$. Dann verwendet jeder Weg von a nach b mindestens eine Ecke aus T . Somit gibt es maximal $|T|$ paarweise eckendisjunkte Wege von a nach b ; d.h.

$$Z^e(a, b) \geq W^e(a, b),$$

falls $a \not\sim b$. Allgemein gilt folgender Satz:

Satz (MENGER). Es seien a, b Ecken eines ungerichteten Graphen mit $a \not\sim b$. Dann gilt $Z^e(a, b) = W^e(a, b)$.

Beweis. Es genügt, zu zeigen, daß $W^e(a, b) \geq Z^e(a, b)$ ist. Dazu wird ein 0-1-Netzwerk N konstruiert. Der maximale Fluß auf diesem Netzwerk hat den Wert $W^e(a, b)$ und ist mindestens $Z^e(a, b)$. Daraus folgt dann die Behauptung. Zunächst beschreiben wir die Konstruktion des Netzwerkes N .

Für jede Ecke e von G enthält N zwei Ecken e' und e'' und eine Kante von e' nach e'' . Für jede Kante (e, w) von G enthält N die gerichteten Kanten (e'', w') und (w'', e') . Alle Kanten haben die Kapazität 1. In N gibt es $2n$ Ecken und $2m + n$ Kanten. Die Quelle von N ist a'' , und b' ist die Senke. Die Kanten mit Endecke a'' oder Anfangsdecke b' müßten noch entfernt werden. Da sie für das folgende nicht von Bedeutung sind, werden sie nicht explizit entfernt. Abbildung 7.9 zeigt einen ungerichteten Graphen G und das dazugehörige Netzwerk N .

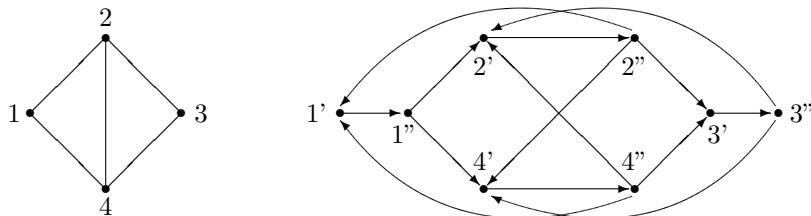


Abbildung 7.9: Ein ungerichteter Graph und das zugehörige Netzwerk N

Zunächst wird gezeigt, daß der Wert eines maximalen Flusses gleich $W^e(a, b)$ ist. Ein Weg W von a nach b in G verwende die Ecken $a, e_1, e_2, \dots, e_s, b$. Zu diesem Weg W gibt es einen entsprechenden Weg W_N von a'' nach b' in N , welcher folgende Ecken

verwendet:

$$a'', e'_1, e''_1, e'_2, e''_2, \dots, e'_s, e''_s, b'$$

Sind zwei Wege von a nach b in G eckendisjunkt, so sind auch die entsprechenden Wege von a'' nach b' in N eckendisjunkt. Somit gibt es mindestens $W^e(a, b)$ eckendisjunkte Wege von a'' nach b' in N . Jeder dieser Wege kann eine Flußeinheit von a'' nach b' transportieren. Damit hat ein maximaler Fluß auf N mindestens den Wert $W^e(a, b)$.

Nach den Ergebnissen aus Abschnitt 6.5 gibt es einen maximalen binären Fluß f auf N . Ferner gibt es $|f|$ kantendisjunkte Wege von a'' nach b' , deren sämtliche Kanten den Fluß 1 tragen. Wegen $g^+(e') = g^-(e'') = 1$ für alle Ecken e von G kann keine Ecke auf zwei verschiedenen solchen Wegen liegen. Somit ist $W^e(a, b) \geq |f|$ und daraus folgt $W^e(a, b) = |f|$.

Um den Beweis zu vervollständigen, genügt es, zu zeigen, daß $|f| \geq Z^e(a, b)$ gilt. Es sei X die Menge aller Ecken aus N , welche bezüglich f durch Erweiterungswege von q aus erreichbar sind. Dann ist $|f| = \kappa(X, \overline{X})$. Sei nun K_X die Menge aller Kanten von N mit Anfangsecke in X und Endecke in \overline{X} und T die Menge der Ecken in G , welche den Enden der Kanten in K_X entsprechen. Ist $k \in K_X$, so gilt $f(k) = 1$. Nach Konstruktion gibt es Ecken $e \neq w$ von G , so daß $k = (w', w'')$ oder $k = (e'', w')$ gilt. Ist $k = (e'', w')$ und $e \neq a$, so trägt wegen $g^-(e'') = 1$ die Kante (e', e'') ebenfalls den Fluß 1. Aus der Flußerhaltungsbedingung folgt, daß der Fluß durch die Kante (e'', x') gleich 0 sein muß. Dies zeigt, daß e'' nicht in X liegen kann. Somit ist $k = (w', w'')$ oder $k = (a'', w')$ für eine Ecke $w \neq a$. Die Enden zweier Kanten in K_X entsprechen somit unterschiedlichen Ecken in T . Also ist $|T| = |K_X| = |f|$.

Jeder Weg W von a nach b in G induziert einen Weg W_N von a' nach b' in N . Hierzu ersetzt man jede Kante (e, f) in W durch die Kanten $(e', e''), (e'', f')$. W_N muß eine Kante aus K_X verwenden und somit muß auch W eine Ecke aus T verwenden, d.h. T ist eine trennende Eckenmenge für a, b . Dies zeigt $|T| \geq Z^e(a, b)$. Daraus folgt

$$W^e(a, b) = |f| = |K_X| = |T| \geq Z^e(a, b).$$

■

Aus dem Beweis des Mengerschen Satzes ergibt sich direkt ein Algorithmus zur Bestimmung von $Z^e(a, b)$. Dazu beachte man, daß N aus insgesamt $2n$ Ecken und $n + 2m$ Kanten besteht. Somit kann $Z^e(a, b)$ mittels des Algorithmus von Dinic mit Aufwand $O(\sqrt{nm})$ bestimmt werden. Mit dem gleichen Aufwand kann auch eine minimale trennende Eckenmenge für a, b gefunden werden.

Die *Eckenzusammenhangszahl* (oder kurz *Zusammenhangszahl*) $Z^e(G)$ eines ungerichteten Graphen G ist wie folgt definiert:

- a) Falls G der vollständige Graph ist, so ist $Z^e(G) = n - 1$.
- b) Andernfalls ist $Z^e(G) = \min \{Z^e(a, b) \mid a, b \text{ Ecken von } G \text{ mit } a \not\sim b\}$.

Für den Graphen aus Abbildung 7.8 ist $Z^e(G) = 2$. Sind a, b Ecken von G mit $a \not\sim b$, so ist $Z^e(a, b) \leq n - 2$. Also gilt $Z^e(G) = n - 1$ genau dann, wenn G vollständig ist. Ist G nicht vollständig, so ist

$$Z^e(G) = \min \{ |T| \mid T \subseteq E, G_{(E \setminus T)} \text{ ist nicht zusammenhängend} \}.$$

Hierbei bezeichnet $G_{(E \setminus T)}$ den von $E \setminus T$ induzierten Untergraphen von G .

Ein Graph G heißt *z-fach zusammenhängend*, falls $Z^e(G) \geq z$ ist. Diese Definition besagt, daß ein Graph genau dann z -fach zusammenhängend ist, wenn er nach dem Entfernen von $z - 1$ beliebigen Ecken immer noch zusammenhängend ist. Für einen ungerichteten Graphen G gilt $Z^e(G) \geq 1$ genau dann, wenn G zusammenhängend ist; d.h. die Begriffe 1-fach zusammenhängend und zusammenhängend sind äquivalent. Somit kann mit Hilfe der Tiefensuche mit Aufwand $O(n + m)$ festgestellt werden, ob $Z^e(G) \geq 1$ ist.

Von allgemeinem Interesse ist die Frage, ob ein Graph z -fach zusammenhängend für ein gegebenes z ist. Für $z = 2$ sind dies die im Kapitel 4 eingeführten zweifach zusammenhängenden Graphen. In Abschnitt 4.9 wurde ein Algorithmus beschrieben, mit dem man mit Aufwand $O(n + m)$ feststellen kann, ob G zweifach zusammenhängend ist. Am Ende dieses Abschnitts wird ein Algorithmus diskutiert, welcher mit Aufwand $O(z^3m + znm)$ feststellt, ob G z -fach zusammenhängend ist.

Ist G nicht vollständig, so gilt nach dem Satz von Menger:

$$Z^e(G) = \min \{ W^e(a, b) \mid a, b \text{ Ecken von } G \text{ mit } a \not\sim b \}$$

In diesem Fall gilt sogar folgende Aussage:

Lemma. Für einen nicht vollständigen ungerichteten Graphen G gilt:

$$Z^e(G) = \min \{ W^e(a, b) \mid a, b \text{ Ecken von } G \}$$

Beweis. Angenommen, die Aussage ist falsch. Dann gibt es eine Kante $k = (e_1, e_2)$ von G , so daß

$$W^e(e_1, e_2) < Z^e(G) = \min \{ W^e(a, b) \mid a, b \text{ Ecken von } G \text{ mit } a \not\sim b \}.$$

Es sei G' der Graph, der aus G entsteht, wenn man k entfernt. In G' ist dann $e_1 \not\sim e_2$. Es sei T eine minimale trennende Eckenmenge für e_1, e_2 in G' . Nach dem Satz von Menger gibt es dann genau $|T|$ eckendisjunkte Wege von e_1 nach e_2 in G' . Somit gibt es in G genau $|T| + 1$ solcher Wege. Daraus folgt:

$$W^e(e_1, e_2) = |T| + 1$$

Da G nicht vollständig ist, gilt $Z^e(G) < n - 1$ und somit $|T| \leq n - 3$. Folglich gibt es eine Ecke x mit $x \notin T \cup \{e_1, e_2\}$. Angenommen, es gibt einen Weg von e_1 nach x ,

der weder die Kante k noch eine Ecke aus T verwendet. Dann folgt, daß $T \cup \{e_1\}$ eine trennende Eckenmenge für x, e_2 ist, und $x \not\sim e_2$. Nach dem Mengerschen Satz gilt dann

$$Z^e(G) \leq W^e(x, e_2) = Z^e(x, e_2) \leq |T| + 1 = W^e(e_1, e_2).$$

Dieser Widerspruch zeigt, daß $T \cup \{e_2\}$ eine trennende Eckenmenge für x, e_1 ist, und $x \not\sim e_1$. Dies führt aber zu dem gleichen Widerspruch. Damit ist das Lemma bewiesen.

■

Für vollständige Graphen gilt $W^e(a, b) = n - 1$ für alle Ecken a, b . Somit folgt, daß $Z^e(G) = \min \{W^e(a, b) \mid a, b \text{ Ecken von } G\}$ für beliebige Graphen gilt. Hieraus ergibt sich ein Kriterium für den z -fachen Zusammenhang eines Graphen. Der folgende Satz wurde 1932 von H. Whitney bewiesen.

Satz (WHITNEY). Ein Graph ist genau dann z -fach zusammenhängend, wenn je zwei Ecken durch mindestens z eckendisjunkte Wege verbunden sind.

Aus dem Beweis des Satzes von Menger folgt, daß für alle Ecken a, b der Wert von $W^e(a, b)$ gleich dem Wert eines maximalen Flusses auf einem entsprechenden Netzwerk ist. Setzt man dies in einen Algorithmus um, so kann $Z^e(G)$ mit Aufwand $O(n^{\frac{5}{2}}m)$ bestimmt werden (der Algorithmus von Dinic wird für jedes Paar von nicht benachbarten Ecken aufgerufen, d.h. maximal $n(n-1)/2$ -mal).

```

function zusammenhangszahl (G : Graph) : Integer;
var
    N : 0-1-Netzwerk;
    z, i, j : Integer;
begin
    N := netzwerk(G);
    z := n - 1; i := 0;
    while i ≤ z do begin
        i := i + 1;
        for j := i + 1 to n do
            if es existiert keine Kante von i nach j then
                z := min {z, maxfluss(N,i,j)};
        end;
        zusammenhangszahl := z;
    end

```

Abbildung 7.10: Die Funktion `zusammenhangszahl`

Dieser Algorithmus kann aber noch verbessert werden. Es ist nämlich nicht in jedem Fall notwendig, für alle nichtbenachbarten Ecken a, b den Wert von $W^e(a, b)$ zu bestimmen. Abbildung 7.10 zeigt die Funktion `zusammenhangszahl`, welche die Zusammenhangszahl eines ungerichteten Graphen G bestimmt. Dabei bestimmt die Funktion `netzwerk(G)` das zu G gehörende Netzwerk gemäß dem Beweis des Satzes von Menger. Die Funktion

`maxfluß(N, i, j)` bestimmt den Wert eines maximalen Flusses auf N mit Quelle i'' und Senke j' .

Zunächst wird die Korrektheit der Funktion `zusammenhangszahl` bewiesen. Die Funktion betrachtet die Ecken in einer festen Reihenfolge und bestimmt für jede Ecke i die Werte von $Z^e(i, j)$ für alle Ecken j mit $j \geq i$, die nicht zu i benachbart sind. Zu jedem Zeitpunkt gilt $z \geq Z^e(G)$, denn z enthält den bisher kleinsten Wert von $Z^e(i, j)$. Es seien a, b Ecken mit $a \not\sim b$ und $Z^e(a, b) = Z^e(G)$. Außerdem sei T eine minimale trennende Eckenmenge für a, b und G' der von $E \setminus T$ induzierte Untergraph von G , wobei E die Eckenmenge von G ist. Dann ist G' nicht zusammenhängend. Nach dem Verlassen der `while`-Schleife ist

$$i > z \geq Z^e(G) = |T|.$$

Somit wurde schon eine Ecke x bearbeitet, welche nicht in T liegt. Es sei y eine Ecke von G' , welche in G' nicht in der gleichen Zusammenhangskomponente wie x liegt. Dann ist T ebenfalls eine trennende Eckenmenge für x, y in G . Somit gilt:

$$Z^e(G) \leq z \leq Z^e(x, y) \leq |T| = Z^e(G)$$

Also ist $z = Z^e(G)$. Damit wurde gezeigt, daß die Funktion `zusammenhangszahl` den Wert $Z^e(G)$ zurückliefert. Falls G ein vollständiger Graph ist, wird die Funktion `maxfluß` nicht aufgerufen.

Zur Bestimmung des Zeitaufwandes der Funktion `zusammenhangszahl` wird noch folgendes Lemma benötigt:

Lemma. Für einen ungerichteten Graphen G gilt $Z^e(G) \leq 2m/n$.

Beweis. Es seien a, b Ecken von G mit $a \not\sim b$. Dann gilt:

$$Z^e(a, b) \leq \min \{g(a), g(b)\}$$

Daraus folgt, daß $Z^e(G) \leq \delta(G)$ ist. Hierbei bezeichnet $\delta(G)$ den kleinsten Eckengrad von G . Da die Summe der Eckengrade aller Ecken gleich $2m$ ist, folgt $n \cdot \delta(G) \leq 2m$. Hieraus folgt die Behauptung. ■

Die Funktion `netzwerk` hat einen Aufwand von $O(n + m)$. Die Funktion `maxfluß` hat einen Aufwand von $O(\sqrt{nm})$, da für die betrachteten 0-1-Netzwerke entweder $g^+(e)$ oder $g^-(e)$ für jede Ecke e gleich 1 ist. In jedem Durchlauf der `while`-Schleife wird die Funktion `maxfluß` maximal $(n - 1)$ -mal aufgerufen. Insgesamt erfolgen maximal $(n - 1)Z^e(G)$ Aufrufe. Nach dem letzten Lemma ist diese Zahl maximal $2m$. Somit gilt folgender Satz:

Satz. Die Zusammenhangszahl eines ungerichteten Graphen kann mit Aufwand $O(\sqrt{nm}^2)$ bestimmt werden.

Durch eine kleine Ergänzung kann mit dem gleichen Verfahren eine Menge T mit einer minimalen Anzahl von Ecken bestimmt werden, so daß der von $E \setminus T$ induzierte Untergraph nicht zusammenhängend ist.

Wendet man die Funktion `zusammenhangszahl` auf den Graphen aus Abbildung 7.8 an, so wird zunächst $Z^e(1, j)$ für $j = 5, \dots, 9$ bestimmt. Da alle Werte gleich 2 sind, ist anschließend z gleich 2. Danach wird $Z^e(2, j)$ ebenfalls für $j = 5, \dots, 9$ bestimmt. Nun ist $i = 3$ und $z = 2$. Die `while`-Schleife wird jetzt verlassen. Somit ist $Z^e(G) = 2$. Insgesamt wurde die Funktion `maxfluß` zehnmal aufgerufen.

In vielen Fällen ist man nicht an der genauen Zusammenhangszahl eines Graphen interessiert, sondern man möchte nur wissen, ob die Zusammenhangszahl nicht unter einem gegebenen Wert liegt. Ist der Graph vollständig, so gibt es nichts zu berechnen. Im folgenden gehen wir davon aus, daß die betrachteten Graphen nicht vollständig sind. Zunächst führen wir zu einem gegebenen Graphen einen Hilfsgraphen ein.

Es sei G ein ungerichteter Graph mit Eckenmenge $E = \{e_1, \dots, e_n\}$. Für $\ell = 1, \dots, n$ sei G_ℓ der ungerichtete Graph, der entsteht, wenn man in G eine zusätzliche Ecke s einfügt und mit e_1, \dots, e_ℓ durch Kanten verbindet. Abbildung 7.11 zeigt links einen ungerichteten Graphen G und rechts den zugehörigen Graphen G_3 .

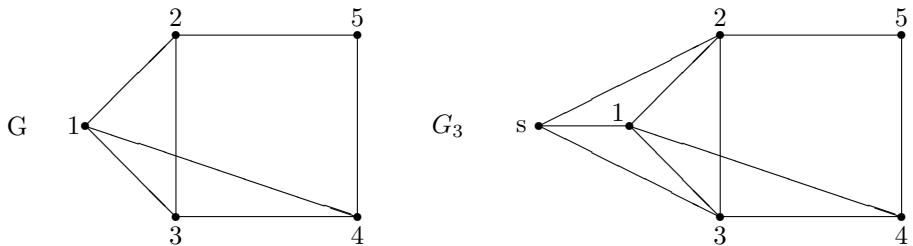


Abbildung 7.11: Ein ungerichteter Graph G und der zugehörige Graph G_3

Mit dieser Bezeichnungsweise gelten folgende zwei Lemmata:

Lemma. Es seien ℓ, z ganze Zahlen mit $\ell \geq z \geq 1$ und $u \in E \setminus \{e_1, \dots, e_\ell\}$, so daß $W^e(e_i, u) \geq z$ in G für $i = 1, \dots, \ell$. Dann ist auch $W^e(s, u) \geq z$ in G_ℓ .

Beweis. Angenommen, es ist $W^e(s, u) < z$ in G_ℓ . Da $s \not\sim u$ ist, gibt es nach dem Mengerschen Satz eine trennende Eckenmenge T für s, u in G_ℓ , so daß $|T| < z$ ist. Wegen $\ell \geq z > |T|$ gibt es ein $i \in \{1, \dots, \ell\}$ mit $e_i \notin T$. Also ist $e_i \not\sim u$. Es sei W ein Weg von u nach e_i in G . Dieser Weg kann zu einem Weg in G_ℓ von u nach s verlängert werden. Somit muß W eine Ecke aus T verwenden. Also ist T eine trennende Eckenmenge für u, e_i in G . Somit gilt $W^e(e_i, u) = Z^e(e_i, u) \leq |T| < z$. Dieser Widerspruch beendet den Beweis. ■

Lemma. Es sei $j \in \{1, \dots, n\}$ die kleinste Zahl, für die es ein $i < j$ gibt, so daß $Z^e(e_i, e_j) < z$ in G für eine ganze Zahl z ist. Dann gilt $Z^e(s, e_j) < z$ in G_{j-1} .

Beweis. Es sei T eine minimale trennende Eckenmenge für e_i, e_j in G . Nach Voraus-

setzung ist $|T| < z$. Es sei

$$F = \{e \in E \mid T \text{ ist trennende Eckenmenge für } e_i, e \text{ in } G\}.$$

Angenommen es gibt eine ganze Zahl $h < j$, so daß $e_h \in F$ ist. Dann ist $Z^e(e_i, e_h) \leq |T| < z$. Dies steht aber im Widerspruch zur Wahl von j : Ist $h < i$, so hätte i und nicht j die im Lemma angegebene Eigenschaft, und ist $i < h$, so wäre es h gewesen. Somit gilt $\{e_1, \dots, e_{j-1}\} \cap F = \emptyset$. Sei nun W ein Weg von s nach e_j in G_{j-1} . Dann verwendet W eine Ecke $e_t \in \{e_1, \dots, e_{j-1}\}$, da dies die einzigen Nachbarn von s sind. Da $e_t \notin F$, gibt es einen Weg von e_i nach e_t , welcher keine Ecke aus T verwendet. Aus diesen beiden Wegen kann ein Weg von e_i nach e_j gebildet werden. Dieser muß eine Ecke aus T verwenden, welche auf W liegt. Somit ist T eine trennende Eckenmenge für s, e_j in G_{j-1} . Hieraus ergibt sich $Z^e(s, e_j) \leq |T| < z$. ■

Mit Hilfe der Ergebnisse dieser beiden Lemmata läßt sich die Korrektheit der in Abbildung 7.12 dargestellten Prozedur **zusammenhangmin** beweisen. Diese Prozedur stellt fest, ob $Z^e(G) \geq z$ ist.

```

procedure zusammenhangmin (G : Graph; z : Integer);
var
    i, j : Integer;
begin
    for i := 1 to z - 1 do
        for j := i + 1 to z do
            if  $W^e(e_i, e_j) < z$  in G then
                exit('Ze(G) < z');
    for j := z + 1 to n do
        if  $W^e(s, e_j) < z$  in  $G_{j-1}$  then
            exit('Ze(G) < z');
    exit('Ze(G) ≥ z');
end

```

Abbildung 7.12: Die Prozedur **zusammenhangmin**

Satz. Für einen ungerichteten Graphen G kann man mit Aufwand $O(z^3m + znm)$ entscheiden, ob $Z^e(G) \geq z$ ist.

Beweis. Zuerst wird die Korrektheit der Prozedur **zusammenhangmin** bewiesen. Es sei G ein ungerichteter Graph mit $Z^e(G) \geq z$. Dann ist $W^e(e_i, e_j) \geq z$ für alle Ecken e_i, e_j von G . Die Korrektheit folgt nun aus dem ersten Lemma. Ist $Z^e(G) < z$, so wähle man die kleinste Zahl j , für die es ein $i < j$ gibt, so daß $Z^e(e_i, e_j) < z$ ist. Dann ist auch $W^e(e_i, e_j) < z$. Gilt $j \leq z$, so wird die Prozedur durch die erste **exit**-Anweisung verlassen. Andernfalls gilt nach dem letzten Lemma

$$Z^e(s, e_j) = W^e(s, e_j) < z$$

in G_{j-1} . Somit wird die Prozedur durch die zweite **exit**-Anweisung verlassen.

Um den Aufwand von **zusammenhangmin** zu bestimmen, beachte man, daß die Ungleichung $W^e(s, e_j) \geq z$ gleichbedeutend ist mit: Es gibt einen Fluß f auf dem zu dem Graphen gehörenden Netzwerk mit $|f| \geq z$. Da die Kapazitäten ganzzahlig sind, müssen also maximal z Erweiterungswege bestimmt werden. Dies erfordert einen Aufwand von $O(zm)$. Dadurch ergibt sich leicht, daß der Gesamtaufwand gleich $O(z^3m + znm)$ ist.

■

Für $z = 1, 2$ sind die in Kapitel 4 angegebenen Algorithmen natürlich effizienter. Auch für $z = 3$ und 4 sind effizientere Algorithmen bekannt, leider sind diese aber sehr kompliziert.

Wendet man die Prozedur **zusammenhangmin** mit $z = 3$ auf den Graphen G aus Abbildung 7.11 an, so stellt man fest, daß $Z^e(G) < 3$ ist. Die einzelnen Zwischenschritte sind:

$$\begin{aligned} W^e(1, 2) &\geq 3 \quad \text{in } G \\ W^e(1, 3) &\geq 3 \quad \text{in } G \\ W^e(2, 3) &\geq 3 \quad \text{in } G \\ W^e(s, 4) &\geq 3 \quad \text{in } G_3 \\ W^e(s, 5) &< 3 \quad \text{in } G_4 \end{aligned}$$

7.4 Kantenzusammenhang in ungerichteten Graphen

Die Fragen über Verletzlichkeit von Kommunikationsnetzen führten im letzten Abschnitt zu dem Begriff der trennenden Eckenmenge. Die Betrachtung der minimalen Anzahl von Leitungen, deren Ausfall die Funktion des Netzwerkes beeinträchtigt, führt zu dem analogen Begriff der trennenden Kantenmenge. Eine Menge T von Kanten eines ungerichteten Graphen heißt *trennende Kantenmenge* für zwei Ecken a und b , falls jeder Weg von a nach b mindestens eine Kante aus T verwendet. Für die Ecken 1 und 4 des Graphen aus Abbildung 7.13 ist $\{(3, 4), (1, 4), (1, 6)\}$ eine trennende Kantenmenge. Ähnlich wie bei trennenden Eckenmengen interessiert man sich meistens für trennende Kantenmengen mit möglichst wenig Kanten. Man definiert daher für zwei Ecken a, b eines ungerichteten Graphen G

$$Z^k(a, b) = \min \{ |T| \mid T \text{ ist trennende Kantenmenge für } a, b \}.$$

Eine trennende Kantenmenge T mit $|T| = Z^k(a, b)$ heißt *minimale trennende Kantenmenge* für a, b . Für den Graphen aus Abbildung 7.13 ist $\{(5, 4), (5, 6)\}$ eine minimale trennende Kantenmenge für die Ecken 6 und 5; somit ist $Z^k(6, 5) = 2$.

Die *Kantenzusammenhangszahl* $Z^k(G)$ eines ungerichteten Graphen G ist gleich

$$\min \{ Z^k(a, b) \mid a, b \text{ Ecken von } G \}.$$

Für den Graphen G aus Abbildung 7.13 gilt $Z^k(G) = 2$. Genau dann ist $Z^k(G) = 0$, wenn G nicht zusammenhängend ist. Ferner ist $Z^k(C_n) = 2$ für $n \geq 3$ und $Z^k(K_n) =$

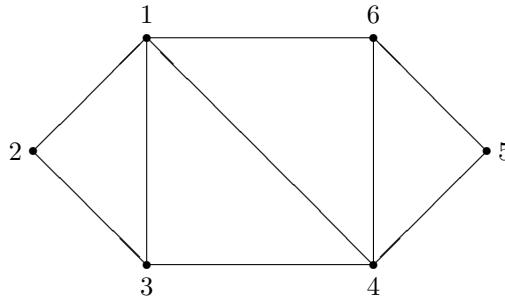


Abbildung 7.13: Ein ungerichteter Graph mit $Z^k(G) = 2$

$n - 1$. Zwischen den Größen $Z^e(G)$, $Z^k(G)$ und dem kleinsten Eckengrad $\delta(G)$ eines ungerichteten Graphen G besteht folgende Beziehung:

Satz. Für einen ungerichteten Graphen G gilt

$$Z^e(G) \leq Z^k(G) \leq \delta(G).$$

Beweis. Ist a eine beliebige Ecke von G , so ist die Menge aller zu a inzidenten Kanten eine trennende Kantenmenge für a und jede andere Ecke. Somit ist

$$Z^k(G) \leq \delta(G).$$

Ist $Z^k(G) = 0$ oder 1 , so ist auch $Z^e(G) = 0$ oder 1 . Es sei nun $Z^k(G) \geq 2$ und a, b Ecken mit $Z^k(a, b) = Z^k(G)$. Es sei $T = \{k_1, \dots, k_s\}$ eine minimale trennende Kantenmenge für a, b . Es sei $k_s = (e, w)$. Zu jeder Kante k_1, \dots, k_{s-1} wähle man eine von e, w verschiedene, mit ihr inzidente Ecke. Mit G' bezeichne man den von den restlichen Ecken induzierten Untergraphen von G . Da die Kanten k_1, \dots, k_{s-1} nicht in G' liegen, ist G' nicht zusammenhängend oder $Z^k(G') = 1$. Im ersten Fall folgt sofort $Z^e(G) < Z^k(G)$. Im zweiten Fall erreicht man durch Entfernen von e oder w , daß der Graph nur noch aus einer Ecke besteht oder nicht zusammenhängend ist. Somit gilt $Z^e(G) \leq Z^k(G)$. ■

Man beachte, daß die im letzten Satz angegebenen Größen alle verschieden sein können. Für den Graphen aus Abbildung 7.8 gilt: $Z^e(G) = 2$, $Z^k(G) = 3$ und $\delta(G) = 4$.

Zur Bestimmung von $Z^k(a, b)$ wird eine weitere Definition benötigt. Sind a, b Ecken eines ungerichteten Graphen, so bezeichnet $W^k(a, b)$ die maximale Anzahl kantendisjunkter Wege von a nach b . Die Verfahren zur Bestimmung von $Z^k(G)$ und $Z^e(G)$ sind sehr ähnlich und verwenden beide Netzwerkalgorithmen.

Das zu einem ungerichteten zusammenhängenden Graphen G gehörende *symmetrische Netzwerk* G_s ist wie folgt definiert:

- a) G_s hat die gleiche Eckenmenge wie G .

- b) Zu jeder Kante (u, v) von G gibt es in G_s die gerichteten Kanten (u, v) und (v, u) .
- c) Alle Kanten haben die Kapazität 1.

Sind a, b beliebige Ecken von G , so ist G_s ein a - b -Netzwerk, wenn man die Kanten mit Endecke a und Anfangsseite b entfernt. Im folgenden werden diese Kanten aber nicht explizit entfernt, da sie keine Auswirkung auf die betrachteten Flüsse haben. Abbildung 7.14 zeigt links einen ungerichteten Graphen G und rechts das dazugehörige symmetrische Netzwerk G_s .

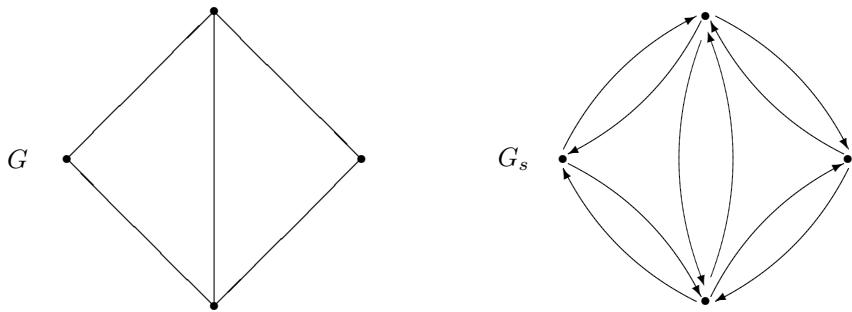


Abbildung 7.14: Ein ungerichteter Graph G und das zugehörige symmetrische Netzwerk G_s

Es gilt folgende Beziehung zwischen $W^k(a, b)$ und einem maximalen Fluß auf G_s .

Lemma. Es seien a, b Ecken eines zusammenhängenden ungerichteten Graphen G und f ein maximaler Fluß von a nach b auf G_s . Dann gilt

$$|f| = W^k(a, b).$$

Beweis. Jeder der $W^k(a, b)$ kantendisjunkten Wege erlaubt den Fluß von einer Einheit von a nach b . Somit ist $|f| \geq W^k(a, b)$. Da G_s ein 0-1-Netzwerk ist, gibt es nach Abschnitt 6.5 einen maximalen binären Fluß f . Existiert in G_s eine Kante (u, v) , so daß $f(u, v) = f(v, u) = 1$ ist, so wird der Fluß auf diesen beiden Kanten auf 0 gesetzt. Dann ist f immer noch ein maximaler Fluß. Analog zu Abschnitt 6.5 zeigt man nun, daß dieser Fluß f durch genau $|f|$ Erhöhungen mittels Erweiterungswegen gebildet werden kann. Die entsprechenden Wege in G haben keine Kante gemeinsam. Somit ist $|f| = W^k(a, b)$.

■

Mit Hilfe dieses Satzes können wir nun eine zum Satz von Menger analoge Aussage für Kanten beweisen.

Satz. Es seien a, b Ecken eines ungerichteten Graphen. Dann gilt

$$W^k(a, b) = Z^k(a, b).$$

Beweis. Liegen a und b in verschiedenen Zusammenhangskomponenten von G , so ist die Aussage trivialerweise erfüllt. Somit kann man annehmen, daß G zusammenhängend ist. Nach dem Satz von Ford und Fulkerson und dem obigen Lemma gibt es einen a - b -Schnitt (X, \bar{X}) von G_s mit $W^k(a, b) = \kappa(X, \bar{X})$. Es sei nun $T = \{(e, w) \mid e \in X, w \in \bar{X}\}$. Da alle Kanten die Kapazität 1 haben, ist $|T| = W^k(a, b)$. T ist eine trennende Kantenmenge für a, b . Somit gilt $Z^k(a, b) \leq W^k(a, b)$. Es sei nun S eine minimale trennende Kantenmenge für a, b in G . Da jeder Weg von a nach b mindestens eine Kante aus S verwenden muß, gilt

$$Z^k(a, b) = |S| \geq W^k(a, b).$$

Damit ist der Satz bewiesen. ■

Der folgende Satz entspricht dem Satz von Whitney für kantendisjunkte Wege. Der Beweis ergibt sich sofort aus dem letzten Lemma.

Satz. Es sei G ein ungerichteter Graph. Genau dann ist $Z^k(G) \geq \ell$, wenn je zwei Ecken durch mindestens ℓ kantendisjunkte Wege verbunden sind.

Ein Algorithmus zur Bestimmung von $Z^k(G)$ ergibt sich leicht aus dem obigen Lemma. Für alle Paare a, b von Ecken bestimmt man $W^k(a, b)$ mit Hilfe des symmetrischen Netzwerkes. Mit Hilfe des Algorithmus von Dinic kann $W^k(a, b)$ mit Aufwand $O(n^{2/3}m)$ bestimmt werden. Das folgende Lemma zeigt, daß man dabei mit $n - 1$ Flußbestimmungen auskommt.

Lemma. Es sei G ein ungerichteter zusammenhängender Graph mit Eckenmenge E . Ist a eine beliebige Ecke von G , so gilt

$$Z^k(G) = \min \{Z^k(a, b) \mid b \in E, b \neq a\}.$$

Beweis. Es seien e, w Ecken von G mit $Z^k(e, w) = Z^k(G)$ und T eine trennende Kantenmenge für G . Aus G entferne man die Kanten, die in T sind. Der resultierende Graph G' ist nicht mehr zusammenhängend. Man wähle nun eine Ecke b , so daß a und b in verschiedenen Zusammenhangskomponenten von G' liegen. Dann ist auch T eine trennende Kantenmenge für G , und es gilt $Z^k(G) = |T| = Z^k(a, b)$. ■

Aus dem letzten Lemma folgt, daß die Kantenzusammenhangszahl eines ungerichteten Graphen mit Aufwand $O(n^{5/3}m)$ bestimmt werden kann. Im nächsten Abschnitt wird ein Algorithmus mit Laufzeit $O(nm)$ für dieses Problem vorgestellt.

7.5 Minimale Schnitte

Das im letzten Abschnitt eingeführte Konzept der trennenden Kantenmenge kann auf kantenbewertete Graphen erweitert werden. Die Summe der Bewertungen der Kanten

einer trennenden Kantenmenge T nennt man die Kosten von T . Eine trennende Kantenmenge eines kantenbewerteten, zusammenhängenden, ungerichteten Graphen G mit minimalen Kosten nennt man einen *minimalen Schnitt* von G . Diese Definition erfolgt in Anlehnung an die Definition der Kapazität eines Schnittes aus Kapitel 6. Ein Schnitt wurde dort als eine Partition (X, \bar{X}) der Ecken des Netzwerkes definiert. Die Kapazität von (X, \bar{X}) ist die Summe der Kapazitäten der Kanten mit Anfangsecke in X und Endecke in \bar{X} . Diese Kanten bilden eine trennende Kantenmenge des Netzwerkes. Abbildung 7.15 zeigt einen kantenbewerteten ungerichteten Graphen. Die drei fett gezeichneten Kanten bilden dabei einen minimalen Schnitt mit Kosten 4.

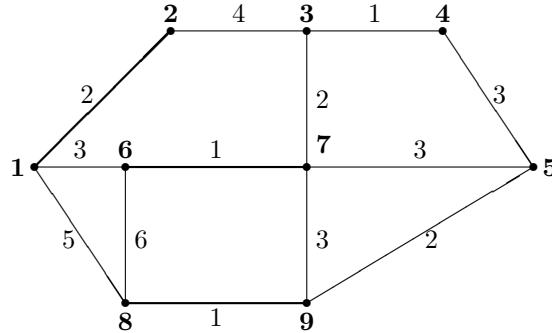


Abbildung 7.15: Ein ungerichteter Graph mit einem minimalen Schnitt

Die Bestimmung von minimalen Schnitten kann mittels des Satzes von Ford und Fulkerson auf die Bestimmung von maximalen Flüssen zurückgeführt werden. Dazu wird wie im letzten Abschnitt zu einem ungerichteten, kantenbewerteten Graphen G das symmetrische Netzwerk G_S definiert. Im Gegensatz zum letzten Abschnitt haben die Kanten nicht die Kapazität 1, sondern die Kapazität einer Kante in G_S ist gleich der Bewertung der entsprechenden ungerichteten Kante in G . Für jedes Paar von Ecken a, b von G kann G_S wieder als ein a - b -Netzwerk betrachtet werden. Nach dem Satz von Ford und Fulkerson ist die Kapazität eines minimalen a - b -Schnittes gleich dem Wert eines maximalen a - b -Flusses. Somit ist der Wert eines minimalen Schnittes gleich

$$\min \{|f_{ab}| \mid f_{ab} \text{ maximaler Fluß auf } G_S \text{ und } a, b \text{ Ecken von } G\}.$$

Somit kann man mit $n(n-1)/2$ Flußbestimmungen den Wert eines minimalen Schnittes bestimmen. Es genügen sogar $n-1$ Flußbestimmungen. Dazu beachte man, daß G_S ein symmetrisches Netzwerk ist. Sind a, b beliebige Ecken von G , so haben maximale Flüsse von a nach b und von b nach a den gleichen Wert. Sei nun a eine feste Ecke von G und (X, \bar{X}) ein minimaler Schnitt von G . Dann gibt es eine Ecke b von G , so daß a und b durch den Schnitt (X, \bar{X}) getrennt werden. Nach dem Satz von Ford und Fulkerson ist der Wert eines maximalen Flusses von a nach b gleich der Kapazität von (X, \bar{X}) . Somit ist der Wert eines minimalen Schnittes von G gleich

$$\min \{|f_{ab}| \mid f_{ab} \text{ maximaler Fluß auf } G_S \text{ und } b \text{ Ecke von } G\}.$$

Unter Verwendung des Algorithmus von Dinic ergibt sich ein Algorithmus mit Aufwand $O(n^4)$ zur Bestimmung eines minimalen Schnittes. Im folgenden wird ein Algorithmus

vorgestellt, welcher nicht nur effizienter arbeitet, sondern auch mit geringem Aufwand zu implementieren ist.

Dieser Algorithmus stammt von M. Stoer und F. Wagner. Er hat eine starke Ähnlichkeit zu dem im Kapitel 3 vorgestellten Algorithmus von Prim zur Bestimmung minimal aufspannender Bäume. Der aufspannende Baum wurde dabei schrittweise erzeugt. In jedem Schritt wurde eine Ecke und eine Kante in den Baum eingefügt. Ist U die Menge der Ecken des aktuellen Baumes und E die Menge der Ecken des Graphen, so wählt man unter den Kanten (u, v) mit $u \in U$ und $v \in E \setminus U$ diejenige mit der kleinsten Bewertung aus und fügt sie in den Baum ein. Dies wiederholt man so lange, bis der Baum ein aufspannender Baum ist, d.h. $U = E$.

Die Bestimmung eines minimalen Schnittes eines kantenbewerteten Graphen erfolgt in $n - 1$ Phasen. Jede einzelne Phase ist ähnlich dem Algorithmus von Prim.

Der wesentliche Unterschied liegt dabei in der Auswahl der Ecke v . Es sei $U \subseteq E$ und v eine beliebige Ecke aus E . Dann setzt man

$$\text{kosten}(U, v) = \sum_{u \in U} \text{kosten}(u, v).$$

In jedem Schritt wird nun die Ecke $e \in E \setminus U$ ausgewählt, für die $\text{kosten}(U, e)$ maximal ist. In jeder Phase wird auf diese Art eine Menge U , die anfangs eine beliebige Ecke enthält, so lange erweitert, bis sie alle Ecken enthält. Der minimale Schnitt dieser Phase besteht aus der Menge der Kanten, welche die zuletzt eingefügte Ecke von dem Rest des Graphen trennt. Die Summe der Bewertungen dieser Kanten nennt man die Kosten der Phase. Am Ende jeder Phase werden die beiden zuletzt ausgewählten Ecken e_1, e_2 zu einer Ecke verschmolzen. Die zu den Ecken e_1 und e_2 inzidenten Kanten sind danach zu der neuen Ecke inzident. Entsteht dabei zwischen zwei Ecken eine Doppelkante, so wird diese zu einer Kante zusammengefaßt, deren Bewertung gleich der Summe der ursprünglichen Bewertungen ist. Der Schnitt mit den geringsten Kosten aus allen $n - 1$ Phasen bildet einen minimalen Schnitt des Graphen.

Abbildung 7.16 zeigt die Funktion `minSchnittPhase`, welche die Kosten einer Phase bestimmt und den Graphen entsprechend reduziert. Alle Phasen beginnen mit der gleichen Ecke, in diesem Fall die Ecke 1. Nach dem Verlassen der **while**-Schleife ist e_2 die zuletzt und e_1 die zu vorletzt betrachtete Ecke. Die Kosten der Phase sind gleich der Summe der Kosten der Kanten von e_2 zu den restlichen Ecken des Graphen. Die Prozedur `verschmelze` nimmt die oben beschriebene Reduktion des Graphen vor und wird hier nicht näher beschrieben.

Die Funktion `minSchnitt`, welche die Kosten eines minimalen Schnittes bestimmt, ist in Abbildung 7.17 dargestellt. Um die Kanten des minimalen Schnittes explizit zu bestimmen, müssen die Reduktionen der Ecken abgespeichert werden. Es sei v die zuletzt betrachtete Ecke in der Phase, in der die geringsten Kosten entstanden. Ferner sei X die Menge der Ecken, die bis dahin zu v reduziert wurden. Dann bilden die Kanten mit Anfangsecke in X und Endecke in \bar{X} einen minimalen Schnitt.

Bevor die Korrektheit des Algorithmus bewiesen und die Laufzeit untersucht wird, soll zunächst ein Beispiel betrachtet werden. Abbildung 7.18 zeigt oben einen kantenbewerteten Graphen mit acht Ecken. In den Abbildungen 7.18 und 7.19 sind die Ergebnisse

```

function minSchnittPhase (var G : Graph) : Integer;
var
    U, E : set of Integer;
    e, e1, e2, wert : Integer;
begin
    Initialisiere wert mit 0 und e2 mit 1;
    Initialisiere E mit der Menge der Ecken von G;
    U := {1}
    while U ≠ E do begin
        e1 := e2;
        wähle e2 ∈ E \ U, so daß kosten(U, e2) =
            max{kosten(U, e) | e ∈ E \ U};
        U.einfügen(e2);
    end;
    for jede Kante (e2, e) do
        wert := wert + kosten(e2, e);
    verschmelze(G, e1, e2);
    minSchnittPhase := wert;
end

```

Abbildung 7.16: Die Funktion minSchnittPhase

der einzelnen Phasen dargestellt. Die fünfte Phase liefert einen Schnitt mit den geringsten Kosten. Hierbei ist $X = \{2, 3, 4, 5\}$, und die Kosten des minimalen Schnittes sind 4.

Die Korrektheit des Algorithmus basiert auf folgendem Lemma. Es sei E_i die Menge der Ecken des Graphen G_i in der i -ten Phase. Man bezeichne mit s_i die letzte und mit q_i die vorletzte Ecke der i -ten Phase.

Lemma. Für $i = 1, \dots, n - 1$ ist $(E_i \setminus \{s_i\}, \{s_i\})$ ein minimaler q_i - s_i -Schnitt in dem zu dem Graphen G_i gehörenden symmetrischen Netzwerk G_{i_S} .

Beweis. Es sei (Y, \bar{Y}) ein beliebiger q_i - s_i -Schnitt von G_{i_S} mit Kapazität κ_Y . Eine Ecke $v \neq 1$ aus G_i heißt bezüglich (Y, \bar{Y}) aktiv, wenn die Ecke w , welche unmittelbar vor v betrachtet wurde, durch (Y, \bar{Y}) von v getrennt wird. Mit U_v wird die Menge der vor v betrachteten Ecken bezeichnet. Für $v \in E_i$ sei κ_v die Summe der Kosten der Kanten (u_1, u_2) mit $u_1 \in Y \cap (U_v \cup \{v\})$ und $u_2 \in \bar{Y} \cap (U_v \cup \{v\})$. Nun wird mittels vollständiger Induktion nach der Anzahl der aktiven Ecken gezeigt, daß

$$\text{kosten}(U_v, v) \leq \kappa_v$$

für alle aktiven v Ecken gilt. Für die erste aktive Ecke v gilt

$$\kappa_v = \sum_{u \in U_v} \text{kosten}(v, u) = \text{kosten}(U_v, v).$$

```

function minSchnitt (G : Graph) : Integer;
var
    minWert : Integer;
begin
    Initialisiere minWert mit 0;
    while G.anzahl > 1 do
        minWert := min(minWert,minSchnittPhase(G));
        minSchnitt := minWert;
    end

```

Abbildung 7.17: Die Funktion minSchnitt

Sei nun v eine beliebige aktive Ecke und w die darauffolgende aktive Ecke. Da v vor w ausgewählt wurde, gilt $kosten(U_v, w) \leq kosten(U_v, v)$. Nach Induktionsvoraussetzung gilt $kosten(U_v, v) \leq \kappa_v$. Somit gilt

$$\begin{aligned}
kosten(U_w, w) &= kosten(U_v, w) + \sum_{u \in U_w \setminus U_v} kosten(u, w) \\
&\leq \kappa_v + \sum_{u \in U_w \setminus U_v} kosten(u, w) \\
&\leq \kappa_w.
\end{aligned}$$

Die letzte Ungleichung gilt, da die Enden der Kanten (u, w) mit $u \in U_w \setminus U_v$ durch (Y, \bar{Y}) getrennt werden.

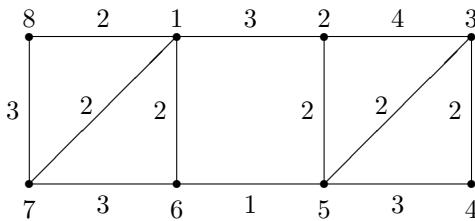
Da die letzte Ecke s_i immer aktiv ist, gilt

$$\text{minSchnittPhase} = kosten(U_{s_i}, s_i) \leq \kappa_{s_i} \leq \kappa_Y.$$

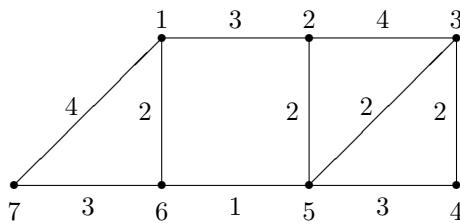
Daraus folgt die Behauptung. ■

Satz. Die Funktion minSchnitt bestimmt mit Aufwand $O(nm \log n)$ die Kosten eines minimalen Schnittes eines ungerichteten Graphen G .

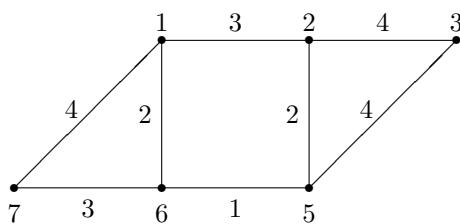
Beweis. Der Beweis der Korrektheit wird durch vollständige Induktion nach der Anzahl n der Ecken geführt. Die Aussage ist für $n = 2$ richtig. Sei nun $n > 2$. Es seien e_1, e_2 die zuletzt betrachteten Ecken der ersten Phase. Ist die Kapazität eines minimalen e_1 - e_2 -Schnittes von G_s gleich den Kosten eines minimalen Schnittes von G , so folgt die Aussage aus dem Lemma. Andernfalls gibt es einen minimalen Schnitt (X, \bar{X}) von G , so daß $e_1, e_2 \in X$ gilt. Es sei G' der Graph zu Beginn der zweiten Phase. Da e_1 und e_2 in G' zu einer Ecke reduziert wurden, sind die Kosten von minimalen Schnitten von G und G' gleich. Beachtet man nun, daß eine Anwendung der Phasen 2 bis $n - 1$ auf G' die gleiche Wirkung hat wie die Anwendung der Funktion minSchnitt auf G' , so folgt die Behauptung aus der Induktionsvoraussetzung.



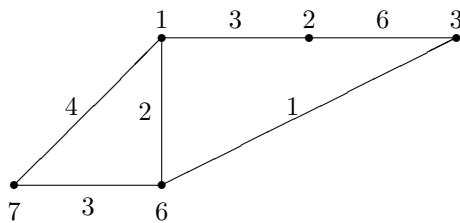
Reihenfolge der Ecken: 1,2,3,5,4,6,7,8
minSchnittPhase: 5



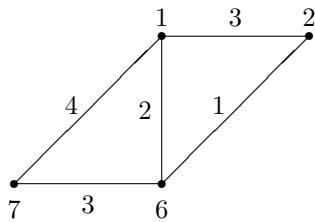
Reihenfolge der Ecken: 1,7,6,2,3,5,4
minSchnittPhase: 5



Reihenfolge der Ecken: 1,7,6,2,3,5
minSchnittPhase: 7



Reihenfolge der Ecken: 1,7,6,2,3
minSchnittPhase: 7



Reihenfolge der Ecken: 1,7,6,2
minSchnittPhase: 4

Abbildung 7.18: Eine Anwendung des Algorithmus von Stoer und Wagner

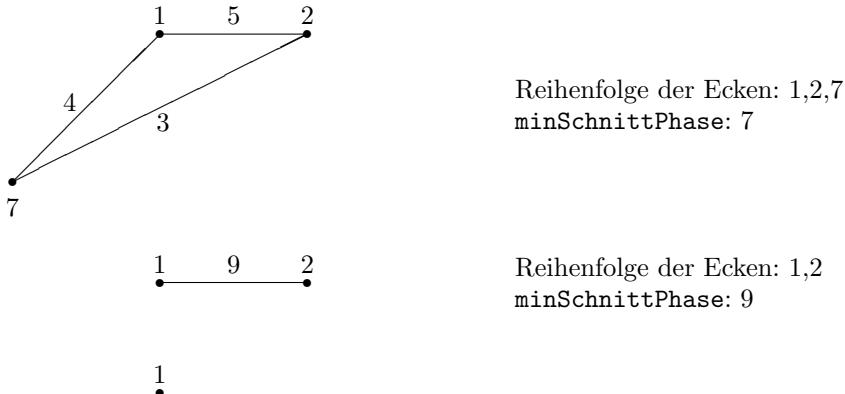


Abbildung 7.19: Eine Anwendung des Algorithmus von Stoer und Wagner

Die Laufzeit der Funktion `minSchnitt` ist gleich der Summe der Laufzeiten der $n - 1$ Aufrufe der Funktion `minSchnittPhase`. Die entscheidende Stelle in dieser Funktion ist die Auswahl der Ecke e_2 , so daß $kosten(U, e_2)$ maximal ist. Dazu werden alle Ecken, die nicht in U liegen, in einer Vorrangwarteschlange gehalten. Jede Ecke v bekommt dabei den Wert $kosten(U, v)$ zugewiesen. Das Initialisieren der Vorrangwarteschlange hat den Aufwand $O(n)$. Immer, wenn eine Ecke v aus dieser Warteschlange entfernt und in U eingefügt wird, müssen die Bewertungen der Ecken in der Warteschlange angepaßt werden. Dazu müssen nur die Nachbarn der Ecken betrachtet werden. Somit sind insgesamt $O(m)$ Änderungen von Prioritäten und $O(n)$ Löschungen aus der Warteschlange notwendig. Unter Verwendung der in Kapitel 3 beschriebenen Realisierung von Warteschlangen ergibt sich ein Aufwand von $O((n + m) \log n)$ für einen Aufruf von `minSchnittPhase`. Insgesamt ergibt dies somit den Aufwand $O(nm \log n)$. ■

Unter Verwendung von Fibonacci-Heaps hat ein Aufruf von `minSchnittPhase` sogar nur einen Aufwand von $O(m + n \log n)$, da diese das Ändern von Prioritäten effizienter unterstützen. Dadurch verringert sich der Aufwand für `minSchnitt` auf $O(nm + n^2 \log n)$. Bis heute ist kein effizienterer Algorithmus für dieses Problem bekannt.

Mit diesem Algorithmus kann auch die Kantenzusammenhangszahl eines ungerichteten Graphen G bestimmt werden. Dazu wird jede Kante von G mit 1 bewertet. Dann sind die Kosten eines minimalen Schnittes von G gleich $Z^k(G)$. Der Aufwand für die Auswahl der Ecken mit maximalen Kosten innerhalb einer Phase kann hierbei sogar noch gesenkt werden. Zwar können die Bewertungen der Kanten in den einzelnen Phasen ansteigen, aber die Summe der Bewertungen aller Kanten des Graphen in jeder Phase ist maximal m . Hierbei ist m die Anzahl der Kanten von G . Die Steigerung der Effizienz beruht auf der Verwendung spezieller Datenstrukturen.

Die verschiedenen Kosten der Ecken aus $E \setminus U$ werden in einer doppelt verketteten Liste `kostenListe` in aufsteigender Reihenfolge gesammelt. Die Anzahl der Einträge dieser Liste ist somit maximal gleich der Anzahl der Ecken in $E \setminus U$. Ein Zeiger `kopf` zeigt jeweils

auf den letzten Eintrag von `kostenListe`. Er zeigt auf das Listenelement, welches die zur Zeit höchsten Kosten enthält.

Die Ecken von $E \setminus U$ werden nach Kosten getrennt in einem Feld `kostenFeld` von doppelt verketteten Listen gesammelt; d.h. `kostenFeld[i]` ist eine doppelt verkettete Liste mit den Ecken aus $E \setminus U$, welche Kosten i haben. Ferner gibt es noch zwei Felder `kostenListeZeiger` und `kostenFeldZeiger` von Zeigern. Sofern es eine Ecke in $E \setminus U$ mit Kosten i gibt, enthält `kostenListeZeiger[i]` einen Zeiger auf das entsprechende Element in der Liste `kostenListe`. Somit hat `kostenListeZeiger` $m+1$ Komponenten. Das zweite Feld enthält für jede Ecke, die in $E \setminus U$ ist, einen Zeiger auf das entsprechende Element in einer der Listen von `kostenFeld`. Des weiteren gibt es noch ein Feld `kosten`, welches für jede Ecke, die in $E \setminus U$ ist, die momentanen Kosten enthält.

Zu Beginn jeder Phase werden alle Ecken in einer doppelt verketteten Liste in `kostenFeld[0]` abgelegt. Hierbei ist die Ecke 1 an erster Stelle. Für jede Ecke e enthält `kostenFeldZeiger[e]` einen Zeiger zu dem entsprechenden Element in `kostenFeld[0]`. Die Liste `kostenListe` enthält nur ein Element für die Kosten 0. Auf dieses zeigen die Zeiger `kopf` und `kostenListeZeiger[0]`. Das Feld `kosten` enthält den Eintrag 0 für jede Ecke. Abbildung 7.20 zeigt einen Teil dieser Datenstruktur zu Beginn der ersten Phase.

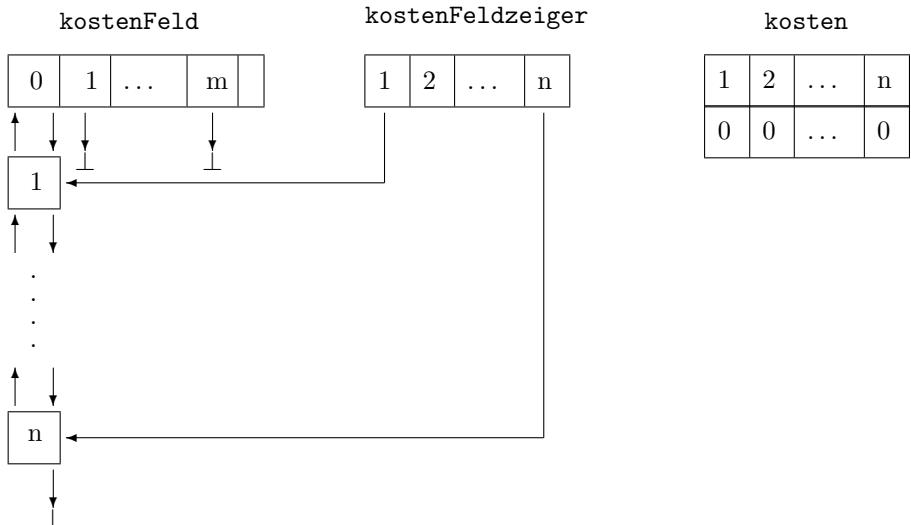


Abbildung 7.20: Die Datenstruktur nach der Initialisierung

Die Auswahl der Ecke aus $E \setminus U$ mit maximalen Kosten kann nun in konstanter Zeit erfolgen. Mit Hilfe des Zeigers `kopf` bekommt man die maximalen Kosten i . Eine zugehörige Ecke e steht an erster Stelle der Liste `kostenFeld[i]`. Das Einfügen dieser Ecke in U kann ebenfalls in konstanter Zeit durchgeführt werden: Die Ecke wird aus `kostenFeld[i]` entfernt; ist diese Liste nun leer, so wird das letzte Element aus der Liste `kostenListe` entfernt und der Zeiger `kopf` entsprechend geändert. Anschließend

müssen noch die Kosten der Nachbarn von e in $E \setminus U$ geändert werden. Für jeden solchen Nachbarn f ergeben sich die neuen Kosten aus der Bewertung der Kante von e nach f und den bisherigen Kosten $\text{kosten}[f]$ von f . Die neuen Kosten werden in $\text{kosten}[f]$ eingetragen. Mit Hilfe von $\text{kostenFeldZeiger}[f]$ wird nun der entsprechende Eintrag in kostenFeld gefunden und entsprechend angehoben. Danach zeigt $\text{kostenFeldZeiger}[f]$ wieder an die korrekte Stelle.

Nun muß noch die Liste kostenListe auf den neuen Stand gebracht werden. Zum einen kann es sein, daß die Ecke f die einzige Ecke in der Liste in kostenFeld war. In diesem Fall muß das entsprechende Element aus der Liste kostenListe entfernt werden. Mit Hilfe des Feldes kostenListeZeiger erfolgt dies in konstanter Zeit. Zum anderen kann es sein, daß es noch keine Ecke mit den neuen Kosten der Ecke f gibt. In diesem Fall muß die entsprechende Stelle in der Liste kosten gefunden werden. Dies erfolgt mittels einer sequentiellen Suche, startend an der alten Stelle in Richtung höherer Kosten. Dabei ist die Anzahl der Schritte durch die Bewertung der Kante e nach f beschränkt. Eventuell muß auch der Zeiger kopf abgeändert werden. Eine solche sequentielle Suche muß im ungünstigsten Fall für jede Kante des Graphen durchgeführt werden. Da aber die Summe der Bewertungen aller Kanten in jeder Phase maximal m ist, ist der Aufwand aller sequentiellen Suchen in einer Phase $O(m)$. Der Gesamtaufwand jeder Phase ist somit $= O(n + m)$. Daraus ergibt sich folgender Satz:

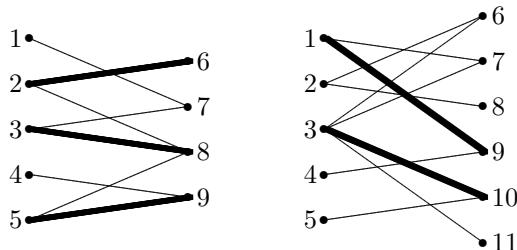
Satz. Die Kantenzusammenhangszahl eines ungerichteten Graphen kann mit Aufwand $O(nm)$ bestimmt werden.

7.6 Literatur

Die Bestimmung von maximalen Zuordnungen in bipartiten Graphen mit Aufwand $O(\sqrt{nm})$ wurde zuerst von J.E. Hopcroft und R.M. Karp beschrieben [68]. H. Alt et al. haben eine Realisierung dieses Algorithmus mit Aufwand $O(n^{1.5} \sqrt{m/\log n})$ entwickelt [5]. Für dichte Graphen ist dies eine Verbesserung um den Faktor $\sqrt{\log n}$. Der Zusammenhang von maximalen Zuordnungen und maximalen Flüssen auf Netzwerken wurde von S. Even und R.E. Tarjan erkannt [37]. Die Bestimmung von maximalen Zuordnungen auf beliebigen Graphen ist algorithmisch aufwendiger, aber die Zeitkomplexität ist ebenfalls $O(\sqrt{nm})$ [97]. Den Satz von Hall findet man in [55] und den von Menger in [96]. Der dort angegebene Beweis basiert aber nicht auf maximalen Flüssen; der hier angegebene Beweis stammt von G.B. Dantzig und D.R. Fulkerson [27]. Den Satz von Whitney findet man in [122]. Der Algorithmus, welcher entscheidet, ob für einen gegebenen Graphen $Z^e(G) \geq z$ ist, stammt von S. Even [38]. J. Hopcroft und R.E. Tarjan haben einen Algorithmus entwickelt, welcher in linearer Zeit feststellt, ob $Z^e(G) \geq 3$ ist [69]. Die Ergebnisse über kantendisjunkte Wege und der Zusammenhang zu maximalen Flüssen stammen von L.R. Ford und D.R. Fulkerson [42]. Der Algorithmus von Stoer und Wagner zur Bestimmung eines minimalen Schnittes ist in [116] beschrieben. Aufgabe 9 ist [54] entnommen.

7.7 Aufgaben

- * 1. Es sei G ein bipartiter Graph mit Zuordnung Z und f der zu Z gehörende Fluß auf N_G . Welche Eigenschaft haben die Erweiterungswege bezüglich f auf N_G ? Folgern Sie daraus, daß eine Zuordnung Z eines bipartiten Graphen genau dann maximal ist, wenn es keinen Weg W gibt, dessen Kanten abwechselnd aus Z bzw. nicht aus Z sind, und dessen erste und letzte Ecke auf keiner Kante von Z liegt. Solche Wege nennt man *Erweiterungswege* bezüglich Z . Verallgemeinern Sie die Aussage auf beliebige ungerichtete Graphen (vergleichen Sie Aufgabe 9 aus Kapitel 9 auf Seite 328).
- * 2. Es sei G ein ungerichteter Graph mit $2n$ Ecken, so daß der Eckengrad jeder Ecke mindestens n ist. Beweisen Sie, daß G eine vollständige Zuordnung besitzt. Geben Sie einen Algorithmus mit Laufzeit $O(m)$ zur Bestimmung einer vollständigen Zuordnung an.
- 3. Bestimmen Sie für die folgenden beiden bipartiten Graphen maximale Zuordnungen. Die fett gezeichneten Kanten bilden jeweils schon Zuordnungen und können als Ausgangspunkt verwendet werden.



4. Bestimmen Sie eine maximale Zuordnung für den Petersen-Graph.
5. Es sei G ein Graph mit $\alpha(G) = 2$ und \overline{G} besitze eine maximale Zuordnung mit z Kanten. Beweisen Sie, daß $\chi(G) = n - z$ ist.
6. Es sei G ein kantenbewerteter, bipartiter Graph mit Eckenmenge $E = E_1 \cup E_2$ und k eine natürliche Zahl, so daß für alle Ecken $e \in E_1$ und $f \in E_2$ folgendes gilt:

$$g(e) \geq k \geq g(f)$$

Beweisen Sie, daß G eine Zuordnung Z mit $|Z| = |E_1|$ besitzt.

7. Es sei G ein ungerichteter Graph und Z eine maximale Zuordnung von G . Beweisen Sie, daß für jede nicht erweiterbare Zuordnung Z_1 von G folgendes gilt:

$$2|Z_1| \geq |Z|$$

8. Es sei G ein kantenbewerteter bipartiter Graph mit Eckenmenge $E = E_1 \cup E_2$. Entwerfen Sie einen effizienten Algorithmus, der feststellt, ob es eine Teilmenge D von E_1 mit $|D| > |N(D)|$ gibt, und gegebenenfalls eine solche Menge D bestimmt.

- * 9. Es sei G ein kantenbewerteter, vollständig bipartiter Graph mit Eckenmenge

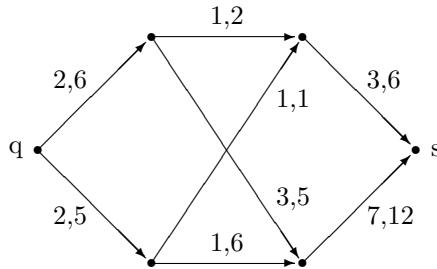
$$E = E_1 \cup E_2 \quad \text{und} \quad |E_1| = |E_2|.$$

Entwerfen Sie einen Algorithmus zur Bestimmung einer vollständigen Zuordnung mit minimalen Kosten von G .

Beweisen Sie zunächst folgende Aussage: Addiert man eine beliebige reelle Zahl b zu den Bewertungen der Kanten, welche zu einer Ecke inzident sind, so ändern sich auch die Kosten jeder vollständigen Zuordnung um b . Auf diese Art erreicht man, daß für jede Ecke $e \in E_1$ gilt: Die kleinste Bewertung unter den zu e inzidenten Kanten ist 0. Es sei nun G_0 der Teilgraph von G mit Eckenmenge E , der aus allen Kanten von G mit Bewertung 0 besteht. Mit Hilfe des in Abschnitt 7.1 angegebenen Algorithmus kann nun eine maximale Zuordnung von G_0 bestimmt werden. Ist diese Zuordnung für G vollständig, erübrigts sich eine weitere Berechnung. Andernfalls bestimmt man eine Teilmenge X von E_1 , so daß $|N_{G_0}(X)| < |X|$ gilt. Hierbei ist $N_{G_0}(X)$ die Menge der Nachbarn von X bezüglich des Graphen G_0 (vergleichen Sie Aufgabe 8). Es sei b_{min} die kleinste Bewertung aller Kanten in G mit Anfangsseite in X und Endecke in $E_2 \setminus N_{G_0}(X)$. Nun addiert man b_{min} zu jeder Kante, die zu einer Ecke aus $N_{G_0}(X)$ inzident ist, und $-b_{min}$ zu jeder Kante, die zu einer Ecke aus X inzident ist. Auf diese Art werden die Bewertungen der Kanten zwischen X und $N_{G_0}(X)$ nicht verändert, und eine neue Ecke wird durch eine Kante mit Bewertung 0 mit X verbunden. Nun wird wieder der Graph G_0 untersucht.

Beweisen Sie, daß man auf diese Art nach endlich vielen Schritten eine vollständige Zuordnung von G mit minimalen Kosten bekommt und daß die Anzahl der Durchgänge $O(n^2)$ ist.

- * 10. Geben Sie einen Algorithmus zur Bestimmung einer maximalen Zuordnung mit minimalen Kosten in einem kantenbewerteten bipartiten Graphen an. Ist jede nicht erweiterbare Zuordnung mit minimalen Kosten auch eine maximale Zuordnung? Kann der Algorithmus so erweitert werden, daß auch eine nicht erweiterbare Zuordnung mit minimalen Kosten bestimmt werden kann?
- 11. Es sei G ein Netzwerk mit unteren und oberen Kapazitätsgrenzen κ_u bzw. κ_o . Geben Sie einen Algorithmus an, welcher feststellt, ob G einen zulässigen Fluß besitzt und gegebenenfalls einen zulässigen Fluß mit minimalem Wert bestimmt. (Hinweis: Konstruieren Sie aus G ein neues Netzwerk, in dem die Richtungen aller Kanten umgedreht sind; die untere Kapazitätsgrenze ist $-\kappa_o$ und die obere Kapazitätsgrenze ist $-\kappa_u$. Betrachten Sie einen maximalen s - q -Fluß!)
- 12. Bestimmen Sie einen zulässigen Fluß mit minimalem Wert für das Netzwerk aus Abbildung 7.5.
- 13. Bestimmen Sie für das folgende Netzwerk mit oberen und unteren Kapazitätsgrenzen einen maximalen und einen minimalen zulässigen Fluß.



14. Es sei G ein Netzwerk mit unteren, aber ohne oberen Grenzen für den Fluß durch die Kanten. Unter welchen Voraussetzungen gibt es einen zulässigen Fluß auf G ?
15. Es sei G ein q - s -Netzwerk mit oberen und unteren Kapazitätsgrenzen, welches einen zulässigen Fluß besitzt. Die Kapazität eines Schnittes (X, \bar{X}) ist in diesem Fall wie folgt definiert:

$$\kappa(X, \bar{X}) = \sum_{\substack{k=(i,j) \in K \\ i \in X, j \in \bar{X}}} \kappa_o(k) - \sum_{\substack{k=(i,j) \in K \\ i \in \bar{X}, j \in X}} \kappa_u(k)$$

Beweisen Sie, daß der minimale Wert eines zulässigen Flusses auf G gleich

$$-\min \{ \kappa(\bar{X}, X) \mid (X, \bar{X}) \text{ } q\text{-}s\text{-Schnitt von } G \}$$

ist.

16. Es sei G ein ungerichteter zusammenhängender Graph mit Eckenmenge E . Es sei a eine beliebige Ecke von G . Dann gilt:

$$Z^k(G) = \min \{ Z^k(a, b) \mid b \in E, b \neq a \}$$

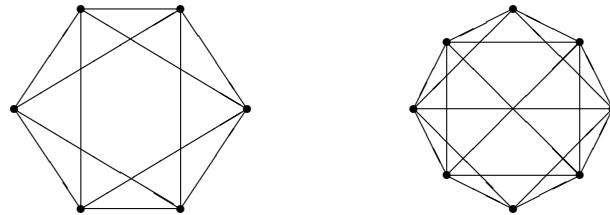
Wieso gilt eine analoge Aussage für $Z^e(G)$ nicht?

17. Es sei G ein ungerichteter zusammenhängender Graph. Beweisen Sie, daß für alle Ecken a, b, c von G folgende Ungleichung gilt:

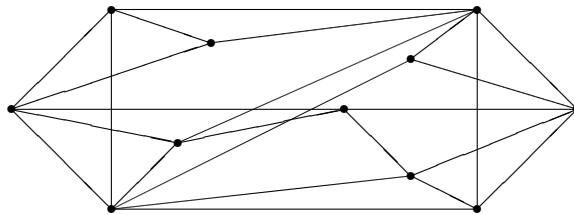
$$Z^k(a, b) \geq \min \{ (Z^k(a, c), Z^k(c, b)) \}$$

18. Es sei G ein ungerichteter zusammenhängender Graph. Für jedes Paar a, b von Ecken sei f_{ab} ein maximaler a - b -Fluß auf dem symmetrischen Netzwerk G_s . Beweisen Sie, daß es mindestens $n/2$ Eckenpaare gibt, so daß die Werte der entsprechenden maximalen Flüsse alle gleich sind.

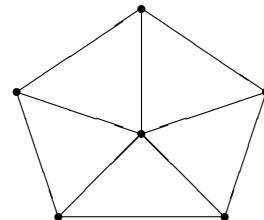
- * 19. Es sei Z eine Zuordnung eines bipartiten Graphen G . Beweisen Sie, daß es auch eine maximale Zuordnung Z' gibt, so daß jede Ecke von G , welche zu einer Kante von Z inzident ist, auch zu einer Kante von Z' inzident ist.
20. Bestimmen Sie die Ecken- und Kantenzusammenhangszahl des Petersen-Graphen.
21. Überprüfen Sie, ob für die folgenden Graphen die Eckenzusammenhangszahl mindestens 4 ist.



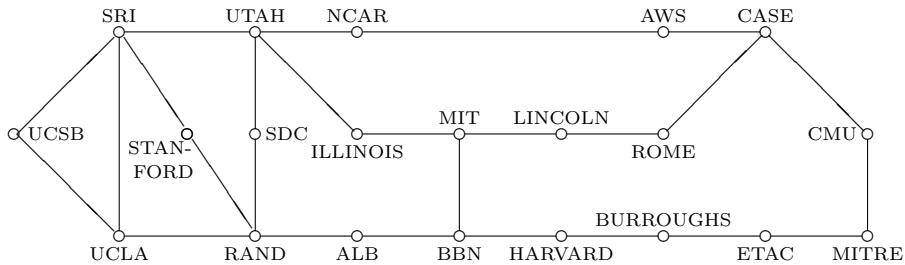
22. Bestimmen Sie die Ecken- und Kantenzusammenhangszahl des folgenden Graphen.



- * 23. Geben Sie einen Algorithmus an, welcher in linearer Zeit $O(m)$ feststellt, ob ein ungerichteter Graph G mindestens die Kantenzusammenhangszahl 2 hat. (Hinweis: Verwenden Sie die Tiefensuche, um aus G einen gerichteten Graphen G' zu machen. Die Richtung der Baumkanten ist von der hohen zur niedrigeren Tiefensuchenummer und die Richtung der anderen Kanten ist umgekehrt. Beweisen Sie: Genau dann ist $Z^k(G) \geq 2$, wenn in G' die Startecke von jeder anderen Ecke aus erreichbar ist.)
24. Aus einem Schachbrett wird ein weißes und ein schwarzes Feld entfernt. Kann man die übriggebliebene Fläche mit Rechtecken der Größe 1×2 vollständig abdecken, wenn die Größe eines Feldes auf dem Schachbrett 1×1 ist? Wie lautet die Antwort, wenn man aus dem Schachbrett zwei diagonal gegenüberliegende Eckfelder entfernt?
25. Es sei G ein ungerichteter Graph mit Eckenmenge E . Eine Teilmenge U von E heißt *Eckenüberdeckung* von G , falls jede Kante von G zu mindestens einer Ecke aus U inzident ist. Eine Eckenüberdeckung U heißt minimal, falls $|U| \leq |U'|$ für jede Überdeckung U' von G gilt.
- Beweisen Sie: Es sei M eine maximale Zuordnung und U eine minimale Überdeckung eines ungerichteten Graphen. Dann gilt $|M| \leq |U|$.
 - Geben Sie für den folgenden Graphen eine minimale Überdeckung und eine maximale Zuordnung an.



- c) Beweisen Sie den Satz von König-Egerváry: In einem bipartiten Graphen G ist die Anzahl der Kanten in einer maximalen Zuordnung gleich der Anzahl der Ecken in einer minimalen Überdeckung. (Hinweis: Wenden Sie den Satz von Menger auf das zu G gehörende Netzwerk N_G an.)
- d) Es sei G ein bipartiter Graph mit Eckenmenge $E = E_1 \cup E_2$. Beweisen Sie, daß G genau dann eine vollständige Zuordnung besitzt, falls jede Eckenüberdeckung mindestens $(|E_1| + |E_2|)/2$ Ecken enthält.
- * e) Es sei G ein bipartiter Graph und \bar{G} sein Komplement. Beweisen Sie, daß $\omega(\bar{G}) = \chi(\bar{G})$ gilt.
- f) Beweisen Sie, daß ein Untergraph G des vollständig bipartiten Graphen $K_{n,n}$ eine Zuordnung mit mindestens s Kanten hat, sofern er mehr als $(s - 1)n$ Kanten enthält.
26. Beweisen Sie, daß $Z^e(G) \leq 5$ für jeden planaren Graphen G gilt.
27. Bestimmen Sie die Zusammenhangszahlen der Graphen I_n für $n \geq 5$ aus Aufgabe 26 aus Kapitel 5.
28. In einem von dem Verteidigungsministerium der USA organisierten Forschungsprojekt wurde das ARPA-Kommunikationsnetz aufgebaut. Die experimentelle Phase begann 1969 mit einem Netz aus vier Knoten; 1988 bestand es bereits aus mehr als 20.000 Knotenrechnern und umspannte unter dem Namen *Internet* die USA, Westeuropa und Teile des Pazifiks. Auf der nächsten Seite ist die Topologie einer frühen Ausbaustufe abgebildet. Bestimmen Sie die Ecken- und Kantenzusammenhangszahl.



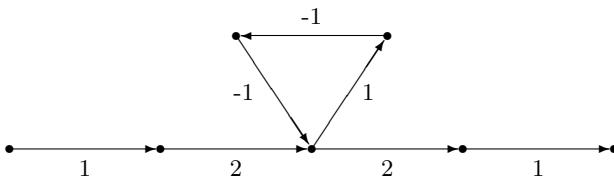
29. Es sei G ein regulärer Graph vom Grad g mit $Z^e(G) = 1$. Beweisen Sie, daß $Z^k(G) \leq g/2$ ist.
30. Es sei G ein ungerichteter Graph und Z eine maximale Zuordnung von G . Beweisen Sie, daß es in G einen Tiefensuchebaum B gibt, welcher jede Kante aus Z enthält.
31. Es sei B ein Baum. Wenden Sie die Tiefensuche auf B an und erzeugen Sie dabei auf folgende Art eine Zuordnung Z von B . Immer, wenn die Tiefensuche eine Ecke e verläßt, wird geprüft, ob e und der Vorgänger f von e noch nicht markiert sind. Ist dies der Fall, so werden die beiden Ecken markiert und die Kante $k = (f, e)$ in Z eingefügt. Beweisen Sie, daß Z eine maximale Zuordnung von B ist; d.h.

maximale Zuordnungen von Bäumen können in linearer Zeit bestimmt werden. Zeigen Sie, daß die so erzeugte Zuordnung Z folgende Eigenschaft hat: Ist i eine Ecke von B , welche zu keiner Kante aus Z inzident ist, und j der Vorgänger von i im Tiefensuchebaum, so existiert ein Nachfolger s von j , so daß (j, s) in Z ist.

- * 32. Es sei G ein ungerichteter Graph und \mathcal{U} die Menge aller Untergraphen von G ohne isolierte Ecken. Für jeden Untergraphen $U \in \mathcal{U}$ mit Eckenmenge E_U und Kantenmenge K_U definiere $f(U) = (|E_U| - 1)/|K_U|$. Es sei $M \in \mathcal{U}$ mit $f(M) = \max\{f(U) \mid U \in \mathcal{U}\}$. Beweisen Sie, daß eine der beiden folgenden Bedingungen gültig ist.
 - a) Die Kanten von M bilden eine maximale Zuordnung von G .
 - b) G ist bis auf isolierte Ecken ein *Sterngraph* (d.h. alle Kanten haben eine gemeinsame Ecke).
- 33. Ein Graph heißt *eindeutig färbar*, wenn jede minimale Färbung die gleiche Zerlegung der Eckenmenge bewirkt. Es sei G ein eindeutig färbarer Graph mit $\chi(G) = c$. Beweisen Sie folgende Aussagen:
 - a) Jede Ecke von G hat mindestens den Grad $c - 1$.
 - b) Der von den Ecken zweier beliebiger Farbklassen induzierte Untergraph ist zusammenhängend.
 - c) G ist $(c - 1)$ -fach zusammenhängend.
- * 34. Es sei G ein bipartiter Graph mit Eckenmenge $E = E_1 \cup E_2$ und t eine natürliche Zahl mit $t \leq |E_1|$. Beweisen Sie, daß G genau dann eine Zuordnung mit t Kanten hat, falls für jede Teilmenge T von E_1 folgende Ungleichung gilt: $|N(T)| \geq |T| + t - |E_1|$.

Kapitel 8

Kürzeste Wege



Ein wichtiges Anwendungsgebiet der Graphentheorie ist die Darstellung von Verkehrs- und Kommunikationsnetzen sowie die Bestimmung optimaler Wege in diesen. Dabei wird ein Weg als optimal betrachtet, wenn er der kürzeste, der billigste oder der sicherste ist. Die dargestellten Algorithmen bestimmen Wege, für welche die Summe der Bewertungen der Kanten minimal ist. Die Interpretation der Bewertungen bleibt der eigentlichen Anwendung überlassen. Es werden mehrere Varianten betrachtet: der kürzeste Weg zwischen zwei vorgegebenen Ecken, kürzeste Wege zwischen einer und allen anderen Ecken und kürzeste Wege zwischen allen Paaren von Ecken. Die dargestellten Verfahren lassen sich sowohl auf gerichtete als auch auf ungerichtete Graphen anwenden. Neben allgemeinen Verfahren werden auch solche diskutiert, welche nur auf Graphen mit speziellen Eigenschaften anwendbar sind. Als Anwendung werden Routingverfahren in Kommunikationsnetzen besprochen. In diesem Kapitel werden auch Algorithmen zur Bestimmung von kürzesten Wegen vorgestellt, wie sie in der künstlichen Intelligenz angewendet werden.

8.1 Einleitung

In diesem Kapitel werden kantenbewertete, gerichtete und ungerichtete Graphen betrachtet. Die Bewertungen $B[i, j]$ sind dabei immer reelle Zahlen. Sie können in verschiedenen Anwendungen verschiedene Bedeutungen tragen: Kosten, Zeitspannen, Wahrscheinlichkeiten oder Längen. Für das Verständnis der vorgestellten Algorithmen ist es vorteilhaft, die Bewertungen als Längen zu interpretieren.

Die Wegeplanung für Roboter aus Abschnitt 1.2 ist ein Beispiel hierfür. Es sind aber auch negative Bewertungen zugelassen. Deshalb liegt es nahe, die *Länge eines Kantenzuges* eines bewerteten gerichteten oder ungerichteten Graphen als die Summe der Längen der Kanten des Kantenzuges zu definieren. Hierbei wird der Begriff der Länge einer Kante für deren Bewertung verwendet. Ist k_1, k_2, \dots, k_s ein Kantenzug Z , so ist die Länge $L(Z)$ von Z durch

$$L(Z) = \sum_{i=1}^s L(k_i)$$

definiert, wobei $L(k_i)$ die Länge der Kante k_i ist. In Kapitel 2 wurde für zwei Ecken e, f eines unbewerteten Graphen der Abstand $d(e, f)$ definiert. Hierbei ist $d(e, f)$ die minimale Anzahl von Kanten eines Weges mit Anfangsecke e und Endecke f . Gibt es keinen solchen Weg, so ist $d(e, f) = \infty$, und es ist $d(e, e) = 0$ für jede Ecke e . Diese Definition läßt sich auf kantenbewertete Graphen übertragen: Der *Abstand* $d(e, f)$ zweier Ecken e, f ist gleich der minimalen Länge eines Weges mit Anfangsecke e und Endecke f . Gibt es keinen solchen Weg, so ist $d(e, f) = \infty$, und es ist $d(e, e) = 0$ für jede Ecke e . Diese Definition ist eindeutig, denn auf jeden Fall gibt es nur endlich viele Wege zwischen zwei Ecken, und somit existiert das Minimum. Es kann aber sein, daß es mehrere verschiedene *kürzeste Wege* zwischen zwei Ecken gibt. Ordnet man jeder Kante eines unbewerteten Graphen die Länge 1 zu, so stimmen die beiden Definitionen überein.

Bei negativen Kantenbewertungen kann es zu der Situation kommen, daß zwischen zwei Ecken e und f Kantenzüge existieren, deren Längen echt kleiner sind als die des kürzesten Weges von e nach f . Abbildung 8.1 zeigt eine solche Situation. Zwischen den Ecken e und f gibt es nur einen Weg, und dieser hat die Länge -2 ; somit ist $d(e, f) = -2$. Zwischen diesen beiden Ecken gibt es aber unendlich viele Kantenzüge. Diese haben die Längen $-2, -8, -14, \dots$. Die Ursache hierfür liegt darin, daß es einen geschlossenen Kantenzug mit negativer Gesamtlänge gibt. Dieser kann beliebig oft durchlaufen werden, wobei die Gesamtlänge jedesmal sinkt.

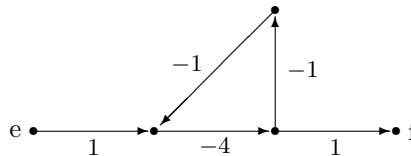


Abbildung 8.1: Ein Graph mit einem geschlossenen Kantenzug mit negativer Länge

Negative Kantenbewertungen kommen z.B. in Anwendungen vor, in denen die Bewertungen Gewinne und Verluste darstellen. In einem Graphen, in dem die Länge jedes geschlossenen Kantenzuges größer oder gleich 0 ist, kann diese Situation nicht auftreten. Insbesondere gilt:

Lemma. Es sei G ein kantenbewerteter Graph, bei dem die Länge jedes geschlossenen Kantenzuges größer oder gleich 0 ist. Es seien e und f Ecken von G und W ein kürzester

Weg von e nach f . Dann gilt für jeden Kantenzug Z von e nach f $L(W) \leq L(Z)$; d.h. es gibt einen kürzesten Kantenzug von e nach f , und dieser kann zu einem einfachen Weg zusammengezogen werden.

Beweis. Es sei Z ein Kantenzug von e nach f . Angenommen, Z ist kein Weg. Dann gibt es eine Ecke v , welche mehrmals auf W vorkommt. Es sei \bar{Z} der geschlossene Kantenzug aus Z vom erstmaligen bis zum zweitmaligen Besuch von v . Nach Voraussetzung ist $L(\bar{Z}) \geq 0$. Entfernt man \bar{Z} aus Z , so entsteht ein Kantenzug Z_1 von e nach f mit $L(Z_1) \leq L(Z)$. Auf diese Weise zeigt man, daß es zu jedem Kantenzug Z von e nach f einen einfachen Weg Z' von e nach f gibt, so daß $L(Z') \leq L(Z)$. Somit gilt $L(W) \leq L(Z)$. Da es einen kürzesten Weg von e nach f gibt, gibt es auch einen kürzesten Kantenzug, und dieser kann zu einem einfachen Weg zusammengezogen werden. ■

Im Rest des Kapitels wird folgende Bezeichnung verwendet: Ein kantenbewerteter Graph hat die *Eigenschaft (*)*, falls die Länge jedes geschlossenen Kantenzuges größer oder gleich 0 ist. Man beachte, daß ein ungerichteter, kantenbewerteter Graph genau dann die Eigenschaft (*) hat, wenn alle Bewertungen größer oder gleich 0 sind. Gibt es eine Kante mit negativer Bewertung, so entsteht ein geschlossener Kantenzug, wenn man diese Kante hin- und zurückläuft; die Gesamtlänge dieses Kantenzuges ist negativ. Nach dem letzten Lemma stimmen für Graphen mit der Eigenschaft (*) die Begriffe kürzester Kantenzug, kürzester Weg und kürzester einfacher Weg überein. Dies wird im folgenden ausgenützt.

Für die Bestimmung kürzester einfacher Wege in Graphen mit beliebigen reellen Bewertungen sind bis heute keine effizienten Algorithmen bekannt (vergleichen Sie Kapitel 9). In diesem Kapitel werden deshalb nur Graphen betrachtet, welche die Eigenschaft (*) haben. Ein Algorithmus, welcher testet, ob die Eigenschaft (*) erfüllt ist, wird in Aufgabe 28 beschrieben.

Im Mittelpunkt dieses Kapitels steht die Bestimmung kürzester Wege. Dieses Problem tritt in mehreren Variationen auf:

- (1) Kürzeste Wege zwischen zwei Ecken
- (2) Kürzeste Wege zwischen einer Ecke und allen anderen Ecken
- (3) Kürzeste Wege zwischen allen Paaren von Ecken

Bis heute ist kein Algorithmus für Problem (1) bekannt, dessen worst case Zeitkomplexität geringer ist als die von den effizientesten Algorithmen für Problem (2). Deshalb betrachten wir nur Algorithmen für Problem (2); diese lösen auch Problem (1). Problem (3) kann dadurch gelöst werden, daß man einen der Algorithmen für Problem (2) auf alle Ecken anwendet. Es gibt aber auch Algorithmen, welche unabhängig von Problem (2) arbeiten. Mit den in diesem Abschnitt vorgestellten Algorithmen können auch *längste Wege* bestimmt werden. Dazu muß nur das Vorzeichen der Bewertung von jeder Kante umgedreht werden. Voraussetzung für die Existenz eines längsten Weges ist, daß es keine geschlossenen Kantenzyklen mit positiver Länge gibt. Bevor konkrete Algorithmen diskutiert werden, werden zunächst Eigenschaften von kürzesten Wegen untersucht.

8.2 Das Optimalitätsprinzip

Im folgenden sei G immer ein kantenbewerteter Graph mit der Eigenschaft (*). Die Bewertung der Kante von i nach j wird mit $B[i, j]$ bezeichnet. Gibt es keine Kante von i nach j , so ist $B[i, j] = \infty$. Ferner ist $B[i, i] = 0$ für jede Ecke i . Analog zum Breitensuchebaum definiert man einen *kürzesten-Wege-Baum* (*kW-Baum*). Ein kW-Baum für eine Ecke s eines kantenbewerteten Graphen G ist ein gerichteter Baum B mit folgenden Eigenschaften:

- (1) Die Eckenmenge E' von B ist gleich der Menge der Ecken von G , welche von s aus erreichbar sind.
- (2) Ist G ein gerichteter Graph, so ist B ein Untergraph von G , und ist G ein ungerichteter Graph, so ist der zu B gehörende ungerichtete Baum ein Untergraph von G .
- (3) s ist eine Wurzel von B .
- (4) Für jede Ecke $e \in E'$ ist der eindeutige Weg von s nach e in B ein kürzester Weg von s nach e in G .

Die Ecke s nennt man die *Startecke*. Abbildung 8.2 zeigt einen kantenbewerteten ungerichteten Graphen und einen zugehörigen kW-Baum mit Startecke 1. Man beachte, daß kW-Bäume genauso wie kürzeste Wege nicht eindeutig bestimmt sind. Später werden wir beweisen, daß jeder Graph mit Eigenschaft (*) für jede Ecke einen kW-Baum besitzt.

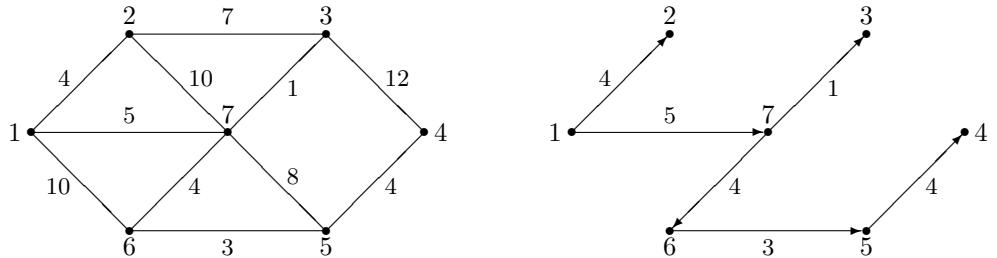


Abbildung 8.2: Ein ungerichteter Graph und ein kW-Baum

Die meisten Algorithmen für Problem (2) bauen einen kW-Baum auf. Als Datenstruktur für kW-Bäume eignet sich die in Abschnitt 3.3 vorgestellte Vorgängerliste für Wurzelbäume. Für den kW-Baum aus Abbildung 8.2 sieht diese wie folgt aus:

Ecke	1	2	3	4	5	6	7
Vorgänger	0	1	7	5	6	7	1

Sind für eine Startecke s die Entferungen $d(s, e)$ für alle erreichbaren Ecken e bekannt, so kann ein kW-Baum mit Aufwand $O(m)$ aufgebaut werden (vergleichen Sie Aufgabe 11). Somit genügt es, die Längen der kürzesten Wege zu bestimmen.

Grundlegend für die weiteren Betrachtungen ist das im nächsten Satz bewiesene *Optimalitätsprinzip*.

Satz (Optimalitätsprinzip). Es sei G ein kantenbewerteter Graph mit der Eigenschaft (*) und s, z Ecken von G . Ist e eine beliebige Ecke auf einem kürzesten Weg W von s nach z , so gilt:

$$d(s, z) = d(s, e) + d(e, z)$$

Beweis. Für $e = s$ oder $e = z$ ist das Optimalitätsprinzip trivial. Sei also $s \neq e \neq z$. Es sei W_1 der Teilweg von W von s nach e und W_2 der von e nach z . Dann gilt $L(W_1) \geq d(s, e)$, $L(W_2) \geq d(e, z)$ und $L(W_1) + L(W_2) = L(W)$. Angenommen, es gilt $L(W_1) > d(s, e)$. Es sei \bar{W}_1 ein kürzester Weg von s nach e . Dann bilden \bar{W}_1 und W_2 zusammen einen Kantenzug von s nach z mit der Länge $L(\bar{W}_1) + L(W_2)$. Nun gilt

$$L(\bar{W}_1) + L(W_2) < L(W_1) + L(W_2) = L(W).$$

Dies steht im Widerspruch zum letzten Lemma. Somit ist $L(W_1) = d(s, e)$. Analog zeigt man $L(W_2) = d(e, z)$. Somit gilt:

$$d(s, z) = L(W) = L(W_1) + L(W_2) = d(s, e) + d(e, z)$$

■

Man beachte, daß das Optimalitätsprinzip nur für Graphen mit der Eigenschaft (*) gilt. Abbildung 8.3 zeigt einen Graphen, welcher die Eigenschaft (*) nicht erfüllt. Der kürzeste Weg von 1 nach 5 hat die Länge 5. Betrachtet man die Ecke 3, so gilt $d(1, 3) = 2$ und $d(3, 5) = 2$. Somit ist das Optimalitätspzinzip für diesen Graphen nicht erfüllt.

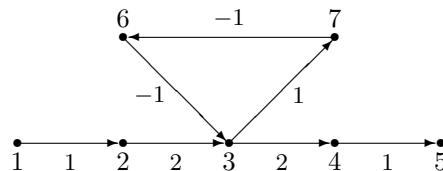


Abbildung 8.3: Ein Graph, der das Optimalitätsprinzip nicht erfüllt

Es sei W ein kürzester Weg von s nach z und e die Vorgängerecke von z auf diesem Weg. Dann besagt das Optimalitätsprinzip

$$d(s, z) = d(s, e) + B[e, z].$$

Hieraus folgt sofort, daß für alle Kanten (e, f)

$$d(s, f) \leq d(s, e) + B[e, f]$$

gilt. Diese beiden Aussagen können auch wie folgt zusammengefasst werden:

$$d(s, f) = \min \{d(s, e) + B[e, f] \mid e \text{ Vorgänger von } f \text{ in } G\}$$

Dieses Ergebnis erlaubt es, die Bestimmung kürzester Wege auf ein System von Gleichungen mit n Unbekannten zu reduzieren. Diese nennt man *Bellmansche Gleichungen*. Mit der Bezeichnung $d_i = d(s, i)$ für alle Ecken i und Startecke s gilt:

$$d_s = 0$$

$$d_j = \min \{d_i + B[i, j] \mid i = 1, \dots, n \text{ und } i \neq j\} \text{ für } j = 1, \dots, n \text{ und } j \neq s$$

Die Gültigkeit der Bellmanschen Gleichungen folgt sofort aus dem Optimalitätsprinzip. Später werden wir zeigen, daß ihre Lösung eindeutig ist. Die Bellmanschen Gleichungen führen direkt zu einem Algorithmus zur Bestimmung der Längen der kürzesten Wege von einer Startecke s zu allen anderen Ecken und zur Bestimmung eines entsprechenden kW-Baumes. Dazu wird ein Feld D der Länge n verwaltet. Für jede Ecke i enthält $D[i]$ eine obere Grenze für $d(s, i)$. Der kW-Baum wird in dem Feld **Vorgänger** der Länge n abgespeichert. In der Initialisierungsphase werden die Komponenten von D auf ∞ und die von **Vorgänger** auf 0 gesetzt. Ferner wird noch $D[s] = 0$ gesetzt. Dies macht die folgende Prozedur **initkW**:

```

procedure initkW (start : Integer);
var
    i : Integer;
begin
    for i := 1 to n do begin
        D[i] := infinity;      Vorgänger[i] := 0;
    end;
    D[start] := 0;
end

```

Dann werden die Ecken in einer beliebigen Reihenfolge betrachtet. Eine Ecke i heißt dabei *verwendbar*, wenn sie dazu genutzt werden kann, den Wert $D[j]$ eines Nachfolgers j zu erniedrigen; d.h. eine Ecke i ist verwendbar, falls es einen Nachfolger j mit

$$D[i] + B[i, j] < D[j]$$

gibt. In diesem Fall kann die obere Grenze für $d(s, j)$ von dem aktuellen Wert von $D[j]$ auf $D[i] + B[i, j]$ abgesenkt werden. Der Vorgänger von j auf einem kürzesten Weg ist die Ecke i . Hierbei beachte man, daß $a + \infty = \infty + a = \infty + \infty = \infty$ ist. Dazu wird folgende Prozedur **verkürze** verwendet:

```

procedure verkürze (i,j : Integer);
begin
  if D[i] + B[i,j] < D[j] then begin
    D[j] := D[i] + B[i,j];
    Vorgänger[j] := i;
  end
end

```

Alle in diesem Kapitel diskutierten Algorithmen verwenden die Prozedur `verkürze`. Die allgemeine Vorgehensweise ist die, daß man die Prozedur `verkürze` so lange anwendet, bis es keine verwendbaren Ecken mehr gibt. Die Algorithmen unterscheiden sich wesentlich darin, in welcher Reihenfolge die Verkürzungen erfolgen.

Abbildung 8.4 zeigt die Prozedur `bellman`, welche sich auf Graphen mit der Eigenschaft (*) anwenden läßt.

```

var D : array[1..max] of Real;
Vorgänger : array[1..max] of Integer;
procedure bellman (G : B-Graph; start : Integer);
var
  i,j : Integer;
begin
  initkW(start);
  while es gibt eine verwendbare Ecke i do
    for jeden Nachbar j von i do
      verkürze(i,j);
end

```

Abbildung 8.4: Die Prozedur `bellman`

Die Eigenschaft (*) bewirkt, daß die Prozedur `bellman` nach endlich vielen Schritten endet. Dazu wird das folgende Lemma benötigt:

Lemma. Wird zu irgendeinem Zeitpunkt der Wert $D[j]$ für eine Ecke j geändert, so führt der durch das Feld `Vorgänger` beschriebene Kantenzug von `start` nach j , und dieser Kantenzug ist ein einfacher Weg der Länge $D[j]$.

Beweis. Der Beweis erfolgt durch vollständige Induktion. Die erste Ecke j , für die $D[j]$ geändert wird, ist ein Nachfolger von `start`. In diesem Fall ist die Aussage trivialerweise erfüllt. Nun sei j eine Ecke, deren Wert $D[j]$ später geändert wird. Der neue Vorgänger von j sei i . Nach Induktionsvoraussetzung führt somit der Kantenzug von s nach j , und der Kantenzug von s nach i ist ein einfacher Weg der Länge $D[i]$. Angenommen, j kommt auf diesem Weg schon vor. Dann galt vor der letzten Änderung von $D[j]$:

$$D[i] + B[i,j] < D[j]$$

Somit ist $D[i] - D[j] + B[i,j] < 0$. Dies ist aber die Länge des geschlossenen Kantenzuges

durch j . Das steht aber im Widerspruch zur Eigenschaft (*). Somit ist der Kantenzug von `start` nach j ein einfacher Weg der Länge $D[j]$. ■

Mit diesem Lemma kann nun der folgende Satz bewiesen werden.

Satz. Für kantenbewertete Graphen mit der Eigenschaft (*) endet die Prozedur `bellmann` nach endlich vielen Schritten. Für jede von `start` erreichbare Ecke i gilt $D[i] = d(\text{start}, i)$. Das Feld `Vorgänger` beschreibt einen kW-Baum mit Wurzel `start`.

Beweis. Nach dem obigen Lemma beschreibt das Feld `Vorgänger` zu jedem Zeitpunkt einen Wurzelbaum mit Wurzel `start`. Da die Werte des Feldes `D` nur erniedrigt werden, sind diese Wurzelbäume alle verschieden. Da die Anzahl dieser Wurzelbäume beschränkt ist, terminiert die Prozedur nach endlich vielen Schritten.

Es sei i eine von `start` erreichbare Ecke. Nach Aufruf der Prozedur `bellman` gilt somit $D[i] \geq d(\text{start}, i)$. Es sei W ein kürzester Weg von `start` nach i . Die verwendeten Ecken seien $\text{start} = t_1, t_2, \dots, t_e = i$. Da es keine verwendbaren Ecken mehr gibt, gilt:

$$\begin{aligned} D[t_2] - D[s] &\leq B[s, t_2] \\ D[t_3] - D[t_2] &\leq B[t_2, t_3] \\ \vdots &\quad \vdots \\ D[i] - D[t_{e-1}] &\leq B[t_{e-1}, i] \end{aligned}$$

Summiert man diese Ungleichungen auf, so folgt wegen $D[s] = 0$:

$$D[i] \leq B[s, t_2] + B[t_2, t_3] + \dots + B[t_{e-1}, i] = L(W) = d(\text{start}, i) \leq D[i]$$

Hieraus folgt $D[i] = d(\text{start}, i)$. Somit enthält das Feld `D` die Längen der kürzesten Wege, und `Vorgänger` beschreibt einen kW-Baum. ■

8.3 Der Algorithmus von Moore und Ford

Bisher wurde noch keine Aussage über die Effizienz des Algorithmus gemacht. Diese hängt sehr stark davon ab, in welcher Reihenfolge die Ecken betrachtet werden. Es kann nicht ausgeschlossen werden, daß die Laufzeit sogar exponentiell wächst. Bei sorgfältiger Auswahl der Ecken ist der obige Algorithmus allerdings effizient. Das im folgenden beschriebene Verfahren stammt von E.F. Moore und L.R. Ford. Dabei wird die Bearbeitung der Ecken auf mehrere Durchgänge aufgeteilt. In jedem Durchgang werden nur die Ecken betrachtet, welche im vorhergehenden Durchgang markiert wurden. Dabei wird eine Ecke i markiert, falls der Wert von $D[i]$ geändert wurde. Vor Beginn des ersten Durchgangs wird die Startecke markiert. Für jede markierte Ecke i , die verwendbar ist, wird für alle Nachfolger j die Prozedur `verkürze` aufgerufen. Wird dabei der Wert von $D[j]$ geändert, so wird die Ecke j markiert. Am Ende des l -ten Durchgangs sind die neu markierten Ecken Nachfolger der Ecken, welche im $(l-1)$ -ten Durchgang markiert wurden.

Die Verwaltung der gerade markierten Ecken kann z.B. mittels einer Warteschlange erfolgen. In der Warteschlange befinden sich die noch zu bearbeitenden Ecken. Es wird immer die erste Ecke aus der Warteschlange entfernt und bearbeitet. Es muß noch beachtet werden, daß eine Ecke nicht in die Warteschlange eingefügt werden muß, falls sie schon enthalten ist. Es kann aber sein, daß eine Ecke mehrmals, also in verschiedenen Durchgängen, in die Warteschlange eingefügt wird. Abbildung 8.5 zeigt eine Realisierung dieses Verfahrens. Die Prozedur `kürzesteWege` verwendet die Prozedur `verkürzeW`. Diese ist eine Erweiterung der Prozedur `verkürze`: Wird der Wert von $D[j]$ geändert, so wird j in die Warteschlange W eingefügt, sofern j noch nicht in W ist.

```

var D : array[1..max] of Real;
Vorgänger : array[1..max] of Integer;
procedure kürzesteWege (G : B-Graph; start : Integer);
var
    i,j : Integer;
    W : Warteschlange of Integer;
begin
    initkW(start);
    W.einfügen(start);
    while W ≠ ∅ do begin
        i := W.entfernen;
        for jeden Nachfolger j von i do
            verkürzeW(i,j);
    end
end

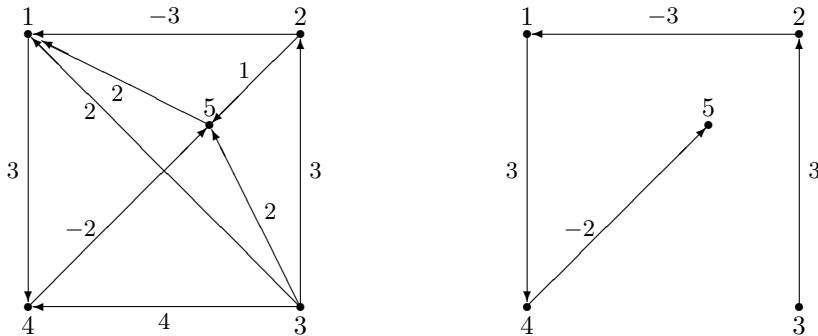
```

Abbildung 8.5: Die Prozedur `kürzesteWege`

Um die Korrektheit der Prozedur `kürzesteWege` zu beweisen, genügt es, zu zeigen, daß es am Ende keine verwendbare Ecke mehr gibt. Dazu beachte man, daß jede von `start` aus erreichbare Ecke mindestens einmal in W auftaucht. Ist eine Ecke i durch einen Weg, bestehend aus l Kanten, von `start` aus erreichbar, so wird sie spätestens im l -ten Durchgang in W eingefügt. Gibt es am Ende von `kürzesteWege` eine verwendbare Ecke i , so besitzt i einen Nachfolger j mit $D[i] + B[i,j] < D[j]$. Als sich die Ecke i zum letzten Mal in der Schlange befand, war der Wert von $D[j]$ eventuell sogar noch größer. Somit wurde beim Entfernen von i aus W der Wert von $D[j]$ auf $D[i] + B[i,j]$ gesetzt. Dieser Widerspruch zeigt die Korrektheit.

Bevor die Laufzeit untersucht wird, wird zuerst ein Beispiel diskutiert. Abbildung 8.6 zeigt links einen gerichteten Graphen und die einzelnen Schritte bei der Bestimmung der kürzesten Wege mit Startecke 3. Die erste Zeile zeigt den Zustand nach dem Aufruf von `initkW`. Die weiteren Zeilen geben den Zustand nach Bearbeiten der einzelnen Ecken an. Oben rechts ist der resultierende `kW`-Baum dargestellt.

Es muß noch die Zeitkomplexität des Algorithmus bestimmt werden. Ein Durchgang der **while**-Schleife untersucht alle Nachfolger einer Ecke, d.h. das erstmalige Abarbeiten



D					Vorgänger					W
1	2	3	4	5	1	2	3	4	5	\emptyset
∞	∞	0	∞	∞	0	0	0	0	0	3
2	3	0	4	2	3	3	0	3	3	$1 \leftarrow 2 \leftarrow 4 \leftarrow 5$
2	3	0	4	2	3	3	0	3	3	$2 \leftarrow 4 \leftarrow 5$
0	3	0	4	2	2	3	0	3	3	$4 \leftarrow 5 \leftarrow 1$
0	3	0	4	2	2	3	0	3	3	$5 \leftarrow 1$
0	3	0	4	2	2	3	0	3	3	1
0	3	0	3	2	2	3	0	1	3	4
0	3	0	3	1	2	3	0	1	4	5
0	3	0	3	1	2	3	0	1	4	\emptyset

Abbildung 8.6: Bestimmung der kürzesten Wege mittels kürzesteWege

aller Ecken hat zusammen die Komplexität $O(m)$. Im folgenden Lemma wird gezeigt, daß jede Ecke maximal n -mal in die Warteschlange eingefügt wird. Somit ergibt sich die Gesamtkomplexität von $O(mn)$.

Lemma. Kommt eine Ecke zum l -ten Mal in die Warteschlange W , so haben alle Ecken, deren kürzester Weg aus maximal $l - 1$ Kanten besteht, schon ihren endgültigen Wert in dem Feld D .

Beweis. Der Beweis wird mittels vollständiger Induktion geführt. Da nur die Startecke einen kürzesten Weg aus 0 Kanten besitzt, ist der Fall $l = 1$ trivial. Sei nun $l > 1$ und e eine Ecke, deren kürzester Weg aus $l - 1$ Kanten besteht. Sei f ein Vorgänger von e auf einem solchen kürzesten Weg. Nach dem Optimalitätsprinzip und Induktionsvoraussetzung hat $D[f]$ schon den endgültigen Wert, als eine Ecke zum $(l - 1)$ -ten Mal in W eingefügt wird. Zu diesem Zeitpunkt ist f schon zum letzten Mal in W eingefügt worden. Kommt f zum nächsten Mal an die Reihe, so erhält $D[e]$ seinen endgültigen Wert. ■

Bezeichnet man mit h die Tiefe des kW-Baumes, welchen die Prozedur **kürzesteWege** erzeugt, so ergibt sich aus dem letzten Lemma, daß der Aufwand der Prozedur $O(mh)$ ist. Wegen $h < n$ gilt folgender Satz:

Satz. Für einen kantenbewerteten Graphen mit Eigenschaft (*) lassen sich die kürzesten Wege und deren Längen von einer Startecke zu allen erreichbaren Ecken mit Aufwand $O(mn)$ bestimmen.

Zum besseren Verständnis des Aufwandes der vorgestellten Implementierung des Algorithmus von Moore und Ford wird eine Folge von Beispielen betrachtet. Abbildung 8.7 zeigt die Adjazenzmatrix eines gerichteten kantenbewerteten Graphen mit n Ecken. Man sieht leicht, daß der Graph die Eigenschaft (*) hat, denn jeder geschlossene Weg hat mindestens eine Kante mit Bewertung 2^n , da alle Kanten (i, j) mit $i < j$ die Bewertung 2^n haben. Die **while**-Schleife wird $1 + (n - 1)n/2$ mal durchlaufen (vergleichen Sie hierzu auch Aufgabe 24).

$$M_n = \begin{pmatrix} 0 & 2^n & 2^n & \dots & 2^n \\ -(2^0 + 1) & 0 & 2^n & \dots & 2^n \\ -(2^1 + 2) & -(2^1 + 1) & 0 & \dots & 2^n \\ -(2^2 + 3) & -(2^2 + 2) & -(2^2 + 1) & \dots & 2^n \\ \dots & \dots & \dots & \dots & \dots \\ -(2^{n-2} + n - 1) & -(2^{n-2} + n - 2) & -(2^{n-2} + n - 3) & \dots & 0 \end{pmatrix}$$

Abbildung 8.7: Die Adjazenzmatrix M_n

Der dargestellte Algorithmus kann leicht erweitert werden, so daß er sich auf beliebige kantenbewertete Graphen anwenden läßt und bei dem Vorhandensein von geschlossenen Wegen negativer Länge abbricht. Dazu beachte man, daß dieser Fall z.B. dadurch erkannt werden kann, daß eine Ecke zum n -ten Mal in S eingefügt werden soll. Es gibt aber auch Algorithmen, welche die Existenz geschlossener Kantenzüge schneller feststellen. Im nächsten Abschnitt wird gezeigt, daß die Zeitkomplexität verbessert werden kann, wenn die untersuchten Graphen spezielle Eigenschaften haben.

Der Algorithmus von Moore und Ford kann leicht erweitert werden, so daß er auch auf Graphen angewendet werden kann, welche nicht die Eigenschaft (*) haben. Dabei führt ein geschlossener Kantenzug mit negativer Länge zum Abbruch. Zur Vorbereitung dieser Erweiterung wird folgendes Lemma benötigt.

Lemma. Wird eine von der Startecke verschiedene Ecke zum l -ten Mal aus der Warteschlange W entfernt, so haben alle Ecken, deren kürzester Weg aus maximal l Kanten besteht, schon ihren endgültigen Wert in dem Feld D .

Beweis. Der Beweis wird mittels vollständiger Induktion nach l geführt. Nachdem die Startecke bearbeitet wurde, haben alle Ecken, deren kürzester Weg aus genau einer Kante besteht, schon ihren endgültigen Wert in D . Dies ist der Induktionsanfang. Sei nun $l > 1$ und e eine Ecke, deren kürzester Weg aus l Kanten besteht. Sei f ein

Vorgänger von e auf einem solchen kürzesten Weg. Nach dem Optimalitätsprinzip und Induktionsvoraussetzung hat $D[f]$ schon den endgültigen Wert, als eine Ecke zum $(l-1)$ -ten Mal aus W entfernt wird. Entweder hat auch $D[e]$ schon seinen endgültigen Wert oder f ist in der Warteschlange. Die Ecken in der Warteschlange sind alle maximal $l-2$ Mal schon entfernt worden. Wenn die erste Ecke zum l -ten Mal entfernt wird, sind diese Ecken alle schon entfernt und bearbeitet worden. Somit wurde auch f bearbeitet und $D[e]$ hat seinen endgültigen Wert. ■

Wendet man dieses Lemma für $l = n - 1$ an, so folgt daraus, daß keine Ecke n Mal in die Warteschlange eingefügt wird, sofern der Graph die Eigenschaft (*) erfüllt. Wendet man den Algorithmus von Moore und Ford auf Graphen an, welche nicht die Eigenschaft (*) haben, so muß mitgezählt werden, wie oft jede Ecke in die Warteschlange eingefügt wird. Wird eine Ecke zum n -ten Mal eingefügt, so muß ein geschlossener Weg negativer Länge vorliegen. Diesen Weg findet man leicht aus dem bisher aufgebauten Teil des kW-Baumes. Die in Abbildung 8.7 angegebenen Graphen M_n zeigen, daß eine Ecke wirklich $n - 1$ Mal in die Warteschlange eingefügt werden kann.

8.4 Anwendungen auf spezielle Graphen

In diesem Abschnitt wird der Algorithmus von Moore und Ford auf Graphen mit speziellen Eigenschaften angewendet. Es wird sich zeigen, daß die Zeitkomplexität in diesen Fällen niedriger ist. Wesentlich ist dabei, daß die Reihenfolge, in der die Ecken bearbeitet werden, problembezogen erfolgt. Auch in diesem Abschnitt wird vorausgesetzt, daß alle Graphen die Eigenschaft (*) haben.

8.4.1 Graphen mit konstanter Kantenzahl

Tragen alle Kanten die gleiche positive Bewertung b , so sind die Wege mit minimalen Kantenzahlen auch die kürzesten Wege. Die Länge eines Weges ist gleich dem Produkt von b und der Anzahl der Kanten des Weges. Somit kann ein Verfahren aus Kapitel 4 herangezogen werden: die Breitensuche. Der Breitensuchebaum ist ein kW-Baum; die Zeitkomplexität ist $O(m)$. Die Breitensuche ist somit ein Spezialfall des Algorithmus von Moore und Ford. Jede Ecke kommt genau einmal in die Warteschlange.

8.4.2 Graphen ohne geschlossene Wege

Ein ungerichteter Graph ohne geschlossene Wege ist ein Wald. Von jeder Ecke gibt es genau einen Weg zu jeder erreichbaren Ecke. Somit ist jeder Weg gleichzeitig ein kürzester Weg. Jede Zusammenhangskomponente bildet einen kW-Baum.

Ein gerichteter Graph ohne geschlossene Wege besitzt eine topologische Sortierung. Diese läßt sich in linearer Zeit bestimmen. Betrachtet man die Ecken in der Reihenfolge aufsteigender topologischer Sortierungsnummern, so muß jede Ecke nur einmal bearbeitet werden. Somit können auch in diesem Fall die kürzesten Wege in worst case Zeit $O(m)$ bestimmt werden (vergleichen Sie Aufgabe 8). Man beachte, daß dieser Algorithmus optimal ist, denn jede der m Kanten muß betrachtet werden. Würde eine

Kante nicht betrachtet werden, so könnte man die Bewertung dieser Kante ändern, und dadurch würde der Algorithmus ein falsches Ergebnis liefern.

8.4.3 Graphen mit nichtnegativen Kantenzahlbewertungen

Im Falle von nichtnegativen Kantenzahlbewertungen ist trivialerweise die Eigenschaft (*) erfüllt. Durch eine geschickte Auswahl der Ecken erreicht man, daß auch in diesem Fall jede Ecke nur einmal betrachtet werden muß. Die Auswahl der als nächstes zu bearbeitenden Ecke kann aber nicht in konstanter Zeit erfolgen, so daß die Komplexität $O(m)$ nicht erreicht wird. Der im folgenden vorgestellte Algorithmus stammt von E.W. Dijkstra und arbeitet ähnlich wie der Algorithmus von Prim. Zur Auswahl der Ecken wird dabei eine Menge B von Ecken verwaltet, für welche schon ein Weg von der Startecke aus bekannt ist. Diese Wege sind aber noch nicht unbedingt die kürzesten Wege. Ein kW-Baum wird schrittweise aufgebaut. In jedem Schritt wird eine Ecke aus B entfernt und bearbeitet. Eine Ecke i ist in B , falls $D[i] \neq \infty$, und falls noch nicht sicher ist, ob $D[i] = d(start, i)$. Ist E die Menge aller Ecken, so gilt zu jedem Zeitpunkt für jede Ecke $i \in B$, $i \neq start$

$$D[i] = \min\{D[j] + B[j, i] \mid j \in E \setminus B\}$$

und für jede Ecke $i \in E \setminus B$

$$D[i] = \infty \quad \text{oder} \quad D[i] = d(start, i).$$

Die Ecke $i \in B$ mit dem minimalen Wert $D[i]$ wird jeweils als nächstes bearbeitet und aus B entfernt. Die Nachbarn j von i mit $D[j] = \infty$ werden in B eingefügt. Für alle Nachbarn j von i wird dann `verkürze(i, j)` aufgerufen. Abbildung 8.8 zeigt eine Realisierung dieses Algorithmus. Die beiden Felder `D` und `Vorgänger` haben die gleiche Bedeutung wie bisher.

Wird der minimale Wert im Feld `D` von mehreren Ecken aus B gleichzeitig angenommen, so wähle man von diesen eine beliebige Ecke aus. Bevor wir die Korrektheit des Verfahrens beweisen, wird zunächst ein Beispiel diskutiert. Abbildung 8.9 zeigt den ungerichteten Graphen, welcher schon in Abschnitt 8.2 betrachtet wurde. Die Tabelle zeigt die Werte von `B`, `D` und `Vorgänger` in jedem Schritt. Die Startecke ist Ecke 1. Ferner sind auch jeweils die Ecken aus B angegeben, welche den minimalen Wert in dem Feld `D` haben. Die letzte Zeile gibt die Längen der kürzesten Wege an, und das Feld `Vorgänger` beschreibt einen kW-Baum. Man vergleiche dazu nochmals Abbildung 8.2.

Die Korrektheit der Prozedur `dijkstra` ergibt sich sofort aus folgendem Lemma:

Lemma. Sei $i \in B$, so daß $D[i] \leq D[j]$ für alle $j \in B$. Dann gilt $D[i] = d(start, i)$.

Beweis. Zum Beweis werden die Ecken in der Reihenfolge betrachtet, in der sie aus B entfernt werden. Die erste Ecke, die aus B entfernt wird, ist die Startecke, und für diese gilt das Lemma trivialerweise. Es sei nun i eine Ecke aus B mit $D[i] \leq D[j]$ für alle Ecken j aus B . Der durch das Feld `Vorgänger` beschriebene Weg von `start` nach i hat die Länge $D[i]$. Dies wurde bereits im letzten Abschnitt bewiesen.

```

var D : array[1..max] of Real;
Vorgänger : array[1..max] of Integer;
procedure dijkstra (G : B-Graph; start : Integer);
var
    i,j : Integer;
    B : set of Integer;
begin
    initkW(start);
    B := {start};
    while B ≠ ∅ do begin
        wähle i ∈ B mit D[i] minimal;
        B.entfernen(i);
        for jeden Nachbar j von i do begin
            if D[j] = ∞ then
                B.einfügen(j);
                verkürze(i,j);
        end
    end
end

```

Abbildung 8.8: Die Prozedur dijkstra

Angenommen, es gibt einen Weg W von $start$ nach i mit

$$L(W) < D[i].$$

Dann muß es eine Ecke e auf W geben, welche noch nicht aus B entfernt wurde, denn für die aus B schon entfernten Ecken enthält D die Längen der kürzesten Wege von der Startecke. Sei nun x die erste Ecke auf W mit dieser Eigenschaft. Bezeichnet man mit W_1 , den Teil von $start$ bis zu x und mit W_2 den Rest. Sei y der Vorgänger von x auf W . Dann ist $D[y] = d(start, y)$ und somit $D[y] + B[y, x] \geq D[x]$. Die letzte Ungleichung gilt, da y schon aus B entfernt wurde. Somit gilt

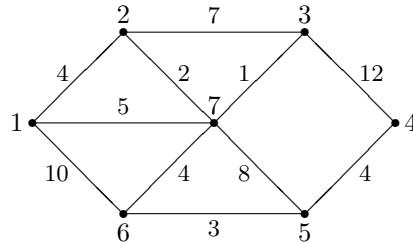
$$L(W) = L(W_1) + L(W_2) = D[y] + B[y, x] + L(W_2) \geq D[x] + L(W_2) \geq D[x].$$

Die Gültigkeit der letzten Ungleichung folgt aus der Voraussetzung, daß alle Kanten eine nichtnegative Bewertung tragen. Da $x \in B$ ist, gilt $D[x] \geq D[i]$. Daraus folgt

$$L(W) \geq D[x] \geq D[i] > L(W).$$

Dieser Widerspruch zeigt die Behauptung. ■

Das Lemma sagt aus, daß, nachdem eine Ecke i aus B ausgewählt und anschließend bearbeitet wurde, sie nicht wieder in B eingefügt wird. Somit wird jede Ecke nur einmal bearbeitet. Aus diesem Grund muß der Prozedurauftrag `verkürze(i, j)` nur für Ecken j mit $j \in B$ oder $D[j] = \infty$ erfolgen. Wird der kürzeste Weg von der Startecke zu einer



B	Min	D							Vorgänger						
		1	2	3	4	5	6	7	1	2	3	4	5	6	7
1		0	∞	∞	∞	∞	∞	∞	0	0	0	0	0	0	0
2,6,7	1	0	4	∞	∞	∞	10	5	0	1	0	0	0	1	1
6,7,3	2	0	4	11	∞	∞	10	5	0	1	2	0	0	1	1
6,3,5	7	0	4	6	∞	13	9	5	0	1	7	0	7	7	1
6,5,4	3	0	4	6	18	13	9	5	0	1	7	3	7	7	1
5,4	6	0	4	6	18	12	9	5	0	1	7	3	6	7	1
4	5	0	4	6	16	12	9	5	0	1	7	5	6	7	1
	4	0	4	6	16	12	9	5	0	1	7	5	6	7	1

Abbildung 8.9: Eine Anwendung der Prozedur `dijkstra`

bestimmten Ecke z gesucht (d.h. Problem (1)), so kann der Algorithmus abgebrochen werden, sobald z aus B entfernt wird.

Die Laufzeit der Prozedur `dijkstra` hängt vor allem davon ab, wie die Speicherung der Menge B und die Suche der Ecke i aus B mit minimalem $D[i]$ gelöst werden. Bei der einfachsten Lösung erfolgt die Speicherung der Menge B mittels eines Feldes der Länge n . Die Suche erfolgt sequentiell. Somit kann dieser Teil mit einem Aufwand von $O(n)$ realisiert werden. Einfügen und Entfernen von Ecken aus B erfordert den gleichen Aufwand. Da jede Ecke maximal einmal in B eingefügt wird, d.h. auch maximal einmal bearbeitet wird, ergibt sich ein Gesamtaufwand von $O(n^2)$.

Betrachtet man die Operationen, welche auf B angewendet werden, so bietet sich ein Heap an. Die Datenstruktur Heap, wie sie in Abschnitt 3.5 vorgestellt wurde, unterstützt die notwendigen Operationen: Einfügen von Elementen, kleinstes Element entfernen und Werte ändern. Man beachte, daß die letzte Operation notwendig ist, wenn ein Wert $D[i]$ einer Ecke i , die sich gerade im Heap befindet, geändert wird. Alle diese Operationen haben den Aufwand $O(\log n)$. Welcher Gesamtaufwand ergibt sich nun? Da jede Ecke maximal einmal eingefügt und entfernt wird, ergibt dies einen Aufwand von $O(n \log n)$. Da jede Ecke nur einmal betrachtet wird, ist die Anzahl der Änderungen von $D[i]$ durch die Anzahl der Vorgänger (bzw. Nachbarn im ungerichteten Fall) beschränkt; d.h. insgesamt sind maximal m Änderungen notwendig. Dies ergibt einen Aufwand von

$O(m \log n)$. Der Gesamtaufwand ist also ebenfalls $O(m \log n)$. Ist $m = O(n^2)$, so ist die erste Variante vorzuziehen. Aber für Graphen mit „wenigen“ Kanten ist die Realisierung mittels eines Heaps effizienter. Eine weitere Alternative für die Darstellung von B wird in Aufgabe 18 diskutiert. Es ergibt sich somit folgender Satz:

Satz. Für einen Graphen mit nichtnegativen Kantenbewertungen lassen sich die kürzesten Wege und deren Längen von einer Startecke zu allen erreichbaren Ecken mit Aufwand $O(n^2)$ bzw. $O(m \log n)$ bestimmen.

Um die Komplexität der auf einem Heap basierenden Realisierung weiter zu senken, ist es notwendig, die Änderungen des Wertes eines Elementes im Heap effizienter zu machen. Diese Operation wird im ungünstigsten Fall $O(m)$ -mal aufgerufen und hat die Komplexität $O(\log n)$. Verwendet man *Fibonacci-Heaps*, so können alle Änderungen zusammen mit Aufwand $O(m)$ durchgeführt werden. Die Realisierung dieser Datenstruktur ist recht aufwendig und wird deshalb hier nicht behandelt. Unter Verwendung von Fibonacci-Heaps hat der Algorithmus von Dijkstra eine worst case Laufzeit von $O(m + n \log n)$.

Zum Abschluß dieses Abschnitts soll anhand eines Beispiels demonstriert werden, daß der Algorithmus von Dijkstra bei negativen Kantenbewertungen nicht mehr korrekt arbeitet. Abbildung 8.10 zeigt einen gerichteten Graphen, bei dem einige Kanten eine negative Bewertung tragen. Der Graph hat die Eigenschaft (*). Der kürzeste Weg von Ecke 1 nach Ecke 4 besteht aus den Kanten $(1,2)$, $(2,3)$ und $(3,4)$ und hat die Länge 4. Wendet man den Algorithmus von Dijkstra auf diesen Graph an, so hätte der kürzeste Weg von 1 nach 4 die Länge 6 und bestünde aus der Kante $(1,4)$.

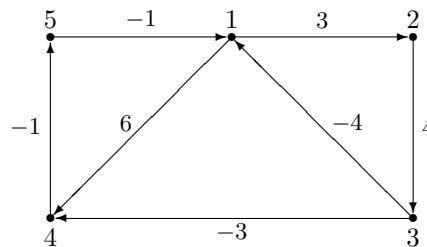


Abbildung 8.10: Ein gerichteter Graph mit negativen Kantenbewertungen

8.5 Routingverfahren in Kommunikationsnetzen

In Kommunikationsnetzen, wie z.B. einem Fernsprechnetz, sind zwei Datenstationen während des Datenaustausches ununterbrochen direkt miteinander verbunden. Computernetzwerke verwenden dagegen sogenannte Datenpaketdienste. Dabei steht der Sender einer Nachricht nicht unmittelbar mit dem Empfänger in Verbindung. Er übergibt vielmehr seine mit der Adresse des Empfängers versehene Nachricht dem Netzwerk,

welches diese selbständig an den Empfänger übermittelt. Dabei wird die Nachricht in kleine Einheiten, sogenannte Pakete, aufgeteilt, und diese werden unabhängig voneinander übertragen. Die Aufgabe, eine Menge von Datenpaketen in einem Netzwerk zu zustellen, nennt man ein *Routing-Problem*. Dabei kommt es darauf an, die Wege der Pakete so zu wählen, daß sich diese möglichst wenig überschneiden. Treffen zwei Pakete gleichzeitig in einer Station des Netzwerkes ein, so muß zunächst eines warten, bevor es weitergeleitet werden kann. Dadurch kann es zu größeren Verzögerungen kommen.

Routingverfahren können in zwei Gruppen aufgeteilt werden: statische und dynamische Verfahren. Ein statisches Routingverfahren verwendet Informationen über die einzelnen Teilstücke, um daraus die günstigsten Verbindungen zwischen den verschiedenen Stationen zu bestimmen. Dazu wird das Netzwerk als gerichteter bewerteter Graph modelliert. Hierbei bilden die Kapazitäten die Bewertungen der Kanten. Die günstigsten Wege können dann mit den in diesem Kapitel vorgestellten Verfahren bestimmt werden. Der Nachteil der statischen Verfahren ist, daß sie anfällig gegenüber dem Ausfall von Stationen und Verbindungen sind. Ferner werden unterschiedliche Netzbelastrungen nicht berücksichtigt. Nachrichten können unnötig verzögert werden, da sie stark ausgelastete Verbindungen verwenden, obwohl Alternativen zur Verfügung stehen. Aus diesem Grund sind dynamische Routingverfahren vorzuziehen. Diese bestimmen periodisch die günstigsten Verbindungen und passen sich dadurch an veränderte Netzwerkzustände und Belastungen an.

Ein dynamisches Routingverfahren mißt periodisch die Netzbelastrung, indem für jede direkte Verbindung die durchschnittliche Verzögerung der Pakete gemessen wird. Die Verzögerung eines Paketes ist dabei die Zeitspanne, welche ein Paket in einer Station verweilen muß. Das Netzwerk wird wiederum als gerichteter bewerteter Graph modelliert, wobei die durchschnittlichen Verzögerungen die Bewertungen bilden. Jede Station verwaltet eine Version dieses Graphen und sendet in periodischen Abständen die Werte der durchschnittlichen Verzögerungen der Verbindungen zu den direkten Nachbarn über das Netzwerk zu allen anderen Stationen. Diese bringen dann ihren Graphen auf den neuesten Stand. Dabei muß dafür gesorgt werden, daß dieser Datenaustausch im Vergleich zur Netzwerkkapazität gering bleibt.

Die Bestimmung der besten Verbindung erfolgt nun dezentral: Jede Station wendet einen entsprechenden Algorithmus auf ihren Graphen an. Dadurch entfällt auch die Abhängigkeit von einem zentralen Rechner. Die Bestimmung der günstigsten Verbindungen kann z.B. mittels des Algorithmus von Dijkstra erfolgen. Dabei wird ein kW-Baum für jede Station bestimmt. Jede Station speichert ihren kW-Baum nicht vollständig ab, sondern es wird nur eine sogenannte *Routingtabelle* verwaltet. In dieser Tabelle ist für jedes Ziel die Nachbarstation eingetragen, über die der günstigste Weg verläuft. Mittels der Routingtabellen leitet jede Station ein Paket immer an die richtige Nachbarstation. Abbildung 8.11 zeigt ein Netzwerk mit sieben Stationen, die entsprechenden durchschnittlichen Verzögerungen und einen kW-Baum für die Station 1.

Die Routingtabelle für Station 1 sieht dann wie folgt aus:

Ziel	2	3	4	5	6	7
Nachbar	2	2	5	5	5	5

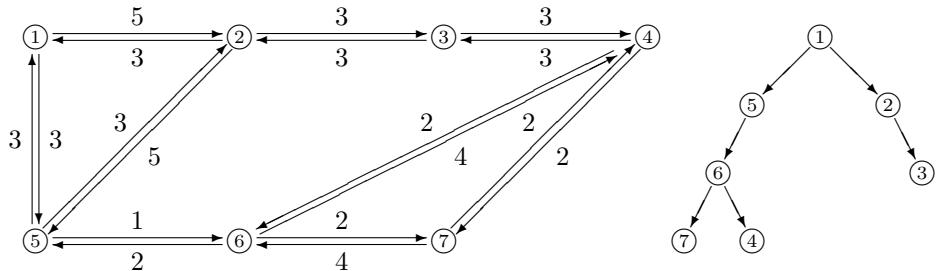


Abbildung 8.11: Ein mit den durchschnittlichen Verzögerungen bewertetes Netzwerk und ein kW-Baum für die Station 1

Soll ein Paket von Station 1 zu Station 4 transportiert werden, so leitet Station 1 das Paket an Station 5 weiter, die anhand ihrer eigenen Routingtabelle das Paket weiterleitet. Erhält eine Station die Information über eine geänderte durchschnittliche Verzögerung oder den Ausfall einer Station, so muß diese Station eine neue Routingtabelle bestimmen.

8.6 Kürzeste-Wege-Probleme in der künstlichen Intelligenz

Viele Probleme der künstlichen Intelligenz lassen sich als Suchprobleme formulieren. Der Suchraum kann dabei häufig durch einen Graphen dargestellt werden. In vielen Fällen ist das Ziel die Bestimmung eines kürzesten Weges von einer Startecke s zu einer Ecke aus einer vorgegebenen Menge von Zielecken. Beispiele hierfür sind Strategiespiele wie *Schach* oder *Dame*. Um bei einer Schachstellung einen Zug zu finden, der zu einer möglichst guten Stellung führt, müssen alle Züge, die der Gegner als nächstes machen kann, bedacht werden. Für jeden dieser Züge muß ein Antwortzug überlegt werden. Dieses Beispiel demonstriert sehr gut, wo die Schwierigkeiten bei vielen Problemen der künstlichen Intelligenz liegen: Die Anzahl der Ecken im Suchgraphen ist sehr groß. Nimmt man an, daß man in einer gegebenen Stellung etwa 25 verschiedene Züge ziehen kann und daß man fünf eigene und fünf gegnerische Züge vorausplanen will, so muß man

$$25^{10} \approx 9.5 \cdot 10^{13}$$

Stellungen überschauen. Ein weiteres Problem ist, daß die Suchgraphen in der Regel nur implizit vorliegen. Im Beispiel des Schachspiels sind die Nachfolger einer Ecke gerade die Stellungen, die durch einen legalen Zug erreicht werden können. Die Bestimmung dieser Stellungen führt zu zusätzlichem Aufwand bei den Suchverfahren. Ziel ist es, möglichst

schnell zu einem Zielknoten zu kommen, ohne dabei zu viele Stellungen zu bestimmen. Suchverfahren wie Breiten- oder Tiefensuche sind dafür zu aufwendig.

In diesem Abschnitt wird der *A^{*}-Algorithmus* vorgestellt. Dieser Algorithmus ist eine Verallgemeinerung des Algorithmus von Dijkstra und bestimmt den kürzesten Weg von einer Startecke s zu einer Ecke aus einer Menge Z von Zielecken. Dabei wird versucht, möglichst schnell zu einer Zielecke zu gelangen, d.h. die Anzahl der besuchten Ecken wird möglichst gering gehalten.

Das Hauptmerkmal des A^{*}-Algorithmus ist eine Schätzfunktion f . Für jede Ecke i ist $f(i)$ eine Abschätzung der Länge des kürzesten Weges von i zu einer Ecke aus Z . Mit Hilfe der Schätzfunktion f wird entschieden, welche Ecke als nächstes betrachtet wird. Es wird wieder schrittweise ein kW-Baum aufgebaut. Wie im Algorithmus von Dijkstra wird ein Feld D verwaltet: Für jede Ecke i enthält $D[i]$ eine obere Grenze für $d(s, i)$. Es wird eine Menge B von Ecken verwaltet, welche noch nicht in dem kW-Baum sind. In jedem Schritt wird eine Ecke aus B ausgewählt und in den kW-Baum eingefügt. Danach wird das Feld $D[i]$ entsprechend abgeändert. Die beiden Algorithmen unterscheiden sich nur in der Auswahl der Ecke i aus B . Der A^{*}-Algorithmus wählt $i \in B$ mit

$$D[i] + f(i) \leq \min\{D[j] + f(j) \mid j \in B\}$$

aus. Für jede Ecke i ist $D[i] + f(i)$ eine Abschätzung der Länge des Weges von s zu einer Ecke aus Z . Die Auswahl von i berücksichtigt also nicht nur den bisher zurückgelegten Weg, sondern auch die geschätzte Länge des Restweges. In vielen Fällen wird dadurch die Anzahl der zu betrachtenden Ecken gesenkt. Der erzielte Vorteil hängt stark von der Güte der verwendeten Schätzfunktion f ab.

Im folgenden wird davon ausgegangen, daß die Menge Z aus genau einer Ecke z besteht. Eine Erweiterung auf beliebige Mengen Z ist einfach. Ferner wird vorausgesetzt, daß die Bewertungen der Kanten positive reelle Zahlen sind. Der A^{*}-Algorithmus wird beendet, falls die Ecke z ausgewählt wird. Es muß zunächst gezeigt werden, daß der A^{*}-Algorithmus korrekt ist. Dazu ist es notwendig, eine Bedingung an die Schätzfunktion f zu stellen. Eine Schätzfunktion f heißt *zulässig*, wenn für jede Ecke i folgende Ungleichung gilt:

$$0 \leq f(i) \leq d(i, z)$$

Zulässige Schätzfunktionen unterschätzen immer die Länge eines kürzesten Weges zur Zielecke z . Somit gilt $f(z) = 0$.

Bei der Verwendung einer zulässigen Schätzfunktion bestimmt der A^{*}-Algorithmus einen kürzesten Weg von der Start- zur Zielecke. Allerdings kann es vorkommen, daß Ecken, nachdem sie aus B entfernt wurden, später wieder in B eingefügt werden. Aus diesem Grund muß die **for**-Schleife in der Prozedur **dijkstra** wie folgt verändert werden:

```
for jeden Nachbar j von i do begin
    verkürzeB(i, j)
```

Die Prozedur **verkürzeB** ist eine Erweiterung der Prozedur **verkürze**. Wird der Wert von $D[j]$ geändert, so wird j in die Menge B eingefügt, sofern j noch nicht in B

ist. Abbildung 8.12 zeigt eine Realisierung des A^* -Algorithmus. Hierbei bezeichnet f eine zulässige Schätzfunktion. Die Korrektheit des A^* -Algorithmus ergibt sich aus dem nachfolgenden Lemma.

```

var D : array[1..max] of Real;
    Vorgänger : array[1..max] of Integer;
procedure A*-Suche (G : B-Graph; start, ziel : Integer);
var
    i,j : Integer;
    B : set of Integer;
begin
    initkW(start);
    B := {start};
    while B ≠ ∅ do begin
        wähle i ∈ B mit D[i] + f(i) minimal;
        if i = ziel then
            exit('Ziel gefunden');
        B.entfernen(i);
        for jeden Nachbar j von i do
            if D[i] + B[i,j] < D[j] then begin
                D[j] := D[i] + B[i,j];
                Vorgänger[j] := i;
                if not B.enthalten(j) then
                    B.einfügen(j);
            end
        end
    end

```

Abbildung 8.12: Die Prozedur A^* -Suche

Lemma. Es sei f eine zulässige Schätzfunktion, s die Startecke und z die Zielecke. In dem Moment, in dem $z \in B$ und $D[z] \leq \min \{D[j] + f(j) \mid j \in B\}$ gilt, ist $D[z] = d(s, z)$.

Beweis. Es sei W ein kürzester Weg von s nach z . W verwende die Ecken $s = e_1, \dots, e_k = z$. Nach dem Optimalitätsprinzip gilt für $i = 1, \dots, k - 1$

$$d(e_i, z) = \sum_{j=i}^{k-1} B[e_j, e_{j+1}].$$

Nachdem s aus B entfernt wurde, gilt $D[e_2] = d(e_2, z)$. Es sei e_j die erste Ecke auf W , welche noch nicht aus B entfernt wurde. Dann gilt $D[e_i] = d(s, e_i)$ für $i = 1, \dots, j$. Es

sei W_1 der Teilweg von W von s nach e_j und W_2 der Restweg. Dann gilt:

$$\begin{aligned} L(W) &= L(W_1) + L(W_2) \\ &= d(s, e_{j-1}) + B[e_{j-1}, e_j] + L(W_2) \\ &= D[e_{j-1}] + B[e_{j-1}, e_j] + d(e_j, z) \\ &\geq D[e_j] + d(e_j, z) \\ &\geq D[e_j] + f(e_j) \\ &\geq D[z] \end{aligned}$$

Die erste Ungleichung gilt, seitdem e_{j-1} aus B entfernt wurde, und die zweite folgt aus der Zulässigkeit der Schätzfunktion f . Die letzte Ungleichung folgt aus der Voraussetzung. ■

Es gilt nun folgender Satz.

Satz. Es sei G ein Graph mit nichtnegativen Kantenbewertungen und f eine zulässige Schätzfunktion. Dann bestimmt der A^* -Algorithmus einen kürzesten Weg von der Startecke zur Zielecke.

Abbildung 8.13 zeigt einen kantenbewerteten Graphen und die Werte einer zulässigen Schätzfunktion f . Wendet man den A^* -Algorithmus auf diesen Graphen an (Startecke 1, Zielecke 4), so stellt man fest, daß Ecke 3, nachdem sie aus B entfernt wurde, später wieder in B eingefügt wird. Wird eine Ecke i aus B entfernt, so gilt in diesem Moment nicht notwendigerweise $D[i] = d(s, i)$. Diese Aussage gilt nur für die Zielecke.

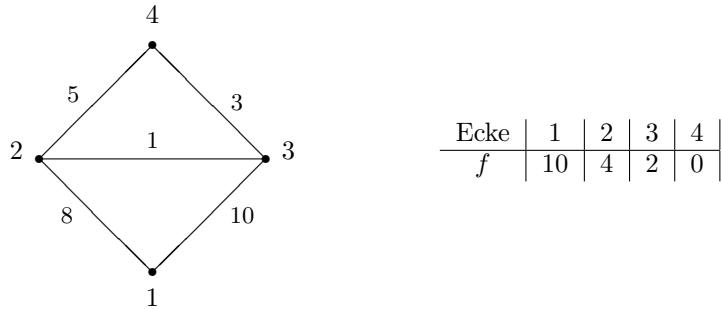


Abbildung 8.13: Ein kantenbewerteter Graph und eine zulässige Schätzfunktion

Eine Schätzfunktion heißt *konsistent*, wenn für jede Kante (i, j) folgende Ungleichung gilt:

$$0 \leq f(i) \leq B[i, j] + f(j)$$

Diese Ungleichung hat die Form einer Dreiecksungleichung. Ist f konsistent, so folgt, daß für alle Ecken i, j gilt:

$$f(i) \leq d(i, j) + f(j)$$

Wendet man diese Ungleichung für $j = z$ an, so folgt, daß eine konsistente Schätzfunktion f mit $f(z) = 0$ zulässig ist. Die in Abbildung 8.13 angegebene Schätzfunktion ist nicht konsistent. Bei der Verwendung einer konsistenten Schätzfunktion hat der A^* -Algorithmus die gleiche Eigenschaft wie der Algorithmus von Dijkstra. Dies wird in dem folgenden Lemma gezeigt.

Lemma. Es sei f eine konsistente Schätzfunktion. Ist $i \in B$ und $D[i] + f(i) \leq \min \{D[j] + f(j) \mid j \in B\}$, so gilt $D[i] = d(s, i)$.

Beweis. Der Beweis verläuft analog zu dem Beweis des entsprechenden Lemmas für den Algorithmus von Dijkstra. Es seien i, x, y und W wie oben gewählt, d.h. $L(W) < D[i]$ und $D[y] + B[y, x] \geq D[x]$. Dann gilt:

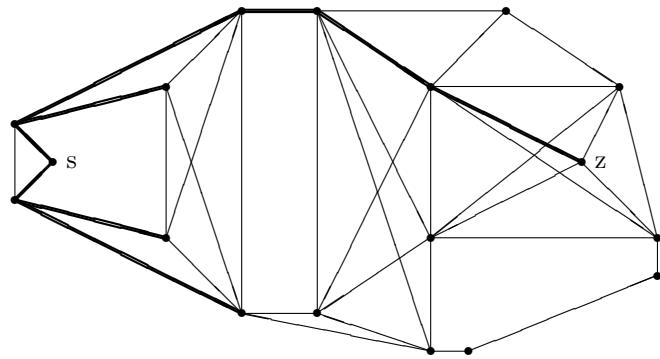
$$\begin{aligned} L(W) &= L(W_1) + L(W_2) \\ &= D[y] + B[y, x] + L(W_2) \\ &\geq D[x] + L(W_2) \\ &\geq D[x] + d(x, i) \\ &\geq D[x] + f(x) - f(i) \\ &\geq D[i] \\ &> L(W) \end{aligned}$$

Die erste Ungleichung gilt, da y schon aus B entfernt wurde. Die dritte Ungleichung folgt aus der Konsistenz von f , und die vorletzte gilt nach Voraussetzung. Dieser Widerspruch zeigt die Behauptung. ■

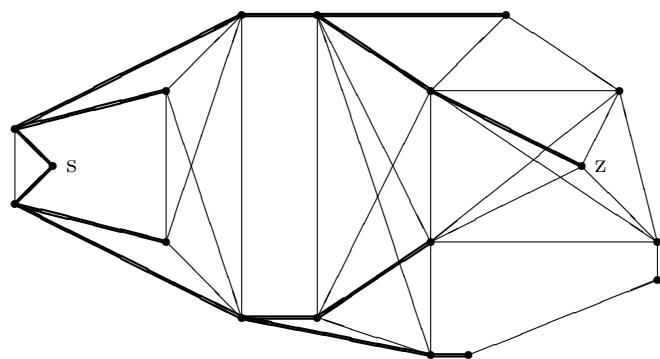
Das in Abbildung 8.12 dargestellte Programm kann bei Verwendung einer konsistenten Schätzfunktion vereinfacht werden. Bei der Iteration über die Nachbarn der ausgewählten Ecke wird eine Ecke j nur dann in B eingefügt, wenn $D[j] = \infty$.

Die Schwierigkeit bei der Anwendung des A^* -Algorithmus ist die Bestimmung einer guten zulässigen oder konsistenten Schätzfunktion. In Abschnitt 1.2 wurde das Problem der Wegplanung für Roboter diskutiert. Dabei wurde ein Graph in der Euklidischen Ebene betrachtet. Die Bewertungen der Kanten waren dabei die Längen der Kanten. Für jede Ecke i sei $f(i)$ der Abstand der Ecke i von der Zielecke z . Sind die Koordinaten von i und z bekannt, so läßt sich $f(i)$ leicht bestimmen. Die Konsistenz dieser Schätzfunktion folgt aus der Gültigkeit der Dreiecksungleichung. Abbildung 8.14 zeigt eine Anwendung des A^* -Algorithmus und des Dijkstra-Algorithmus auf das Wegplanungsproblem aus Abbildung 1.3. Die fett gezeichneten Kanten bilden jeweils den konstruierten kW-Baum. Man sieht, daß der A^* -Algorithmus schneller das Ziel z erreicht und dabei weniger Ecken besucht.

Wählt man als Schätzfunktion die Nullfunktion (d.h. $f(i) = 0$ für jede Ecke i), so sieht man, daß der Algorithmus von Dijkstra ein Spezialfall des A^* -Algorithmus ist. Hat die Schätzfunktion immer den genauen Wert (d.h. $f(i) = d(i, z)$ für alle Ecken i), so betrachtet der A^* -Algorithmus nur solche Ecken, welche auf einem kürzesten Weg zwischen s und z liegen.



A^* -Algorithmus



Algorithmus von Dijkstra

Abbildung 8.14: Bestimmung des kürzesten Weges von s nach z

8.6.1 Der iterative A^* -Algorithmus

Unglücklicherweise ist die Anwendung des A^* -Algorithmus auf sehr große Graphen, wie sie oft bei Problemen der künstlichen Intelligenz vorkommen, durch den verfügbaren Speicherplatz stark eingeschränkt. Eine explizite Speicherung der Vorgänger aller Ecken und die Verwaltung der Menge B erfordert in diesen Anwendungen sehr viel Speicherplatz, meist mehr, als die meisten Computer bereitstellen können. Die Breitensuche ist aus demselben Grund nicht anwendbar. In Abschnitt 4.11 wurde mit der iterativen Tiefensuche ein zur Breitensuche äquivalentes Suchverfahren vorgestellt, welches nicht unter dieser Einschränkung leidet. Eine ähnliche Vorgehensweise kann auch für den A^* -Algorithmus angewendet werden. Verschiedene Durchgänge entsprechen hierbei nicht steigenden Suchtiefen, sondern steigenden Kosten der Wege.

Der *iterative A^* -Algorithmus* arbeitet wie folgt: In jedem Durchgang wird eine Tiefensuche durchgeführt, welche nur solche Ecken e betrachtet, für die die Summe

$$D[e] + f(e)$$

eine vorgegebene Schranke B nicht überschreitet. Am Anfang ist $B = f(\text{start})$. Falls ein Durchgang nicht zum Ziel führt, wird ein neuer Durchgang mit einer höheren Schranke B gestartet. Der neue Wert für B ist gleich dem Minimum aller Summen $D[i] + f(i)$, welche im letzten Durchgang die Schranke überschritten. Die Suche kann beendet werden, sobald in einem Durchgang die Zielecke gefunden wurde. Man beachte, daß in jedem Durchgang mehr Ecken besucht werden. Für eine besuchte Ecke e gilt, daß die Länge des Weges von start nach e zusammen mit der geschätzten Länge des kürzesten Weges von e zur Zielecke maximal B ist.

Ist f eine zulässige Schätzfunktion, so findet der iterative A^* -Algorithmus immer den kürzesten Weg. Der Grund hierfür ist, daß der Wert der Schranke B nie die Länge des kürzesten Weges überschreitet. Irgendwann ist B gleich der Länge des kürzesten Weges. In diesem Durchgang wird die Zielecke gefunden.

Der iterative A^* -Algorithmus kann ähnlich zur iterativen Tiefensuche mittels eines Stapsels realisiert werden. Der benötigte Speicherplatz ist dabei abhängig von der Anzahl der Kanten des kürzesten Weges. Man beachte, daß der iterative A^* -Algorithmus keine Ecken untersucht, welche der normale A^* -Algorithmus nicht untersuchen würde.

Abbildung 8.15 zeigt eine Realisierung des iterativen A^* -Algorithmus. Hierbei ist f eine zulässige Schätzfunktion. Die Prozedur **iterativer- A^*** verwendet einen Stapel S . Auf ihm sind zu jedem Zeitpunkt die Ecken abgelegt, welche den Kantenzug von der Startecke zur aktuellen Ecke bilden. Diese Prozedur besteht aus zwei ineinander verschachtelten **while**-Schleifen. In jedem Durchgang der äußeren Schleife wird der Wert der Schranke B erhöht. Jeder Durchgang der inneren Schleife entspricht einer Tiefensuche, wobei nur Ecken e besucht werden, für die die Summe $D[e] + f(e)$ nicht die Schranke B übersteigt.

Um die Speicheranforderungen gering zu halten, werden nicht alle Nachfolger einer Ecke auf einmal betrachtet, sondern diese werden einzeln bearbeitet und eventuell auf den Stapel abgelegt. Dazu wird die Funktion **nächsterNachbar** verwendet. Sie liefert bei jedem Aufruf einen weiteren Nachbarn zurück. Nachdem alle Nachbarn einer Ecke

```

procedure iterativer-A* (G : B-Graph; start, ziel : Integer);
var
  j : Integer;
  B,BNeu,b : Real;
  e : Eintrag;
  S : stapel of Eintrag;
  gefunden : Boolean;
begin
  if start = ziel then
    exit('start und ziel identisch');
  Initialisiere gefunden mit false;
  B := f(start);
  while not gefunden and B < infinity do begin
    S.einfügen(new Eintrag(start,0));
    BNeu := infinity;
    while not gefunden and S ≠ ∅ do begin
      e := S.kopf;
      j := nächsterNachbar(e.ecke);
      if j ≠ 0 then begin
        if j = ziel and
          e.kosten+kosten(e.ecke,j) ≤ B then begin
          gefunden := true;
          ausgabe(j);
          while S ≠ ∅ do
            ausgabe(S.entfernen.ecke);
        end
        else begin
          b := f(j) + e.kosten+kosten(e.ecke,j);
          if b > B then
            BNeu := min {b, BNeu}
          else
            S.einfügen(new Eintrag(j,
              e.kosten+kosten(e.ecke,j)));
        end
      end
      else
        S.entfernen;
    end
    B := BNeu;
  end
  if not gefunden then
    ausgabe(ziel 'nicht erreichbar');
end

```

Abbildung 8.15: Eine Realisierung des iterativen A*-Algorithmus

zurückgeliefert wurden, wird der Wert 0 zurückgegeben. Dadurch wird erkannt, daß die Ecke vom Stapel entfernt werden kann. Der Vorteil dieser Vorgehensweise ist neben der Speicherplatz einsparung der, daß zu jedem Zeitpunkt die Ecken im Stapel von unten nach oben den Kantenzug von der Startecke zur aktuellen Ecke bilden.

Die Längen der Wege von der Startecke zur i -ten Ecke im Stapel werden nicht wie beim Algorithmus von Dijkstra in einem separaten Feld D verwaltet, sie werden zusammen mit den Ecken direkt im Stapel abgelegt. Hierzu wird folgende Datenstruktur verwendet:

```
Eintrag = record
    ecke : Integer;
    kosten : Real;
end
```

In jedem Durchgang der inneren **while**-Schleife wird der neue Wert für die Schranke B bestimmt. Dazu wird eine Variable B_{Neu} eingeführt, welche zu Beginn jedes Durchgangs auf ∞ gesetzt wird. Wird eine Ecke j nicht auf den Stapel abgelegt, weil ihr Wert die Schranke B übersteigt, so wird dieser mit B_{Neu} verglichen. Ist der Wert echt kleiner als B_{Neu} , so wird B_{Neu} aktualisiert. Wird das Ziel gefunden, so werden die Ecken des gefundenen Weges in umgekehrter Reihenfolge ausgegeben.

Bei ungerichteten Graphen sollte verhindert werden, daß von einer Ecke aus direkt wieder der Vorgänger besucht wird. Dies läßt sich leicht realisieren, da der Vorgänger jeder Ecke im Stapel genau unter der Ecke liegt. Man beachte, daß die Suche dabei trotzdem in geschlossene Wege geraten kann. Jedoch terminiert jeder Durchgang der inneren **while**-Scheife, denn für jede Ecke e eines geschlossenen Weges steigt die Länge $D[e]$ ständig an. Sofern die Zielecke von der Startecke aus erreichbar ist, terminiert der Algorithmus. Ist die Zielecke jedoch nicht erreichbar, so terminiert der Algorithmus nur, wenn keine geschlossenen Wege existieren. In einem solchen Fall wird irgendwann in der inneren **while**-Schleife keine neue Ecke mehr gefunden und dann wird B_{NEU} der Wert ∞ zugewiesen. Damit der Algorithmus auch dann terminiert, wenn das Ziel nicht erreichbar ist, kann zum Beispiel eine obere Grenze für den Wert der Schranke B vorgegeben werden. Übersteigt die Schranke diese Grenze, so wird die Suche abgebrochen. Ein sinnvoller Wert für diese Grenze ist eine obere Abschätzung für die Länge des Weges von **start** nach **ziel**.

Abbildung 8.16 zeigt links oben einen Raum, in dem ein Roboter sich von einem Startpunkt S zu einem Zielpunkt Z bewegen soll, ohne dabei ein Hindernis zu berühren. Der Raum ist in quadratische Zellen aufgeteilt, und die schwarzen Zellen bilden die Hindernisse. Der Roboter kann sich nur horizontal oder vertikal von Zelle zu Zelle bewegen. Um den kürzesten Weg zu finden, wird der iterative A^* -Algorithmus verwendet. Die Entfernung zwischen zwei benachbarten Zellen ist gleich 1. Die verwendete Schätzfunktion ordnet jeder Zelle die Anzahl der Zellen auf dem kürzesten Weg zum Ziel zu, ohne dabei die Hindernisse zu beachten. Dieser Wert läßt sich direkt aus den Koordinaten der Zielzelle und der aktuellen Zelle bestimmen. Die Bewertung der Startzelle ist in dem dargestellten Beispiel gleich 10. Die Nachbarzellen werden in der Reihenfolge Osten, Norden, Westen und Süden betrachtet.

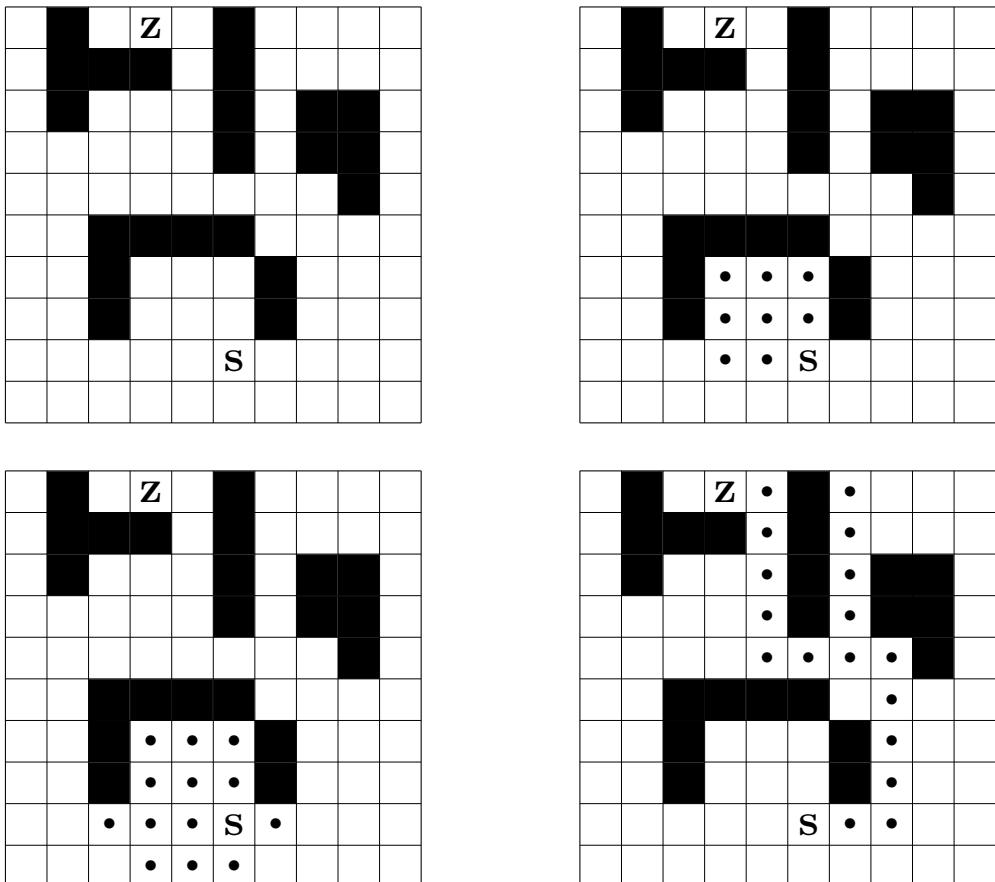


Abbildung 8.16: Eine Anwendung des iterativen A^ -Algorithmus*

In Abbildung 8.16 ist auch die Anwendung des iterativen A^* -Algorithmus dargestellt. Der Algorithmus benötigt für das dargestellte Beispiel drei Durchgänge. Die Werte der Schranke B sind dabei 10, 12 und 14. Für jeden Durchgang wurde der Raum dargestellt. Die Zellen, die im Verlauf der drei Durchgänge auf dem Stapel abgelegt werden, sind durch einen Punkt gekennzeichnet. Viele Zellen werden dabei im Verlauf eines Durchgangs mehrmals auf dem Stapel abgelegt. Man beachte, daß im ersten und zweiten Durchgang jeweils mehr Ecken auf dem Stapel abgelegt werden als im letzten Durchgang (unten rechts dargestellt).

8.6.2 Umkreissuche

Die Laufzeit des A^* -Algorithmus wird wesentlich durch die Auswertungen der Schätzfunktion bestimmt. Aus diesem Grund sollte die Anzahl der Auswertungen möglichst niedrig sein und ein einzelner Aufruf der Schätzfunktion sollte geringen Aufwand haben.

Häufig beeinflussen sich die beiden Größen gegenseitig: Qualitativ gute Schätzfunktionen führen schneller zum Ziel und ziehen deshalb weniger Auswertungen nach sich. Gleichzeitig steigt der Aufwand für die Auswertung einer Schätzfunktionen oft mit der Qualität der Schätzung an. Bei der in diesem Abschnitt vorgestellten Umkreissuche ist es möglich, die Qualität der Schätzfunktion einzustellen (mit entsprechenden Auswirkungen auf die Laufzeit). Der Vorteil besteht darin, daß in Abhängigkeit des Problems ein guter Mittelweg zwischen Aufwand und Qualität gefunden werden kann. Die Umkreissuche ist mit jeder konsistenten Schätzfunktion anwendbar.

Abbildung 8.17 verdeutlicht das Prinzip der Umkreissuche. In einem ersten Schritt wird der sogenannte Umkreis U_γ gebildet. Es handelt sich dabei um alle Ecken e des Graphen für die $d(\text{ziel}, e) \leq \gamma$ für eine feste Schranke $\gamma \geq 0$ gilt. Für jede Ecke aus U_γ wird in diesem Schritt auch der kürzeste Weg zum Ziel bestimmt. In einem zweiten Schritt wird dann der kürzeste Weg von der Startecke zu einer Ecke in U_γ bestimmt. Aus diesen beiden Teillösungen wird dann die Lösung für das eigentliche Suchproblem zusammengesetzt.

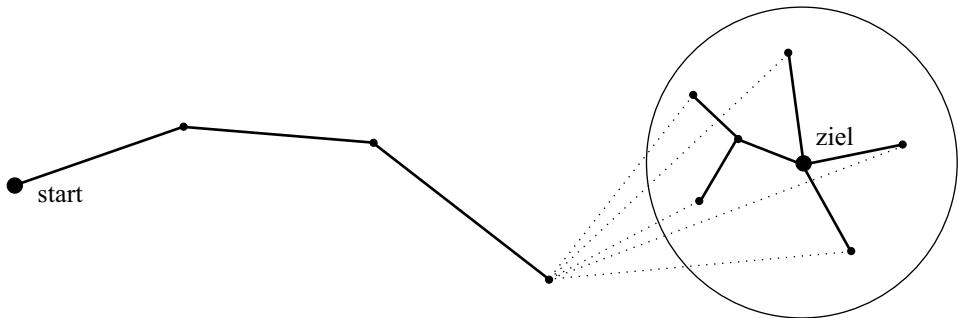


Abbildung 8.17: Das Prinzip der Umkreissuche

Für das Verfahren wird die Menge U_γ noch ausgedünnt. Hierzu werden alle Ecken entfernt, deren sämtliche Nachbarn in U_γ liegen. Diese Menge wird im folgenden ebenfalls mit U_γ bezeichnet. Die so gebildete Menge hat eine wichtige Eigenschaft: Jeder Weg von der Startecke zur Zielecke enthält mindestens eine Ecke aus U_γ . Es gilt $U_0 = \{\text{ziel}\}$.

Die Umkreissuche verwendet eine Schätzfunktion $f(e_1, e_2)$, welche die Länge des kürzesten Weges zwischen Paaren von Ecken abschätzt. Aus f wird eine neue Schätzfunktion $f_\gamma(e)$ gebildet, für Ecken e mit $d(e, \text{ziel}) > \gamma$ gilt:

$$f_\gamma(e) = \min_{u \in U_\gamma} (f(e, u) + d(u, \text{ziel}))$$

Hierbei bezeichnet $d(u, \text{ziel})$ die Länge des kürzesten Weges einer Ecke $u \in U_\gamma$ zur Zielecke. Es gilt $f_0(e) = f(e, \text{ziel})$.

Eine Schätzfunktion $f(e_1, e_2)$ heißt *monoton*, falls für jedes Paar e_1, e_2 von Ecken

- a) $0 \leq f(e_1, e_2) \leq \text{kosten}(e_1, e) + f(e, e_2)$ für alle Nachbarn e von e_1 ,

- b) $0 \leq f(e_1, e_2) \leq f(e_1, e) + \text{kosten}(e, e_2)$ für alle Nachbarn e von e_2

und $f(e, e) = 0$ für jede Ecke e gilt.

Folgendes Lemma beweist einige Eigenschaften von f_γ .

Lemma. Es sei f eine monotone Schätzfunktion und $\gamma \geq 0$. Dann gilt

- a) f_γ ist eine konsistente und zulässige Schätzfunktion und
- b) für $\gamma' \geq \gamma$ ist $f_{\gamma'}(e) \geq f_\gamma(e)$. D.h., mit steigendem γ steigt auch die Qualität der Schätzfunktion f_γ .

Beweis. Es sei $e = e_1, e_2, \dots, e_n = \text{ziel}$ ein kürzester Weg von e zur Zielecke und j minimal mit $e_j \in U_\gamma$.

- a) Da f monoton ist, gilt

$$\begin{aligned} d(e, \text{ziel}) &= d(e_1, e_j) + d(e_j, e_n) \\ &= f(e_1, e_1) + \text{kosten}(e_1, e_2) + \dots + \text{kosten}(e_{j-1}, e_j) + d(e_j, e_n) \\ &\geq f(e_1, e_2) + \text{kosten}(e_2, e_3) + \dots + \text{kosten}(e_{j-1}, e_j) + d(e_j, e_n) \\ &\geq \dots \\ &\geq f(e_1, e_{j-2}) + \text{kosten}(e_{j-2}, e_{j-1}) + \text{kosten}(e_{j-1}, e_j) + d(e_j, e_n) \\ &\geq f(e_1, e_{j-1}) + \text{kosten}(e_{j-1}, e_j) + d(e_j, e_n) \\ &\geq f(e_1, e_j) + d(e_j, e_n) \\ &\geq f_\gamma(e). \end{aligned}$$

Somit ist f_γ definiert auf der Menge $\{e \mid d(e, \text{ziel}) > \gamma\}$ eine Schätzfunktion. Aus der Monotonie von f folgt die Konsistenz und Zulässigkeit von f_γ .

- b) Ist $e_j \in U_{\gamma'}$, dann gilt $f_{\gamma'}(e) = f_\gamma(e)$. Ist $e_j \notin U_{\gamma'}$, so sei $i < j$ minimal mit $e_i \in U_{\gamma'}$. Dann ist

$$\begin{aligned} f_{\gamma'}(e) &= f(e, e_i) + d(e_i, \text{ziel}) \\ &\geq f(e, e_{i+1}) - \text{kosten}(e_i, e_{i+1}) + d(e_i, \text{ziel}) \\ &= f(e, e_{i+1}) + d(e_{i+1}, \text{ziel}) \\ &\geq \dots \\ &\geq f(e, e_j) + d(e_j, \text{ziel}) \\ &= f_\gamma(e). \end{aligned}$$

Die so gebildete Schätzfunktion f_γ kann sowohl mit dem A*- als auch mit dem iterativen A*-Algorithmus verwendet werden. Im folgenden wird die Kombination mit dem

iterativen A*-Algorithmus näher betrachtet. Dabei wird sich zeigen, daß eine Auswertung von f_γ mit weniger als $|U_\gamma|$ Auswertungen von f auskommt. Dies führt zu einer erheblichen Reduktion der Laufzeit.

Zunächst wird ein Durchgang des iterativen A*-Algorithmus betrachtet. Es sei B die aktuelle Schranke und $start = e_0, e_1, \dots, e_i$ ein Pfad mit Kosten $K(e_i)$. Definiere

$$g_\gamma(e_i) = K(e_i) + f_\gamma(e_i)$$

und

$$U_\gamma(e_i, B) = \{u \in U_\gamma \mid K(e_i) + f(e_i, u) + d(u, ziel) \leq B\}.$$

Ist $e_i \notin U_\gamma$, dann gilt $g_\gamma(e) > B$ genau dann, wenn $U_\gamma(e, B) = \emptyset$. Es sei $start = e_0, e_1, \dots, e_i$ die Menge der Ecken, welche bei einer Iteration auf dem Stapel liegen. Ist j ein Nachbar von e_i und $u \in U_\gamma$, so folgt aus der Monotonie der Schätzfunktion f

$$\begin{aligned} & K(e_i) + f(e_i, u) + d(u, ziel) \\ &= K(j) - kosten(e_i, j) + f(e_i, u) + d(u, ziel) \\ &\leq K(j) + f(j, u) + d(u, ziel). \end{aligned}$$

Somit gilt:

$$U_\gamma(j, B) \subseteq U_\gamma(e_i, B)$$

Dieses Ergebnis kann zur Reduktion der Auswertungen der Schätzfunktion f genutzt werden. Für jede Ecke e_i im Stapel wird die Menge $U_\gamma(e_i, B)$ abgespeichert. Wird für einen Nachfolger j der obersten Ecke e_i geprüft, ob $g_\gamma(e) > B$ gilt, so muß hierfür die Schätzfunktion f nur für Ecken aus $U_\gamma(e_i, B)$ ausgewertet werden. Dabei wird gleichzeitig die Menge $U_\gamma(j, B)$ bestimmt. Da die Mengen $U_\gamma(e_i, B)$ für die Ecken e_i des Stapels eine aufsteigende Kette bilden, kann die Speicherung aller Mengen mittels eines Feldes erfolgen. Hierzu wird für jede Ecke e_j lediglich der Index vermerkt, bei dem die erste Ecke der Menge abgelegt ist. Alle Ecken mit einem höheren Index gehören dann zu $U_\gamma(e_j, B)$. Bei der Bestimmung der Menge $U_\gamma(j, B)$ müssen lediglich die Ecken des Vorgängers e_i umgeordnet werden. Abbildung 8.18 zeigt diese Datenstruktur.

Die Umkreissuche auf der Basis des iterativen A*-Algorithmus folgt dem in Abbildung 8.15 dargestellten Programm und verwendet folgende Datenstrukturen.

```

Eintrag = record
  ecke : Integer;
  kosten : Real;
  anfang : Integer;
end

var Umkreis : array[1..max] of Integer;
  KostenUmkreis : array[1..max] of Real;
  VorgängerMatrix : array[1..max,1..max] of Integer;

```

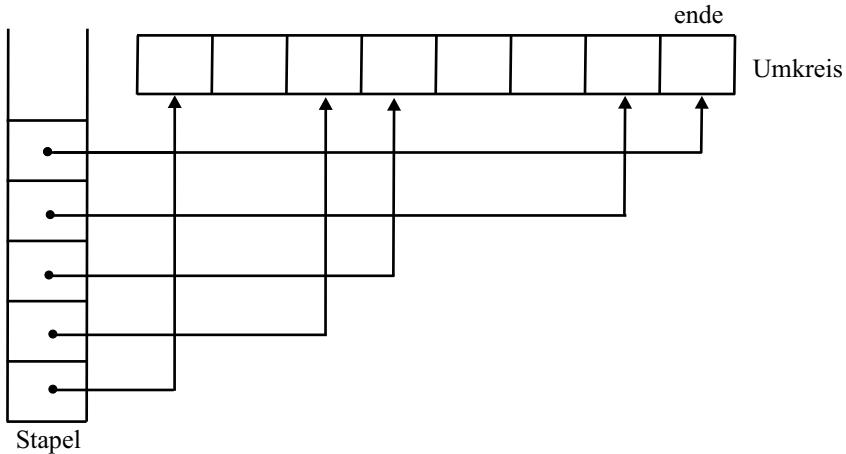


Abbildung 8.18: Verwaltung des sichtbaren Teils des Umkreises

Zu Beginn des Verfahrens wird für einen gegebenen Radius γ die Menge U_γ bestimmt. Hierfür wird der Algorithmus von Dijkstra verwendet. Die in Abbildung 8.8 vorgestellte Implementierung dieses Algorithmus kann abgebrochen werden, wenn eine Ecke i aus B entfernt wird, für die $D[i] > \gamma$ gilt. Danach werden die Ecken aus U_γ entfernt, welche keinen Nachbarn außerhalb von U_γ haben. Die Ecken aus U_γ werden in dem Feld **Umkreis** und die Entfernung dieser Ecken zur Zielecke werden im Feld **KostenUmkreis** gespeichert. Die kürzesten Wege von Ecken aus U_γ zur Zielecke können dem Feld **VorgängerMatrix** entnommen werden. Abbildung 8.19 zeigt die Funktion **redUmkreis**, mit ihr wird die Menge $U_\gamma(j, B)$ für einen Nachfolger j der aktuellen Ecke bestimmt.

Abbildung 8.20 zeigt eine Realisierung der Umkreissuche auf der Basis des iterativen A*-Algorithmus, man vergleiche den Code mit der in Abbildung 8.15 dargestellten Implementierung des iterativen A*-Algorithmus.

Praktische Untersuchungen haben gezeigt, daß der Wert von γ sorgfältig gewählt werden muß. Zwar sinkt mit steigendem γ die Anzahl der Auswertungen der Schätzfunktion, gleichzeitig steigt aber auch der Specheraufwand für U_γ an. Die Umkreissuche ist eine spezielle Ausprägung der bidirektionalen Suche. Dabei werden gleichzeitig zwei Suchläufe durchgeführt: Von der Startecke zur Zielecke und umgekehrt. Die Suche ist beendet, wenn sich die beiden Suchbäume treffen.

8.7 Kürzeste Wege zwischen allen Paaren von Ecken

In diesem Abschnitt werden Algorithmen für Problem (3) vorgestellt. Gegeben ist ein kantenbewerteter Graph, und gesucht sind die kürzesten Wege und deren Längen zwis-

```

function redUmkreis (kosten : Real; anfang,ende : Integer;
                     schranke : Real; var schrankeNeu : Real) : Integer;
var
    anfangNeu, i : Integer;
    b : Real;
begin
    i := anfang;
    anfangNeu := ende + 1;
    while i < anfangNeu do begin
        b := kosten + f(j,Umkreis[i]) + KostenUmkreis[i];
        if b ≤ schranke then begin
            vertausche(Umkreis[i],Umkreis[anfangNeu-1]));
            vertausche(KostenUmkreis[i],KostenUmkreis[anfangNeu-1]));
            anfangNeu := anfangNeu-1;
        end
        else begin
            i := i + 1;
            schrankeNeu := min {b, schrankeNeu}
        end
    end
    redUmkreis := anfangNeu;
end

```

Abbildung 8.19: Reduktion des sichtbaren Umkreises

schen allen Paaren von Ecken. Es wird weiterhin vorausgesetzt, daß alle Graphen die Eigenschaft (*) haben. Bei den Problemen (1) und (2) wurden zur Darstellung der kürzesten Wege kW-Bäume verwendet. Diese wurden mittels Vorgängerkeldern abgespeichert. Für das Problem (3) ergeben sich n verschiedene kW-Bäume. Diese lassen sich in einer $n \times n$ Matrix abspeichern. Dabei ist die i -te Zeile das Vorgängerkfeld des kW-Baumes mit Wurzel i . Eine solche Matrix nennt man *Vorgängermatrix*. Die Längen der kürzesten Wege werden ebenfalls in einer $n \times n$ Matrix abgespeichert, der Eintrag in der i -ten Zeile und j -ten Spalte ist gleich $d(i, j)$. Diese Matrix nennt man *Distanzmatrix*. Abbildung 8.21 zeigt einen kantenbewerteten Graphen, seine Distanz- und seine Vorgängermatrix.

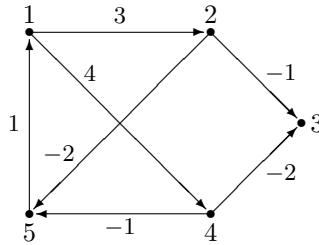
Eine Möglichkeit, das Problem (3) zu lösen, ist die n -fache Anwendung eines der in Abschnitt 8.4 vorgestellten Algorithmen. Für Graphen mit nichtnegativen Bewertungen erhält man so einen Algorithmus der Komplexität $O(nm \log n)$ bzw. $O(nm + n^2 \log n)$ unter Verwendung des Algorithmus von Dijkstra, basierend auf Heaps bzw. Fibonacci-Heaps. Tragen alle Kanten die gleiche positive Bewertung, so kann mit Hilfe der Breitensuche die Distanzmatrix mit Aufwand $O(nm)$ bestimmt werden. Für allgemeine Graphen mit der Eigenschaft (*) ergibt eine n -fache Anwendung des Algorithmus von Moore und Ford einen Aufwand von $O(n^2m)$. Durch einen kleinen Kunstgriff läßt sich der allgemeine Fall auf den Fall nichtnegativer Kantenbewertungen zurückführen. Dadurch erzielt man in beiden Fällen die gleiche worst case Komplexität. Die Grundidee

```

procedure umkreisSuche (G : B-Graph; start,ziel : Integer; radius : Real);
var
    j,anfang, ende : Integer;
    B,BNeu,b,kosten : Real;
    e : Eintrag;
    S : stapel of Eintrag;
    gefunden : Boolean;
begin
    if bestimmeUmkreis(G, start, ziel, radius) = true then
        exit('start innerhalb Umkreis');
    ende := Anzahl der Ecken in Umkreis;
    gefunden := false;
    B := f_U(start, Umkreis);
    while not gefunden and B < infinity do begin
        anfang := redUmkreis(0,1,ende,B,BNeu);
        S.einfügen(new Eintrag(start,0,anfang));
        BNeu := infinity;
        while not gefunden and S ≠ ∅ do begin
            e := S.kopf;
            j := nächsterNachbar(e.ecke);
            if j ≠ 0 then begin
                kosten := e.kosten + kosten(e.ecke,j);
                if e.ecke in Umkreis then begin
                    Bestimme Index k von j in Umkreis;
                    b := kosten + KostenUmkreis[k];
                    if b ≤ B then begin
                        gefunden := true;
                        ausgabe des gefundenen Weges;
                    end
                    else
                        BNeu := min {b, BNeu}
                end
                else begin
                    anfang := redUmkreis(kosten,anfang,ende,B,BNeu);
                    if anfang ≤ Ende then
                        S.einfügen(new Eintrag(j,kosten,anfang));
                end
            end
            else
                S.entfernen;
        end
        B := BNeu;
    end
    if not gefunden then
        ausgabe(ziel 'nicht erreichbar');
end

```

Abbildung 8.20: Realisierung der Umkreissuche auf der Basis des iterativen A*-Algorithmus



$$\begin{pmatrix} 0 & 3 & 2 & 4 & 1 \\ -1 & 0 & -1 & 3 & -2 \\ \infty & \infty & 0 & \infty & \infty \\ 0 & 3 & -2 & 0 & -1 \\ 1 & 4 & 3 & 5 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 2 & 1 & 2 \\ 5 & 0 & 2 & 1 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 1 & 4 & 0 & 4 \\ 5 & 1 & 2 & 1 & 0 \end{pmatrix}$$

Abbildung 8.21: Ein Graph und die zugehörige Distanz- und Vorgängermatrix

ist eine Transformation der Kantenbewertung B in eine Kantenbewertung B' mit ausschließlich nichtnegativen Werten, so daß die kürzesten Wege bezüglich B' auch kürzeste Wege bezüglich B und umgekehrt sind.

Die Konstruktion der neuen Bewertung stützt sich auf eine Funktion h , die jeder Ecke eine reelle Zahl zuordnet. Diese Funktion h muß die Eigenschaft haben, daß

$$h(x) + B[x, y] - h(y) \geq 0$$

für alle Kanten (x, y) gilt. Wie man eine solche Funktion findet, wird später gezeigt. Die neue Bewertung B' wird nun folgendermaßen gebildet: Für alle Kanten (x, y) setzt man

$$B'[x, y] = h(x) + B[x, y] - h(y).$$

Die Eigenschaft der Funktion h garantiert nun, daß $B'[x, y] \geq 0$ für alle Kanten (x, y) gilt. Das folgende Lemma zeigt die enge Beziehung der beiden Bewertungen:

Lemma. Es sei G ein kantenbewerteter Graph mit Bewertung B und Eckenmenge E . Ferner sei h eine Funktion von E nach \mathbb{R} mit der Eigenschaft, daß $h(x) + B[x, y] - h(y) \geq 0$ für alle Kanten (x, y) von G ist. Für jede Kante (x, y) von G setze $B'[x, y] = h(x) + B[x, y] - h(y)$. Genau dann hat ein Weg W minimale Länge bezüglich B , wenn er minimale Länge bezüglich B' hat.

Beweis. Es sei W irgendein Weg von G . W besuche die Ecken e_1, e_2, \dots, e_s in dieser

Reihenfolge. Dann gilt:

$$\begin{aligned}
 L_{B'}(W) &= \sum_{i=1}^{s-1} B'[e_i, e_{i+1}] = \sum_{i=1}^{s-1} (h(e_i) + B[e_i, e_{i+1}] - h(e_{i+1})) \\
 &= \left(\sum_{i=1}^{s-1} B[e_i, e_{i+1}] \right) + h(e_1) - h(e_2) + h(e_2) - h(e_3) + \dots \\
 &\quad \dots + h(e_{s-1}) - h(e_s) \\
 &= L_B(W) + h(e_1) - h(e_s)
 \end{aligned}$$

Es sei nun W ein Weg von e_1 nach e_s , welcher bezüglich der Bewertung B minimale Länge hat. Angenommen, W ist bezüglich B' nicht minimal. Dann gibt es einen Weg \overline{W} von e_1 nach e_s mit $L_{B'}(\overline{W}) < L_{B'}(W)$. Nach der Vorbetrachtung gilt nun

$$L_B(\overline{W}) + h(e_1) - h(e_s) < L_B(W) + h(e_1) - h(e_s).$$

Hieraus folgt $L_B(\overline{W}) < L_B(W)$. Dies ist ein Widerspruch, da W bezüglich B minimal ist. Somit ist auch W bezüglich B' minimal. Analog zeigt man, daß auch jeder bezüglich B' minimale Weg bezüglich B minimal ist. ■

Somit genügt es, die kürzesten Wege bezüglich B' zu bestimmen. Diese sind dann auch bezüglich B kürzeste Wege. Die Längen lassen sich dann wie im obigen Beweis bestimmen. Es bleibt noch zu zeigen, wie eine Funktion h mit der angegebenen Eigenschaft gefunden wird. Dazu wird der Graph G um eine Ecke s erweitert. Von dieser neuen Ecke s führt eine Kante zu jeder Ecke von G , und diese neuen Kanten haben alle die Bewertung 0. Der neue Graph wird mit G' bezeichnet. Man beachte, daß diese Konstruktion nur für gerichtete Graphen durchgeführt wird, denn ungerichtete Graphen mit negativen Kantenbewertungen haben nicht die Eigenschaft (*).

Ist Z ein geschlossener Kantenzug von G' , so kann s nicht auf Z liegen, da $g^-(s) = 0$. Somit ist Z ein geschlossener Kantenzug in G ; d.h. G' hat die Eigenschaft (*), falls G diese besitzt. Für jede Ecke e von G definiert man nun

$$h(e) = d(s, e),$$

wobei $d(s, e)$ die Länge des kürzesten Weges von s nach e in G' ist. Sei nun (x, y) eine Kante von G . Dann ist auch (x, y) eine Kante von G' und nach dem Optimalitätsprinzip gilt:

$$h(y) \leq h(x) + B[x, y]$$

Somit hat man eine Funktion h mit der gewünschten Eigenschaft gefunden. Man beachte, daß mit Hilfe des Algorithmus von Moore und Ford die Funktion h mit der Komplexität $O(nm)$ bestimmt werden kann. Da die n -fache Anwendung des Algorithmus von Dijkstra einen höheren Aufwand hat, gilt somit:

Satz. Es sei G ein kantenbewerteter Graph mit der Eigenschaft (*). Die Bestimmung der kürzesten Wege und deren Längen zwischen allen Paaren von Ecken von G kann in

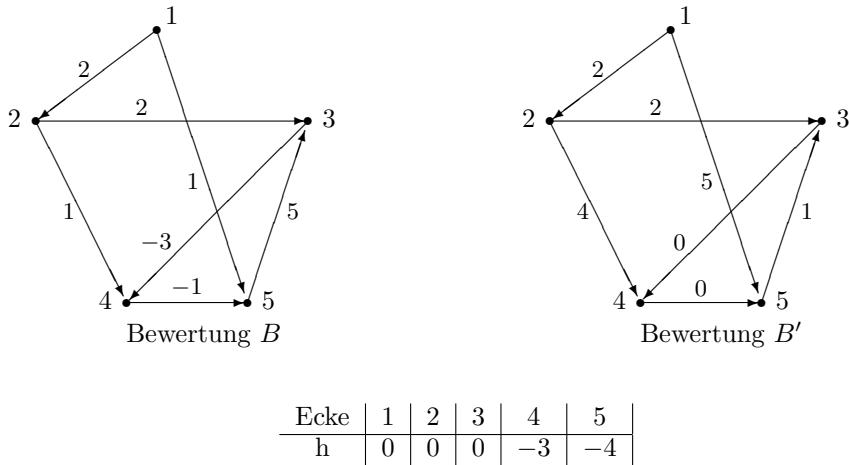


Abbildung 8.22: Die Transformation der Bewertung eines Graphen

der Zeit $O(nm \log n)$ (bzw. $O(nm + n^2 \log n)$) unter Verwendung von Fibonacci-Heaps) erfolgen.

Abbildung 8.22 zeigt einen gerichteten Graphen mit Bewertung B , die Werte der Funktion h und den Graphen mit der neuen Bewertung B' .

Für Graphen mit vielen Kanten (d.h. $O(m) = n^2$) gibt es ein effizientes Verfahren, welches den zusätzlichen Vorteil hat, daß es sehr einfach zu implementieren ist. Dieses Verfahren stammt von R. Floyd und wird im nächsten Abschnitt besprochen.

8.8 Der Algorithmus von Floyd

In Abschnitt 2.6 wurde ein Algorithmus zur Bestimmung des transitiven Abschlusses eines Graphen diskutiert. Warshalls Algorithmus läßt sich leicht erweitern, so daß Problem (3) mit Aufwand $O(n^3)$ gelöst werden kann. Der Algorithmus arbeitet auf der Adjazenzmatrix A und wandelt diese in n Schritten in die Erreichbarkeitsmatrix um. Die dabei entstehenden Matrizen A_1, A_2, \dots, A_n haben eine wichtige Eigenschaft: Der Eintrag (i, k) der Matrix A_j zeigt an, ob es einen Weg von der Ecke i zu der Ecke k gibt, der nur Ecken aus der Menge $\{1, \dots, j\}$ verwendet (außer Anfangs- und Endecke). Durch eine kleine Änderung kann man erreichen, daß der Eintrag (i, k) der Matrix A_j die Länge des kürzesten Weges von i nach k enthält, der nur Ecken aus der Menge $\{1, \dots, j\}$ verwendet.

Die Änderung des Algorithmus von Warshall stammt von R. Floyd. Das Ziel ist die Bestimmung der Distanzmatrix D . Diese entsteht in n Schritten; die dabei auftretenden Matrizen werden mit D_0, D_1, \dots, D_n bezeichnet und haben die oben angegebene Eigenschaft. Die Matrix D_0 enthält die Längen der Kanten, d.h. $D_0 = B$, wobei B die

bewertete Adjazenzmatrix ist. Wie erfolgt nun der Übergang von D_{j-1} nach D_j ? Der Eintrag (i, k) von D_j soll die Länge des kürzesten Weges von i nach k enthalten, welcher nur Ecken aus $\{1, \dots, j\}$ verwendet. Wie sieht ein solcher Weg W aus? Dazu betrachten wir zwei Fälle:

Fall I: W verwendet nicht die Ecke j . Dann verwendet W nur Ecken aus $\{1, \dots, j-1\}$, und somit enthält der Eintrag (i, k) von D_{j-1} die Länge von W .

Fall II: W verwendet die Ecke j . Da der Graph die Eigenschaft (*) hat, kommt j nur einmal auf W vor. Der Weg W wird nun in die beiden Teile W_1 und W_2 aufgeteilt, so daß j die Endecke von W_1 und die Anfangsseite von W_2 ist. Dann gilt

$$L(W) = L(W_1) + L(W_2).$$

Nach dem Optimalitätsprinzip sind W_1 und W_2 kürzeste Wege. Beide Wege verwenden nur Ecken aus $\{1, \dots, j-1\}$. Ihre Längen kann man somit aus der Matrix D_{j-1} entnehmen.

Der Übergang von D_{j-1} nach D_j ist somit sehr einfach: Für alle $i, k \in \{1, \dots, n\}$ setzt man

$$(D_j)_{ik} = \min \left\{ (D_{j-1})_{ik}, (D_{j-1})_{ij} + (D_{j-1})_{jk} \right\}.$$

Dies läßt sich durch folgendes Programmstück realisieren:

```

D := B;
for j := 1 to n do
    for i := 1 to n do
        for k := 1 to n do
            D[i,k] := min(D[i,k], D[i,j]+D[j,k])

```

Es ist ein großer Vorteil, daß die Änderungen der Matrix an Ort und Stelle durchgeführt werden können. Der Grund hierfür ist einfach: Beim Übergang von D_{j-1} nach D_j entscheiden ausschließlich die j -te Zeile und die j -te Spalte, ob ein Eintrag geändert wird. Diese bleiben jedoch selbst unverändert. Der Speicheraufwand ist somit $O(n^2)$.

Durch eine kleine Erweiterung ist es auch möglich, die Vorgängermatrix parallel zur Distanzmatrix zu konstruieren. Dazu betrachten wir noch einmal die beiden Fälle. In Fall I war die Länge des kürzesten Weges schon bekannt, da die Ecke j nicht verwendet wurde; d.h. in diesem Fall bleibt der Vorgänger unverändert. In Fall II setzt sich der kürzeste Weg W aus den beiden Teilwegen W_1 und W_2 zusammen, deren Vorgänger schon bekannt sind. Somit ist der neue Vorgänger der Ecke k gerade der Vorgänger von k auf dem Weg W_2 .

Bleibt noch zu klären, wie die Initialisierung der Vorgängermatrix V erfolgt. Für jede Kante (i, j) von G wird $V[i, j] = i$ und alle anderen Einträge auf 0 gesetzt. Die Prozedur `floyd` in Abbildung 8.23 ist eine Realisierung des Algorithmus von Floyd.

```

var D : array[1..n,1..n] of Real;
V : array[1..n,1..n] of Integer;
procedure floyd (G : B-Graph);
var
  i,j,k : Integer;
begin
  D := B;
  for i := 1 to n do
    for j := 1 to n do
      if Kante(i,j) existiert then
        V[i,j] := i
      else
        V[i,j] := 0;
  for j := 1 to n do
    for i := 1 to n do
      for k := 1 to n do
        if D[i,k] > D[i,j] + D[j,k] then begin
          D[i,k] := D[i,j] + D[j,k];
          V[i,k] := V[j,k];
        end
  end
end

```

Abbildung 8.23: Die Prozedur `floyd`

Die Korrektheit der Prozedur `floyd` wurde schon oben gezeigt. Die Komplexitätsanalyse ist sehr einfach: Die drei geschachtelten `for`-Schleifen ergeben einen Aufwand von $O(n^3)$. Die Prozedur `floyd` kann leicht dahingehend erweitert werden, daß die Bedingung (*) überprüft wird. Wird nämlich ein Diagonaleintrag von D negativ, so bedeutet dies, daß ein geschlossener Kantenzug mit negativer Länge vorliegt. Abbildung 8.24 zeigt eine Anwendung der Prozedur `floyd` auf den Graphen aus Abbildung 8.22. Die Stellen, an denen Änderungen auftreten, sind eingerahmmt.

8.9 Steiner Bäume

Das in Kapitel 3 eingeführte Konzept der minimal aufspannenden Bäume läßt sich noch verallgemeinern. Es sei G ein kantenbewerteter, zusammenhängender, ungerichteter Graph und S eine Teilmenge der Ecken von G . Einen Untergraph von G , dessen Eckenmenge S enthält und ein Baum ist, nennt man *Steiner-Baum* für S . Einen kostenminimalen Steiner Baum für S nennt man *minimalen Steiner Baum*. In der Regel wird ein minimaler Steiner Baum neben den Ecken aus S auch weitere Ecken verwenden, diese nennt man *Steiner Ecken* oder auch *Steiner Punkte*. Abbildung 8.25 zeigt links einen kantenbewerteten, ungerichteten Graph G , in der Mitte einen minimal aufspannenden Baum für G und rechts einen minimalen Steiner Baum für die drei fett dargestellten Ecken von G . Der minimale Steiner Baum enthält eine Steiner Ecke. Das Problem der

$$\begin{array}{rcl}
 D_0 & = & \begin{pmatrix} 0 & 3 & \infty & 4 & \infty \\ \infty & 0 & -1 & \infty & -2 \\ \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & -2 & 0 & -1 \\ 1 & \infty & \infty & \infty & 0 \end{pmatrix} \\
 & & V_0 = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 4 \\ 5 & 0 & 0 & 0 & 0 \end{pmatrix} \\
 \\
 D_1 & = & \begin{pmatrix} 0 & 3 & \infty & 4 & \infty \\ \infty & 0 & -1 & \infty & -2 \\ \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & -2 & 0 & -1 \\ 1 & \boxed{4} & \infty & \boxed{5} & 0 \end{pmatrix} \\
 & & V_1 = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 4 \\ 5 & \boxed{1} & 0 & \boxed{1} & 0 \end{pmatrix} \\
 \\
 D_2 & = & \begin{pmatrix} 0 & 3 & \boxed{2} & 4 & \boxed{1} \\ \infty & 0 & -1 & \infty & -2 \\ \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & -2 & 0 & -1 \\ 1 & 4 & \boxed{3} & 5 & 0 \end{pmatrix} \\
 & & V_2 = \begin{pmatrix} 0 & 1 & \boxed{2} & 1 & \boxed{2} \\ 0 & 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 4 \\ 5 & 1 & \boxed{2} & 1 & 0 \end{pmatrix} \\
 \\
 D_3 & = & \begin{pmatrix} 0 & 3 & 2 & 4 & 1 \\ \infty & 0 & -1 & \infty & -2 \\ \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & -2 & 0 & -1 \\ 1 & 4 & 3 & 5 & 0 \end{pmatrix} \\
 & & V_3 = \begin{pmatrix} 0 & 1 & 2 & 1 & 2 \\ 0 & 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 4 \\ 5 & 1 & 2 & 1 & 0 \end{pmatrix} \\
 \\
 D_4 & = & \begin{pmatrix} 0 & 3 & 2 & 4 & 1 \\ \infty & 0 & -1 & \infty & -2 \\ \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & -2 & 0 & -1 \\ 1 & 4 & 3 & 5 & 0 \end{pmatrix} \\
 & & V_4 = \begin{pmatrix} 0 & 1 & 2 & 1 & 2 \\ 0 & 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 4 \\ 5 & 1 & 2 & 1 & 0 \end{pmatrix} \\
 \\
 D_5 & = & \begin{pmatrix} 0 & 3 & 2 & 4 & 1 \\ \boxed{-1} & 0 & -1 & \boxed{3} & -2 \\ \infty & \infty & 0 & \infty & \infty \\ \boxed{0} & \boxed{3} & -2 & 0 & -1 \\ 1 & 4 & 3 & 5 & 0 \end{pmatrix} \\
 & & V_5 = \begin{pmatrix} 0 & 1 & 2 & 1 & 2 \\ \boxed{5} & 0 & 2 & \boxed{1} & 2 \\ 0 & 0 & 0 & 0 & 0 \\ \boxed{5} & \boxed{1} & 4 & 0 & 4 \\ 5 & 1 & 2 & 1 & 0 \end{pmatrix}
 \end{array}$$

Abbildung 8.24: Eine Anwendung des Algorithmus von Floyd auf den Graphen aus Abbildung 8.21

Bestimmung von minimalen Steiner Bäumen tritt in vielen praktischen Anwendungen auf, beispielsweise im VLSI-Chip-Design.

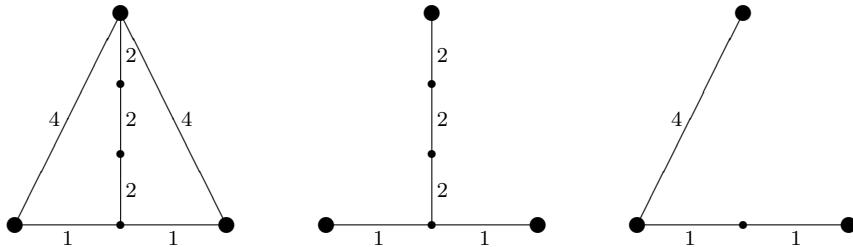


Abbildung 8.25: Ein minimal aufspannender Baum und ein minimaler Steiner Baum

Zwei Spezialfälle von Steiner Bäumen wurden bereits betrachtet. Ist $|S| = n$, d.h., S enthält alle Ecken von G , so ist jeder minimal aufspannende Baum ein minimaler Steiner Baum. Ist $|S| = 2$, so ist jeder kürzeste Weg zwischen den beiden Ecken von S ein minimaler Steiner Baum. Für diese Spezialfälle wurden effiziente Algorithmen vorgestellt. Leider ist für das allgemeine Problem bis heute kein polynomiauer Algorithmus bekannt (vergleichen Sie hierzu Aufgabe 36 in Kapitel 9). Das folgende Lemma zeigt, daß man sich bei der Bestimmung minimaler Steiner Bäume auf vollständige Graphen zurückziehen kann.

Lemma. Es sei G ein ungerichteter, zusammenhängender Graph, dessen Kanten eine nichtnegative Bewertung tragen und S eine Teilmenge der Ecken von G . Ferner sei G' der vollständige Graph mit der gleichen Eckenmenge wie G . Eine Kante (e, f) von G' trägt als Bewertung die Länge des kürzesten Weges von e nach f in G . Es sei T' ein minimaler Steiner Baum von G' für S . Ersetzt man in T' jede Kante (e, f) durch den kürzesten Weg von e nach f in G , so erhält man einen minimalen Steiner Baum T von G für S . Insbesondere sind die Kosten eines minimalen Steiner Baumes von G für S gleich den Kosten eines minimalen Steiner Baumes von G' für S .

Beweis. Die Bewertung einer Kante (e, f) eines beliebigen Steiner Baumes von G für S ist mindestens so hoch wie die Bewertung von (e, f) in G' . Somit sind die Kosten eines minimalen Steiner Baumes von G' für S höchstens gleich den minimalen Kosten eines Steiner Baumes von G für S . Der konstruierte Graph T ist ein zusammenhängender Graph, welcher alle Ecken aus S enthält, er hat die gleichen Kosten wie T' . T kann keine doppelten Kanten oder geschlossenen Wege enthalten, sonst gäbe es einen Steiner Baum von G für S , dessen Kosten kleiner als die Kosten von T' wären. Somit ist T ein minimaler Steiner Baum von G für S . ■

Lemma. Es sei G ein vollständiger Graph, dessen Kanten eine nichtnegative Bewertung tragen und S eine Teilmenge der Ecken.

- Erfüllen die Bewertungen der Kanten die Dreiecksungleichung, dann gibt es einen minimalen Steiner Baum von G für S mit maximal $|S| - 2$ Steiner Ecken.

- b) Es sei T ein minimaler Steiner Baum von G für S und S' die Menge der Steiner Ecken von T . Dann ist jeder minimal aufspannende Baum des von $S \cup S'$ induzierten Untergraphen U von G ebenfalls ein minimaler Steiner Baum von G für S .

Beweis. a) Der Eckengrad einer Steiner Ecke in einem minimalen Steiner Baum ist mindestens gleich 2. Gibt es eine Steiner Ecke e mit Eckengrad 2, so konstruiere man einen neuen Steiner Baum: Die Ecke e und die zu ihr inzidenten Kanten werden entfernt, die die beiden Nachbarn von e verbindende Kante wird neu eingefügt. Da die Dreiecksungleichung gilt, steigen dadurch nicht die Kosten des Baumes. Somit gibt es einen minimalen Steiner Baum T , in dem jede Steiner Ecke mindestens den Eckengrad 3 hat. Es sei a die Anzahl der Steiner Ecken in T . Dann gilt $2(|S| + a - 1) \geq 3a + |S|$ bzw. $|S| - 2 \geq a$.

b) Ein minimal aufspannender Baum von U ist ein Steiner Baum und T ist ein aufspannender Baum von U . Somit sind die Kosten eines minimal aufspannenden Baumes von U gleich den Kosten von T . Hieraus folgt die Behauptung. ■

Nach diesen Vorbereitungen kann ein Algorithmus zur Bestimmung eines minimalen Steiner Baumes angegeben werden. Es sei G ein ungerichteter, zusammenhängender Graph, dessen Kanten eine nichtnegative Bewertung tragen und S eine Teilmenge der Ecken von G . Mit Hilfe des Algorithmus von Floyd wird die Vorgänger- und die Distanzmatrix von G bestimmt. Damit kann der oben beschriebene vollständige Graph G' konstruiert werden. Man beachte, daß G' die Dreiecksungleichung erfüllt. Nach den letzten beiden Lemmas genügt es, die *richtige* Menge S' von Steiner Ecken zu finden, dann ist ein minimal aufspannender Baum von $S \cup S'$ ein minimaler Steiner Baum. Man beachte, daß es maximal $|S| - 2$ Steiner Ecken geben kann. Ist E die Menge der Ecken von G , so werden nun alle Teilmengen von $E \setminus S$ mit höchstens $|S| - 2$ Elementen generiert, wegen

$$\sum_{i=0}^{|S|-2} \binom{n-|S|}{i} \leq \sum_{i=0}^{n-|S|} \binom{n-|S|}{i} = 2^{n-|S|}$$

gibt es höchstens $2^{n-|S|}$ solcher Mengen. Für jede solche Teilmenge M wird mit Hilfe des Algorithmus von Prim ein minimal aufspannender Baum T des von $S \cup M$ induzierten Untergraphen von G' bestimmt. Unter all diesen Bäumen wird der mit den minimalen Kosten ausgewählt und mit Hilfe der Vorgängermatrix zurück nach G transformiert. Dies ist dann ein minimaler Steiner Baum von G für S . Der Algorithmus von Prim wird für Graphen mit $O(|S|)$ Ecken aufgerufen. Hieraus ergibt sich ein Gesamtaufwand von $O(n^3 + 2^{n-|S|}|S|^2)$. Die Laufzeit wächst exponentiell mit $n - |S|$. Verwendet man folgende Abschätzung für die Anzahl der zu erzeugenden Teilmengen

$$\sum_{i=0}^{|S|-2} \binom{n-|S|}{i} \leq (n-|S|)^{|S|-2},$$

so sieht man, daß der Algorithmus für konstantes $|S|$ polynomial in n ist. In Aufgabe 37 auf Seite 335 wird ein approximativer Algorithmus mit Laufzeit $O(|S|n^2)$ angegeben,

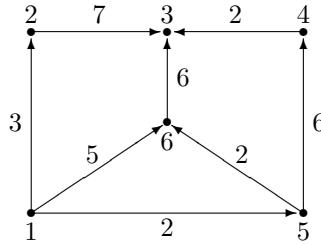
die Kosten des berechneten Steiner Baumes sind maximal doppelt so hoch wie die eines minimalen Steiner Baumes.

8.10 Literatur

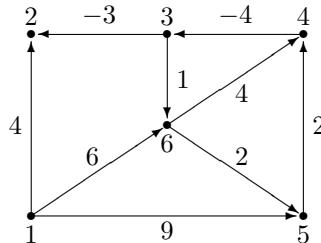
Die Bellmanschen Gleichungen sind in [12] beschrieben. Der allgemeine Algorithmus zur Bestimmung kürzester Wege in Graphen mit Eigenschaft (*) wurde unabhängig von L.R. Ford [43] und E.F. Moore [98] beschrieben; die angegebene Darstellung folgt [15]. Algorithmen zur Überprüfung der Eigenschaft (*) sind in [34] und [115] beschrieben. Dijkstras Algorithmus [29] erschien 1959, enthielt aber noch keinen Hinweis auf die Verwendung von Heaps. Die Implementierung mittels Fibonacci-Heaps ist in [44] beschrieben. Unter der Voraussetzung, daß die Kantenbewertungen positive ganze Zahlen aus der Menge $\{0, \dots, C\}$ mit $C \geq 2$ sind, haben R.K. Ahuja et al. einen Algorithmus mit Laufzeit $O(m + n\sqrt{\log C})$ angegebenen [3]. Für planare Graphen mit positiven Kantenbewertungen kann Problem (2) mit Aufwand $O(n\sqrt{\log n})$ gelöst werden [45]. Goldberg und Radzik haben einen Algorithmus für Graphen mit der Eigenschaft (*) entwickelt, der die gleiche worst case Laufzeit wie der Algorithmus von Moore und Ford hat, aber für kreisfreie Graphen in linearer Zeit $O(n + m)$ läuft [51]. Eine Diskussion des A^* -Algorithmus ist in [60] enthalten. Eine genau Untersuchung des iterativen A^* -Algorithmus findet man in [82]. Die Beschreibung der Umkreissuche basiert auf einer Arbeit von G. Manzini [93]. Die Transformation der Bewertung zur Lösung von Problem (3) stammt von J. Edmonds und R.M. Karp [35]. Floyds Algorithmus zur Lösung von Problem (3) findet man in [41]. Die Ähnlichkeit des Floyd Algorithmus mit dem Algorithmus von Warshall zur Bestimmung des transitiven Abschlusses führte zur Untersuchung von Graphen, deren Bewertungen zu einem *Semiring* gehören. Dadurch wurden auf elegante Weise verschiedene Verfahren zusammengefaßt [94]. Aufgabe 17 findet man in [33], Aufgabe 28 in [39] und Aufgabe 18 in [28]. Cherkassky et al. haben eine aktuelle Übersicht und einen experimentellen Vergleich von Algorithmen zur Lösung von Problem (1) erstellt [20]. Einen guten Überblick über Anwendungen von Steiner Bäumen in Netzwerken gibt Winter in [125].

8.11 Aufgaben

1. Der Algorithmus von Moore und Ford bestimmt für kantenbewertete Graphen mit der Eigenschaft (*) die kürzesten Wege und deren Längen von einer Startecke zu allen erreichbaren Ecken. Zeigen Sie, daß der Algorithmus auch dann korrekt arbeitet, wenn die Graphen nicht die Eigenschaft (*), sondern folgende Eigenschaft haben:
Alle von der Startecke aus erreichbaren geschlossenen Kantenzüge haben eine nichtnegative Länge.
2. Geben Sie für den folgenden gerichteten Graphen zwei verschiedene kW-Bäume mit Startecke 1 an!

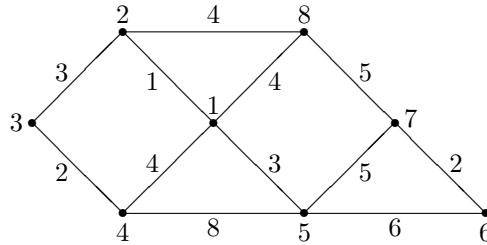


3. Welches Verfahren wendet man am besten für die Bestimmung der kürzesten Wege und deren Längen des Graphen aus Übungsaufgabe 2 an?
4. Wie kann man in einem kantenbewerteten Graphen mit der Eigenschaft (*) folgende Probleme lösen?
 - a) Bestimmung des kürzesten Kantenzuges zwischen zwei Ecken e und f , der durch eine vorgegebene Ecke führt.
 - b) Bestimmung des kürzesten Kantenzuges zwischen zwei Ecken e und f , der durch ℓ vorgegebene Ecken e_1, \dots, e_ℓ geht.
5. Es sei G ein kantenbewerteter, zusammenhängender, ungerichteter Graph. Beweisen Sie: Es gibt mindestens $n - 1$ Paare (e, f) von Ecken, so daß der kürzeste Weg von e nach f aus genau einer Kante besteht.
6. Bestimmen Sie in dem folgenden Graphen die kürzesten Wege und deren Längen von Ecke 1 zu allen anderen Ecken. Verwenden Sie den Algorithmus von Moore und Ford.



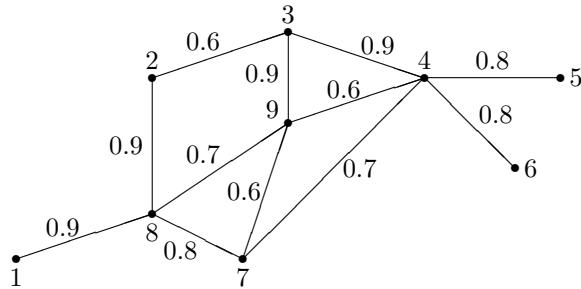
- * 7. Der Algorithmus von Moore und Ford verwaltet die markierten Ecken in einer Warteschlange. Die jeweils erste Ecke i wird entfernt und bearbeitet. Es sei j der momentane Vorgänger von i im kW-Baum. Warum ist es nicht sinnvoll, die Ecke i zu bearbeiten, falls sich j ebenfalls in der Warteschlange befindet? Ändern Sie die Prozedur `kürzesteWege` aus Abbildung 8.5 dahingehend, daß solche Ecken zwar aus der Warteschlange entfernt, aber nicht bearbeitet werden. Beweisen Sie die Korrektheit dieses Verfahrens und bestimmen Sie die Laufzeit.

- * 8. Entwerfen Sie einen Algorithmus mit linearer Laufzeit zur Bestimmung der kürzesten Wege in einem gerichteten kreisfreien Graphen. Verwenden Sie dabei die Prozedur `topsort` aus Kapitel 4.
- * 9. Hat der Algorithmus von Moore und Ford für kreisfreie gerichtete Graphen eine worst case Laufzeit von $O(n + m)$?
- 10. Bestimmen Sie in dem folgenden Graphen die kürzesten Wege und deren Längen von Ecke 3 zu allen anderen Ecken. Verwenden Sie den Algorithmus von Dijkstra.



- 11. Es sei G ein kantenbewerteter Graph mit der Eigenschaft (*). Für eine Ecke e seien die Abstände $d(e, i)$ zu allen von e erreichbaren Ecken i bekannt. Geben Sie einen Algorithmus an, der einen kW-Baum für G mit Wurzel e bestimmt. Der Algorithmus soll eine worst case Komplexität von $O(m)$ haben.
- 12. Es sei G ein kantenbewerteter Graph mit Eigenschaft (*), e und f Ecken von G und W ein kürzester Weg von e nach f . Ferner seien e_1 und f_1 Ecken von G , welche nacheinander auf W liegen. Beweisen Sie: Das Teilstück \bar{W} von W , welches e_1 und f_1 verbindet, ist ebenfalls ein kürzester Weg.
- * 13. Der auf der folgenden Seite abgebildete Graph repräsentiert ein *Datenübertragungsnetzwerk*. Die Ecken sind die Datenempfänger bzw. die Datensender, und die Kanten sind die Übertragungsleitungen. Die Bewertungen der Kanten geben die Wahrscheinlichkeit dafür an, daß eine Nachricht über diese Leitung korrekt übertragen wird. Man interessiert sich nun für Übertragungswege zwischen Ecken mit der größten Wahrscheinlichkeit einer korrekten Übertragung. Die Wahrscheinlichkeit, daß eine Nachricht auf einem Weg korrekt übertragen wird, ergibt sich aus dem Produkt der Kantenwahrscheinlichkeiten. Die in diesem Kapitel diskutierten Algorithmen lassen sich nicht direkt auf dieses Problem anwenden, denn die „Länge“ eines Weges ist hier das Produkt und nicht die Summe der Kantenbewertungen. Dieses Problem kann man umgehen, indem man die Bewertung b_{ij} der Kante von i nach j durch $-\log b_{ij}$ ersetzt. Warum?

Bestimmen Sie den Übertragungsweg von Sender 1 zum Empfänger 5 mit der größten Wahrscheinlichkeit einer korrekten Übertragung.



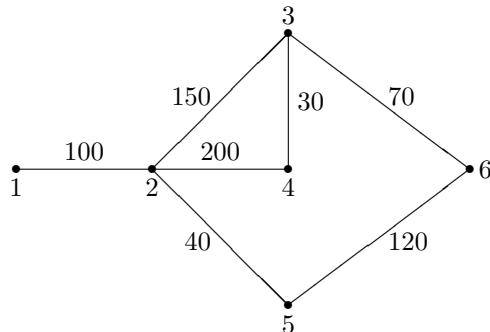
- ** 14. Es sei G ein Graph, dessen Kanten mit Werten aus der Menge $\{1, 2, 3, \dots, C\}$ bewertet sind. Ändern Sie den Algorithmus von Dijkstra derart, daß seine Laufzeit $O((n + m) \log C)$ beträgt. (Hinweis: Zu jedem Zeitpunkt des Algorithmus gilt $W = \{D[i] \mid i \in B\} \subseteq \{s, s+1, s+2, \dots, s+C\}$, wobei $s = \min W$.)
15. Es sei G ein Graph, dessen Kanten alle die gleiche Bewertung tragen. Beweisen Sie, daß der von dem Dijkstra-Algorithmus erzeugte kW-Baum ein Breitensuchbaum ist.
16. Bestimmen Sie die kürzesten Wege von Ecke 1 zu allen erreichbaren Ecken für den Graphen aus Abbildung 8.10.
17. Verändern Sie den Algorithmus von Moore und Ford derart, daß eine Ecke i nur dann am Ende der Warteschlange eingefügt wird, wenn der Wert $D[i]$ zum ersten Mal geändert wurde. Andernfalls wird die Ecke i am Anfang der Warteschlange eingefügt. Vergleichen Sie die beiden Implementierungen anhand konkreter Beispiele.
18. Die Laufzeit der Prozedur `dijkstra` hängt wesentlich davon ab, wie die Speicherung der Menge B und die Suche der Ecke i aus B mit minimalem $D[i]$ gelöst wird. Eine Möglichkeit dazu ist *Bucket-Sort*, d.h. Sortieren mit Fächern. Bei dieser Methode werden die Ecken aus B teilweise sortiert gespeichert. Es sei B_{\max} die größte Kantenbewertung. Beweisen Sie folgende Ungleichung:

$$\max\{D[i] \mid i \in B\} - \min\{D[i] \mid i \in B\} \leq B_{\max}$$

Dies macht man sich zu Nutzen, indem man B in „Fächern“ abspeichert. Diese Fächer bilden eine Aufteilung des Bereichs $[0, B_{\max}]$. Eine Ecke i wird in dem Fach mit der Nummer $D[i] \bmod (B_{\max} + 1)$ abgelegt. Realisieren Sie diese Version des Algorithmus von Dijkstra und bestimmen Sie die Laufzeit.

19. Gegeben sei ein Graph, in dem sowohl die Kanten als auch die Ecken Bewertungen tragen. Die Länge eines Kantenzuges ist gleich der Summe der Bewertungen der verwendeten Ecken und Kanten. Wie kann die Bestimmung kürzester Wege in solchen Graphen mit einem der in diesem Kapitel beschriebenen Algorithmen gelöst werden?

20. Geben Sie ein Beispiel für einen ungerichteten zusammenhängenden Graphen mit nichtnegativen Kantenbewertungen an, der einen kW-Baum besitzt, welcher kein minimal aufspannender Baum ist.
21. Bestimmen Sie für den gerichteten Graphen aus Übungsaufgabe 6 die kürzesten Wege und deren Längen für alle Paare von Ecken. Verwenden Sie dabei zwei verschiedene Algorithmen:
- den Algorithmus von Floyd,
 - den Algorithmus von Dijkstra mit einer entsprechenden Transformation.
22. Der unten stehende Graph repräsentiert das Netz von Filialen einer Kaufhauskette. Die Ecken sind die Filialen und die Kanten die Straßen. Die Bewertung einer Kante gibt die Länge der Straße an. In welcher Filiale soll die Kaufhauskette ihr Zentrallager anlegen? Der Ort soll so gewählt werden, daß die weiteste Wegstrecke möglichst kurz ist.

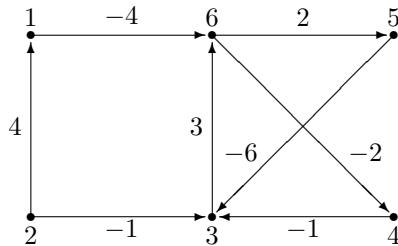


23. In einem kantenbewerteten Graphen werden die Bewertungen aller Kanten
- um einen konstanten positiven Betrag erhöht,
 - mit einer positiven Konstanten multipliziert.
- In welchem der beiden Fälle bleiben die kürzesten Wege die gleichen? Wie verhalten sich die Längen der kürzesten Wege?
24. Ein gerichteter kantenbewerteter Graph ist durch seine bewertete Adjazenzmatrix gegeben:

$$\begin{pmatrix} 0 & 64 & 64 & 64 & 64 & 64 \\ -2 & 0 & 64 & 64 & 64 & 64 \\ -4 & -3 & 0 & 64 & 64 & 64 \\ -7 & -6 & -5 & 0 & 64 & 64 \\ -12 & -11 & -10 & -9 & 0 & 64 \\ -21 & -20 & -19 & -18 & -17 & 0 \end{pmatrix}$$

Zeigen Sie, daß der Graph die Eigenschaft (*) hat. Bestimmen Sie die kürzesten Wege und deren Längen für die Startecke 1 mit Hilfe des Algorithmus von Moore und Ford. Wie sieht der kW-Baum aus und wie oft wird die **while**-Schleife ausgeführt?

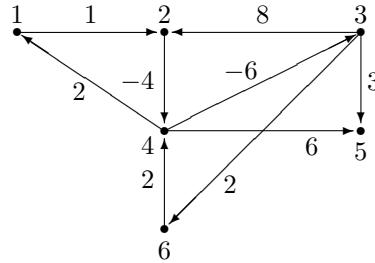
25. Bestimmen Sie für den Graphen aus Abbildung 8.10 die positive Kantenbewertung B' gemäß Abschnitt 8.7.
26. Wenden Sie den Algorithmus von Floyd auf den folgenden kantenbewerteten Graphen an! Was stellen Sie fest?



27. Es sei G ein kantenbewerteter, zusammenhängender, ungerichteter Graph und B ein minimal aufspannender Baum von G . Die Bewertungen der Kanten sind nicht-negativ. Geben Sie einen Algorithmus an, welcher in linearer Zeit $O(m)$ feststellt, ob B für eine gegebene Ecke e ein kW-Baum ist oder nicht.
- ** 28. Entwerfen Sie einen auf der Tiefensuche basierenden Algorithmus, welcher feststellt, ob ein gerichteter kantenbewerteter Graph die Eigenschaft (*) hat. Die Tiefensuche wird dabei auf jede Ecke angewendet, und es werden nur solche Kanten verfolgt, für die die Gesamtlänge von der Startecke aus betrachtet negativ ist. Trifft man dabei wieder auf die Startecke, so liegt ein geschlossener Weg mit negativer Länge vor. Ist diese Suche für alle Ecken erfolglos, so hat der Graph die Eigenschaft (*).

Die Korrektheit des Algorithmus stützt sich auf folgenden Satz: Es sei Z ein geschlossener Weg mit negativer Länge in einem gerichteten kantenbewerteten Graphen. Dann gibt es eine Ecke e auf Z mit der Eigenschaft, daß alle bei e startenden Teilwege von Z eine negative Länge haben. (Hinweis zum Beweis: Führen Sie einen indirekten Beweis. Von einer beliebigen Ecke von Z aus startend bilde man fortlaufend Teilwege Z_1, Z_2, \dots von Z , indem immer dann ein neuer Weg begonnen wird, sobald der vorhergehende eine positive Länge hat. Irgendwann muß ein Weg Z_i zum zweiten Mal auftreten. Die dazwischenliegenden Wege haben alle positive Länge. Dies steht im Widerspruch zur negativen Gesamtlänge von Z .)

29. Bestimmen Sie für folgenden gerichteten Graphen die längsten Wege von Ecke 1 zu allen anderen Ecken.

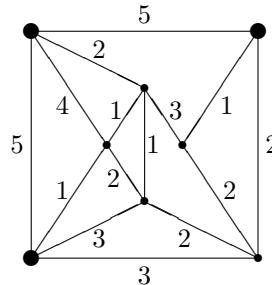


30. Die Länge eines Kantenzuges $Z = k_1, k_2, \dots, k_s$ in einem kantenbewerteten Graphen kann auch abweichend von Abschnitt 8.1 folgendermaßen definiert werden:

$$L'(W) = \min\{L(k_i) \mid i = 1, \dots, s\}$$

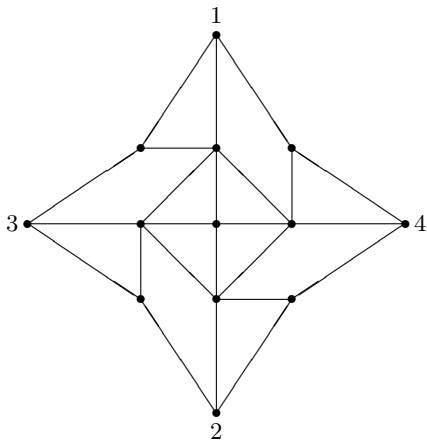
Das heißt, die Länge L' eines Weges W ist gleich der Länge der kürzesten Kante von W . Ändern Sie den Algorithmus von Floyd derart, daß die so definierten kürzesten Wege zwischen allen Paaren von Ecken bestimmt werden.

31. Es seien a, b positive reelle Zahlen und f_1, f_2 konsistente Schätzfunktionen für einen Graphen. Beweisen Sie, daß auch $(af_1 + bf_2)/(a+b)$ eine konsistente Schätzfunktion ist.
- ** 32. Betrachten Sie noch einmal das 8-Puzzle aus Abschnitt 4.11 des Kapitels 4 auf Seite 121. Geben Sie zwei verschiedene zulässige Schätzfunktionen an, und wenden Sie den A^* -Algorithmus auf die angegebene Stellung an. Sind die Schätzfunktionen konsistent?
- * 33. Betrachten Sie noch einmal das in Abschnitt 6.6 beschriebene Problem der kostenminimalen Flüsse. Geben Sie unter der Voraussetzung, daß die Kapazitäten ganze Zahlen sind, einen Algorithmus zur Bestimmung eines maximalen kostenminimalen Flusses an. Bestimmen Sie die Laufzeit des Algorithmus.
- * 34. Ändern Sie den Algorithmus von Dijkstra derart ab, daß er unter allen kürzesten Wegen zwischen zwei Ecken denjenigen findet, welcher aus den wenigsten Kanten besteht.
35. Bestimmen Sie für die drei fett dargestellten Ecken des folgenden Graphen einen minimalen Steiner Baum.



Kapitel 9

Approximative Algorithmen



In diesem Kapitel werden Probleme behandelt, für die es höchstwahrscheinlich nur Algorithmen mit exponentieller Laufzeit gibt. Zunächst wird eine relativ informelle Einführung in die Komplexitätsklassen \mathcal{P} , \mathcal{NP} und \mathcal{NPC} gegeben. Danach werden approximative Algorithmen eingeführt, und es werden Maßzahlen definiert, mit denen die Qualität dieser Algorithmen charakterisiert werden kann. Diese Maßzahlen sind zum einen wichtig, um verschiedene approximative Algorithmen zu vergleichen und zum anderen, um eine Abschätzung der Abweichung der erzeugten Lösung von der optimalen Lösung zu haben. Unter der Voraussetzung $\mathcal{P} \neq \mathcal{NP}$ wird gezeigt werden, daß die meisten \mathcal{NP} -vollständigen Probleme keine approximativen Algorithmen mit beschränktem absoluten Fehler besitzen. Ferner wird gezeigt, daß sich die Probleme aus \mathcal{NPC} bezüglich der Approximierbarkeit mit beschränktem relativen Fehler sehr unterschiedlich verhalten. Am Ende dieses Kapitels werden approximative Algorithmen für das Färbungsproblem und das Traveling-Salesman Problem vorgestellt. Weiterhin werden Abschätzungen für den Wirkungsgrad dieser Algorithmen untersucht.

9.1 Die Komplexitätsklassen \mathcal{P} , \mathcal{NP} und \mathcal{NPC}

Dieser Abschnitt dient der Motivation der in diesem Kapitel behandelten Approximationsalgorithmen. Die Ausführungen sind relativ informell, und neue Konzepte werden nur dann näher erläutert, wenn sie für das Verständnis des weiteren Stoffes notwendig sind. Eine streng formale Darstellung würde den Rahmen dieses Kapitels sprengen. Eine ausgezeichnete Darstellung dieses Themas findet man in dem Buch von A.R. Garey und D.S. Johnson [46].

In Kapitel 2 wurde der Begriff des effizienten Algorithmus eingeführt: Ein Algorithmus heißt effizient, wenn er polynomialem Aufwand $O(p(n))$ für ein Polynom $p(n)$ hat. Hierbei ist n ein Maß für die Länge der Eingabe. Algorithmen, deren Laufzeit nicht polynomial ist, nennt man *exponentiell*. Der Unterschied zwischen diesen beiden Gruppen von Algorithmen wurde bereits in Kapitel 2 diskutiert und ist aus Abbildung 2.19 ersichtlich. Exponentielle Algorithmen sind im allgemeinen nur auf Eingaben „kleiner“ Länge anwendbar. In Bezug auf Graphalgorithmen bedeutet dies, daß sie nur für Graphen mit wenigen Ecken und Kanten in überschaubarer Zeit terminieren. Die Einteilung von Algorithmen in diese beiden Gruppen wurde erstmals in den 60er Jahren von A. Cobham und J. Edmonds vorgenommen. Probleme, für die es keine polynomialen Algorithmen gibt, werden auch *intractable* genannt, da sie schwer zu handhaben sind.

Die getroffene Definition von exponentiellen Algorithmen schließt aber nicht aus, daß sie überhaupt nicht zu verwenden sind. Es kann durchaus sein, daß ein Algorithmus mit Aufwand $O(n^3)$ für Werte von n unter 20 schneller arbeitet als ein Algorithmus mit Aufwand $O(n^2)$ (vergleichen Sie hierzu auch Abbildung 2.19). Der Grund liegt darin, daß dies nur worst case Angaben sind. Dies bedeutet deshalb auch nur, daß es mindestens eine Eingabe gibt, für welche soviel Zeit benötigt wird. Ferner wird auch über die Größen der beteiligten Konstanten nichts ausgesagt. Einige exponentielle Algorithmen wie z.B. der *Simplex-Algorithmus* für das Problem der *linearen Programmierung* werden erfolgreich in der Praxis eingesetzt, obwohl es auch polynomiale Algorithmen gibt. Nur für speziell konstruierte Eingaben benötigt der Simplex-Algorithmus wirklich exponentiell viele Schritte. Allerdings ist die überwiegende Mehrheit exponentieller Algorithmen praktisch nicht anwendbar. Auf der anderen Seite ist natürlich auch ein polynomialer Algorithmus mit Laufzeit $O(n^{20})$ und selbst ein linearer Algorithmus mit extrem hohen Konstanten praktisch nicht brauchbar. Die in diesem Buch vorgestellten polynomialen Algorithmen fallen jedoch nicht in diese Kategorie. Die Untersuchung von Problemen, für die keine polynomialen Algorithmen bekannt sind, führte zu interessanten Einblicken in die Komplexitätstheorie von Algorithmen.

Im folgenden wird nochmal das Problem der Bestimmung der chromatischen Zahl eines ungerichteten Graphen aus Kapitel 5 betrachtet. Dies ist ein Beispiel für ein *Optimierungsproblem*: Unter allen Färbungen ist diejenige gesucht, welche die wenigsten Farben verwendet. Demgegenüber stehen *Entscheidungsprobleme*; sie haben eine der beiden Lösungen ja oder nein. Die im folgenden beschriebene Komplexitätstheorie beschäftigt sich ausschließlich mit Entscheidungsproblemen. Dies wird im wesentlichen aus technischen Gründen gemacht und stellt keine große Einschränkung dar. Jedem Optimierungsproblem kann auf einfache Art ein Entscheidungsproblem zugeordnet werden. Ist das Optimierungsproblem ein Minimierungsproblem, so kommt man durch einen zusätzlichen Parameter C zu einem Entscheidungsproblem: Gibt es eine Lösung für das

Optimierungsproblem, deren Wert höchstens C ist? Analog verfährt man mit Maximierungsproblemen. Die Bestimmung der chromatischen Zahl eines ungerichteten Graphen G ist ein Minimierungsproblem. Das zugehörige Entscheidungsproblem lautet: Gibt es eine Färbung von G , welche höchstens C Farben verwendet? Das auf diese Weise beschriebene Entscheidungsproblem ist nicht schwerer zu lösen als das zugehörige Optimierungsproblem: Gibt es einen polynomialen Algorithmus für das Optimierungsproblem, so auch für das Entscheidungsproblem.

Der Grund für die Beschränkung auf Entscheidungsprobleme liegt darin, daß sie mit Hilfe von *formalen Sprachen* beschrieben und die entsprechenden Algorithmen durch *Turing Maschinen* dargestellt werden können. Dadurch erreicht man, daß die in diesem Abschnitt relativ formlos beschriebenen Konzepte auch streng formal erfaßt werden können. Für die Motivation dieses Kapitels ist aber eine informelle Beschreibung ausreichend.

In der Komplexitätstheorie faßt man Probleme mit gleicher Komplexität zu Klassen zusammen. Die Komplexitätsklasse \mathcal{P} umfaßt alle Entscheidungsprobleme, die sich durch polynomiale Algorithmen lösen lassen. In Kapitel 8 wurde gezeigt, daß folgendes Problem in \mathcal{P} liegt:

Kürzester Weg

Es sei G ein Graph mit Kantenbewertungen aus \mathbb{N} , e und f Ecken von G , und C sei eine positive natürliche Zahl.

Entscheidungsproblem: Gibt es in G einen einfachen Weg von e nach f , dessen Länge höchstens C ist?

Der Algorithmus von Dijkstra löst dieses Problem mit Aufwand $O(m \log n)$. Die im folgenden beschriebene Komplexitätsklasse \mathcal{NP} (*non-deterministic polynomial*) umfaßt die Klasse \mathcal{P} . Ein Entscheidungsproblem liegt in \mathcal{NP} , wenn eine positive Lösung in polomialer Zeit überprüft werden kann. Wie man zu einer Lösung kommt, ist dabei nicht von Bedeutung. Das folgende Problem liegt in der Klasse \mathcal{NP} :

Längster Weg

Es sei G ein Graph mit Kantenbewertungen aus \mathbb{N} , e und f Ecken von G , und C sei eine positive natürliche Zahl.

Entscheidungsproblem: Gibt es in G einen einfachen Weg von e nach f , dessen Länge mindestens C ist?

Die Überprüfung einer positiven Lösung W ist in diesem Fall einfach: Es wird überprüft, ob W ein Weg ist, und die Bewertungen der Kanten von W werden addiert und mit C verglichen. Diese Überprüfung kann in linearer Zeit erfolgen, und somit liegt das Problem in \mathcal{NP} . Liegt es eventuell sogar in der Klasse \mathcal{P} ? Multipliziert man die Bewertungen aller Kanten mit -1 und sucht man nun nach einem einfachen Weg, dessen Länge höchstens $-C$ ist, so hat ein so gefundener Weg bezüglich der ursprünglichen Bewertung eine Länge von mindestens C . Leider lassen sich die in Kapitel 8 entwickelten Algorithmen nicht auf das neue Problem anwenden, denn der neue Graph hat nicht die Eigenschaft (*), d.h. es können geschlossene Wege mit negativer Länge auftreten. Dies

ist genau dann der Fall, wenn es in dem ursprünglichen Graphen einen geschlossenen Weg gibt. Bis heute ist kein polynomialer Algorithmus für dieses Problem bekannt.

Man beachte, daß die Definition von \mathcal{NP} keine Bedingung an die Überprüfung einer negativen Lösung stellt. Für das Beispiel der längsten Wege bedeutet eine negative Lösung, daß es keinen einfachen Weg von e nach f gibt, dessen Länge mindestens C ist. Eine Überprüfung erfordert im Prinzip die Betrachtung aller Wege von e nach f . Bisher ist dafür kein polynomialer Algorithmus bekannt. Somit ist nicht bekannt, ob die negative Version des Längsten-Wege-Problems in \mathcal{NP} liegt.

Obwohl die Definition der Klasse \mathcal{NP} auf den ersten Blick nichts über den Aufwand zur Bestimmung von Lösungen aussagt, hat man gezeigt, daß es für jedes Problem aus \mathcal{NP} einen Algorithmus mit Aufwand $O(2^{p(n)})$ gibt, der das Problem löst. Hierbei ist $p(n)$ ein Polynom. Eines der größten ungelösten Probleme der theoretischen Informatik ist die Frage, ob die Klassen \mathcal{P} und \mathcal{NP} übereinstimmen. Ein wichtiges Konzept, das in diesem Zusammenhang entwickelt wurde, ist das der *polynomialen Transformation*. In Kapitel 7 wurden einige Probleme auf die Bestimmung von maximalen Flüssen auf Netzwerken zurückgeführt. Zum Beispiel können maximale Zuordnungen von bipartiten Graphen mittels maximaler Flüsse auf 0-1-Netzwerken bestimmt werden. Der so entstehende Algorithmus besteht aus drei Schritten:

- Transformation des bipartiten Graphen G in ein geeignetes Netzwerk N
- Bestimmung eines maximalen Flusses f auf N
- Bestimmung einer maximalen Zuordnung von G mittels f und N

Der Aufwand des Algorithmus ist gleich der Summe der Aufwände der drei Teilschritte. Ist der Aufwand der Schritte a) und c) zusammen polynomial, so spricht man von einer *polynomialen Transformation*.

Sind P_1 und P_2 Probleme und gibt es eine polynomiale Transformation von P_1 nach P_2 , so schreibt man $P_1 \propto P_2$. Gibt es einen polynomialen Algorithmus für P_2 , so folgt aus $P_1 \propto P_2$, daß es auch einen polynomialen Algorithmus für P_1 gibt.

Die Suche nach einer Antwort auf die Frage, ob $\mathcal{P} = \mathcal{NP}$ ist, führte zur Definition einer Unterklasse von \mathcal{NP} . Die Komplexitätsklasse \mathcal{NPC} umfaßt alle Entscheidungsprobleme P aus \mathcal{NP} , für die gilt: Für alle $P' \in \mathcal{NP}$ gilt $P' \propto P$. Diese Definition besagt, daß ein Problem in \mathcal{NPC} ist, wenn es für jedes Problem aus \mathcal{NP} eine polynomiale Transformation auf dieses Problem gibt. Dies ist eine starke Anforderung, und es ist nicht leicht ersichtlich, daß es überhaupt Probleme gibt, die in \mathcal{NPC} liegen. Probleme aus \mathcal{NPC} werden auch *\mathcal{NP} -vollständig* (*\mathcal{NP} -complete*) genannt.

Im folgenden wird diese Bezeichnung auch für die zugehörigen Optimierungsprobleme verwendet. Man beachte allerdings, daß nicht jedes Entscheidungsproblem aus \mathcal{NPC} auch ein zugehöriges Optimierungsproblem hat (z.B. beim Hamiltonschen Kreis). Aus der Definition von \mathcal{NPC} ergeben sich zwei wichtige Folgerungen:

- Satz.** a) Gibt es ein Problem $P \in \mathcal{NPC}$, welches auch in \mathcal{P} liegt, so ist $\mathcal{NP} = \mathcal{P}$.
b) Ist $P \in \mathcal{NPC}$ und $P' \in \mathcal{NP}$ ein weiteres Problem mit $P \propto P'$, so ist auch $P' \in \mathcal{NPC}$.

Beweis. Zum Beweis von a) sei P' ein beliebiges Problem aus \mathcal{NP} . Dann folgt $P' \propto P$. Da $P \in \mathcal{P}$ ist, gibt es somit einen polynomiauen Algorithmus für P' . Somit ist $P' \in \mathcal{P}$, und es gilt $\mathcal{NP} \subseteq \mathcal{P}$. Zu b) beachte man, daß die Relation \propto transitiv ist. ■

Die Bedeutung der Klasse \mathcal{NPC} liegt in der ersten Folgerung. Gelingt es, für ein einziges Problem $P \in \mathcal{NPC}$ einen polynomiauen Algorithmus anzugeben, so hat man schon $\mathcal{NP} = \mathcal{P}$ bewiesen; d.h. die Klasse \mathcal{NPC} enthält die vermeintlich schwierigsten Probleme aus \mathcal{NP} . Zuvor muß aber erst gezeigt werden, daß es überhaupt ein Entscheidungsproblem gibt, welches in \mathcal{NPC} liegt. Dies gelang S.A. Cook im Jahre 1971. Er zeigte, daß das sogenannte *Satisfiability-Problem SAT* für Boolesche Ausdrücke in \mathcal{NPC} liegt. Dies war der Startpunkt für viele Arbeiten, welche nachwiesen, daß bestimmte Probleme in \mathcal{NPC} liegen. Dabei machte man sich meistens die zweite Folgerung zunutze. Heute sind weit über 1000 Probleme aus den verschiedensten Anwendungsgebieten bekannt, von denen man zeigen kann, daß sie in \mathcal{NPC} liegen. Auch das oben angesprochene Problem der Bestimmung von längsten Wegen liegt in \mathcal{NPC} .

Im folgenden sind fünf weitere Probleme aus dem Gebiet der Graphentheorie aufgelistet, von denen nachgewiesen ist, daß sie in \mathcal{NPC} liegen.

Färbbarkeit von Graphen:

Es sei G ein ungerichteter Graph und C eine positive natürliche Zahl mit $C \leq n$. Entscheidungsproblem: Gibt es eine C -Färbung von G ?

Unabhängige Mengen in Graphen:

Es sei G ein ungerichteter Graph und C eine positive natürliche Zahl mit $C \leq n$. Entscheidungsproblem: Enthält G eine unabhängige Menge mit mindestens C Ecken?

Hamiltonscher Kreis:

Es sei G ein ungerichteter Graph.

Entscheidungsproblem: Gibt es einen einfachen geschlossenen Weg in G , welcher jede Ecke genau einmal verwendet?

Traveling-Salesman:

Es sei G ein vollständiger Graph mit positiven natürlichen Kantenbewertungen, und C sei eine positive natürliche Zahl.

Entscheidungsproblem: Gibt es in G einen Hamiltonschen Kreis, dessen Länge höchstens C ist?

Aufspannender Baum mit begrenzten Eckengraden:

Es sei G ein ungerichteter Graph und C eine positive natürliche Zahl mit $C \leq n$. Entscheidungsproblem: Gibt es einen aufspannenden Baum B für G , so daß der Eckengrad von jeder Ecke in B maximal C ist?

Bis heute ist es nicht gelungen, die Frage zu beantworten, ob $\mathcal{P} = \mathcal{NP}$ ist. Trotz inten-

siver Bemühungen konnte für kein Entscheidungsproblem aus \mathcal{NPC} ein polynomialer Algorithmus angegeben werden. Dies spricht eher dafür, daß $\mathcal{NP} \neq \mathcal{P}$ ist, aber das Problem bleibt weiterhin ungelöst. Da viele Probleme in \mathcal{NPC} von praktischer Bedeutung sind, besteht großes Interesse an alternativen Vorgehensweisen zur Lösung dieser Probleme.

Die wichtigsten bisher betrachteten Algorithmen zur Lösung von Problemen aus \mathcal{NPC} sind mehr oder weniger Variationen des in Kapitel 5 vorgestellten Backtracking-Verfahrens. Im Prinzip werden dabei alle möglichen Lösungen, d.h. der komplette Lösungsbau, durchsucht. Man kann nun versuchen, schon relativ früh zu erkennen, daß ein Teilbaum zu keiner Lösung führen kann. Ein solcher Teilbaum muß dann nicht weiter durchsucht werden. Das „*Branch-and-bound*“ Verfahren ist ein Beispiel für diese Vorgehensweise. Dies ändert zwar nichts daran, daß die Algorithmen exponentiell sind, aber es lassen sich „größere“ Probleme in überschaubarer Zeit lösen.

Die Beschränkung auf Entscheidungsprobleme ist in den meisten Fällen keine wesentliche Einschränkung. Dazu wird beispielhaft das Problem der Färbbarkeit betrachtet (vergleichen Sie auch Aufgabe 35). Im folgenden wird gezeigt, daß es genau dann einen polynomialen Algorithmus für das Entscheidungsproblem gibt, wenn es einen solchen für das Optimierungsproblem gibt. Die eine Richtung der Aussage gilt für alle Optimierungsprobleme, wie oben schon gezeigt wurde.

Es sei nun A ein polynomialer Algorithmus für das Entscheidungsproblem der Färbbarkeit von Graphen. Daraus wird ein polynomialer Algorithmus für das entsprechende Optimierungsproblem konstruiert. Zunächst wird mit Hilfe des Algorithmus A die chromatische Zahl C des Graphen G bestimmt. Dazu sind maximal $\log_2 n$ Aufrufe des Algorithmus A notwendig. Die Vorgehensweise ist dabei die gleiche wie bei der binären Suche. In einem ersten Schritt testet man, ob $C \leq n/2$ oder $C > n/2$ gilt etc. Somit kann C in polynomialer Zeit bestimmt werden.

In einem zweiten Schritt wird nun eine C -Färbung von G bestimmt. Dazu betrachtet man jedes Paar e, f von nicht benachbarten Ecken. Mit Hilfe von A stellt man fest, ob durch Hinzufügen einer Kante von e nach f sich die chromatische Zahl auf $C+1$ erhöht. Ist dies nicht der Fall, so fügt man die Kante in G ein. Man erhält so in maximal n^2 Schritten einen Graphen G' mit folgenden Eigenschaften:

- a) $\chi(G') = C$
- b) Eine minimale Färbung von G' ist auch eine minimale Färbung für G .
- c) In einer minimalen Färbung von G' haben nicht benachbarte Ecken die gleiche Farbe.

Eine minimale Färbung für G' läßt sich nun leicht bestimmen, indem man die letzte Eigenschaft ausnutzt (vergleichen Sie Aufgabe 14). Insgesamt kann also mit polynomialen Aufwand eine minimale Färbung für G gefunden werden.

9.2 Einführung in approximative Algorithmen

Falls $\mathcal{NP} \neq \mathcal{P}$ ist, so bedeutet dies für ein Optimierungsproblem aus \mathcal{NPC} genaugenommen nur, daß es keinen Algorithmus gibt, der die *optimale Lösung* für *alle* Ausprägungen eines Problems in polynomialer Zeit findet. Schränkt man diese strenge Bedingung ein, so kommt man in vielen Fällen zu praktisch anwendbaren Algorithmen.

Die erste Möglichkeit besteht darin, die Forderung, daß ein Algorithmus für *alle* Ausprägungen eines Problems die optimale Lösung finden soll, fallen zu lassen. In vielen Anwendungen haben die zu untersuchenden Probleme eine stark eingeschränkte Struktur. Für diese findet man eventuell einen effizienten Algorithmus. Ein Beispiel hierfür ist das Problem der Hamiltonschen Kreise. Es gibt einen effizienten Algorithmus, welcher in *fast jedem* Graphen einen Hamiltonschen Kreis findet, sofern es einen solchen gibt. Der Beweis einer solchen Aussage beruht meistens auf einer Annahme über die Wahrscheinlichkeitsverteilung aller Eingaben. Leider ist es häufig sehr schwer, zu entscheiden, ob in einer gegebenen Situation diese Annahme erfüllt ist. Solche Algorithmen werden *probabilistische Algorithmen* genannt.

Die prinzipielle Vorgehensweise eines probabilistischen Algorithmus sei an dem Algorithmus von J. Turner für das Färbungsproblem erläutert. Es sei \mathcal{G}_n^c die Menge aller ungerichteten Graphen G mit n Ecken und $\chi(G) \leq c$. Die Wahrscheinlichkeit, daß der Algorithmus von Turner für einen beliebigen Graphen $G \in \mathcal{G}_n^c$ eine c -Färbung findet, strebt mit wachsendem n und festem c gegen 1. Dabei werden allerdings für konkrete Werte von n und c keine Angaben gemacht. Die Laufzeit des Algorithmus ist $O(n + m \log n)$.

Die zweite Möglichkeit besteht darin, die Forderung, daß ein Algorithmus immer die *optimale Lösung* findet, fallen zu lassen. In vielen praktischen Anwendungen genügt es, eine Lösung zu finden, die hinreichend nahe am Optimum liegt. Häufig werden Verfahren eingesetzt, deren Ergebnisse in vielen Fällen nahe an einer optimalen Lösung liegen. Solche Verfahren entstehen oft aus praktischen Erfahrungen und empirischen Untersuchungen. Können keine Garantien gegeben werden, wie *gut* das gewonnene Resultat ist, so nennt man solche Verfahren *Heuristiken*. Interessant sind vor allem solche Verfahren, für die es eine Garantie über die Qualität der gefundenen Resultate gibt. Sie werden approximative Algorithmen genannt.

Approximative Algorithmen stehen im Mittelpunkt dieses Kapitels. Zunächst werden die Begriffe Optimierungsproblem und approximativer Algorithmus exakt erfaßt.

Ein Optimierungsproblem P ist ein Minimierungs- oder Maximierungsproblem, das aus drei Teilen besteht:

- eine Menge \mathcal{A}_P von Ausprägungen von P und eine Längenfunktion l , die jedem $a \in \mathcal{A}_P$ eine natürliche Zahl zuordnet; $l(a)$ ist die Länge von a .
- für jede Ausprägung $a \in \mathcal{A}_P$ eine Menge $\mathcal{L}_P(a)$ von zulässigen Lösungen von a .
- eine Funktion m_P , welche jeder Lösung $L \in \mathcal{L}_P(a)$ eine positive Zahl $m_P(L)$ zuordnet, den Wert der Lösung L .

Ein Minimierungsproblem P besteht nun darin, zu einer Ausprägung $a \in \mathcal{A}_P$ eine Lösung $L \in \mathcal{L}_P(a)$ zu finden, für die $m_P(L)$ minimal ist.

Für das Optimierungsproblem P der Bestimmung der chromatischen Zahl ist \mathcal{A}_P die Menge aller ungerichteten Graphen; die Länge einer Ausprägung ist die Anzahl n der Ecken des Graphen, die zulässigen Lösungen $\mathcal{L}_P(a)$ sind die Färbungen des Graphen und der Wert $m_P(L)$ einer Lösung ist die Anzahl der Farben, welche die Färbung vergibt.

Ein *approximativer Algorithmus* für ein Optimierungsproblem P ist ein polynomialer Algorithmus, der jeder Ausprägung $a \in \mathcal{A}_P$ eine zulässige Lösung aus $\mathcal{L}_P(a)$ zuordnet. Ist A ein approximativer Algorithmus für P und $a \in \mathcal{A}_P$, so bezeichnet man mit $A(a)$ den Wert der Lösung, die Algorithmus A für die Ausprägung a liefert. Den Wert einer optimalen Lösung aus $\mathcal{L}_P(a)$ bezeichnet man mit $OPT(a)$. Da nur Algorithmen mit polynomialer Laufzeit für Probleme aus NPC interessant sind, wurde dies in die Definition eines approximativen Algorithmus aufgenommen.

Für das Färbungsproblem ist $OPT(a) = \chi(a)$ und $A(a)$ die Anzahl der Farben, die A für a vergibt. In Kapitel 5 wurde mit dem Greedy-Algorithmus (Abbildung 5.2) ein approximativer Algorithmus zur Bestimmung einer minimalen Färbung eines ungerichteten Graphen vorgestellt. Dort wurde auch schon festgestellt, daß der Greedy-Algorithmus nicht in allen Fällen eine optimale Färbung liefert. Die Qualität der vom Greedy-Algorithmus erzeugten Lösung hängt stark von der Reihenfolge ab, in der die Ecken betrachtet werden.

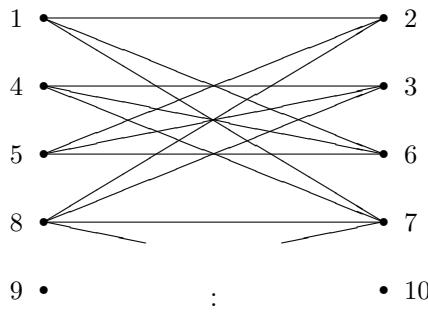


Abbildung 9.1: Ein bipartiter Graph mit $2n$ Ecken, für den der Greedy-Algorithmus mindestens n Farben benötigt

Es gibt Beispiele, für die der Greedy-Algorithmus ein extrem schlechtes Verhalten zeigt. Im folgenden wird für jedes $n \in \mathbb{N}$ mit $n \geq 3$ ein bipartiter Graph mit $2n$ Ecken konstruiert, für dessen Färbung der Greedy-Algorithmus mindestens n Farben benötigt. Die Konstruktion geht von dem vollständig bipartiten Graphen $K_{n,n}$ aus. Die Ecken werden wie in Abbildung 9.1 numeriert. Aus diesem Graphen werden nun maximal n Kanten entfernt. Für Ecken i mit $i \equiv 0(4)$ werden die Kanten von i nach $i - 2$ entfernt und für Ecken i mit $i \equiv 1(4)$ die Kanten von i nach $i + 2$, sofern diese Kanten vorhanden sind. Den so entstandenen Graphen nennen wir G . Der Greedy-Algorithmus färbt die Ecken 1 und 3 dieses Graphen mit Farbe 1 und die Ecken 2 und 4 mit Farbe 2. Keine

der restlichen Ecken wird mit Farbe 1 oder 2 gefärbt. Das gleiche gilt für die nächsten beiden Farben. Somit bekommen jeweils maximal zwei Ecken die gleiche Farbe. Also werden mindestens n Farben verwendet.

Dieses Beispiel zeigt, daß die von dem Greedy-Algorithmus erzeugte Färbung beliebig weit entfernt von der optimalen Lösung liegen kann. Die einzige Garantie ist die, daß maximal $\Delta + 1$ Farben verwendet werden, wobei Δ der maximale Eckengrad ist. Im obigen Beispiel ist $\Delta = n$ oder $\Delta = n - 1$.

Im folgenden werden nun einige Maßzahlen eingeführt, mit denen die Qualität eines approximativen Algorithmus charakterisiert werden kann. Dies ist zum einen wichtig, um verschiedene approximative Algorithmen zu vergleichen, und zum anderen, um eine Abschätzung der Abweichung der erzeugten Lösung von der optimalen Lösung zu haben.

9.3 Absolute Qualitätsgarantien

Ein approximativer Algorithmus für das Färbungsproblem wäre sicherlich akzeptabel, wenn er immer nur ein oder zwei Farben mehr als notwendig verwenden würde. Allgemein wäre es wünschenswert, wenn der Wert der gefundenen Lösung sich nur um einen konstanten Wert von der optimalen Lösung unterscheidet. Ein Kriterium für die Güte eines approximativen Algorithmus ist der absolute Fehler

$$|OPT(a) - A(a)|.$$

Es wird sich zeigen, daß es nur wenige Beispiele gibt, für die der absolute Fehler beschränkt werden kann. Der Greedy-Algorithmus hat offensichtlich nicht diese Eigenschaft. Ist $\mathcal{P} = \mathcal{NP}$, so gibt es für jedes Optimierungsproblem aus \mathcal{NPC} einen polynomialen Algorithmus zur Bestimmung der optimalen Lösung. Falls man also zeigen möchte, daß es für ein Problem keinen approximativen Algorithmus mit beschränktem absoluten Fehler gibt, so muß man dies unter der Annahme $\mathcal{P} \neq \mathcal{NP}$ tun. Das folgende Lemma ist ein Beispiel dafür, daß die Beschränktheit des absoluten Fehlers eine sehr starke Eigenschaft ist.

Lemma. Es sei C eine natürliche Zahl und A ein approximativer Algorithmus für die Bestimmung der Unabhängigkeitszahl eines ungerichteten Graphen. Gilt $|OPT(G) - A(G)| \leq C$ für alle ungerichteten Graphen G , so ist $\mathcal{P} = \mathcal{NP}$.

Beweis. Es wird ein polynomialer Algorithmus A' konstruiert, der die Unabhängigkeitszahl exakt bestimmt. Da man gezeigt hat, daß dieses Problem in \mathcal{NPC} liegt, folgt damit sofort $\mathcal{P} = \mathcal{NP}$. Es sei G ein ungerichteter Graph. Der Algorithmus A' konstruiert zunächst einen neuen Graphen G' . Dieser besteht aus $C + 1$ Kopien von G , d.h. G' hat $(C + 1)$ -mal soviele Kanten und Ecken wie G . Es folgt sofort, daß $\alpha(G') = (C + 1)\alpha(G)$ ist. Nun wird der Algorithmus A auf G' angewendet und eine unabhängige Menge U für G' gebildet. Dann besteht U aus $A(G')$ Ecken. Da der absolute Fehler von A beschränkt ist, gilt:

$$|OPT(G') - A(G')| = |(C + 1)OPT(G) - A(G')| \leq C.$$

Nun bilden für jede Kopie von G die Ecken, die in U liegen, eine unabhängige Menge für diese Kopie. Auf diese Weise konstruiert A' eine unabhängige Menge für G , welche mindestens $\lceil A(G')/(C+1) \rceil$ Ecken enthält. Somit folgt

$$|OPT(G) - A'(G)| \leq |OPT(G) - A(G')/(C+1)| \leq C/(C+1) < 1.$$

Da $A'(G)$ eine natürliche Zahl ist, bestimmt A' exakt die Unabhängigkeitszahl von G . A' ist ein polynomialer Algorithmus, da C eine von G unabhängige Konstante ist. ■

Eine ähnliche Aussage läßt sich noch für viele andere Probleme aus \mathcal{NPC} beweisen. Die Beweise sind im Aufbau alle gleich. Der entscheidende Punkt ist die Konstruktion einer neuen Ausprägung G' . Im obigen Fall bestand G' aus $C+1$ Kopien von G . Im folgenden Lemma wird eine analoge Aussage für das Färungsproblem bewiesen.

Lemma. Es sei C eine natürliche Zahl und A ein approximativer Algorithmus für die Bestimmung der chromatischen Zahl eines ungerichteten Graphen. Gilt $|OPT(G) - A(G)| \leq C$ für alle ungerichteten Graphen G , so ist $\mathcal{P} = \mathcal{NP}$.

Beweis. Es wird ein polynomialer Algorithmus A' konstruiert, welcher die chromatische Zahl exakt bestimmt. Da man gezeigt hat, daß dieses Problem in \mathcal{NPC} liegt, folgt damit sofort $\mathcal{P} = \mathcal{NP}$. Es sei G ein ungerichteter Graph. Der Algorithmus A' konstruiert zunächst wieder einen neuen Graphen G' . Dieser besteht aus $C+1$ Kopien von G und zusätzlichen Kanten zwischen je zwei Ecken aus verschiedenen Kopien; d.h. G' hat $(C+1)n$ Ecken und $(C+1)(n^2C/2 + m)$ Kanten. Die Konstruktion von G' erfolgt in polynomialer Zeit. Es folgt sofort, daß $\chi(G') = (C+1)\chi(G)$ ist. Nun wird der Algorithmus A auf G' angewendet und eine Färbung für G' gebildet. Diese verwendet $A(G')$ Farben. Da der absolute Fehler von A beschränkt ist, gilt:

$$|OPT(G') - A(G')| = |(C+1)OPT(G) - A(G')| \leq C$$

Die Färbung für G' induziert auf jeder Kopie von G eine Färbung. Da je zwei Ecken aus verschiedenen Kopien inzident sind, kann aus dieser Färbung für G' eine Färbung für G erzeugt werden, die mindestens $\lceil A(G')/(C+1) \rceil$ Farben verwendet. Somit folgt

$$|OPT(G) - A'(G)| \leq |OPT(G) - A(G')/(C+1)| \leq C/(C+1) < 1.$$

Da $A'(G)$ eine natürliche Zahl ist, bestimmt A' exakt die chromatische Zahl von G . A' ist ein polynomialer Algorithmus, da die Anwendung von A auf G' in polynomialer Zeit erfolgt (C ist eine von G unabhängige Konstante). ■

Es gibt aber auch einige Optimierungsprobleme in \mathcal{NPC} , für die approximative Algorithmen mit beschränktem absoluten Fehler bekannt sind. Dazu wird noch einmal das Färungsproblem betrachtet. Man hat gezeigt, daß das Entscheidungsproblem, ob ein planarer Graph eine 3-Färbung besitzt, in \mathcal{NPC} liegt. Für das dazugehörige Optimierungsproblem läßt sich leicht ein approximativer Algorithmus mit beschränktem Wirkungsgrad angeben. Dies wurde schon in Kapitel 5 getan. Die dort angegebene Prozedur 5-färbung bestimmt für planare Graphen eine Färbung, die maximal fünf Farben verwendet. Nach dem Vier-Farben-Satz läßt sich jeder planare Graph mit vier Farben

färben. Da man mit Hilfe der Tiefensuche feststellen kann, ob ein Graph eine 2-Färbung besitzt, kann folgender Satz leicht bewiesen werden.

Satz. Es gibt einen approximativen Algorithmus A zur Bestimmung von Färbungen für planare Graphen mit $|OPT(G) - A(G)| \leq 2$ für alle planaren Graphen G .

In Kapitel 5 wurden auch Kantenfärbungen für Graphen betrachtet. Die kantenchromatische Zahl ist sehr eng mit dem größten Eckengrad Δ verbunden. Nach dem Satz von Vizing ist die kantenchromatische Zahl eines Graphen gleich Δ oder gleich $\Delta + 1$. Das Entscheidungsproblem, welcher der beiden Fälle vorliegt, ist erstaunlicherweise ein \mathcal{NP} -vollständiges Problem. Der von Vizing angegebene Beweis lässt sich aber in einen effizienten Algorithmus umsetzen, der immer eine Kantenfärbung mit maximal $\Delta + 1$ Farben findet. In diesem Fall ist der absolute Fehler sogar höchstens gleich eins. Der auf dem Satz von Vizing basierende Algorithmus wird in diesem Buch nicht dargestellt, ein einfacher approximativer Algorithmus für die Bestimmung der kantenchromatischen Zahl wird in Aufgabe 24 behandelt.

Die letzten beiden Beispiele zeigen, daß es Optimierungsprobleme gibt, für die approximative Algorithmen mit beschränktem absolutem Fehler existieren. Die beiden betrachteten Probleme haben eine sehr spezielle Struktur: Die Werte der Lösungen liegen alle in einem sehr schmalen Intervall, das unabhängig von der eigentlichen Ausprägung ist. Bis heute sind keine Optimierungsprobleme aus \mathcal{NPC} bekannt, für die es einen approximativen Algorithmus gibt, dessen absoluter Fehler beschränkt ist, und welche nicht diese Eigenschaft haben.

9.4 Relative Qualitätsgarantien

Die Ergebnisse aus dem letzten Abschnitt zeigen, daß die Forderung nach der Beschränktheit des absoluten Fehlers eines approximativen Algorithmus für Probleme aus \mathcal{NPC} zu stark ist. Aus diesem Grund verwendet man den relativen Fehler zur Bewertung der Güte eines approximativen Algorithmus. Mit Hilfe des relativen Fehlers kann nun der *Wirkungsgrad* \mathcal{W}_A eines approximativen Algorithmus definiert werden. Dazu sei P ein Minimierungsproblem und A ein approximativer Algorithmus für P und $n \in \mathbb{N}$. Dann ist

$$\mathcal{W}_A(n) = \sup \left\{ \frac{A(a)}{OPT(a)} \mid a \in \mathcal{A}_P \text{ und } l(a) \leq n \right\}$$

der Wirkungsgrad¹ von A für n . Je näher der Wirkungsgrad bei 1 liegt, desto besser ist die Approximation. Um einen Ausdruck zu bekommen, der unabhängig von n ist, definiert man den *asymptotischen Wirkungsgrad* \mathcal{W}_A^∞ . Ist die Folge $\mathcal{W}_A(n)$ unbeschränkt, so setzt man $\mathcal{W}_A^\infty = \infty$ und andernfalls

$$\mathcal{W}_A^\infty = \inf \left\{ r \mid \frac{A(a)}{OPT(a)} \leq r \text{ für alle } a \in \mathcal{A}_P \text{ bis auf endlich viele Ausnahmen} \right\},$$

¹sup bezeichnet das Supremum, d.h. die kleinste obere Schranke, und inf das Infimum, d.h. die größte untere Schranke.

falls P ein Minimierungsproblem ist. Für Maximierungsprobleme ersetzt man in den obigen Definitionen jeweils

$$\frac{A(a)}{OPT(a)} \quad \text{durch} \quad \frac{OPT(a)}{A(a)}.$$

Für das Färbungsproblem ist

$$\mathcal{W}_A(n) = \max \left\{ \frac{A(G)}{\chi(G)} \mid G \text{ ungerichteter Graph mit höchstens } n \text{ Ecken} \right\}.$$

Wie bestimmt man den Wirkungsgrad eines approximativen Algorithmus? Die exakte Bestimmung ist meistens sehr aufwendig, deshalb werden oft nur untere Schranken bestimmt. Dazu wird eine Folge von Graphen konstruiert, für die der Algorithmus schlecht arbeitet. Damit kann dann zumindest eine untere Schranke für $\mathcal{W}_A(n)$ angegeben werden.

Was ist der Wirkungsgrad des Greedy-Algorithmus? Hierbei beachte man, daß der Greedy-Algorithmus von der Reihenfolge, in der die Ecken betrachtet werden, abhängt; d.h. zur Bestimmung des Wirkungsgrades müssen alle Numerierungen der Ecken betrachtet werden. Die in Abbildung 9.1 angegebene Folge von bipartiten Graphen liefert eine untere Schranke für den Wirkungsgrad. Die Graphen sind bipartit, und der Algorithmus vergibt mindestens halb so viele Farben, wie der Graph Ecken hat. Somit ist $\mathcal{W}_A(n) \geq n/4$ und $\mathcal{W}_A^\infty = \infty$.

Im folgenden wird gezeigt werden, daß es einige Optimierungsprobleme in \mathcal{NP} gibt, für die es approximative Algorithmen mit beschränktem asymptotischen Wirkungsgrad gibt. Es gibt aber auch einige Probleme, für die man zeigen kann, daß es unter der Voraussetzung $\mathcal{P} \neq \mathcal{NP}$ einen solchen Algorithmus nicht geben kann. Für viele Probleme ist diese Frage aber noch völlig offen.

Zunächst wird das Problem der Eckenüberdeckung betrachtet. Es sei G ein ungerichteter Graph mit Eckenmenge E . Eine *Eckenüberdeckung* für G ist eine Teilmenge E' von E , so daß jede Kante von G mindestens eine Endcke in E' hat. Eine Eckenüberdeckung heißt *minimal*, wenn es keine andere Eckenüberdeckung mit weniger Ecken gibt. Das Entscheidungsproblem, ob ein gegebener Graph eine Eckenüberdeckung mit höchstens C Ecken hat, ist \mathcal{NP} -vollständig. Das Optimierungsproblem kann mit polynomialem Aufwand auf das Entscheidungsproblem reduziert werden (siehe Aufgabe 39).

Für bipartite Graphen existiert ein effizienter Algorithmus für dieses Problem (vergleichen Sie Aufgabe 25 aus Kapitel 7 auf Seite 238). Für Bäume kann eine minimale Eckenüberdeckung in linearer Zeit bestimmt werden (vergleichen Sie Aufgabe 27). Im folgenden wird ein Algorithmus vorgestellt, der eine minimale Eckenüberdeckung für einen ungerichteten Graphen G explizit bestimmt. Der Algorithmus baut implizit einen Binärbaum B auf und durchläuft B mittels der Breitensuche. Die Ecken des Binärbaumes sind Paare (U, T) , wobei U ein Untergraph von G und T eine Teilmenge der Ecken von G ist. Die Wurzel von B ist (G, \emptyset) . Es sei (U, T) eine Ecke mit $U \neq \emptyset$ und (e, f) eine Kante von U . Die beiden Nachfolger von (U, T) sind wie folgt definiert:

- $(U_e, T \cup \{e\})$, wobei U_e aus U entsteht, indem alle zu e inzidenten Kanten und alle isolierten Ecken entfernt werden.

- $(U_f, T \cup \{f\})$, wobei U_f aus U entsteht, indem alle zu f inzidenten Kanten und alle isolierten Ecken entfernt werden.

Auf jedem Niveau von B gibt es eine Ecke (U, T) , so daß die Vereinigung von T mit einer minimalen Eckenüberdeckung von U eine minimale Eckenüberdeckung von G bildet. Diese Aussage wird mit vollständiger Induktion nach der Höhe des Niveaus bewiesen. Dies Aussage gilt trivialerweise für die Wurzel. Es sei nun (U^i, T^i) die Ecke auf Niveau i , für die die Behauptung zutrifft. Wählt man eine beliebige Kante (e, f) aus U^i , so muß e oder f in einer minimalen Eckenüberdeckung von U^i liegen. Es sei E_l bzw. E_r eine minimale Eckenüberdeckung von U_e^i bzw. U_f^i . Dann sind $\{e\} \cup E_l$ und $\{f\} \cup E_r$ Eckenüberdeckungen von U und eine von beiden muß eine minimale Überdeckung sein. Die Behauptung folgt nun aus der Induktionsvoraussetzung.

Ist (U, T) eine Ecke auf Niveau i , so ist $|T| = i$. Es sei (U, T) die erste durch die Breitensuche gefundene Ecke mit $U = \emptyset$. Dann ist T eine Eckenüberdeckung für G . Es sei (U_0, T_0) eine Ecke auf dem gleichen Niveau. Wegen $|T| = |T_0|$ folgt aus der gerade bewiesenen Eigenschaft des Binärbaumes, daß T eine minimale Eckenüberdeckung von G ist. Bis zum Erreichen einer Ecke (\emptyset, T) sind maximal $1 + 2 + 4 + \dots + 2^{OPT-1} = 2^{OPT}$ Ecken von B zu erzeugen, jeweils mit Aufwand $O(n)$. Hierbei ist OPT die Mächtigkeit einer minimalen Eckenüberdeckung von G . Es gilt also folgender Satz.

Satz. Eine minimale Eckenüberdeckung eines ungerichteten Graphen kann mit Aufwand $O(2^{OPT} n)$ bestimmt werden.

Eine sehr naheliegende Vorgehensweise für einen approximativen Algorithmus bildet ein Greedy-Algorithmus. Hierbei werden die Kanten in einer beliebigen Reihenfolge betrachtet, und wenn keine der beiden Enden der aktuellen Kante schon in der Eckenüberdeckung ist, so wird eine beliebige Endecke der Kante in die aktuelle Eckenüberdeckung eingefügt. Abbildung 9.2 zeigt die Struktur dieses Algorithmus A_1 .

```

 $E' := \emptyset;$ 
Initialisiere  $K$  mit der Menge der Kanten von  $G$ ;
while  $K \neq \emptyset$  do begin
    Es sei  $k = (e, f)$  eine beliebige Kante aus  $K$ ;
    Füge  $e$  in  $E'$  ein;
    Entferne aus  $K$  alle zu  $e$  inzidenten Kanten;
end

```

Abbildung 9.2: Der Algorithmus A_1

Man zeigt leicht, daß der Algorithmus A_1 eine Eckenüberdeckung E' erzeugt. Um den Wirkungsgrad zu bestimmen, wird eine Folge von bipartiten Graphen G_r konstruiert ($r \in \mathbb{N}$). Die Eckenmenge ist hierbei in die disjunkten Mengen L und R aufgeteilt, wobei L genau r Ecken enthält. Die Menge R ist wiederum in r disjunkte Mengen R_i mit je $[r/i]$ Ecken aufgeteilt. Jede Ecke aus R_i ist zu genau i Ecken in L benachbart, und je zwei Ecken aus R_i haben in L keinen gemeinsamen Nachbarn. Es kann sein, daß

nicht jede Ecke aus L einen Nachbarn in jeder Menge R_i hat. Es folgt, daß jede Ecke aus R_i den Eckengrad i und jede Ecke aus L maximal den Eckengrad r hat.

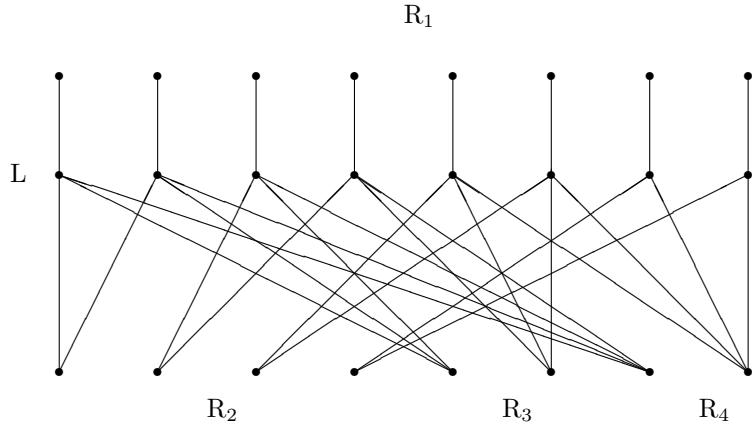


Abbildung 9.3: Der Graphen G_8 ohne die Ecken in R_5 bis R_8

Abbildung 9.3 zeigt den Graphen G_8 . Die Mengen R_5 bis R_8 enthalten jeweils genau eine Ecke. Diese Ecken und die dazugehörigen Kanten sind hier nicht dargestellt. Der Graph G_r hat genau

$$r + \sum_{i=1}^r \lfloor r/i \rfloor$$

Ecken. Die Menge L ist eine minimale Eckenüberdeckung mit r Ecken, somit gilt $OPT(G_r) = r$.

Wendet man den Algorithmus A_1 auf die Graphen G_r an, so kann es passieren, daß die Kanten in einer Reihenfolge betrachtet werden, so daß ihre Enden in R der Reihe nach in den Mengen R_r, R_{r-1}, \dots liegen. Wird hierbei jeweils die in R liegende Ecke in E' eingefügt, dann bestimmt A_1 die Menge R als Eckenüberdeckung. Daraus folgt

$$A_1(G_r) = \sum_{i=1}^r \lfloor r/i \rfloor \approx r \log_2 r.$$

Also ist $\mathcal{W}_{A_1}(n) = O(\log n)$ und damit nicht beschränkt. Somit ist der asymptotische Wirkungsgrad von Algorithmus A_1 gleich ∞ .

Durch eine kleine Veränderung von Algorithmus A_1 erhält man einen Algorithmus mit beschränktem asymptotischen Wirkungsgrad. Anstatt nur eine Ecke jeder ausgewählten Kante in E' einzufügen, werden beide Enden eingefügt. Obwohl diese Vorgehensweise auf den ersten Blick nicht sehr wirksam aussieht, ist der asymptotische Wirkungsgrad gleich 2. Abbildung 9.4 zeigt die Struktur dieses Algorithmus A_2 .

Lemma. Es gilt $\mathcal{W}_{A_2}^\infty = 2$.

```

E' := ∅;
Initialisiere K mit der Menge der Kanten von G;
while K ≠ ∅ do begin
    Es sei k = (e,f) eine beliebige Kante aus K;
    Füge e und f in E' ein;
    Entferne aus K alle zu e oder f inzidenten Kanten;
end

```

Abbildung 9.4: Der Algorithmus A_2

Beweis. Man sieht sofort, daß E' eine Eckenüberdeckung ist. Es sei Z die Menge aller Kanten, deren Enden in E' eingefügt worden sind. Die Menge Z bildet eine Zuordnung des Graphen G . Es sei Z_{max} eine maximale Zuordnung des Graphen. Jede Eckenüberdeckung muß mindestens eine Ecke von jeder Kante aus Z_{max} enthalten. Somit gilt:

$$A_2(G)/2 = |E'|/2 = |Z| \leq |Z_{max}| \leq OPT(G)$$

Daraus folgt $\mathcal{W}_{A_2}(n) \leq 2$. Wendet man den Algorithmus A_2 auf die vollständig bipartiten Graphen $K_{n,n}$ an, so erhält man jeweils eine Eckenüberdeckung mit $2n$ Ecken; eine minimale Eckenüberdeckung besteht aber aus n Ecken. Damit ist das Lemma bewiesen. ■

Ein weiterer Approximationsalgorithmus mit linearem Aufwand und Wirkungsgrad 2 wird in Aufgabe 28 behandelt. Zur Zeit ist kein Algorithmus mit einem besseren Wirkungsgrad bekannt. Das Problem der minimalen Eckenüberdeckung ist eng verwandt mit dem Problem der maximalen unabhängigen Menge in einem Graphen. Dies zeigt das folgende einfache Lemma.

Lemma. Es sei G ein ungerichteter Graph mit Eckenmenge E . Eine Menge $E' \subset E$ ist genau dann eine Eckenüberdeckung von G , wenn $E \setminus E'$ eine unabhängige Menge von G ist.

Leider kann man aus diesem Lemma nicht folgern, daß es einen approximativen Algorithmus mit beschränktem asymptotischen Wirkungsgrad zur Bestimmung einer maximalen unabhängigen Menge gibt. Dazu betrachte man einen Graphen G mit n Ecken, dessen minimale Eckenüberdeckung $n/2 - 1$ Ecken enthält. Der Algorithmus A_2 findet für diesen Graphen eine Eckenüberdeckung mit maximal $n - 2$ Ecken. Dies bedeutet aber, daß die daraus gewonnene unabhängige Menge im schlechtesten Fall nur zwei Ecken enthält, die maximale unabhängige Menge aber enthält $n/2 + 1$ Ecken.

Im folgenden wird der bestmögliche Wirkungsgrad $\mathcal{W}_{MIN}(P)$ eines approximativen Algorithmus für ein Problem P definiert:

$$\mathcal{W}_{MIN}(P) = \inf \{r \mid A \text{ approximativer Algorithmus für } P \text{ mit } \mathcal{W}_A^\infty \leq r\}$$

Ist $\mathcal{P} = \mathcal{NP}$, so ist $\mathcal{W}_{MIN}(P) = 1$ für jedes Problem P aus \mathcal{NP} . Aus diesem Grund nehmen wir für den Rest dieses Abschnitts an, daß $\mathcal{P} = \mathcal{NP}$ gilt. Die optimale Situation

für ein Problem P liegt dann vor, wenn $\mathcal{W}_{MIN}(P) = 1$ ist. Man beachte, daß dies nicht bedeutet, daß es einen polynomiauen Algorithmus für P gibt. Es bedeutet lediglich, daß es zu jedem $\epsilon > 1$ einen approximativen Algorithmus A_ϵ für P mit $\mathcal{W}_{A_\epsilon}^\infty \leq \epsilon$ gibt. Die Laufzeit der Algorithmen A_ϵ hängt von ϵ ab. Unter der Voraussetzung $\mathcal{P} \neq \mathcal{NP}$ ist sie um so größer, je näher ϵ an 1 herankommt. Gibt es für ein Problem P keinen approximativen Algorithmus mit beschränktem Wirkungsgrad, so setzt man $\mathcal{W}_{MIN}(P) = \infty$.

Um zu untersuchen, wie sich \mathcal{W}_{MIN} für das Problem der Bestimmung einer maximalen unabhängigen Menge verhält, muß zunächst das Produkt von zwei Graphen definiert werden. Es seien G_1 und G_2 ungerichtete Graphen mit den Eckenmengen E_1 bzw. E_2 . Dann ist das Produkt $G_1 \circ G_2$ der Graphen G_1 und G_2 ein ungerichteter Graph mit der Eckenmenge $E_1 \times E_2$. Zwischen den beiden Ecken (e_1, e_2) und (f_1, f_2) gibt es genau dann eine Kante, wenn eine der folgenden beiden Bedingungen erfüllt ist: (e_1, f_1) ist eine Kante in G_1 oder $e_1 = f_1$ und (e_2, f_2) ist eine Kante von G_2 . Das Produkt zweier Graphen ist nicht kommutativ, d.h. im allgemeinen ist $G_1 \circ G_2 \neq G_2 \circ G_1$. Abbildung 9.5 zeigt einen Graphen G und den Graphen $G \circ G$.

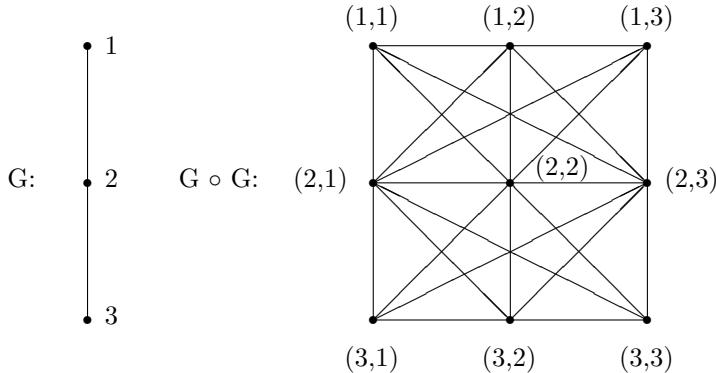


Abbildung 9.5: Ein ungerichteter Graph G und $G \circ G$

Die maximalen unabhängigen Mengen eines Produkts setzen sich gerade aus den unabhängigen Mengen der beiden Faktoren zusammen.

Lemma. Es seien G_1 und G_2 ungerichtete Graphen. Dann gilt:

$$\alpha(G_1 \circ G_2) = \alpha(G_1)\alpha(G_2)$$

Beweis. Es seien U_1 und U_2 unabhängige Mengen von G_1 bzw. G_2 . Dann ist $U_1 \times U_2$ eine unabhängige Menge von $G_1 \circ G_2$. Somit ist $\alpha(G_1 \circ G_2) \geq \alpha(G_1)\alpha(G_2)$. Sei U eine maximale unabhängige Menge von $G_1 \circ G_2$ und U_1 die Menge aller Ecken e aus G_1 , für die es eine Ecke f aus G_2 mit $(e, f) \in U$ gibt. Nach Konstruktion bildet U_1 eine unabhängige Menge in G_1 . Für $e \in U_1$ sei U_2 die Menge aller Ecken f aus G_2 , so daß $(e, f) \in U$ gilt. Dann ist U_2 eine unabhängige Menge in G_2 . Nun kann die Ecke e so

gewählt werden, daß $|U_1||U_2| \geq |U|$ gilt. Daraus folgt $\alpha(G_1)\alpha(G_2) \geq \alpha(G_1 \circ G_2)$, und damit ist das Lemma bewiesen. ■

Satz. Für das Optimierungsproblem der unabhängigen Mengen in ungerichteten Graphen gilt entweder $\mathcal{W}_{MIN} = 1$ oder $\mathcal{W}_{MIN} = \infty$.

Beweis. Angenommen, es gilt $\mathcal{W}_{MIN} < \infty$. Dann gibt es eine Zahl $w > 1$ und einen approximativen Algorithmus A_1 mit $\mathcal{W}_{A_1}^\infty < w$. Mit Hilfe von A_1 wird ein neuer approximativer Algorithmus A_2 konstruiert, so daß

$$\mathcal{W}_{A_2}^\infty < \sqrt{w}$$

gilt. Der Algorithmus A_2 konstruiert zu einem gegebenen Graphen G den Graphen $G \circ G$. Der Algorithmus A_1 findet in diesem Graphen eine unabhängige Menge U , so daß

$$\frac{\alpha(G \circ G)}{|U|} < w$$

gilt. Nach dem letzten Lemma kann man mit Hilfe von U eine unabhängige Menge U_1 in G finden, so daß gilt:

$$\frac{\alpha(G)}{|U_1|} < \sqrt{w}$$

Die einzelnen Schritte können alle in polynomialer Zeit durchgeführt werden. Somit gilt $\mathcal{W}_{A_2}^\infty < \sqrt{w}$. Auf diese Art kann man eine Folge von approximativen Algorithmen konstruieren, deren asymptotischer Wirkungsgrad gegen 1 konvergiert. Somit folgt $\mathcal{W}_{MIN} = 1$. ■

Erst 1992 haben S. Arora und S. Safra gezeigt, daß es keinen approximativen Algorithmus mit beschränktem Wirkungsgrad für die Bestimmung von maximalen unabhängigen Mengen gibt, falls $\mathcal{NP} \neq \mathcal{P}$ gilt. In Aufgabe 31 wird ein approximativer Algorithmus für dieses Problem diskutiert. Es gibt noch weitere Optimierungsprobleme, von denen man $\mathcal{W}_{MIN} = \infty$ nachgewiesen hat. In den folgenden Abschnitten werden zwei solcher Probleme ausführlicher behandelt: die Färbung von Graphen und das Traveling-Salesman Problem.

9.5 Approximative Färbungsalgorithmen

In diesem Abschnitt werden approximative Algorithmen für das Färbungsproblem diskutiert. Zunächst wird das Verhalten des Greedy-Algorithmus näher untersucht. Die Anzahl der Farben, die dieser Algorithmus vergibt, ist durch $\Delta + 1$ beschränkt. Es gibt auch eine obere Abschätzung in Abhängigkeit der Anzahl der Kanten des Graphen. Für

Graphen mit wenigen Kanten liefert der Greedy-Algorithmus häufig gute Approximationen. Der folgende Satz liefert eine obere Schranke für die Anzahl der Farben, die der Greedy-Algorithmus vergibt.

Satz. Der Greedy-Algorithmus verwendet höchstens $\lceil \sqrt{2m} \rceil$ Farben, wobei m die Anzahl der Kanten des Graphen ist.²

Beweis. Der Beweis wird mittels vollständiger Induktion nach m geführt. Für $m = 1$ braucht der Greedy-Algorithmus genau zwei Farben, und es gilt $\lceil \sqrt{2} \rceil = 2$. Sei nun $m > 1$. Es sei M die Menge der Ecken, die mit der ersten Farbe gefärbt werden. Es sei G' der von den restlichen Ecken induzierte Untergraph von G . Ferner sei

$$A = \sum_{e \in M} g(e)$$

die Summe der Eckengrade der Ecken aus M . Dann hat G' genau $m - A$ Kanten, und nach Induktionsannahme benötigt der Greedy-Algorithmus maximal $\lceil \sqrt{2(m - A)} \rceil$ Farben für G' . Das heißt, für G benötigt er

$$\lceil \sqrt{2(m - A)} \rceil + 1 = \lceil \sqrt{2(m - A)} + 1 \rceil$$

Farben. Es genügt also, zu zeigen, daß $\sqrt{2m} \geq \sqrt{2(m - A)} + 1$ ist. Dazu beachte man, daß G' maximal A Ecken hat, denn jede Ecke von G' ist zu einer Ecke aus M inzident. Somit gilt:

$$\frac{A(A - 1)}{2} \geq m - A$$

Hieraus folgt, daß $A^2 + A \geq 2m$ bzw. $(2A + 1)^2 \geq 8m$ ist. Somit ist $2A + 1 \geq 2\sqrt{2m}$, und es ergibt sich leicht $\sqrt{2m} + 1 \geq \sqrt{2(m - A)} + 2$. ■

Im folgenden wird ein Algorithmus von D.S. Johnson vorgestellt, der einen besseren Wirkungsgrad als der Greedy-Algorithmus hat. Es handelt sich dabei um eine Verfeinerung des Greedy-Algorithmus, bei der die Ecken nicht mehr in einer beliebigen Reihenfolge betrachtet werden. Zuerst wird die Ecke mit minimalem Eckengrad mit Farbe 1 gefärbt. Danach wird diese Ecke samt ihrer Nachbarn und den mit diesen Ecken inzidenten Kanten entfernt. In diesem Graphen wird wieder die Ecke mit minimalem Eckengrad mit Farbe 1 gefärbt, und der obige Prozeß wird wiederholt, bis keine Ecke mehr übrig ist. Danach wird der Graph, bestehend aus den noch nicht gefärbten Ecken und den zugehörigen Kanten, betrachtet. Auf diesen Graphen wird wieder das obige Verfahren mit Farbe 2 angewendet etc. Abbildung 9.6 zeigt eine Realisierung des Algorithmus von Johnson. Hierbei bezeichnet $N_U(u)$ die Menge der Nachbarn in dem von U induzierten Untergraphen. Die Funktion `johson-färbung` kann so implementiert werden, daß ihre Laufzeit $O(n^2)$ ist.

Abbildung 9.7 zeigt die Belegung der Variablen U, W und u für den in Abbildung 9.1 dargestellten Graphen mit $n = 3$ und die 2-Färbung, die erzeugt wird. Der Algorithmus

²Für eine reelle Zahl x bezeichnet $\lceil x \rceil$ die kleinste natürliche Zahl größer oder gleich x .

```

var f : array[1..max] of Integer;
function johnson-färbung(G : Graph) : Integer;
var
    U, W set of Integer;
    farbe : Integer;
begin
    Initialisiere f mit 0 und farbe mit 0;
    W := Menge der Ecken von G;
repeat
    U:= W;
    farbe := farbe + 1;
    while U ≠ ∅ do begin
        Sei u eine Ecke mit minimalem Eckengrad in dem
        von U induzierten Untergraph;
        f[u] := farbe;
        U := U-{u}- N_U(u);
        W := W-{u};
    end;
until W = ∅;
johnson-färbung := farbe;
end

```

Abbildung 9.6: Die Funktion johnson-färbung

von Johnson ist für diese Graphen besser als der Greedy-Algorithmus, denn es wird immer eine 2-Färbung erzeugt.

Um den Wirkungsgrad des Algorithmus von Johnson zu bestimmen, wird wieder eine Folge von Graphen konstruiert, für die der Algorithmus schlecht arbeitet. Die rekursiv definierten Graphen B_n sind bipartit und haben 2^{n+1} Ecken. Es sei B_1 der in Abbildung 9.8 dargestellte ungerichtete Graph. Mit Hilfe von B_1 kann eine Folge von Graphen B_n , $n \in \mathbb{N}$, konstruiert werden. Dabei geht B_{n+1} aus B_n hervor, indem an jede Ecke von B_n eine zusätzliche Ecke angehängt wird. Diese neuen Ecken numeriert man von 1 bis 2^{n+1} , die restlichen Ecken werden in der gleichen Reihenfolge wie in B_n mit $2^{n+1}+1, \dots, 2^{n+2}$ numeriert. Der Graph B_3 ist in Abbildung 9.8 dargestellt.

Wendet man den Algorithmus von Johnson auf den Graphen B_n an, so bekommen die Ecken $1, 2, \dots, 2^n$ die Farbe 1, die Ecken $2^n+1, \dots, 2^n+2^{n-1}$ die Farbe 2 etc. Es werden somit genau $n+2$ Farben benötigt. Hieraus folgt, daß der Wirkungsgrad $\mathcal{W}_A(n)$ des Algorithmus von Johnson mindestens $(1 + \log_2 n)/2$ ist. Also ist wiederum $\mathcal{W}_A = \infty$. Im folgenden Satz wird noch eine obere Abschätzung für $\mathcal{W}_A(n)$ bewiesen.

Satz. Für einen ungerichteten Graphen G erzeugt der Algorithmus von Johnson eine Färbung, die maximal $\lceil 4n \log(\chi(G))/\log n \rceil$ Farben verwendet.

$$\begin{array}{lll}
 W = \{1, 2, 3, 4, 5, 6\} & & \\
 U = \{1, 2, 3, 4, 5, 6\} & & \\
 u = 1 \quad U = \{3, 4, 5\} & W = \{2, 3, 4, 5, 6\} & \\
 u = 4 \quad U = \{5\} & & W = \{2, 3, 5, 6\} \\
 u = 5 \quad U = \emptyset & & W = \{2, 3, 6\}
 \end{array}$$

$$\begin{array}{lll}
 W = \{2, 3, 6\} & & \\
 U = \{2, 3, 6\} & & \\
 u = 2 \quad U = \{3, 6\} & W = \{3, 6\} & \\
 u = 3 \quad U = \{6\} & & W = \{6\} \\
 u = 6 \quad U = \emptyset & & W = \emptyset
 \end{array}$$

Ecke	1	2	3	4	5	6	
Farbe	1	2	2	1	1	2	

Abbildung 9.7: Die einzelnen Schritte des Algorithmus von Johnson

Beweis. Es sei $\chi(G) = c$. Es wird ein Durchlauf der äußeren Schleife betrachtet. Der von U induzierte Untergraph G_U habe n_u Ecken. In Kapitel 5 wurde bewiesen, daß G_U eine unabhängige Menge M mit mindestens $\lceil n_u/c \rceil$ Ecken besitzt. Es sei $e \in M$. Dann ist e zu keiner anderen Ecke aus M benachbart. Somit gilt für den Eckengrad einer Ecke u mit dem kleinsten Eckengrad in G_U :

$$g(u) \leq g(e) \leq n_u - \lceil n_u/c \rceil.$$

Das heißt nach dem ersten Durchlauf der inneren Schleife sind in U noch mindestens $\lceil n_u/c - 1 \rceil$ Ecken. Analog zeigt man, daß nach dem zweiten Durchlauf der inneren Schleife in U noch mindestens $\lceil n_u/c^2 - 1/c - 1 \rceil$ Ecken sind. Nach dem l -ten Durchgang sind somit noch

$$\left[\frac{n_u}{c^l} - \sum_{i=0}^{l-1} \frac{1}{c^i} \right]$$

Ecken in U . Für das folgende beachte man, daß $c \geq 2$ und folgende Ungleichung gilt

$$\sum_{i=0}^{l-1} \frac{1}{c^i} = \frac{c}{c-1} \left(1 - \frac{1}{c^l} \right) < 2.$$

Zunächst wird gezeigt, daß die innere Schleife mindestens $\lfloor \log_c n_u \rfloor$ mal durchgeführt wird. Dazu sei $l \leq \log_c n_u$. Dann gilt $n_u \geq c^l$ bzw.

$$0 \leq \frac{n_u}{c^{l-1}} - 2 < \frac{n_u}{c^{l-1}} - \sum_{i=0}^{l-2} \frac{1}{c^i}.$$

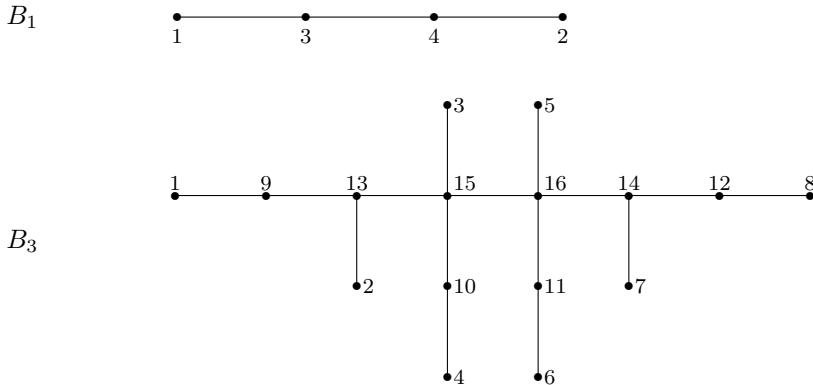


Abbildung 9.8: Die Graphen B_1 und B_3

Somit ist U nach dem $(l - 1)$ -ten Durchgang noch nicht leer. Hieraus folgt die Behauptung. Ferner folgt auch, daß in jedem Durchgang der äußeren Schleife mindestens $\lfloor \log_c |W| \rfloor$ Ecken mit der gleichen Farbe gefärbt werden. Für $|W| \geq 2n / \log_c n$ gilt:

$$\log_c |W| \geq \log_c (2n / \log_c n) > \log_c (\sqrt{n}) = 1/2 \log_c n,$$

da $2\sqrt{n} > \log_c n$ für $c \geq 2$ und alle n gilt. Somit folgt

$$\lfloor \log_c |W| \rfloor > 1/2 \log_c n - 1.$$

Es sei A die Anzahl der Durchgänge, bei denen zu Beginn

$$|W| \geq 2n / \log_c n$$

gilt. Zu Beginn des A -ten Durchgangs ist folgende Ungleichung erfüllt:

$$\frac{2n}{\log_c n} \leq |W| < n - (A - 1) \left(\frac{\log_c n}{2} - 1 \right).$$

Hieraus folgt $A < 2n / \log_c n + 1$ bzw. $A \leq \lceil 2n / \log_c n \rceil$. Nach dem A -ten Durchgang gilt $|W| < 2n / \log_c n$, d.h. für die restlichen Ecken werden noch höchstens $\lfloor 2n / \log_c n \rfloor$ weitere Farben benötigt. Da $\log_c n = \log n / \log \chi(G)$, ist der Beweis damit vollständig. ■

Für die beiden dargestellten Algorithmen gilt $\mathcal{W}_A^\infty = \infty$. Es stellt sich die Frage, ob es überhaupt einen approximativen Algorithmus für die Bestimmung der chromatischen Zahl gibt, dessen asymptotischer Wirkungsgrad durch eine Konstante beschränkt ist. Im folgenden wird gezeigt, daß unter der Voraussetzung $\mathcal{P} \neq \mathcal{NP}$ der asymptotische Wirkungsgrad mindestens $4/3$ sein muß.

Satz. Gibt es einen approximativen Algorithmus A für die Bestimmung der chromatischen Zahl mit asymptotischem Wirkungsgrad $\mathcal{W}_A^\infty < 4/3$, so ist $\mathcal{P} = \mathcal{NP}$.

Beweis. Es wird ein polynomialer Algorithmus A' konstruiert, der entscheidet, ob ein ungerichteter Graph eine 3-Färbung besitzt. Da man gezeigt hat, daß dieses Problem in \mathcal{NPC} liegt, folgt sofort $\mathcal{P} = \mathcal{NP}$. Da $\mathcal{W}_A^\infty < 4/3$ ist, gibt es eine natürliche Zahl C , so daß $A(G) < (4/3)OPT(G)$ für alle ungerichteten Graphen G mit $\chi(G) \geq C$ gilt. Wichtig ist an dieser Stelle, daß $\mathcal{W}_A^\infty < 4/3$ ist, und nicht nur $\mathcal{W}_A^\infty \leq 4/3$ gilt. Es sei nun G ein beliebiger ungerichteter Graph. Der neue Algorithmus A' konstruiert zunächst wieder einen neuen Graphen G' . Dieser besteht aus C Kopien von G und zusätzlichen Kanten zwischen je zwei Ecken aus verschiedenen Kopien; d.h. G' hat Cn Ecken und $C(n^2(C-1)/2 + m)$ Kanten. Die Konstruktion von G' erfolgt in polynomialer Zeit. Es folgt sofort, daß $\chi(G') = \chi(G)C \geq C$ ist. Nun wird der Algorithmus A auf G' angewendet und eine Färbung für G' gebildet. Diese verwendet $A(G')$ Farben, und es gilt $A(G') < (4/3)OPT(G')$. Besitzt G eine 3-Färbung, so gilt

$$A(G') < (4/3)OPT(G') = (4/3)\chi(G)C \leq (4/3)3C = 4C.$$

Besitzt G keine 3-Färbung, so gilt

$$A(G') \geq OPT(G') \geq 4C.$$

Somit besitzt G genau dann eine 3-Färbung, wenn $A(G') < 4C$; d.h. der Algorithmus A' entscheidet, ob ein ungerichteter Graph eine 3-Färbung besitzt. A' ist ein polynomialer Algorithmus, da die Anwendung von A auf G' in polynomialer Zeit erfolgt (C ist eine von G unabhängige Konstante). ■

Bis heute ist kein Algorithmus mit polynomialem Aufwand zum Färben von Graphen bekannt, dessen asymptotischer Wirkungsgrad \mathcal{W}_A^∞ durch eine Konstante beschränkt ist. C. Lund und M. Yannakakis haben unter der Voraussetzung $\mathcal{P} \neq \mathcal{NP}$ bewiesen, daß es eine Konstante $\epsilon > 0$ gibt, so daß der Wirkungsgrad $\mathcal{W}_A(n)$ jedes approximativen Algorithmus A für das Färbungsproblem größer als n^ϵ ist. Das heißt, aus $\mathcal{P} \neq \mathcal{NP}$ folgt $\mathcal{W}_{MIN} = \infty$ für das Färbungsproblem.

Viele Minimierungsprobleme können auch als Maximierungsprobleme formuliert werden. Das Beispiel der Eckenüberdeckung bzw. der unabhängigen Menge zeigt, daß sich die bestmöglichen Wirkungsgrade der zugehörigen approximativen Algorithmen extrem unterscheiden können. Im ersten Fall gibt es einen approximativen Algorithmus mit Wirkungsgrad 2, und im zweiten Fall ist $\mathcal{W}_{MIN} = \infty$.

Das Färbungsproblem für Graphen kann auch als Maximierungsproblem formuliert werden. In diesem Fall zeigt sich ein sehr ähnliches Verhalten der zugehörigen approximativen Algorithmen. Als Maximierungsproblem formuliert, gibt es einen approximativen Algorithmus mit konstantem asymptotischen Wirkungsgrad.

Zum Färben eines Graphen G mit n Ecken stehen n Farben zur Verfügung. Eine Färbung ist optimal, wenn möglichst viele der n Farben nicht verwendet werden. Unter allen Färbungen des Graphen ist diejenige gesucht, für die $n - c$ maximal ist. Hierbei ist c die Anzahl der verwendeten Farben. Für dieses Maximierungsproblem ist $OPT(G) = n - \chi(G)$. Ist A ein Färbungsalgorithmus, der $\chi_A(G)$ Farben vergibt, so ist $A(G) = n - \chi_A(G)$. Um den Wirkungsgrad zu bestimmen, muß eine obere Schranke für

$$\frac{OPT(G)}{A(G)} = \frac{n - \chi(G)}{n - \chi_A(G)}$$

bestimmt werden.

Im folgenden wird zunächst ein einfacher approximativer Algorithmus für dieses Maximierungsproblem mit Wirkungsgrad 2 vorgestellt. Es sei G ein ungerichteter Graph und Z eine nicht erweiterbare Zuordnung des Komplements \bar{G} von G , bestehend aus z Kanten. Mittels Z kann leicht eine Färbung von G mit $n - z$ Farben angegeben werden. Die beiden Enden jeder Kante aus Z bilden in G eine unabhängige Menge und können somit die gleiche Farbe bekommen. Die $2z$ Ecken, die zu Kanten aus Z inzident sind, können somit mit z Farben gefärbt werden. Die restlichen Ecken werden noch mit jeweils einer anderen Farbe gefärbt. Die so erzielte Färbung verwendet $n - z$ Farben. Je mehr Kanten Z enthält, desto weniger Farben werden vergeben.

Am einfachsten gelangt man zu einer nicht erweiterbaren Zuordnung mit Hilfe des Greedy-Verfahrens. Dabei werden die Kanten von \bar{G} in einer festen Reihenfolge betrachtet, und so wird schrittweise eine Zuordnung Z erzeugt. Mit Aufwand $O(n + m)$ kann auf diese Art eine Zuordnung bestimmt werden, die zwar nicht maximal ist, sich aber auch nicht erweitern lässt (vergleichen Sie hierzu Aufgabe 11). Es sei X die Menge der Ecken von G , die zu keiner Kante aus Z inzident sind. Dann ist der von X induzierte Untergraph G_X von G vollständig und enthält $n - 2z$ Ecken. Somit ist $n - 2z \leq \chi(G)$. Für diesen Algorithmus A_1 gilt:

$$\frac{OPT(G)}{A_1(G)} = \frac{n - \chi(G)}{n - (n - z)} = \frac{n - \chi(G)}{z} \leq \frac{n - (n - 2z)}{z} = 2$$

Somit ist der Wirkungsgrad von A_1 kleiner oder gleich 2.

Abbildung 9.9 zeigt einen Graphen G mit $\chi(G) = 3$. Verwendet der Algorithmus A_1 die nicht erweiterbare Zuordnung $Z = \{(2, 5)\}$ von \bar{G} , so ergibt sich eine Färbung, die vier Farben verwendet. Somit ist $OPT(G)/A_1(G) = 2$.



Abbildung 9.9: Eine Anwendung von Algorithmus A_1

Der Algorithmus A_1 kann noch verbessert werden. Ausgangspunkt von A_1 ist eine Zuordnung Z von \bar{G} . Z ist eine Menge von disjunkten, zweielementigen, unabhängigen Mengen von G . Der Algorithmus A_2 verwendet stattdessen dreielementige unabhängige Mengen. Es sei Z eine nicht erweiterbare Menge von disjunkten, dreielementigen, unabhängigen Mengen von G . Z kann wieder mit Hilfe eines Greedy-Verfahrens mit Aufwand $O(n^3)$ bestimmt werden. Bezeichnet man wieder mit X die restlichen Ecken, so hat der von X induzierte Untergraph G_X von G eine interessante Eigenschaft. Da eine unabhängige Menge von G_X aus maximal zwei Ecken besteht, stehen die maximalen Zuordnungen von \bar{G}_X in eindeutiger Beziehung zu den minimalen Färbungen von G_X .

Die Paare von Ecken, die bei einer minimalen Färbung die gleiche Farbe bekommen, induzieren eine maximale Zuordnung von \overline{G}_X und umgekehrt. Die Bestimmung einer minimalen Färbung von G_X kann also auf die Bestimmung einer maximalen Zuordnung von \overline{G}_X reduziert werden. Wie schon in Kapitel 7 erwähnt wurde, kann eine maximale Zuordnung mit Aufwand $O(\sqrt{nm})$ bestimmt werden (hierbei ist m die Anzahl der Kanten in \overline{G}_X). Auf die Darstellung eines geeigneten Algorithmus wird an dieser Stelle verzichtet. Ein alternatives Verfahren mit einer besseren Laufzeit wird in Aufgabe 15 beschrieben.

Der Algorithmus A_2 vergibt nun für jede unabhängige Menge aus Z eine Farbe und färbt die restlichen Ecken mit $\chi(G_X)$ Farben. Ist z die Anzahl der unabhängigen Mengen in Z , so ist $A_2(G) = n - z - \chi(G_X)$. Abbildung 9.10 zeigt einen Graphen G mit $\chi(G) = 3$. Man überzeugt sich leicht, daß $Z = \{\{1, 4, 7\}, \{2, 3, 6\}\}$ aus unabhängigen disjunkten Mengen besteht und nicht erweiterbar ist. Der Graph G_X besteht aus den drei isolierten Ecken 5, 8 und 9. Somit ist $\chi(G_X) = 3$, und A_2 vergibt fünf Farben. Daraus folgt $OPT(G)/A_2(G) = 3/2$.

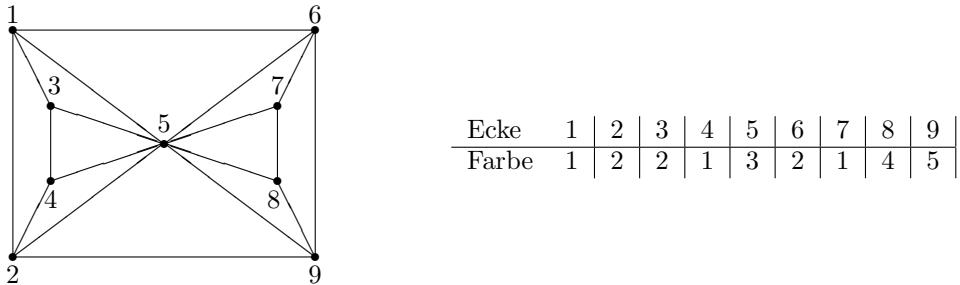


Abbildung 9.10: Eine Anwendung von Algorithmus A_2

Was ist der asymptotische Wirkungsgrad von A_2 ? Die Anzahl der Ecken von G_X wird mit x bezeichnet. Dann gilt:

$$\begin{aligned} A_2(G) &= n - \chi(G_X) - \frac{(n-x)}{3} = \frac{2}{3}n + \frac{x}{3} - \chi(G_X) \\ &\geq \frac{2}{3}n - \frac{2}{3}\chi(G_X) \geq \frac{2}{3}(n - \chi(G)) = \frac{2}{3}OPT(G) \end{aligned}$$

Der asymptotische Wirkungsgrad von A_2 ist somit durch $3/2$ beschränkt. Der Algorithmus A_2 stammt von R. Hassin und A. Lahar und lässt sich noch leicht verbessern. Anstatt jede dreielementige unabhängige Menge von G mit einer neuen Farbe zu färben, zieht man die drei Ecken zu einer zusammen. Die Nachbarn dieser Ecke sind die Nachbarn der drei zusammengezogenen Ecken. Die drei Ecken bekommen später die gleiche Farbe wie die neue Ecke. Dadurch erreicht man, daß auch unabhängige Mengen mit mehr als drei Ecken mit einer Farbe gefärbt werden können. Abbildung 9.11 zeigt die Funktion **3clique-färbung**, welche diese Erweiterung von A_2 realisiert. Die neue Ecke bekommt dabei immer die Nummer der größten der drei Eckennummern.

```

var f : array[1..max] of Integer;
function 3clique-färbung(G : Graph) : Integer;
var
  C : array[1..max] of Integer;
  i, j, s, farbe : Integer;
begin
  Initialisiere f,C und farbe mit 0;
  while es gibt in G eine unabhängige Menge {i,j,s} do begin
    Es sei s ≥ i und s ≥ j;
    for jeden Nachbar l von i oder j do
      Füge die Kante (s,l) in G ein;
    Entferne i und j aus G;
    C[i] := C[j] := s;
  end;
  Es sei M eine maximale Zuordnung des Komplementes von G;
  for jede Kante (i,j) ∈ M do
    farbe := farbe + 1;
    f[i] := f[j] := farbe;
  end;
  for jede Ecke i von G do
    if f[i] = 0 then begin
      farbe := farbe + 1;
      f[i] := farbe;
    end;
  for i := n to 1 do
    if f[i] = 0 then
      f[i] := f[C[i]];
  3clique-färbung := farbe;
end

```

Abbildung 9.11: Die Funktion 3clique-färbung

Wendet man die Funktion 3clique-färbung auf den Graphen aus Abbildung 9.10 an, so wird zunächst die unabhängige Menge $\{1, 4, 7\}$ und dann $\{2, 3, 6\}$ zu jeweils einer Ecke zusammengezogen. Abbildung 9.12 zeigt die dadurch entstehenden Graphen.

Eine maximale Zuordnung von $\overline{G_X}$ besteht aus den beiden Kanten $(7, 9)$ und $(6, 8)$. Die Funktion 3clique-färbung vergibt somit folgende Farben:

Ecke	1	2	3	4	5	6	7	8	9
Farbe	1	2	2	1	3	2	1	2	1

Die verbesserte Version von A_2 bestimmt also eine minimale Färbung von G . Der asymptotische Wirkungsgrad bleibt aber unverändert. Dazu betrachte man den in Abbildung 9.13 dargestellten Graphen G mit $\chi(G) = 2$. Wählt die Funktion 3clique-färbung die

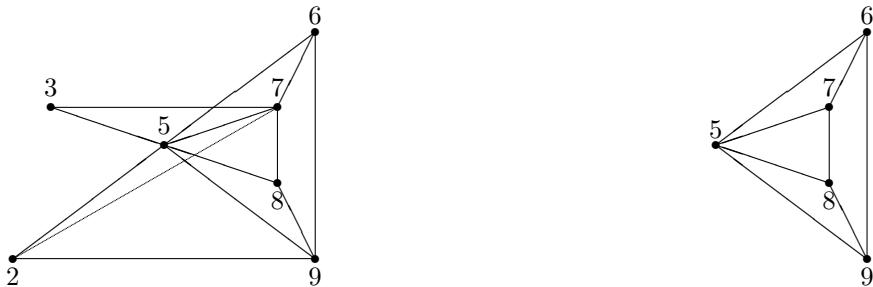


Abbildung 9.12: Eine Anwendung von 3clique-färbung auf den Graphen aus Abbildung 9.10

unabhängige Menge $\{1, 2, 3\}$, so entsteht ein vollständiger Graph mit drei Ecken; d.h. 3clique-färbung vergibt drei Farben, und es gilt

$$OPT(G)/A_2(G) = 3/2.$$

Mit einer Verallgemeinerung des letzten Beispiels kann leicht gezeigt werden, daß der asymptotische Wirkungsgrad von A_2 gleich $3/2$ ist.

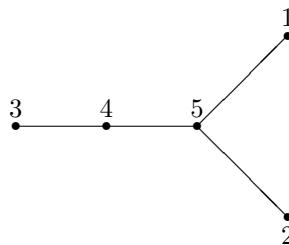


Abbildung 9.13: Ein ungerichteter Graph G mit $OPT(G)/A_2(G) = 3/2$

Ausgangspunkt von A_2 ist eine nicht erweiterbare Menge Z von disjunkten, dreielementigen, unabhängigen Mengen von G . Eine naheliegende Erweiterung wäre die Betrachtung von vierelementigen, unabhängigen Mengen von G . Bezeichnet man wieder mit X die restlichen Ecken, so gilt $\alpha(G_X) \leq 3$ für den von X induzierten Untergraphen G_X . Die Bestimmung einer minimalen Färbung von G_X lässt sich in diesem Fall leider nicht mehr so leicht durchführen. Das zugehörige Entscheidungsproblem ist \mathcal{NP} -vollständig (vergleichen Sie Aufgabe 30).

Mit Hilfe von anderen Techniken hat M. M. Halldórsson den Algorithmus A_2 noch verbessert und einen Wirkungsgrad von $4/3$ erzielt.

9.6 Das Problem des Handlungsreisenden

Ein Handlungsreisender soll n Städte nacheinander, aber jede Stadt nur einmal besuchen. Die Fahrstrecke soll dabei möglichst gering sein. Am Ende der Reise soll er wieder in die Ausgangsstadt zurückkehren. Dieses sogenannte *Handlungsreisendenproblem (Traveling-Salesman Problem)* ist ein graphentheoretisches Optimierungsproblem: Gegeben ist ein ungerichteter, kantenbewerteter, zusammenhängender Graph. Unter allen geschlossenen einfachen Wegen, auf denen alle Ecken des Graphen liegen, ist ein solcher mit minimaler Länge gesucht. Im folgenden wird immer vorausgesetzt, daß die Kantenbewertungen positiv sind. Für den in Abbildung 9.14 links dargestellten Graphen ist der geschlossene Weg 1,2,4,5,3,1 eine Lösung der Länge 15.

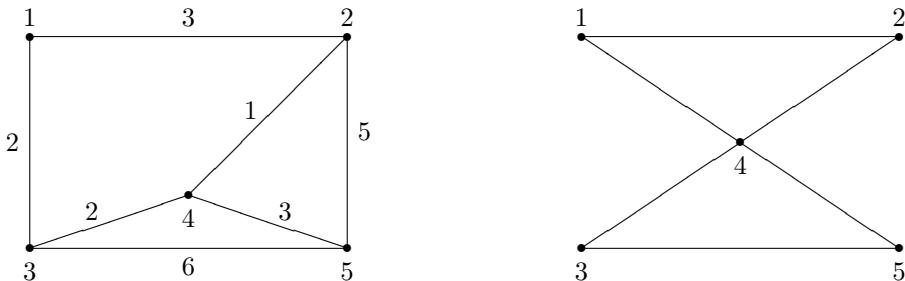


Abbildung 9.14: Beispiele für das Problem des Handlungsreisenden

Das Problem des Handlungsreisenden ist nicht für jeden Graphen lösbar, wie der rechte Graph aus Abbildung 9.14 zeigt. In diesem Fall gibt es keinen geschlossenen einfachen Weg, der alle Ecken enthält. Einen solchen Weg nennt man *Hamiltonschen Kreis*. Ein ungerichteter Graph heißt *Hamiltonscher Graph*, wenn er einen Hamiltonschen Kreis besitzt. In Abschnitt 9.1 wurde bereits angegeben, daß das Entscheidungsproblem für Hamiltonsche Kreise in $\mathcal{NP}\text{-C}$ liegt. Somit liegt auch das Entscheidungsproblem des Handlungsreisenden in $\mathcal{NP}\text{-C}$.

Eine Variante des Handlungsreisendenproblems besteht darin, einen geschlossenen Kantenzug minimaler Länge zu finden, auf dem jede Ecke mindestens einmal liegt. Für den linken Graphen aus Abbildung 9.14 ist der geschlossene Kantenzug 1,2,4,5,4,3,1 eine Lösung der Länge 14. In dieser Form hat das Problem des Handlungsreisenden immer eine Lösung. Diese Variante läßt sich leicht auf das Ausgangsproblem zurückführen. Zu einem gegebenen Graphen G mit n Ecken wird ein vollständiger Graph G' mit der gleichen Eckenanzahl konstruiert. Die Kante von i nach j in G' trägt als Bewertung die Länge des kürzesten Weges von i nach j in G . Jeder Kante von G' entspricht somit ein kürzester Weg in G . Abbildung 9.15 zeigt den Graphen G' für den linken Graphen aus Abbildung 9.14.

Jedem geschlossenen einfachen Weg W' in G' kann eindeutig ein geschlossener Kantenzug W in G zugeordnet werden. Jede Kante von W' wird dabei durch den entsprechenden kürzesten Weg zwischen den beiden Ecken in G ersetzt. Enthält W' alle Ecken von G' , so enthält auch W alle Ecken von G . Ferner haben W' und W die gleiche Länge. Dem geschlossenen einfachen Weg 1,2,4,5,3,1 des Graphen G' aus Abbildung 9.15 ent-

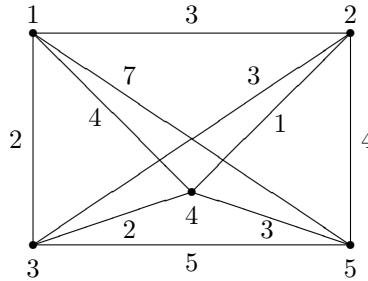


Abbildung 9.15: Der Graph G' für den linken Graphen aus Abbildung 9.14

spricht der geschlossene Kantenzug 1,2,4,5,4,3,1 in G . Für die so eingeführten Größen W' und W gilt nun folgendes Lemma:

Lemma. Es sei W' ein geschlossener einfacher Weg minimaler Länge von G' , auf dem alle Ecken liegen. Dann ist W ein geschlossener Kantenzug minimaler Länge in G , auf dem alle Ecken liegen.

Beweis. Es sei \bar{W} ein geschlossener Kantenzug minimaler Länge von G , auf dem alle Ecken liegen. Bezeichne mit \bar{W}' den geschlossenen Kantenzug in G' , welcher die Ecken in der gleichen Reihenfolge wie \bar{W} besucht. Nach Konstruktion von G' gilt $L(\bar{W}') \leq L(\bar{W})$. Mit Hilfe von \bar{W}' wird ein geschlossener einfacher Weg \hat{W} in G' konstruiert, welcher alle Ecken enthält. Jede Ecke von G' liegt mindestens einmal auf \bar{W}' . Es sei nun e eine Ecke, die mehr als einmal auf \bar{W}' vorkommt. Ersetze nun die Kanten (e_1, e) und (e, e_2) auf \bar{W}' durch die Kante (e_1, e_2) . Dann liegen immer noch alle Ecken auf \bar{W}' . Aus der Konstruktion von G' ergibt sich, daß die Länge von \bar{W}' hierdurch nicht vergrößert wurde. Auf diese Art entsteht ein geschlossener einfacher Weg \hat{W} in G' , welcher alle Ecken enthält. Nun gilt:

$$L(\bar{W}) \geq L(\bar{W}') \geq L(\hat{W}) \geq L(W') = L(W).$$

Somit ist W ein geschlossener Kantenzug minimaler Länge in G . ■

Aus diesem Lemma folgt, daß diese Variante nur ein Spezialfall des allgemeinen Problems des Handlungsreisenden ist. Dieses Ergebnis und die Tatsache, daß das Problem in der allgemeinen Form das Problem des Hamiltonschen Kreises enthält, führt zu folgender Definition des Problems des Handlungsreisenden: Gegeben sei ein vollständiger ungerichteter Graph mit positiven Kantenbewertungen. Gesucht ist ein geschlossener einfacher Weg minimaler Länge, der jede Ecke enthält. Dieses Problem wird im folgenden mit *TSP* bezeichnet. Da nur vollständige Graphen betrachtet werden, gibt es immer eine Lösung. Da es nur endlich viele geschlossene, einfache Wege gibt, die alle Ecken enthalten, kann das TSP in endlicher Zeit gelöst werden. Die Anzahl dieser Wege wächst allerdings mit der Anzahl n der Ecken stark an. Für n Ecken gibt es $(n - 1)!/2$ verschiedene Hamiltonsche Kreise in K_n . Dadurch wird ein vollständiges Durchprobieren

aller Möglichkeiten für größere n unmöglich. Bis heute ist kein effizienter Algorithmus für dieses Problem bekannt. Es gilt folgender Satz:

Satz. Das zu TSP gehörende Entscheidungsproblem ist \mathcal{NP} -vollständig.

Beweis. Das Entscheidungsproblem liegt offenbar in \mathcal{NP} . Im folgenden wird nun eine polynomiale Transformation des \mathcal{NP} -vollständigen Problems Hamiltonscher Kreise auf TSP beschrieben. Aus dem Satz aus Abschnitt 9.1 folgt dann, daß auch TSP \mathcal{NP} -vollständig ist. Es sei G ein ungerichteter Graph mit n Ecken. Sei nun G' ein vollständiger kantenbewerteter Graph mit n Ecken. Eine Kante von G' trägt die Bewertung 1, falls es diese Kante auch in G gibt, und andernfalls die Bewertung 2. Der Graph G' läßt sich in polynomialer Zeit konstruieren. Eine Lösung des TSP für G' hat mindestens die Länge n . Es gilt sogar, daß eine Lösung genau dann die Länge n hat, wenn der Graph G einen Hamiltonschen Kreis besitzt. Somit folgt $\text{TSP} \in \mathcal{NPC}$. ■

Alle bis heute bekannten Algorithmen zur Lösung des TSP laufen im Grunde genommen auf eine vollständige Analyse aller Möglichkeiten hinaus. Das in Abschnitt 5.3 beschriebene Backtracking-Verfahren läßt sich auch auf TSP anwenden. Eine Variante dieses Verfahrens ist *Branch-and-bound*. Das Backtracking-Verfahren wird dabei effizienter, indem das Durchsuchen ganzer Teilbäume überflüssig gemacht wird.

Eine weitere Möglichkeit zur Lösung des TSP bietet die Technik des *dynamischen Programmierens*. Dabei wird ein Problem in kleinere Unterprobleme zerlegt und dann wird die Lösung rekursiv bestimmt. Zur Bestimmung der Unterprobleme für das TSP wird noch folgende Definition benötigt: Die Ecken von G werden mit $1, \dots, n$ bezeichnet. Für jede Teilmenge M von $\{2, \dots, n\}$ und $e \in \{2, \dots, n\}$ bezeichne mit $L(M, e)$ die Länge eines kürzesten einfachen Weges von 1 nach e , der jede Ecke aus M verwendet. Dann gilt:

$$OPT(G) = \min \{ L(\{2, \dots, n\}, e) + b_{e1} \mid e \in \{2, \dots, n\} \}$$

Die Größen $L(\{2, \dots, n\}, e)$ werden nun rekursiv bestimmt. Hierbei macht man sich das in Kapitel 8 diskutierte Optimalitätsprinzip zunutze. Für jede Teilmenge M von $\{2, \dots, n\}$ mit mindestens zwei Elementen und jede Ecke e gilt

$$L(M, e) = \min \{ L(M \setminus \{e\}, f) + b_{fe} \mid f \in M, e \neq f \}.$$

Ferner gilt

$$L(\{e\}, e) = b_{e1}.$$

Die Bestimmung der $L(M, e)$ erfolgt rekursiv für alle Teilmengen M von $\{2, \dots, n\}$ mit mindestens zwei Elementen und alle Ecken e . Dies sind insgesamt

$$\sum_{i=2}^{n-1} \binom{n-1}{i} i$$

verschiedene Berechnungen. Für eine Menge mit i Elementen sind dabei $i - 1$ Additionen und Vergleiche notwendig. Um $OPT(G)$ zu bestimmen, sind noch einmal $n - 1$

Operationen erforderlich. Dies gibt insgesamt folgenden Gesamtaufwand:

$$\begin{aligned}
 n - 1 + \sum_{i=2}^{n-1} \binom{n-1}{i} i(i-1) &= \\
 n - 1 + (n-1)(n-2) \sum_{i=2}^{n-1} \binom{n-3}{i-1} &= \\
 n - 1 + (n-1)(n-2) \sum_{i=0}^{n-3} \binom{n-3}{i} &= \\
 n - 1 + (n-1)(n-2)2^{n-3} &= O(n^2 2^n).
 \end{aligned}$$

Die Laufzeit dieses Algorithmus zeigt zwar immer noch ein exponentielles Wachstum, aber im Vergleich zu dem Durchprobieren aller $(n-1)!$ Hamiltonscher Kreise ist dies schon eine wesentliche Verbesserung. Es sollte aber angemerkt werden, daß auch der Speicherbedarf exponentiell anwächst.

Der letzte Satz legt es nahe, approximative Algorithmen für TSP zu suchen. Der folgende Satz zeigt, daß es einen approximativen Algorithmus mit beschränktem asymptotischen Wirkungsgrad für TSP nur dann gibt, wenn $\mathcal{P} = \mathcal{NP}$ ist.

Satz. Gibt es einen approximativen Algorithmus A für TSP mit asymptotischen Wirkungsgrad $W_A^\infty < \infty$, so ist $\mathcal{P} = \mathcal{NP}$.

Beweis. Es wird gezeigt, daß der Algorithmus A dazu verwendet werden kann, das Entscheidungsproblem des Hamiltonschen Kreises zu lösen. Da dieses Problem in \mathcal{NP} liegt, folgt aus dem Satz aus Abschnitt 9.1, daß $\mathcal{P} = \mathcal{NP}$ ist. Da $W_A^\infty < \infty$ ist, gibt es eine Konstante $C \in \mathbb{N}$, so daß $W_A(n) < C$ für alle $n \in \mathbb{N}$ gilt. Es sei nun G ein ungerichteter Graph mit n Ecken. Ferner sei G' ein vollständiger kantenbewerteter Graph mit n Ecken. Die Kante (i, j) von G' trägt die Bewertung 1, falls i und j in G benachbart sind, und andernfalls die Bewertung nC . Der Graph G' kann in polynomialer Zeit konstruiert werden. Besitzt G einen Hamiltonschen Kreis, so ist $OPT(G') = n$. Andernfalls muß eine Lösung des TSP für G' mindestens eine Kante enthalten, welche die Bewertung nC trägt. In diesem Fall gilt:

$$A(G') \geq OPT(G') \geq n - 1 + nC > nC$$

Nach Voraussetzung gilt $A(G') < OPT(G')C$. Somit gilt $A(G') \leq nC$ genau dann, wenn G einen Hamiltonschen Kreis enthält. Somit gibt es einen polynomiellen Algorithmus für das Entscheidungsproblem des Hamiltonschen Kreises. ■

Der letzte Satz zeigt, daß es genau so schwer ist, einen effizienten approximativen Algorithmus für TSP zu finden, wie einen effizienten Algorithmus für die exakte Lösung des TSP zu finden. Das TSP tritt in vielen praktischen Problemen in einer speziellen Form auf. Die Bewertungen b_{ij} der Kanten erfüllen in diesen Fällen die sogenannte *Dreiecksungleichung*: Für jedes Tripel i, j, e von Ecken gilt:

$$b_{ij} \leq b_{ie} + b_{ej}$$

Diese Voraussetzung ist zum Beispiel erfüllt, wenn die Bewertungen der Kanten den Abständen von n Punkten in der Euklidischen Ebene entsprechen. Auch die Bewertungen des zu Beginn dieses Abschnittes konstruierten Graphen G' erfüllen die Dreiecksungleichung; dies folgt aus den in Kapitel 8 angegebenen Bellmanschen Gleichungen. Diese spezielle Ausprägung des TSP wird im folgenden mit $\Delta\text{-TSP}$ bezeichnet. Die Hoffnung, daß es für diese Variante des TSP einen effizienten Algorithmus gibt, ist verfehlt. Der Beweis, daß $\text{TSP} \in \mathcal{NPC}$ ist, läßt sich leicht auf $\Delta\text{-TSP}$ übertragen. Die Kanten des im Beweis konstruierten Graphen G' tragen die Bewertungen 1 oder 2. Der Graph G' erfüllt also die Dreiecksungleichung. Somit gilt:

Satz. Das zu $\Delta\text{-TSP}$ gehörende Entscheidungsproblem ist \mathcal{NP} -vollständig.

Der große Unterschied zwischen TSP und $\Delta\text{-TSP}$ besteht darin, daß es für $\Delta\text{-TSP}$ approximative Algorithmen mit konstantem asymptotischen Wirkungsgrad gibt. Man beachte, daß sich der Beweis für die Nichtexistenz eines „guten“ approximativen Algorithmus für TSP nicht auf $\Delta\text{-TSP}$ übertragen läßt, da der konstruierte Graph nicht die Dreiecksungleichung erfüllt.

Im folgenden stellen wir nun drei verschiedene approximative Algorithmen für $\Delta\text{-TSP}$ vor. Der erste Algorithmus A_1 (*Nächster-Nachbar Algorithmus*) baut einen einfachen Weg schrittweise auf. Der Weg startet bei einer beliebigen Ecke e_1 . Es sei $W_1 = \{e_1\}$. Der Übergang von $W_l = \{e_1, \dots, e_l\}$ nach W_{l+1} geschieht folgendermaßen: Es sei e_{l+1} die Ecke des Graphen, die noch nicht in W_l ist und zu e_l den geringsten Abstand hat. Dann ist $W_{l+1} = \{e_1, \dots, e_{l+1}\}$. Am Ende des Algorithmus bildet W_n einen geschlossenen einfachen Weg, der alle Ecken enthält. Für den Graphen aus Abbildung 9.15 ergibt sich für die Startecke 1 der Weg 1,3,4,2,5,1 mit Länge 16. Dagegen hat der optimale Weg die Länge 14. Es gilt folgender Satz:

Satz. Für alle $n \in \mathbb{N}$ ist $W_{A_1}(n) \leq (\lceil \log_2 n \rceil + 1)/2$.

Beweis. Es sei G ein kantenbewerteter, ungerichteter, vollständiger Graph mit n Ecken. Die Bewertungen b_{ij} der Kanten von G erfüllen die Dreiecksungleichung. Es sei W ein geschlossener einfacher Weg minimaler Länge, auf dem alle Ecken von G liegen. Die Länge der von A_1 an die Ecke i angehängten Kante sei l_i . Die Ecke 1 sei die Startecke, und die Ecken seien so numeriert, daß $l_1 \geq l_2 \geq \dots \geq l_n$ gilt. Aus der Dreiecksungleichung folgt $L(W) \geq 2l_1$. Für $r \in \mathbb{N}$ sei W_r der Weg, der entsteht, wenn die Ecken $\{1, 2, \dots, 2r\}$ in der gleichen Reihenfolge wie auf W durchlaufen werden. Aus der Dreiecksungleichung folgt $L(W) \geq L(W_r)$.

Es sei nun (i, j) eine Kante von W_r . Falls die Ecke i vor der Ecke j von A_1 ausgewählt wurde, so gilt $b_{ij} \geq l_i$ und andernfalls $b_{ij} \geq l_j$. Somit gilt $b_{ij} \geq \min(l_i, l_j)$ für alle Kanten (i, j) von W_r . Die Summation über alle Kanten ergibt dann

$$L(W) \geq L(W_r) \geq l_2 + l_3 + \dots + l_{2r} + l_{2r} \geq 2 \sum_{i=r+1}^{2r} l_i.$$

Analog zeigt man

$$L(W) \geq 2 \sum_{i=\lceil n/2 \rceil + 1}^n l_i.$$

Daraus ergibt sich

$$\begin{aligned} 2A_1(G) &= 2 \sum_{i=1}^n l_i \\ &\leq 2l_1 + \sum_{i=0}^{\lceil \log_2 n \rceil - 2} L(W_{2^i}) + 2 \sum_{i=\lceil n/2 \rceil + 1}^n l_i \\ &\leq (\lceil \log_2 n \rceil + 1) OPT(G). \end{aligned}$$

■

Es kann sogar gezeigt werden, daß $(\log_2(n+1) + 4/3)/3 < W_{A_1}(n)$ ist. Hieraus folgt dann sofort, daß $W_{A_1}^\infty = \infty$ ist. Der Beweis beruht auf einer Folge von Graphen, für die der Wirkungsgrad von A_1 diese untere Schranke übersteigt. Die rekursive Konstruktion dieser Graphen ist relativ kompliziert und wird hier nicht nachvollzogen. Abbildung 9.16 zeigt einen Graphen G mit sieben Ecken und Wirkungsgrad $11/7 > (3 + 4/3)/3 = 13/9$. Dabei wurden nicht alle Kanten eingezeichnet. Die Länge der Kante (i, j) ist, sofern sie nicht explizit angegeben ist, gleich der Länge des kürzesten Weges von i nach j unter Verwendung der dargestellten Kanten. Die Bewertungen erfüllen die Dreiecksungleichung. Der einfache Weg $1, 2, 3, 4, 5, 6, 7, 1$ hat die Länge 7, somit ist $OPT(G) = 7$. Der Algorithmus A_1 wählt den Weg $1, 3, 2, 5, 7, 6, 4, 1$ mit der Länge 11.

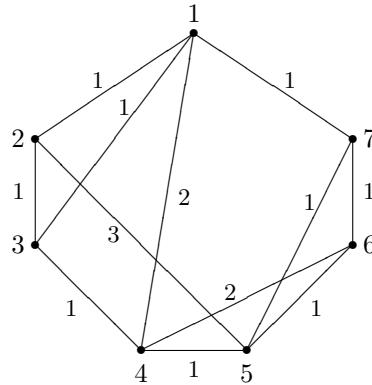


Abbildung 9.16: Eine Anwendung von Algorithmus A_1

Der Algorithmus A_1 demonstriert, daß eine sehr naheliegende Vorgehensweise zu einem approximativen Algorithmus mit unbeschränktem Wirkungsgrad führen kann. Durch

eine kleine Abänderung von A_1 erhält man einen approximativen Algorithmus mit asymptotischem Wirkungsgrad 2 (vergleichen Sie Aufgabe 21). Der Aufwand von A_1 ist $O(n^2)$.

Die nächsten beiden approximativen Algorithmen für Δ -TSP basieren auf folgendem Lemma:

Lemma. Es sei G ein kantenbewerteter, ungerichteter, vollständiger Graph. Die Bewertungen der Kanten erfüllen die Dreiecksungleichung. Ferner sei B ein minimal aufspannender Baum von G mit Kosten K und W ein geschlossener einfacher Weg minimaler Länge, auf dem alle Ecken von G liegen. Dann gilt:

$$K < L(W) \leq 2K$$

Beweis. Entfernt man eine Kante von W , so ergibt sich ein aufspannender Baum von G . Somit ist $K < L(W)$. Mit Hilfe der Tiefensuche, angewendet auf B , kann ein geschlossener Kantenzug W_1 erzeugt werden, der jede Kante von B genau zweimal enthält. Dazu wird sowohl beim Erreichen als auch beim Verlassen einer Ecke die entsprechende Kante in W_1 eingefügt. Die Länge von W_1 ist gleich $2K$. Nun durchläuft man W_1 und überspringt dabei schon besuchte Ecken. Auf diese Art entsteht aus W_1 ein geschlossener einfacher Weg W_2 von G , der alle Ecken enthält. Dies ist möglich, da G vollständig ist. Aus der Dreiecksgleichung folgt, daß $L(W_2) \leq L(W_1) \leq 2K$ ist. Somit ist auch $L(W) \leq 2K$. ■

Aus dem letzten Lemma kann die Vorgehensweise von Algorithmus A_2 direkt abgeleitet werden. Zunächst wird ein minimal aufspannender Baum konstruiert und anschließend mittels der Tiefensuche ein entsprechender Kantenzug erzeugt. Dieser wird dann nochmal durchlaufen, um daraus einen geschlossenen einfachen Weg zu machen. Der Aufwand von A_2 wird im wesentlichen durch den Aufwand bestimmt, den aufspannenden Baum zu konstruieren. Unter Verwendung des Algorithmus von Prim hat der Algorithmus einen Aufwand von $O(n^2)$. Aus dem letzten Lemma folgt:

Satz. Für alle $n \in \mathbb{N}$ ist $W_{A_2}(n) < 2$.

Abbildung 9.17 zeigt eine Anwendung von A_2 auf den Graphen aus Abbildung 9.15. Links ist ein aufspannender Baum mit Kosten 8 und rechts der von A_2 erzeugte geschlossene einfache Weg dargestellt. Dieser hat die Länge 16. Der optimale Weg hat die Länge 14. Mit Hilfe einer Folge von Graphen kann man zeigen, daß $W_{A_2}^\infty = 2$ ist.

Zum Abschluß dieses Abschnitts wird noch der Algorithmus für Δ -TSP mit dem zur Zeit kleinsten asymptotischen Wirkungsgrad vorgestellt. Dieser Algorithmus stammt von N. Christofides und hat einen Wirkungsgrad von $3/2$. Zur Darstellung von A_3 wird die Definition von *Eulerschen Graphen* benötigt.

Ein ungerichteter Graph heißt Eulerscher Graph, wenn er einen geschlossenen Kantenzug besitzt, auf dem jede Kante genau einmal vorkommt. Der Ursprung dieser Definition geht auf L. Euler zurück, der als erster im sogenannten *Königsberger Brückenproblem*

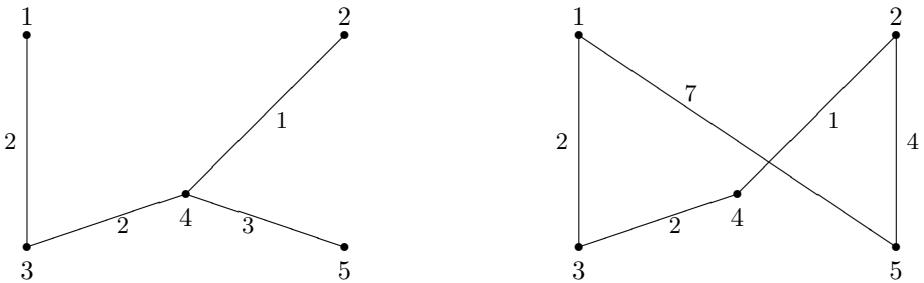


Abbildung 9.17: Eine Anwendung des Algorithmus A_2

die Frage nach der Existenz eines solchen geschlossenen Kantenzuges untersuchte. Abbildung 9.18 zeigt links einen Eulerschen und rechts einen Nichteulerschen Graphen.



Abbildung 9.18: Ein Eulerscher und ein nicht Eulerscher Graph

Obwohl die Konzepte des Eulerschen und des Hamiltonschen Graphen sehr ähnlich sind, liegen die zugehörigen Entscheidungsprobleme in verschiedenen Komplexitätsklassen. Das Problem der Hamiltonschen Kreise liegt in $\mathcal{NP}C$. Hingegen lässt sich leicht ein Algorithmus angeben, welcher mit Aufwand $O(m)$ feststellt, ob es sich um einen Eulerschen Graphen handelt und gegebenenfalls einen entsprechenden Kantenzug bestimmt. Der folgende Satz gibt eine Charakterisierung Eulerscher Graphen an.

Satz. Es sei G ein zusammenhängender ungerichteter Graph. Genau dann ist G ein Eulerscher Graph, wenn der Eckengrad jeder Ecke eine gerade Zahl ist.

Beweis. Es sei zunächst W ein geschlossener Kantenzug von G , welcher jede Kante genau einmal enthält. Es sei e eine beliebige Ecke von G . Jedes Vorkommen von e auf W trägt 2 zum Eckengrad von e bei. Da alle Kanten genau einmal auf W vorkommen, muß $g(e)$ eine gerade Zahl sein.

Es sei nun G ein zusammenhängender ungerichteter Graph, so daß der Eckengrad jeder Ecke gerade ist. Nun starte man einen Kantenzug an einer beliebigen Ecke des Graphen. Man durchlaufe den Graphen und entferne jede verwendete Kante. Bei jedem Durchlaufen einer Ecke sinkt der Eckengrad um 2. Da alle Ecken geraden Eckengrad haben, muß der Kantenzug bei der Startecke enden. Wurden alle Kanten schon ver-

wendet, so handelt es sich um einen Eulerschen Graphen. Andernfalls gibt es eine Ecke auf dem Kantenzug, welche noch zu unbenutzten Kanten inzident ist. Von dieser Ecke wird erneut ein Kantenzug auf die gleiche Art erzeugt. Dieser Kantenzug muß ebenfalls an seiner Startecke enden. Die beiden Kantenzüge werden nun zu einem Kantenzug verschmolzen. Man beachte, daß die Ecken in dem Restgraphen alle geraden Eckengrad haben. Auf diese Art wird ein geschlossener Kantenzug erzeugt. Dieser enthält jede Kante genau einmal. Somit ist G ein Eulerscher Graph. ■

Der zweite Teil des Beweises läßt sich leicht in einen Algorithmus umsetzen. Eine rekursive Implementierung dieses Algorithmus mit Laufzeit $O(m)$ ist in Aufgabe 23 beschrieben. Man beachte, daß das Lemma auch für nicht schlichte Graphen gilt.

Der Algorithmus A_3 für Δ -TSP läßt sich nun leicht angeben. Es sei G ein kantenbewerteter, ungerichteter vollständiger Graph, dessen Kantenbewertungen die Dreiecksungleichungen erfüllen. Der Algorithmus A_3 besteht aus fünf Schritten:

- a) Bestimme einen minimal aufspannenden Baum B von G .
- b) Es sei E' die Menge der Ecken von B , welche ungeraden Eckengrad haben. Es sei G' der von E' induzierte kantenbewertete Untergraph von G .
- c) Bestimme eine vollständige Zuordnung Z von G' mit minimalen Kosten.
- d) Es sei B' der Graph, der aus B entsteht, indem die Kanten aus Z eingefügt werden. Bestimme einen geschlossenen Kantenzug W von B' , der jede Kante von B' genau einmal enthält.
- e) Durchlaufe W und entferne jede dabei schon besuchte Ecke.

Schritt a) wird mit Hilfe des Algorithmus von Prim mit Aufwand $O(n^2)$ durchgeführt. Die Anzahl der Ecken in E' ist gerade. Auf die Darstellung eines Algorithmus für Schritt c) wird an dieser Stelle verzichtet und auf entsprechende Literatur verwiesen. Bei Verwendung eines geeigneten Algorithmus kann Schritt c) mit Aufwand $O(n^3)$ durchgeführt werden. Man beachte, daß der Graph B' aus Schritt d) nicht mehr schlicht sein muß. Der Eckengrad jeder Ecke von B' ist nach Konstruktion gerade. Somit kann W mit Aufwand $O(n^2)$ bestimmt werden. Die Schritte d) und e) können in linearer Zeit durchgeführt werden. Somit ist der Aufwand von A_3 insgesamt $O(n^3)$.

Abbildung 9.19 zeigt eine Anwendung von A_3 auf den Graphen aus Abbildung 9.15. Hierbei ist links ein minimal aufspannender Baum dargestellt. In der Mitte ist der Graph G' abgebildet. Die fett gezeichneten Kanten $(1, 2)$ und $(4, 5)$ bilden eine vollständige Zuordnung von G' mit minimalen Kosten. Der Graph B' ist rechts abgebildet. Insgesamt ergibt sich der geschlossene einfache Weg $1, 3, 4, 5, 2, 1$ der Länge 14. Dies ist auch der kürzeste einfache Weg.

Es bleibt noch, den Wirkungsgrad von A_3 zu bestimmen.

Lemma. Für alle $n \in \mathbb{N}$ ist $W_{A_3}(n) < 3/2$.

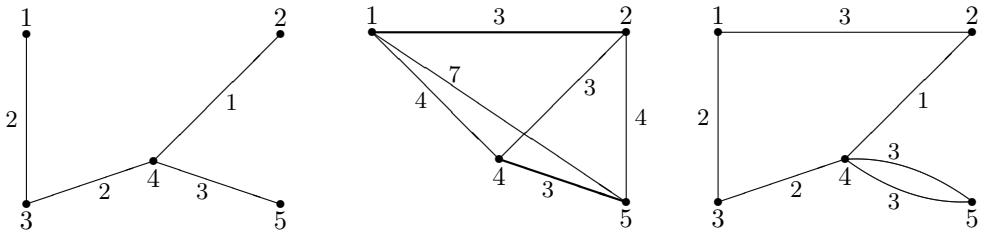


Abbildung 9.19: Eine Anwendung des Algorithmus A_3

Beweis. Es sei \overline{W} ein Kantenzug minimaler Länge von G , so daß jede Ecke von G genau einmal auf \overline{W} liegt. Es genügt nun, zu zeigen, daß die Länge des Kantenzuges W aus Schritt d) echt kleiner als $3L(\overline{W})/2$ ist. Dazu wird gezeigt, daß die Summe der Kantenbewertungen der Kanten aus Z maximal $L(\overline{W})/2$ ist. Aus \overline{W} entferne man alle Ecken, die nicht in E' liegen. Man bezeichne die verbleibenden Kanten in der entsprechenden Reihenfolge mit k_1, \dots, k_{2s} . Dann sind $Z_1 = \{k_1, k_3, \dots, k_{2s-1}\}$ und $Z_2 = \{k_2, k_4, \dots, k_{2s}\}$ vollständige Zuordnungen von G' . Ferner ist die Summe der Kosten der beiden Zuordnungen maximal $L(\overline{W})$. Somit sind die Kosten von Z maximal $L(\overline{W})/2$. Nach dem oben bewiesenen Lemma sind die Kosten von B in Schritt a) echt kleiner als $L(\overline{W})$. Somit gilt $W_{A_3}(n) < 3/2$. ■

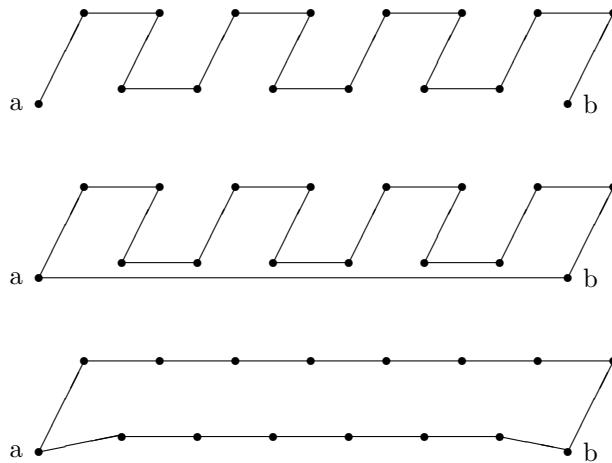


Abbildung 9.20: Ein Graph, für den der Algorithmus A_3 schlecht arbeitet

Mit Hilfe von speziell konstruierten Graphen kann man zeigen, daß $W_{A_3}^\infty = 3/2$ ist. Dazu betrachte man eine Folge von planaren Graphen G_n mit $2n$ Ecken. Die Ecken sind in gleichen Abständen auf zwei parallelen Geraden angeordnet, so daß jeweils drei Ecken ein gleichseitiges Dreieck bilden. Die Endpunkte der unteren Geraden sind leicht

verschoben. Abbildung 9.20 zeigt den Graphen G_8 mit den verschobenen Ecken a, b . Im oberen Teil der Abbildung ist ein minimal aufspannender Baum dargestellt. Die einzigen Ecken mit ungeradem Eckengrad sind die Ecken a und b . Im mittleren Teil ist der geschlossene Kantenzug abbgebildet, den der Algorithmus A_3 erzeugt, und unten ist ein minimaler Hamiltonscher Kantenzug abbgebildet. Es gilt also:

$$OPT(G_n)/A_3(G_n) \approx \frac{3n - 2}{2n}$$

Somit ist der asymptotische Wirkungsgrad von A_3 gleich $3/2$.

Bis heute ist kein approximativer Algorithmus für Δ -TSP bekannt, dessen Wirkungsgrad echt kleiner $3/2$ ist.

9.7 Literatur

Der Begriff approximativer Algorithmus wurde zuerst in einer Arbeit von Johnson eingeführt [73]. Die Definition des Wirkungsgrades eines approximativen Algorithmus stammt von Garey und Johnson [46]. Ihr Buch enthält eine umfangreiche Liste von \mathcal{NP} -vollständigen Problemen aus den verschiedensten Anwendungsgebieten. Dort findet man auch eine ausführliche Diskussion der Resultate über Wirkungsgrade von approximativen Algorithmen. Den Satz von Vizing findet man in [120]. I. Holyer hat bewiesen, daß das Problem der Kantenfärbung eines Graphen \mathcal{NP} -vollständig ist [66]. Der Algorithmus von Johnson wird in [72] beschrieben. Approximationsalgorithmen für das Färbungsproblem mit einem besseren Wirkungsgrad werden in [13, 56, 14] diskutiert. Der Algorithmus von Turner wird in [119] besprochen. Untersuchungen über die durchschnittliche Höhe des Suchbaumes beim Backtracking-Verfahren findet man in [123].

Die Beweise für die Nichtexistenz approximativer Algorithmen für unabhängige Mengen und Cliques in Graphen findet man in [9, 10]. Die untere Schranke von n^ϵ für den Wirkungsgrad eines approximativen Färbungsalgorithmus stammt von Lund und Yannakakis [90]. Einen Überblick über die Entwicklung approximativer Algorithmen findet man in [75]. Untersuchungen über das Verhalten des Greedy-Algorithmus findet man in [86] und [53]. J.C. Culberson hat Varianten des Greedy-Algorithmus betrachtet [26]. Er hat auch für Graphfärbungen relevante Ressourcen wie z.B. Graphgeneratoren, Färbungsprogramme und Beispielgraphen zusammengestellt. Halldórsson hat einen approximativen Algorithmus mit asymptotischem Wirkungsgrad $4/3$ für das Färbungsproblem als Minimierungsproblem angegeben [57]. Den angegebenen Algorithmus A_2 für dieses Problem findet man in [61].

Das Problem des Handlungsreisenden ist wahrscheinlich das am häufigsten untersuchte unter allen \mathcal{NP} -vollständigen Problemen. Einen guten Überblick über die verschiedenen Lösungansätze findet man in dem Buch von Lawler, Lenstra, Rinnooy Kan und Shmoys [87]. Der Wirkungsgrad des Nächster-Nachbar Algorithmus wurde von Rosenkrantz, Stearns und Lewis näher untersucht [110]. Der Algorithmus mit dem zur Zeit besten asymptotischen Wirkungsgrad stammt von Christofides und ist in [23] beschrieben. Das Beispiel, welches den asymptotischen Wirkungsgrad von $3/2$ belegt, findet man in [25]. Die Lösung des TSP in der Praxis erfolgt zur Zeit meistens so, daß mittels eines schnellen

Verfahrens ein Weg bestimmt wird, der danach mit lokalen Techniken verbessert wird. Ein approximativer Algorithmus, der so verfährt und sich in der Praxis gut bewährt hat, stammt von S. Lin und B. Kernighan [89]. G. Reinelt hat einen Vergleich hinsichtlich der Laufzeit und der Qualität der Lösung von aktuellen Verfahren durchgeführt [107]. Aufgabe 13c stammt von Johnson [73], und die Aufgaben 31, 28 und 37 findet man in [104], [113] bzw. [77]. Einen Approximationsalgorithmus für die Bestimmung eines Steiner Baumes mit Wirkungsgrad 11/6 findet man in [127].

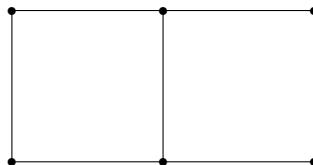
9.8 Aufgaben

- * 1. Die ersten approximativen Algorithmen wurden nicht für Graphen entwickelt. Ein Beispiel hierfür ist der Scheduling-Algorithmus von R.L. Graham. Die Aufgabe besteht darin, n Programme mit Laufzeiten t_1, \dots, t_n (ganze Zahlen) auf m Prozessoren eines Multiprozessorsystems aufzuteilen, so daß die Gesamtaufzeit minimal ist. Es ist bekannt, daß das zugehörige Entscheidungsproblem in \mathcal{NPC} liegt (sogar für $m = 2$). Der von Graham entwickelte Algorithmus betrachtet die n Programme nacheinander und ordnet jedes Programm dem Prozessor zu, der noch die geringste Last hat. Unter der Last eines Prozessors versteht man hierbei die Gesamtzeit aller Programme, die ihm zugeordnet wurden.

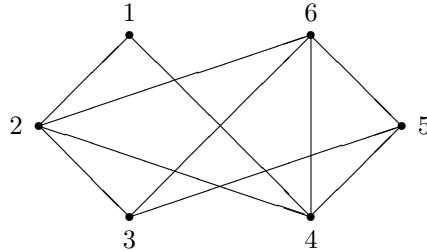
- Beweisen Sie die Gültigkeit folgender Schranke für den Wirkungsgrad dieses Algorithmus:

$$\mathcal{W}_A(n) \leq 2 - \frac{1}{m}$$

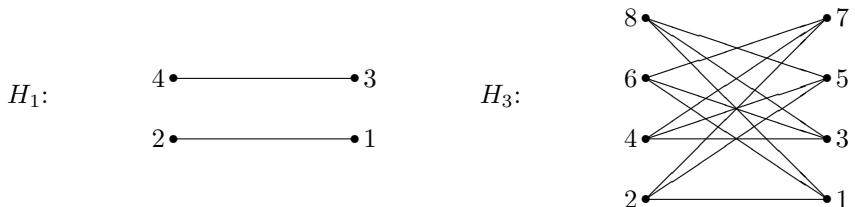
- Geben Sie eine Implementierung des Algorithmus mit Laufzeit $O(n \log m)$ an.
 - Zeigen Sie mittels eines Beispiels, daß diese obere Schranke auch angenommen werden kann.
 - Wie ändert sich der Wirkungsgrad, wenn die Programme zunächst sortiert werden, so daß ihre Laufzeiten absteigend sind?
- * 2. Betrachten Sie den Graphen aus Abbildung 9.1. Zeigen Sie, daß der Greedy-Algorithmus mit der Wahrscheinlichkeit $(4n - 5)/(4n - 2)$ eine 2-Färbung erzeugt. Mit welcher Wahrscheinlichkeit erzeugt der Greedy-Algorithmus eine Färbung mit mindestens n Farben? Wie verhalten sich diese Wahrscheinlichkeiten für wachsendes n ?
 - 3. Für welche Reihenfolgen der Ecken des folgenden Graphen vergibt der Greedy-Algorithmus die meisten bzw. die wenigsten Farben?



4. Wenden Sie den Algorithmus von Johnson und den Greedy-Algorithmus auf den folgenden Graphen an. Bestimmen Sie χ .



- * 5. Es sei G ein ungerichteter Graph und f eine Färbung von G , die genau c Farben verwendet. Ferner sei F_i die Menge der Ecken e von G mit $f(e) = i$ und π eine Permutation der Menge $\{1, \dots, c\}$. Die Ecken von G werden nun neu numeriert: zunächst die Ecken aus $F_{\pi(1)}$, danach die aus $F_{\pi(2)}$ und so weiter. Beweisen Sie, daß der Greedy-Algorithmus bezüglich dieser Numerierung eine Färbung erzeugt, die maximal c Farben verwendet. Geben Sie ein Beispiel an, bei dem die Anzahl der verwendeten Farben abnimmt. Gibt es unter der Voraussetzung $c > \chi(G)$ in jedem Fall eine Permutation π , so daß die neue Färbung bezüglich π weniger als c Farben verwendet?
6. a) Beweisen Sie, daß die Anzahl der Farben, die der Greedy-Algorithmus vergibt, durch $\max \{\min(i, g(e_i) + 1) \mid i = 1, \dots, n\}$ beschränkt ist.
- b) Zeigen Sie, daß der Ausdruck $\max \{\min(i, g(e_i) + 1) \mid i = 1, \dots, n\}$ minimal wird, wenn man die Ecken in der Reihenfolge absteigender Eckengrade betrachtet.
- c) Bestimmen Sie den asymptotischen Wirkungsgrad des Greedy-Algorithmus in der Variante, in der die Ecken in der Reihenfolge absteigender Eckengrade betrachtet werden. Betrachten Sie dazu die im folgenden definierten Graphen H_i mit $i \in \mathbb{N}$. Die Graphen H_1 und H_3 sind unten dargestellt. Der Graph H_i geht aus H_{i-1} hervor, indem noch die beiden Ecken $2i+1$ und $2i+2$ hinzugefügt werden. Die erste Ecke wird mit allen Ecken aus H_{i-1} mit geraden Nummern durch eine Kante verbunden und die zweite Ecke mit den restlichen Ecken.



7. Bestimmen Sie den Wirkungsgrad der folgenden beiden approximativen Algorithmen zur Bestimmung einer maximalen Clique in einem ungerichteten Graphen

G . Das zugehörige Entscheidungsproblem ist \mathcal{NP} -vollständig. Eine Implementierung dieser Algorithmen mit Aufwand $O(n + m)$ findet man in Abschnitt 2.8 und Aufgabe 27 auf Seite 50.

Algorithmus A_1 :

```
C := ∅;
while es gibt noch eine Ecke in G do begin
    Füge die Ecke e mit maximalem Eckengrad in G in C ein;
    Entferne e und alle nicht zu e benachbarten Ecken aus G;
end
```

Algorithmus A_2 :

```
C := G;
while C keine Clique do begin
    Sei e eine Ecke mit minimalem Eckengrad in C;
    Entferne e aus C;
end
```

(Hinweis: Wenden Sie A_1 auf einen Graphen an, der entsteht, wenn die Wurzel eines Baumes der Höhe 1 und $s+1$ Ecken mit einer beliebigen Ecke des vollständigen Graphen K_s verbunden wird. Für A_2 betrachten Sie einen Graphen, der entsteht, wenn eine beliebige Ecke aus dem vollständigen Graphen K_s mit einer beliebigen Ecke aus dem vollständig bipartiten Graphen $K_{s,s}$ verbunden wird.)

8. Beweisen Sie, daß für das Optimierungsproblem der maximalen Clique in einem ungerichteten Graphen $\mathcal{W}_{MIN} = 1$ oder $\mathcal{W}_{MIN} = \infty$ gilt.
9. Es sei G ein ungerichteter Graph mit Eckenmenge E und Z_1, Z_2 Zuordnungen von G , so daß $|Z_2| > |Z_1|$ ist. Es sei G' der Graph mit Eckenmenge E und den Kanten von G , die in Z_1 oder Z_2 , aber nicht in beiden Zuordnungen liegen. Beweisen Sie, daß es in G' mindestens $|Z_2| - |Z_1|$ eckendisjunkte Erweiterungsmengen bezüglich Z_1 gibt. (Hinweis: Zeigen Sie, daß die Zusammenhangskomponenten von G' in drei Gruppen zerfallen: in isolierte Ecken und geschlossene und offene Wege, deren Kanten abwechselnd aus $Z_2 \setminus Z_1$ bzw. $Z_1 \setminus Z_2$ sind. Beweisen Sie dann, daß die letzte Gruppe aus mindestens $|Z_2| - |Z_1|$ offenen Wegen besteht. Vergleichen Sie auch Aufgabe 1 aus Kapitel 7 auf Seite 234.)
- * 10. Zeigen Sie, daß die unten angegebene Funktion **greedy-Zuordnung** eine nicht erweiterbare Zuordnung für einen ungerichteten Graphen G bestimmt. Geben Sie eine Realisierung dieser Funktion an, die eine worst case Laufzeit von $O(n+m)$ hat. Bestimmen Sie mit Hilfe von Aufgabe 9 den Wirkungsgrad dieses Algorithmus.

```

function greedy-Zuordnung(G : Graph) : set of Kanten;
var
    Z : set of Kanten;
    i,j : Integer;
    k : Kante;
begin
    Z := ∅;
    for jede Ecke i do
        if es gibt eine Kante k = (i,j), welche mit keiner Kante
            aus Z eine gemeinsame Ecke hat then
                Z.einfügen(k);
    greedy-Zuordnung := Z;
end

```

- * 11. Bestimmen Sie einen Algorithmus A mit $\mathcal{W}_A(n) \leq 2$ und Aufwand $O(n+m)$, welcher für das Komplement eines ungerichteten Graphen G eine nicht erweiterbare Zuordnung bestimmt. Hierbei ist m die Anzahl der Kanten von G und nicht die von \bar{G} .
- * 12. Die Zuordnung Z , die die Funktion greedy-Zuordnung aus Aufgabe 10 erzeugt, hat die Eigenschaft, daß jeder Erweiterungsweg aus mindestens drei Kanten besteht. Ändern Sie die Funktion derart, daß jeder Erweiterungsweg bezüglich Z aus mindestens fünf Kanten besteht. Achten Sie darauf, daß die Laufzeit weiterhin linear bleibt. Bestimmen Sie den Wirkungsgrad dieses Algorithmus.
- 13. Eine *dominierende Menge* von Ecken eines ungerichteten Graphen mit Eckenmenge E ist eine Menge E' von Ecken, so daß jede Ecke $e \in E \setminus E'$ zu einer Ecke aus E' benachbart ist.
 - a) Was ist der Zusammenhang zwischen einer Eckenüberdeckung und einer dominierenden Menge?
 - b) Das Entscheidungsproblem, ob ein ungerichteter Graph G eine dominierende Menge mit maximal C Ecken hat, ist \mathcal{NP} -vollständig. Betrachten Sie folgenden Greedy-Algorithmus für das zugehörige Optimierungsverfahren: Wähle in G eine Ecke mit dem höchsten Eckengrad aus und füge diese in E' ein. Danach werden e und alle Nachbarn von e aus G entfernt. Dies wird solange wiederholt, bis der Graph G keine Ecken mehr hat.
Zeigen Sie, daß dieser Algorithmus eine dominierende Menge für G erzeugt, und geben Sie eine Realisierung an, deren Laufzeit $O(n+m)$ ist.
 - c) Bestimmen Sie den asymptotischen Wirkungsgrad. Betrachten Sie dazu die im folgenden definierte Folge von Graphen G_l mit $l \in \mathbb{N}$. Es sei T eine Matrix mit $l!$ Zeilen und l Spalten. Für $i = 1, \dots, l!$ und $j = 1, \dots, l$ ist

$$T_{ij} = (j-1)! + i.$$

Es sei

$$r = l! \sum_{j=1}^l \frac{1}{j}.$$

Nun werden $r + l!$ Mengen gebildet. Für $j = 1, \dots, l$ unterteile man die Einträge der j -ten Spalte in $l!/j$ Mengen mit je j Elementen. Diese Mengen werden in jeder Spalte von oben nach unten gebildet und mit S_1, \dots, S_r bezeichnet. Hinzu kommen noch $l!$ Mengen $S_{r+1}, \dots, S_{r+l!}$, wobei S_{r+i} gerade aus den Einträgen der i -ten Zeile von T besteht. Bilde nun den Graphen G_l , in dem die Mengen $S_1, \dots, S_{r+l!}$ die Ecken repräsentieren und zwei Ecken durch eine Kante verbunden sind, wenn die zugehörigen Mengen nichtleeren Schnitt haben. Für $l = 3$ ergibt sich folgende Matrix T

$$T = \begin{pmatrix} 1 & 7 & 13 \\ 2 & 8 & 14 \\ 3 & 9 & 15 \\ 4 & 10 & 16 \\ 5 & 11 & 17 \\ 6 & 12 & 18 \end{pmatrix}$$

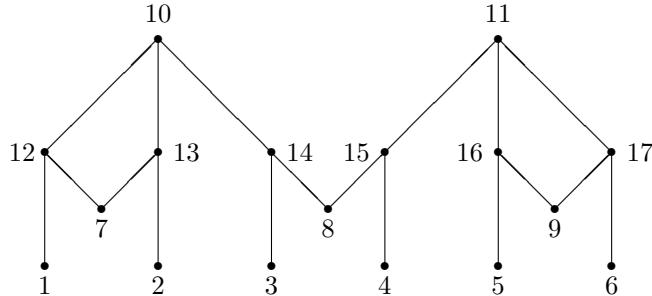
und folgende 17 Mengen

$$S_1 = \{1\} \quad S_2 = \{2\} \quad \dots \quad S_6 = \{6\}$$

$$S_7 = \{7, 8\} \quad S_8 = \{9, 10\} \quad S_9 = \{11, 12\} \quad S_{10} = \{13, 14, 15\} \quad S_{11} = \{16, 17, 18\}$$

$$S_{12} = \{1, 7, 13\} \quad S_{13} = \{2, 8, 14\} \quad S_{14} = \{3, 9, 15\} \quad \dots \quad S_{17} = \{6, 12, 18\}$$

Der Graph G_3 sieht nun folgendermaßen aus:



Beweisen Sie, daß die Ecken $\{r + 1, \dots, r + l!\}$ eine minimale dominierende Menge für G_l bilden. Der oben angegebene Algorithmus bestimmt jedoch die Menge $\{1, \dots, r\}$, falls man die Reihenfolge der Ecken geeignet wählt.

14. Ein ungerichteter Graph G heißt *k-partit*, wenn die Eckenmenge E von G in k disjunkte Teilmengen E_1, \dots, E_k aufgeteilt werden kann, so daß die Enden aller Kanten von G in verschiedenen Teilmengen E_i liegen. Die 2-partiten Graphen sind die bipartiten Graphen. Ein *k*-partiter Graph heißt *vollständig k-partit*, wenn alle Ecken aus verschiedenen Teilmengen E_i benachbart sind.

Beweisen Sie, daß der Greedy-Algorithmus jeden vollständig *k*-partiten Graphen unabhängig von der Reihenfolge der Ecken mit der minimalen Anzahl von Farben färbt. Gilt auch die Umkehrung dieser Aussage, d.h. färbt der Greedy-Algorithmus einen Graphen G für jede Reihenfolge der Ecken mit der minimalen Anzahl von Farben, so ist G vollständig *k*-partit.

15. Der approximative Färbungsalgorithmus A_2 beruht auf der Bestimmung einer maximalen Zuordnung des Graphen \overline{G}_X . Beweisen Sie, daß der Wirkungsgrad von A_2 immer noch durch $3/2$ beschränkt ist, wenn anstatt einer maximalen Zuordnung von \overline{G}_X eine Zuordnung verwendet wird, die mindestens $2/3$ soviel Kanten wie eine maximale Zuordnung enthält. Vergleichen Sie dazu die Aufgaben 11 und 12.
16. Das Färbungsproblem für Graphen, deren maximaler Eckengrad kleiner oder gleich 4 ist, liegt in \mathcal{NP} . Geben Sie einen linearen Algorithmus an, welcher für zusammenhängende Graphen, deren maximaler Eckengrad kleiner oder gleich 3 ist, eine minimale Färbung bestimmt (Hinweis: Satz von Brooks).
17. * a) Es sei G ein ungerichteter Graph mit $n \geq 3$ Ecken und e, f nicht benachbarte Ecken von G mit $g(e) + g(f) \geq n$. Es sei G' der Graph, der aus G durch Einfügen der Kante (e, f) entsteht. Beweisen Sie, daß G genau dann Hamiltonsch ist, wenn G' Hamiltonsch ist.
b) Beweisen Sie folgende Aussagen für einen ungerichteten Graphen G mit $n \geq 3$ Ecken:
- (i) Ist die Summe der Eckengrade jedes Paares nicht benachbarter Ecken von G mindestens n , so ist G Hamiltonsch.
 - (ii) Ist der Eckengrad jeder Ecke von G mindestens $n/2$, so ist G Hamiltonsch.
18. Beweisen Sie, daß der Petersen-Graph keinen Hamiltonschen Kreis besitzt.
19. Es sei G ein ungerichteter, kantenbewerteter, zusammenhängender Graph mit Eckenmenge $\{1, \dots, n\}$. Ein Untergraph B von G heißt *1-Baum*, falls der Grad der Ecke 1 in B gleich 2 ist und falls der von den Ecken $2, \dots, n$ induzierte Untergraph von B ein aufspannender Baum des entsprechenden induzierten Untergraphen von G ist. Ein *minimaler 1-Baum* von G ist ein 1-Baum, für den die Summe seiner Kantenbewertungen minimal unter allen 1-Bäumen von G ist. Minimale 1-Bäume werden zur Bestimmung von unteren Schranken für die Länge von Hamiltonschen Kreisen genutzt.
- a) Geben Sie einen Algorithmus zur Bestimmung eines minimalen 1-Baumes an. Bestimmen Sie den Aufwand des Algorithmus.
 - b) Es sei G ein kantenbewerteter vollständiger Graph und W ein geschlossener, einfacher Weg minimaler Länge von G , auf dem alle Ecken liegen. Beweisen Sie, daß die Länge von W mindestens so groß ist wie die Kosten eines minimalen 1-Baumes von G .
20. Es sei G ein kantenbewerteter vollständiger Graph, dessen Kantenbewertungen die Dreiecksungleichung erfüllen. Es sei W_1 ein geschlossener Kantenzug minimaler Länge, auf dem alle Ecken liegen, und W_2 ein geschlossener einfacher Weg minimaler Länge, auf dem alle Ecken liegen. Beweisen Sie, daß W_1 und W_2 die gleiche Länge haben.

- * 21. Es sei G ein kantenbewerteter vollständiger Graph, dessen Kantenbewertungen die Dreiecksungleichungen erfüllen. Beweisen Sie, daß der Wirkungsgrad des folgenden approximativen Algorithmus für Δ -TSP kleiner oder gleich 2 ist. Der Algorithmus baut einen geschlossenen einfachen Weg W schrittweise auf. In jedem Schritt wird der Weg um eine Ecke erweitert. Der Algorithmus startet mit einer beliebigen Ecke e_1 von G . Diese bildet den geschlossenen einfachen Weg W_1 . Sei nun $W_i = e_1, \dots, e_i, e_1$ der bisher konstruierte Weg. Unter allen Kanten (e, f) von G mit $e \in \{e_1, \dots, e_i\}$ und $f \notin \{e_1, \dots, e_i\}$ wird diejenige mit der kleinsten Bewertung gewählt. W_{i+1} entsteht aus W_i , indem die Ecke f vor e in W_i eingefügt wird. Am Ende setzt man $W = W_n$. Bestimmen Sie den Aufwand des Algorithmus.
- 22. Es sei G ein kantenbewerteter vollständiger Graph mit Eckenmenge E . Es sei $e \in E$ und $b_1 \leq b_2 \leq \dots \leq b_{n-1}$ die Bewertungen der zu e inzidenten Kanten. Setze $m(e) = b_1 + b_2$. Beweisen Sie folgende Aussage: Ist W ein geschlossener, einfacher Weg, auf dem alle Ecken von G liegen, so gilt:

$$\sum_{e \in E} m(e) \leq 2L(W)$$

- ** 23. Es sei G ein ungerichteter zusammenhängender Graph, bei dem die Eckengrade aller Ecken gerade sind. Ein geschlossener Kantenzug für G , auf dem alle Kanten liegen, kann mittels der folgenden Prozedur **eulersch** bestimmt werden. Die Prozedur arbeitet ähnlich wie die Tiefensuche. Im Unterschied zur Tiefensuche kann allerdings eine Ecke mehrmals besucht werden. Eine Ecke wird verlassen, falls alle inzidenten Kanten schon einmal verwendet wurden (unabhängig von der Richtung). Hierbei wird die Nummer der Ecke ausgegeben. Der Aufruf der Prozedur erfolgt durch **eulersch(1)**. Beweisen Sie, daß die ausgegebenen Ecken in dieser Reihenfolge einen geschlossenen Kantenzug bilden und daß jede Kante dabei genau einmal vorkommt. Zeigen Sie, daß die Prozedur so realisiert werden kann, daß der Aufwand $O(m)$ ist.

```

procedure eulersch(G : Graph; i : Integer);
var
    j : Integer;
begin
    for jeden Nachbar j von i do
        if Kante (i,j) ist noch nicht markiert then begin
            markiere die Kante (i,j);
            eulersch(j);
        end;
        ausgabe(i);
    end

```

- 24. Entwerfen Sie einen approximativen Algorithmus zur Bestimmung einer minimalen Kantenfärbung eines ungerichteten Graphen, basierend auf maximalen Zuordnungen. Solange der Graph noch Kanten enthält, wird eine maximale Zuordnung bestimmt. Die Kanten dieser Zuordnung erhalten alle die gleiche Farbe und werden danach entfernt. Bestimmen Sie den Aufwand des Algorithmus.

25. Es sei G ein ungerichteter Graph mit Eckenmenge E und Kantenmenge K . Bestimmen Sie den asymptotischen Wirkungsgrad des folgenden approximativen Algorithmus zur Bestimmung einer minimalen Eckenüberdeckung E' . Bestimmen Sie die Laufzeit des Algorithmus.

```

 $E' := \emptyset;$ 
Initialisiere K mit der Menge der Kanten von G;
while  $K \neq \emptyset$  do begin
    Es sei e eine Ecke aus E mit dem maximalen Eckengrad
    in dem von K induzierten Graphen;
    Füge e in E' ein;
    Entferne aus K alle zu e inzidenten Kanten;
end

```

26. Geben Sie ein Beispiel für einen ungerichteten Graphen an, dessen minimale Eckenüberdeckung mehr Ecken enthält als eine maximale Zuordnung Kanten.
- ** 27. Entwerfen Sie einen Algorithmus, der in linearer Zeit eine minimale Eckenüberdeckung für einen Baum bestimmt.
28. Es sei E' die Menge aller Ecken eines Tiefensuchebaumes eines ungerichteten Graphen G , die keine Blätter sind. Beweisen Sie, daß E' eine Eckenüberdeckung von G ist, die maximal doppelt so viele Ecken enthält wie eine minimale Eckenüberdeckung.
- * 29. Es sei G ein ungerichteter Graph mit Eckenmenge E . Ein Schnitt (X, \bar{X}) von G mit der höchsten Anzahl von Kanten heißt *maximaler Schnitt*. Das Entscheidungsproblem, ob ein ungerichteter Graph einen Schnitt mit mindestens C Kanten hat, ist \mathcal{NP} -vollständig. Betrachten Sie folgenden approximativen Algorithmus für das zugehörige Optimierungsproblem. Es sei X eine beliebige nichtleere Teilmenge von E . Solange es in X (bzw. \bar{X}) eine Ecke e gibt, so daß weniger als die Hälfte der Nachbarn von e in \bar{X} (bzw. in X) liegen, wird e von X nach \bar{X} (bzw. von \bar{X} nach X) verschoben. Zeigen Sie, daß maximal m Schritte notwendig sind und daß der Wirkungsgrad kleiner oder gleich 2 ist.
- * 30. Es seien X, Y, Z disjunkte Mengen mit je q Elementen und M eine Teilmenge des karthesischen Produktes $X \times Y \times Z$ mit folgender Eigenschaft: Sind $(x, b, c), (a, y, c), (a, b, z) \in M$, so liegt auch (a, b, c) in M (diese Eigenschaft wird paarweise Konsistenz genannt). Eine *3-dimensionale Zuordnung* von M ist eine Teilmenge M' von M mit der Eigenschaft, daß die Elemente von M' paarweise in allen drei Komponenten verschieden sind. Das Entscheidungsproblem, ob M eine 3-dimensionale Zuordnung, bestehend aus q Elementen besitzt, ist \mathcal{NP} -vollständig. Folgern Sie daraus, daß das Färbungsproblem für ungerichtete Graphen, deren Unabhängigkeitsszahl kleiner oder gleich 3 ist, ebenfalls \mathcal{NP} -vollständig ist.
- ** 31. Zeigen Sie, daß der Wirkungsgrad $\mathcal{W}_A(n)$ des folgenden approximativen Algorithmus A für das Optimierungsproblem einer maximalen unabhängigen Menge I in

einem ungerichteten zusammenhängenden Graphen G durch folgende Größe beschränkt ist:

$$\frac{\Delta}{2} \left(1 + \frac{1}{n-1} \right) + \frac{1}{2}$$

```

Initialisiere K mit der Menge der Kanten von G;
Initialisiere E mit der Menge der Ecken von G;
I1 := E; I2 := ∅;
while K ≠ ∅ do begin
    Es sei (e,f) eine beliebige Kante aus K;
    Entferne e und f aus I1;
    Entferne aus K alle zu e oder f inzidenten Kanten;
end;
while E ≠ ∅ do begin
    Es sei e die Ecke mit kleinsten Eckengrad in dem von E
        induzierten Untergraphen von G;
    Füge e in I2 ein;
    Entferne e und alle Nachbarn von e aus E;
end;
if |I1| > |I2| then
    I := I1
else
    I := I2
```

(Hinweis: Zeigen Sie, daß $\alpha(G) \leq n/2 + |I_1|/2$ und $|I_2| \geq (n - \delta - 1)/\Delta + 1$ gilt. Hierbei bezeichnet δ den kleinsten und Δ den größten Eckengrad von G .)

- * 32. Beweisen Sie, daß die Menge I_2 aus Aufgabe 31 mindestens $n/(\bar{d} + 1)$ Ecken enthält. Hierbei bezeichnet \bar{d} den durchschnittlichen Eckengrad.
- * 33. Es sei $f(x)$ eine monoton steigende Funktion mit $f(1) = 1$ und $k > 0$ eine natürliche Zahl. Die Funktion

```
function unabhängigeMenge(Graph G) : Set of Integer;
```

bestimmt für jeden Graph G mit $\chi(G) \leq k$ eine unabhängige Menge mit mindestens $f(n)$ Ecken. Beweisen Sie, daß die folgende Funktion **färbung** für jeden Graph G mit $\chi(G) \leq k$ eine Färbung mit maximal

$$\sum_{i=1}^n \frac{1}{f(i)}$$

Farben erzeugt.

```
var f : array[1..max] of Integer;

function färbung(G : Graph) : Integer;
```

```

var
    farbe, e : Integer;
    U Set of Integer;
begin
    Initialisiere f mit 0;
    farbe := 0;
    while G ≠ ∅ do begin
        farbe := farbe + 1;
        U := unabhangigeMenge(G);
        for alle Ecken e ∈ U do begin
            entferne e aus G;
            f[e] := farbe;
        end;
    end;
    färbung := farbe;
end

```

34. Gegeben ist ein ungerichteter, kantenbewerteter, zusammenhängender Graph G und B ein minimal aufspannender Baum von G mit Kosten K . Es sei L die Länge des kürzesten geschlossenen einfachen Weges, auf dem alle Ecken von G liegen. Gibt es unabhängig von G eine Konstante c , so daß $L \leq cK$ gilt?

- * 35. Beweisen Sie, daß es genau dann einen polynomialen Algorithmus für das Entscheidungsproblem des Handlungsreisenden gibt, wenn es einen polynomialen Algorithmus für das zugehörige Optimierungsproblem gibt.
- * 36. Beweisen Sie folgende Aussage: Das Entscheidungsproblem, ob ein kantenbewerteter, zusammenhängender, ungerichteter Graph für eine gegebene Teilmenge der Ecken einen minimalen Steiner Baum mit Gewicht kleiner oder gleich k besitzt, liegt in $\mathcal{NP}\mathcal{C}$.

Hinweis: Führen Sie eine Reduktion des \mathcal{NP} -vollständigen Entscheidungsproblems der minimalen Eckenüberdeckung in ungerichteten Graphen auf das Steiner Baum Problem durch. Konstruieren Sie hierzu für einen gegebenen Graphen G einen neuen Graphen G' , indem Sie eine neue Ecke e_n hinzufügen und diese mit allen Ecken von G verbinden und jede Kante (e, f) von G durch einen Pfad e, x_{ef}, f ersetzen. Dann gilt für den neuen Graphen $n' = n+m+1$ und $m' = 2m+n$. Jede Kante von G' trage die Bewertung 1. Es sei S die Menge der in G' neu hinzugekommenen Ecken ($|S| = m+1$). Beweisen Sie nun, daß sich die Größe einer minimalen Eckenüberdeckung von G und das Gewicht eines minimalen Steiner Baums von G' für S genau um m unterscheidet.

- * 37. Es sei G ein kantenbewerteter, zusammenhängender, ungerichteter Graph und S eine Teilmenge der Ecken von G . Beweisen Sie, daß der Wirkungsgrad des folgenden approximativen Algorithmus für die Bestimmung eines minimalen Steiner Baumes für S kleiner oder gleich 2 ist.
 - Konstruiere einen vollständigen, kantenbewerteten Graphen G' mit Eckenmenge S . In G' ist die Bewertung der Kante von i nach j gleich der Länge des kürzesten Weges von i nach j in G .

- b) Bestimme einen minimal aufspannenden Baum T' für G' .
- c) T' induziert in G einen Untergaphen G'' : Ersetze alle Kanten von T' durch die entsprechenden kürzesten Wege in G .
- d) Bestimme einen minimal aufspannenden Baum T'' für G'' .
- e) Entferne rekursiv alle Blätter von T'' , welche nicht in S liegen.
- f) Der resultierende Baum T^* ist ein Steiner Baum für S .

Bestimmen Sie die Laufzeit des Algorithmus.

- 38. Betrachten Sie noch einmal Aufgabe 35 auf Seite 288. Bestimmen Sie für die angegebene Eckenmenge mit dem in Aufgabe 37 beschriebenen Approximationsalgorithmus einen Steinerbaum. Ist das Ergebnis optimal?
- * 39. Beweisen Sie, daß es genau dann einen polynomialem Algorithmus für das Entscheidungsproblem der minimalen Eckenüberdeckung gibt, wenn es einen polynomialem Algorithmus für das zugehörige Optimierungsproblem gibt.

Anhang A

Angaben zu den Graphen an den Kapitelanfängen

Kapitel 1

Den dargestellten Graphen nennt man *Birkhoffsschen Diamant*. Er tritt im Beweis des 4-Farben-Satzes auf. Ein planarer Graph G mit $\chi(G) = 5$ heißt *minimal*, wenn jeder echte Untergraph von G eine 4-Färbung besitzt. Ist der 4-Farben-Satz falsch, so gibt es einen minimalen Graphen. Eine *Konfiguration* ist ein Graph und eine Einbettung in einen anderen Graphen. Eine Konfiguration heißt *reduzibel*, wenn sie in keinen minimalen Graphen eingebettet werden kann. Der Birkhoffssche Diamant ist ein Beispiel für eine reduzible Konfiguration. Der Beweis des 4-Farben-Satzes basiert auf der Konstruktion einer Menge M von unvermeidbaren Konfigurationen, d.h. jeder minimale Graph muß mindestens eine Konfiguration aus M enthalten.

Kapitel 2

Ein regulärer Polyeder ist ein dreidimensionaler Körper, der von regulären n -Ecken begrenzt wird. Es gibt genau fünf verschiedene reguläre Polyeder: Tetraeder, Hexaeder (Würfel), Oktaeder, Dodekaeder und Ikosaeder. Diese werden *Platonische Körper* genannt. Die Kanten eines Polyeders bilden einen planaren Graphen. Der dargestellte Graph gehört zu einem Dodekaeder.

Kapitel 3

Die Anzahlbestimmung für verschiedene Klassen von Graphen ist ein schwieriges Problem. Die Lösung beruht häufig darauf, daß man eine Funktionalgleichung für die sogenannte Anzahlpotenzreihe findet. Es sei t_n die Anzahl der verschiedenen Bäume mit n Ecken. Dann ist

$$t(x) = \sum_{n=1}^{\infty} t_n x^n$$

die Anzahlpotenzreihe für Bäume. Für $t(x)$ gelten folgende Gleichungen:

$$t(x) = T(x) - (T^2(x) - T(x^2))/2, \quad T(x) = x \exp \sum_{n=1}^{\infty} \frac{T(x^n)}{n}$$

Aus diesen beiden Gleichungen können die Werte der t_n bestimmt werden. Es gilt $t_6 = 6$. Die dargestellten Bäume sind alle verschiedenen Bäume mit genau sechs Ecken.

Kapitel 4

Der *Kantengraph* $L(G)$ eines Graphen G mit Kantenmenge K ist ein Graph mit Eckenmenge K . Zwei Ecken in $L(G)$ sind genau dann inzident, wenn die entsprechenden Kanten in G eine gemeinsame Ecke haben. Ein Graph ist genau dann ein Kantengraph, wenn er keinen der dargestellten neun Graphen als Untergraph hat.

Kapitel 5

Ein Graph heißt *eindeutig färbar*, wenn jede minimale Färbung die gleiche Zerlegung der Eckenmenge bewirkt. Eindeutig färbbare Graphen sind relativ rar. Interessanterweise haben Harary, Hedetniemi und Robinson gezeigt, daß es für jede natürliche Zahl c mit $c \geq 3$ eindeutig c -färbbare Graphen gibt, die keine Untergraphen vom Typ K_c besitzen [58]. Der dargestellte Graph ist eindeutig 3-färbar und besitzt keinen Untergraphen vom Typ K_3 .

Kapitel 6

Viele Algorithmen zur Bestimmung von maximalen Flüssen auf Netzwerken basieren auf dem Konzept des Erweiterungsweges. Durch eine geeignete Wahl von Erweiterungswegen wird in endlich vielen Schritten ein maximaler Fluß erreicht. Die Anzahl der Schritte ist dabei unabhängig von den Kapazitäten der Kanten durch eine von n und m abhängige Größe beschränkt. Der angegebene Graph zeigt, daß bei unbedachter Auswahl der Erweiterungswege die Anzahl der Schritte unabhängig von n und m wachsen kann. Bei geeigneter Wahl der Erweiterungswege ist die Anzahl der Schritte in dem dargestellten Netzwerk gleich C , der maximalen Kapazität einer Kante.

Kapitel 7

Ein wichtiges Konzept beim Lösen von Problemen ist das der Transformation. Dabei wird ein Problem auf ein anderes transformiert, für das ein Algorithmus bekannt ist. Ein Beispiel hierfür ist die Bestimmung von maximalen Zuordnungen in bipartiten Graphen. Dieses Problem kann auf die Bestimmung eines maximalen Flusses in einem 0-1-Netzwerk transformiert werden. Der abgebildete Graph ist ein Beispiel für ein solches Netzwerk.

Kapitel 8

Die meisten Algorithmen zur Bestimmung von kürzesten Wegen in Graphen basieren auf dem Optimalitätsprinzip. Dies sagt aus, daß jeder Teilweg eines kürzesten Weges für sich gesehen wieder ein kürzester Weg ist. Das Optimalitätsprinzip gilt unter der Voraussetzung, daß es keine geschlossenen Wege mit negativer Länge gibt. Der dargestellte Graph zeigt, daß dies eine notwendige Voraussetzung ist.

Kapitel 9

Der dargestellte Graph ist ein Ersetzungsgraph, mit dessen Hilfe gezeigt wird, daß das Entscheidungsproblem, ob ein planarer Graph eine 3-Färbung besitzt, in $\text{NP}\subset$ liegt. Das Problem, ob ein allgemeiner Graph G eine 3-Färbung besitzt, wird dabei auf planare Graphen reduziert. Dazu wird der Graph G auf eine beliebige Art in die Ebene eingebettet. Dabei kann es dazu kommen, daß sich Kanten schneiden. Jeder solche Schnittpunkt wird dabei durch den dargestellten Graphen ersetzt. Der so entstehende Graph G' ist planar und besitzt genau dann eine 3-Färbung, wenn G eine 3-Färbung besitzt. Der dargestellte Ersetzungsgraph G hat folgende Eigenschaften:

1. Für jede 3-Färbung f von G gilt: $f(1) = f(2)$ und $f(3) = f(4)$.
2. Es gibt 3-Färbungen f_1 und f_2 von G , so daß gilt:

$$f_1(1) = f_1(2) = f_1(3) = f_1(4) \text{ und } f_2(1) = f_2(2) \neq f_2(3) = f_2(4).$$

Anhang A

Angaben zu den Graphen an den Kapitelanfängen

Kapitel 1

Den dargestellten Graphen nennt man *Birkhoffsschen Diamant*. Er tritt im Beweis des 4-Farben-Satzes auf. Ein planarer Graph G mit $\chi(G) = 5$ heißt *minimal*, wenn jeder echte Untergraph von G eine 4-Färbung besitzt. Ist der 4-Farben-Satz falsch, so gibt es einen minimalen Graphen. Eine *Konfiguration* ist ein Graph und eine Einbettung in einen anderen Graphen. Eine Konfiguration heißt *reduzibel*, wenn sie in keinen minimalen Graphen eingebettet werden kann. Der Birkhoffssche Diamant ist ein Beispiel für eine reduzible Konfiguration. Der Beweis des 4-Farben-Satzes basiert auf der Konstruktion einer Menge M von unvermeidbaren Konfigurationen, d.h. jeder minimale Graph muß mindestens eine Konfiguration aus M enthalten.

Kapitel 2

Ein regulärer Polyeder ist ein dreidimensionaler Körper, der von regulären n -Ecken begrenzt wird. Es gibt genau fünf verschiedene reguläre Polyeder: Tetraeder, Hexaeder (Würfel), Oktaeder, Dodekaeder und Ikosaeder. Diese werden *Platonische Körper* genannt. Die Kanten eines Polyeders bilden einen planaren Graphen. Der dargestellte Graph gehört zu einem Dodekaeder.

Kapitel 3

Die Anzahlbestimmung für verschiedene Klassen von Graphen ist ein schwieriges Problem. Die Lösung beruht häufig darauf, daß man eine Funktionalgleichung für die sogenannte Anzahlpotenzreihe findet. Es sei t_n die Anzahl der verschiedenen Bäume mit n Ecken. Dann ist

$$t(x) = \sum_{n=1}^{\infty} t_n x^n$$

die Anzahlpotenzreihe für Bäume. Für $t(x)$ gelten folgende Gleichungen:

$$t(x) = T(x) - (T^2(x) - T(x^2))/2, \quad T(x) = x \exp \sum_{n=1}^{\infty} \frac{T(x^n)}{n}$$

Aus diesen beiden Gleichungen können die Werte der t_n bestimmt werden. Es gilt $t_6 = 6$. Die dargestellten Bäume sind alle verschiedenen Bäume mit genau sechs Ecken.

Kapitel 4

Der *Kantengraph* $L(G)$ eines Graphen G mit Kantenmenge K ist ein Graph mit Eckenmenge K . Zwei Ecken in $L(G)$ sind genau dann inzident, wenn die entsprechenden Kanten in G eine gemeinsame Ecke haben. Ein Graph ist genau dann ein Kantengraph, wenn er keinen der dargestellten neun Graphen als Untergraph hat.

Kapitel 5

Ein Graph heißt *eindeutig färbar*, wenn jede minimale Färbung die gleiche Zerlegung der Eckenmenge bewirkt. Eindeutig färbare Graphen sind relativ rar. Interessanterweise haben Harary, Hedetniemi und Robinson gezeigt, daß es für jede natürliche Zahl c mit $c \geq 3$ eindeutig c -färbare Graphen gibt, die keine Untergraphen vom Typ K_c besitzen [58]. Der dargestellte Graph ist eindeutig 3-färbar und besitzt keinen Untergraphen vom Typ K_3 .

Kapitel 6

Viele Algorithmen zur Bestimmung von maximalen Flüssen auf Netzwerken basieren auf dem Konzept des Erweiterungsweges. Durch eine geeignete Wahl von Erweiterungswegen wird in endlich vielen Schritten ein maximaler Fluß erreicht. Die Anzahl der Schritte ist dabei unabhängig von den Kapazitäten der Kanten durch eine von n und m abhängige Größe beschränkt. Der angegebene Graph zeigt, daß bei unbedachter Auswahl der Erweiterungswege die Anzahl der Schritte unabhängig von n und m wachsen kann. Bei geeigneter Wahl der Erweiterungswege ist die Anzahl der Schritte in dem dargestellten Netzwerk gleich C , der maximalen Kapazität einer Kante.

Kapitel 7

Ein wichtiges Konzept beim Lösen von Problemen ist das der Transformation. Dabei wird ein Problem auf ein anderes transformiert, für das ein Algorithmus bekannt ist. Ein Beispiel hierfür ist die Bestimmung von maximalen Zuordnungen in bipartiten Graphen. Dieses Problem kann auf die Bestimmung eines maximalen Flusses in einem 0-1-Netzwerk transformiert werden. Der abgebildete Graph ist ein Beispiel für ein solches Netzwerk.

Kapitel 8

Die meisten Algorithmen zur Bestimmung von kürzesten Wegen in Graphen basieren auf dem Optimalitätsprinzip. Dies sagt aus, daß jeder Teilweg eines kürzesten Weges für sich gesehen wieder ein kürzester Weg ist. Das Optimalitätsprinzip gilt unter der Voraussetzung, daß es keine geschlossenen Wege mit negativer Länge gibt. Der dargestellte Graph zeigt, daß dies eine notwendige Voraussetzung ist.

Kapitel 9

Der dargestellte Graph ist ein Ersetzungsgraph, mit dessen Hilfe gezeigt wird, daß das Entscheidungsproblem, ob ein planarer Graph eine 3-Färbung besitzt, in NPC liegt. Das Problem, ob ein allgemeiner Graph G eine 3-Färbung besitzt, wird dabei auf planare Graphen reduziert. Dazu wird der Graph G auf eine beliebige Art in die Ebene eingebettet. Dabei kann es dazu kommen, daß sich Kanten schneiden. Jeder solche Schnittpunkt wird dabei durch den dargestellten Graphen ersetzt. Der so entstehende Graph G' ist planar und besitzt genau dann eine 3-Färbung, wenn G eine 3-Färbung besitzt. Der dargestellte Ersetzungsgraph G hat folgende Eigenschaften:

1. Für jede 3-Färbung f von G gilt: $f(1) = f(2)$ und $f(3) = f(4)$.
2. Es gibt 3-Färbungen f_1 und f_2 von G , so daß gilt:

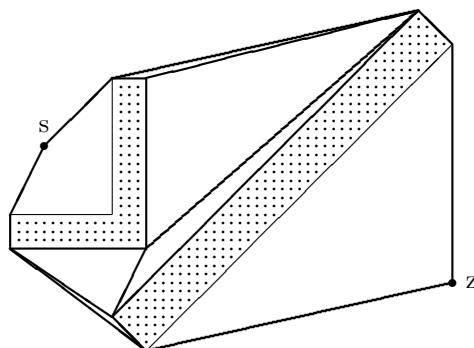
$$f_1(1) = f_1(2) = f_1(3) = f_1(4) \text{ und } f_2(1) = f_2(2) \neq f_2(3) = f_2(4).$$

Anhang B

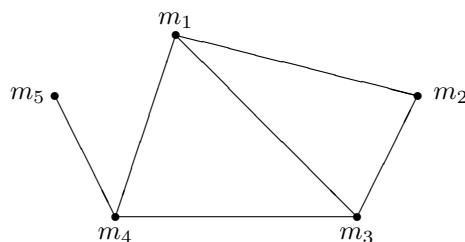
Lösungen der Übungsaufgaben

B.1 Kapitel 1

- Der Verbindungszusammenhang der drei Netzwerke von links nach rechts ist 3,3 und 2 und der Knotenzusammenhang ist 3,3 und 1.
- Die fett gezeichneten Linien bilden das System der Geradensegmente für den gesuchten Weg. Der kürzeste Weg von s nach z führt über die Segmente auf der unteren konvexen Hülle.



- Die optimale Reihenfolge der Produktionsaufträge ist 1–3–2–4 mit einer Gesamtumrüstzeit von 5.5.
- Im folgenden ist zunächst der Konfliktgraph und dann die Dispatchtabelle für die gegebene Klassenhierarchie und die angegebenen Methoden dargestellt.



Methode	m_1	m_2	m_3	m_4	m_5
Zeile	1	2	3	2	3

	A	B	C	D	E	F	G	H	I
1	$m_1 A$	$m_1 B$	$m_1 C$	$m_1 D$	$m_1 A$	$m_1 F$	$m_1 G$		
2		$m_4 B$	$m_4 B$	$m_4 B$	$m_2 E$	$m_2 E$	$m_2 E$	$m_4 H$	$m_4 H$
3			$m_3 C$	$m_3 C$	$m_3 E$	$m_3 E$	$m_3 E$		$m_5 I$

5. Es ergibt sich folgendes Gleichungssystem

$$PR(A) = 0.5 + 0.5 (PR(B) + PR(D)/2)$$

$$PR(B) = 0.5 + 0.5 PR(C)$$

$$PR(C) = 0.5 + 0.5 (PR(A)/2 + PR(D)/2)$$

$$PR(D) = 0.5 + 0.5 PR(A)/2$$

mit dieser Lösung:

$$PR(A) = 6/5 \quad PR(B) = 1 \quad PR(C) = 1 \quad PR(D) = 4/5.$$

6. Es gilt

$$\sum_{W \in M} PR(W) = |M|(1 - d) + d \sum_{W \in M} \sum_{D \in PL(W)} \frac{PR(D)}{LC(D)}.$$

Da jede Web-Seite W aus M genau $LC(W)$ -mal in der rechten Doppelsumme vorkommt, gilt:

$$\sum_{W \in M} PR(W) = |M|(1 - d) + d \sum_{W \in M} LC(W) \frac{PR(W)}{LC(W)}$$

und somit ist

$$\sum_{W \in M} PR(W) = |M|.$$

D.h. die Summe der Werte des PageRank aller Seiten ist gleich der Anzahl der Seiten. Für die Menge aller existierenden Web-Seiten gilt diese Aussage nicht.

B.2 Kapitel 2

- Über die Anzahl der Ecken geraden Grades kann keine allgemeine Aussage gemacht werden. Als Beispiel betrachte man die Graphen C_n .
- Es sei G ein ungerichteter Graph mit n Ecken und m Kanten. Nach Voraussetzung gilt $3n = 2m$. Somit ist 2 ein Teiler von n .

3. Es sei G ein bipartiter Graph mit Eckenmenge $E_1 \cup E_2$ und $|E_1| = n_1$. Dann gilt $m \leq n_1(n - n_1)$. Die rechte Seite nimmt ihren größten Wert für $n_1 = n/2$ (n gerade) bzw. $n_1 = (n + 1)/2$ (n ungerade) an. In beiden Fällen folgt $4m \leq n^2$.
4. Ist Δ der Grad einer Ecke, so gilt $\Delta|E_1| = \Delta|E_2|$ bzw. $|E_1| = |E_2|$.
5. Es sei G ein ungerichteter Graph mit $m > (n - 1)(n - 2)/2$. Angenommen, die Eckenmenge von G lässt sich so in zwei nichtleere Teilmengen E_1, E_2 zerlegen, daß keine Kante Ecken aus E_1 und E_2 verbindet. Ist $|E_1| = n_1$, so gilt

$$2m \leq n_1(n_1 - 1) + (n - n_1)(n - n_1 - 1)$$

(Gleichheit gilt genau dann, wenn beide Teilgraphen vollständig sind). Die rechte Seite dieser Ungleichung wird maximal für $n_1 = 1$ bzw. $n_1 = n - 1$. Dies führt in beiden Fällen zum Widerspruch.

Die Graphen $K_{n-1} \cup K_1$ haben die angegebene Anzahl von Ecken und sind nicht zusammenhängend.

6. $K_n \cup K_m$

7. Für jede Ecke e aus G gilt $g(e) + \bar{g}(e) = 5$, wobei $\bar{g}(e)$ den Eckengrad von e im Komplement \bar{G} von G bezeichnet. Somit kann man annehmen, daß es in G eine Ecke e mit $g(e) \geq 3$ gibt. Sind die Nachbarn von e paarweise nicht benachbart, so enthält \bar{G} den Graphen C_3 und ansonsten gilt dies für G .
8. Es sei $W = (e_0, e_1, \dots, e_s)$ ein geschlossener Weg. Von e_0 aus startend, verfolge man W , bis man zum ersten Mal an eine Ecke e_j kommt, an der man schon einmal war, d.h. $e_j = e_i$ für $i < j$. Da W geschlossen ist, gibt es auf jeden Fall eine solche Ecke. Dann ist e_i, \dots, e_j ein einfacher geschlossener Weg.
9. Es sei e eine Ecke mit Eckengrad $\Delta(G)$ und L die Menge der Ecken von G , welche nicht zu e benachbart sind. Dann ist nach Voraussetzung jede Kante von G zu einer Ecke von L inzident. Nun folgt:

$$m \leq \sum_{a \in L} g(a) \leq \Delta(G)|L| = \Delta(G)(n - \Delta(G))$$

10. Für jede Teilmenge X der Eckenmenge E bezeichne G_X den von X induzierten Untergraph von G . Die Bestimmung des Herzens erfolgt in zwei Phasen. Die erste Phase bestimmt einen Untergraphen G_F von G .

```

F := ∅;
while E ≠ ∅ do begin
    wähle eine Ecke e aus E und füge e in F ein;
    entferne e und alle Vorgänger von e in G_E aus E;
end

```

Von jeder Ecke $e \in E \setminus F$ kann mit einer Kante eine Ecke in F erreicht werden. Man beachte, daß es für eine neu in F eingefügte Ecke keine Nachfolger in F geben kann. In der zweiten Phase werden nun die Ecken von F in umgekehrter Reihenfolge, in der sie in F eingefügt wurden, betrachtet.

```

H := ∅
while F ≠ ∅ do begin
    wähle aus F die zuletzt eingefügte Ecke f aus;
    füge f in H ein und entferne f und alle Vorgänger von f in G_F aus F;
end

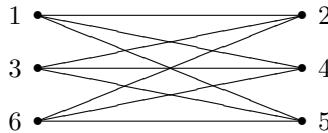
```

Zwischen den Ecken von H gibt es keine Kanten und von jeder Ecke $e \in E \setminus H$ gibt es einen Weg mit höchstens zwei Kanten zu einer Ecke in H .

11. Es sei $E_1 \cup E_2$ die Kantenmenge von G und $k = (e, f)$ die einzige Kante mit $e \in E_1$ und $f \in E_2$. Nach dem Entfernen von k haben die von E_1 und E_2 induzierten Graphen genau eine Ecke mit ungeradem Eckengrad. Dies steht im Widerspruch zu den Ergebnissen aus Abschnitt 2.1.
12. Die Adjazenzmatrix, die Adjazenzliste und die Kantenliste sehen wie folgt aus.

$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$	1 → 2 2 → 4, 6 3 → 4 4 → 5 5 → 3 6 → 1, 4, 5	<table border="1"> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr> <tr><td>1</td><td>2</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>6</td><td>6</td></tr> <tr><td>2</td><td>4</td><td>6</td><td>4</td><td>5</td><td>3</td><td>1</td><td>4</td><td>5</td></tr> </table>	1	2	3	4	5	6	7	8	9	1	2	2	3	4	5	6	6	6	2	4	6	4	5	3	1	4	5
1	2	3	4	5	6	7	8	9																					
1	2	2	3	4	5	6	6	6																					
2	4	6	4	5	3	1	4	5																					

- 13.



14. Die Funktion `nachfolger(i, j)` überprüft, ob es eine Kante von i nach j gibt.
(Zuerst folgt die einfache, danach die verbesserte Version)

```

function nachfolger (i, j : Integer) : Boolean;
var l, obergrenze : Integer;
begin
    nachfolger := false; l := N[i];
    if i = n then
        obergrenze := m
    else
        obergrenze := N[i+1]-1;
    while not nachfolger and l ≤ obergrenze do begin
        if A[l] = j then
            nachfolger := true;
        l := l + 1;
    end
end

function nachfolger (i, j : Integer) : Boolean;
var l : Integer;
begin
    nachfolger := false; l := N[i];

```

```

while not nachfolger and 1 ≤ N[i+1]-1 do begin
    if A[1] = j then
        nachfolger := true;
    l := l + 1;
end
end

```

Die Funktion `ausgrad(i)` bestimmt den Ausgrad der Ecke i (zuerst folgt die einfache, danach die verbesserte Version).

```

function ausgrad(i : Integer) : Integer;
begin
    if i = n then
        ausgrad := m + 1 - N[i]
    else
        ausgrad := N[i+1] - N[i];
end

function ausgrad(i : Integer) : Integer;
begin
    ausgrad := N[i+1] - N[i];
end

```

Die Funktion `eingrad(i)` bestimmt den Eingrad der Ecke i . Hier führt die Einführung des zusätzlichen Eintrages zu keiner Vereinfachung.

```

function eingrad(i : Integer) : Integer;
var e, l : Integer
begin
    e := 0;
    for l := 1 to m do
        if A[l] = i then
            e := e + 1;
    eingrad := e;
end

```

15. Es sei B das Feld der Länge $(n^2 - n)/2$ und A die Adjazenzmatrix des Graphen. Für natürliche Zahlen i, j setze:

$$f(i, j) = (i - 1)n - i(i + 1)/2 + j.$$

Dann gilt $A[i, j] = B[f(i, j)]$ für $i < j$ und $A[i, j] = B[f(j, i)]$ für $i > j$. Ist $i > j$, so sind die Ecken i und j genau dann benachbart, wenn $B[f(i, j)] \neq 0$ ist. Der Eckengrad $g(i)$ einer Ecke i bestimmt sich wie folgt:

$$g(i) = \sum_{k=1}^{i-1} B[f(k, i)] + \sum_{k=i+1}^n B[f(i, k)].$$

16. Um festzustellen, ob zwei Ecken i und j benachbart sind, muß für den Fall $j \geq i$ die Nachbarliste von i und im anderen Fall die von j durchsucht werden. Die Bestimmung der Anzahl der Nachbarn von i erfordert das Durchsuchen der Nachbarlisten der Ecken $1, \dots, i$. Gegenüber der normalen Adjazenzliste ändert sich der Aufwand für die erste Aufgabe nicht. Für die zweite Aufgabe ist der Aufwand jedoch höher. Für Ecke n müssen zum Beispiel alle Nachbarschaftslisten durchsucht werden, d.h. der Aufwand ist $O(n + m)$ gegenüber $O(g(n))$ bei normalen Adjazenzlisten.

17. Die Funktion `nachfolger` verwendet die Datenstruktur `Listenelement`:

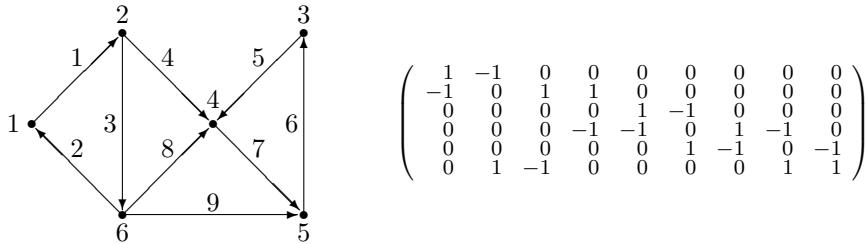
```

type Listenelement = record
    wert : Integer;
    nachfolger : zeiger Listenelement;
end

var A : array[1..max] of zeiger Listenelement;
function nachfolger(i, j : Integer) : Boolean;
var eintrag : Listenelement;
begin
    nachfolger := false;
    eintrag := A[i];
    while not nachfolger and eintrag ≠ nil and eintrag→wert ≤ j begin
        if eintrag→wert = j then
            nachfolger := true;
        eintrag := eintrag→nachfolger;
    end
end

```

18. Numeriert man die Kanten des Graphen aus Aufgabe 12 wie links dargestellt, so ergibt sich folgende Inzidenzmatrix.



Für ungerichtete Graphen entscheidet folgende Funktionen, ob j ein Nachfolger von i ist.

```

function nachfolger (i, j : Integer) : Boolean;
var k : Integer;
begin
    nachfolger := false;
    k := 1;
    while not nachfolger and k ≤ m do begin
        if C[j,k] = 1 and C[i,k] = -1 then
            nachfolger := true;
        k := k + 1;
    end
end

```

Für ungerichtete Graphen muß in der **if**-Bedingung die Zahl -1 durch 1 ersetzt werden.

Der Ausgrad einer Ecke i in einem gerichteten Graphen ist gleich der Anzahl der Einträge mit Wert 1 in der i -ten Zeile und der Eingrad gleich der der Einträge mit Wert -1 in der i -ten Zeile. Der Grad einer Ecke i in einem ungerichteten Graphen ist gleich der Anzahl der Einträge mit Wert 1 in der i -ten Zeile.

19. Adjazenzlisten basierend auf Feldern.
20. Eine Datei schließt sich genau dann selbst ein, wenn die zugehörige Ecke in dem gerichteten Graphen auf einem geschlossenen Weg liegt. Dies kann an den Diagonaleinträgen der Erreichbarkeitsmatrix erkannt werden.
21. Durch Ausmultiplizieren zeigt man, daß

$$S = A + A^2 + \dots + A^x \text{ mit } x = 2^{\lfloor \log_2 n \rfloor + 1} - 1$$

ist. Da x mindestens $n - 1$ ist, folgt mit Hilfe des in Abschnitt 2.6 bewiesenen Lemmas, daß E die Erreichbarkeitsmatrix von G ist. Zur Bestimmung von S sind $2\lfloor \log_2 n \rfloor$ Matrixmultiplikationen notwendig.

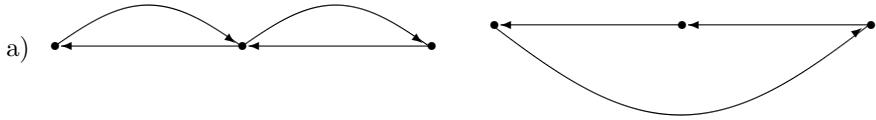
22. a) $n^2 + 2n + 3 \leq 6n^2$ für $n \geq 1$.
 b) $\log n^2 = 2 \log n$ für $n \geq 1$.
 c) $\sum_{i=1}^n i = n(n+1)/2 \leq n^2$ für $n \geq 1$.
 d) $\log n! = \sum_{i=1}^n \log i \leq n \log n$ für $n \geq 1$.
23. Es sei A die Adjazenzmatrix des Graphen. Eine Ecke i ist genau dann eine Senke, wenn die i -te Zeile von A nur Nullen enthält und die i -te Spalte bis auf den i -ten Eintrag keine Nullen enthält. Für einen Eintrag a_{ji} von A mit $i \neq j$ bedeutet dies: Ist $a_{ji} = 0$, so ist i keine Senke und ist $a_{ij} \neq 0$, so ist j keine Senke. Der folgende Algorithmus entdeckt in jedem Schritt eine Ecke, welche keine Senke ist.

```
i := 1;
for j := 2 to n do
  if A[j,i] = 0 then
    i := j;
```

Nachdem diese Schleife durchlaufen wurde, gibt es nur noch einen Kandidaten für die Senke, den letzten Wert von i . Die Überprüfung, ob i eine Senke ist, kann mit Aufwand $O(n)$ durchgeführt werden.

Liegt die Adjazenzliste des Graphen vor, so stellt man mit Aufwand $O(n)$ fest, ob es genau eine Ecke mit Ausgrad 0 gibt. Falls nicht, so gibt es keine Senke. Andernfalls überprüft man mit Aufwand $O(m)$, ob e in den Nachfolgelisten aller anderen Ecken vorkommt. Falls ja, so ist e eine Senke und ansonsten gibt es keine Senke.

24. Folgende einfache Eigenschaft von transitiven Reduktionen wird im folgenden verwendet: Es sei $k = (e, f)$ eine Kante in einer transitiven Reduktion R . Dann verwendet jeder Weg in R von e nach f die Kante k .



- b) Angenommen es gibt zwei verschiedene transitive Reduktionen R_1 und R_2 . Es sei $k = (e, f)$ eine Kante in $R_1 \setminus R_2$. Da f von e in R_2 erreichbar sein muß, gibt es eine Ecke v , so daß es in R_2 einen Weg von e über v nach f gibt. Da es in R_1 keine geschlossenen Wege gibt, existieren dort Wege von e nach v und von v nach f , welche k nicht enthalten. Dies kann aber nicht sein.
- c) Es sei G der vollständig bipartite Graph mit Eckenmenge $E_1 \cup E_2$ und $|E_1| = |E_2| = n/2$. Jede Kante (e_1, e_2) mit $e_1 \in E_1$ und $e_2 \in E_2$ wird nun in Richtung von den Ecken aus E_2 orientiert. Dieser Graph hat genau die angegebene Eigenschaft.
- d) Es sei R die transitive Reduktion eines gerichteten Graphen ohne geschlossene Wege. Angenommen es gibt eine Kante $k = (e, f)$ in R , mit $\text{maxlen}(e, f) > 1$ ist, d.h. es gibt einen Weg W von e nach f , welcher mindestens zwei Kanten enthält. Da der Graph keine geschlossenen Wege besitzt, ist k nicht in W enthalten. Dies führt zum Widerspruch.
25. Der Algorithmus verwendet die in Aufgabe 15 angegebene Funktion f , um Paare von Ecken in einem eindimensionalen Feld A zu indizieren. Das Feld wird mit 0 initialisiert. Haben zwei Ecken i und j einen gemeinsamen Nachbarn, so wird dieser an der Stelle $A[f(i, j)]$ eingetragen.

```

var A : array[1..(n-1)n/2] of Integer;
Initialisiere A mit 0;
for jede Ecke e do
  for jeden Nachbar i von e do
    for jeden Nachbar j ≠ i von e do
      if A[f(i,j)] ≠ 0 then
        exit('e,i,j und A[f(i,j)] bilden einen geschlossenen Weg');
      else
        A[f(i,j)] := e;
exit('Es gibt keinen geschlossenen Weg mit vier Ecken');

```

Wurde der Algorithmus nach $n(n-1)/2$ Schritten noch nicht beendet, so sind alle Einträge von A ungleich 0 und der Algorithmus endet im nächsten Schritt. Der Aufwand ist $O(n^2)$.

Um festzustellen, ob ein Graph einen Untergraph vom Typ $K_{s,s}$ hat, muß ein Feld der Länge ($\binom{n}{s}$) betrachtet werden. Ferner werden für jede Ecke e alle s -elementigen Teilmengen der Nachbarn von e betrachtet. Daraus ergibt sich ein Aufwand von $O(n^s)$.

26. Es sei $k = (i, j)$ die zusätzliche Kante und $k' = (l, s)$ eine Kante, welche neu im transitiven Abschluß liegt (d.h. $E'[l, s] = 1$ und $E[l, s] = 0$). Dann muß es einen einfachen Weg von l nach s geben, welcher k enthält. Da E die Erreichbarkeitsmatrix ist, gilt: $E[l, i] = 1$ und $E[j, s] = 1$. D.h. jede neue Kante geht von einem Vorgänger von i zu einem Nachfolger von j . Folgender Algorithmus bestimmt E' mit Aufwand $O(n^2)$.

```

Initialisiere E' mit E;
for l := 1 to n do
    if E[l, i] = 1 then
        for s := 1 to n do
            if E[j, s] = 1 then
                E'[l, s] = 1;

```

Es sei G ein Graph, welcher aus zwei Teilgraphen G_1 und G_2 mit je $n/2$ Ecken besteht, so daß es keine Kante von G_2 nach G_1 gibt. Ferner besitze G_1 eine Ecke j , von der aus jede Ecke aus G_1 erreichbar ist, und in G_2 eine Ecke i , welche von jeder Ecke aus G_2 erreichbar ist. Wird die Kante (i, j) neu in G eingefügt, so hat der neue transitive Abschluß $n^2/4$ zusätzliche Kanten.

27. Der angegebene Algorithmus ist ein Greedy-Algorithmus. Eine Implementierung mit Aufwand $O(n + m)$ kann mit den in Abschnitt 2.8 vorgestellten Datenstrukturen erreicht werden. Der einzige Unterschied besteht darin, daß die Ecke mit kleinstem Eckengrad entfernt wird, hierzu wird ein Zeiger auf das Ende der Liste `EckengradListe` verwaltet. Ferner werden noch Zähler für die Anzahl der in C verbliebenen Kanten und Ecken verwaltet. Mit diesen Zählern kann einfach überprüft werden, ob C eine Clique ist.

B.3 Kapitel 3

1. Es sei G ein zusammenhängender Graph.
 - a) Ist G ein Baum, so gilt $m = n - 1 < n$. Gilt umgekehrt $n > m$, so betrachte man einen aufspannenden Baum B von G . Da B $n - 1$ Kanten hat, gilt $m = n - 1$ und somit $G = B$.
 - b) Wenn G einen einzigen geschlossenen Weg W enthält, so erhält man einen aufspannenden Baum, falls man eine beliebige Kante aus W entfernt. Somit hat G genau $n - 1 + 1 = n$ Kanten. Ist umgekehrt die Anzahl der Kanten in G gleich der Anzahl der Ecken, so besteht G aus einem aufspannenden Baum und einer zusätzlichen Kante. Diese bewirkt, daß es in G genau einen geschlossenen Weg gibt.
2. Ein Graph W_n mit Eckenmenge $\{1, \dots, n\}$ und n ungerade heißt *Windmühlen-graph*, wenn er folgende Kantenmenge besitzt:

$$\{(1, i) | i = 2, \dots, n\} \cup \{(i, i+1) | i = 2, 4, 6, \dots, n-1\}.$$

Der Graph W_n hat $3(n - 1)/2$ Kanten. Die Graphen W_n haben die angegebene Eigenschaft. Im folgenden wird bewiesen, daß ein Graph G mit der angegebenen Eigenschaft ein Windmühlengraph ist.

Es sei e eine beliebige Ecke und G_e der von e und den Nachbarn von e induzierte Untergraph. Man sieht leicht, daß G_e ein Windmühlengraph ist. Falls alle Ecken den Grad 2 haben, so ist $G = W_3 = K_3$. Hat G eine Ecke e mit $g(e) > 2$, so daß alle anderen Ecken zu e inzident sind, so ist G ebenfalls ein Windmühlengraph. Angenommen G hat keine dieser beiden Eigenschaften. Es sei e eine Ecke mit maximalem Eckengrad Δ . Dann ist $\Delta > 2$. Es sei N die Menge der Nachbarn von e und $f \neq e$ eine Ecke, welche nicht in N ist. Für jede Ecke $u \in N$ gibt es genau eine Ecke a_u , so daß f, a_u, u ein Weg ist. Für $u_1 \neq u_2$ gilt $a_{u_1} \neq a_{u_2}$, sonst wäre e, u_1, a_{u_1}, u_2, e ein geschlossener Weg der Länge vier. Somit gilt:

$$\Delta \geq g(f) \geq g(e) = \Delta, \text{ bzw. } g(f) = g(e)$$

Im folgenden wird gezeigt, daß G regulär ist. Dazu muß noch gezeigt werden, daß $g(e) = g(n)$ für jede Ecke $n \in N$ gilt. Wiederholte man die Argumentation aus dem letzten Abschnitt mit f anstelle von e , so folgt daß der Eckengrad jeder nicht zu f benachbarten Ecke ebenfalls Δ ist. Somit verbleibt maximal ein Nachbar n von e , von dem noch nicht gezeigt wurde, daß $g(n) = \Delta$ ist. Wiederholte man die letzte Argumentation für einen Nachbarn $n_1 \neq n$ von e , so folgt auch $g(n) = \Delta$, d.h. G ist regulär.

Es sei A die Adjazenzmatrix von G . Da G regulär ist, folgt aus der Voraussetzung $A^2 = (\Delta - 1)I + E$. Hierbei ist I die Einheitsmatrix und E die Matrix, deren Einträge alle gleich 1 sind. Die Matrizen A und E sind symmetrisch und es gilt $AE = EA = \Delta I$. Somit sind A und E simultan diagonalisierbar. Die Eigenwerte von E sind 1 und 0 mit den Vielfachheiten 1 und $n - 1$. Der Eigenvektor zum Eigenwert 1 ist $v_1 = (1, \dots, 1)$. Als Eigenvektor von A hat v_1 den Eigenwert Δ . Es sei v ein weiterer gemeinsamer Eigenvektor. Dann ist $Ev = 0$ und somit $A^2v = (\Delta - 1)v$ bzw. $Av = \pm\sqrt{\Delta - 1}v$. Somit sind $\Delta, \sqrt{\Delta - 1}$ und $-\sqrt{\Delta - 1}$ die Eigenwerte von A mit den Vielfachheiten 1, r und s . Es gilt $n = 1 + r + s$. Betrachtet man die Spur von A , so folgt $s - r = \Delta/\sqrt{\Delta - 1}$. Da $s - r$ eine ganze Zahl ist, muß $\Delta - 1$ eine Quadratzahl sein, deren Wurzel Δ teilt. Hieraus folgt $\Delta = 2$ im Widerspruch zur Annahme. Somit ist G ein Windmühlengraph.

3. Die Aussage wird durch vollständige Induktion bewiesen. Für $n = 2$ ist $d_1 = d_2 = 1$ und der Graph C_2 ist der gesuchte Baum. Sei nun $n > 2$. Es muß Indizes i, j mit $d_i = 1$ und $d_j > 1$ geben. Entfernt man d_i und d_j aus der Zahlenfolge und fügt die Zahl $d_j - 1$ ein, so kann die Induktionsvoraussetzung angewendet werden. An den so erhaltenen Baum muß nur noch eine zusätzliche Kante an die Ecke mit Grad $d_j - 1$ angehängt werden.
4. Würde es in \overline{B} drei Zusammenhangskomponenten geben, so gäbe es in B einen geschlossenen Weg der Länge drei. Da B ein Baum ist, kann das nicht sein. Nach Voraussetzung besteht B somit aus zwei Zusammenhangskomponenten. Gäbe es in einer der beiden Zusammenhangskomponenten zwei nicht inzidente Ecken, so würde dies wieder zu einem Widerspruch führen. Somit sind beide Zusammenhangskomponenten vollständige Graphen. Hätten beide Komponenten mehr als

eine Ecke, so gäbe es einen geschlossenen Weg der Länge vier in B . Somit muß eine der beiden Zusammenhangskomponenten aus genau einer Ecke bestehen.

5. Ein Binärbaum mit der angegebenen Eigenschaft hat $2b - 1$ Ecken. Der Beweis erfolgt durch vollständige Induktion. Für $b = 1$ ist die Aussage klar. Es sei nun $b > 1$ und e eine Ecke auf dem vorletzten Niveau von B . Dann hat e genau zwei Nachfolger. Entfernt man diese aus B , so erhält man einen Binärbaum mit $b - 1$ Blätter für den die Voraussetzung erfüllt ist. Somit hat dieser Baum $2(b - 1) - 1$ Blätter. Daraus folgt die Aussage.
6. Der Beweis erfolgt durch vollständige Induktion nach der Anzahl der inneren Ecken. Für $i = 1$ ist die Aussage klar. Sei nun $i > 1$ und x eine Ecke auf dem vorletzten Niveau. Entfernt man alle Nachfolger von x , so gilt für den entstandenen Wurzelbaum B_x nach Induktionsvoraussetzung:

$$|B_x| = \sum_{e \in E \setminus B, e \neq x} (\text{anz}(e) - 1) + 1.$$

Da $|B| = |B_x| + \text{anz}(x) - 1$ gilt, ist die Aussage bewiesen.

7. Die Anzahl der Blätter in einem Binärbaum ist genau dann maximal, wenn alle Niveaus voll besetzt sind. Somit hat ein Binärbaum der Höhe h maximal 2^h Blätter.
8. Es sei G quasi stark zusammenhängend. Folgendes Verfahren bestimmt eine Wurzel w von G . Es sei zunächst w eine beliebige Ecke von G . Gibt es eine Ecke e , welche nicht von w erreichbar ist, so muß es eine Ecke v geben, so daß e und w von v erreichbar sind. Setze w gleich v und wiederhole das Verfahren bis alle Ecken von w erreichbar. Besitzt G eine Wurzel, so folgt direkt, daß G auch quasi stark zusammenhängend ist.
9. Es werden folgende Typen zur Darstellung von Dateien und Verzeichnissen verwendet.

```

Eintrag = record
    name : String;
    verzeichnis : Boolean;
    inhalt : zeiger Eintragliste;
    daten : Inhaltstyp;
  end

Eintragliste = record
    inhalt : zeiger Eintrag;
    nachbar : zeiger Eintragliste;
  end

```

Folgende Prozedur gibt die Namen aller Dateien im angegebenen Verzeichnis und allen darunterliegenden Unterverzeichnissen aus.

```

procedure ausgabeVerzeichnis(e : zeiger Eintrag);
var naechster : zeiger Eintragliste;
begin
  if e ≠ nil then begin
    if e → verzeichnis then begin
      ausgabe(Verzeichnis: , e → name);
      naechster := e → inhalt;
    
```

```

        while naechster ≠ nil do begin
            ausgabeVerzeichnis(naechster → inhalt);
            naechster := naechster → nachbar;
        end
    else
        ausgabe(Datei: , e → name);
    end
end

```

10. Um einen Eintrag mit dem Schlüssel x in einem binären Suchbaum zu löschen, muß zunächst die zugehörige Ecke e lokalisiert werden. Falls e ein Blatt oder genau einen Nachfolger hat, so kann das Löschen einfach durchgeführt werden. Falls jedoch e zwei Nachfolger hat, so ist der Vorgang aufwendiger. Ein Algorithmus zum Löschen von Einträgen arbeitet wie folgt.
- Falls e keine Nachfolger hat, so lösche e .
 - Falls e genau einen Nachfolger hat, so ersetze e durch seinen Nachfolger.
 - Falls e zwei Nachfolger hat, so ersetze e durch die Ecke mit dem größten Schlüssel im linken Teilbaum oder durch die Ecke mit dem kleinsten Schlüssel im rechten Teilbaum von e . Um die Ecke mit dem größten (kleinsten) Schlüssel im linken (rechten) Teilbaum zu finden, verfolge man vom linken (rechten) Nachfolger der Ecke startend immer den rechten (linken) Nachfolger bis ein Blatt erreicht ist, dies ist die gewünschte Ecke.
11.

```

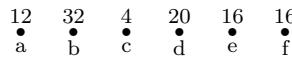
procedure aufsteigendeReihenfolge(b : Suchbaum);
begin
    if b ≠ nil then begin
        aufsteigendeReihenfolge(b → linkerNachfolger);
        ausgabe(b → Daten);
        aufsteigendeReihenfolge(b → rechterNachfolger);
    end
end

```
12. Bei einem Binärbaum mit zehn Ecken minimaler Höhe sind die ersten drei Ebenen voll besetzt und auf der vierten Ebene sind drei Ecken. Der Baum hat also die Höhe drei. Entartet der Suchbaum zu einer linearen Liste, so liegt ein Binärbaum maximaler Höhe vor (vergleichen Sie Abbildung 3.9).
13. Liegt a auf einem höheren Niveau als b , so ist $l_a > l_b$. Wäre die Wahrscheinlichkeit von a echt kleiner als die von b , so könnte man die Codierung von a und b vertauschen und so einen Präfix-Code mit kleinerer mittlerer Wortlänge erhalten. Dies widerspricht der Optimalität des Huffman-Algorithmus, die im folgenden bewiesen wird.
- Es seien p_1, \dots, p_n die Häufigkeiten der Zeichen in aufsteigender Reihenfolge. Im folgenden werden die Ecken mit ihren Häufigkeiten identifiziert. Zunächst wird gezeigt, daß es einen Präfix-Code minimaler Wortlänge gibt, in dessen Binärbaum es auf dem vorletzten Niveau eine Ecke gibt, deren Nachfolger p_1 und p_2 sind. Dazu betrachte man eine beliebige Ecke x auf dem vorletzten Niveau. Es seien c und d mit $c \leq d$ die Nachfolger von x . Falls $c = p_1$ und $d = p_2$, so ist die Aussage

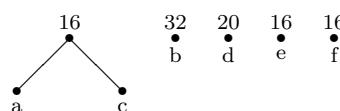
erfüllt. Ist $d > p_2$, so folgt aus der Minimalität des Codes $l_{p_2} \geq l_d$. Da d auf dem untersten Niveau liegt, gilt $l_d \geq l_{p_2}$. Somit ist $l_d = l_{p_2}$ und analog $l_c = l_{p_1}$. Vertauscht man nun p_2 und d bzw. p_1 und c im Binärbaum, so erhält man den gewünschten Präfix-Code.

Die Optimalität des vom Huffman-Algorithmus erzeugten Präfix-Code wird durch vollständige Induktion nach n bewiesen. Für $n = 1, 2$ ist die Optimalität offensichtlich gegeben. Sei nun $n > 2$ und B ein optimaler Binärbaum für p_1, \dots, p_n , in dem p_1 und p_2 Brüder sind. Man entferne die zu p_1 und p_2 gehörenden Blätter und markiere das neu entstandene Blatt mit $p_1 + p_2$. Es sei B' der neue Baum. Dies ist ein Binärbaum für die Häufigkeiten $p_1 + p_2, p_3, \dots, p_n$. Für die mittleren Wortlängen l_B von B und $l_{B'}$ von B' gilt $l_B = l_{B'} + p_1 + p_2$. Sei nun B'_1 ein vom Huffman-Algorithmus erstellter Binärbaum für die Häufigkeiten $p_1 + p_2, p_3, \dots, p_n$. Dann ist B'_1 nach Induktionsvoraussetzung optimal. Es sei B_1 der Baum, welcher aus B'_1 hervorgeht, indem die Ecke $p_1 + p_2$ zwei Nachfolger mit den Häufigkeiten p_1 und p_2 bekommt. B_1 ist gerade der vom Huffman-Algorithmus erstellte Baum für die Häufigkeiten p_1, \dots, p_n . Nun gilt $l_{B_1} = l_{B'_1} + p_1 + p_2$. Nun kann aber l_{B_1} nicht echt größer als l_B sein, denn dies würde $l_{B'_1} > l_{B_1}$ implizieren, was wegen der Optimalität von B'_1 nicht gilt. Somit ist $l_{B_1} = l_B$, d.h. B_1 ist optimal.

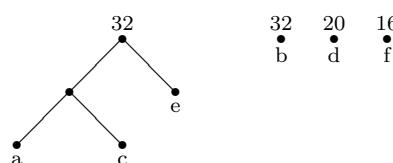
14. a)



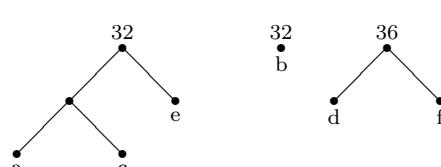
b)



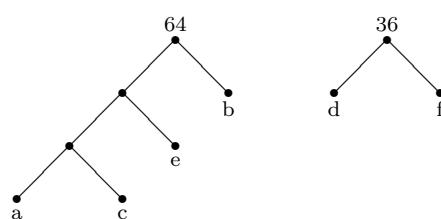
c)

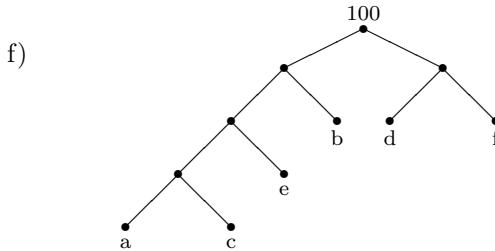


d)



e)





Der erzeugte Präfix-Code hat eine mittlere Codewortlänge von 2.48.

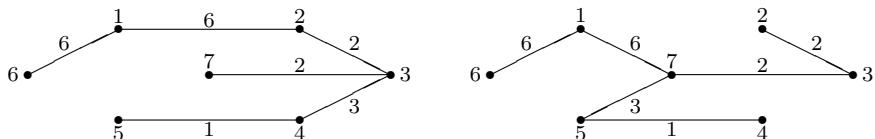
a:	0 0 0 0	c:	0 0 0 1	e:	0 0 1
b:	0 1	d:	1 0	f:	1 1

15.

```
var huffman : Wald;
procedure codewörter(wurzel : Integer);
var S : stapel of Integer;
begin
  if huffman.Ecken[wurzel].Bewertung ist ein Zeichen then begin
    ausgabe(Code für , huffman.Ecken[wurzel].Bewertung, ist:);
    ausgabe(Inhalt von S von unten nach oben);
  end
  else begin
    S.einfügen(0);
    codewörter(huffman.Ecken[wurzel].linkerNachfolger);
    S.entfernen;
    S.einfügen(0);
    codewörter(huffman.Ecken[wurzel].rechterNachfolger);
    S.entfernen;
  end
end
```

Der Aufruf `codewörter(huffman.wurzeln[1])` gibt den Präfix-Code aus.

16. Die Kosten eines minimal aufspannenden Baumes betragen 20.



17. Ein aufspannender Baum B von G , der k enthält, heißt k -minimal aufspannender Baum von G , falls kein anderer aufspannender Baum B' von G existiert, dessen Kosten niedriger sind und der k enthält. Der Beweis der folgenden Aussage erfolgt analog zum Beweis des ersten Lemmas in Abschnitt 3.6.

Es sei G ein kantenbewerteter zusammenhängender Graph mit Eckenmenge E und $k = (e, f)$ eine Kante von G . Ferner sei U eine Teilmenge von E mit $e, f \in U$ und (u, v) eine Kante mit minimalen Kosten mit $u \in U$ und $v \in E \setminus U$. Dann existiert ein k -minimal aufspannender Baum von G , der die Kante (u, v) enthält.

Es seien nun k_0, k_1, \dots, k_{m-1} die Kanten von G mit $k_0 = k$, wobei die Kanten k_1, \dots, k_{m-1} nach aufsteigenden Bewertungen sortiert sind. Der Algorithmus von

Kruskal wird auf G angewendet, wobei die Kanten in der angegebenen Reihenfolge betrachtet werden. Auf diese Art entsteht ein aufspannender Baum von G , welcher k enthält. Der Beweis, daß dies auch ein k -minimal aufspannender Baum ist, erfolgt analog zum Korrektheitsbeweis des Algorithmus von Kruskal.

18. Zur Konstruktion eines maximal aufspannenden Baumes kann jeder Algorithmus zur Bestimmung eines minimal aufspannenden Baumes verwendet werden. Dazu muß nur die Relation \leq durch \geq ersetzt werden. Eine andere Möglichkeit besteht darin, die Bewertung b_k jeder Kante k auf $C - b_k$ zu ändern, wobei C die höchste Bewertung aller Kanten ist. Danach wird für diese Bewertung ein minimal aufspannender Baum bestimmt. Dieser ist ein maximal aufspannender Baum für die ursprüngliche Bewertung.
19. Es sei B ein maximal aufspannender Baum von G und e, f Ecken in G . Ferner sei W ein Weg von e nach f in G mit maximalem Querschnitt und \bar{W} der Weg in B von e nach f . Im folgenden wird gezeigt, daß die Querschnitte von W und \bar{W} übereinstimmen. Es sei $k = (k_a, k_b)$ eine Kante von W , welche nicht in B liegt. Fügt man k in B ein, so entsteht ein geschlossener Weg W' . Da B ein maximal aufspannender Baum ist, gilt $\text{Bewertung}(k') \geq \text{Bewertung}(k)$ für alle Kanten k' von W' . Somit ist der Querschnitt des Weges von k_a nach k_b in B mindestens so groß wie die Bewertung von k . Hieraus ergibt sich, daß der Querschnitt von \bar{W} mindestens so groß ist wie der von W . Da der Querschnitt von W maximal ist, stimmen beide überein. Um den Durchsatz aller Paare zu bestimmen, wird zunächst ein maximal aufspannender Baum B bestimmt. Von jeder Ecke e von G wird eine Tiefensuche in B gestartet und dabei der Durchsatz für alle $n - 1$ Paare e, f bestimmt. Dabei wird mit Hilfe eines Stapels die maximale Bewertung auf dem aktuellen Weg gespeichert. Der Aufwand hierfür ist $O(n)$. Zusätzlich zur Bestimmung eines maximal aufspannenden Baums erfordert das Verfahren einen Aufwand von $O(n^2)$.
20. Für eine Kante k aus einem aufspannenden Baum T besteht der Graph $T \setminus \{k\}$ aus zwei Zusammenhangskomponenten mit Eckenmengen E_1, E_2 . Setze

$$K_k(T) = \{(a, b) \text{ Kante von } G \mid a \in E_1, b \in E_2\}.$$

Es gilt $K_k(T) \cap T = \{k\}$. Es sei T' ein minimal aufspannender Baum von G und T der vom Algorithmus erzeugte Baum. Ferner sei $k \in T \setminus T'$ und W der geschlossene Weg in $T' \cup \{k\}$. Für alle $l \in K_k(T) \cap W$ mit $l \neq k$ gilt $l \in T', l \notin T$ und $k \in K_l(T')$. Da l nicht in T liegt, wurde l durch den Algorithmus entfernt. D.h. l war die Kante mit der höchsten Bewertung in einem geschlossenen Weg. Somit ist die Bewertung von l mindestens so groß wie die Bewertung jeder Kante $k' \neq l$ aus dem geschlossenen Weg in $T \cup \{l\}$. Da $l \in K_k(T)$ ist, muß k auf diesem Weg liegen. Somit gilt $\text{Bewertung}(l) \geq \text{Bewertung}(k)$. Da T' ein minimal aufspannender Baum ist, kann nicht $\text{Bewertung}(l) > \text{Bewertung}(k)$ gelten. Somit stimmen die Bewertungen von l und k überein. Ersetzt man in T' die Kante l durch k , so erhält man einen neuen minimalen aufspannenden Baum T'' mit $|T'' \cap T| > |T' \cap T|$. Durch Wiederholung dieser Vorgehensweise zeigt man, daß auch T ein minimal aufspannender Baum von G ist.

21. Hat ein Graph eine *Brücke* (siehe Aufgabe 11 in Kapitel 2), deren Bewertung größer ist als die aller anderen Kanten, so liegt diese in jedem aufspannenden Baum.
22. Angenommen es gibt zwei verschiedene minimal aufspannende Bäume B_1 und B_2 . Es sei k_1 eine Kante aus B_1 , welche nicht in B_2 liegt. Fügt man k_1 in B_2 ein, so entsteht ein geschlossener Weg W . Da B_2 ein minimal aufspannender Baum ist und die Kanten verschiedene Bewertungen haben, sind die Bewertungen aller Kanten aus $W \cap B_2$ echt größer als die von k_1 . Da B_1 ein Baum ist, muß es in $W \setminus B_1$ eine Kante k_2 mit $\text{Bewertung}(k_1) < \text{Bewertung}(k_2)$ geben. Dieses Argument kann nun wiederholt werden. So entsteht eine unendliche Folge von Kanten k_1, k_2, \dots aus G mit echt aufsteigenden Gewichten. Dieser Widerspruch zeigt, daß es nur einen minimal aufspannenden Baum gibt.
23. Für kleine Werte von p ist der Algorithmus von Kruskal überlegen, während für Werte von p in der Nähe von 1 der Algorithmus von Prim schneller ist. Bei welchem Wert von p der Übergang erfolgt, hängt von der konkreten Implementierung der Algorithmen ab.
24. Die spezielle Eigenschaft der Bewertung kann ausgenutzt werden, um das Sortieren der Kanten nach ihren Bewertungen mit Aufwand $O(m + C)$ durchzuführen (Zeit und Speicher). Der Algorithmus von Kruskal hat in diesem Fall den Aufwand $O(m + C + \log n)$.
25. Der Beweis erfolgt durch vollständige Induktion nach der Anzahl n der Ecken. Für $n = 2$ ist die Aussage klar. Es sei $n > 1$ und k die Kante mit der kleinsten Bewertung und G' der Graph, welcher durch das Verschmelzen der Entdecken von k und das Ändern der Bewertungen entsteht. Nach Induktionsvoraussetzung bestimmt der Algorithmus einen minimal aufspannenden Baum B' von G' . Nach dem in Abschnitt 3.6 bewiesenen Satz gibt es einen minimal aufspannenden Baum B von G , welcher k enthält. Entfernt man k aus diesem Baum und ändert wie angegeben die Bewertungen der Kanten, so erhält man einen aufspannenden Baum für G' . Es gilt:

$$\text{Kosten}(B') \leq \text{Kosten}(B) - n \text{Bewertung}(k)$$

B' ist auch ein Baum in G . Fügt man die Kante k zu B' hinzu und ändert wieder die Bewertungen, so erhält man einen aufspannenden Baum von G mit Kosten $n\text{Bewertung}(k) + \text{Kosten}(B')$. Aus obiger Gleichung folgt, daß dieser Baum ein minimal aufspannender Baum von G ist. Dies ist gleichzeitig auch der Baum, den der Algorithmus konstruiert.

26. Es sei G' der Graph, welcher aus B , der neuen Ecke e und den neuen Kanten besteht. Nach Aufgabe 23 ist ein minimal aufspannender Baum von G' auch ein minimal aufspannender Baum von G . Wendet man den dort beschriebenen Algorithmus an, so müssen nur die neuen Kanten in B eingefügt und entsprechende Kanten entfernt werden. Dies kann mit Aufwand $O(ng(e))$ durchgeführt werden. Da der Graph G' $n-1+g(e) < 2n$ Kanten hat, bestimmt sowohl der Algorithmus von Kruskal als auch der von Prim in diesem Fall einen minimal aufspannenden Baum mit Aufwand $O(n \log n)$.

B.4 Kapitel 4

1. Im folgenden ist die Aufrufhierarchie der Prozedur Tiefensuche, die Numerierung der Ecken und der Tiefensuche-Wald angegeben.

a) tiefensuche(1)
 tiefensuche(2)
 tiefensuche(3)
 tiefensuche(4)
 tiefensuche(7)
 tiefensuche(6)
 tiefensuche(8)
 tiefensuche(5)

b) tiefensuche(1)
 tiefensuche(5)
 tiefensuche(3)
 tiefensuche(6)
 tiefensuche(9)
 tiefensuche(2)
 tiefensuche(4)
 tiefensuche(7)
 tiefensuche(8)

Ecke	1	2	3	4	5	6	7	8
Nr.	1	2	3	4	8	6	5	7

1 → 2 2 → 3,5
 3 → 4,7 4 → –
 5 → – 6 → –
 7 → 6,8 8 → –

Ecke	1	2	3	4	5	6	7	8	9
Nr.	1	6	3	7	2	4	8	9	5

1 → 5 2 → 4
 3 → 6 4 → 7,8
 5 → 3 6 → 9
 7 → – 8 → –
 9 → –

2. Im folgenden ist die Aufrufhierarchie der Prozedur Tiefensuche, die Numerierung der Ecken und der Tiefensuche-Wald angegeben.

a) tiefensuche(1)
 tiefensuche(2)
 tiefensuche(6)
 tiefensuche(4)
 tiefensuche(3)
 tiefensuche(5)

b) tiefensuche(1)
 tiefensuche(2)
 tiefensuche(3)
 tiefensuche(4)
 tiefensuche(5)
 tiefensuche(6)

Ecke	1	2	3	4	5	6
Nr.	1	2	5	4	6	3

1 → 2 2 → 6,3
 3 → – 4 → –
 5 → – 6 → 4

Ecke	1	2	3	4	5	6
Nr.	1	2	3	4	5	6

1 → 2 2 → 3
 3 → 4 4 → –
 5 → 6 6 → –

3. a) Es genügt, den Fall $\text{TSB}[e] < \text{TSB}[f]$ zu betrachten. Gilt $\text{TSE}[e] < \text{TSB}[f]$, so sind die Intervalle disjunkt. Andernfalls erfolgt der Aufruf von **tiefensuche(f)** innerhalb des Aufrufs **tiefensuche(e)**. Somit muß $\text{TSE}[f] < \text{TSE}[e]$ gelten, d.h. es gilt die dritte Bedingung.
 b) Ist k eine Baum- oder Vorwärtskante, so erfolgt der Aufruf von **tiefensuche(f)** innerhalb des Aufrufs **tiefensuche(e)** und endet deshalb auch vor diesem, d.h.

$$\text{TSB}[e] < \text{TSB}[f] < \text{TSE}[f] < \text{TSE}[e].$$

Gilt umgekehrt diese Ungleichungskette, so erfolgte der Aufruf von **tiefensuche(f)** innerhalb von **tiefensuche(e)**. Erfolgte der Aufruf direkt, so ist k eine Baumkante und andernfalls eine Vorwärtskante.

Ist k eine Rückwärtskante, so erfolgt der Aufruf von **tiefensuche(e)** innerhalb **tiefensuche(f)** und endet deshalb auch vor diesem, d.h.

$$\text{TSB}[f] < \text{TSB}[e] < \text{TSE}[e] < \text{TSE}[f].$$

Gilt umgekehrt diese Ungleichungskette, so erfolgte der Aufruf von **tiefensuche(e)** innerhalb von **tiefensuche(f)**, d.h. f ist Vorgänger von e und k ist somit eine Rückwärtskante.

Die Kante k ist genau dann eine Querkante, wenn der Aufruf von **tiefensuche(f)** schon vor dem Aufruf **tiefensuche(e)** endet. D.h. k ist genau dann eine Querkante, wenn

$$\text{TSB}[f] < \text{TSE}[f] < \text{TSB}[e] < \text{TSE}[e]$$

gilt.

4. Die Aussage ist wahr. Innerhalb des Aufrufs **tiefensuche(e)** werden alle von e erreichbaren unbesuchten Ecken besucht. Wegen $\text{TSNummer}[e] < \text{TSNummer}[f]$, muß f im Tiefensuchebaum von e erreichbar sein.
5. Nein, für die Kante $(4, 3)$ ist die Bedingung einer topologischen Sortierung nicht erfüllt.
6. Die Aufrufhierarchie der Prozedur **tsprozedur** sieht wie folgt aus:

```
tsprozedur(1)
  tsprozedur(3)
    tsprozedur(4)
      tsprozedur(7)
    tsprozedur(5)
      tsprozedur(6)
  tsprozedur(2)
```

Ecke	1	2	3	4	5	6	7
Sortierungsnummer	1	2	3	6	4	5	7

7. Beide Graphen haben keine topologische Sortierung, da sie geschlossene Wege haben $((4,2,6,4)$ bzw. $(2,3,4,2)$).
8. Der erste Graph besitzt keine topologische Sortierung, da er den geschlossenen Weg $(1, 3, 4, 1)$ besitzt. Die beiden anderen Graphen besitzen eine topologische Sortierung, die Sortierungsnummern sind $(2, 1, 3, 4, 5, 6)$ und $(1, 3, 2, 4, 5, 6)$ (in der Reihenfolge aufsteigender Eckenzahlen).

```
a) topsort(1)
  tsprozedur(1)
  tsprozedur(2)
  tsprozedur(3)
  tsprozedur(4)
  exit()
```

```
b) topsort(1)
  tsprozedur(1)
  tsprozedur(3)
  tsprozedur(4)
  tsprozedur(5)
  tsprozedur(6)
  topsort(2)
  tsprozedur(2)
```

```
c) topsort(1)
  tsprozedur(1)
  tsprozedur(2)
  tsprozedur(4)
  tsprozedur(5)
  tsprozedur(6)
  tsprozedur(3)
```

9. Die Adjazenzmatrix hat auf und unterhalb der Diagonalen nur Nullen.

10. Nein, dies zeigt eine topologische Sortierung, die zwischen zwei Teilgraphen *hin-* und *herspringt*. Betrachten Sie folgendes Beispiel:

Adjazenzliste: $1 \rightarrow 2, 4 \quad 2 \rightarrow 3$
 Numerierung: **1:1** **2:2** **3:4** **4:3**

11.

```
function Höhe(i : Integer) : Integer;
var hmax : Integer;
begin
  hmax := 0;
  for jeden Nachfolger j von i do
    hmax := max(hmax, Höhe(j) + 1);
  Höhe := hmax;
end
```

Ist w die Wurzel des Baumes, so ist $\text{Höhe}(w)$ die Höhe des Wurzelbaumes.

12. $\{2, 4, 5, 6\}, \{1\}, \{3\}, \{7\}, \{8\}$

13. Es sei T ein Tiefesuchebaum von G' mit Wurzel w und e, f Ecken aus T . Als Ecke w besucht wurde, war e noch nicht besucht worden. Somit ist $\text{TSE}[w] > \text{TSE}[e]$. D.h. der Aufruf von **tiefensuche(e)** war vor dem Aufruf von **tiefensuche(w)** beendet. Da e in G' von w erreichbar ist, ist w in G von e erreichbar. Wäre e vor w in G betrachtet worden, so wäre $\text{TSE}[e] > \text{TSE}[w]$. Somit wurde w vor e in G betrachtet. Aus $\text{TSE}[w] > \text{TSE}[e]$ folgt nun, daß der Aufruf von **tiefensuche(e)** in G innerhalb des Aufrufs **tiefensuche(w)** erfolgte. Somit ist e in G von w erreichbar, d.h. e und w liegen auf einem geschlossenen Weg in G . Das gleiche gilt für f und w und somit auch für e und f . Hieraus folgt, daß die Ecken von T in einer starken Zusammenhangskomponente von G liegen.

Sind umgekehrt e und f Ecken von G , welche in einer starken Zusammenhangskomponente liegen, so liegen e und f in G und G' auf einem geschlossenen Weg. Somit müssen auch e und f im gleichen Tiefesuchebaum von G' liegen. Hieraus folgt, daß die Ecken einer starken Zusammenhangskomponente von G in einem Tiefesuchebaum von G' liegen.

14. Es wird eine topologische Sortierung des DAG's betrachtet. Die Ecke mit Sortierungsnummer 1 hat Eingrad 0 und die mit Sortierungsnummer n hat Ausgrad 0. Eine Ecke mit Ausgrad 0 findet man, indem man an einer beliebigen Ecke einen einfachen Weg startet. Da der Graph keine geschlossenen Wege hat, endet der Weg an einer Ecke mit Ausgrad 0. Ein Algorithmus zur Bestimmung einer topologischen Sortierung sucht wiederholt eine Ecke mit Ausgrad 0, entfernt diese und alle inzidenten Kanten aus G und numeriert die entfernten Ecken in absteigender Reihenfolge. Da die Bestimmung einer Ecke mit Ausgrad 0 den Aufwand $O(n)$ hat, ergibt sich ein Gesamtaufwand von $O(n^2)$. Der auf der Tiefensuche aufbauende Algorithmus arbeitet mit Aufwand $O(n + m)$.
15. Zwei Ecken i und j sind genau dann in einer starken Zusammenhangskomponenten, wenn sie auf einem geschlossenen Weg liegen. Dies ist genau dann der Fall,

wenn e_{ji} und e_{ij} beide ungleich 0 sind, bzw. wenn $e_{ji}e_{ij}$ ungleich 0 ist. Für den i -ten Diagonaleintrag d_{ii} von E^2 gilt:

$$d_{ii} = e_{1i}e_{i1} + \dots + e_{ji}e_{ij} + \dots + e_{ni}e_{in}$$

Ist $d_{ii} = 0$, so bildet Ecke e eine komplett Zusammenhangskomponente. Ist $d_{ii} \neq 0$, so muß auch $e_{ii} = 1$ sein. Ferner gibt es noch $d_{ii} - 1$ weitere Ecken j mit $e_{ji}e_{ij} \neq 0$. Hieraus folgt sofort die Aussage.

```

16.  var Besucht : array[0..max] of Boolean;
      Vorgänger : array[0..max] of Integer;
procedure geschlossenerWeg(G : Graph);
var
  i : Integer;
begin
  Initialisiere Besucht mit false und Vorgänger mit 0;
  for jede Ecke i do
    if Besucht[i] = false then
      tiefensuche(i);
  end

  procedure tiefensuche(i : Integer);
  begin
    Besucht[i] := true;
    for jeden Nachbar j von i do
      if Besucht[j] = false then begin
        Vorgänger[j] := i;
        tiefensuche(j);
      end
      else if not Vorgänger[j] = i then
        exit('Geschlossener Weg');
    end
  end
end

```

Am Ende des Algorithmus (entweder regulär oder durch `exit`) hat der Algorithmus einen Baum durchlaufen, d.h. die Anzahl der betrachteten Kanten ist kleiner als n . Somit ist die Laufzeit $O(n)$.

- 17. $\{0, 2, 5, 7, 13\}, \{1, 3, 12, 9, 10, 8\}, \{4, 11, 6, 14\}$
- 18. Es werden zwei Tiefensuchedurchgänge gestartet mit den Startecken a bzw. b . Hierbei wird das gleiche Feld `TSNummer` benutzt (im zweiten Durchgang wird es nicht mehr initialisiert). Jede nicht besuchte Ecke ist weder von a noch von b erreichbar und muß somit entfernt werden. Bei der Verwendung von Adjazenzlisten basierend auf Zeigern müssen nur die entsprechenden Listen komplett entfernt werden. Somit ist der Aufwand $O(n + m)$.

```

19.  var TSNummer, MinNummer, Vorgänger : array[1..max] of Integer;
      zähler : Integer;
      TrennendeEcken : array[1..max] of Boolean;
procedure trennendeEcken(G : Graph);
var
  i : Integer;
begin
  Initialisiere TSNummer und zähler mit 0;
  Initialisiere TrennendeEcken mit false;

```

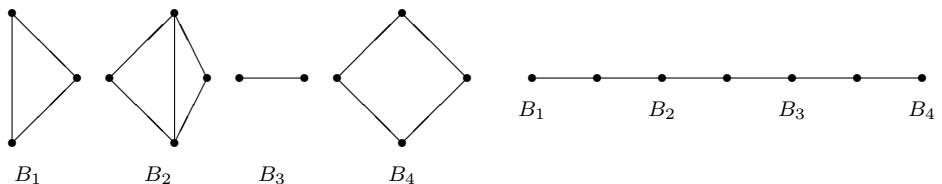
```

for jede Ecke i do
    if TSNummer[i] = 0 then begin
        blockproz(i);
        if i hat mehr als 2 Nachfolger then
            TrennendeEcken[i] := true;
        else
            TrennendeEcken[i] := false;
    end;
    for jede Ecke i do
        if TrennendeEcken[i] = true then
            ausgabe(i, ist trennende Ecke);
end

procedure blockproz(i : Integer);
var
    j : Integer;
begin
    zähler := zähler + 1;
    TSNummer[i] := MinNummer[i] := zähler;
    for jeden Nachbar j von i do begin
        if TSNummer[j] = 0 then begin
            Vorgänger[j] := i;
            blockproz(j);
            if MinNummer[j] >= TSNummer[i] then
                TrennendeEcken[i] := true;
            else
                MinNummer[i] := min(MinNummer[i], MinNummer[j]);
        end
        else
            if j ≠ Vorgänger[i] then
                MinNummer[i] := min(MinNummer[i], TSNummer[j]);
    end
end

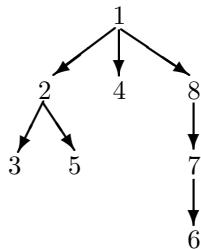
```

20. Links sind die Blöcke des Graphen und rechts ist der Blockgraph dargestellt.

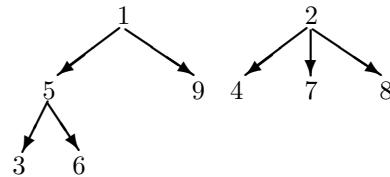


21. Der Beweis erfolgt durch vollständige Induktion nach b . Ist $b = 1$, so ist $t = 0$ und die Aussage ist wahr. Sei $b > 1$, B ein Block, welcher zu einem Blatt im Blockgraph gehört, und x die zugehörige Ecke. Entferne aus G alle Ecken in $B \setminus \{x\}$. Der neue Graph hat $b - 1$ Blöcke und mindestens $t - 1$ trennende Ecken. Die Aussage folgt nun aus der Induktionsvoraussetzung.
22. Ist der Graph zweifach zusammenhängend, so ist keine Ecke eine trennende Ecke. Andernfalls hat der Blockgraph mindestens zwei Blätter (jeder Baum mit mindestens zwei Ecken hat auch mindestens zwei Blätter). Ein Block, welcher zu einem Blatt im Blockgraph gehört, hat mindestens eine Ecke, welche nicht trennend ist.

23. a)



b)



24. Eine Kante liegt genau dann auf einem geschlossenen Weg, wenn sie zu einer starken Zusammenhangskomponente von G gehört. Mit dem in Abschnitt 4.5 beschriebenen Verfahren werden zunächst die starken Zusammenhangskomponenten bestimmt. Die gesuchte Kantenmenge besteht aus allen Kanten von G , welche in keiner starken Zusammenhangskomponente liegen.
25. Wendet man die Breitensuche n -mal auf den Graphen an, wobei jedesmal eine andere Startecke gewählt wird, so werden für jede Ecke die erreichbaren Ecken bestimmt. Der Aufwand ist $O(nm)$.
26. Es sei e eine beliebige Ecke und l_e die Länge des kürzesten Weges, der e enthält. Es wird eine Breitensuche mit Startecke e gestartet. Sei j die erste Ecke, welche die Breitensuche zum zweiten Mal besucht und i der aktuelle Vorgänger von j . Dann gilt

$$Niv(j) = Niv(i) \text{ oder } Niv(j) = Niv(i) + 1.$$

Im ersten Fall liegt ein geschlossener Weg der Länge $2Niv(i) + 1$ und im zweiten Fall der Länge $2Niv(i) + 2$ vor. Im ersten Fall gilt $l_e = 2Niv(i) + 1$ und im zweiten $l_e = 2Niv(i) + 2$ oder $l_e = 2Niv(i) + 1$. Um eine Unterscheidung zu treffen, müssen alle Ecken x mit $Niv(x) = Niv(i)$ abgearbeitet werden. Danach kann die Breitensuche beendet werden. Trifft man dabei auf eine weitere schon besuchte Ecke j mit $Niv(j) = Niv(i)$, so gilt $l_e = 2Niv(i) + 1$ und andernfalls $l_e = 2Niv(i) + 2$.

Ein Algorithmus zur Bestimmung der Länge eines kürzesten Weges startet von jeder Ecke e eine Breitensuche. Diese wird wie beschrieben benutzt, um die Länge des kürzesten geschlossenen Weges W mit $e \in W$ zu bestimmen. So erhält man die Länge des kürzesten geschlossenen Weges des Graphen mit Aufwand $O(nm)$.

27. Zunächst wird in linearer Zeit eine topologische Sortierung des Graphen bestimmt. Danach werden die Ecken in umgekehrter Reihenfolge ihrer topologischen Sortierungsnummern bearbeitet und die Menge der erreichbaren Ecken wie folgt bestimmt:

$$\text{Erreichbar}(i) := \bigcup_{j \text{ Nachfolger von } i} \text{Erreichbar}(j) \cup \{j\}$$

28. Wird die Tiefensuche auf einen Wurzelbaum angewandt, so sind Tiefensuche- und Breitensuchebaum identisch mit dem Graphen. D.h. für einen Wurzelbaum der Höhe h haben sowohl Tiefensuche- als auch Breitensuchebaum die Höhe h .

29. Zur Bestimmung der Zusammenhangskomponenten eines ungerichteten Graphen mit Hilfe der Breitensuche kann die Prozedur `zusammenhangskomponenten` aus Abbildung 4.16 verwendet werden. Dabei wird der Aufruf `zusammenhang(i)` durch den Aufruf `breitensuche(G, i)` ersetzt. Die Prozedur `breitensuche` aus Abbildung 4.29 muß dabei nur geringfügig geändert werden. An den beiden Stellen, an denen das Feld `niveau` geändert wird, muß die entsprechende Komponente von `ZKNummer` auf `zähler` gesetzt werden. D.h. die beiden Anweisungen `ZKNummer[startecke] = zähler` und `ZKNummer[j] = zähler` müssen eingefügt werden.
30.

```
var niveau : array[1..max] of Integer;
procedure breitensuche(G : Graph; var B : Graph;
                      startecke : Integer);
var
  i, j : Integer;
  W : warteschlange of Integer;
begin
  Initialisiere niveau mit -1;
  niveau[startecke] := 0;
  W.einfügen(startecke);
  while W ≠ ∅ do begin
    i := W.entfernen;
    for jeden Nachbar j von i do
      if niveau[j] = -1 then begin
        niveau[j] := niveau[i] + 1;
        W.einfügen(j);
        Füge Kante (i,j) in B ein;
      end
      else if niveau[j] = niveau[i] + 1 then
        Füge Kante (i,j) in B ein;
    end
  end
end
```
31. a) Der Beweis erfolgt durch vollständige Induktion nach n . Für $n = 1$ ist die Aussage wahr. Sei nun $n > 1$ und x eine Ecke von B mit $g^+(x) = 0$. Setze $G' = G \setminus \{x\}$ und $B' = B \setminus \{x\}$. Da B' die gleiche Eigenschaften wie B hat, folgt aus der Induktionsvoraussetzung, daß B' ein Tiefensuchebaum von G' ist. Es sei nun y der Vorgänger von x in B und (x, z) mit $z \neq y$ eine Kante in G . Nach Voraussetzung ist z Vorgänger von x in B , d.h. wenn die Tiefensuche bei Ecke y ankommt, sind schon alle Nachbarn von x besucht worden. Somit wird nach y die Ecke x besucht und sofort wieder zu y zurückgekehrt. Also ist B ein Tiefensuchebaum von G .
- b) Für eine beliebige Ecke x von G bezeichne $d_B(e, x)$ die Länge des Weges von e nach x in B . Mittels vollständiger Induktion nach $d(e, x)$ wird gezeigt, daß $d_B(e, x) = d(e, x)$ für alle Ecken x von G gilt. Ist $d(e, x) = 0$, so ist $x = e$ und die Aussage ist wahr. Sei nun $d(e, x) > 0$ und y der Vorgänger von x in G auf dem Weg von e nach x . Dann ist $d(e, y) < d(e, x)$ und somit ist $d_B(e, y) = d(e, y)$ nach Induktionsvoraussetzung. Also gilt: $d_B(e, x) \geq d(e, x) = d(e, y) + 1 = d_B(e, y) + 1$. Da (x, y) eine Kante in G ist, unterscheiden sich $d_B(e, x)$ und $d_B(e, y)$ maximal um 1. Hieraus folgt $d_B(e, x) = d(e, x)$.

Man beachte, daß B nicht notwendigerweise durch die in Abschnitt 4.10 beschriebene Realisierung der Breitensuche erzeugt werden kann.

32. Da B keine geschlossenen Wege enthält, können die Verfahren etwas vereinfacht werden. Wird die Tiefensuche wie in Abschnitt 4.2 beschrieben mit Hilfe eines Stacks realisiert, so ist der Speicheraufwand proportional zur maximalen Stapeltiefe. Da im ungünstigsten Fall bis zu den Blättern gesucht werden muß, ist der Speicheraufwand $O(C)$. Bei der iterativen Tiefensuche wird keine Ecke jenseits von Tiefe T betrachtet. Somit ist der Speicheraufwand $O(T)$. Der Speicheraufwand der Breitensuche ist proportional zur Länge der Warteschlange. Im ungünstigsten Fall wird die gesuchte Ecke als letzte Ecke auf Tiefe T betrachtet. Da in diesem Fall alle Ecken der Tiefe T in der Warteschlange sind, ist der Speicheraufwand $O(b^T)$.

Die Laufzeit der drei Suchverfahren ist proportional zur Anzahl der betrachteten Ecken. Für die Tiefensuche sind dies $(b^{C+1} - 1)/(b - 1)$ und für die Breitensuche $(b^{T+1} - 1)/(b - 1)$ Ecken. Bei der iterativen Tiefensuche werden etwa $b^{T+2}/(b - 1)^2$ Ecken betrachtet (vergleichen Sie Abschnitt 4.11).

33. Ist (e, f) eine Rückwärtskante eines ungerichteten Graphen, so ist entweder e ein Vorgänger oder ein Nachfolger von f im Tiefensuchewald, d.h. Rückwärtskanten verbinden niemals Blätter des Tiefensuchebaums. Da auch Vorwärtskanten keine Blätter verbinden, ist die Aussage bewiesen.
34. Der Algorithmus ist eine Variante der Tiefensuche angewendet auf die Ausgangs-
ecke des inversen Graph des Schaltkreises (d.h. die Richtung jeder Kante des
Graphen wird umgedreht).

```

function schaltkreis(e : Ecke) : Boolean;
begin
  switch(e.type) begin
    case Eingabeecke:
      schaltkreis := Variable_< i>e;
    case Konjunktion:
      schaltkreis := schaltkreis(Nachbar_1(e)) AND
                    schaltkreis(Nachbar_2(e));
    case Disjunktion:
      schaltkreis := schaltkreis(Nachbar_1(e)) OR
                    schaltkreis(Nachbar_2(e));
    case Negation:
      schaltkreis := NOT schaltkreis(Nachbar(e));
    case Ausgabeecke:
      schaltkreis := schaltkreis(Nachbar(e));
  end
end

```

B.5 Kapitel 5

1. Die chromatische Zahl der vier Graphen von links nach rechts und oben nach unten ist 2, 3, 4 und 3.
2. Es gilt $\chi(L_{z,s}) = 2$ (man färbe die Ecken wie ein Schachbrett) und $\omega(L_{z,s}) = 2$.
3. Man unterteile die Ecken von H_d in zwei Teilmengen, je nachdem ob die Anzahl der Einsen gerade oder ungerade ist. Da die beiden Mengen unabhängige Mengen sind, ist $\chi(H_d) = 2$. Da H_d keine Dreiecke enthält, ist $\omega(H_d) = 2$.
4. Es sei B ein beliebiger Tiefensuchebaum von G der Höhe h . Es gilt $h \leq m$. Die Ecken von G werden derart gefärbt, so daß die Ecken auf einem Niveau jeweils die gleiche Farbe bekommen. Dazu werden $h+1$ Farben verwendet. Da nach den Ergebnissen aus Abschnitt 4.7 eine Kante niemals Ecken aus dem gleichen Niveau verbindet, liegt eine zulässige Färbung vor. Somit gilt $\chi(G) \leq m+1$.
5. Wendet man die Breitensuche auf einen Baum an, so wird jede Kante zu einer Baumkante, d.h. für jede Kante $k = (e, f)$ gilt: $Niv(e) + Niv(f) \equiv 1(2)$. Die Aussage folgt nun aus dem letzten Lemma in Abschnitt 4.10.
6. Man betrachte den in Abschnitt 4.9 definierten Blockgraph G_B und erzeuge für einen beliebigen Block B eine minimale Färbung. Danach wird G_B anhand der Tiefensuche mit Startecke B durchlaufen. Da G_B ein Baum ist, ist in jedem neuen Block genau eine Ecke schon gefärbt. Diese wird jeweils zu einer minimalen Färbung erweitert. Hieraus folgt die Behauptung. Färbungsalgorithmen können diese Eigenschaft nutzen und zunächst in linearer Zeit die Blöcke bestimmen und danach wie beschrieben weiter verfahren. Der Vorteil liegt darin, daß gegebenenfalls kleinere Graphen gefärbt werden müssen.
7. Sei U' ein induzierter Untergraph von G , so daß $\delta(U') \geq \delta(U)$ für alle induzierten Untergraphen U von G ist. Zum Beweis wird eine Reihenfolge der Ecken bestimmt, für die der Greedy-Algorithmus maximal $\delta(U') + 1$ Farben vergibt. Es sei e_n eine Ecke von minimalem Eckengrad in G . Sind e_n, \dots, e_{i+1} schon bestimmt, so wähle man aus $G \setminus \{e_n, \dots, e_{i+1}\}$ eine Ecke e_i mit minimalem Eckengrad. Für $i = 1, \dots, n$ sei G_i der von $\{e_1, \dots, e_i\}$ induzierte Untergraph. Dann gilt $\delta(G_i) = d_{G_i}(e_i)$. Wird der Greedy-Algorithmus auf G angewandt, wobei die Ecken in der Reihenfolge e_1, e_2, \dots betrachtet werden, so hat die jeweils aktuelle Ecke e_i schon $g_{G_i}(e_i)$ gefärbte Nachbarn. Insgesamt vergibt der Greedy-Algorithmus somit maximal $1 + \max\{g_{G_i}(e_i) \mid i = 1, \dots, n\}$ Farben. Sei j minimal, so daß $U' \subseteq G_j$ gilt. Da $U' \not\subseteq G_{j-1}$ ist, liegt e_j in U' . Nun gilt:

$$\begin{aligned} \delta(U') &\leq g_{U'}(e_j) \\ &\leq g_{G_j}(e_j) \\ &\leq \max\{g_{G_i}(e_i) \mid i = 1, \dots, n\} \\ &\leq \max\{\delta(G_i) \mid i = 1, \dots, n\} \\ &\leq \delta(U') \end{aligned}$$

Hieraus folgt die Aussage.

8. Es sei A die Menge der Ecken von G , deren Grad mindestens $\chi(G) - 1$ ist. Angenommen es gilt $|A| \leq \chi(G) - 1$. Dann färbe man die Ecken von G mit dem Greedy-Algorithmus, wobei zuerst die Ecken aus A betrachtet werden. Hierfür werden maximal $\chi(G) - 1$ Farben verwendet. Da die restlichen Ecken maximal den Eckengrad $\chi(G) - 2$ haben, vergibt der Greedy-Algorithmus insgesamt höchstens $\chi(G) - 1$ Farben. Dieser Widerspruch beweist $|A| \geq \chi(G)$.
9. Es sei e eine beliebige Ecke von G . Dann ist nach Voraussetzung $N(e)$ eine unabhängige Menge von G . Also gilt $\alpha(G) \geq \Delta(G)$. Somit folgt

$$(\alpha(G) + 1)^2 > \alpha(G)(\alpha(G) + 1) \geq \alpha(G)(\Delta(G) + 1) \geq \alpha(G)\chi(G) \geq n.$$

Dies beweist die Behauptung.

10. a) Für jede Ecke e von G gilt $\chi(G \setminus \{e\}) \geq \chi(G) - 1$. Da nach Voraussetzung $\chi(G \setminus \{e\}) < \chi(G)$ für jede Ecke gilt, muß $\chi(G \setminus \{e\}) = \chi(G) - 1$ sein.
- b) Ist G ein 2-kritischer Graph, so hat $G \setminus \{e\}$ keine Kanten. Daraus folgt $G = C_2$. Ein 3-kritischer Graph G ist nicht bipartit und enthält somit einen geschlossenen Weg W ungerader Länge. Alle Ecken von G müssen auf W liegen. Gäbe es Kanten, welche nicht auf W liegen, so würde es eine Ecke e geben, so daß $G \setminus \{e\}$ immer noch einen geschlossenen Weg ungerader Länge enthält. Da $\chi(G \setminus \{e\}) = 2$ ist, kann dies nicht sein. Somit ist G vom Typ C_{2i+1} mit $i \geq 1$.
- c) Es sei G ein kritischer Graph und B_i die Blöcke von G . Nach Aufgabe 6 gilt:

$$\chi(G) = \max \{\chi(B_i) \mid i = 1, \dots, s\}.$$

Angenommen es ist $s > 1$. Ohne Einschränkung kann man annehmen, daß $\chi(G) = \chi(B_1)$ ist. Dann gibt es eine Ecke e in $B_2 \setminus B_1$. Somit ist $\chi(G \setminus \{e\}) = \chi(B_1) = \chi(G)$. Dieser Widerspruch zeigt, daß $s = 1$ gilt, d.h. G ist zweifach zusammenhängend.

11. Da es zwischen den Ecken einer unabhängigen Menge keine Kanten gibt, gilt

$$m \leq n(n-1)/2 - \alpha(G)(\alpha(G)-1)/2$$

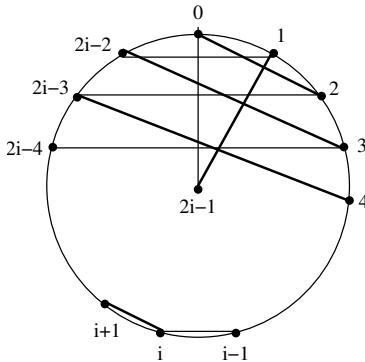
bzw. $2m \leq n^2 - \alpha(G)^2$. Hieraus folgt die Behauptung. Gilt $\alpha(G) = \sqrt{n^2 - 2m}$, so muß $\alpha(G) = n$ sein, d.h. G ist der leere Graph.

12. Für eine minimale Färbung muß es zu jedem Paar von Farben eine Kante geben, welche Ecken mit dieser Farbe verbindet. Somit gilt: $\chi(G)(\chi(G)-1)/2 \leq m$ bzw. $(\chi(G)-1/2)^2 \leq 2m + 1/4$. Hieraus folgt die Behauptung.
13. Die Intervalle, welche zu einer Farbklasse einer Färbung des Schnittgraphen gehören, überlappen sich paarweise nicht. Somit kann ein Arbeiter diese Arbeitsaufträge ausführen. Umgekehrt induziert eine Zuordnung von Arbeitern zu den Aufträgen eine Färbung von G . Somit ist $\chi(G)$ die minimale Anzahl von Arbeitern zur Ausführung der Aufträge. Auf der Menge der Intervalle wird eine

partielle Ordnung definiert: $T_i < T_j$, falls $b_i < a_j$. Ohne Einschränkung der Allgemeinheit kann man annehmen, daß die Intervalle in der Reihenfolge T_1, T_2, \dots gefärbt werden. Es seien $\{T_1, \dots, T_s\}$ die mit Farbe 1 gefärbten Intervalle und G' der von den restlichen Intervallen induzierte Graph. Im folgenden wird gezeigt, daß $\omega(G') < \omega(G)$ gilt. Daraus kann gefolgert werden, daß der Algorithmus maximal $\omega(G)$ Farben vergibt. Wegen $\omega(G) \leq \chi(G)$ bestimmt der Algorithmus somit eine minimale Färbung. Angenommen es gilt $\omega(G') = \omega(G)$. Dann gibt es in G' eine Clique C mit $\omega(G)$ Intervallen. Es sei I_C der Schnitt aller Intervalle aus C . Es ist $I_C \neq \emptyset$. Angenommen es gilt $T_i \cap I_C = \emptyset$ für alle $i = 1, \dots, s$. Sei j maximal mit $T_j < I_C$ (man beachte, daß $I_C < T_1$ nicht gelten kann). Somit gibt es in C ein Intervall I mit $T_j < I$ und es ist $I_C < T_{j+1}$. Dies steht im Widerspruch zur Wahl von T_j . Also gibt es ein Intervall $I \in \{T_1, \dots, T_s\}$ mit $I \cap I_C \neq \emptyset$. Da C eine maximale Clique ist, muß $I \in C$ sein. Somit liegt C nicht in G' . Dieser Widerspruch zeigt $\omega(G') < \omega(G)$.

14. $\chi(G) = 4$ und $\omega(G) = 2$ (Vergleichen Sie Aufgabe 27 für einen Beweis).
15. Die kantenchromatische Zahl der vier Graphen von links nach rechts und oben nach unten ist 3, 3, 3 und 4.
16. Da alle zu einer Ecke inzidenten Kanten verschiedene Farben haben müssen, gilt $\chi'(G) \geq \Delta(G)$.
17. Es sei Δ der maximale Eckengrad des bipartiten Graphen G . Die Aussage wird durch vollständige Induktion nach m bewiesen. Für $m = 1$ ist die Aussage wahr. Sei nun $m > 1$ und $k = (i, j)$ eine beliebige Kante von G . Nach Induktionsvoraussetzung können die Kanten des Graphen $G' = G \setminus \{k\}$ mit $\Delta(G')$ Farben gefärbt werden. Ist $\Delta(G') < \Delta$ oder gibt es eine Farbe, mit der keine der zu i und j inzidenten Kanten gefärbt ist, so ist die Aussage bewiesen. Andernfalls erreicht man durch Vertauschung von Farben, daß auch die Kante k mit einer schon verwendeten Farbe gefärbt werden kann. Zu jeder verwendeten Farbe f gibt es eine mit f gefärbte Kante, welche zu i oder j inzident ist. Da jedoch i und j maximal zu jeweils $\Delta - 1$ Kanten inzident sind, muß es Farben f_1 und f_2 geben, so daß eine Kante der Farbe f_1 zu i , aber keine Kante dieser Farbe zu j inzident ist. Ferner muß eine Kante der Farbe f_2 zu j , aber keine Kante dieser Farbe zu i inzident sein. Es sei $G(i, j)$ der von den mit f_1 und f_2 gefärbten Kanten gebildete Untergraph von G . Da die Färbung zulässig ist, haben die Ecken in $G(i, j)$ den Grad 1 oder 2, wobei i und j den Grad 1 haben. Die Zusammenhangskomponenten von $G(i, j)$ sind offene oder geschlossene Wege, deren Kanten abwechselnd mit f_1 und f_2 gefärbt sind. Es sei W ein Pfad, welcher mit der Ecke i anfängt. Da G bipartit ist, haben alle geschlossenen Wege gerade Länge. Somit hat W ungerade Länge (zusammen mit k entsteht ein geschlossener Weg). Also hat die letzte Kante von W die Farbe f_1 , d.h. j liegt nicht auf W . Vertauscht man die Farben f_1 und f_2 der Kanten in W , so entsteht wieder eine zulässige Färbung. Nun kann aber die Kante k mit der Farbe f_1 gefärbt werden.
18. Das Problem kann als Kantenfärbungsproblem formuliert werden. Jede Mannschaft entspricht einer Ecke in einem vollständigen Graphen. Die Anzahl der Far-

ben in einer minimalen Kantenfärbung entspricht der Anzahl der Tage des Wettbewerbes. Die Kanten mit gleicher Farbe entsprechen den Partien eines Spieltages. Um die minimale Dauer des Wettbewerbes zu bestimmen, muß $\chi'(K_n)$ bestimmt werden.



Zunächst wird gezeigt, daß $\chi'(K_{2i}) = 2i - 1$ ist. Dazu werden die Ecken von K_{2i} mit den Zahlen $\{0, \dots, 2i - 1\}$ nummeriert und eine graphische Darstellung von K_{2i} betrachtet, bei der die Ecken $\{0, \dots, 2i - 2\}$ gleichmäßig auf einem Kreis angeordnet sind und die Ecke $2i - 1$ im Mittelpunkt liegt. Folgende Kanten können mit Farbe 1 gefärbt werden: $(2i-1, 0), (1, 2i-2), (2, 2i-3), \dots, (i-1, i)$. In der graphischen Darstellung sieht man, daß alle Kanten bis auf die erste parallel sind und somit keine gemeinsame Ecke haben. Eine neue Farbklass hat man, indem man zu jeder Ecke jeder Kante, ausgenommen der ersten Ecke der ersten Kante, 1 modulo $2i - 1$ addiert. Die neue Farbklass besteht aus den Kanten

$$(2i-1, 1), (2, 0), (3, 2i-2), \dots, (i, i+1)$$

und ist durch fette Kanten dargestellt. Man erhält diese Kanten, indem man die Kanten der ersten Farbklass um den Mittelpunkt um den Winkel $2\pi/(2i-1)$ dreht. Auf diese Art erhält man $2i-1$ Farbklassen mit je i Kanten. Da alle Kanten gefärbt sind und alle Farbklassen die maximale Anzahl von Kanten enthalten, ist $\chi'(K_{2i}) = 2i - 1$.

Es bleibt noch $\chi'(K_{2i+1})$ zu bestimmen. Eine Farbklass in K_{2i+1} kann maximal i Kanten enthalten. Somit ist $\chi'(K_{2i+1}) \geq 2i + 1$. Da K_{2i+1} ein Untergraph von K_{2i+2} ist, induziert eine Kantenfärbung von K_{2i+2} mit $2i + 1$ Farben eine Kantenfärbung mit $2i + 1$ Farben auf K_{2i+1} . Somit ist $\chi'(K_{2i+1}) = 2i + 1$.

19. Für $n \geq 3$ gilt $\chi(G_n) = 4$. Der Aufruf `faerbung` $(G_n, 3)$ liefert **false** zurück. Jede der Ecken $1, \dots, n-1$ hat im Suchbaum mindestens den Eckengrad 2. Somit hat der Suchbaum $O(2^n)$ Ecken.
20. Die Änderungen an der Prozedur `faerbung` verbessern die absolute Laufzeit, so daß kleine Graphen in akzeptabler Zeit gefärbt werden können. Die Laufzeit wächst jedoch weiterhin exponentiell.
21. Auf dem ersten Niveau hat der Suchbaum $n-1$ Ecken und jede dieser Ecken hat wieder $n-1$ Nachfolger. Jeweils eine dieser Ecken hat keinen Nachfolger mehr. Somit sind auf dem dritten Niveau $(n-1)^2(n-2)$ Ecken. Dabei haben jeweils zwei Nachfolger einer Ecke auf Niveau drei keine Nachfolger etc. Somit sind auf dem vierten Niveau $(n-1)^2(n-2)(n-3)$ Ecken. Wiederholt man dieses Argument, so erhält man für die Gesamtanzahl der Ecken im Suchbaum:

$$(n-1) \sum_{s=0}^{n-1} \prod_{j=1}^s (n-j) = (n-1)(n-1)! \sum_{i=1}^n \frac{1}{(n-i)!}$$

Da die letzte Summe gegen e konvergiert, ist die Anzahl der Ecken im Suchbaum annährend $e(n-1)(n-1)!$.

22. Der Algorithmus verwaltet zwei Variablen **unten** und **oben**. Zu jedem Zeitpunkt gibt es eine Färbung mit **oben** Farben, aber keine mit **unten** Farben. Aus diesem Grund wird vorausgesetzt, daß der Graph nicht leer ist. In jedem Schritt wird das Intervall, in dem die chromatische Zahl liegt, halbiert.

```

function chromatischeZahl(Graph: G) : Integer;
var unten, oben, mitte : Integer;
begin
    unten := 1; oben := n;
    while unten < (oben - 1) do begin
        mitte := (oben - unten)/2;
        if färbung(G,mitte) then
            oben := mitte
        else
            unten := mitte;
    end;
    chromatischeZahl = oben;
end

```

23. a) Es sei I eine maximale unabhängige Menge von G . Ist $e \notin I$, so ist I eine maximale unabhängige Menge von $G \setminus \{e\}$. Ist $e \in I$, so ist $I \setminus \{e\}$ eine unabhängige Menge von $G \setminus (\{e\} \cup N(e))$. Wäre $I \setminus \{e\}$ keine maximale unabhängige Menge von $G \setminus (\{e\} \cup N(e))$, dann gäbe es eine unabhängige Menge I' von $G \setminus (\{e\} \cup N(e))$ mit $|I'| > |I| - 1$. Somit wäre auch $I' \cup \{e\}$ eine unabhängige Menge mit mehr als $|I| + 1$ Ecken. Hieraus folgt die Behauptung.
b) Es genügt, eine maximale unabhängige Menge für jede Zusammenhangskomponente zu bestimmen. Ist der maximale Eckengrad von G kleiner oder gleich 2, so sind die Zusammenhangskomponenten von G isolierte Ecken, Pfade oder geschlossene Wege. In jedem dieser Fälle läßt sich eine maximale unabhängige Menge in linearer Zeit bestimmen.
c) Die Korrektheit folgt direkt aus Teil a).
d) Bezeichne mit $s(n)$ die Anzahl der Schritte, die die Funktion **unabhängig** für Graphen mit n Ecken benötigt. Dann gilt für $n > 4$ folgende Beziehung:

$$s(n) \leq s(n-1) + s(n-4) + Cn \quad (C > 0)$$

Schätzt man $s(n)$ durch λa^n mit $a > 1$ ab und wählt a , so daß

$$a^n \approx a^{n-1} + a^{n-4} + o(1)$$

erfüllt ist, dann ist folgende Ungleichung zu lösen:

$$a^4 \geq a^3 + 1 + \epsilon$$

Der Wert $a \approx 1.39$ erfüllt diese Ungleichung. Somit ist $O(1.39^n)$ der Aufwand der Funktion **unabhängig**.

24. Zunächst wird die erste Ungleichung betrachtet. Es seien f_1 bzw. f_2 minimale Färbungen von G bzw. \overline{G} . Ordne jeder Ecke e aus G das Paar $(f_1(e), f_2(e))$ zu. Angenommen es gibt zwei Ecken $a \neq b$, welche das gleiche Paar zugeordnet bekommen. Dann ist $f_1(a) = f_1(b)$ und $f_2(a) = f_2(b)$. Somit können a und b weder in G noch in \overline{G} benachbart sein. Dieser Widerspruch beweist, daß $n \leq \chi(G)\chi(\overline{G})$ gilt. Hieraus folgt

$$4n \leq 4\chi(G)\chi(\overline{G}) \leq (\chi(G) + \chi(\overline{G}))^2$$

und somit $2\sqrt{n} \leq \chi(G)\chi(\overline{G})$. Für C_4 gilt $\chi(C_4) + \chi(\overline{C_4}) = 4 = 2\sqrt{4}$ und für $K_{s,s}$ mit $s > 2$ gilt $\chi(K_{s,s}) + \chi(\overline{K_{s,s}}) = 2 + s > 2\sqrt{2s} = 2\sqrt{n}$.

Der Beweis der zweiten Ungleichung wird durch vollständige Induktion nach n geführt. Für $n = 2$ ist die Aussage wahr. Sei nun $n > 2$, e eine Ecke von G und $G_e = G \setminus \{e\}$. Dann gilt $\chi(G_e) + 1 \geq \chi(G)$ und $\chi(\overline{G_e}) + 1 \geq \chi(\overline{G})$. Nach Induktionsvoraussetzung ist $\chi(G_e) + \chi(\overline{G_e}) \leq n$. Gilt sogar $\chi(G_e) + \chi(\overline{G_e}) < n$, so folgt die Aussage direkt. Somit bleibt noch der Fall $\chi(G_e) + \chi(\overline{G_e}) = n$. Ist $g(e) < \chi(G_e)$, so folgt $\chi(G) = \chi(G_e)$ und hieraus ergibt sich $\chi(G) + \chi(\overline{G}) \leq n+1$. Ist $g(e) \geq \chi(G_e)$, so gilt

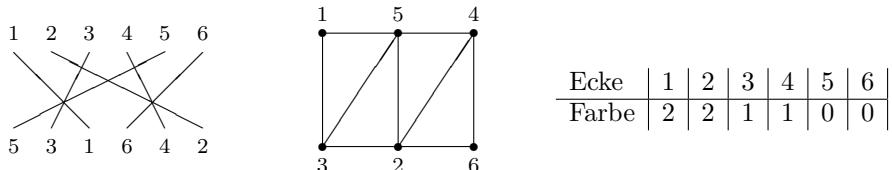
$$\bar{g}(e) = n - 1 - g(e) \leq n - 1 - \chi(G_e) = n - 1 - (n - \chi(\overline{G_e})) = \chi(\overline{G_e}) - 1$$

und somit $\bar{g}(e) < \chi(\overline{G_e})$. Also ist $\chi(\overline{G}) = \chi(\overline{G_e})$, woraus die Aussage folgt.

Für die Graphen $K_{s,s}$ gilt $\chi(K_{s,s}) + \chi(\overline{K_{s,s}}) = 2 + s < 2s + 1 = n + 1$ und für die Graphen K_n gilt $\chi(K_n) + \chi(\overline{K_n}) = n + 1$.

25. Es gilt $\chi(R_n) = 1 + \chi(C_n)$ für alle $n \geq 3$. Somit ist $\chi(R_{2n}) = 3$ und $\chi(R_{2n+1}) = 4$.
26. Da I_5 vollständig ist, gilt $\chi(I_5) = 5$. Für $n \geq 6$ gilt nach dem Satz von Brooks $\chi(I_n) \leq 4$. Da I_n einen geschlossenen Weg der Länge drei enthält, ist $\chi(I_n) \geq 3$. Ist $n \equiv 0(3)$, so können die Ecken im Uhrzeigersinn mit den Farben 1, 2, 3, 1, 2, 3, ... gefärbt werden. Somit ist $\chi(I_{3n}) = 3$ für $n \geq 2$. Ist $n \not\equiv 0(3)$, so legen die Farben von zwei auf C_n benachbarten Ecken eine Färbung fest. In diesem Fall kommt man nicht mit drei Farben aus, d.h. die chromatische Zahl ist 4.
27. Der Beweis wird durch vollständige Induktion nach n geführt. Für $n = 3$ ist die Aussage wahr. Sei nun $n > 3$. Es ist zu zeigen, daß G_n keinen geschlossen Weg der Länge drei enthält. Dazu beachte man zunächst, daß der von den neuen Ecken induzierte Untergraph diese Eigenschaft hat. Nach Induktionsvoraussetzung und Konstruktion bilden die Nachbarn der Ecken von G_{n-1} in G_n jeweils eine unabhängige Menge. Somit ist $\omega(G_n) = 2$. Eine minimale Färbung von G_{n-1} kann zu einer Färbung von G_n erweitert werden, indem die Ecke e' die gleiche Farbe wie e und die zusätzliche Ecke f eine neue Farbe bekommt. Somit gilt $\chi(G_n) \leq n$ nach Induktionsvoraussetzung. Angenommen es gibt eine Färbung von G_n , welche nur $n - 1$ Farben verwendet. Da Ecke f zu jeder neuen Ecke benachbart ist, ist keine dieser Ecken mit der gleichen Farbe gefärbt. Nun kann für den Untergraphen G_{n-1} von G_n eine Färbung mit $n - 2$ Farben erzeugt werden. Dazu bekommt jede Ecke e die Farbe von e' . Dies widerspricht der Induktionsvoraussetzung und somit ist $\chi(G_n) = n$.

28. Für die Graphen I_n gilt: $\chi'(I_{2n+1}) = 5$ und $\chi'(I_{2n}) = 4$. Für die Graphen R_n gilt: $\chi'(R_n) = n$.
29. Es werden maximal zwei weitere Farben benötigt. Man wende die Breitensuche auf B an und färbe die noch nicht gefärbten Ecken in ungeraden Niveaus mit der Farbe 2 und die in geraden Niveaus mit der Farbe 3.
30. Es wird ein Graph gebildet, in dem jede Radiostation einer Ecke entspricht und zwei Ecken verbunden sind, wenn die Entfernung der zugehörigen Stationen unter dem vorgegebenen Wert liegt. Radiostationen, welche in einer festen minimalen Färbung dieses Graphen die gleiche Farbe haben, können die gleiche Sendefrequenz benutzen.
31. Nein. Die Prozedur stützt sich bei der Färbung einer Ecke e mit $g(e) < 5$ wesentlich auf die Planarität des Graphen. Der Graph K_6 erfüllt die angegebene Bedingung, aber es gilt $\chi(K_6) > 5$.
32. $G^\pi = G_{\pi^{-1}}$.
33. Enthält jede Clique von G weniger als \sqrt{n} Ecken, so folgt aus dem letzten Lemma aus Abschnitt 5.5, daß $\chi(G) < \sqrt{n}$ ist. Somit gilt $n/\chi(G) > \sqrt{n}$. Aus dem gleichen Lemma folgt, daß G eine unabhängige Menge der Größe \sqrt{n} hat. Somit hat \bar{G} eine Clique der Größe \sqrt{n} .
34. Orientiert man jede Kante des Permutationsgraphen in Richtung der höheren Eckennummern, so entsteht eine kreisfreie transitive Orientierung. Die chromatische Zahl des Permutationsgraphen ist $h + 1$, wobei h die maximale Höhe einer Ecke ist. Aus Aufgabe 42 folgt nun, daß der Greedy-Algorithmus für Permutationsgraphen eine minimale Färbung erzeugt.
35. Definiere eine Permutation ρ durch $\rho(i) = \pi(n + 1 - i)$ für $i = 1, \dots, n$. Es gilt $\rho^{-1}(i) = n + 1 - \pi^{-1}(i)$. Ferner ist $\pi^{-1}(i) < \pi^{-1}(j)$ genau dann, wenn $\rho^{-1}(i) > \rho^{-1}(j)$ ist. Somit ist G_ρ das Komplement von G_π .
36. Im folgenden ist das Permutationsdiagramm, der Graph und die minimale Färbung dargestellt.



37. Eine Ecke j ist zu $\pi(1)$ benachbart, wenn π^{-1} die Reihenfolge von j und $\pi(1)$ vertauscht. Für $j \neq \pi(1)$ gilt $\pi^{-1}(j) > \pi^{-1}(\pi(1)) = 1$. Somit sind die Ecken $j = 1, \dots, \pi(1)-1$ zu $\pi(1)$ benachbart und es gilt $g(\pi(1)) = \pi(1)-1$. Eine Ecke j ist zu $\pi(n)$ benachbart, wenn π^{-1} die Reihenfolge von j und $\pi(n)$ vertauscht. Für $j \neq \pi(n)$ gilt $\pi^{-1}(j) < \pi^{-1}(\pi(n)) = n$. Somit sind die Ecken $j = n, \dots, \pi(1)+1$ zu $\pi(n)$ benachbart und es gilt $g(\pi(n)) = n - \pi(n)$.

38. Es sei $I = \{I_1, \dots, I_n\}$ eine Menge von Intervallen der Form $I_i = [a_i, b_i]$ und G der zugehörige Intervallgraph. Auf der Menge I kann eine partielle Ordnung eingeführt werden: $I_i < I_j$ genau dann, wenn $I_i \cap I_j = \emptyset$ und $b_i < a_j$ gilt. Das Komplement \overline{G} hat die Kantenmenge $K = \{(I_i, I_j) \mid I_i < I_j \text{ oder } I_i > I_j\}$. Wird jede Kante in K in Richtung des größeren Intervall es ausgerichtet, so erhält man eine transitive Orientierung von \overline{G} .
39. Es sei G ein ungerichteter Graph, so daß G und \overline{G} planar sind. Bezeichne die Anzahl der Kanten von G mit m und die von \overline{G} mit \bar{m} . Dann gilt nach den Ergebnissen aus Abschnitt 5.4: $n(n-1)/2 = m + \bar{m} \leq 2(3n-6)$. Hieraus folgt $n(n-1)/(n-2) \leq 12$. Diese Ungleichung ist für $n > 10$ nicht erfüllt.
40. Der Graph ist 4-kritisch.
41. Es sei s die Anzahl der Farbklassen in einer nicht trivialen Färbung, welche genau eine Ecke enthalten. Diese bilden eine Clique. Somit ist $s \leq \chi(G)$. Da die restlichen Farbklassen mindestens zwei Farben enthalten, gilt $\chi_n(G) \leq s + (n-s)/2 \leq (n+\chi(G))/2$ bzw. $2(n-\chi_n(G)) \geq (n-\chi(G))$. Hieraus folgt die Behauptung. Der Greedy-Algorithmus erzeugt eine nicht triviale Färbung.
42. Es genügt folgende Aussage zu beweisen: Für jede Ecke i gilt: $h(i) + f(i) \leq h+1$. Hierbei bezeichnet $f(i)$ die vom Greedy-Algorithmus vergebene Farbnummer und $h(i)$ die Höhe der Ecke i . Der Beweis erfolgt durch vollständige Induktion nach der Farbnummer. Für Ecken i mit $f(i) = 1$ ist die Aussage klar. Sei nun i eine Ecke mit $f(i) > 1$. Da i nicht mit der Farbnummer $f(i)-1$ gefärbt wurde, muß es einen Nachbarn j von i mit $j < i$ und $f(j) = f(i)-1$ geben. Dann folgt $h(j) \geq h(i)+1$. Nach Induktionsvoraussetzung gilt nun:

$$h+1 \geq h(j) + f(j) \geq h(i) + 1 + f(i) - 1 = h(i) + f(i)$$

43. Man beachte zunächst, daß Färbungen der Graphen G^+ und G^- Färbungen auf G induzieren, welche genau soviel Farben verwenden. Somit ist $\chi(G^+) \geq \chi(G)$ und $\chi(G^-) \geq \chi(G)$. Es sei f eine minimale Färbung von G . Ist $f(a) = f(b)$, so induziert f eine Färbung auf G^+ . Somit gilt $\chi(G) \geq \chi(G^+)$ bzw. $\chi(G) = \chi(G^+)$. Ist hingegen $f(a) \neq f(b)$, so induziert f eine Färbung auf G^- . In diesem Fall gilt $\chi(G) \geq \chi(G^-)$ bzw. $\chi(G) = \chi(G^-)$. Hieraus folgt $\chi(G) = \min(\chi(G^+), \chi(G^-))$.

Die Gleichung führt direkt zu einem rekursiven Algorithmus zur Bestimmung einer minimalen Färbung. Ist G leer oder vollständig, so kann eine minimale Färbung direkt angegeben werden. Andernfalls werden zwei nicht benachbarte Ecken a, b ausgewählt und rekursiv minimale Färbungen für G^+ und G^- bestimmt. Die Färbung, welche die wenigsten Farben verwendet, wird zu einer Färbung auf G erweitert. Der durch dieses Verfahren aufgebaute Lösungsbaum heißt *Zykov Baum*.

B.6 Kapitel 6

1. Es sei f ein maximaler Fluß auf G und f' ein maximaler Fluß auf G' . Da f' ein zulässiger Fluß für G ist, gilt: $|f| \geq |f'|$. Es sei (X, \bar{X}) ein q-s-Schnitt von G'

mit $|f'| = \kappa_{G'}(X, \bar{X})$. Da keine Kante aus H in dem Schnitt (X, \bar{X}) liegt, gilt $|f'| = \kappa_G(X, \bar{X})$. Analog zu dem Beweis des Lemmas aus Abschnitt 6.1 folgt nun:

$$|f| = f(X, \bar{X}) - f(\bar{X}, X) \leq \kappa_G(X, \bar{X}) = |f'|$$

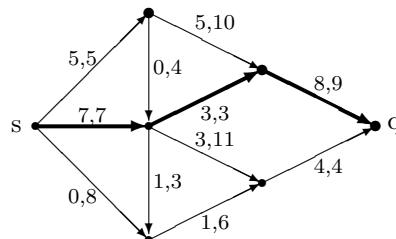
Man beachte hierzu die in Abschnitt 6.1 angegebene Definition von $|f|$. Hieraus folgt $|f'| = |f|$.

2. Es sei X die Menge der vier Ecken auf der linken Seite des Netzwerkes. Dann gilt $\kappa(X, \bar{X}) = 1 + \lambda + \lambda^2 = 2$. Somit ist der Fluß, welcher die drei mittleren waagrechten Kanten sättigt, maximal und hat den Wert 2. Wählt man die Erweiterungswege W_i wie in dem Hinweis angegeben, so ist für $i > 0$ jeweils eine der drei mittleren Kanten eine Rückwärtskante und die anderen beiden sind Vorwärtskanten. Die Wahl der Rückwärtskante ändert sich dabei zyklisch: jeweils die darüberliegende Kante ist im nächsten Erweiterungsweg die Rückwärtskante. Unter Ausnutzung der im Hinweis angegebenen Beziehung $\lambda^{i+2} = \lambda^i - \lambda^{i+1}$ folgt, daß die entstehenden Flüsse $|f_i|$ folgende Werte haben:

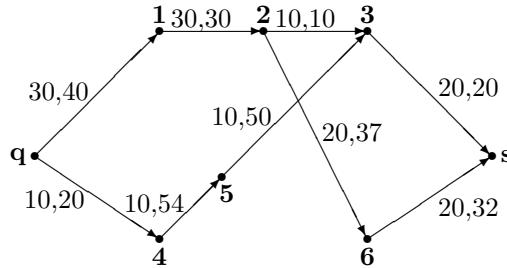
$$|f_i| = 1 + \sum_{j=2}^{i+1} \lambda^j = -\lambda + \sum_{j=0}^{i+1} \lambda^j < -\lambda + \lambda + 2 = 2$$

Ferner gilt $\lim_{i \rightarrow \infty} |f_i| = 2$. Fügt man noch eine zusätzliche Kante von q nach s mit Kapazität 1 ein, so ist der maximale Fluß dieses Netzwerkes gleich 3. Somit konvergiert die Folge der Flüsse f_i noch nicht einmal gegen den maximalen Fluß.

3. a) Der Wert von f ist 11.
- b) Die fett gezeichneten drei Kanten bilden einen Erweiterungsweg für f mit $f_\Delta = 1$. Der erhöhte Fluß hat den Wert 12 und ist unten dargestellt.
- c) Es sei X die Menge der drei fett gezeichneten Ecken. Dann ist $\kappa(\bar{X}, X) = 12$ und somit ist $\kappa(X, X)$ ein Schnitt mit minimaler Kapazität und der erhöhte Fluß ist maximal.
- d) Der unten dargestellte Fluß ist maximal und hat den Wert 12.

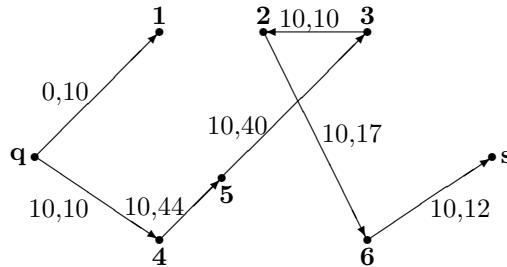


4. Das zu dem trivialen Fluß f_0 gehörende geschichtete Hilfsnetzwerk G'_{f_0} sieht wie folgt aus. Die Bewertungen der Kanten geben den blockierenden Fluß f_1 und die Kapazitäten an.



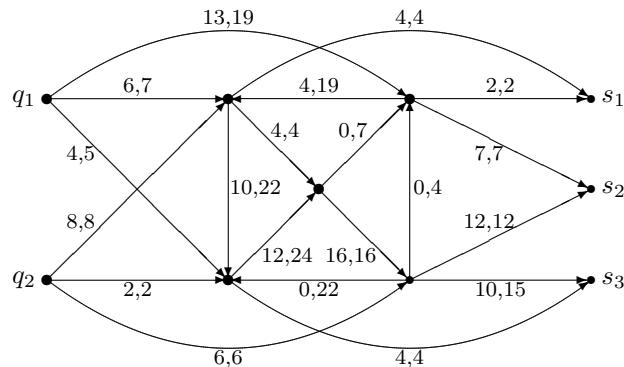
Ecke	q	1	2	3	4	5	6	s
Durchsatz	60	30	30	20	20	50	32	52

Bei der Bestimmung von f_1 hat zunächst Ecke 3 und danach Ecke 1 minimalen Durchsatz. Der blockierende Fluß f_1 hat den Wert 40. Das zu f_1 gehörende ge schichtete Hilfsnetzwerk C'_{f_1} sieht wie folgt aus.



Hieraus ergibt sich ein blockierender Fluß f_2 mit dem Wert 10 und somit ein maximaler Fluß f mit dem Wert 50. Im Gegensatz zum ursprünglichen Netzwerk wird die Prozedur **blockfluss** nur zweimal aufgerufen.

5. Die zusätzlichen Kanten müssen eine unbeschränkte Kapazität haben. Der unten dargestellte Fluß hat den Wert 39. Die fett dargestellten Ecken bilden einen Schnitt mit minimaler Kapazität.



6. Ist $X_1 \subseteq X_2$ oder $X_2 \subseteq X_1$, so ist die Aussage wahr. Im folgenden wird der Fall betrachtet, daß $X_1 \not\subseteq X_2 \not\subseteq X_1$ gilt. Dann sind folgende Mengen nicht leer und paarweise disjunkt:

$$A = X_1 \cap X_2, B = \overline{X_1} \cap X_2, C = X_1 \cap \overline{X_2}, D = \overline{X_1} \cap \overline{X_2}.$$

Ferner gilt $q \in A, s \in D, \overline{X_1 \cap X_2} = B \cup C \cup D, D = \overline{X_1 \cup X_2}$ und $X_1 \cup X_2 = A \cup B \cup C$. Da $(X_1, \overline{X_1})$ und $(X_2, \overline{X_2})$ minimale Schnitte sind, gilt:

$$\begin{aligned}\kappa(X_1, \overline{X_1}) &\leq \kappa(A, B \cup C \cup D) \\ \kappa(X_1, \overline{X_1}) &\leq \kappa(A \cup B \cup C, D) \\ \kappa(X_2, \overline{X_2}) &\leq \kappa(A, B \cup C \cup D) \\ \kappa(X_2, \overline{X_2}) &\leq \kappa(A \cup B \cup C, D)\end{aligned}$$

Wegen $X_1 = A \cup C$ und $\overline{X_1} = B \cup D$ folgt aus der ersten Ungleichung

$$\kappa(A, B) + \kappa(A, D) + \kappa(C, B) + \kappa(C, D) \leq \kappa(A, B) + \kappa(A, C) + \kappa(A, D)$$

bzw.

$$\kappa(C, B) + \kappa(C, D) \leq \kappa(A, C).$$

Aus den anderen drei Ungleichungen ergibt sich analog:

$$\begin{aligned}\kappa(A, B) + \kappa(C, B) &\leq \kappa(B, D) \\ \kappa(B, C) + \kappa(B, D) &\leq \kappa(A, B) \\ \kappa(A, C) + \kappa(B, C) &\leq \kappa(C, D)\end{aligned}$$

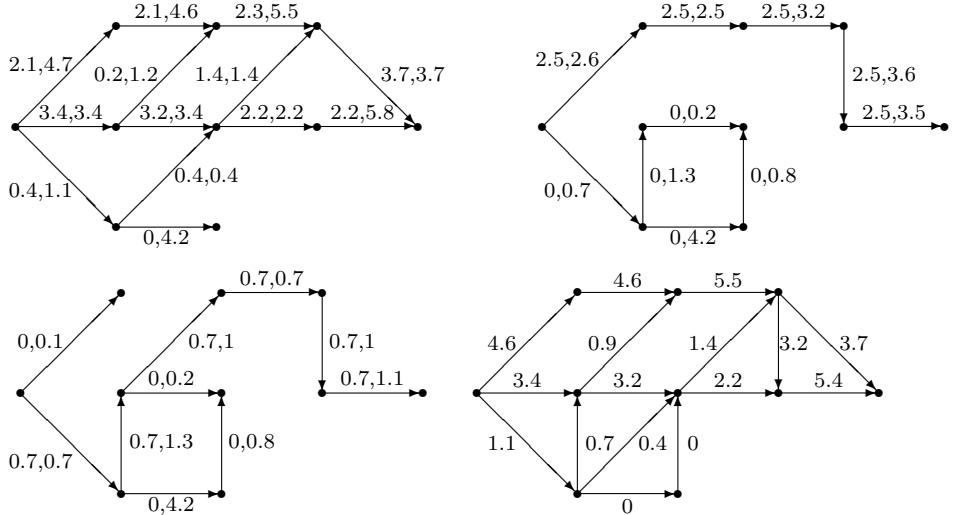
Addiert man die letzten vier Ungleichungen, so folgt $\kappa(C, B) = \kappa(B, C) = 0$, da alle Kapazitäten nicht negativ sind. Setzt man dies in die letzten vier Ungleichungen ein, so ergibt sich $\kappa(A, B) = \kappa(B, D)$ und $\kappa(A, C) = \kappa(C, D)$. Schließlich gilt:

$$\begin{aligned}\kappa(X_1, \overline{X_1}) &= \kappa(A, B) + \kappa(A, D) + \kappa(C, D) \\ &= \kappa(A, B) + \kappa(A, D) + \kappa(A, C) \\ &= \kappa(A, B \cup D \cup C) \\ &= \kappa(X_1 \cap X_2, \overline{X_1 \cap X_2})\end{aligned}$$

und

$$\begin{aligned}\kappa(X_1, \overline{X_1}) &= \kappa(A, B) + \kappa(A, D) + \kappa(C, D) \\ &= \kappa(B, D) + \kappa(A, D) + \kappa(C, D) \\ &= \kappa(B \cup A \cup C, D) \\ &= \kappa(X_1 \cup X_2, \overline{X_1 \cup X_2})\end{aligned}$$

7. Enthält ein Netzwerk einen geschlossenen Weg W , so daß $|f| < \kappa(k)$ für alle Kanten k von W gilt, so gibt es einen maximalen Fluß f' und eine Kante k auf W mit $f'(k) > |f'| = |f|$.
 8. Im folgenden sind die drei konstruierten geschichteten Hilfsnetzwerke und die dazugehörigen blockierenden Flüsse dargestellt. Die letzte Abbildung zeigt den maximalen Fluß mit Wert 9.1.



- Die Prozedur `blockfluf` findet unabhängig vom Wert von b einen blockierenden Fluß mit Wert 4. Dieser verwendet die beiden oberen Kanten. Ist $b = 0$, so ist der blockierende Fluß maximal. Ein maximaler Fluß hat den Wert $\min(4 + b, 10)$.
 - Es sei f ein maximaler Fluß und W ein Weg von q nach s mit $f(k) > 0$ für alle Kanten k von W . Ferner sei k_0 eine Kante von W mit $f(k_0) \leq f(k)$ für alle Kanten k von W . Man entferne k_0 und setze $f'(k) = f(k) - f(k_0)$ für alle k auf W und ansonsten $f'(k) = f(k)$. Dieses Verfahren kann maximal m mal wiederholt werden. Die so entstehenden Wege bilden eine Folge von Erweiterungswegen, die zu einem maximalen Fluß führen. Leider ergibt sich hieraus kein neuer Algorithmus zur Bestimmung eines maximalen Flusses, da ein solcher Fluß schon für die Bestimmung der Wege benötigt wird.
 - Der Algorithmus konstruiere die Flüsse f_0, f_1, \dots, f_k , wobei f_0 der triviale und f_k ein maximaler Fluß ist. Setze $\Delta_i = |f_{i+1}| - |f_i|$. Nach Aufgabe 11 kann mit maximal m Erweiterungswegen ein maximaler Fluß gefunden werden (mit f_i startend). Somit gilt $\Delta_i \geq (|f_{max}| - |f_i|)/m$. Hieraus folgt:

$$|f_{max}| - |f_i| \leq m\Delta_i = m(|f_{i+1}| - |f_i|) = m(|f_{i+1}| - |f_{max}|) + m(|f_{max}| - |f_i|)$$

Somit ergibt sich $|f_{max}| - |f_{i+1}| \leq (|f_{max}| - |f_i|)(1 - 1/m)$. Da f_0 der triviale Fluß ist, kann mittels vollständiger Induktion gezeigt werden, daß folgende Ungleichung

gilt:

$$|f_{max}| - |f_i| \leq |f_{max}|(1 - 1/m)^i$$

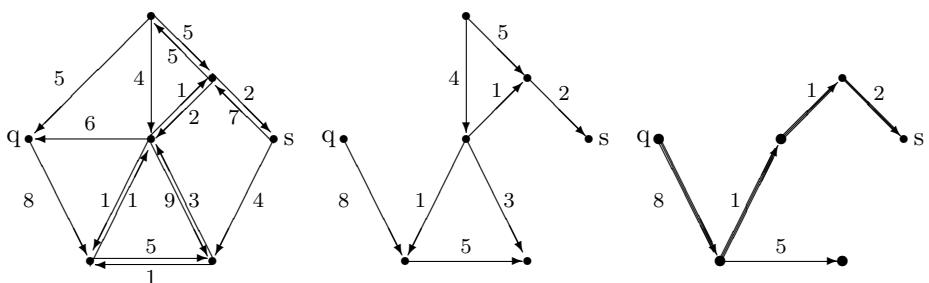
Da alle Kapazitäten ganzzahlig sind, folgt:

$$1 \leq |f_{max}| - |f_{k-1}| \leq |f_{max}|(1 - 1/m)^{k-1}$$

Unter Verwendung der Ungleichung $m(\log m - \log(m-1)) \geq 1$ zeigt man nun $k-1 \leq m \log |f_{max}|$. Hieraus folgt die Aussage.

12. Es sei f ein maximaler Fluß des Netzwerkes.

- a) Wird die Kapazität jeder Kante um C erhöht, so erhöht sich auch der Wert eines maximalen Flusses um mindestens C . Dazu wird der Wert von f auf den Kanten eines Weges W von q nach s jeweils um C erhöht. Gibt es noch andere Wege W' von q nach s , welche mit W keine gemeinsamen Kanten haben, so kann der Wert von f noch weiter erhöht werden.
 - b) Wird die Kapazität jeder Kante um den Faktor C erhöht, so setze man $f'(k) = Cf(k)$ für jede Kante k des Netzwerkes. Dann ist f' ein maximaler Fluß in dem neuen Netzwerk und es gilt $|f'| = C|f|$.
13. Die Kapazitäten der Kanten seien z_i/n_i für $i = 1, \dots, m$, wobei z_i und n_i positive ganze Zahlen sind. Ferner sei V das kleinste gemeinsame Vielfache der Nenner n_1, \dots, n_m . Dann ist $f_\Delta \geq 1/V$ für jeden Erweiterungsweg jedes Flusses f . Ist M der Wert eines maximalen Flusses, so endet der in Abschnitt 6.1 beschriebene Algorithmus spätestens nach MV Schritten.
14. Der Wert eines Schnittes ist in beiden Netzwerken gleich. Die Behauptung folgt aus dem Satz von Ford und Fulkerson.
15. Im folgenden ist zunächst der Graph G_f dargestellt. Daneben der Graph, welcher von den Vorrwärtskanten von G_f induziert wird. Da es in diesem Graph keinen Weg von q nach s gibt, ist f ein blockierender Fluß. Ganz rechts ist der Graph G'_f dargestellt. Die fett dargestellten Kanten bilden einen blockierenden Fluß h mit Wert 1. Der neue Fluß $f + h$ hat den Wert 12 und ist maximal. Dazu betrachte man die in G'_f fett gezeichnete Menge X von Ecken und zeige, daß $\kappa(X, \bar{X}) = 12$ ist.



16. Da f_1 und f_2 Flüsse auf G sind, gilt $f(k) \geq 0$ für jede Kante k von G und die Flußberhaltungsbedingung ist für f erfüllt. Somit ist f genau dann ein Fluß von G , wenn $f_1(k) + f_2(k) \leq \kappa(k)$ für jede Kante k von G gilt.
17. a) Wird die Kapazität einer Kante um 1 erhöht, so erhöht sich der maximale Fluß höchstens um 1. Da die Kapazitäten ganzzahlig sind, findet der Algorithmus von Edmonds und Karp in einem Schritt (mit f startend) einen maximalen Fluß. Somit ist der Aufwand $O(n + m)$.
- b) Wird die Kapazität einer Kante um 1 erniedrigt, so erniedrigt sich der maximale Fluß höchstens um 1. Ist $f(k) < \kappa(k)$, so folgt aus der Ganzzahligkeit der Kapazitäten, daß f auch in dem neuen Netzwerk ein maximaler Fluß ist. Ist $f(k) = \kappa(k)$, so findet man mit Aufwand $O(n + m)$ einen Weg W von q nach s , welcher die Kante k enthält und der Fluß durch jede Kante von W positiv ist. Bilde einen neuen Fluß f' durch $f'(k) = f(k) - 1$ für alle Kanten k auf W und $f'(k) = f(k)$ ansonsten. Dann ist f' ein zulässiger Fluß auf dem neuen Netzwerk mit $|f'| = |f| - 1$. Der Algorithmus von Edmonds und Karp findet in einem Schritt (mit f' startend) einen maximalen Fluß. Somit ist der Aufwand $O(n + m)$.
18. Die Prozeduren **erweiterevorwärts** und **und erweitererückwärts** können nicht direkt verwendet werden, sondern müssen noch angepaßt werden. Das folgende Programm ist eine Variante der Prozedur **erweitererückwärts**. Man beachte, daß am Ende der **while**-Anweisung die Flußberhaltungsbedingung für die entfernte Ecke i erfüllt ist (nur für $i \neq e$). Die Prozedur **erweiterevorwärts** muß entsprechend abgeändert werden.

```

var durchfluß : array[1..max] of Real;
W : warteschlange of Integer;
i,j : Integer;
m : Real;
begin
  Initialisiere durchfluss mit 0;
  durchfluß[e] := f_e;
  W.einfügen(e);
repeat
  i := W.entfernen;
  while durchfluß[i] ≠ 0 do begin
    sei k=(i,j) eine Kante mit f(k) > 0;
    m := min(f(k), durchfluß[i]);
    f(k) := f(k) - m;
    if j ≠ s then
      W.einfügen(j);
    durchfluß[j] := durchfluß[j] + m;
    durchfluß[i] := durchfluß[i] - m;
  end;
  until W = ∅;
end

```

19. a) Wegen $\alpha \in [0, 1]$ gilt für alle Kanten k :

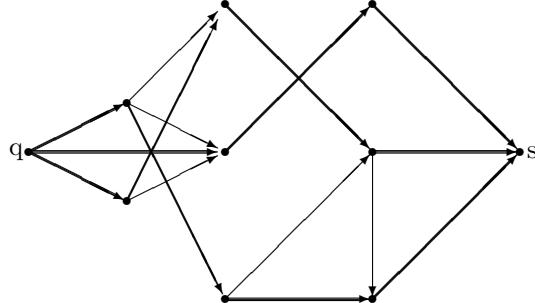
$$0 \leq f_\alpha(k) = \alpha f_1(k) + (1 - \alpha) f_2(k) \leq \alpha \kappa(k) + (1 - \alpha) \kappa(k) = \kappa(k)$$

b) Für jede Ecke $e \neq q, s$ von G gilt:

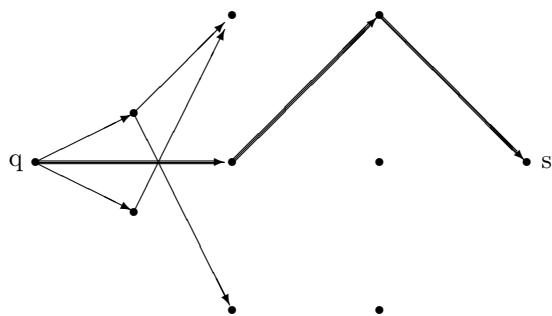
$$\begin{aligned}
 \sum_{k=(j,e) \in K} f_\alpha(k) &= \sum_{k=(j,e) \in K} \alpha f_1(k) + (1 - \alpha) f_2(k) \\
 &= \alpha \sum_{k=(j,e) \in K} f_1(k) + (1 - \alpha) \sum_{k=(j,e) \in K} f_2(k) \\
 &= \alpha \sum_{k=(e,j) \in K} f_1(k) + (1 - \alpha) \sum_{k=(e,j) \in K} f_2(k) \\
 &= \sum_{k=(e,j) \in K} \alpha f_1(k) + (1 - \alpha) f_2(k) \\
 &= \sum_{k=(e,j) \in K} f_\alpha(k)
 \end{aligned}$$

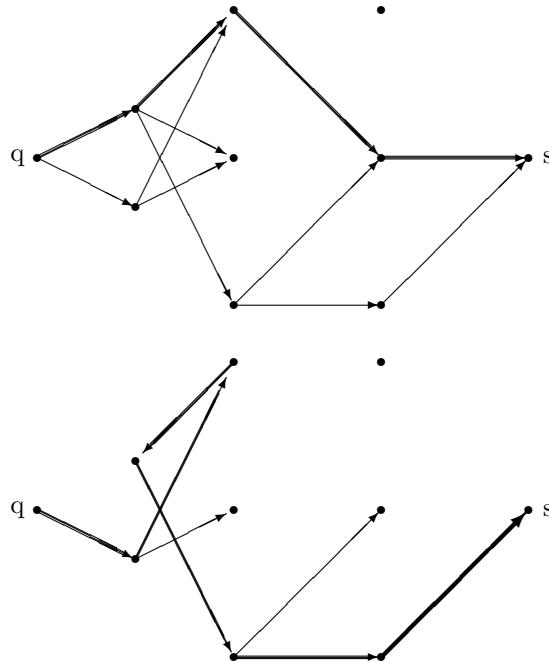
c) $|f_\alpha| = \alpha|f_1| + (1 - \alpha)|f_2|$.

20. Es sei k eine Kante eines minimalen Schnittes von G mit positiver Kapazität. Dann ist die Kapazität eines minimalen Schnittes von $G \setminus \{k\}$ echt kleiner als die eines minimalen Schnittes von G . Die Aussage folgt nun aus dem Satz von Ford und Fulkerson.
21. Die fett gezeichneten Kanten bilden einen maximalen Fluß.



Hierzu werden die im folgenden dargestellten drei blockierenden Flüsse gebildet. Dargestellt sind jeweils die geschichteten Hilfsnetzwerke und die blockierenden Flüsse (fette Kanten).





22. Der Beweis der Aussagen erfolgt analog zu den Beweisen in Abschnitt 6.5. Der Beweis für Aussage a) ergibt sich aus dem Beweis des Lemmas aus diesem Abschnitt und der für Aussage b) aus dem Beweis des nachfolgenden Satzes.

B.7 Kapitel 7

1. In Abschnitt 7.1 wurde gezeigt, daß es eine eindeutige Beziehung zwischen den Zuordnungen eines bipartiten Graphen G und den binären Flüssen des zugehörigen 0-1-Netzwerkes N_G gibt. Es sei Z eine Zuordnung eines bipartiten Graphen G . Ein Weg W , welcher zwei nicht zugeordnete Ecken verbindet, heißt Erweiterungsweg für Z , falls jede zweite Kante von W in Z liegt. Die Erweiterungswägen für Z in G entsprechen eindeutig den Erweiterungswägen bezüglich f_Z in N_G . Der Begriff des Erweiterungsweges läßt sich auch auf beliebige ungerichtete Graphen übertragen. Es gilt folgender Satz.

Satz. Eine Zuordnung Z in einem ungerichteten Graphen G ist genau dann maximal, wenn es in G für Z keinen Erweiterungsweg gibt.

Beweis. Es sei Z eine Zuordnung und W ein Erweiterungsweg für Z . Ferner sei Z' die symmetrische Differenz von Z und W (Z' besteht aus den Kanten, die entweder nur in Z oder nur in W liegen, aber nicht in beiden Mengen). Dann ist Z' eine Zuordnung für G und es gilt $|Z'| = |Z| + 1$. Somit kann es für eine maximale Zuordnung keinen Erweiterungsweg geben.

Es sei nun Z eine Zuordnung, welche nicht maximal ist, und Z' eine maximale Zuordnung. Bezeichne mit G' den von den Kanten aus $Z \cup Z'$ induzierten Untergraphen von G . Für jede Ecke e von G' ist $g(e) \leq 2$. Ist $g(e) = 2$, so ist e zu genau einer Kante aus Z und einer Kante aus Z' inzident. Somit können die Zusammenhangskomponenten von G' in zwei Gruppen aufgeteilt werden: geschlossene und einfache Wege. Auf diesen Wegen liegen abwechselnd Kanten aus Z und Z' . Da geschlossene Wege jeweils gleichviel Kanten aus Z und Z' enthalten und es in Z' mehr Kanten als in Z gibt, muß es in G' einen Weg geben, welcher zwei bezüglich Z nicht zugeordnete Ecken verbindet. Dies ist ein Erweiterungsweg für Z in G .

■

2. Mit einem Greedy-Verfahren wird eine nicht erweiterbare Zuordnung Z bestimmt (Aufwand $O(n + m)$). Ist Z noch keine vollständige Zuordnung, so ist $|Z| < 2n$. Dann gibt es Ecken e und f von G , welche zu keiner Kante aus Z inzident sind. Da Z nicht erweiterbar ist, gibt es für jeden Nachbarn j von e oder f genau eine zu j inzidente Kante aus Z . Wegen $|Z| < 2n$ und $g(e) + g(f) \geq 2n$ gibt es eine Kante $k = (j, j') \in Z$, so daß (e, j) und (j', f) Kanten in G sind. Somit ist auch

$$Z' = Z \setminus \{k\} \cup \{(e, j), (j', f)\}$$

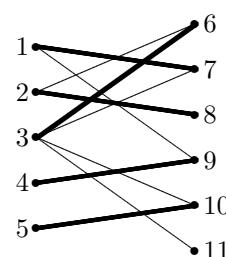
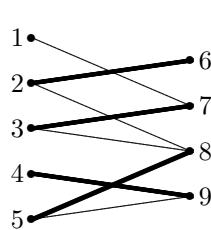
eine Zuordnung von G . Dieses Verfahren wiederholt man bis eine vollständige Zuordnung vorliegt. Der Aufwand des folgenden Algorithmus ist $O(n + m)$.

```

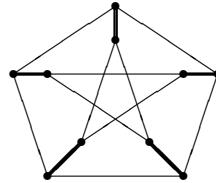
var L : array[1..max] of Integer;
Bestimme mit einem Greedy-Verfahren nicht erweiterbare Zuordnung Z;
Initialisiere L mit 0;
for alle nicht zugeordneten Ecken x,y do begin
    for jeden Nachbar j von x do begin
        sei k = (j,j') ∈ Z;
        L[j'] = j;
    end
    suche Nachbarn j von y mit L[j] ≠ 0;
    setze Z := Z \ {(j,L[j])} ∪ {(x,L[j]), (j,y)};
    for jeden Nachbar j von x do begin
        sei k = (j,j') ∈ Z;
        L[j'] = 0;
    end
end

```

3. Die fett gezeichneten Kanten bilden jeweils eine maximale Zuordnung.



4. Die fett gezeichneten Kanten bilden eine vollständige Zuordnung.



5. Wegen $\alpha(G) = 2$ besteht jede Farbklasse einer Färbung von G aus maximal zwei Ecken. Es sei a die Anzahl der Farbklassen mit zwei Ecken einer minimalen Färbung von G . Dann ist $\chi(G) = a + (n - 2a) = n - a$. Die Farbklassen mit zwei Ecken induzieren eine Zuordnung auf \overline{G} mit a Kanten. Somit gilt $a \leq z$ und $\chi(G) \geq n - z$. Jede maximale Zuordnung auf \overline{G} induziert eine Färbung mit $n - z$ Farben auf G . Somit gilt $\chi(G) \leq n - z$.
6. Es sei $T \subseteq E_1$. Ferner sei K_1 die Menge der Kanten von G , welche zu einer Ecke aus T inzident sind, und K_2 die Menge der Kanten von G , welche zu einer Ecke aus $N(T)$ inzident sind. Es gilt $K_1 \subseteq K_2$. Aus der Voraussetzung folgt $|T|k \leq |K_1|$ und $|N(T)|k \geq |K_2|$. Somit gilt $|N(T)| \geq |T|$ für alle Teilmengen T von E_1 . Die Aussage folgt nun aus dem Satz von Hall.
7. Es sei U_1 bzw. U die Menge der Ecken, welche zu keiner Kante aus Z_1 bzw. Z inzident sind. Dann sind U_1 und U unabhängige Mengen mit $|U_1| = n - 2|Z_1|$ und $|U| = n - 2|Z|$. Eine maximale unabhängige Menge von G hat maximal $|Z| + |U|$ Ecken. Somit gilt:

$$n - 2|Z_1| = |U_1| \leq \alpha(G) \leq |Z| + |U| = n - |Z|$$

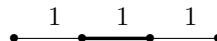
Hieraus folgt $|Z| \leq 2|Z_1|$.

8. Nach dem Satz von Hall hat G genau dann keine Zuordnung Z mit $|Z| = |E_1|$, wenn es eine Teilmenge D von E_1 mit $|D| > |N(D)|$ gibt. Zunächst wird ein maximaler binärer Fluß f auf dem zu G gehörenden 0-1-Netzwerk N_G bestimmt. Gilt $|f| = |E_1|$, so existiert keine Teilmenge D von E_1 mit $|D| > |N(D)|$. Gilt $|f| < |E_1|$, so wird die Menge X aller Ecken aus N_G bestimmt, welche von q über Erweiterungswege bezüglich f erreichbar sind. Es sei $D = X \cap E_1$. Wie im Beweis des Satzes von Hall in Abschnitt 7.1 zeigt man $|f| = |E_1 \setminus X| + |N(D)|$. Wegen $|E_1| = |E_1 \setminus X| + |D|$ folgt nun $|D| > |N(D)|$. Dieser Algorithmus hat eine Laufzeit von $O(\sqrt{nm})$.
9. Es sei G ein kantenbewerteter, vollständig bipartiter Graph und e eine beliebige Ecke von G . Es sei G' eine Kopie von G , in der die Bewertungen aller zu e inzidenten Kanten um b geändert sind. Dann ist jede vollständige Zuordnung von G auch eine vollständige Zuordnung von G' (und umgekehrt). Die Kosten der beiden Bewertungen unterscheiden sich um b . Es sei $B = b_1 + \dots + b_{|E_1|}$, wobei $b_i = \min\{\text{kosten}(k) \mid k \text{ zu } e_i \in E_1 \text{ inzidente Kante}\}$ ist. Ist die maximale Zuordnung für G_0 auch für G vollständig, so hat man eine vollständige Zuordnung für G mit minimalen Kosten B . Andernfalls bestimme man, wie in Aufgabe 8 beschrieben,

eine Teilmenge X von E_1 mit $|N_{G_0}(X)| < |X|$ und verändere den Graphen G_0 . Man beachte, daß G_0 mindestens eine zusätzliche Kante besitzt. Ist die maximale Zuordnung für G_0 auch für G vollständig, so hat man eine vollständige Zuordnung für G mit minimalen Kosten $B + (|X| - |N_{G_0}(X)|)b_{min}$. Andernfalls wird der letzte Schritt wiederholt. Da in jedem Schritt G_0 mindestens eine zusätzliche Kante bekommt, sind maximal $|E_1|(|E_1| - 1) < n^2$ Durchgänge notwendig.

10. Es sei G ein kantenbewerteter bipartiter Graph und T die Summe der Bewertungen aller Kanten von G . Durch Hinzufügen neuer Ecken und Kanten mit Bewertung T erhält man einen Graph G' mit den in Aufgabe 9 angegebenen Eigenschaften. Jede Zuordnung von G kann zu einer Zuordnung von G' ergänzt werden. Die Wahl von T bedingt, daß aus einer Zuordnung mit minimalen Kosten für G eine vollständige Zuordnung mit minimalen Kosten für G' entsteht. Es sei Z' eine vollständige Zuordnung von G' mit minimalen Kosten. Entfernt man alle Kanten mit Bewertung T aus Z' , so erhält man eine Zuordnung Z von G . Dies ist eine maximale Zuordnung mit minimalen Kosten. Der in Aufgabe 9 angegebene Algorithmus kann also zur Bestimmung maximaler Zuordnungen mit minimalen Kosten in beliebigen kantenbewerteten bipartiten Graphen verwendet werden.

Eine nicht erweiterbare Zuordnung mit minimalen Kosten ist nicht notwendigerweise eine maximale Zuordnung wie der folgende Graph zeigt. Die fett gezeichnete Kante bildet eine nicht erweiterbare Zuordnung mit minimalen Kosten. Diese Zuordnung ist aber nicht maximal.



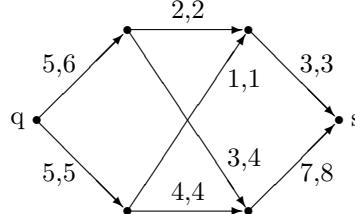
Der Zusammenhang zwischen maximalen Zuordnungen mit minimalen Kosten in kantenbewerteten, vollständig bipartiten Graphen vom Typ $G_{n,n}$ und maximalen Zuordnungen in beliebigen kantenbewerteten bipartiten Graphen ist nicht auf nicht erweiterbare Zuordnungen übertragbar. Somit kann der angegebene Algorithmus nicht zur Bestimmung nicht erweiterbarer Zuordnungen mit minimalen Kosten verwendet werden.

11. Es sei G' das angegebene neue Netzwerk. Jedem q - s -Fluß f auf G entspricht ein s - q -Fluß f' auf G' und umgekehrt. Dazu definiert man

$$f((a, b)) = -f'((b, a))$$

für alle Kanten (a, b) . Man beweist direkt, daß f' genau dann zulässig für G' ist, wenn f zulässig für G ist. Ferner ist f genau dann minimal zulässig, wenn f' maximal zulässig ist. Somit genügt es einen maximalen Fluß für das Netzwerk G' zu konstruieren. Hierzu konstruiert man zunächst wie in Abschnitt 7.2 angegeben einen zulässigen Fluß für G , dieser induziert einen zulässigen Fluß auf G' . Mit Hilfe von Erweiterungswegen kann dieser zu einem zulässigen maximalen Fluß für G' erweitert werden. Dieser entspricht dann einen minimalen zulässigen Fluß auf G .

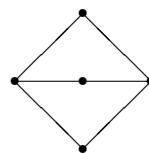
12. Der Fluß mit Wert 2 auf der Kante $(1, 2)$ und dem Wert 1 auf allen anderen Kanten hat unter den zulässigen Flüssen des Netzwerkes aus Abbildung 7.6 den kleinsten Wert.
13. Die Bewertungen der Kanten geben den Wert eines minimalen und eines maximalen Flusses an. Die Werte der Füsse sind 10 und 11.



14. Genau dann gibt es einen zulässigen Fluß auf G , wenn jede Kante von G auf einem Weg von q nach s liegt.
15. In Aufgabe 11 wurde aus einem Netzwerk G mit oberen und unteren Kapazitätsgrenzen ein neues Netzwerk G' konstruiert. Der Wert eines minimalen zulässigen q - s -Flusses f_{MIN} auf G ist gleich dem negativen Wert eines maximal zulässigen s - q -Flusses f_{MAX} auf G' . Ist (X, \bar{X}) ein q - s -Schnitt von G , so ist (\bar{X}, X) ein s - q -Schnitt von G' und es gilt $\kappa_G(X, \bar{X}) = -\kappa_{G'}(\bar{X}, X)$. Somit gilt:

$$\begin{aligned} |f_{MIN}| &= -|f_{MAX}| \\ &= -\min\{\kappa_{G'}(X, \bar{X}) \mid (X, \bar{X}) \text{ } s\text{-}q\text{-Schnitt von } G'\} \\ &= -\min\{-\kappa_G(\bar{X}, X) \mid (\bar{X}, X) \text{ } q\text{-}s\text{-Schnitt von } G\} \\ &= -\min\{\kappa_G(X, \bar{X}) \mid (\bar{X}, X) \text{ } q\text{-}s\text{-Schnitt von } G\} \\ &= -\min\{\kappa_G(\bar{X}, X) \mid (X, \bar{X}) \text{ } q\text{-}s\text{-Schnitt von } G\} \end{aligned}$$

16. Der Beweis aus Abschnitt 7.4 für die angegebene Aussage über den Kantenzusammenhang kann nicht auf den Eckenzusammenhang übertragen werden. In einem Graph G kann es eine Ecke a geben, welche in jeder trennenden Eckenmenge jedes Paares e, f von Ecken mit $Z^e(e, f) = Z^e(G)$ liegt. Dies ist z.B. gegeben, wenn eine Ecke a zu jeder anderen Ecke benachbart ist. Betrachten Sie z.B. die Ecken mit Grad 3 in folgendem Graph:



17. Es sei T eine minimale trennende Kantenmenge für a, b . Der Graph $G \setminus T$ ist nicht mehr zusammenhängend und a und b liegen in verschiedenen Zusammenhangskomponenten. Dann liegt c entweder in der gleichen Zusammenhangskomponente wie a oder b oder in einer anderen Zusammenhangskomponente. Auf jeden

Fall ist T eine trennende Eckenmenge für a, c oder b, c . Somit gilt $Z^k(a, b) \geq \min\{Z^k(a, c), Z^k(c, b)\}$.

18. Es sei E die Eckenmenge von G und G' der vollständige Graph mit Eckenmenge E . Jede Kante (a, b) von G' wird mit $|f_{ab}|$ bewertet (man beachte, daß $|f_{ab}| = |f_{ba}|$ gilt, dies gilt nicht für unsymmetrische Netzwerke). Es sei B ein maximal aufspannender Baum von G' und x, y beliebige Ecken aus E . Für die Kanten $k_i = (a_i, b_i)$ eines Weges W von x nach y in B gilt somit:

$$|f_{xy}| \leq \min\{|f_{a_i b_i}| \mid (a_i, b_i) \in W\}$$

In Abschnitt 7.4 wurde bewiesen, daß $|f_{ab}| = Z^k(a, b)$ für jedes Paar von Ecken a, b aus G gilt. Aus Aufgabe 17 folgt somit $|f_{xy}| = \min\{|f_{a_i b_i}| \mid (a_i, b_i) \in W\}$. Zu jedem Paar von Ecken a, b von G gibt es also eine Kante (x, y) von B mit $|f_{ab}| = |f_{xy}|$. Da B $n - 1$ und G' $n(n - 1)/2$ Kanten hat, muß es mindestens $n/2$ Eckenpaare geben, so daß die Werte der entsprechenden maximalen Flüsse alle gleich sind.

19. Es sei N_G das zu G gehörende 0-1-Netzwerk und f ein binärer Fluß auf N_G . Ferner sei W ein einfacher Erweiterungsweg von N_G bezüglich f und f' der durch W entstandene erhöhte Fluß. Die spezielle Struktur von N_G bwirkt, daß sich auf W Vorwärts- und Rückwärtskanten abwechseln. Des weiteren beginnt und endet W mit einer Vorwärtskante. Für jede Ecke e sei $f(e)$ bzw. $f'(e)$ der Fluß durch die Ecke e bezüglich f bzw. f' . Es gilt $f'(e) \geq f(e)$ für jede Ecke e von G .

Die Kanten aus Z induzieren einen binären Fluß f_Z auf N_G (hierzu vergleiche man den Beweis des Lemmas aus Abschnitt 7.1). Es sei f ein binärer maximaler Fluß von N_G , welcher durch eine Folge von Erweiterungswegen aus f_Z entsteht. Dann gilt $f(e) \geq f_Z(e)$ für jede Ecke e von G . Hieraus folgt die zu beweisende Aussage.

20. Sowohl die Ecken- als auch Kantenzusammenhangszahl ist gleich 3.
21. Für beide Graphen gilt $Z^e(G) = 4$.
22. Es gilt $Z^e(G) = Z^k(G) = 3$.
23. Es sei G ein 2-fach kantenzusammenhängender Graph und X die Menge der Ecken, welche in G' von der Startecke s aus erreichbar sind. Ferner sei Y die Menge der restlichen Ecken. Angenommen Y ist nicht leer. Es sei $e \in Y$ die Ecke mit der kleinsten Tiefensuchenummer. Da G zusammenhängend ist, sind alle Ecken aus Y im Tiefensuchebaum Nachfolger von e . Es sei (a, b) eine Kante aus G mit $a \in X$ und $b \in Y$. Im folgenden wird gezeigt, daß es nur eine solche Kante gibt. Dies steht aber im Widerspruch zu $Z^k(G) \geq 2$. Somit ist Y leer. Da b nicht in X liegt, ist (b, a) eine Kante in G' . Wäre (a, b) in G eine Rückwärtskante, so wäre $TSN[b] < TSN[a]$, d.h. a wäre Nachfolger von b im Tiefensuchebaum. Dann wäre aber b von a in G' erreichbar und somit auch b von s . Somit muß (a, b) in G eine Baumkante sein. Da b im Tiefensuchebaum ein Nachfolger von e ist, muß $b = e$ sein und a der Vorgänger von e im Tiefensuchebaum sein. Also ist (a, b) die einzige Kante in G , die Ecken aus X und Y verbindet.

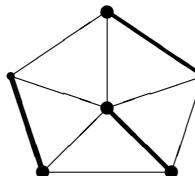
Sei nun umgekehrt jede Ecke in G' von der Startecke s aus erreichbar. Dann ist G zusammenhängend. Angenommen $Z^k(G) = 1$ und $\{k\}$ ist eine trennende Kantenmenge für G . Dann muß $k = (a, b)$ in G eine Baumkante sein. Also ist b in G' nicht von s aus erreichbar.

Der Aufwand ist $O(m)$, da die Tiefensuche zweimal angewendet wird.

24. Es sei R das Rechteck, bei dem die entfernten Felder in diagonal gegenüberliegenden Ecken liegen. Eine Seite von R hat gerade und die andere ungerade Länge. Die Fläche des Schachbrettes außerhalb von R kann in 4 Rechtecke aufgeteilt werden. Mindestens eine Seite jedes Rechteckes hat gerade Länge (eventuell 0). Dieser Teil des Schachbrettes kann somit mit Rechtecken der Größe 1×2 vollständig abgedeckt werden. R ohne die beiden Felder kann in 3 Rechtecke aufgeteilt werden, bei denen jeweils eine Seite gerade Länge hat (eventuell 0). Somit gibt es auch eine Überdeckung für R .

Die schwarzen und weißen Felder des Schachbrettes induzieren einen bipartiten Graph. Eine vollständige Überdeckung mit Recken entspricht einer vollständigen Zuordnung dieses Graphen. Da zwei diagonal gegenüberliegende Felder die gleiche Farbe haben, besitzt der Restgraph keine vollständige Zuordnung. Somit gibt es auch keine vollständige Überdeckung der Restfläche.

25. a) Zu jeder Kante von M muß eine Ecke aus U inzident sein. Da M eine Zuordnung ist, sind diese Ecken alle verschieden. Somit ist $|M| \leq |U|$.
 b) Die 4 fett gezeichneten Ecken bilden eine minimale Eckenüberdeckung und die 3 fett gezeichneten Kanten bilden eine maximale Zuordnung.



- c) Der Graph G wird zu einem ungerichteten Graphen G' mit zwei zusätzlichen Ecken a und b erweitert. Hierbei ist a zu jeder Ecke aus E_1 und b zu jeder Ecke aus E_2 inzident. Nach dem Satz von Menger gilt $Z^e(a, b) = W^e(a, b)$ für G' . Die Ecken aus einer minimalen trennenden Eckenmenge für a, b bilden eine Eckenüberdeckung. Des weiteren ist $|W^e(a, b)|$ gleich der Anzahl der Kanten in einer maximalen Zuordnung von G , d.h. $|U| \leq |M|$. Die Aussage folgt nun aus Teil a) dieser Aufgabe.
 d) Gibt es eine vollständige Zuordnung von G , so gilt $|E_1| = |E_2|$ und eine Eckenüberdeckung muß mindestens $|E_1| = (|E_1| + |E_2|)/2$ Ecken enthalten. Für die umgekehrte Beweisrichtung beachte man, daß sowohl E_1 als auch E_2 Eckenüberdeckungen von G sind. Da beide deshalb mindestens $(|E_1| + |E_2|)/2$ Ecken enthalten, folgt daraus $|E_1| = |E_2|$. Nun folgt aus dem Satz von König-Egerváry, daß G eine vollständige Zuordnung besitzt.
 e) Die Cliquenpartitionszahl $\theta(G)$ eines Graphen G ist die minimale Anzahl von Mengen einer Partition der Eckenmenge von G , bei der die von den

Teilmengen induzierten Graphen vollständig sind. Es ist $\theta(G) = \chi(\overline{G})$. Da $\omega(\overline{G}) = \alpha(G)$ ist, genügt es zu zeigen, daß für bipartite Graphen $\theta(G) = \alpha(G)$ ist.

Ist e eine isolierte Ecke von G , so liegt diese in jeder maximalen unabhängigen Menge und $\{e\}$ ist in jeder Cliquenpartition enthalten. Somit kann angenommen werden, daß G keine isolierten Ecken besitzt.

Nach dem Satz von König-Egerváry ist die Anzahl u der Ecken in einer minimalen Eckenüberdeckung von G gleich der Anzahl s der Kanten in einer maximalen Zuordnung. Da $n - \alpha(G) = u$ ist, genügt es zu zeigen, daß $n - \theta(G) = s$ ist. Hieraus folgt, daß $\theta(G) = \alpha(G)$ ist.

Es sei zunächst M eine maximale Zuordnung mit s Kanten. Dann erhält man leicht eine Cliquenpartition mit $s + n - 2s = n - s$ Mengen. Somit gilt $\theta(G) \leq n - s$. Es sei nun M' die von den zwei-elementigen Mengen einer minimalen Cliquenpartition gebildete Zuordnung. Ferner wähle man zu jeder ein-elementigen Menge eine zu dieser Ecke inzidente Kante. Die so entstandene Kantenmenge W bildet einen Wald mit $\theta(G)$ Kanten und n Ecken. Nach den Ergebnissen aus Abschnitt 3.1 besteht W aus $n - \theta(G)$ Zusammenhangskomponenten. Da die Anzahl der Zusammenhangskomponenten durch s beschränkt ist, gilt $n - \theta(G) \leq s$. Hieraus folgt $n - \theta(G) = s$.

- f) Eine Ecke e in einer minimalen Eckenüberdeckung von G überdeckt maximal n Kanten. Hieraus folgt, daß eine minimale Eckenüberdeckung aus mindestens s Ecken bestehen muss. Die Aussage folgt nun aus dem Satz von König-Egerváry.
26. Nach den Ergebnissen aus Abschnitt 5.4 gilt $\delta(G) \leq 5$. Nach dem ersten Satz in Abschnitt 7.4 gilt dann $Z^e(G) \leq \delta(G) \leq 5$.
27. Für die Graphen I_n mit $n \geq 5$ gilt $Z^k(I_n) = Z^e(I_n) = 4$.
28. Sowohl die Ecken- als auch Kantenzusammenhangszahl ist gleich 2.
29. Es seien a, b Ecken mit $Z^e(a, b) = 1$ und $\{x\}$ eine trennende Eckenmenge für a, b . Der Graph $G \setminus \{x\}$ zerfällt in die Zusammenhangskomponenten Z_1, \dots, Z_s mit $s > 1$. Es sei K_i die Menge der Kanten, welche x mit Ecken aus Z_i verbinden. Diese Mengen sind trennende Kantenmengen für a, b . Da x den Grad g hat, gibt es ein K_i mit $|K_i| \leq g/2$. Somit gilt $Z^k(G) \leq |K_i| \leq g/2$.
30. Das Tiefesucheverfahren legt die Reihenfolge, in der die Nachbarn einer Ecke besucht werden, nicht fest. Es wird folgende Reihenfolge betrachtet: für jede Ecke wird zunächst der unbesuchte Nachbar betrachtet, welcher über eine Kante aus Z erreichbar ist (falls vorhanden). Es sei B der hierdurch entstehende Tiefensuchebaum und (x, y) eine beliebige Kante aus Z . Ohne Einschränkung der Allgemeinheit kann angenommen werden, daß x vor y besucht wurde. Somit wurde beim Besuch von x der Nachbar y zuerst betrachtet. Da zu diesem Zeitpunkt y noch nicht besucht wurde, wird (x, y) in B eingefügt. Somit sind alle Kanten aus Z auch in B enthalten.

31. Es sei i eine Ecke von B , welche zu keiner Kante aus Z inzident ist. Somit wurde i im Verlauf des Algorithmus nicht markiert. Beim Verlassen von i durch die Tiefensuche muß somit der Vorgänger j von i schon markiert sein. Da die Tiefensuche in diesem Moment j noch nicht verlassen hat, muß es einen Nachfolger s von j geben, so daß die Kante (j, s) in Z liegt.

Angenommen es gibt in B einen Erweiterungsweg $W = \{e_1, \dots, e_s\}$ bezüglich Z (vergleichen Sie Aufgabe 9). Nach Konstruktion von Z ist $s \geq 4$. Ohne Einschränkung der Allgemeinheit kann angenommen werden, daß e_2 der Vorgänger von e_1 im Tiefensuchebaum ist (sonst ist e_{s-1} ein Vorgänger von e_s). Nach der oben bewiesenen Eigenschaft muß e_3 ein Nachfolger von e_2 sein. Somit ist e_s, e_{s-1}, \dots, e_2 ein gerichteter Weg im Tiefensuchebaum. Analog zu oben zeigt man, daß e_{s-2} ein Nachfolger von e_{s-1} ist. Dieser Widerspruch zeigt, daß es bezüglich Z keinen Erweiterungsweg gibt. Somit ist Z eine maximale Zuordnung von B .

32. Es sei $M \in \mathcal{U}$ mit $f(M) = \max\{f(U) \mid U \in \mathcal{U}\}$. Wird eine Kante aus M entfernt, so entsteht eine isolierte Ecke (da die Anzahl der Ecken unverändert bleibt, würde andernfalls $f(M') > f(M)$ für den neuen Graphen M' gelten). Somit enthält M keinen geschlossenen Weg und M ist ein Wald. Aus dem gleichen Grund gibt es in M auch keinen Weg der Länge 4. Daraus folgt, daß jede Zusammenhangskomponente von M ein Sterngraph ist. Es seien M_1, \dots, M_s die Zusammenhangskomponenten von M . Bezeichnet man mit k die Anzahl der Kanten in M , so enthält M genau $k+s$ Ecken und es gilt $f(M) = (k+s-1)/k$. Angenommen $s > 1$. Nun wird eine neuer Untergraph M' von G gebildet. M' enthält aus jeder Zusammenhangskomponente M_i genau eine Kante. Dann gilt $M' \in \mathcal{U}$ und $f(M') = (2s-1)/s$. Aus $f(M) \geq f(M')$ folgt nun $s \geq k$, d.h. die Kanten in M bilden eine Zuordnung. Aus der Maximalität von $f(M)$ folgt, daß diese Zuordnung maximal ist. Andernfalls ist $s = 1$ und man sieht direkt, daß G bis auf isolierte Ecken ein Sterngraph ist.

33. a) Jede Ecke muß zu mindestens einer Ecke einer anderen Farbklasse benachbart sein, andernfalls würde es eine zweite Färbung von G geben.
- b) Es seien F_1 und F_2 zwei Farbklassen. Angenommen der von $F_1 \cup F_2$ induzierte Untergraph U ist nicht zusammenhängend. Es seien U_1 und U_2 Zusammenhangskomponenten von U . Da jede Ecke zu mindestens einer Ecke einer anderen Farbklasse benachbart ist, enthalten U_1 und U_2 Ecken unterschiedlicher Farben. Vertauscht man die beiden Farben der Ecken in U_1 , so gelangt man zu einer zweiten Färbung von G . Dieser Widerspruch zeigt die Behauptung.
- c) Angenommen es gibt eine Menge M mit $c - 2$ Ecken, so daß der durch das Entfernen dieser Ecken entstehende Graph G' nicht mehr zusammenhängend ist. Dann gibt es zwei Farben f_1 und f_2 , mit denen keine der Ecken in M gefärbt ist. Es seien e_1 bzw. e_2 Ecken, die mit f_1 bzw. f_2 gefärbt sind. Nach Teil b) gibt es zwischen diesen Ecken einen Pfad P , dessen Ecken nur mit f_1 und f_2 gefärbt sind. Somit liegen alle mit f_1 bzw. f_2 gefärbten Ecken in einer einzigen Zusammenhangskomponente G_1 von G' . Ändert man nun die Färbung einer Ecke in $G' \setminus G_1$ in f_1 um, so erhält man eine zweite Färbung von G . Dieser Widerspruch zeigt die Behauptung.

34. Es sei Z eine Zuordnung von G mit t Kanten, Z_1 die Menge der Ecken aus E_1 , welche zu Kanten aus Z inzident sind, und $T \subseteq E_1$. Dann ist

$$|E_1| \geq |T \cup Z_1| = |T| + |Z_1| - |T \cap Z_1|.$$

Wegen $|N(T)| \geq |T \cap Z_1|$ und $|Z_1| = t$ folgt:

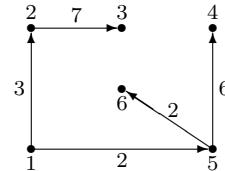
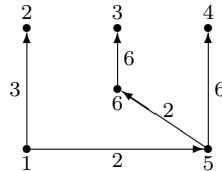
$$|N(T)| \geq |T| + t - |E_1|$$

Sei nun umgekehrt $|N(T)| \geq |T| + t - |E_1|$ für jede Teilmenge T von E_1 . Man betrachte das zugehörige Netzwerk N_G und einen maximalen binären Fluß f . Wie in dem Beweis des Satzes von Hall zeigt man nun $|f| = \kappa(X, \bar{X}) = |E_1 \setminus X| + |N(X \cap E_1)|$. Nach Voraussetzung gilt nun $|f| \geq |E_1 \setminus X| + |X \cap E_1| + t - |E_1| = t$. Sei nun Z die Menge aller Kanten aus G , für die der Fluß f durch die entsprechende Kante in N_G gerade 1 ist. Da jede Ecke aus E_1 in N_G den Eingrad 1 und jede Ecke aus E_2 in N_G den Ausgrad 1 hat, folgt aus der Flußerhaltungsbedingung, daß Z eine Zuordnung mit $|f| \geq t$ Kanten ist.

B.8 Kapitel 8

1. Es sei S die Menge der von der Startecke s aus erreichbaren Ecken und G_S der von S induzierte Untergraph von G . Nach Voraussetzung hat G_S die Eigenschaft (*). Eine Anwendung des Algorithmus von Moore und Ford auf G verwendet nur Ecken und Kanten aus G_S , d.h. beide Anwendungen sind identisch. Somit arbeitet der Algorithmus auch auf G korrekt.

2.



3. Der Graph besitzt keine geschlossenen Wege. Mit dem in Aufgabe 8 beschriebenen Verfahren können die kürzesten Wege in linearer Zeit bestimmt werden.
4. a) Der kürzeste Weg von e durch eine vorgegebene dritte Ecke v nach f besteht aus dem kürzesten Weg von e nach v gefolgt von dem kürzesten Weg von v nach f .
- b) Für $i, j = 1, \dots, l$ bestimme die kürzesten Wege von e nach e_i , e_i nach e_j und von e_i nach f (insgesamt $l^2 + l$ Wege). Danach bestimme man unter allen $l!$ Reihenfolgen der Ecken e_1, \dots, e_l den kürzesten der Wege $e \rightarrow e_{i_1} \rightarrow \dots \rightarrow e_{i_l} \rightarrow f$. Hierfür können die in Abschnitt 9.6 beschriebenen Verfahren wie *Branch-and-bound* oder *dynamisches Programmieren* verwendet werden.
5. Es sei B ein minimal aufspannender Baum von G , (e, f) eine der $n - 1$ Kanten von B und W ein kürzester Weg von e nach f in G . Wäre die Länge von W kleiner als die Länge von (e, f) , so würde dies zu einem aufspannenden Baum mit geringeren Kosten als B führen. Somit ist (e, f) der kürzeste Weg von e nach f in G .

6. Die folgende Abbildung zeigt den entstehenden kW-Baum und die kürzesten Entfernungen zur Startecke.



7. Die Prozedur `kürzesteWege` aus Abbildung 8.5 muß wie folgt geändert werden.

```

while W ≠ ∅ do begin
    i := W.entfernen;
    if not W.enthalten(Vorgänger[i]) then
        for jeden Nachfolger j von i do
            verkürzeW(i,j);
    end

```

Es sei i die erste Ecke in der Warteschlange und j der momentane Vorgänger von i im kW-Baum. Dann wurde j nach i in die Warteschlange eingefügt, d.h. der aktuelle Wert von $D[j]$ ist kleiner als zu dem Zeitpunkt, als $D[i]$ zuletzt aktualisiert wurde. Somit ist der aktuelle Wert von $D[i]$ zu hoch. Spätestens bei der Abarbeitung von j wird der Wert von $D[i]$ erniedrigt und i erneut in die Warteschlange eingefügt. Somit ist es überflüssig, Ecke i vor ihrem aktuellen Vorgänger j zu bearbeiten. Die Korrektheit dieser Variante des Algorithmus von Moore und Ford ergibt sich aus dieser Beobachtung und der Korrektheit der Orginalversion. Diese Variante hat in vielen Fällen eine geringere Laufzeit. Die *worst case* Komplexität bleibt aber unverändert. Dazu betrachte man die in der Lösung von Aufgabe 24 angegebene Folge von Graphen. Für diese ergibt sich keine Verbesserung, d.h. die Anzahl der Durchläufe der `while`-Schleife ist weiterhin $O(n^2)$.

8. Die Ecken eines gerichteten, kreisfreien Graphen seien mit ihren topologischen Sortierungsnummern numeriert. Der folgende Algorithmus bestimmt einen kW-Baum für eine beliebige Ecke.

```

procedure kWbaum (G : K-G-Graph; start : Integer);
var i,j : Integer;
begin
    initkW(start);
    for i := start to n-1 do
        for jeden Nachbar j von i do
            verkürze(i,j);
    end

```

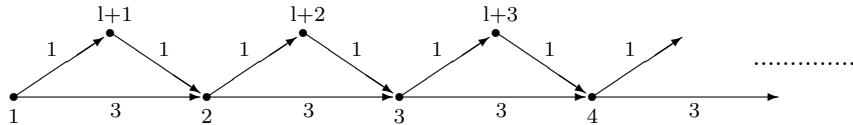
Ohne Einschränkung der Allgemeinheit kann $start = 1$ angenommen werden. Die Korrektheit dieser Prozedur ergibt sich aus folgender Aussage: Nach dem Ende des i -ten Durchlaufs gilt $D[j] = d(start, j)$ für $j = 1, \dots, i + 1$. Diese Aussage wird mit Hilfe von vollständiger Induktion nach i bewiesen. Zu jedem Zeitpunkt gilt $D[j] \geq d(start, j)$. Da Ecke 2 nur einen Vorgänger hat, gilt nach dem ersten

Durchlauf $D[j] = d(start, j)$ für $j = 1, 2$. Sei nun $i > 1$. Nach dem $i - 1$ -ten Durchgang werden die Werte für $D[1], \dots, D[i]$ nicht mehr geändert. Es muß also nur gezeigt werden, daß nach dem i -ten Durchgang $D[i + 1] = d(start, i + 1)$ gilt. Es sei j der Vorgänger von $i + 1$ auf einem kürzesten Weg von 1 nach $i + 1$. Dann gilt $j < i + 1$ bzw. $j \leq i$ und somit war $D[j] = d(start, j)$ vor dem j -ten Durchgang. Nach dem j -ten Durchgang gilt:

$$D[i + 1] \leq D[j] + B[j, i + 1] = d(start, i + 1)$$

Wegen $d(start, i + 1) \leq D[i + 1]$ gilt also $D[i + 1] = d(start, i + 1)$ schon nach dem j -ten und somit erst recht nach dem i -ten Durchgang.

9. Es sei n eine ungerade Zahl und $l = (n + 1)/2$. Der unten dargestellte kantenbewertete gerichtete Graph mit n Ecken und $3(n - 1)/2$ Kanten ist kreisfrei. Bei der Anwendung des Algorithmus von Moore und Ford auf diesen Graph werden die Nachfolger jeder Ecke in der Reihenfolge aufsteigender Eckennummern betrachtet. Für $j = 1, \dots, l - 1$ gilt: Die Ecken j und $l + j$ werden j -mal in die Warteschlange eingefügt. Somit ist die Laufzeit des Algorithmus für diesen Graph $O(n^2)$.



10. Die folgende Tabelle zeigt die Werte von B, Min, D und Vorgänger für die einzelnen Schritte.

B	Min	D								Vorgänger							
		1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
3		∞	∞	0	∞	∞	∞	∞	∞	0	0	0	0	0	0	0	0
2,4	3	∞	3	0	2	∞	∞	∞	∞	0	3	0	3	0	0	0	0
2,1,5	4	6	3	0	2	10	∞	∞	∞	4	3	0	3	4	0	0	0
1,5,8	2	4	3	0	2	10	∞	∞	7	2	3	0	3	4	0	0	2
5,8	1	4	3	0	2	7	∞	∞	7	2	3	0	3	1	0	0	2
8,6,7	5	4	3	0	2	7	13	12	7	2	3	0	3	1	5	5	2
6,7	8	4	3	0	2	7	13	12	7	2	3	0	3	1	5	5	2
6	7	4	3	0	2	7	13	12	7	2	3	0	3	1	5	5	2
	6	4	3	0	2	7	13	12	7	2	3	0	3	1	5	5	2

11. Der folgende Algorithmus basiert auf der Tiefensuche und hat eine Laufzeit von $O(m)$. Eine neue Ecke i wird nur dann besucht, wenn Sie nicht schon als besucht markiert ist und wenn die Entfernung im Tiefensuchebaum von der Startecke e mit $d(e, i)$ übereinstimmt. Zum Beweis der Korrektheit der Prozedur kWBaum genügt es zu zeigen, daß alle Ecken erreicht werden. Es sei v eine beliebige Ecke von G und W der Weg von der Startecke zu v in einem beliebigen kW-Baum (da G die Eigenschaft (*) hat, muß es einen kW-Baum geben). Angenommen v ist nicht in dem erzeugten Baum B enthalten. Es sei j die erste Ecke auf W , welche

nicht in B ist. Dann ist der Vorgänger i von j auf W in B enthalten und es gilt $d(e, j) = d(e, i) + b(i, j)$. Beim Besuch von i durch **aufbauen** wird dann j besucht. Dieser Widerspruch zeigt, daß B ein kW-Baum ist.

```

var Besucht : array[1..max] of Boolean;
      Vorgänger : array[1..max] of Integer;
procedure aufbauen(i,e : Integer);
var j : Integer;
begin
    Besucht[i] := true;
    for jeden Nachbar j von i do
        if Besucht[j] = false and d(e,j) = d(e,i) + b(i,j) then begin
            Vorgänger[j] := i;
            aufbauen(j,e);
        end
    end
end

procedure kWBaum(G : B-Graph; e : Integer);
begin
    Initialisiere Besucht mit false und Vorgänger mit 0;
    aufbauen(e,e);
end

```

12. Da G die Eigenschaft (*) erfüllt, gilt das Optimalitätsprinzip. Dann ist der Teilweg W_1 von e_1 nach f auf W ein kürzester Weg von e_1 nach f . Für W_1 gilt wieder das Optimalitätsprinzip, somit gilt $L(\overline{W}) = d(e_1, f_1)$, d.h. \overline{W} ist ein kürzester Weg.
13. Es seien a und b positive reelle Zahlen. Dann gilt: $ab = 1/e^{-\log a - \log b}$. Da die e -Funktion monoton steigend ist, ist das Produkt ab um so größer, um so kleiner $-\log a - \log b$ ist. Für $a \in [0, 1]$ ist $-\log a > 0$. Ändert man die Bewertung jeder Kante von b_{ij} auf $-\log b_{ij}$, so kann der Algorithmus von Dijkstra angewendet werden. Dann entsprechen die kürzesten Wege bezüglich der neuen Bewertung den Wegen, bei den das Produkt der einzelnen Wahrscheinlichkeiten maximal ist. Der Übertragungsweg von Sender 1 zum Empfänger 5 mit der größten Wahrscheinlichkeit einer korrekten Übertragung ist 1, 8, 9, 3, 4, 5. Die Übertragungswahrscheinlichkeit ist 0.40824.
14. Während der Ausführung des Algorithmus von Dijkstra gilt:

$$W = \{D[i] \mid i \in B\} \subseteq \{s, s+1, s+2, \dots, s+C\}$$

wobei $s = \min W$ ist. Diese Eigenschaft ist zu Beginn des Algorithmus erfüllt und bleibt auch erhalten, wenn eine Ecke i mit minimalen Wert entfernt wird. Danach werden die Nachbarn j von i bearbeitet und es gilt $D[i] \leq D[i] + B[i, j]$ bzw. $s \leq D[j] \leq s + C$. Also bleibt die obige Eigenschaft stets erhalten. Zu jedem Zeitpunkt gilt:

$$\{D[i] \bmod (C+1) \mid i \in B\} \subseteq \{0, \dots, C\}$$

Dies macht man sich bei der Speicherung der Werte $D[i]$ zunutze. Es werden $C+1$ Fächer vorgesehen. In das i -te Fach werden die Werte abgespeichert, welche

modulo $C + 1$ gleich i sind. Ein Fach wird durch eine doppelt verkettete Liste implementiert. In einem Feld FA der Länge $C + 1$ werden die Zeiger auf die Fächer verwaltet. Damit das Ändern des Wertes einer Ecke in konstanter Zeit erfolgen kann, wird für jede Ecke ein Zeiger zu dem zugehörigen Element im entsprechenden Fach abgespeichert. Hierzu dient das Feld E der Länge n . Folgende Datenstrukturen werden verwendet.

```

Fach = record
    vorgänger, nachfolger : zeiger Fach;
    nummer : Integer;
end;
Ecke = record
    wertZeiger : zeiger Fach;
    wert : Integer;
end;
E : array[1..n] of Ecke;
FA : array[0..C] of zeiger Fach;

```

Mit Hilfe dieser Datenstrukturen kann das Einfügen einer neuen Ecke in konstanter Zeit realisiert werden. Jede Ecke wird genau einmal betrachtet. Da die Bearbeitung eines Nachbarn in konstanter Zeit erfolgt, ist der Gesamtaufwand für alle Änderungsoperationen $O(m)$. Es bleibt noch zu zeigen, wie die Ecken mit minimalem Wert gefunden werden.

Die einfachste Variante verwaltet den Index \min des Faches mit dem kleinsten Wert. Da die minimalen Werte im Laufe des Algorithmus immer größer werden, kann dieser Index leicht aktualisiert werden:

```

while FA[min] = leere Liste do
    min := min + 1 mod (C+1);

```

Der Aufwand hierfür ist $O(L)$, wobei L die Länge des längsten aller kürzesten Wege ist. Wegen $L \leq (n - 1)C$ ist der Gesamtaufwand für den Algorithmus von Dijkstra $O(m + nC)$. Man beachte, daß der Specheraufwand jetzt auch von C abhängt.

Eine andere Variante besteht darin, die Indices des Feldes FA , welche nichtleere Listen enthalten, in einem Heap zu verwalten. Dann sind zu jedem Zeitpunkt maximal $C + 1$ Elemente im Heap. Jede einzelne Heap-Operation hat den Aufwand $O(\log C)$. Der Gesamtaufwand für den Algorithmus von Dijkstra ist bei dieser Realisierung $O((m + n) \log C)$.

15. Ohne Einschränkung der Allgemeinheit kann man annehmen, daß alle Kanten die Bewertung 1 haben (vergleichen Sie Aufgabe 23). Eine in Abschnitt 8.2 bewiesene Folgerung des Optimalitätsprinzip ist: Für alle Kanten (e, f) gilt:

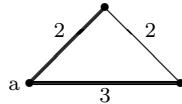
$$d(s, f) \leq d(s, e) + B[e, f]$$

Die Aussage der Aufgabe folgt nun aus Aufgabe 31 in Kapitel 4.

16. Die folgende Tabelle zeigt die Werte von D und Vorgänger für einen kW-Baum mit Startecke 1.

Startecke	D					Vorgänger				
	1	2	3	4	5	1	2	3	4	5
1	0	3	6	3	2	0	1	2	3	4

17. Der Korrektheitsbeweis für diese Version des Algorithmus von Moore und Ford folgt dem Orginalbeweis aus Abschnitt 8.3. Untersuchungen haben gezeigt, daß diese Variante in vielen Fällen eine geringere Laufzeit aufweist. Wendet man den neuen Algorithmus auf die in der Lösung von Aufgabe 24 beschriebenen Graphen an, so stellt man allerdings fest, daß die Anzahl der Ausführungen der `while`-Schleife unverändert ist.
18. Man vergleiche hierzu die Lösung von Aufgabe 14. Sind die Kantenbewertungen reelle Zahlen, so haben die Ecken innerhalb eines Faches nicht unbedingt die gleichen Werte. Dies erhöht die Zugriffszeit und die in Aufgabe 14 angegebenen Laufzeiten werden nicht erreicht. Entscheidend ist die Verteilung der Ecken auf die Fächer. Je gleichmäßiger die Verteilung ist, desto größer ist der Vorteil von Bucket-Sort.
19. Es wird ein neuer Graph G' mit gleicher Ecken- und Kantenmenge gebildet. Die Kanten (e, f) von G' tragen die Bewertung $B'[e, f] = B[e, f] + (B[e] + B[f])/2$. Genau dann ist W ein kürzester Weg von s nach z in G' bezüglich B' , wenn W ein kürzester Weg von s nach z in G bezüglich B ist. Die Länge von W in G ist gleich $L'(W) - (B[s] + B[z])/2$, wobei $L'(W)$ die Länge von W in G' ist.
20. Die fett gezeichneten Kanten bilden einen kW-Baum für Ecke a , aber keinen minimal aufspannenden Baum.



21. a) Distanz- und Vorgängermatrix des Graphen aus Übungsaufgabe 6:

$$D = \begin{pmatrix} 0 & 3 & 6 & 10 & 8 & 6 \\ \infty & 0 & \infty & \infty & \infty & \infty \\ \infty & -3 & 0 & 5 & 3 & 1 \\ \infty & -7 & -4 & 0 & -1 & -3 \\ \infty & -5 & -2 & 2 & 0 & -1 \\ \infty & -3 & 0 & 4 & 2 & 0 \end{pmatrix} \quad V = \begin{pmatrix} 0 & 3 & 4 & 6 & 6 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 5 & 6 & 3 \\ 0 & 3 & 4 & 0 & 6 & 3 \\ 0 & 3 & 4 & 5 & 0 & 3 \\ 0 & 2 & 4 & 6 & 6 & 0 \end{pmatrix}$$

- b) Die Werte für die Kantenbewertung B' sind: $b_{12} = 11$, $b_{15} = 10$, $b_{16} = 9$, $b_{32} = 0$, $b_{36} = 0$, $b_{43} = 0$, $b_{54} = 1$, $b_{64} = 1$, $b_{65} = 0$.
22. Zunächst wird der Abstand zwischen allen Paaren von Orten bestimmt. Danach wird für jeden Ort die weiteste Entfernung bestimmt (größter Wert in der entsprechenden Zeile der Distanzmatrix) und der Ort mit der kleinsten weitesten

Entfernung ausgewählt. In diesem Fall ist es Ort 2 mit einer maximalen Entfernung von 180. Im folgenden ist die Distanzmatrix dargestellt. Die weitesten Wegstrecken für die einzelnen Orte sind eingerahmt.

$$D = \begin{pmatrix} 0 & 100 & 250 & 280 & 140 & \boxed{260} \\ 100 & 0 & 150 & \boxed{180} & 40 & 160 \\ \boxed{250} & 150 & 0 & 30 & 190 & 70 \\ \boxed{280} & 180 & 30 & 0 & 220 & 100 \\ 140 & 40 & 190 & \boxed{220} & 0 & 120 \\ \boxed{260} & 160 & 70 & 100 & 120 & 0 \end{pmatrix}$$

23. Es sei W ein Weg bestehend aus l Kanten, $L(W)$ bzw. $L'(W)$ bezeichne die Länge von W vor bzw. nach der Erhöhung um C . Im ersten Fall gilt $L'(W) = L(W) + lC$ und im zweiten Fall $L'(W) = CL(W)$. Im zweiten Fall bleiben die kürzesten Wege die gleichen.
24. Der Graph hat die Eigenschaft (*): Jeder geschlossene Weg hat mindestens eine Kante mit Bewertung 64, da alle Kanten (i, j) mit $i < j$ die Bewertung 64 haben. Der kW-Baum mit Startecke 1 ist der Weg 1, 6, 5, 4, 3, 2 und die Längen der kürzesten Wege sind 64, 47, 38, 33, 30. Die **while**-Schleife wird 16 mal ausgeführt.
25. Die Werte für die Kantenbewertung B' sind: $b_{12} = 0$, $b_{14} = 4$, $b_{23} = 2$, $b_{31} = 1$, $b_{34} = 0$, $b_{45} = 0$, $b_{51} = 0$.
26. Der Graph hat nicht die Eigenschaft (*), der geschlossene Weg 6, 5, 3 hat die Länge -1 . Wendet man die Prozedur **floyd** an, so wird der Diagonaleintrag $D[6, 6]$ im vorletzten Durchgang negativ.
27. Mit Hilfe der Tiefensuche wird für jede Ecke u die Länge $d_B(e, u)$ des kürzesten Weges von e nach u in B bestimmt (Aufwand $O(n)$). Nach den Ergebnissen aus Abschnitt 8.2 ist B genau dann ein kW-Baum, wenn für jede Kante (u, v) von G die Ungleichungen $d_B(e, u) + B[u, v] \geq d_B(e, v)$ und $d_B(e, v) + B[u, v] \geq d_B(e, u)$ erfüllt sind. Dies kann mit Aufwand $O(m)$ überprüft werden.
28. Falls es einen geschlossenen Weg W mit negativer Länge gibt, so gibt es eine Ecke **start** auf W , so daß alle in **start** startenden Teilwege von W negative Länge haben. Die Prozedur **negativerKreis** überprüft die Existenz eines solchen geschlossenen Weges für die Ecke **start**. Wird ein solcher Weg gefunden, so wird das Programm beendet. Der Graph hat dann nicht die Eigenschaft (*). Im Hauptprogramm wird diese Prozedur für jede Ecke aufgerufen. Gibt es für keine Ecke des Graphen einen solchen Weg, so erfüllt der Graph die Eigenschaft (*).

```

for jede Ecke i do
    negativerKreis(i, i, 0);
    Exit('Graph hat die Eigenschaft (*)')

procedure negativerKreis(start, akt, länge : Integer);
var j : Integer;
begin

```

```

for jeden Nachfolger j von akt do
  if läng + bew(akt,j) < 0 then
    if j = start then
      Exit('Graph hat nicht die Eigenschaft (*)')
    else
      negativerKreis(start, j, läng + bew(akt,j));
  end

```

29. Die folgende Tabelle zeigt die Werte von D und Vorgänger für einen IW-Baum (*längste Wege Baum*) mit Startecke 1.

Startecke	D						Vorgänger					
	1	2	3	4	5	6	1	2	3	4	5	6
	0	1	-9	-3	3	-7	0	1	4	2	4	3

30. Die letzte **if**-Anweisung im Programm aus Abbildung 8.23 muß wie folgt abgeändert werden.

```

if max{D[i,j],D[j,k]} < ∞ and D[i,k] > min{D[i,j],D[j,k]} then begin
  D[i,k] := min{D[i,j],D[j,k]};
  V[i,k] := V[j,k];
end

```

31. Da f_1 und f_2 konsistent sind, gilt $0 \leq f_i(u) \leq B[u, v] + f_i(v)$ für $i = 1, 2$ und jede Kante (u, v) . Multipliziert man die erste Ungleichung mit a und die zweite mit b und addiert diese dann, so erhält man:

$$0 \leq af_1(u) + bf_2(u) \leq (a+b)B[u, v] + af_1(v) + bf_2(v).$$

Dividiert man diese Ungleichung durch $a+b$, so folgt die Konsistenz von $(af_1 + bf_2)/(a+b)$.

32. Für einen beliebigen Brettzustand b und einen Zielzustand z sei

$f_1(b)$ die Anzahl der Plättchen, welche in b eine andere Position als in z haben;
 $f_2(b)$ die Summe der für jedes Plättchen von b (ohne Beachtung der anderen Plättchen) mindestens notwendigen Verschiebungen, um die Position in z zu erreichen.

Für die in Abbildung 4.30 dargestellte Startstellung s gilt $f_1(s) = 8$ und $f_2(s) = 13$. Sowohl f_1 als auch f_2 sind zulässig. Beide Schätzfunktionen können mit Hilfe von Metriken auf dem Raum $Z \times Z$ definiert werden: für a, b aus $Z \times Z$ sei $d_1(a, b) = 0$, falls $a = b$, und 1, falls $a \neq b$ und $d_2(a, b) = |a_x - b_x| + |a_y - b_y|$. Bezeichnet man mit $P_a(i)$ die Position von Plättchen i im Zustand a , so gilt

$$f_i(a) = d_i(P_a(1), P_z(1)) + \dots + d_i(P_a(8), P_z(8))$$

für $i = 1, 2$. Aus der Dreiecksungleichung für d_1 und d_2 folgt:

$$f_i(a) - f_i(b) = \sum_{j=1}^8 d_i(P_a(j), P_z(j)) - d_i(P_b(j), P_z(j)) \leq \sum_{j=1}^8 d_i(P_a(j), P_b(j))$$

Da die rechte Seite dieser Ungleichung höchstens so groß ist, wie die minimale Anzahl von Verschiebungen, um Zustand b von Zustand a aus zu erreichen, sind f_1 und f_2 konsistent. Wegen $f_1(a) \leq f_2(a)$ wird das Ziel mit Hilfe der zweiten Schätzfunktion im Allgemeinen schneller gefunden.

33. Startend mit dem trivialen Fluß f_0 werden mit Hilfe von Erweiterungswegen minimaler Kosten neue kostenminimale Flüsse f_i bestimmt. Dazu beachte man, daß der triviale Fluß ein Fluß mit Wert 0 und minimalen Kosten ist. Einen Erweiterungsweg mit minimalen Kosten findet man mit dem Algorithmus von Moore und Ford. Dieser Algorithmus ist anwendbar, da es in den Graphen G_{f_i} nach Teil a) des Satzes aus Abschnitt 6.6 keine geschlossenen Wege negativer Länge gibt. Das Verfahren wird beendet, sobald Ecke s in G_{f_i} nicht mehr von q aus erreichbar ist. Dann liegt ein maximaler Fluß mit minimalen Kosten vor. Da alle Kapazitäten ganzzahlig sind, gilt $|f_{i+1}| \geq |f_i| + 1$, d.h. der Algorithmus terminiert nach maximal $|f_{max}|$ Durchgängen. Unter Verwendung des Algorithmus von Moore und Ford ergibt sich eine Laufzeit von $O(|f_{max}|nm)$.
34. Der in Abbildung 8.8 dargestellte Algorithmus von Dijkstra expandiert immer die Ecke aus dem kW-Baum, welche aktuell den kürzesten Abstand zur Startecke hat. Zur Lösung der gestellten Aufgabe muß die Ordnung auf den Ecken des kW-Baumes neu definiert werden. Ist die Entfernung zweier Ecken zur Startecke gleich, so wird die Ecke, bei der der Weg von der Startecke weniger Kanten enthält, als kleiner angesehen. Hierzu wird neben der eigentlichen Entfernung zur Startecke auch die Anzahl der Kanten des Weges im kW-Baum in einem Feld `kantenAnzahl` gespeichert. Dieses Feld wird mit ∞ initialisiert. Die Prozedur `verkürze` muß wie folgt geändert werden.

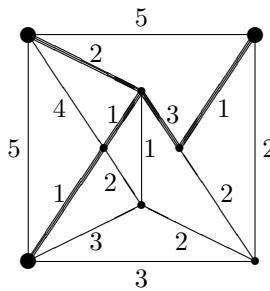
```

procedure verkürze (i,j : Integer);
begin
  if D[i] + B[i,j] < D[j] then begin
    D[j] := D[i] + B[i,j];
    Vorgänger[j] := i;
    kantenAnzahl[j] := kantenAnzahl[i] + 1;
  end
  else
    if D[i] + B[i,j] = D[j] and
        kantenAnzahl[j] > kantenAnzahl[i] + 1 then begin
      Vorgänger[j] := i;
      kantenAnzahl[j] := kantenAnzahl[i] + 1;
    end
  end
end

```

Die Implementierung des Algorithmus von Dijkstra wird an zwei Stellen geändert. Die Auswahl einer Ecke aus B erfolgt gemäß der neuen Ordnung und beim Einfügen einer Ecke in B wird das Feld `kantenAnzahl` auf 1 gesetzt.

35. Die fett dargestellten Kanten bilden einen minimalen Steinerbaum mit 3 Steiner-ecken und Kosten 8.



B.9 Kapitel 9

1. a) Ohne Einschränkung der Allgemeinheit kann angenommen werden, daß nach der Zuordnung der Programme zu Prozessoren Prozessor P_1 die höchste Last hat. Bezeichne mit T die Gesamtaufzeit aller P_1 zugeordneter Programme. Das zuletzt an P_1 zugeordnete Programm habe die Laufzeit t_j . Da Programm j als letztes zugeordnet wurde, hat jeder Prozessor mindestens eine Laufzeit von $T - t_j$. Hieraus folgt:

$$\sum_{i=1}^n t_i \geq m(T - t_j) + t_j$$

Ferner ist $A(n) = T$ und somit gilt

$$OPT(n) \geq \frac{\sum_{i=1}^n t_i}{m} \geq (T - t_j) + \frac{t_j}{m} = A(n) - (1 - \frac{1}{m})t_j.$$

Wegen $t_j \leq OPT(n)$ folgt

$$A(n) \leq OPT(n) + (1 - \frac{1}{m})t_j \leq OPT(n)(2 - \frac{1}{m}).$$

Hieraus ergibt sich die Behauptung.

- b) Die Lasten der einzelnen Prozessoren werden in einem Heap verwaltet. In jedem Schritt wird der Prozessor mit der geringsten Last ausgewählt und die Last dieses Prozessors wird um die Laufzeit des aktuellen Programmes erhöht. Diese Operation hat den Aufwand $O(\log m)$. Für alle n Programme ergibt sich zusammen der Aufwand $O(n \log m)$.
- c) Es sei $n = m(m - 1) + 1$, die ersten $n - 1$ Programme haben die Laufzeit 1 und das letzte Programm die Laufzeit m . Hieraus folgt $A(n) = 2m - 1$ und $OPT(n) = m$. Somit ist $\mathcal{W}_A(n) \leq 2 - \frac{1}{m}$.
- d) Da das zuletzt zugeordnete Programm die kürzeste Laufzeit hat, gilt

$$t_j \leq \frac{\sum_{i=1}^n t_i}{n} \leq \frac{m}{n} OPT(n).$$

Hieraus folgt

$$\mathcal{W}_A(n) \leq 1 + \frac{m}{n}(1 - \frac{1}{m}).$$

Wegen $n > m$ wird also der Wirkungsgrad verbessert. Mit einer genaueren Analyse kann gezeigt werden, daß $\mathcal{W}_A(n) \leq \frac{4}{3} - \frac{1}{3m}$ gilt.

2. Aus den $(2n)!$ möglichen Reihenfolgen sind die zu bestimmen, die zu einer 2-Färbung führen. Im folgenden werden die Ecken auf der linken Seite mit E_1 und auf der rechten Seite mit E_2 bezeichnet. Sind die ersten beiden Ecken aus E_1 , so bekommen diese die Farbe 1. Daraufhin kann keine der Ecken aus E_2 die Farbe 1 mehr bekommen. Somit wird in diesem Fall eine 2-Färbung erzeugt. Die gleiche Aussage gilt für E_2 . Dies sind insgesamt $2n(n-1)(2n-2)!$ verschiedene Reihenfolgen. Sind die ersten beiden Ecken durch eine Kante verbunden und die dritte Ecke entweder aus der gleichen Menge wie die erste Ecke oder mit der ersten Ecke verbunden, so wird ebenfalls eine 2-Färbung erzeugt. Dies sind noch einmal $2n(n-1)(2n-3)(2n-3)!$ verschiedene Reihenfolgen. In allen anderen Fällen bekommen eine Ecke aus E_1 und eine Ecke aus E_2 die gleiche Farbe und dadurch werden mindestens 3 Farben vergeben. Somit ist die Wahrscheinlichkeit, daß der Greedy-Algorithmus eine 2-Färbung erzeugt, gleich

$$\frac{2n(n-1)(2n-2)! + 2n(n-1)(2n-3)(2n-3)!}{(2n)!} = \frac{4n-5}{4n-2}.$$

Die Wahrscheinlichkeit, daß der Algorithmus für diesen Graphen eine optimale Färbung erzielt, strebt also mit wachsendem n gegen 1. Für $n = 20$ liegt diese Wahrscheinlichkeit schon über 96%.

3. Numeriere die Ecken des Graphen im Uhrzeigersinn beginnend bei der Ecke oben links mit der Nummer 1. Für die Reihenfolge 1, 4, 6, 3, 2, 5 vergibt der Greedy-Algorithmus vier Farben und für die Reihenfolge 1, 5, 3, 6, 2, 4 zwei Farben.

4.

Ecke	1	2	3	4	5	6
Greedy-Algorithmus	1	2	1	3	2	4
Johnson Algorithmus	1	1	2	2	3	4
Optimale Färbung	1	2	3	3	2	1

5. Mittels vollständiger Induktion nach i wird gezeigt, daß Ecken aus $F_{\pi(i)}$ eine Farbe mit der Nummer i oder kleiner bekommen. Hieraus folgt dann direkt die Behauptung. Die Ecken aus $F_{\pi(i)}$ haben bezüglich f alle die gleiche Farbe, d.h. sie sind nicht benachbart. Somit bekommen die Ecken aus $F_{\pi(1)}$ alle die Farbe 1 zugeordnet. Angenommen eine Ecke a aus $F_{\pi(i)}$ bekommt die Farbe $i+1$ zugeordnet. Somit muss a zu einer Ecke b mit Farbe i benachbart sein. Nach Induktionsvoraussetzung kann b nicht in $F_{\pi(1)}, \dots, F_{\pi(i-1)}$ liegen. Also liegen a und b in $F_{\pi(i)}$. Dies bedeutet aber, daß a und b nicht benachbart sind. Dieser Widerspruch zeigt die Behauptung.

Wendet man den Greedy-Algorithmus in der angegebenen Reihenfolge auf die Ecken des Graphen aus Aufgabe 4 an, so erhält man eine Färbung mit vier Farben.

Ordnet man die Ecken nach absteigenden Farbklassen um, d.h. man betrachtet die Ecken in der Reihenfolge $(6, 4, 5, 2, 3, 1)$, dann erhält man hingegen eine Färbung mit drei Farben.

Es sei G ein bipartiter Graph mit Eckenmenge $E = \{a_1, \dots, a_n\} \cup \{b_1, \dots, b_n\}$ und Kantenmenge $K = \{(a_i, b_j) \mid 1 \leq i, j \leq n, i \neq j\}$. Im ungünstigsten Fall ordnet der Greedy-Algorithmus für jedes i den Ecken a_i und b_i die gleiche Farbe zu. Dann vergibt der Algorithmus n Farben, obwohl 2 ausreichen. In diesem Fall wird für jede Permutation der Farbklassen ebenfalls eine Färbung mit n Farben erzeugt.

6. a) Der Greedy-Algorithmus betrachtet bei der Bestimmung der Farbe einer Ecke e die schon gefärbten Nachbarn und wählt dann die kleinste Farbnummer aus, welche noch nicht für diese Nachbarn verwendet wurde. Zur Färbung der i -ten Ecke e_i werden $\min(i - 1, g(e_i))$ Nachbarn betrachtet, d.h. die Farbnummer von e_i ist maximal $\min(i, g(e_i) + 1)$. Eine Obergrenze für die Anzahl der durch den Greedy-Algorithmus vergebenen Farben ist:

$$\max \{\min(i, g(e_i) + 1) \mid i = 1, \dots, n\}$$

- b) Die Ecken der folgenden Menge liegen innerhalb der ersten k Farbklassen:

$$\{e_i \mid g(e_i) < k\} \cup \{e_1, \dots, e_k\}$$

Es gilt also, diese Menge für festes k möglichst groß zu machen. Dies erreicht man, indem Ecken mit kleinem Eckengrad über die erste Teilmenge und Ecken mit großem Eckengrad über die zweite Teilmenge abgedeckt werden. Hierzu betrachtet man beispielsweise die Ecken in der Reihenfolge absteigender Eckengrade.

- c) Alle Ecken des Graphen H_i haben den Eckengrad i . Betrachtet man die Ecken der Graphen H_i in der Reihenfolge absteigender Eckennummern (d.h. $2i - 1, 2i - 2, \dots, 1$), so vergibt der Greedy-Algorithmus $i + 1 = n/2$ Farben. Somit gilt für den Wirkungsgrad: $\mathcal{W}_A(n) \geq n/4$ und es ist $\mathcal{W}_A^\infty = \infty$.
7. a) Es sei G ein Graph, der entsteht, wenn die Wurzel eines Baumes der Höhe 1 und $s + 1$ Ecken mit einer beliebigen Ecke des vollständigen Graphen K_s verbunden wird. G hat $2s + 1$ Ecken und $\omega(G) = s$. Die Wurzel des Baumes hat in G maximalen Eckengrad. Entfernt man diese und alle nicht zu ihr benachbarten Ecken, so verbleiben $s + 1$ isolierte Ecken. Somit erzeugt A_1 eine Clique der Größe 2 und es gilt $OPT(G)/A_1(G) = s/2 = (n - 1)/4$ bzw. $\mathcal{W}_{A_1}^\infty = \infty$.
- b) Es sei G ein Graph G , der entsteht, wenn eine beliebige Ecke aus dem vollständigen Graphen K_s mit einer beliebigen Ecke aus dem vollständig bipartiten Graphen $K_{s,s}$ verbunden wird. G hat $3s$ Ecken und $\omega(G) = s$. In K_s gibt es $s - 1$ Ecken mit Eckengrad $s - 1$, diese werden von A_2 als Erstes aus G entfernt. Somit erzeugt A_2 eine Clique der Größe 2 und es gilt $OPT(G)/A_2(G) = s/2 = n/6$ bzw. $\mathcal{W}_{A_2}^\infty = \infty$.

8. In Kapitel 5 wurde gezeigt, daß die Größe einer maximalen Clique $\omega(G)$ gleich der Größe einer maximalen unabhängigen Menge $\alpha(\bar{G})$ des Komplementes des Graphen ist. Somit ergibt sich aus jedem approximativen Algorithmus für das Optimierungsproblem von Cliques direkt ein approximativer Algorithmus für das Optimierungsproblem der unabhängigen Menge mit gleichem Wirkungsgrad. Die in der Aufgabe gemachte Aussage folgt nun direkt aus dem letzten Satz aus Abschnitt 9.4.
9. Ein Erweiterungsweg bezüglich einer Zuordnung Z ist ein Weg, der bei einer nicht zugeordneten Ecke beginnt und endet und bei dem jede zweite Kante nicht in Z enthalten ist. Mit Hilfe eines Erweiterungsweges W kann eine gegebene Zuordnung vergrößert werden. Hierzu werden aus Z alle Kanten entfernt, welche auf W liegen. Die restlichen Kanten von W werden in Z eingefügt. Die so entstandene Zuordnung enthält eine Kante mehr als die ursprüngliche Zuordnung. Zum Beweis der Aussage in der Aufgabe wird der Graph G' betrachtet. Da die Kanten von G' aus zwei Zuordnungen stammen, ist der Eckengrad jeder Ecke von G' kleiner oder gleich 2. Es sei H eine Zusammenhangskomponente von G' . H ist entweder eine isolierte Ecke, ein einfacher, geschlossener Weg mit der gleichen Anzahl von Kanten aus Z_1 und Z_2 oder ein offener einfacher Weg, dessen Kanten abwechselnd aus Z_1 und Z_2 stammen. Komponenten vom letzten Typ mit einer ungeraden Anzahl von Kanten sind entweder Erweiterungswäge bezüglich Z_1 oder Z_2 (je nachdem ob mehr Kanten aus Z_2 oder Z_1 stammen). Da alle Kanten aus Z_1 und Z_2 auf die Zusammenhangskomponenten verteilt sind, muss es $|Z_2| - |Z_1|$ Komponenten vom letzten Typ geben, bei denen die Mehrzahl der Kanten aus Z_2 kommt. Dies sind alles eckendisjunkte Erweiterungswäge bezüglich Z_1 .
10. Die Funktion **greedy-Zuordnung** erzeugt offensichtlich eine nicht erweiterbare Zuordnung Z von G . Es sei M eine maximale Zuordnung von G . Dann gibt es nach der letzten Aufgabe $|M| - |Z|$ eckendisjunkte Erweiterungswäge bezüglich Z . Nach Konstruktion enthält jeder dieser Erweiterungswäge mindestens zwei Kanten aus M . Da die Wege eckendisjunkt sind, ist $|M| - |Z| \leq |M|/2$ bzw. $2|Z| \geq |M|$. Hieraus folgt, daß der Wirkungsgrad des angegebenen Algorithmus kleiner oder gleich 2 ist.

Es sei l eine gerade Zahl und G_l ein bipartiter Graph mit Eckenmenge

$$E = \{a_1, \dots, a_l\} \cup \{b_1, \dots, b_l\}$$

und Kantenmenge

$$\begin{aligned} K = & \{(a_i, b_i) \mid i = 1, \dots, l\} \cup \{(a_i, b_{i+1}) \mid i = 1, 3, 5, \dots, l-1\} \\ & \cup \{(a_i, b_{i-1}) \mid i = 3, 5, \dots, l-1\}. \end{aligned}$$

Die Kanten $\{(a_i, b_i) \mid i = 1, \dots, l\}$ bilden eine maximale Zuordnung mit l Kanten. Die Kanten $\{(a_i, b_{i+1}) \mid i = 1, 3, 5, \dots, l-1\}$ bilden eine Zuordnung, die durch die Funktion **greedy-Zuordnung** bestimmt wurden. Somit gilt

$$OPT(G_l)/A(G_l) = l/2l = 2$$

und der asymptotische Wirkungsgrad ist 2.

In der folgenden Implementierung von **greedy-Zuordnung** werden die Nachbarn jeder Ecke genau einmal betrachtet, deshalb ist die worst case Laufzeit gleich $O(n + m)$. Die Endenken der Kanten der Zuordnung können in linearer Zeit aus dem Feld **Zuordnung** entnommen werden.

```

Zuordnung : array[1..max] of Integer;
procedure greedy-Zuordnung(G : Graph);
var
    i,j : Integer;
begin
    Initialisiere Zuordnung mit 0;
    for jede Ecke i do
        if Zuordnung[i] = 0 then
            for jeden Nachbar i von j do
                if Zuordnung[j] = 0 then begin
                    Zuordnung[i] := j; Zuordnung[j] := i;
                    break;
                end
            end
    end
end

```

11. Der Algorithmus aus der letzten Aufgabe hat für das beschriebene Problem die worst-case Laufzeit $O(n + \bar{m})$, wobei \bar{m} die Anzahl der Kanten von \bar{G} ist. Der folgende Algorithmus ist ebenfalls ein Greedy-Algorithmus, d.h. die Aussage über den Wirkungsgrad bleibt gültig. Er verwendet eine verkettete Liste zur effizienten Iteration über die Menge der noch nicht zugeordneten Ecken.

```

Zuordnung : array[1..max] of Integer;
procedure greedy-Zuordnung(G : Graph);
var
    nachbarn : array[1..max] of Boolean;
    unbedeckt Liste;
    i,j : Integer;
begin
    Initialisiere Zuordnung mit 0;
    Initialisiere nachbarn mit false;
    Füge alle Ecken von G in unbedeckt ein;
    while L ≠ ∅ begin
        i = unbedeckt.entreneKopf();
        for jeden Nachbar j von i do
            nachbarn[j] := true;
        for jede Ecke j in L do
            if nachbarn[j] = true begin
                unbedeckt.entrene(j);
                Zuordnung[i] := j; Zuordnung[j] := i;
                break;
            end
        for jeden Nachbar j von i do
            nachbarn[j] := false;
    end
end

```

Für die Analyse der Laufzeit beachte man, daß jede der drei **for**-Schleifen innerhalb der **while**-Schleife den Aufwand $O(g(i))$ hat. Hieraus ergibt sich, daß der Algorithmus lineare Laufzeit $O(n + m)$ hat.

12. Nach Aufruf der Prozedur aus der letzten Aufgabe werden die z Kanten der Zuordnung Z der Reihe nach betrachtet. Gibt es für die Enden einer Kante (e, f) nicht zugeordnete Nachbarn e' und f' mit $e \neq f$, so wird (e, f) aus Z entfernt und die beiden neuen Kanten (e', e) und (f', f) werden in Z eingefügt. Man überzeugt sich schnell, daß nach Betrachtung der z Kanten jeder Erweiterungsweg bezüglich Z aus mindestens fünf Kanten besteht (drei davon gehören nicht zu Z). Die Laufzeit des Verfahrens bleibt weiterhin linear.

Es sei M eine maximale Zuordnung von G . Dann gibt es nach der vorletzten Aufgabe $|M| - |Z|$ eckendisjunkte Erweiterungswege bezüglich Z . Nach Konstruktion enthält jeder dieser Erweiterungswege mindestens drei Kanten aus M . Da die Wege eckendisjunkt sind, ist $|M| - |Z| \leq |M|/3$ bzw. $3|Z| \geq 2|M|$. Somit ist der Wirkungsgrad des Algorithmus kleiner oder gleich $3/2$.

Es sei l eine durch 3 teilbare Zahl und G_l ein bipartiter Graph mit Eckenmenge

$$E = \{a_1, \dots, a_l\} \cup \{b_1, \dots, b_l\}$$

und Kantenmenge

$$\begin{aligned} K = & \{(a_i, b_i) \mid i = 1, \dots, l\} \cup \{(a_i, b_{i+1}) \mid 1 \leq i < l, i \text{ nicht durch 3 teilbar}\} \\ & \cup \{(b_i, a_{i+1}) \mid 1 \leq i < l, i \text{ durch 3 teilbar}\}. \end{aligned}$$

Die Kanten $\{(a_i, b_i) \mid i = 1, \dots, l\}$ bilden eine maximale Zuordnung mit l Kanten. Die Kanten $\{(a_i, b_{i+1}) \mid 1 \leq i < l, i \text{ nicht durch 3 teilbar}\}$ bilden eine Zuordnung, die durch den Algorithmus bestimmt wurde. Somit gilt

$$OPT(G_l)/A(G_l) = l/(2/3)l = 3/2$$

und der asymptotische Wirkungsgrad ist gleich $3/2$.

13. a) Die Ecken einer Eckenüberdeckung bilden auch eine dominierende Menge, aber nicht umgekehrt. Es sei W_n der Windmühlengraph mit n Ecken (Windmühlengraphen werden auf Seite 349 eingeführt). Die zentrale Ecke bildet eine minimale dominierende Menge von Ecken von W_n , eine minimale Eckenüberdeckung hat dagegen $(n - 1)/2$ Ecken.
- b) Jede Ecke aus $E \setminus E'$ ist zu einer Ecke aus E' benachbart. Mit Hilfe der in Abschnitt 2.8 eingeführten Datenstrukturen kann der Algorithmus so implementiert werden, daß er linearen Zeitaufwand hat.
- c) Im folgenden werden die Ecken S_1, \dots, S_r Spaltenecken und die restlichen Ecken $S_{r+1}, \dots, S_{r+l!}$ Zeilenecken genannt. Sowohl die Zeilen- als auch die Spaltenecken bilden unabhängige Mengen, d.h. die Graphen G_l sind bipartit. Es sei U die Menge der ersten $l!$ Spaltenecken und aller Zeilenecken. Der von U induzierte Untergraph besteht aus $l!$ Zusammenhangskomponenten, jede

besteht aus genau einer Kante. Hieraus folgt, daß die Menge der $l!$ Zeilenecken eine minimale dominierende Menge von Ecken für G_l bildet.

Jede Zeilenecke hat den Eckengrad l und die von der j -ten Spalte induzierten Spaltenecken haben den Eckengrad j . Der angegebene Algorithmus wählt zunächst die von der letzten Spalte induzierten Spaltenecken. Danach verbleiben noch die restlichen Spaltenecken als isolierte Ecken. Der Algorithmus bestimmt somit die Menge der r Spaltenecken als dominierende Menge. Der Wirkungsgrad des Algorithmus ist damit mindestens

$$\frac{r}{l!} = \sum_{j=1}^l \frac{1}{j}.$$

Da die harmonische Reihe divergiert, ist der asymptotische Wirkungsgrad unendlich.

14. Für einen vollständig k -partiten Graphen G gilt $\chi(G) = k$. Nachdem der Greedy-Algorithmus der ersten Ecke einer Teilmenge E_i die Farbe f_i gegeben hat, kann diese Farbe an keine Ecke einer anderen Teilmenge E_j mehr vergeben werden. Die restlichen Ecken von E_i werden jedoch unabhängig von ihrer Reihenfolge mit dieser Farbe gefärbt. Somit vergibt der Algorithmus genau $k = \chi(G)$ Farben. Der Greedy-Algorithmus färbt die Graphen C_n mit ungeradem n für jede Reihenfolge der Ecken mit 3 Farben, d.h. mit der minimalen Anzahl von Farben. Die Graphen C_n sind jedoch nicht vollständig k -partit.
15. Der Algorithmus A_2 bestimmt die chromatische Zahl von G_X mittels einer maximalen Zuordnung $|Z|$ des Komplements von G_X . Es gilt $\chi(G_X) = |X| - |Z|$. Verwendet man anstatt Z eine Zuordnung Z' mit mindestens $2/3|Z|$ Kanten, so wird eine Färbung von G_X mit höchstens $|X| - 2/3|Z| = |X|/3 + 2/3\chi(G_X)$ Farben verwendet. Hieraus kann der Wirkungsgrad des neuen Algorithmus A'_2 bestimmt werden:

$$\begin{aligned} A'_2(G) &\geq n - (|X|/3 + 2/3\chi(G_X)) - \frac{(n - |X|)}{3} \\ &= \frac{2}{3}n - \frac{2}{3}\chi(G_X) \\ &\geq \frac{2}{3}(n - \chi(G)) \\ &= \frac{2}{3}OPT(G) \end{aligned}$$

16. In konstanter Zeit kann festgestellt werden, ob es sich um den vollständigen Graphen mit vier Ecken handelt. In diesem Fall ist $\chi(G) = 4$. Andernfalls folgt aus dem Satz von Brooks, daß $\chi(G) \leq 3$ gilt. Mit linearem Aufwand kann geprüft werden, ob ein bipartiter Graph vorliegt, d.h. ob $\chi(G) = 2$ ist. Ist G nicht bipartit, dann besteht G aus genau einer Ecke oder es gilt $\chi(G) = 3$.
17. a) Für $n = 3$ ist die Aussage trivialerweise richtig. Sei nun $n > 3$. Falls G Hamiltonsch ist, so gilt dies auch für G' . Sei nun umgekehrt G' Hamiltonsch

und H ein Hamiltonscher Kreis von G' . Liegt die Kante (e, f) nicht auf H , dann ist auch H ein Hamiltonscher Kreis für G . Andernfalls ergibt sich aus H ein einfacher Weg $e = e_1, e_2, \dots, e_n = f$ in G , auf dem alle Ecken von G liegen. Im folgenden werden die Mengen

$$A = \{e_i \mid (f, e_{i-1}) \text{ ist Kante in } G \text{ und } 3 \leq i \leq n-1\}$$

und

$$B = \{e_i \mid (e, e_i) \text{ ist Kante in } G \text{ und } 3 \leq i \leq n-1\}$$

betrachtet. Es gilt $\{e_1, e_2, e_n\} \cap A = \emptyset$, $\{e_1, e_2, e_n\} \cap B = \emptyset$, $|A| \geq |N(f)| - 1$ und $|B| = |N(e)| - 1$. Nun folgt aus der Voraussetzung:

$$n - 3 \geq |A \cup B| = |A| + |B| - |A \cap B| \geq n - 2 - |A \cap B|$$

Somit ist $|A \cap B| > 0$, d.h. es gibt eine Ecke e_i , so daß (e, e_i) und (f, e_{i-1}) Kanten in G sind. Dann ist $e_1, e_2, \dots, e_{i-1}, e_n, e_{n-1}, \dots, e_i, e_1$ ein Hamiltonscher Kreis in G .

- b) (i) Da vollständige Graphen Hamiltonsch sind, kann die Aussage leicht mittels Teil a) bewiesen werden.
 - (ii) Folgt direkt aus (i).
18. Der Petersen-Graph besteht aus zwei Kopien von C_5 , die Ecken dieser beiden Graphen sind durch fünf *Speichen* verbunden (siehe Abbildung 2.7) auf Seite 23. Man überlegt sich schnell, daß in einem Hamiltonschen Kreis entweder genau zwei oder genau vier Speichen vorkommen müssen. Im ersten Fall müßten dann jeweils vier Kanten jeder Kopie des zyklischen Graphen C_5 auf dem Hamiltonschen Kreis vorkommen. Dies geht jedoch nicht. Im zweiten Fall müßten von der einen Kopie zwei und von der anderen Kopie drei Kanten auf dem Hamiltonschen Kreis liegen. Man sieht sofort, daß auch dies unmöglich ist.
19. a) Ein 1-Baum existiert nur, falls der von den Ecken $2, \dots, n$ induzierte Untergraph G' zusammenhängend und $g(1) \geq 2$ ist. Es sei B ein minimal aufspannender Baum von G' und K_1 die Menge der zu Ecke 1 inzidenten Kanten. Die beiden Kanten aus K_1 mit den geringsten Bewertungen bilden zusammen mit den Kanten aus B einen minimalen 1-Baum. Der Aufwand des Algorithmus hängt vom Aufwand der Bestimmung eines minimal aufspannenden Baumes für G' ab. Hierzu vergleiche man die in Kapitel 3 vorgestellten Algorithmen.
- b) Man beachte, daß W ein 1-Baum ist.
20. Es seien W_1 und W_2 wie in der Aufgabe beschrieben. Gibt es eine Ecke e , welche mehr als einmal auf W_1 vorkommt, dann gibt es Kanten (e_1, e) und (e, e_2) , welche nacheinander auf W_1 vorkommen. Ersetzt man diese beiden Kanten durch die Kante (e_1, e_2) , so folgt aus der Dreiecksungleichung und der Minimalität von W_1 , daß der neu entstandene Weg die gleiche Länge wie W_1 hat. Auf diese Art kann ein geschlossener einfacher Weg W'_1 mit gleicher Länge wie W_1 erzeugt werden, auf dem alle Ecken von G liegen. Hieraus folgt:

$$L(W_2) \geq L(W_1) = L(W'_1) \geq L(W_2)$$

Dies zeigt die Behauptung.

21. Es sei L_i die Summe der Längen der Kanten, welche bis zum i -ten Schritt ausgewählt und in den Weg W_i eingefügt werden. Es sei $W_i = e_1, \dots, e_i, e_1$ der konstruierte Weg. Mittels vollständiger Induktion nach i wird bewiesen, daß $L(W_i) \leq 2L_i$ für $i = 2, \dots, n$ gilt. Für $i = 2$ ist die Aussage klar. Sei nun $i > 2$ und (e_s, f) die neu ausgewählte Kante. Da G die Dreiecksungleichung erfüllt, gilt

$$d(e_{s-1}, f) \leq d(e_{s-1}, e_s) + d(e_s, f)$$

bzw.

$$d(e_{s-1}, f) + d(e_s, f) - d(e_{s-1}, e_s) \leq 2d(e_s, f).$$

Mit Hilfe der Induktionsvoraussetzung folgt nun

$$\begin{aligned} L(W_i) &= L(W_{i-1}) + d(e_{s-1}, f) + d(e_s, f) - d(e_{s-1}, e_s) \\ &\leq 2L_{i-1} + 2d(e_s, f) \\ &= 2L_i. \end{aligned}$$

(Ist $s = 1$, so ersetze man in diesen Ungleichungen e_{s-1} durch e_i). Man beachte, daß die ausgewählten Kanten genau die Kanten sind, welche auch der Algorithmus von Prim auswählt. Somit ist $L(W_n) \leq 2K < 2OPT(G)$, wobei K die Kosten eines minimal aufspannenden Baumes von G sind. Hieraus folgt die Aussage über den Wirkungsgrad. Der Algorithmus hat die gleiche Laufzeit wie der Algorithmus von Prim.

22. Auf W liegen für jede Ecke e genau zwei verschiedene zu e inzidente Kanten. Diese haben zusammen mindestens die Länge $m(e)$. Da jede Kante auf W zu genau zwei Ecken inzident ist, gilt:

$$\frac{1}{2} \sum_{e \in E} m(e) \leq L(W)$$

23. Der dargestellte Algorithmus folgt im Prinzip dem zweiten Teil des Beweises des Satzes über die Charakterisierung von Eulerschen Graphen auf Seite 322. In diesem Beweis werden sukzessive geschlossene Kantenzüge bestimmt und miteinander verschmolzen. Die rekursive Prozedur `euler` verschränkt die Suchen nach geschlossenen Kantenzügen. Ausgehend von einer Startecke wird ein geschlossener Kantenzug bestimmt und die Kanten werden als besucht markiert. Die Eckengrade des verbleibenden Graphen sind weiterhin gerade. Die Suche wird bei der zuletzt besuchten Ecke fortgesetzt, die zu noch nicht markierten Kanten inzident ist. Dies wird durch die Rekursion umgesetzt. Da die Ausgabe der Ecken am Ende der Prozedur erfolgt, wird auch das korrekte Verschmelzen der Kantenzüge garantiert. Zwei nacheinander ausgegebene Ecken sind benachbart. Da G zusammenhängend ist, besucht die Prozedur auf jeden Fall alle Kanten. Der Korrektheitsbeweis unterscheidet zwei Fälle.

Fall 1: Bis auf den letzten Aufruf bewirkt jeder Aufruf von `euler` genau einen weiteren Aufruf der Prozedur. Die Aufrufhierarchie sei `euler(i1)` bis `euler(is)`, wobei `euler(ik)` nur `euler(ik+1)` aufruft. Dann ist $i_1 = i_s$, $s = n+1$ und die Ausgabe lautet: $i_1, i_n, i_{n-1}, \dots, i_1$. Dies ist notwendigerweise ein Eulerscher Kreis.

Fall 2: Es gibt eine Aufruffolge `euler(i0)` bis `euler(is)`, so daß `euler(ik)` für $k = 1, \dots, s-1$ nur `euler(ik+1)` aufruft, `euler(is)` keinen weiteren Aufruf macht und `euler(i1)` ist der letzte, aber nicht der erste Aufruf von `euler(i0)`. Dann ist $i_0 = i_s$ und es wird $i_s, i_{s-1}, \dots, i_1, i_s$ ausgegeben. Dies ist ein geschlossener Kantenzug. Läßt man die Kanten dieses Kreises aus G weg, so ist die Eckengradbedingung immer noch erfüllt. Per Induktion produziert die Prozedur für diesen Graphen einen Eulerschen Kreis, hierbei werden die Kanten in der gleichen Reihenfolge wie im Orginalgraph besucht. Die Zusammenfassung der beiden Kantenzüge ergibt einen Eulerschen Kreis für den Ausgangsgraphen, dieser ist auch identisch mit der Ausgabe des Gesamtverfahrens.

Der entscheidende Punkt bei der Bestimmung des Aufwandes der Prozedur ist die Umsetzung der Markierungen der Kanten und die Überprüfung der Kantenummarkierungen. Kann dies in konstanter Zeit erfolgen, so ist der Gesamtaufwand $O(m)$. Für die Verwaltung der Markierungen wird ein boolsches Feld `Markierung` der Länge m verwendet, dieses Feld wird mit `false` initialisiert. Weiterhin wird die Adjazenzliste erweitert. Für jede Ecke wird eine Liste von Records vom Typ `Eintrag` mit zwei Komponenten angelegt: Eckenummer des Nachbarn und Index der Kante im Feld `Markierung`. Die Initialisierung dieser neuen Adjazenzliste erfolgt mit Hilfe der normalen Adjazenzliste mit Aufwand $O(m)$:

```
i,j,kantenummer : Integer;
kantenummer := 1:
for jede Ecke i do
    for jeden Nachbar i von j do
        if i < j then begin
            nachbarn(i).anhängen(Eintrag(j, kantenummer));
            nachbarn(j).anhängen(Eintrag(i, kantenummer));
            kantenummer := kantenummer + 1;
        end
```

Die Umsetzung der `if`-Anweisung innerhalb der Prozedur `euler` ist nun sehr einfach. Über die Adjazenzliste erhält man die Kantenummer einer Kante und über das Feld `Markierung` erfährt man, ob die Kante schon markiert ist (unabhängig von der Durchlaufrichtung). Die Markierung einer durch die Kantenummer gegebenen Kante erfolgt in konstanter Zeit.

24. Die Funktion `kanten-Färbung` gibt die Anzahl A der vergebenen Farben zurück. Der Aufwand der Funktion ist gleich $O(A(n + m))$.

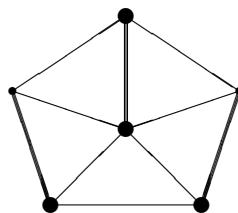
```
function kanten-Färbung(G : Graph) : Integer;
var
    Zuordnung : array[1..max] of Integer;
    i,j,farbe : Integer;
```

```

begin
    farbe := 0;
    Initialisiere Zuordnung mit 0;
    while G enthält noch Kanten do begin
        farbe := farbe + 1;
        for jede Ecke i do
            if Zuordnung[i] < farbe then
                for jeden Nachbar j von i do
                    if Zuordnung[j] < farbe then begin
                        Zuordnung[i] := farbe;
                        Zuordnung[j] := farbe;
                        färbe Kante (i,j) mit Farbe farbe;
                        entferne Kante (i,j);
                        break;
                    end
                end
            kanten-Färbung := farbe;
        end
    end

```

25. Zur Bestimmung des Wirkungsgrades des dargestellten Algorithmus werden die in Abschnitt 9.4 konstruierten Graphen G_r betrachtet. Sie haben eine minimale Eckenüberdeckung mit r Ecken. Die Konstruktion der Graphen bewirkt, daß der Algorithmus der Reihe nach die Ecken aus R_r, R_{r-1}, \dots, R_1 auswählt, d.h. die Menge R wird als Eckenüberdeckung konstruiert. Wie in Abschnitt 9.4 gezeigt, gilt auch für diesen Algorithmus $\mathcal{W}_A(n) = O(\log n)$ und $\mathcal{W}_A^\infty = \infty$. Mit Hilfe der in Abschnitt 2.8 entwickelten Datenstrukturen kann der Algorithmus mit Aufwand $O(n + m)$ umgesetzt werden.
26. Eine minimale Eckenüberdeckung des folgenden Graphen enthält vier Ecken (fett dargestellt), eine maximale Zuordnung enthält aber nur drei Kanten (fett dargestellt).



27. Der Algorithmus verwaltet ein Feld **Überdeckung**, in dem alle zur Überdeckung gehörenden Ecken markiert werden. Die Ecken des Baumes werden gemäß der Tiefensuche besucht. Bei diesem Algorithmus gehören Blätter nicht zur konstruierten Überdeckung. Die Funktion **eckenüberdeckung(i, vorgänger)** zeigt beim Verlassen einer Ecke an, ob die Kante $(i, \text{vorgänger})$ noch überdeckt werden muss. Der Funktionsaufruf **eckenüberdeckung(start, 0)** mit einer beliebigen Ecke **start** bestimmt eine minimale Eckenüberdeckung in linearer Zeit $O(n + m)$.

```

Überdeckung : array[1..max] of Boolean;
function eckenüberdeckung(i : int, int vorgänger) : Boolean;
var

```

```

j : Integer;
begin
  Überdeckung[i] := false;
  for jeden Nachbar j von i do
    if j ≠ vorgänger then do
      Überdeckung[i] := eckenüberdeckung(j,i) or Überdeckung[i]
      eckenüberdeckung = not Überdeckung[i];
  end
end

```

Die Korrektheit des Verfahrens wird durch vollständige Induktion nach n , der Anzahl der Ecken des Baumes B , geführt. Für $n = 2$ erzeugt der Algorithmus offensichtlich eine minimale Eckenüberdeckung mit einer Ecke. Sei nun $n > 2$. Es sei b ein Blatt und v der Vorgänger von b im Tiefensuchebaum. Hat v keinen weiteren Nachfolger, so bestimmt der Algorithmus nach Induktionsvoraussetzung eine minimale Eckenüberdeckung U für den Baum $B \setminus \{v, b\}$. Somit ist $U \cup \{v\}$ eine minimale Eckenüberdeckung von B . Wie man leicht sieht, ist dies aber genau die Eckenüberdeckung, welche der Algorithmus für B produziert. Gibt es in B kein Blatt mit dieser Eigenschaft, so muß es in B ein Blatt b geben, so daß der Vorgänger v ein weiteres Blatt als Nachfolger hat. Nach Induktionsvoraussetzung bestimmt der Algorithmus für den Baum $B \setminus \{b\}$ eine minimale Eckenüberdeckung U . Wie man wieder leicht sieht, ist dies genau die Eckenüberdeckung, welche der Algorithmus für B produziert.

28. Da nach Aufgabe 33 aus Kapitel 4 auf Seite 129 die Blätter eines Tiefensuchebaumes eine unabhängige Menge sind, bildet die Menge der inneren Ecken eine Eckenüberdeckung. Mittels des in Aufgabe 31 in Kapitel 7 beschriebenen Algorithmus kann eine maximale Zuordnung Z des Baumes bestimmt werden. Man überzeugt sich leicht, daß die erste Ecke, die der Algorithmus besucht, durch Z überdeckt wird. Ferner wird auch jede innere Ecke außer der Wurzel durch Z überdeckt. Ist k die Anzahl der Kanten in Z und $e = |E'|$, so gilt $2k \geq e$. Zur Überdeckung der Kanten aus Z werden mindestens k Ecken benötigt. Hieraus folgt die Behauptung.
29. Ist e eine Ecke von G und T eine beliebige Teilmenge der Ecken von G , so wird mit $N_T(e)$ im folgenden die Menge der in T liegenden Nachbarn von e bezeichnet. Es sei e eine Ecke aus X mit $N_X(e) > N_{\bar{X}}(e)$. Wird nun e von X nach \bar{X} verschoben, so erhält der Schnitt (X, \bar{X}) genau $N_{\bar{X}}(e) - N_X(e) > 0$ zusätzliche Kanten. Auch bei der Verschiebung einer Ecke e aus \bar{X} mit $N_{\bar{X}}(e) > N_X(e)$ nach X wird die Anzahl der Kanten im Schnitt erhöht. Da ein Schnitt maximal m Kanten enthält, endet das Verfahren spätestens nach m Schritten. Dann gilt $N_X(e) \leq N_{\bar{X}}(e)$ für alle Ecken e aus X , und $N_{\bar{X}}(e) \leq N_X(e)$ für alle Ecken aus \bar{X} . Nun kann der Wirkungsgrad leicht bestimmt werden. Es ist

$$2A(G) = \sum_{e \in X} N_{\bar{X}}(e) + \sum_{e \in \bar{X}} N_X(e).$$

Hieraus folgt

$$\begin{aligned}
 OPT(G) &\leq m \\
 &= \frac{1}{2} \left(\sum_{e \in X} (N_X(e) + N_{\bar{X}}(e)) + \sum_{e \in \bar{X}} (N_X(e) + N_{\bar{X}}(e)) \right) \\
 &\leq \sum_{e \in X} N_{\bar{X}}(e) + \sum_{e \in \bar{X}} N_X(e) \\
 &= 2A(G).
 \end{aligned}$$

30. Das betrachtete Entscheidungsproblem liegt offenbar in \mathcal{NP} . Es wird ein Graph G mit Eckenmenge $E = X \cup Y \cup Z$ konstruiert. Zwei Ecken $a, b \in E$ sind genau dann benachbart, wenn es kein Element $m \in M$ gibt, welches a und b gleichzeitig als Komponenten enthält. Der Graph G hat $3q$ Ecken und lässt sich in polynomialer Zeit konstruieren. Ist $(x, y, z) \in M$, dann bilden x, y, z in G eine unabhängige Menge. Die von den Mengen X, Y und Z induzierten Untergraphen sind vollständig, somit ist $\chi(G) \geq q$. Es sei A eine vier-elementige Teilmenge von E . Dann muß es in A zwei Elemente geben, welche in der gleichen der drei Cliquen liegen. Somit ist $\alpha(G) \leq 3$. Im folgenden wird bewiesen, daß M genau dann eine 3-dimensionale Zuordnung besitzt, wenn $\chi(G) = q$ gilt. Hieraus folgt unmittelbar die in der Aufgabe gemachte Aussage. Ist $\chi(G) = q$, so besteht G aus genau q Farbklassen mit je drei Elementen. Es sei $\{a, b, c\}$ mit $a \in X, b \in Y$ und $c \in Z$ eine solche Farbklasse. Nach Konstruktion von G gibt es $x \in X, y \in Y$ und $z \in Z$, so daß $(x, b, c), (a, y, c), (a, b, z)$ in M liegen. Wegen der paarweisen Konsistenz muss dann auch (a, b, c) in M liegen. Hieraus folgt, daß die aus den Farbklassen gebildeten Elemente aus M eine 3-dimensionale Zuordnung bilden. Umgekehrt gilt, daß jede 3-dimensionale Zuordnung auf G eine disjunkte Überdeckung mit 3-elementigen unabhängigen Mengen induziert, d.h. es gilt $\chi(G) = q$.
31. Der erste Teil des Algorithmus entspricht der Funktion **greedy-Zuordnung** aus Aufgabe 10. Besteht die erzeugte Zuordnung Z aus m' Kanten, so gilt $|I_1| = n - 2m'$. Wegen $m' < n/2$ gibt es eine Zahl $\gamma \in [0, 1]$ mit $m' = n(1 - \gamma)/2$ bzw. $|I_1| = n\gamma$. Nun bildet $|Z| + |I_1|$ eine obere Abschätzung für $\alpha(G)$, hieraus folgt:

$$\alpha(G) \leq \frac{n}{2}(1 + \gamma) \quad (\text{B.1})$$

Nun wird der zweite Teil des Algorithmus betrachtet. Im ersten Durchlauf der **while**-Schleife werden $\delta + 1$ Ecken entfernt. In jedem weiteren Schritt werden höchstens Δ Ecken aus E entfernt. Dazu beachte man, daß es wegen des Zusammenhangs von G mindestens eine Ecke im Restgraphen geben muss, der Grad kleiner oder gleich $\Delta - 1$ ist. Somit gilt

$$|I_2| \geq \frac{n - (\delta + 1)}{\Delta} + 1. \quad (\text{B.2})$$

Aus Gleichung (B.1) und $|I_1| = n\gamma$ folgt

$$\frac{\alpha(G)}{|I_1|} \leq \frac{\gamma + 1}{2\gamma}. \quad (\text{B.3})$$

Aus Gleichung (B.1) und (B.2) folgt

$$\frac{\alpha(G)}{|I_2|} \leq \frac{\frac{n}{2}(1+\gamma)}{\frac{n-(\delta-1)+\Delta}{\Delta}} = \frac{\Delta n(1+\gamma)}{2(n-(\delta+1)+\Delta)} \leq \frac{\Delta n(1+\gamma)}{2(n-1)}. \quad (\text{B.4})$$

Für die folgende Analyse beachte man, daß für $\gamma \in [0, 1]$ die Funktion $(\gamma+1)/2\gamma$ monoton fallend und die Funktion $\Delta n(1+\gamma)/2(n-1)$ monoton steigend ist. Für die Bestimmung des Wirkungsgrades des Algorithmus ist der Schnittpunkt $S = (n-1)/\Delta n$ der beiden Funktionen von Bedeutung. Für $\gamma \leq S$ ist $|I_2| \geq |I_1|$ und für $\gamma \geq S$ ist $|I_1| \geq |I_2|$. Aus den Gleichungen (B.3) bzw. (B.4) ergibt sich die angegebene Schranke für den Wirkungsgrad.

32. Es sei d_i der Eckengrad der Ecke im Restgraphen, welche in Durchlauf i entfernt wurde. Dann gilt folgende Beziehung zwischen n und den d_i :

$$n = \sum_{i=1}^{|I_2|} (d_i + 1)$$

Da jedesmal die Ecke mit dem kleinsten Eckengrad ausgewählt wird, haben im i -ten Schritt alle entfernten Ecken mindestens den Eckengrad d_i . Somit werden im i -ten Schritt mindestens $d_i(d_i + 1)/2$ Kanten entfernt. Hieraus folgt:

$$\frac{\bar{d}n}{2} = |E| \geq \sum_{i=1}^{|I_2|} d_i(d_i + 1)/2$$

Addiert man das zweifache der letzten Gleichung zu der ersten Gleichung, so erhält man mittels der Cauchy-Schwartz Ungleichung und der ersten Gleichung:

$$(\bar{d} + 1)n \geq \sum_{i=1}^{|I_2|} (d_i + 1)^2 \geq \frac{1}{|I_2|} \left(\sum_{i=1}^{|I_2|} (d_i + 1) \right)^2 \geq \frac{n^2}{|I_2|}$$

Hieraus ergibt sich sofort die Behauptung.

33. Der Beweis erfolgt mittels vollständiger Induktion nach der Anzahl der Ecken des Graphen. Für $n = 1$ ist die Behauptung trivial. Es sei nun G ein Graph mit $n > 1$ Ecken. Die Funktion **unabhängigeMenge** bestimmt eine unabhängige Menge U mit $f(n)$ Ecken. Es sei $G' = G \setminus U$. Nach Induktionsvoraussetzung erzeugt **färbung** für G' eine Färbung mit höchstens

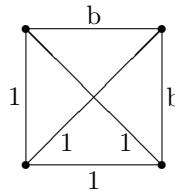
$$\sum_{i=1}^{n-f(n)} \frac{1}{f(i)}$$

Farben und für G wird genau eine zusätzliche Farbe benötigt. Da f monoton steigend ist, gilt

$$\sum_{i=n-f(n)+1}^n \frac{1}{f(i)} \geq \sum_{i=n-f(n)+1}^n \frac{1}{f(n)} = 1.$$

Hieraus folgt die Behauptung.

34. Erfüllen die Bewertungen der Kanten die Dreiecksungleichung, so gilt $L \leq 2K$. Im allgemeinen gibt es aber keine solche Konstante, wie der folgende Graph zeigt, hierbei ist L von b abhängig und K von b unabhängig.



35. Ein polynomialer Algorithmus für das Optimierungsproblem löst natürlich auch das zugehörige Entscheidungsproblem. Es sei nun A ein polynomialer Algorithmus für das Entscheidungsproblem. In einer ersten Phase werden die Kosten C für eine optimale Lösung berechnet, die Bestimmung der optimalen Rundreise erfolgt dann in der zweiten Phase. Ist l die Länge der Kodierung einer Instanz des Problems des Handlungsreisenden, so liegt C zwischen 0 und 2^l . Mittels binärer Suche in Kombination mit Algorithmus A kann C in höchstens n Schritten bestimmt werden. In der zweiten Phase wird Algorithmus A für jede Kante einmal aufgerufen. Hierzu werden die Kosten der betrachteten Kante auf $C + 1$ gesetzt. Ergibt die Anwendung von A , daß dieser Graph keinen Hamiltonschen Kreis mit Kosten C besitzt, so wird die Bewertung der Kante wieder auf den ursprünglichen Wert zurückgesetzt, andernfalls wird die Kante entfernt. Am Ende dieser Phase bilden die verbliebenen Kanten einen Hamiltonschen Kreis mit Kosten C und das Optimierungsproblem ist gelöst. Insgesamt wurde Algorithmus A höchstens $n + m$ -mal aufgerufen, d.h. das Verfahren hat polynomialen Aufwand.
36. Das betrachtete Entscheidungsproblem liegt offenbar in \mathcal{NP} . Für einen gegebenen zusammenhängenden Graphen G wird wie im Hinweis zur Aufgabe ein neuer Graphen G' konstruiert. Es sei

$$S = \{x_{ef} \mid (e, f) \text{ ist Kante von } G\} \cup \{e_n\}.$$

Es sei s_{stei} das Gewicht eines minimalen Steiner Baums von G' für S und s_{deck} die Anzahl der Ecken in einer minimalen Eckenüberdeckung für G . Im folgenden wird bewiesen, daß $s_{stei} = s_{deck} + m$ gilt. Hieraus folgt unmittelbar die in der Aufgabe gemachte Aussage.

Es sei M eine minimale Eckenüberdeckung von G . Dann ist der von $S \cup M$ induzierte Untergraph U von G' zusammenhängend. Hierzu beachte man, daß e_n zu jeder Ecke aus M benachbart ist und jede Ecke aus $S \setminus \{e_n\}$ zu einer Ecke aus M benachbart sein muß (M ist eine Eckenüberdeckung). Da alle Kanten die Bewertung 1 haben, hat ein aufspannender Baum von U das Gewicht $|S \cup M| = s_{deck} + m$. Hieraus folgt $s_{stei} \leq s_{deck} + m$. Sei nun T ein minimaler Steiner Baum von G' für S und S' die Menge der Steiner Ecken von T . Nach Konstruktion von G' bildet S' eine Eckenüberdeckung von G : Ist $k = (e, f)$ eine Kante von G , so gibt es in T einen Weg von e_n nach x_{ef} , dieser verwendet eine der Ecken e und f , d.h. e oder f liegt in S' . Hieraus folgt:

$$s_{deck} \leq |S'| = s_{stei} + 1 - |S| = s_{stei} - m$$

37. Der konstruierte Baum T^* ist sicherlich ein Steiner Baum für S . Es gilt:

$$\text{kosten}(T^*) \leq \text{kosten}(T'') \leq \text{kosten}(G'') \leq \text{kosten}(T')$$

Es sei T_{OPT} ein minimaler Steiner Baum für S . Mit Hilfe der Tiefensuche, angewendet auf T_{OPT} , kann ein geschlossener Kantenzug W in G erzeugt werden, der jede Kante von T_{OPT} genau zweimal enthält. Dazu wird sowohl beim Erreichen als auch beim Verlassen einer Ecke die entsprechende Kante in W eingefügt. Somit gilt $\text{kosten}(W) = 2 \cdot \text{kosten}(T_{OPT})$. Es sei $s_1 \in S$ ein Blatt von T_{OPT} . Nun durchläuft man W beginnend bei s_1 . Dabei trifft man auf die restlichen Ecken von S . Bezeichne mit $s_2, \dots, s_{|S|}$ die restlichen Ecken von S in der Reihenfolge, in der ihnen zum ersten Mal auf W begegnet wird. Da G' vollständig ist, bilden die Kanten $\{(s_i, s_{i+1}) \mid 1 \leq i \leq |S|-1\}$ einen aufspannenden Baum B für G' . Aus der Dreiecksgleichung für G' folgt $\text{kosten}(B) \leq \text{kosten}(W)$. Da T' ein minimal aufspannender Baum für G' ist, gilt $\text{kosten}(T') \leq \text{kosten}(B)$. Zusammenfassend erhält man

$$\text{kosten}(T^*) \leq \text{kosten}(T') \leq \text{kosten}(B) \leq \text{kosten}(W) = 2 \cdot \text{kosten}(T_{OPT}).$$

Mit Hilfe der Algorithmen von Dijkstra und Prim wird eine Laufzeit von $O(|S|n^2)$ erzielt.

38. Der Algorithmus bestimmt den links dargestellten minimal aufspannenden Baum T' von G' . Da alle Kanten von T' auch Kanten von G sind, ist $G'' = T' = T''$. Der resultierende Steinerbaum T^* mit Gewicht 9 ist rechts dargestellt. Ein minimaler Steinerbaum hat Gewicht 8.



39. Ein polynomialer Algorithmus für das Optimierungsproblem löst natürlich auch das zugehörige Entscheidungsproblem. Es sei

```
function eckenüberdeckung(G : Graph, wert : Integer) : Boolean;
```

eine Funktion, welche mit polynomialem Aufwand entscheidet, ob G eine Eckenüberdeckung mit $wert$ Ecken hat. Mit maximal OPT Aufrufen dieser Funktion kann die Größe OPT der kleinsten Eckenüberdeckung von G bestimmt werden. Es sei nun (e, f) eine Kante des Graphen G . Dann liegt e oder f in einer minimalen Eckenüberdeckung. Es sei U_e bzw. U_f eine minimale Eckenüberdeckung des Graphen, der sich aus G ergibt, wenn alle zu e bzw. zu f inzidenten Kanten entfernt werden. Ist $|U_e| = OPT - 1$, so ist $U_e \cup \{e\}$ eine minimale Eckenüberdeckung von G , andernfalls $U_f \cup \{f\}$. Der beschriebene Algorithmus wird durch den Aufruf `minüberdeckung(G, OPT)` der folgenden rekursiven Funktion umgesetzt:

```
function minüberdeckung(G : Graph, wert Integer) : set of Kanten;
begin
    if G hat keine Kanten then
        minüberdeckung := ∅;
    else begin
        wähle beliebige Kante (e,f) aus G;
         $G_e := G$  ohne die zu e inzidenten Kanten;
        if eckenüberdeckung( $G_e$ , wert-1) then
            minüberdeckung := {e}  $\cup$  minüberdeckung( $G_e$ , wert-1);
        else begin
             $G_f := G$  ohne die zu f inzidenten Kanten;
            minüberdeckung := {f}  $\cup$  minüberdeckung( $G_f$ , wert-1);
        end
    end
end
```

Insgesamt wird diese Funktion höchstens n -mal aufgerufen. Damit wird auch die Funktion eckenüberdeckung höchstens n -mal aufgerufen, d.h., es liegt ein Algorithmus mit polynomialem Aufwand vor.

Literaturverzeichnis

- [1] Aho, A.V., Hopcroft, J.E. und Ullman, J.D., “Data Structures and Algorithms”, *Addison-Wesley*, 1983.
- [2] Aho, A.V., Sethi, R. und Ullman, J.D., “Compilerbau”, *Addison-Wesley*, 1988.
- [3] Ahuja, R.K., Mehlhorn, K., Orlin J.B. und Tarjan, R.E., “Faster Algorithms for the Shortest Path Problem”, *Technical Report CS-TR-154-88* Dept. of Comp. Science, Princeton Univ., 1988.
- [4] Ahuja, R.K., Magnanti, T.L. und Orlin J.B., “Network flows: theory, algorithms and applications”, *Prentice Hall, Englewood Cliffs, N.J.*, 1993.
- [5] Alt, H., Blum, N., Mehlhorn, K. und Paul, M., “Computing a maximum cardinality matching in a bipartite graph in time $O(n^{1.5} \sqrt{m/\log n})$ ”, *Information Processing Letters 37*, 237–240, 1991.
- [6] André, P. und Royer, J.-C., “Optimizing method search with lookup caches and incremental coloring”, *OOPSLA Conference Proceedings '92*, 110–126, 1992.
- [7] Appel, K. und Haken, W., “Every planar map is four colorable”, *Bulletin of the American Mathematical Society*, Vol. 82, 711–712, 1976.
- [8] Appel, K. und Haken, W., “Every planar map is four colorable”, *Illinois Journal of Mathematics*, Vol. 21, 421–567, 1977.
- [9] Arora, S. und Safra, S., “Probabilistic checking of proofs”, *Proc. 33rd IEEE Symp. on the Foundations of Computer Science*, 2–13, 1992.
- [10] Arora, S., Lund, C., Motwani, R., Sudan, M. und Szegedy, M., “Proof verification and hardness of approximation problems”, *Proc. 33rd IEEE Symp. on the Foundations of Computer Science*, 14–23, 1992.
- [11] Bell, T., Cleary, J. und Witten, I., “Text Compression”, *Prentice Hall*, 1990.
- [12] Bellman, R.E., “On a routing problem”, *Quart. Appl. Math. 16*, 87–90, 1958.
- [13] Berger, B. und Rompel, J., “A Better Performance Guarantee for Approximate Graph Coloring”, *Algorithmica 5*, 459–466, 1990.
- [14] Blum, A., “New Approximation Algorithms for Graph Coloring”, *Journal of the ACM*, 470–516, 1994.
- [15] Braers, D., “Die Bestimmung kürzester Pfade in Graphen und passende Datenstrukturen”, *Computing 8*, 171–181, 1971.

- [16] Brin, S. und Page, L., "The Anatomy of a Large-Scale Hypertextual Web Search Engine", Proc. 7th Int'l World Wide Web Conf., *ACM Press*, New York, 107–117, 1998.
- [17] Brooks, R.L., "On Colouring the Nodes of a Network", *Proc. Cambridge Phil. Soc.*, Vol. 37, 194–197, 1941.
- [18] Buckley, C., "Path-Planning Methods for Robot Motion", in: Rembold, U., *Robot Technology and Applications*, Marcel Dekker, 1990.
- [19] Chaitin, G.J., "Register allocation und spilling via graph colouring", *ACM SIGPLAN Notices* 17:G, 201–207, 1982.
- [20] Cherkassky, B.V., Goldberg, A.V. und Radzik, T., "Shortest Paths Algorithms: Theory and Experimental Evaluation", *Stanford University*, Technical Report, 1993.
- [21] Cherkassky, B.V. und Goldberg, A.V., "On implementing push-relabel method for the maximum flow problem", *Stanford University*, Technical Report, 1993.
- [22] Chiba, N., Nishizeki, T. und Saito, N., "A linear Algorithm for five-coloring a planar Graph", *Lecture Notes in Computer Science*, 108, *Graph Theory and Algorithms*, Springer Verlag, 9–19, 1980.
- [23] Christofides, N., "Worst-case analysis of a new heuristic for the traveling salesman problem", *Technical Report*, Graduate School of Industrial Administration, Carnegie-Mellon University, 1976.
- [24] Cormen T.H., Leiserson C.E. und Rivets R.L., "Introduction to Algorithms", *MIT Press*, 1990.
- [25] Cornuejols, G. und Nemhauser, G.L., "Tight Bounds for Christofides' Traveling Salesman Heuristic", *Mathematical Programming* 14, 116–121, 1978.
- [26] Culberson, J.C., "Iterated Greedy Graph Coloring und the Difficulty Landscape", *Technical Report TR 92-07*, University of Alberta, Canada, June 1992 (Zu beziehen über <http://web.cs.ualberta.ca/~joe/>).
- [27] Dantzig, G.B. und Fulkerson, D.R., "On the Max-Flow Min-Cut Theorem of Networks", In: *Linear Inequalities and Related Systems*, Annals of Math. Study 38, Princeton University Press, 1956, 215–221.
- [28] Dial, R.B., "Algorithm 360: Shortest Path Forest with Topological Ordering", *Comm. ACM* 12, 632–633, 1969.
- [29] Dijkstra, E.W., "A note on two problems in connexion with graphs", *Numerische Mathematik* 1, 269–271, 1959.
- [30] Center for Discrete Mathematics and Theoretical Computer Science, "Clique and coloring Problems: A brief introduction with project ideas", *Technical Report*, New Brunswick, 1992 (Zu beziehen per ftp von [dimacs.rutgers.edu](ftp://dimacs.rutgers.edu)).

- [31] Dinic, E.A., "Algorithms for solution of a problem of maximum flow in networks with power estimation", *Soviet Math. Dokl.* 11, 1277–1280, 1970.
- [32] Domschke, W., "Logistik: Rundreisen und Touren", *Oldenbourg Verlag*, 1989.
- [33] Domschke, W., "Logistik: Transport", *Oldenbourg Verlag*, 1989.
- [34] Domschke, W., "Zwei Verfahren zur Suche negativer Zyklen in bewerteten Digraphen", *Computing* 11, 125–136, 1973.
- [35] Edmonds, J. und Karp, R.M., "Theoretical improvements in algorithmic efficiency for network flow problems", *Journal of the ACM* 19 (2), 248–264, 1972.
- [36] Even, S., Pnueli, A. und Lempel, A., "Permutation Graphs and Transitive Graphs", *Journal of the ACM*, 400–410, 1972.
- [37] Even, S. und Tarjan, R.E., "Network Flow and Testing Graph Connectivity", *SIAM Journal on Computing* 4, 507–518, 1975.
- [38] Even, S., "Algorithm for Determining whether the Connectivity of a Graph ist at least k", *SIAM Journal on Computing* 4, 393–396, 1977.
- [39] Florian, M. und Robert, P., "A direct search method to locate negative cycles in a graph", *Manag. Sci.* 17, 307–310, 1971.
- [40] Floyd, R.W., "Algorithm 245: treesort 3", *Comm. ACM* 7:12, 701, 1964.
- [41] Floyd, R.W., "Algorithm 97: shortest path", *Comm. ACM* 5:6, 345, 1962.
- [42] Ford, L.R. und Fulkerson, D.R., "Maximal flow through a network", *Canad. J. Math.* 8, 399–404, 1956.
- [43] Ford, L.R., "Network Flow Theory", *Rand Corporation, Santa Monica, Calif.*, 293, 1956
- [44] Fredman, M. und Tarjan, R.E., "Fibonacci heaps and their use in improved network optimization algorithms", *Journal of the ACM* 34 (3), 596–615, 1987.
- [45] Fredrickson, G.N., "Fast algorithms for shortest paths in planar graphs, with applications", *SIAM Journal on Computing* 16, 1004–1022, 1987.
- [46] Garey, M.L. und Johnson, D.S., "Computers and Intractability — A guide to the theory of NP-Completeness", *W.H. Freeman And Company*, 1979.
- [47] Gilmore, P. und Hoffman, A., "A Characterization of Comparability Graphs and of Interval Graphs", *Canad. J. Math.* 16, 539–548, 1964.
- [48] Goldberg, A.V., "A new max-flow algorithm", *Tech. Memorandum MIT/LCS/TM-291*, Lab. for Comp. Science, MIT, 1985.
- [49] Goldberg, A.V. und Tarjan, R.E., "A New Approach to the Maximum Flow Problem", *Journal of the ACM* 35, 921–940, 1988.

- [50] Goldberg, A.V., Tardos, E. und Tarjan, R.E., "Network flow algorithms", *Technical Report STAN-CS-89-1252*, Dept. of Comp. Science, Stanford Univ., 1989.
- [51] Goldberg, A.V. und Radzik, T., "A Heuristic Improvement of the Bellman-Ford Algorithm", *Applied Math. Let.* 6, 3–6, 1993.
- [52] Goldreich, O., Micali, S. und Wigderson A., "Proofs that yield nothing but their validity, and a methodology of cryptographic protocol design", *Proc. 27th IEEE Symp. on the Foundations of Computer Science*, 174–187, 1986.
- [53] Grimmett, G.R. und McDiarmid, C.J.H., "On coloring random graphs", *Math. Proc. Cambridge Philos. Soc.* 77, 313–324, 1975.
- [54] Grötschel, M. und Lovász, L., "Combinatorial Optimization: A Survey", *DIMACS Technical Report 93-29*, Dept. of Comp. Science, Princeton Univ., 1993.
- [55] Hall, P., "On Representatives of Subsets", *J. London Math. Soc.*, Vol 10, 1935, 26–30.
- [56] Halldórsson, M.M., "A still better performance guarantee for approximate graph coloring", *Information Processing Letters* 45, 19–23, 1993.
- [57] Halldórsson, M.M., "Approximating Set Cover via Local Improvements", *JAIST Technical Report IS-RR-95-0002F*, 1995.
- [58] Harary, F., Hedetniemi, S. und Robinson, R.W., "Uniquely colorable graphs", *Journal Combinatorial Theory* 6, 264–270, 1969.
- [59] Harary, F., "Graphentheorie", *Oldenbourg Verlag*, 1974.
- [60] Hart, P., Nilsson, N. und Raphael, B., "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", *IEEE Trans. Sys. Sci. Cybern.* 2, 100–107, 1968.
- [61] Hassin, R. und Lahav, S., "Maximizing the number of unused colors in the vertex coloring problem", *Information Processing Letters* 52, 87–90, 1989.
- [62] Hayes, B., "Graph Theory in Practice: Part I", *American Scientist* 88(1), 9–13, 2000.
- [63] Hayes, B., "Graph Theory in Practice: Part II", *American Scientist* 88(2), 104–109, 2000.
- [64] Heesch, H., "Untersuchungen zum Vierfarbenproblem", *Hochschulskriptum 810/a/b*, Bibliographisches Institut, Mannheim, 1969.
- [65] Hochbaum, D.S., "Approximation Algorithms for NP-Hard Problems", *PWS Publishing Company*, 1997.
- [66] Holyer, I., "The NP-completeness of edge coloring", *SIAM Journal on Computing* 10, 718–720, 1981.

- [67] Hopcroft, J.E. und Tarjan, R.E., "Efficient planarity testing", *Journal of the ACM* 21(4), 549–568, 1974.
- [68] Hopcroft, J.E. und Karp, R.M., "An $n^{5/2}$ Algorithm for Maximum Matching in Bipartite Graphs", *SIAM Journal on Computing* 2:4, 225–231, 1973.
- [69] Hopcroft, J.E. und Tarjan, R.E., "Dividing a graph into triconnected components", *SIAM Journal on Computing* 2, 135–158, 1973.
- [70] Huffman, D.A., "A method for the construction of minimum-redundancy codes", *Proc. IRE* 40, 1098–1101, 1952.
- [71] Jensen, T.R. und Toft, B., "Graph Coloring Problems", *Wiley Interscience*, 1995.
- [72] Johnson, D.S., "Worst-case behaviour of graph-colouring algorithms", *Proc. 5th South-Eastern Conf. on Combinatorics, Graph Theory and Computing, Utilitas Mathematica Publishing*, Winnipeg, 513–527, 1974.
- [73] Johnson, D.S., "Approximation Algorithms for Combinatorial Problems", *Journal of Computer and System Sciences* 9, 256–278, 1974.
- [74] Johnson, D.S. und McGeoch, C.C., "Network Flows and Matching: First DIMACS Implementation Challenge", *AMS*, 1993.
- [75] Johnson, D.S., "The tale of the second prover", *Journal of Algorithms* 13, 502–524, 1992.
- [76] Jungnickel, D., "Graphen, Netzwerke und Algorithmen", *Wissenschaftsverlag, Mannheim*, 1994.
- [77] Kou, L., Markowsky, G. und Berman, L., "A fast algorithm for Steiner trees," *Acta Informatica*, Vol. 15, 141–145, 1981.
- [78] Knuth, D.E., "The Art of Computer Programming Vol. III: Sorting and Searching", *Addison-Wesley, Reading, Mass.*, 1973.
- [79] Knuth, D.E., "The Stanford GraphBase", *Addison-Wesley, Reading, Mass.*, 1993.
- [80] Kohlas, J., "Zuverlässigkeit und Verfügbarkeit", *Teubner Studienbücher*, 1987.
- [81] Korf, R.E., "Depth-first iterative-deepening: An optimal admissible tree search", *Artificial Intelligence*, Vol. 27, No. 1, 97–109, 1985.
- [82] Korf, R.E., "Optimal Path-Finding Algorithms", in: Search in Artificial Intelligence, Editoren: Kumal, L. und Kumar, V., *Springer Verlag*, 1988.
- [83] Kozen, D.C., "The Design and Analysis of Algorithms", *Springer Verlag*, 1992.
- [84] Kruskal, J., "On the shortest spanning subtree of a graph and the traveling salesman problem", *Proc. AMS* 7:1, 48–50, 1956.
- [85] Kubale, M. und Jackowski, B., "A generalized implicit enumeration algorithm for graph coloring", *Comm. ACM* 28, 412–418, 1985.

- [86] Kučera, L., "The Greedy Coloring is a Bad Probabilistic Algorithm", *Journal of Algorithms* 12, 674–684, 1991.
- [87] Lawler, E.L., Lenstra, J.K., Rinnoy Kan, A.H.G. und Shmoys, D.B., "The Traveling Salesman Problem", *John Wiley & Sons*, 1985.
- [88] van Leeuwen, J., "Graph Algorithms", in: *Handbook of Theoretical Computer Science, Volume A*, Elsevier, 1990.
- [89] Lin, S. und Kernighan, B.W., "An effective heuristic for the traveling salesman problem", *Operations Research* 21, 498–516, 1973.
- [90] Lund, C. und Yannakakis, M., "On the hardness of approximating minimization problems", *Proc. of the 25th Annual ACM Symposium on Theory of Computing*, 286–293, 1993.
- [91] Malhotra, V.M., Pramodh Kumar, M. und Mahaswari, S.N., "An $O(|V|^3)$ algorithm for finding maximum flows in networks", *Information Processing Letters* 7, 277–278, 1978.
- [92] Manber, U., "Introduction to Algorithms", Addison-Wesley, 1989.
- [93] Manzini, G., "BIDA*: an improved perimeter search algorithm", *Artificial Intelligence* 75, 347–360, 1995.
- [94] Mehlhorn, K., "Data Structures and Algorithms", Volume 1–3, Springer-Verlag, 1984.
- [95] Mycielski, J., "Sur le coloriage des Graphes", *Colloq. Math.* 3, 161–162, 1955.
- [96] Menger, V., "Zur allgemeinen Kurventheorie", *Fund. Math.* 10, 26–30, 1927.
- [97] Micali, S. und Vazirani, V.V., "An $O(\sqrt{|V||E|})$ Algorithm for Finding Matching in General Graphs", *Proc. 21st Ann. IEEE Symp. Foundations of Computing Science*, Syracuse, 17–27, 1980.
- [98] Moore, E.F., "The Shortest Path through a Maze", *Proc. Int. Symp. on Theory of Switching, Part II*, 285–292, 1959.
- [99] Munro, J.I., "Efficient determination of the transitive closure of a directed graph", *Information Processing Letters* 1, 56–58, 1971.
- [100] Nuutila, E. und Soisalon-Soininen, E., "On finding the strongly connected components in a directed graph", *Information Processing Letters* 49, 9–14, 1994.
- [101] Ottmann, T. und Widmayer, P., "Algorithmen und Datenstrukturen", Wissenschaftsverlag, 1993.
- [102] Page, L., Brin, S., Motwani, R. und Winograd, T., "The PageRank Citation Ranking: Bringing Order to the Web", Stanford Digital Library Technologies Project, 1998.

- [103] Papadimitriou, C.H., "Computational Complexity", *Addison-Wesley*, 1994.
- [104] Paschos, V.T. "A $(\Delta/2)$ -approximation algorithm for the maximum independent set problem", *Information Processing Letters* 44, 11–13, 1992.
- [105] Pnueli, A., Lempel, A. und Even, S., "Transitive Orientation of Graphs and Identification of Permutation graphs", *Canad. J. Math.*, 160–175, 1971.
- [106] Prim, R., "Shortest connection networks and some generalization", *Bell System Technical J.* 36, 1389–1401, 1957.
- [107] Reinelt, G., "The Traveling Salesman", *Lecture Notes in Computer Science* 840, Springer Verlag, 1994.
- [108] Rhee, C., Liang, Y.L., Dhall, S.K. und Lakshmivarahan S., "Efficient algorithms for finding depth-first and breadth-first search trees in permutation graphs", *Information Processing Letters* 49, 45–50, 1994.
- [109] Robertson, N., Sanders, D., Seymour, P. und Thomas, R., "A new proof of the four-colour theorem", *Electronic Research Announcements Of The AMS*, Vol. 2, Number 1, 1996.
- [110] Rosenkrantz, D.J., Stearns, R.E. und Lewis, P.M., "An analysis of several heuristics for the traveling salesman problem", *SIAM Journal Computing* 6, 563–581, 1977.
- [111] Roy, B., "Transitivité et connexité", *C.R. Acad. Sci. Paris* 249, 216–218, 1959.
- [112] Saaty, T.L. und Kainen, P.C., "The four-color problem, Assaults and Conquest", *Dover Publications*, 1986.
- [113] Savage, C., "Depth-First Search and the Vertex Cover Problem", *Information Processing Letters* 14, 233–235, 1982.
- [114] Simon, K., "Effiziente Algorithmen für perfekte Graphen", *B.G. Teubner*, 1992.
- [115] Spirakis, P. und Tsakalidis, A., "A very fast, practical algorithm for finding a negative cycle in a digraph", *Proc. of 13th ICALP*, 397–406, 1986.
- [116] Stoer, M. und Wagner F., "A Simple Min Cut Algorithm", *Algorithms, Proc. Sec. Annual European Symposium*, Springer Lecture Notes in Comp. Science 855, 141–147, 1994.
- [117] Tarjan, R.E., "Depth first search and linear graph algorithms", *SIAM Journal Computing* 1, 146–160, 1972.
- [118] Thomas, R., "An Update on the Four-Color Theorem", *Notices of the American Mathematical Society*, Volume 45, Number 7, August 1998.
- [119] Turner, J.S., "Almost all k-colorable graphs are easy to color", *Journal of Algorithms*, 9, 63–82, 1988.

- [120] Vizing, V.G., “Über die Abschätzung der chromatischen Klasse eines p-Graphen”, *Diskret. Analiz.* 3, 25–30, 1964.
- [121] Warshall, S.A., “A theorem in Boolean matrices”, *Journal of the ACM* 9, 11–12, 1962.
- [122] Whitney, M., “Congruent graphs and the connectivity of graphs”, *Amer. J. Math.* 54, 150–168, 1932.
- [123] Wilf, H.S., “Backtrack: An O(1) expected time algorithm for the graph coloring problem”, *Information Processing Letters* 18, 119–121, 1984.
- [124] Williams, J., “Algorithm 232 : Heapsort”, *Comm. ACM* 7:6, 347–348, 1964.
- [125] Winter, P., “Steiner tree problem in networks: a survey”, *Networks* 17, 129–167, 1987.
- [126] Wirth, N., “Algorithmen und Datenstrukturen”, *Teubner Studienbücher*, 1975.
- [127] Zelikovski, A.Z., “An 11/6-approximation algorithm for the network Steiner problem”, *Algorithmica* 9, 463–470, 1993.

Index

Symbolen

- I_n , 238
- $N(e)$, 19
- $OPT(a)$, 296
- $W^e(a, b)$, 215
- $W^k(a, b)$, 223
- $Z^e(a, b)$, 214
- $Z^k(G)$, 222
- $\Delta\text{-TSP}$, 319, 332
- $\Delta(G)$, 19
- $\alpha(G)$, 137
- \mathcal{NP} (non-deterministic polynomial), 291
- \mathcal{NP} -complete, 292
- \mathcal{NP} -vollständig, 292
- \mathcal{NPC} , 292
- \mathcal{P} , 291
- $\chi'(G)$, 160
- $\chi(G)$, 132
- $\chi_n(G)$, 163
- $\delta(G)$, 19
- $\kappa(X, \overline{X})$, 168
- κ_o , 209
- κ_u , 209
- $\kappa(k)$, 166
- $\omega(G)$, 133
- \propto , 292
- $a \not\sim b$, 214
- $d(e, f)$, 20
- f_Δ , 170
- $g(e)$, 19
- \mathcal{W}_A , 299
- $\mathcal{W}_A(n)$, 299
- \mathcal{W}_A^∞ , 299
- $\mathcal{W}_{MIN}(P)$, 303
- 0-1-Netzwerk, 190, 202, 205, 292
- 1-Baum, 331
 - minimaler, 331
- 8-Zusammenhang, 108

A

- A^* -Algorithmus, 259, 288
- Ableitungsbaum, 54
- Abstand, 20, 242
- Adjazenzliste, 26
- Adjazenzmatrix, 25
 - bewertete, 28
- Ahuja, R.K., 282
- Algorithmus
 - A^* , 259, 282
 - approximativer, 138, 295, 296
 - ausgabesensitiver, 40
 - Dijkstra, 253, 285, 286, 288, 291
 - Dinic, 181, 190, 206, 213, 216, 218, 225, 226
 - Edmonds und Karp, 174, 176, 213
 - effizienter, 37
 - exponentieller, 290
 - Floyd, 276
 - Greedy, 296, 300, 306, 311, 325–327
 - greedy, 40
 - Huffman, 61
 - iterativer A^* , 264
 - Johnson, 306
 - Kruskal, 75, 77
 - Moore und Ford, 248, 251, 272, 282–285, 287
 - Nächster-Nachbar, 319
 - Preflow-Push, 197
 - Prim, 80, 227, 323, 406
 - probabilistischer, 295
 - Simplex, 290
 - Turner, 295
- Alt, H., 233
- Anfangsecke, 20
- Appel, K., 145, 156
- Arora, S., 305
- Ausgangsgrad, 19
- average case Komplexität, 36

B

Backtracking, 294, 325
 Backtracking-Verfahren, 141
 Baum, 22, 51, 53
 aufspannender, 52, 129, 293
 binärer, 22
 geordneter, 55
 minimal aufspannender, 74
 Baumkante, 92, 106
 Bellman, R.E., 282
 Bellmansche Gleichungen, 246
 benachbart, 19
 Bildverarbeitung, 108
 Binärbaum, 22, 56
 Birkhoffsscher Diamant, 337
 Blatt, 53
 Block, 112, 128
 Blockgraph, 112
 Brücke, 46, 356
 Branch-and-bound, 294, 317, 389
 breadth-first-search, 116
 Breitensuche, 87, 116, 174, 180, 184
 Breitensuchebaum, 116, 174
 Breitensuchenummer, 116
 Brin, S., 12, 13
 Brooks, R.L., 135, 156
 Bucket-Sort, 285
 Busstruktur, 2

C

C_n , 22
 c-Färbung, 131
 c-Kantenfärbung, 160
 Chaitin, G.J., 157
 Cherkassky, B.V., 197, 282
 Christofides, N., 321, 325
 chromatische Zahl, 132, 298
 Clique, 41, 133
 maximale, 327
 Cliquenzahl, 133
 Cobham, A., 290
 Cook, S.A., 293
 Culberson, J.C., 325

D

DAG, 94
 Dame, 258

Dantzig, G.B., 233

Datenübertragungsnetzwerk, 284
 Datenkompression, 59
 De Morgan, A., 145
 depth-first-search, 88
 Dijkstra, E.W., 253, 282
 Dinic, E.A., 181, 196
 directed acyclic graph, 94
 Directory, 54
 Dispatching, 7
 Dispatchtabelle, 8
 Distanzmatrix, 272
 divide and conquer, 24
 dominierende Menge, 329
 Dreieckssehne, 150
 Dreiecksungleichung, 318
 Durchsatz, 84, 186
 dynamisches Programmieren, 24, 317, 389

E

Ecke, 17
 aktive, 228
 innere, 22, 53
 isolierte, 19
 trennende, 110
 verwendbare, 246
 Eckenüberdeckung, 237, 300, 333
 minimale, 300, 333, 336
 eckendisjunkt, 215
 Eckenmenge
 minimal trennende, 215
 trennende, 214
 unabhängige, 137
 Eckenzusammenhangszahl, 216
 Edmonds, J., 174, 196, 282, 290
 Eigenschaft (*), 243
 Eingangsgrad, 19
 Endecke, 20
 Entscheidungsproblem, 290
 erreichbar, 21
 Erreichbarkeitsbaum, 88
 Erreichbarkeitsmatrix, 32
 Erweiterungskreis, 194
 Erweiterungsweg, 169, 172, 207, 234
 Euler, L., 321
 Eulersche Polyederformel, 145
 Eulerscher Graph, 332

Even, S., 157, 233

F

Färbbarkeit, 294

Färbung, 131, 293, 298, 305

einheitige, 239, 338

minimale, 132, 331

nicht triviale, 163

Fehler

absoluter, 297

Fertigungszelle, 5

Fibonacci-Heap, 81, 256, 272, 282

Floyd, R., 82, 276, 282

Fluß, 167

binärer, 191

blockierender, 181

kostenminimaler, 194

maximaler, 167

trivialer, 167

Wert, 167

zulässiger, 209

Ford, L.R., 196, 210, 233, 248, 252, 282

formale Sprache, 291

Fredman, M., 82

Fulkerson, D.R., 196, 210, 233

G

Güte, 297

Garey, A.R., 290, 325

geschichtetes Hilfsnetzwerk, 181, 191

Gilmore, P., 151

Gittergraph, 157

Goldberg, A.V., 197, 282

Grad, 19

Graph

k -partiter, 330

azyklischer, 94

bewerteter, 28

bipartiter, 22, 132, 204

c-kritischer, 158

eckenbewerteter, 28

Eulerscher, 321, 332

gerichtet, 18

Hamiltonscher, 315, 331

invertierter, 127

kantenbewerteter, 28

kreisfreier, 94

kritischer, 158

minimaler, 337

perfekter, 152

Petersen, 22

plättbarer, 23

planarer, 23, 124, 238, 282

regulärer, 22

schlichter, 18

transitiv orientierbarer, 150

transitiver, 150

ungerichtet, 17

vollständig bipartiter, 22

vollständiger, 22

vollständig k -partiter, 330

zufälliger, 44

zusammenhängender, 21

zyklischer, 22

greedy-Techniken, 24, 40

H

Höhe, 54, 149

Haken, W., 145, 156

Hall, P., 207, 233

Halldórsson, M.M., 314, 325

Hamiltonscher Kreis, 293, 295, 315

Handlungsreisendenproblem, 315, 335

Harary, F., 45, 338

Hassin, R., 312

Hayes, B., 13

Heap, 67, 255, 272

heapsort, 67

Heawood, P., 147

Hedetniemi, S., 338

Heesch, H., 145, 156

Herz, 46

Heuristik, 138, 295

hierarchisches Dateisystem, 54

Hoffmann, A., 151

Holyer, I., 325

Hopcroft, J.E., 124, 233

Huffman, D.A., 82

Hyperwürfel, 158

I

implizite Darstellung, 28

include-Mechanismus, 30

induzierter Untergraph, 21

Internet, 238
 Intervallgraph, 28, 163
 intractable, 290
 inzident, 19
 Inzidenzmatrix, 47

J

Jackowski, B., 156
 Johnson, D.S., 290, 306, 325, 326

K

Königsberger Brückenproblem, 131, 322
 künstliche Intelligenz, 108, 121, 258
 kürzester-Wege-Baum, 244
 Kante, 17
 gerichtete, 18
 kritische, 176
 Kantenbewertung, 241
 konstante, 252
 negative, 242
 nichtnegative, 253
 kantenchromatische Zahl, 160, 299, 332
 Kantenfärbung, 160, 299, 332
 Kantengraph, 338
 Kantenliste, 27
 Kantenmenge
 minimale trennende, 222
 trennende, 222
 Kantenzug, 20
 Länge, 242, 288
 minimaler Länge, 316
 Kantenzusammenhang, 222
 Kantenzusammenhangszahl, 222, 237
 Kapazität, 165
 Karp, R.M., 174, 196, 233, 282
 Kempe, A., 145, 147
 Kernighan, B.W., 326
 Knotenzusammenhang, 3
 Knuth, D., 45
 Kohäsion, 3
 Komplement, 21
 Komplexitätstheorie, 35
 Konfiguration, 337
 reduzible, 337
 Konfliktgraph, 9, 138
 Kosten, 74
 Kruskal, J.B., 75

Kubale, M., 156
 kW-Baum, 244

L

Lahar, A., 312
 Landausches Symbol O, 36
 late binding, 7
 Lawler, E.L., 325
 Lempel, A., 157
 Lenstra, J.K., 325
 Lewis, P.M., 325
 lexikographische Ordnung, 56
 Lin, S., 326
 lineare Programmierung, 290
 Link, 10
 Lovasz, L., 156
 Lund, C., 310, 325

M

Maheswari, S.N., 181
 Malhotra, V.M., 181
 Manzini, G., 282
 Markierungsalgorithmus, 88
 Matching, 204
 Matroid, 45
 Maximierungsproblem, 295
 Menger, V., 233
 Minimierungsproblem, 295
 mittlere Codewortlänge, 61
 Moore, E.F., 248, 252, 282
 multiple inheritance, 7
 Mycielski, J., 133, 157

N

Nachbar, 19
 Nachbarschaftsgraph, 108
 Nachfolger, 21, 53
 Netzwerk, 167
 Kosten, 194
 obere Kapazitäten, 209
 symmetrisches, 223, 236
 untere Kapazitäten, 209
 Niveau, 54, 116

O

objektorientierte Programmiersprache, 6
 Operations Research, 203, 208

Optimalitätsprinzip, 245, 317
Optimierungsproblem, 290
Ordnung, 36
Orientierung, 148
 kreisfrei, 149

P

Page, L., 12, 13
PageRank, 11, 13
Partition, 167
Permutationsdiagramm, 152
Permutationsgraph, 152
Petersen, J., 22
Petersen-Graph, 22, 23, 234, 236, 331
Platonische Körper, 337
Pnueli, A., 157
polynomiale Transformation, 292
Präfix-Codes, 60
Pramodh Kumar, M., 181
Prim, R., 80, 227, 253
Prioritäten, 71
Produkt von Graphen, 304
Public-Key Kryptosysteme, 140
Puzzle, 121

Q

q-s-Fluß, 166
q-s-Netzwerk, 165, 166
q-s-Schnitt, 167
Qualitätsgarantie
 absolute, 297
 relative, 299
Quelle, 165
Querkante, 92
Querschnitt, 84
Quicksort, 36

R

Rückwärtskante, 92, 107, 169, 213
Radzik, T., 282
Registerinterferenzgraph, 139
Reinelt, G., 326
Rekursion
 direkte, 96
 indirekte, 96
Ringstruktur, 2
Rinnoy Kan, A.H.G., 325

Robertson, N., 145
Robertson, N., 156
Robinson, R.W., 338
Robotik, 3, 262, 266
Rosenkrantz, D.J., 325
Routing-Problem, 257
Routingtabelle, 257
Roy, B., 33
RSA-System, 140

S

Safra, S., 305
Sanders, D., 145, 156
Satisfiability-Problem SAT, 293
Satz
 Ford und Fulkerson, 170, 213
 Hall, 207, 233, 389
 König-Egerváry, 238
 Menger, 215, 224, 233
 Whitney, 218, 225, 233
Schätzfunktion
 konsistente, 261, 288
 monotone, 268
 zulässige, 259
Schach, 237, 258
Scheduler, 72
Scheduling, 326
Schlüssel, 56
Schnitt, 2, 167
 maximaler, 333
 minimaler, 3, 226
Schnittgraphen, 159
Semiring, 282
Senke, 48, 165
Seymour, P., 145, 156
Shmoys, D.B., 325
Simplex-Algorithmus, 290
single inheritance, 7
Startecke, 244
Stearns, R.E., 325
Steiner Baum, 278, 335
 minimaler, 278
Steiner Ecke, 278
Sterngraph, 239
Sternstruktur, 2
Stoer, M., 227, 233
Strukturgraph, 99, 103

Suchbaum, 55
 binärer, 56
 höhenbalancierter, 59
 Suchbaumbedingung, 56
 Suche
 bidirektional, 271
 Suchmaschine, 10
 Symboltabelle, 55

T

Tait, P.G., 145
 Tarjan, R.E., 82, 124, 233
 Teilbaum mit Wurzel e , 54
 Thomas, R., 145, 156
 Tiefensuche, 87, 88, 125
 beschränkte, 123
 iterative, 123
 Tiefensuchenummer, 89
 Tiefensuchewald, 92
 topologische Sortierung, 94, 96, 99, 105,
 126, 127
 transitive Reduktion, 103
 transitiver Abschluß, 30, 31, 93, 103
 Transportproblem, 195
 Traveling-Salesman Problem, 13, 293, 315
 TSP, 316, 325
 Turing Maschine, 291
 Turner, J., 295, 325

U

Umkreissuche, 267
 unabhängige Eckenmenge, 45, 293, 304
 maximale, 333
 Unabhängigkeitszahl, 137, 297
 Untergraph, 21

V

Verbindungszusammenhang, 3
 Verletzlichkeit, 2, 214
 vermaschte Struktur, 2
 Verzeichnis, 54
 Vier-Farben-Problem, 131, 145
 Vizing, V.G., 299
 Vorgänger, 21, 53
 Vorgängermatrix, 272
 Vorrang-Warteschlange, 72
 Vorwärtskante, 92, 169, 213

W

Wagner, F., 227, 233
 Wald, 51
 Warshall, S.A., 33, 276, 282
 Web-Roboter, 10
 Weg, 20
 einfacher, 20
 geschlossener, 20
 kürzester, 3, 242, 291
 längster, 243, 292
 minimaler Länge, 316, 321
 offener, 20
 optimaler, 241
 Wegeplanungsverfahren, 13
 Whitney, H., 218
 Whitney, M., 233
 Williams, J., 82
 Windmühlengraph, 349
 Wirkungsgrad, 299
 asymptotischer, 299
 worst case Komplexität, 36
 Wurzel, 99
 Wurzelbaum, 22

Y

Yannakakis, M., 310, 325

Z

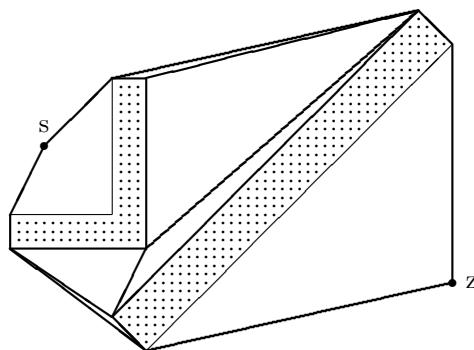
zero-knowledge Protokolle, 141
 Zuordnung, 204, 328, 331
 3-dimensionale, 333
 maximale, 204, 234, 292, 312
 minimale Kosten, 235, 323
 nicht erweiterbare, 204
 vollständige, 204
 zusammenhängend
 z -fach, 217
 quasi stark, 83
 stark, 98
 zweifach, 110
 Zusammenhangskomponente, 22, 107
 starke, 98
 Wurzel, 99
 Zusammenhangszahl, 216
 Zuverlässigkeitstheorie, 13
 zyklische Adreßkette, 78
 Zykov Baum, 372

Anhang B

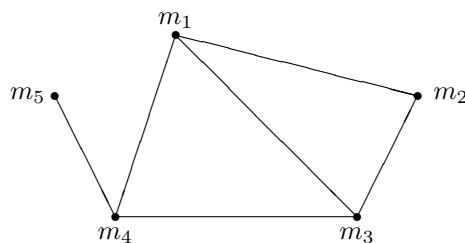
Lösungen der Übungsaufgaben

B.1 Kapitel 1

- Der Verbindungszusammenhang der drei Netzwerke von links nach rechts ist 3,3 und 2 und der Knotenzusammenhang ist 3,3 und 1.
- Die fett gezeichneten Linien bilden das System der Geradensegmente für den gesuchten Weg. Der kürzeste Weg von s nach z führt über die Segmente auf der unteren konvexen Hülle.



- Die optimale Reihenfolge der Produktionsaufträge ist 1–3–2–4 mit einer Gesamtumrüstzeit von 5.5.
- Im folgenden ist zunächst der Konfliktgraph und dann die Dispatchtabelle für die gegebene Klassenhierarchie und die angegebenen Methoden dargestellt.



Methode	m_1	m_2	m_3	m_4	m_5					
Zeile	1	2	3	2	3					
1	$m_1 A$	$m_1 B$	$m_1 C$	$m_1 D$	$m_1 A$	$m_1 F$	$m_1 G$			I
2		$m_4 B$	$m_4 B$	$m_4 B$	$m_2 E$	$m_2 E$	$m_2 E$	$m_4 H$	$m_4 H$	
3			$m_3 C$	$m_3 C$	$m_3 E$	$m_3 E$	$m_3 E$			$m_5 I$

5. Es ergibt sich folgendes Gleichungssystem

$$\begin{aligned} PR(A) &= 0.5 + 0.5 (PR(B) + PR(D)/2) \\ PR(B) &= 0.5 + 0.5 PR(C) \\ PR(C) &= 0.5 + 0.5 (PR(A)/2 + PR(D)/2) \\ PR(D) &= 0.5 + 0.5 PR(A)/2 \end{aligned}$$

mit dieser Lösung:

$$PR(A) = 6/5 \quad PR(B) = 1 \quad PR(C) = 1 \quad PR(D) = 4/5.$$

6. Es gilt

$$\sum_{W \in M} PR(W) = |M|(1 - d) + d \sum_{W \in M} \sum_{D \in PL(W)} \frac{PR(D)}{LC(D)}.$$

Da jede Web-Seite W aus M genau $LC(W)$ -mal in der rechten Doppelsumme vorkommt, gilt:

$$\sum_{W \in M} PR(W) = |M|(1 - d) + d \sum_{W \in M} LC(W) \frac{PR(W)}{LC(W)}$$

und somit ist

$$\sum_{W \in M} PR(W) = |M|.$$

D.h. die Summe der Werte des PageRank aller Seiten ist gleich der Anzahl der Seiten. Für die Menge aller existierenden Web-Seiten gilt diese Aussage nicht.

B.2 Kapitel 2

- Über die Anzahl der Ecken geraden Grades kann keine allgemeine Aussage gemacht werden. Als Beispiel betrachte man die Graphen C_n .
- Es sei G ein ungerichteter Graph mit n Ecken und m Kanten. Nach Voraussetzung gilt $3n = 2m$. Somit ist 2 ein Teiler von n .

3. Es sei G ein bipartiter Graph mit Eckenmenge $E_1 \cup E_2$ und $|E_1| = n_1$. Dann gilt $m \leq n_1(n - n_1)$. Die rechte Seite nimmt ihren größten Wert für $n_1 = n/2$ (n gerade) bzw. $n_1 = (n + 1)/2$ (n ungerade) an. In beiden Fällen folgt $4m \leq n^2$.
4. Ist Δ der Grad einer Ecke, so gilt $\Delta|E_1| = \Delta|E_2|$ bzw. $|E_1| = |E_2|$.
5. Es sei G ein ungerichteter Graph mit $m > (n - 1)(n - 2)/2$. Angenommen, die Eckenmenge von G lässt sich so in zwei nichtleere Teilmengen E_1, E_2 zerlegen, daß keine Kante Ecken aus E_1 und E_2 verbindet. Ist $|E_1| = n_1$, so gilt

$$2m \leq n_1(n_1 - 1) + (n - n_1)(n - n_1 - 1)$$

(Gleichheit gilt genau dann, wenn beide Teilgraphen vollständig sind). Die rechte Seite dieser Ungleichung wird maximal für $n_1 = 1$ bzw. $n_1 = n - 1$. Dies führt in beiden Fällen zum Widerspruch.

Die Graphen $K_{n-1} \cup K_1$ haben die angegebene Anzahl von Ecken und sind nicht zusammenhängend.

6. $K_n \cup K_m$

7. Für jede Ecke e aus G gilt $g(e) + \bar{g}(e) = 5$, wobei $\bar{g}(e)$ den Eckengrad von e im Komplement \bar{G} von G bezeichnet. Somit kann man annehmen, daß es in G eine Ecke e mit $g(e) \geq 3$ gibt. Sind die Nachbarn von e paarweise nicht benachbart, so enthält \bar{G} den Graphen C_3 und ansonsten gilt dies für G .
8. Es sei $W = (e_0, e_1, \dots, e_s)$ ein geschlossener Weg. Von e_0 aus startend, verfolge man W , bis man zum ersten Mal an eine Ecke e_j kommt, an der man schon einmal war, d.h. $e_j = e_i$ für $i < j$. Da W geschlossen ist, gibt es auf jeden Fall eine solche Ecke. Dann ist e_i, \dots, e_j ein einfacher geschlossener Weg.
9. Es sei e eine Ecke mit Eckengrad $\Delta(G)$ und L die Menge der Ecken von G , welche nicht zu e benachbart sind. Dann ist nach Voraussetzung jede Kante von G zu einer Ecke von L inzident. Nun folgt:

$$m \leq \sum_{a \in L} g(a) \leq \Delta(G)|L| = \Delta(G)(n - \Delta(G))$$

10. Für jede Teilmenge X der Eckenmenge E bezeichne G_X den von X induzierten Untergraph von G . Die Bestimmung des Herzens erfolgt in zwei Phasen. Die erste Phase bestimmt einen Untergraphen G_F von G .

```

F := ∅;
while E ≠ ∅ do begin
    wähle eine Ecke e aus E und füge e in F ein;
    entferne e und alle Vorgänger von e in G_E aus E;
end

```

Von jeder Ecke $e \in E \setminus F$ kann mit einer Kante eine Ecke in F erreicht werden. Man beachte, daß es für eine neu in F eingefügte Ecke keine Nachfolger in F geben kann. In der zweiten Phase werden nun die Ecken von F in umgekehrter Reihenfolge, in der sie in F eingefügt wurden, betrachtet.

```

H := ∅
while F ≠ ∅ do begin
    wähle aus F die zuletzt eingefügte Ecke f aus;
    füge f in H ein und entferne f und alle Vorgänger von f in G_F aus F;
end

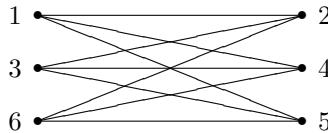
```

Zwischen den Ecken von H gibt es keine Kanten und von jeder Ecke $e \in E \setminus H$ gibt es einen Weg mit höchstens zwei Kanten zu einer Ecke in H .

11. Es sei $E_1 \cup E_2$ die Kantenmenge von G und $k = (e, f)$ die einzige Kante mit $e \in E_1$ und $f \in E_2$. Nach dem Entfernen von k haben die von E_1 und E_2 induzierten Graphen genau eine Ecke mit ungeradem Eckengrad. Dies steht im Widerspruch zu den Ergebnissen aus Abschnitt 2.1.
12. Die Adjazenzmatrix, die Adjazenzliste und die Kantenliste sehen wie folgt aus.

$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$	1 → 2 2 → 4, 6 3 → 4 4 → 5 5 → 3 6 → 1, 4, 5	<table border="1"> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr> <tr><td>1</td><td>2</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>6</td><td>6</td></tr> <tr><td>2</td><td>4</td><td>6</td><td>4</td><td>5</td><td>3</td><td>1</td><td>4</td><td>5</td></tr> </table>	1	2	3	4	5	6	7	8	9	1	2	2	3	4	5	6	6	6	2	4	6	4	5	3	1	4	5
1	2	3	4	5	6	7	8	9																					
1	2	2	3	4	5	6	6	6																					
2	4	6	4	5	3	1	4	5																					

- 13.



14. Die Funktion `nachfolger(i, j)` überprüft, ob es eine Kante von i nach j gibt.
(Zuerst folgt die einfache, danach die verbesserte Version)

```

function nachfolger (i, j : Integer) : Boolean;
var l, obergrenze : Integer;
begin
    nachfolger := false; l := N[i];
    if i = n then
        obergrenze := m
    else
        obergrenze := N[i+1]-1;
    while not nachfolger and l ≤ obergrenze do begin
        if A[l] = j then
            nachfolger := true;
        l := l + 1;
    end
end

function nachfolger (i, j : Integer) : Boolean;
var l : Integer;
begin
    nachfolger := false; l := N[i];

```

```

while not nachfolger and 1 ≤ N[i+1]-1 do begin
    if A[1] = j then
        nachfolger := true;
    l := l + 1;
end
end

```

Die Funktion `ausgrad(i)` bestimmt den Ausgrad der Ecke i (zuerst folgt die einfache, danach die verbesserte Version).

```

function ausgrad(i : Integer) : Integer;
begin
    if i = n then
        ausgrad := m + 1 - N[i]
    else
        ausgrad := N[i+1] - N[i];
end

function ausgrad(i : Integer) : Integer;
begin
    ausgrad := N[i+1] - N[i];
end

```

Die Funktion `eingrad(i)` bestimmt den Eingrad der Ecke i . Hier führt die Einführung des zusätzlichen Eintrages zu keiner Vereinfachung.

```

function eingrad(i : Integer) : Integer;
var e, l : Integer
begin
    e := 0;
    for l := 1 to m do
        if A[l] = i then
            e := e + 1;
    eingrad := e;
end

```

15. Es sei B das Feld der Länge $(n^2 - n)/2$ und A die Adjazenzmatrix des Graphen. Für natürliche Zahlen i, j setze:

$$f(i, j) = (i - 1)n - i(i + 1)/2 + j.$$

Dann gilt $A[i, j] = B[f(i, j)]$ für $i < j$ und $A[i, j] = B[f(j, i)]$ für $i > j$. Ist $i > j$, so sind die Ecken i und j genau dann benachbart, wenn $B[f(i, j)] \neq 0$ ist. Der Eckengrad $g(i)$ einer Ecke i bestimmt sich wie folgt:

$$g(i) = \sum_{k=1}^{i-1} B[f(k, i)] + \sum_{k=i+1}^n B[f(i, k)].$$

16. Um festzustellen, ob zwei Ecken i und j benachbart sind, muß für den Fall $j \geq i$ die Nachbarliste von i und im anderen Fall die von j durchsucht werden. Die Bestimmung der Anzahl der Nachbarn von i erfordert das Durchsuchen der Nachbarlisten der Ecken $1, \dots, i$. Gegenüber der normalen Adjazenzliste ändert sich der Aufwand für die erste Aufgabe nicht. Für die zweite Aufgabe ist der Aufwand jedoch höher. Für Ecke n müssen zum Beispiel alle Nachbarschaftslisten durchsucht werden, d.h. der Aufwand ist $O(n + m)$ gegenüber $O(g(n))$ bei normalen Adjazenzlisten.

17. Die Funktion `nachfolger` verwendet die Datenstruktur `Listenelement`:

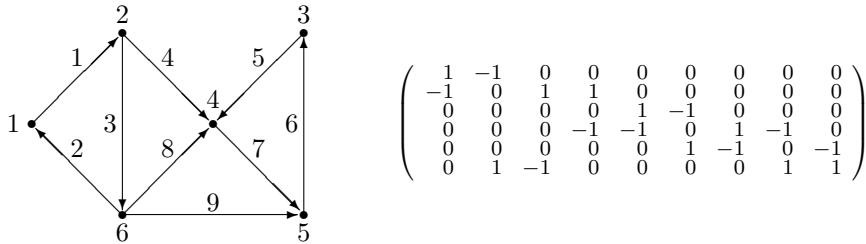
```

type Listenelement = record
    wert : Integer;
    nachfolger : zeiger Listenelement;
end

var A : array[1..max] of zeiger Listenelement;
function nachfolger(i, j : Integer) : Boolean;
var eintrag : Listenelement;
begin
    nachfolger := false;
    eintrag := A[i];
    while not nachfolger and eintrag ≠ nil and eintrag→wert ≤ j begin
        if eintrag→wert = j then
            nachfolger := true;
        eintrag := eintrag→nachfolger;
    end
end

```

18. Numeriert man die Kanten des Graphen aus Aufgabe 12 wie links dargestellt, so ergibt sich folgende Inzidenzmatrix.



Für ungerichtete Graphen entscheidet folgende Funktionen, ob j ein Nachfolger von i ist.

```

function nachfolger (i, j : Integer) : Boolean;
var k : Integer;
begin
    nachfolger := false;
    k := 1;
    while not nachfolger and k ≤ m do begin
        if C[j,k] = 1 and C[i,k] = -1 then
            nachfolger := true;
        k := k + 1;
    end
end

```

Für ungerichtete Graphen muß in der **if**-Bedingung die Zahl -1 durch 1 ersetzt werden.

Der Ausgrad einer Ecke i in einem gerichteten Graphen ist gleich der Anzahl der Einträge mit Wert 1 in der i -ten Zeile und der Eingrad gleich der der Einträge mit Wert -1 in der i -ten Zeile. Der Grad einer Ecke i in einem ungerichteten Graphen ist gleich der Anzahl der Einträge mit Wert 1 in der i -ten Zeile.

19. Adjazenzlisten basierend auf Feldern.
20. Eine Datei schließt sich genau dann selbst ein, wenn die zugehörige Ecke in dem gerichteten Graphen auf einem geschlossenen Weg liegt. Dies kann an den Diagonaleinträgen der Erreichbarkeitsmatrix erkannt werden.
21. Durch Ausmultiplizieren zeigt man, daß

$$S = A + A^2 + \dots + A^x \text{ mit } x = 2^{\lfloor \log_2 n \rfloor + 1} - 1$$

ist. Da x mindestens $n - 1$ ist, folgt mit Hilfe des in Abschnitt 2.6 bewiesenen Lemmas, daß E die Erreichbarkeitsmatrix von G ist. Zur Bestimmung von S sind $2\lfloor \log_2 n \rfloor$ Matrixmultiplikationen notwendig.

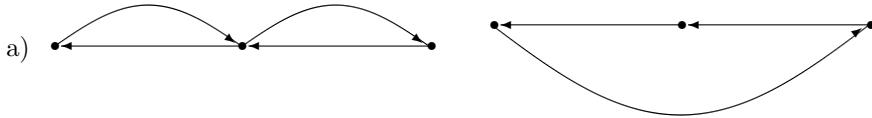
22. a) $n^2 + 2n + 3 \leq 6n^2$ für $n \geq 1$.
 b) $\log n^2 = 2 \log n$ für $n \geq 1$.
 c) $\sum_{i=1}^n i = n(n+1)/2 \leq n^2$ für $n \geq 1$.
 d) $\log n! = \sum_{i=1}^n \log i \leq n \log n$ für $n \geq 1$.
23. Es sei A die Adjazenzmatrix des Graphen. Eine Ecke i ist genau dann eine Senke, wenn die i -te Zeile von A nur Nullen enthält und die i -te Spalte bis auf den i -ten Eintrag keine Nullen enthält. Für einen Eintrag a_{ji} von A mit $i \neq j$ bedeutet dies: Ist $a_{ji} = 0$, so ist i keine Senke und ist $a_{ij} \neq 0$, so ist j keine Senke. Der folgende Algorithmus entdeckt in jedem Schritt eine Ecke, welche keine Senke ist.

```
i := 1;
for j := 2 to n do
  if A[j,i] = 0 then
    i := j;
```

Nachdem diese Schleife durchlaufen wurde, gibt es nur noch einen Kandidaten für die Senke, den letzten Wert von i . Die Überprüfung, ob i eine Senke ist, kann mit Aufwand $O(n)$ durchgeführt werden.

Liegt die Adjazenzliste des Graphen vor, so stellt man mit Aufwand $O(n)$ fest, ob es genau eine Ecke mit Ausgrad 0 gibt. Falls nicht, so gibt es keine Senke. Andernfalls überprüft man mit Aufwand $O(m)$, ob e in den Nachfolgelisten aller anderen Ecken vorkommt. Falls ja, so ist e eine Senke und ansonsten gibt es keine Senke.

24. Folgende einfache Eigenschaft von transitiven Reduktionen wird im folgenden verwendet: Es sei $k = (e, f)$ eine Kante in einer transitiven Reduktion R . Dann verwendet jeder Weg in R von e nach f die Kante k .



- b) Angenommen es gibt zwei verschiedene transitive Reduktionen R_1 und R_2 . Es sei $k = (e, f)$ eine Kante in $R_1 \setminus R_2$. Da f von e in R_2 erreichbar sein muß, gibt es eine Ecke v , so daß es in R_2 einen Weg von e über v nach f gibt. Da es in R_1 keine geschlossenen Wege gibt, existieren dort Wege von e nach v und von v nach f , welche k nicht enthalten. Dies kann aber nicht sein.
- c) Es sei G der vollständig bipartite Graph mit Eckenmenge $E_1 \cup E_2$ und $|E_1| = |E_2| = n/2$. Jede Kante (e_1, e_2) mit $e_1 \in E_1$ und $e_2 \in E_2$ wird nun in Richtung von den Ecken aus E_2 orientiert. Dieser Graph hat genau die angegebene Eigenschaft.
- d) Es sei R die transitive Reduktion eines gerichteten Graphen ohne geschlossene Wege. Angenommen es gibt eine Kante $k = (e, f)$ in R , mit $\text{maxlen}(e, f) > 1$ ist, d.h. es gibt einen Weg W von e nach f , welcher mindestens zwei Kanten enthält. Da der Graph keine geschlossenen Wege besitzt, ist k nicht in W enthalten. Dies führt zum Widerspruch.
25. Der Algorithmus verwendet die in Aufgabe 15 angegebene Funktion f , um Paare von Ecken in einem eindimensionalen Feld A zu indizieren. Das Feld wird mit 0 initialisiert. Haben zwei Ecken i und j einen gemeinsamen Nachbarn, so wird dieser an der Stelle $A[f(i, j)]$ eingetragen.

```

var A : array[1..(n-1)n/2] of Integer;
Initialisiere A mit 0;
for jede Ecke e do
  for jeden Nachbar i von e do
    for jeden Nachbar j ≠ i von e do
      if A[f(i,j)] ≠ 0 then
        exit('e,i,j und A[f(i,j)] bilden einen geschlossenen Weg');
      else
        A[f(i,j)] := e;
exit('Es gibt keinen geschlossenen Weg mit vier Ecken');

```

Wurde der Algorithmus nach $n(n-1)/2$ Schritten noch nicht beendet, so sind alle Einträge von A ungleich 0 und der Algorithmus endet im nächsten Schritt. Der Aufwand ist $O(n^2)$.

Um festzustellen, ob ein Graph einen Untergraph vom Typ $K_{s,s}$ hat, muß ein Feld der Länge ($\binom{n}{s}$) betrachtet werden. Ferner werden für jede Ecke e alle s -elementigen Teilmengen der Nachbarn von e betrachtet. Daraus ergibt sich ein Aufwand von $O(n^s)$.

26. Es sei $k = (i, j)$ die zusätzliche Kante und $k' = (l, s)$ eine Kante, welche neu im transitiven Abschluß liegt (d.h. $E'[l, s] = 1$ und $E[l, s] = 0$). Dann muß es einen einfachen Weg von l nach s geben, welcher k enthält. Da E die Erreichbarkeitsmatrix ist, gilt: $E[l, i] = 1$ und $E[j, s] = 1$. D.h. jede neue Kante geht von einem Vorgänger von i zu einem Nachfolger von j . Folgender Algorithmus bestimmt E' mit Aufwand $O(n^2)$.

```

Initialisiere E' mit E;
for l := 1 to n do
    if E[l, i] = 1 then
        for s := 1 to n do
            if E[j, s] = 1 then
                E'[l, s] = 1;

```

Es sei G ein Graph, welcher aus zwei Teilgraphen G_1 und G_2 mit je $n/2$ Ecken besteht, so daß es keine Kante von G_2 nach G_1 gibt. Ferner besitze G_1 eine Ecke j , von der aus jede Ecke aus G_1 erreichbar ist, und in G_2 eine Ecke i , welche von jeder Ecke aus G_2 erreichbar ist. Wird die Kante (i, j) neu in G eingefügt, so hat der neue transitive Abschluß $n^2/4$ zusätzliche Kanten.

27. Der angegebene Algorithmus ist ein Greedy-Algorithmus. Eine Implementierung mit Aufwand $O(n + m)$ kann mit den in Abschnitt 2.8 vorgestellten Datenstrukturen erreicht werden. Der einzige Unterschied besteht darin, daß die Ecke mit kleinstem Eckengrad entfernt wird, hierzu wird ein Zeiger auf das Ende der Liste `EckengradListe` verwaltet. Ferner werden noch Zähler für die Anzahl der in C verbliebenen Kanten und Ecken verwaltet. Mit diesen Zählern kann einfach überprüft werden, ob C eine Clique ist.

B.3 Kapitel 3

1. Es sei G ein zusammenhängender Graph.
 - a) Ist G ein Baum, so gilt $m = n - 1 < n$. Gilt umgekehrt $n > m$, so betrachte man einen aufspannenden Baum B von G . Da B $n - 1$ Kanten hat, gilt $m = n - 1$ und somit $G = B$.
 - b) Wenn G einen einzigen geschlossenen Weg W enthält, so erhält man einen aufspannenden Baum, falls man eine beliebige Kante aus W entfernt. Somit hat G genau $n - 1 + 1 = n$ Kanten. Ist umgekehrt die Anzahl der Kanten in G gleich der Anzahl der Ecken, so besteht G aus einem aufspannenden Baum und einer zusätzlichen Kante. Diese bewirkt, daß es in G genau einen geschlossenen Weg gibt.
2. Ein Graph W_n mit Eckenmenge $\{1, \dots, n\}$ und n ungerade heißt *Windmühlen-graph*, wenn er folgende Kantenmenge besitzt:

$$\{(1, i) | i = 2, \dots, n\} \cup \{(i, i+1) | i = 2, 4, 6, \dots, n-1\}.$$

Der Graph W_n hat $3(n - 1)/2$ Kanten. Die Graphen W_n haben die angegebene Eigenschaft. Im folgenden wird bewiesen, daß ein Graph G mit der angegebenen Eigenschaft ein Windmühlengraph ist.

Es sei e eine beliebige Ecke und G_e der von e und den Nachbarn von e induzierte Untergraph. Man sieht leicht, daß G_e ein Windmühlengraph ist. Falls alle Ecken den Grad 2 haben, so ist $G = W_3 = K_3$. Hat G eine Ecke e mit $g(e) > 2$, so daß alle anderen Ecken zu e inzident sind, so ist G ebenfalls ein Windmühlengraph. Angenommen G hat keine dieser beiden Eigenschaften. Es sei e eine Ecke mit maximalem Eckengrad Δ . Dann ist $\Delta > 2$. Es sei N die Menge der Nachbarn von e und $f \neq e$ eine Ecke, welche nicht in N ist. Für jede Ecke $u \in N$ gibt es genau eine Ecke a_u , so daß f, a_u, u ein Weg ist. Für $u_1 \neq u_2$ gilt $a_{u_1} \neq a_{u_2}$, sonst wäre e, u_1, a_{u_1}, u_2, e ein geschlossener Weg der Länge vier. Somit gilt:

$$\Delta \geq g(f) \geq g(e) = \Delta, \text{ bzw. } g(f) = g(e)$$

Im folgenden wird gezeigt, daß G regulär ist. Dazu muß noch gezeigt werden, daß $g(e) = g(n)$ für jede Ecke $n \in N$ gilt. Wiederholt man die Argumentation aus dem letzten Abschnitt mit f anstelle von e , so folgt daß der Eckengrad jeder nicht zu f benachbarten Ecke ebenfalls Δ ist. Somit verbleibt maximal ein Nachbar n von e , von dem noch nicht gezeigt wurde, daß $g(n) = \Delta$ ist. Wiederholt man die letzte Argumentation für einen Nachbarn $n_1 \neq n$ von e , so folgt auch $g(n) = \Delta$, d.h. G ist regulär.

Es sei A die Adjazenzmatrix von G . Da G regulär ist, folgt aus der Voraussetzung $A^2 = (\Delta - 1)I + E$. Hierbei ist I die Einheitsmatrix und E die Matrix, deren Einträge alle gleich 1 sind. Die Matrizen A und E sind symmetrisch und es gilt $AE = EA = \Delta I$. Somit sind A und E simultan diagonalisierbar. Die Eigenwerte von E sind 1 und 0 mit den Vielfachheiten 1 und $n - 1$. Der Eigenvektor zum Eigenwert 1 ist $v_1 = (1, \dots, 1)$. Als Eigenvektor von A hat v_1 den Eigenwert Δ . Es sei v ein weiterer gemeinsamer Eigenvektor. Dann ist $Ev = 0$ und somit $A^2v = (\Delta - 1)v$ bzw. $Av = \pm\sqrt{\Delta - 1}v$. Somit sind $\Delta, \sqrt{\Delta - 1}$ und $-\sqrt{\Delta - 1}$ die Eigenwerte von A mit den Vielfachheiten 1, r und s . Es gilt $n = 1 + r + s$. Betrachtet man die Spur von A , so folgt $s - r = \Delta/\sqrt{\Delta - 1}$. Da $s - r$ eine ganze Zahl ist, muß $\Delta - 1$ eine Quadratzahl sein, deren Wurzel Δ teilt. Hieraus folgt $\Delta = 2$ im Widerspruch zur Annahme. Somit ist G ein Windmühlengraph.

3. Die Aussage wird durch vollständige Induktion bewiesen. Für $n = 2$ ist $d_1 = d_2 = 1$ und der Graph C_2 ist der gesuchte Baum. Sei nun $n > 2$. Es muß Indizes i, j mit $d_i = 1$ und $d_j > 1$ geben. Entfernt man d_i und d_j aus der Zahlenfolge und fügt die Zahl $d_j - 1$ ein, so kann die Induktionsvoraussetzung angewendet werden. An den so erhaltenen Baum muß nur noch eine zusätzliche Kante an die Ecke mit Grad $d_j - 1$ angehängt werden.
4. Würde es in \overline{B} drei Zusammenhangskomponenten geben, so gäbe es in B einen geschlossenen Weg der Länge drei. Da B ein Baum ist, kann das nicht sein. Nach Voraussetzung besteht B somit aus zwei Zusammenhangskomponenten. Gäbe es in einer der beiden Zusammenhangskomponenten zwei nicht inzidente Ecken, so würde dies wieder zu einem Widerspruch führen. Somit sind beide Zusammenhangskomponenten vollständige Graphen. Hätten beide Komponenten mehr als

eine Ecke, so gäbe es einen geschlossenen Weg der Länge vier in B . Somit muß eine der beiden Zusammenhangskomponenten aus genau einer Ecke bestehen.

5. Ein Binärbaum mit der angegebenen Eigenschaft hat $2b - 1$ Ecken. Der Beweis erfolgt durch vollständige Induktion. Für $b = 1$ ist die Aussage klar. Es sei nun $b > 1$ und e eine Ecke auf dem vorletzten Niveau von B . Dann hat e genau zwei Nachfolger. Entfernt man diese aus B , so erhält man einen Binärbaum mit $b - 1$ Blätter für den die Voraussetzung erfüllt ist. Somit hat dieser Baum $2(b - 1) - 1$ Blätter. Daraus folgt die Aussage.
6. Der Beweis erfolgt durch vollständige Induktion nach der Anzahl der inneren Ecken. Für $i = 1$ ist die Aussage klar. Sei nun $i > 1$ und x eine Ecke auf dem vorletzten Niveau. Entfernt man alle Nachfolger von x , so gilt für den entstandenen Wurzelbaum B_x nach Induktionsvoraussetzung:

$$|B_x| = \sum_{e \in E \setminus B, e \neq x} (\text{anz}(e) - 1) + 1.$$

Da $|B| = |B_x| + \text{anz}(x) - 1$ gilt, ist die Aussage bewiesen.

7. Die Anzahl der Blätter in einem Binärbaum ist genau dann maximal, wenn alle Niveaus voll besetzt sind. Somit hat ein Binärbaum der Höhe h maximal 2^h Blätter.
8. Es sei G quasi stark zusammenhängend. Folgendes Verfahren bestimmt eine Wurzel w von G . Es sei zunächst w eine beliebige Ecke von G . Gibt es eine Ecke e , welche nicht von w erreichbar ist, so muß es eine Ecke v geben, so daß e und w von v erreichbar sind. Setze w gleich v und wiederhole das Verfahren bis alle Ecken von w erreichbar. Besitzt G eine Wurzel, so folgt direkt, daß G auch quasi stark zusammenhängend ist.
9. Es werden folgende Typen zur Darstellung von Dateien und Verzeichnissen verwendet.

```

Eintrag = record
    name : String;
    verzeichnis : Boolean;
    inhalt : zeiger Eintragliste;
    daten : Inhaltstyp;
  end

Eintragliste = record
    inhalt : zeiger Eintrag;
    nachbar : zeiger Eintragliste;
  end

```

Folgende Prozedur gibt die Namen aller Dateien im angegebenen Verzeichnis und allen darunterliegenden Unterverzeichnissen aus.

```

procedure ausgabeVerzeichnis(e : zeiger Eintrag);
var naechster : zeiger Eintragliste;
begin
  if e ≠ nil then begin
    if e → verzeichnis then begin
      ausgabe(Verzeichnis: , e → name);
      naechster := e → inhalt;
    
```

```

        while naechster ≠ nil do begin
            ausgabeVerzeichnis(naechster → inhalt);
            naechster := naechster → nachbar;
        end
    else
        ausgabe(Datei: , e → name);
    end
end

```

10. Um einen Eintrag mit dem Schlüssel x in einem binären Suchbaum zu löschen, muß zunächst die zugehörige Ecke e lokalisiert werden. Falls e ein Blatt oder genau einen Nachfolger hat, so kann das Löschen einfach durchgeführt werden. Falls jedoch e zwei Nachfolger hat, so ist der Vorgang aufwendiger. Ein Algorithmus zum Löschen von Einträgen arbeitet wie folgt.
- Falls e keine Nachfolger hat, so lösche e .
 - Falls e genau einen Nachfolger hat, so ersetze e durch seinen Nachfolger.
 - Falls e zwei Nachfolger hat, so ersetze e durch die Ecke mit dem größten Schlüssel im linken Teilbaum oder durch die Ecke mit dem kleinsten Schlüssel im rechten Teilbaum von e . Um die Ecke mit dem größten (kleinsten) Schlüssel im linken (rechten) Teilbaum zu finden, verfolge man vom linken (rechten) Nachfolger der Ecke startend immer den rechten (linken) Nachfolger bis ein Blatt erreicht ist, dies ist die gewünschte Ecke.
11.

```

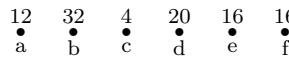
procedure aufsteigendeReihenfolge(b : Suchbaum);
begin
    if b ≠ nil then begin
        aufsteigendeReihenfolge(b → linkerNachfolger);
        ausgabe(b → Daten);
        aufsteigendeReihenfolge(b → rechterNachfolger);
    end
end

```
12. Bei einem Binärbaum mit zehn Ecken minimaler Höhe sind die ersten drei Ebenen voll besetzt und auf der vierten Ebene sind drei Ecken. Der Baum hat also die Höhe drei. Entartet der Suchbaum zu einer linearen Liste, so liegt ein Binärbaum maximaler Höhe vor (vergleichen Sie Abbildung 3.9).
13. Liegt a auf einem höheren Niveau als b , so ist $l_a > l_b$. Wäre die Wahrscheinlichkeit von a echt kleiner als die von b , so könnte man die Codierung von a und b vertauschen und so einen Präfix-Code mit kleinerer mittlerer Wortlänge erhalten. Dies widerspricht der Optimalität des Huffman-Algorithmus, die im folgenden bewiesen wird.
- Es seien p_1, \dots, p_n die Häufigkeiten der Zeichen in aufsteigender Reihenfolge. Im folgenden werden die Ecken mit ihren Häufigkeiten identifiziert. Zunächst wird gezeigt, daß es einen Präfix-Code minimaler Wortlänge gibt, in dessen Binärbaum es auf dem vorletzten Niveau eine Ecke gibt, deren Nachfolger p_1 und p_2 sind. Dazu betrachte man eine beliebige Ecke x auf dem vorletzten Niveau. Es seien c und d mit $c \leq d$ die Nachfolger von x . Falls $c = p_1$ und $d = p_2$, so ist die Aussage

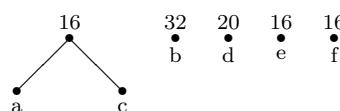
erfüllt. Ist $d > p_2$, so folgt aus der Minimalität des Codes $l_{p_2} \geq l_d$. Da d auf dem untersten Niveau liegt, gilt $l_d \geq l_{p_2}$. Somit ist $l_d = l_{p_2}$ und analog $l_c = l_{p_1}$. Vertauscht man nun p_2 und d bzw. p_1 und c im Binärbaum, so erhält man den gewünschten Präfix-Code.

Die Optimalität des vom Huffman-Algorithmus erzeugten Präfix-Code wird durch vollständige Induktion nach n bewiesen. Für $n = 1, 2$ ist die Optimalität offensichtlich gegeben. Sei nun $n > 2$ und B ein optimaler Binärbaum für p_1, \dots, p_n , in dem p_1 und p_2 Brüder sind. Man entferne die zu p_1 und p_2 gehörenden Blätter und markiere das neu entstandene Blatt mit $p_1 + p_2$. Es sei B' der neue Baum. Dies ist ein Binärbaum für die Häufigkeiten $p_1 + p_2, p_3, \dots, p_n$. Für die mittleren Wortlängen l_B von B und $l_{B'}$ von B' gilt $l_B = l_{B'} + p_1 + p_2$. Sei nun B'_1 ein vom Huffman-Algorithmus erstellter Binärbaum für die Häufigkeiten $p_1 + p_2, p_3, \dots, p_n$. Dann ist B'_1 nach Induktionsvoraussetzung optimal. Es sei B_1 der Baum, welcher aus B'_1 hervorgeht, indem die Ecke $p_1 + p_2$ zwei Nachfolger mit den Häufigkeiten p_1 und p_2 bekommt. B_1 ist gerade der vom Huffman-Algorithmus erstellte Baum für die Häufigkeiten p_1, \dots, p_n . Nun gilt $l_{B_1} = l_{B'_1} + p_1 + p_2$. Nun kann aber l_{B_1} nicht echt größer als l_B sein, denn dies würde $l_{B'_1} > l_{B_1}$ implizieren, was wegen der Optimalität von B'_1 nicht gilt. Somit ist $l_{B_1} = l_B$, d.h. B_1 ist optimal.

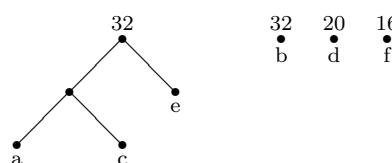
14. a)



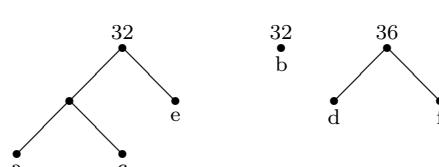
b)



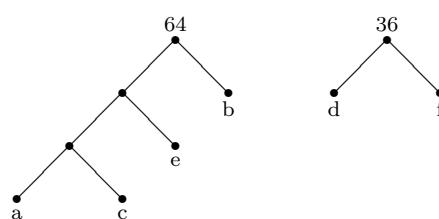
c)

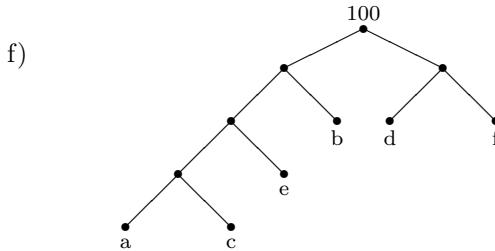


d)



e)





Der erzeugte Präfix-Code hat eine mittlere Codewortlänge von 2.48.

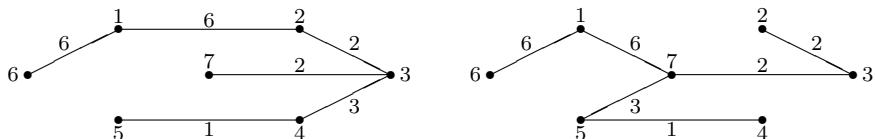
a:	0 0 0 0	c:	0 0 0 1	e:	0 0 1
b:	0 1	d:	1 0	f:	1 1

15.

```
var huffman : Wald;
procedure codewörter(wurzel : Integer);
var S : stapel of Integer;
begin
  if huffman.Ecken[wurzel].Bewertung ist ein Zeichen then begin
    ausgabe(Code für , huffman.Ecken[wurzel].Bewertung, ist:);
    ausgabe(Inhalt von S von unten nach oben);
  end
  else begin
    S.einfügen(0);
    codewörter(huffman.Ecken[wurzel].linkerNachfolger);
    S.entfernen;
    S.einfügen(0);
    codewörter(huffman.Ecken[wurzel].rechterNachfolger);
    S.entfernen;
  end
end
```

Der Aufruf `codewörter(huffman.wurzeln[1])` gibt den Präfix-Code aus.

16. Die Kosten eines minimal aufspannenden Baumes betragen 20.



17. Ein aufspannender Baum B von G , der k enthält, heißt k -minimal aufspannender Baum von G , falls kein anderer aufspannender Baum B' von G existiert, dessen Kosten niedriger sind und der k enthält. Der Beweis der folgenden Aussage erfolgt analog zum Beweis des ersten Lemmas in Abschnitt 3.6.

Es sei G ein kantenbewerteter zusammenhängender Graph mit Eckenmenge E und $k = (e, f)$ eine Kante von G . Ferner sei U eine Teilmenge von E mit $e, f \in U$ und (u, v) eine Kante mit minimalen Kosten mit $u \in U$ und $v \in E \setminus U$. Dann existiert ein k -minimal aufspannender Baum von G , der die Kante (u, v) enthält.

Es seien nun k_0, k_1, \dots, k_{m-1} die Kanten von G mit $k_0 = k$, wobei die Kanten k_1, \dots, k_{m-1} nach aufsteigenden Bewertungen sortiert sind. Der Algorithmus von

Kruskal wird auf G angewendet, wobei die Kanten in der angegebenen Reihenfolge betrachtet werden. Auf diese Art entsteht ein aufspannender Baum von G , welcher k enthält. Der Beweis, daß dies auch ein k -minimal aufspannender Baum ist, erfolgt analog zum Korrektheitsbeweis des Algorithmus von Kruskal.

18. Zur Konstruktion eines maximal aufspannenden Baumes kann jeder Algorithmus zur Bestimmung eines minimal aufspannenden Baumes verwendet werden. Dazu muß nur die Relation \leq durch \geq ersetzt werden. Eine andere Möglichkeit besteht darin, die Bewertung b_k jeder Kante k auf $C - b_k$ zu ändern, wobei C die höchste Bewertung aller Kanten ist. Danach wird für diese Bewertung ein minimal aufspannender Baum bestimmt. Dieser ist ein maximal aufspannender Baum für die ursprüngliche Bewertung.
19. Es sei B ein maximal aufspannender Baum von G und e, f Ecken in G . Ferner sei W ein Weg von e nach f in G mit maximalem Querschnitt und \bar{W} der Weg in B von e nach f . Im folgenden wird gezeigt, daß die Querschnitte von W und \bar{W} übereinstimmen. Es sei $k = (k_a, k_b)$ eine Kante von W , welche nicht in B liegt. Fügt man k in B ein, so entsteht ein geschlossener Weg W' . Da B ein maximal aufspannender Baum ist, gilt $\text{Bewertung}(k') \geq \text{Bewertung}(k)$ für alle Kanten k' von W' . Somit ist der Querschnitt des Weges von k_a nach k_b in B mindestens so groß wie die Bewertung von k . Hieraus ergibt sich, daß der Querschnitt von \bar{W} mindestens so groß ist wie der von W . Da der Querschnitt von W maximal ist, stimmen beide überein. Um den Durchsatz aller Paare zu bestimmen, wird zunächst ein maximal aufspannender Baum B bestimmt. Von jeder Ecke e von G wird eine Tiefensuche in B gestartet und dabei der Durchsatz für alle $n - 1$ Paare e, f bestimmt. Dabei wird mit Hilfe eines Stapels die maximale Bewertung auf dem aktuellen Weg gespeichert. Der Aufwand hierfür ist $O(n)$. Zusätzlich zur Bestimmung eines maximal aufspannenden Baums erfordert das Verfahren einen Aufwand von $O(n^2)$.
20. Für eine Kante k aus einem aufspannenden Baum T besteht der Graph $T \setminus \{k\}$ aus zwei Zusammenhangskomponenten mit Eckenmengen E_1, E_2 . Setze

$$K_k(T) = \{(a, b) \text{ Kante von } G \mid a \in E_1, b \in E_2\}.$$

Es gilt $K_k(T) \cap T = \{k\}$. Es sei T' ein minimal aufspannender Baum von G und T der vom Algorithmus erzeugte Baum. Ferner sei $k \in T \setminus T'$ und W der geschlossene Weg in $T' \cup \{k\}$. Für alle $l \in K_k(T) \cap W$ mit $l \neq k$ gilt $l \in T', l \notin T$ und $k \in K_l(T')$. Da l nicht in T liegt, wurde l durch den Algorithmus entfernt. D.h. l war die Kante mit der höchsten Bewertung in einem geschlossenen Weg. Somit ist die Bewertung von l mindestens so groß wie die Bewertung jeder Kante $k' \neq l$ aus dem geschlossenen Weg in $T \cup \{l\}$. Da $l \in K_k(T)$ ist, muß k auf diesem Weg liegen. Somit gilt $\text{Bewertung}(l) \geq \text{Bewertung}(k)$. Da T' ein minimal aufspannender Baum ist, kann nicht $\text{Bewertung}(l) > \text{Bewertung}(k)$ gelten. Somit stimmen die Bewertungen von l und k überein. Ersetzt man in T' die Kante l durch k , so erhält man einen neuen minimalen aufspannenden Baum T'' mit $|T'' \cap T| > |T' \cap T|$. Durch Wiederholung dieser Vorgehensweise zeigt man, daß auch T ein minimal aufspannender Baum von G ist.

21. Hat ein Graph eine *Brücke* (siehe Aufgabe 11 in Kapitel 2), deren Bewertung größer ist als die aller anderen Kanten, so liegt diese in jedem aufspannenden Baum.
22. Angenommen es gibt zwei verschiedene minimal aufspannende Bäume B_1 und B_2 . Es sei k_1 eine Kante aus B_1 , welche nicht in B_2 liegt. Fügt man k_1 in B_2 ein, so entsteht ein geschlossener Weg W . Da B_2 ein minimal aufspannender Baum ist und die Kanten verschiedene Bewertungen haben, sind die Bewertungen aller Kanten aus $W \cap B_2$ echt größer als die von k_1 . Da B_1 ein Baum ist, muß es in $W \setminus B_1$ eine Kante k_2 mit $\text{Bewertung}(k_1) < \text{Bewertung}(k_2)$ geben. Dieses Argument kann nun wiederholt werden. So entsteht eine unendliche Folge von Kanten k_1, k_2, \dots aus G mit echt aufsteigenden Gewichten. Dieser Widerspruch zeigt, daß es nur einen minimal aufspannenden Baum gibt.
23. Für kleine Werte von p ist der Algorithmus von Kruskal überlegen, während für Werte von p in der Nähe von 1 der Algorithmus von Prim schneller ist. Bei welchem Wert von p der Übergang erfolgt, hängt von der konkreten Implementierung der Algorithmen ab.
24. Die spezielle Eigenschaft der Bewertung kann ausgenutzt werden, um das Sortieren der Kanten nach ihren Bewertungen mit Aufwand $O(m + C)$ durchzuführen (Zeit und Speicher). Der Algorithmus von Kruskal hat in diesem Fall den Aufwand $O(m + C + \log n)$.
25. Der Beweis erfolgt durch vollständige Induktion nach der Anzahl n der Ecken. Für $n = 2$ ist die Aussage klar. Es sei $n > 1$ und k die Kante mit der kleinsten Bewertung und G' der Graph, welcher durch das Verschmelzen der Entdecken von k und das Ändern der Bewertungen entsteht. Nach Induktionsvoraussetzung bestimmt der Algorithmus einen minimal aufspannenden Baum B' von G' . Nach dem in Abschnitt 3.6 bewiesenen Satz gibt es einen minimal aufspannenden Baum B von G , welcher k enthält. Entfernt man k aus diesem Baum und ändert wie angegeben die Bewertungen der Kanten, so erhält man einen aufspannenden Baum für G' . Es gilt:

$$\text{Kosten}(B') \leq \text{Kosten}(B) - n \text{Bewertung}(k)$$

B' ist auch ein Baum in G . Fügt man die Kante k zu B' hinzu und ändert wieder die Bewertungen, so erhält man einen aufspannenden Baum von G mit Kosten $n\text{Bewertung}(k) + \text{Kosten}(B')$. Aus obiger Gleichung folgt, daß dieser Baum ein minimal aufspannender Baum von G ist. Dies ist gleichzeitig auch der Baum, den der Algorithmus konstruiert.

26. Es sei G' der Graph, welcher aus B , der neuen Ecke e und den neuen Kanten besteht. Nach Aufgabe 23 ist ein minimal aufspannender Baum von G' auch ein minimal aufspannender Baum von G . Wendet man den dort beschriebenen Algorithmus an, so müssen nur die neuen Kanten in B eingefügt und entsprechende Kanten entfernt werden. Dies kann mit Aufwand $O(ng(e))$ durchgeführt werden. Da der Graph G' $n-1+g(e) < 2n$ Kanten hat, bestimmt sowohl der Algorithmus von Kruskal als auch der von Prim in diesem Fall einen minimal aufspannenden Baum mit Aufwand $O(n \log n)$.

B.4 Kapitel 4

1. Im folgenden ist die Aufrufhierarchie der Prozedur Tiefensuche, die Numerierung der Ecken und der Tiefensuche-Wald angegeben.

a) tiefensuche(1)
 tiefensuche(2)
 tiefensuche(3)
 tiefensuche(4)
 tiefensuche(7)
 tiefensuche(6)
 tiefensuche(8)
 tiefensuche(5)

b) tiefensuche(1)
 tiefensuche(5)
 tiefensuche(3)
 tiefensuche(6)
 tiefensuche(9)
 tiefensuche(2)
 tiefensuche(4)
 tiefensuche(7)
 tiefensuche(8)

Ecke	1	2	3	4	5	6	7	8
Nr.	1	2	3	4	8	6	5	7

1 → 2 2 → 3,5
 3 → 4,7 4 → –
 5 → – 6 → –
 7 → 6,8 8 → –

Ecke	1	2	3	4	5	6	7	8	9
Nr.	1	6	3	7	2	4	8	9	5

1 → 5 2 → 4
 3 → 6 4 → 7,8
 5 → 3 6 → 9
 7 → – 8 → –
 9 → –

2. Im folgenden ist die Aufrufhierarchie der Prozedur Tiefensuche, die Numerierung der Ecken und der Tiefensuche-Wald angegeben.

a) tiefensuche(1)
 tiefensuche(2)
 tiefensuche(6)
 tiefensuche(4)
 tiefensuche(3)
 tiefensuche(5)

b) tiefensuche(1)
 tiefensuche(2)
 tiefensuche(3)
 tiefensuche(4)
 tiefensuche(5)
 tiefensuche(6)

Ecke	1	2	3	4	5	6
Nr.	1	2	5	4	6	3

1 → 2 2 → 6,3
 3 → – 4 → –
 5 → – 6 → 4

Ecke	1	2	3	4	5	6
Nr.	1	2	3	4	5	6

1 → 2 2 → 3
 3 → 4 4 → –
 5 → 6 6 → –

3. a) Es genügt, den Fall $\text{TSB}[e] < \text{TSB}[f]$ zu betrachten. Gilt $\text{TSE}[e] < \text{TSB}[f]$, so sind die Intervalle disjunkt. Andernfalls erfolgt der Aufruf von **tiefensuche(f)** innerhalb des Aufrufs **tiefensuche(e)**. Somit muß $\text{TSE}[f] < \text{TSE}[e]$ gelten, d.h. es gilt die dritte Bedingung.
 b) Ist k eine Baum- oder Vorwärtskante, so erfolgt der Aufruf von **tiefensuche(f)** innerhalb des Aufrufs **tiefensuche(e)** und endet deshalb auch vor diesem, d.h.

$$\text{TSB}[e] < \text{TSB}[f] < \text{TSE}[f] < \text{TSE}[e].$$

Gilt umgekehrt diese Ungleichungskette, so erfolgte der Aufruf von **tiefensuche(f)** innerhalb von **tiefensuche(e)**. Erfolgte der Aufruf direkt, so ist k eine Baumkante und andernfalls eine Vorwärtskante.

Ist k eine Rückwärtskante, so erfolgt der Aufruf von **tiefensuche(e)** innerhalb **tiefensuche(f)** und endet deshalb auch vor diesem, d.h.

$$\text{TSB}[f] < \text{TSB}[e] < \text{TSE}[e] < \text{TSE}[f].$$

Gilt umgekehrt diese Ungleichungskette, so erfolgte der Aufruf von **tiefensuche(e)** innerhalb von **tiefensuche(f)**, d.h. f ist Vorgänger von e und k ist somit eine Rückwärtskante.

Die Kante k ist genau dann eine Querkante, wenn der Aufruf von **tiefensuche(f)** schon vor dem Aufruf **tiefensuche(e)** endet. D.h. k ist genau dann eine Querkante, wenn

$$\text{TSB}[f] < \text{TSE}[f] < \text{TSB}[e] < \text{TSE}[e]$$

gilt.

4. Die Aussage ist wahr. Innerhalb des Aufrufs **tiefensuche(e)** werden alle von e erreichbaren unbesuchten Ecken besucht. Wegen $\text{TSNummer}[e] < \text{TSNummer}[f]$, muß f im Tiefensuchebaum von e erreichbar sein.
5. Nein, für die Kante $(4, 3)$ ist die Bedingung einer topologischen Sortierung nicht erfüllt.
6. Die Aufrufhierarchie der Prozedur **tsprozedur** sieht wie folgt aus:

```
tsprozedur(1)
  tsprozedur(3)
    tsprozedur(4)
      tsprozedur(7)
    tsprozedur(5)
      tsprozedur(6)
  tsprozedur(2)
```

Ecke	1	2	3	4	5	6	7
Sortierungsnummer	1	2	3	6	4	5	7

7. Beide Graphen haben keine topologische Sortierung, da sie geschlossene Wege haben $((4,2,6,4)$ bzw. $(2,3,4,2)$).
8. Der erste Graph besitzt keine topologische Sortierung, da er den geschlossenen Weg $(1, 3, 4, 1)$ besitzt. Die beiden anderen Graphen besitzen eine topologische Sortierung, die Sortierungsnummern sind $(2, 1, 3, 4, 5, 6)$ und $(1, 3, 2, 4, 5, 6)$ (in der Reihenfolge aufsteigender Eckenzahlen).
 - a)

```
topsort(1)
  tsprozedur(1)
  tsprozedur(2)
  tsprozedur(3)
  tsprozedur(4)
  exit()
```
 - b)

```
topsort(1)
  tsprozedur(1)
  tsprozedur(3)
  tsprozedur(4)
  tsprozedur(5)
  tsprozedur(6)
  topsort(2)
  tsprozedur(2)
```
 - c)

```
topsort(1)
  tsprozedur(1)
  tsprozedur(2)
  tsprozedur(4)
  tsprozedur(5)
  tsprozedur(6)
  tsprozedur(3)
```
9. Die Adjazenzmatrix hat auf und unterhalb der Diagonalen nur Nullen.

10. Nein, dies zeigt eine topologische Sortierung, die zwischen zwei Teilgraphen *hin-* und *herspringt*. Betrachten Sie folgendes Beispiel:

Adjazenzliste: $1 \rightarrow 2, 4 \quad 2 \rightarrow 3$
 Numerierung: **1:1** **2:2** **3:4** **4:3**

11.

```
function Höhe(i : Integer) : Integer;
var hmax : Integer;
begin
  hmax := 0;
  for jeden Nachfolger j von i do
    hmax := max(hmax, Höhe(j) + 1);
  Höhe := hmax;
end
```

Ist w die Wurzel des Baumes, so ist $\text{Höhe}(w)$ die Höhe des Wurzelbaumes.

12. $\{2, 4, 5, 6\}, \{1\}, \{3\}, \{7\}, \{8\}$

13. Es sei T ein Tiefesuchebaum von G' mit Wurzel w und e, f Ecken aus T . Als Ecke w besucht wurde, war e noch nicht besucht worden. Somit ist $\text{TSE}[w] > \text{TSE}[e]$. D.h. der Aufruf von **tiefensuche(e)** war vor dem Aufruf von **tiefensuche(w)** beendet. Da e in G' von w erreichbar ist, ist w in G von e erreichbar. Wäre e vor w in G betrachtet worden, so wäre $\text{TSE}[e] > \text{TSE}[w]$. Somit wurde w vor e in G betrachtet. Aus $\text{TSE}[w] > \text{TSE}[e]$ folgt nun, daß der Aufruf von **tiefensuche(e)** in G innerhalb des Aufrufs **tiefensuche(w)** erfolgte. Somit ist e in G von w erreichbar, d.h. e und w liegen auf einem geschlossenen Weg in G . Das gleiche gilt für f und w und somit auch für e und f . Hieraus folgt, daß die Ecken von T in einer starken Zusammenhangskomponente von G liegen.

Sind umgekehrt e und f Ecken von G , welche in einer starken Zusammenhangskomponente liegen, so liegen e und f in G und G' auf einem geschlossenen Weg. Somit müssen auch e und f im gleichen Tiefesuchebaum von G' liegen. Hieraus folgt, daß die Ecken einer starken Zusammenhangskomponente von G in einem Tiefesuchebaum von G' liegen.

14. Es wird eine topologische Sortierung des DAG's betrachtet. Die Ecke mit Sortierungsnummer 1 hat Eingrad 0 und die mit Sortierungsnummer n hat Ausgrad 0. Eine Ecke mit Ausgrad 0 findet man, indem man an einer beliebigen Ecke einen einfachen Weg startet. Da der Graph keine geschlossenen Wege hat, endet der Weg an einer Ecke mit Ausgrad 0. Ein Algorithmus zur Bestimmung einer topologischen Sortierung sucht wiederholt eine Ecke mit Ausgrad 0, entfernt diese und alle inzidenten Kanten aus G und numeriert die entfernten Ecken in absteigender Reihenfolge. Da die Bestimmung einer Ecke mit Ausgrad 0 den Aufwand $O(n)$ hat, ergibt sich ein Gesamtaufwand von $O(n^2)$. Der auf der Tiefensuche aufbauende Algorithmus arbeitet mit Aufwand $O(n + m)$.
15. Zwei Ecken i und j sind genau dann in einer starken Zusammenhangskomponenten, wenn sie auf einem geschlossenen Weg liegen. Dies ist genau dann der Fall,

wenn e_{ji} und e_{ij} beide ungleich 0 sind, bzw. wenn $e_{ji}e_{ij}$ ungleich 0 ist. Für den i -ten Diagonaleintrag d_{ii} von E^2 gilt:

$$d_{ii} = e_{1i}e_{i1} + \dots + e_{ji}e_{ij} + \dots + e_{ni}e_{in}$$

Ist $d_{ii} = 0$, so bildet Ecke e eine komplett Zusammenhangskomponente. Ist $d_{ii} \neq 0$, so muß auch $e_{ii} = 1$ sein. Ferner gibt es noch $d_{ii} - 1$ weitere Ecken j mit $e_{ji}e_{ij} \neq 0$. Hieraus folgt sofort die Aussage.

```

16.  var Besucht : array[0..max] of Boolean;
      Vorgänger : array[0..max] of Integer;
procedure geschlossenerWeg(G : Graph);
var
  i : Integer;
begin
  Initialisiere Besucht mit false und Vorgänger mit 0;
  for jede Ecke i do
    if Besucht[i] = false then
      tiefensuche(i);
  end

  procedure tiefensuche(i : Integer);
  begin
    Besucht[i] := true;
    for jeden Nachbar j von i do
      if Besucht[j] = false then begin
        Vorgänger[j] := i;
        tiefensuche(j);
      end
      else if not Vorgänger[j] = i then
        exit('Geschlossener Weg');
    end
  end
end

```

Am Ende des Algorithmus (entweder regulär oder durch `exit`) hat der Algorithmus einen Baum durchlaufen, d.h. die Anzahl der betrachteten Kanten ist kleiner als n . Somit ist die Laufzeit $O(n)$.

- 17. $\{0, 2, 5, 7, 13\}, \{1, 3, 12, 9, 10, 8\}, \{4, 11, 6, 14\}$
- 18. Es werden zwei Tiefensuchedurchgänge gestartet mit den Startecken a bzw. b . Hierbei wird das gleiche Feld `TSNummer` benutzt (im zweiten Durchgang wird es nicht mehr initialisiert). Jede nicht besuchte Ecke ist weder von a noch von b erreichbar und muß somit entfernt werden. Bei der Verwendung von Adjazenzlisten basierend auf Zeigern müssen nur die entsprechenden Listen komplett entfernt werden. Somit ist der Aufwand $O(n + m)$.

```

19.  var TSNummer, MinNummer, Vorgänger : array[1..max] of Integer;
      zähler : Integer;
      TrennendeEcken : array[1..max] of Boolean;
procedure trennendeEcken(G : Graph);
var
  i : Integer;
begin
  Initialisiere TSNummer und zähler mit 0;
  Initialisiere TrennendeEcken mit false;

```

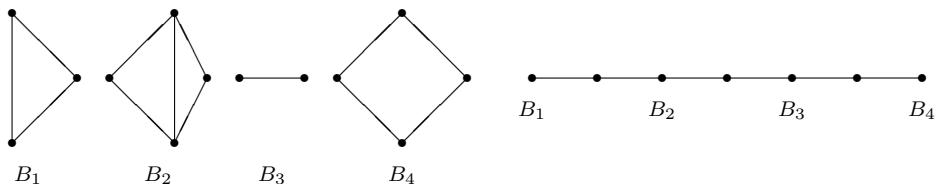
```

for jede Ecke i do
    if TSNummer[i] = 0 then begin
        blockproz(i);
        if i hat mehr als 2 Nachfolger then
            TrennendeEcken[i] := true;
        else
            TrennendeEcken[i] := false;
    end;
for jede Ecke i do
    if TrennendeEcken[i] = true then
        ausgabe(i, ist trennende Ecke);
end

procedure blockproz(i : Integer);
var
    j : Integer;
begin
    zähler := zähler + 1;
    TSNummer[i] := MinNummer[i] := zähler;
    for jeden Nachbar j von i do begin
        if TSNummer[j] = 0 then begin
            Vorgänger[j] := i;
            blockproz(j);
            if MinNummer[j] >= TSNummer[i] then
                TrennendeEcken[i] := true;
            else
                MinNummer[i] := min(MinNummer[i], MinNummer[j]);
        end
        else
            if j ≠ Vorgänger[i] then
                MinNummer[i] := min(MinNummer[i], TSNummer[j]);
    end
end

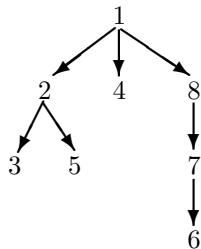
```

20. Links sind die Blöcke des Graphen und rechts ist der Blockgraph dargestellt.

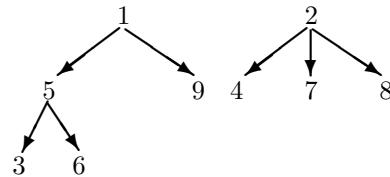


21. Der Beweis erfolgt durch vollständige Induktion nach b . Ist $b = 1$, so ist $t = 0$ und die Aussage ist wahr. Sei $b > 1$, B ein Block, welcher zu einem Blatt im Blockgraph gehört, und x die zugehörige Ecke. Entferne aus G alle Ecken in $B \setminus \{x\}$. Der neue Graph hat $b - 1$ Blöcke und mindestens $t - 1$ trennende Ecken. Die Aussage folgt nun aus der Induktionsvoraussetzung.
22. Ist der Graph zweifach zusammenhängend, so ist keine Ecke eine trennende Ecke. Andernfalls hat der Blockgraph mindestens zwei Blätter (jeder Baum mit mindestens zwei Ecken hat auch mindestens zwei Blätter). Ein Block, welcher zu einem Blatt im Blockgraph gehört, hat mindestens eine Ecke, welche nicht trennend ist.

23. a)



b)



24. Eine Kante liegt genau dann auf einem geschlossenen Weg, wenn sie zu einer starken Zusammenhangskomponente von G gehört. Mit dem in Abschnitt 4.5 beschriebenen Verfahren werden zunächst die starken Zusammenhangskomponenten bestimmt. Die gesuchte Kantenmenge besteht aus allen Kanten von G , welche in keiner starken Zusammenhangskomponente liegen.
25. Wendet man die Breitensuche n -mal auf den Graphen an, wobei jedesmal eine andere Startecke gewählt wird, so werden für jede Ecke die erreichbaren Ecken bestimmt. Der Aufwand ist $O(nm)$.
26. Es sei e eine beliebige Ecke und l_e die Länge des kürzesten Weges, der e enthält. Es wird eine Breitensuche mit Startecke e gestartet. Sei j die erste Ecke, welche die Breitensuche zum zweiten Mal besucht und i der aktuelle Vorgänger von j . Dann gilt

$$Niv(j) = Niv(i) \text{ oder } Niv(j) = Niv(i) + 1.$$

Im ersten Fall liegt ein geschlossener Weg der Länge $2Niv(i) + 1$ und im zweiten Fall der Länge $2Niv(i) + 2$ vor. Im ersten Fall gilt $l_e = 2Niv(i) + 1$ und im zweiten $l_e = 2Niv(i) + 2$ oder $l_e = 2Niv(i) + 1$. Um eine Unterscheidung zu treffen, müssen alle Ecken x mit $Niv(x) = Niv(i)$ abgearbeitet werden. Danach kann die Breitensuche beendet werden. Trifft man dabei auf eine weitere schon besuchte Ecke j mit $Niv(j) = Niv(i)$, so gilt $l_e = 2Niv(i) + 1$ und andernfalls $l_e = 2Niv(i) + 2$.

Ein Algorithmus zur Bestimmung der Länge eines kürzesten Weges startet von jeder Ecke e eine Breitensuche. Diese wird wie beschrieben benutzt, um die Länge des kürzesten geschlossenen Weges W mit $e \in W$ zu bestimmen. So erhält man die Länge des kürzesten geschlossenen Weges des Graphen mit Aufwand $O(nm)$.

27. Zunächst wird in linearer Zeit eine topologische Sortierung des Graphen bestimmt. Danach werden die Ecken in umgekehrter Reihenfolge ihrer topologischen Sortierungsnummern bearbeitet und die Menge der erreichbaren Ecken wie folgt bestimmt:

$$\text{Erreichbar}(i) := \bigcup_{j \text{ Nachfolger von } i} \text{Erreichbar}(j) \cup \{j\}$$

28. Wird die Tiefensuche auf einen Wurzelbaum angewandt, so sind Tiefensuche- und Breitensuchebaum identisch mit dem Graphen. D.h. für einen Wurzelbaum der Höhe h haben sowohl Tiefensuche- als auch Breitensuchebaum die Höhe h .

29. Zur Bestimmung der Zusammenhangskomponenten eines ungerichteten Graphen mit Hilfe der Breitensuche kann die Prozedur `zusammenhangskomponenten` aus Abbildung 4.16 verwendet werden. Dabei wird der Aufruf `zusammenhang(i)` durch den Aufruf `breitensuche(G, i)` ersetzt. Die Prozedur `breitensuche` aus Abbildung 4.29 muß dabei nur geringfügig geändert werden. An den beiden Stellen, an denen das Feld `niveau` geändert wird, muß die entsprechende Komponente von `ZKNummer` auf `zähler` gesetzt werden. D.h. die beiden Anweisungen `ZKNummer[startecke] = zähler` und `ZKNummer[j] = zähler` müssen eingefügt werden.
30.

```
var niveau : array[1..max] of Integer;
procedure breitensuche(G : Graph; var B : Graph;
                      startecke : Integer);
var
  i, j : Integer;
  W : warteschlange of Integer;
begin
  Initialisiere niveau mit -1;
  niveau[startecke] := 0;
  W.einfügen(startecke);
  while W ≠ ∅ do begin
    i := W.entfernen;
    for jeden Nachbar j von i do
      if niveau[j] = -1 then begin
        niveau[j] := niveau[i] + 1;
        W.einfügen(j);
        Füge Kante (i,j) in B ein;
      end
      else if niveau[j] = niveau[i] + 1 then
        Füge Kante (i,j) in B ein;
    end
  end
end
```
31. a) Der Beweis erfolgt durch vollständige Induktion nach n . Für $n = 1$ ist die Aussage wahr. Sei nun $n > 1$ und x eine Ecke von B mit $g^+(x) = 0$. Setze $G' = G \setminus \{x\}$ und $B' = B \setminus \{x\}$. Da B' die gleiche Eigenschaften wie B hat, folgt aus der Induktionsvoraussetzung, daß B' ein Tiefensuchebaum von G' ist. Es sei nun y der Vorgänger von x in B und (x, z) mit $z \neq y$ eine Kante in G . Nach Voraussetzung ist z Vorgänger von x in B , d.h. wenn die Tiefensuche bei Ecke y ankommt, sind schon alle Nachbarn von x besucht worden. Somit wird nach y die Ecke x besucht und sofort wieder zu y zurückgekehrt. Also ist B ein Tiefensuchebaum von G .
- b) Für eine beliebige Ecke x von G bezeichne $d_B(e, x)$ die Länge des Weges von e nach x in B . Mittels vollständiger Induktion nach $d(e, x)$ wird gezeigt, daß $d_B(e, x) = d(e, x)$ für alle Ecken x von G gilt. Ist $d(e, x) = 0$, so ist $x = e$ und die Aussage ist wahr. Sei nun $d(e, x) > 0$ und y der Vorgänger von x in G auf dem Weg von e nach x . Dann ist $d(e, y) < d(e, x)$ und somit ist $d_B(e, y) = d(e, y)$ nach Induktionsvoraussetzung. Also gilt: $d_B(e, x) \geq d(e, x) = d(e, y) + 1 = d_B(e, y) + 1$. Da (x, y) eine Kante in G ist, unterscheiden sich $d_B(e, x)$ und $d_B(e, y)$ maximal um 1. Hieraus folgt $d_B(e, x) = d(e, x)$.

Man beachte, daß B nicht notwendigerweise durch die in Abschnitt 4.10 beschriebene Realisierung der Breitensuche erzeugt werden kann.

32. Da B keine geschlossenen Wege enthält, können die Verfahren etwas vereinfacht werden. Wird die Tiefensuche wie in Abschnitt 4.2 beschrieben mit Hilfe eines Stacks realisiert, so ist der Speicheraufwand proportional zur maximalen Stapeltiefe. Da im ungünstigsten Fall bis zu den Blättern gesucht werden muß, ist der Speicheraufwand $O(C)$. Bei der iterativen Tiefensuche wird keine Ecke jenseits von Tiefe T betrachtet. Somit ist der Speicheraufwand $O(T)$. Der Speicheraufwand der Breitensuche ist proportional zur Länge der Warteschlange. Im ungünstigsten Fall wird die gesuchte Ecke als letzte Ecke auf Tiefe T betrachtet. Da in diesem Fall alle Ecken der Tiefe T in der Warteschlange sind, ist der Speicheraufwand $O(b^T)$.

Die Laufzeit der drei Suchverfahren ist proportional zur Anzahl der betrachteten Ecken. Für die Tiefensuche sind dies $(b^{C+1} - 1)/(b - 1)$ und für die Breitensuche $(b^{T+1} - 1)/(b - 1)$ Ecken. Bei der iterativen Tiefensuche werden etwa $b^{T+2}/(b - 1)^2$ Ecken betrachtet (vergleichen Sie Abschnitt 4.11).

33. Ist (e, f) eine Rückwärtskante eines ungerichteten Graphen, so ist entweder e ein Vorgänger oder ein Nachfolger von f im Tiefensuchewald, d.h. Rückwärtskanten verbinden niemals Blätter des Tiefensuchebaums. Da auch Vorwärtskanten keine Blätter verbinden, ist die Aussage bewiesen.
34. Der Algorithmus ist eine Variante der Tiefensuche angewendet auf die Ausgangs-
ecke des inversen Graph des Schaltkreises (d.h. die Richtung jeder Kante des
Graphen wird umgedreht).

```

function schaltkreis(e : Ecke) : Boolean;
begin
  switch(e.type) begin
    case Eingabeecke:
      schaltkreis := Variable_e;
    case Konjunktion:
      schaltkreis := schaltkreis(Nachbar_1(e)) AND
                    schaltkreis(Nachbar_2(e));
    case Disjunktion:
      schaltkreis := schaltkreis(Nachbar_1(e)) OR
                    schaltkreis(Nachbar_2(e));
    case Negation:
      schaltkreis := NOT schaltkreis(Nachbar(e));
    case Ausgabeecke:
      schaltkreis := schaltkreis(Nachbar(e));
  end
end

```

B.5 Kapitel 5

1. Die chromatische Zahl der vier Graphen von links nach rechts und oben nach unten ist 2, 3, 4 und 3.
2. Es gilt $\chi(L_{z,s}) = 2$ (man färbe die Ecken wie ein Schachbrett) und $\omega(L_{z,s}) = 2$.
3. Man unterteile die Ecken von H_d in zwei Teilmengen, je nachdem ob die Anzahl der Einsen gerade oder ungerade ist. Da die beiden Mengen unabhängige Mengen sind, ist $\chi(H_d) = 2$. Da H_d keine Dreiecke enthält, ist $\omega(H_d) = 2$.
4. Es sei B ein beliebiger Tiefensuchebaum von G der Höhe h . Es gilt $h \leq m$. Die Ecken von G werden derart gefärbt, so daß die Ecken auf einem Niveau jeweils die gleiche Farbe bekommen. Dazu werden $h+1$ Farben verwendet. Da nach den Ergebnissen aus Abschnitt 4.7 eine Kante niemals Ecken aus dem gleichen Niveau verbindet, liegt eine zulässige Färbung vor. Somit gilt $\chi(G) \leq m+1$.
5. Wendet man die Breitensuche auf einen Baum an, so wird jede Kante zu einer Baumkante, d.h. für jede Kante $k = (e, f)$ gilt: $Niv(e) + Niv(f) \equiv 1(2)$. Die Aussage folgt nun aus dem letzten Lemma in Abschnitt 4.10.
6. Man betrachte den in Abschnitt 4.9 definierten Blockgraph G_B und erzeuge für einen beliebigen Block B eine minimale Färbung. Danach wird G_B anhand der Tiefensuche mit Startecke B durchlaufen. Da G_B ein Baum ist, ist in jedem neuen Block genau eine Ecke schon gefärbt. Diese wird jeweils zu einer minimalen Färbung erweitert. Hieraus folgt die Behauptung. Färbungsalgorithmen können diese Eigenschaft nutzen und zunächst in linearer Zeit die Blöcke bestimmen und danach wie beschrieben weiter verfahren. Der Vorteil liegt darin, daß gegebenenfalls kleinere Graphen gefärbt werden müssen.
7. Sei U' ein induzierter Untergraph von G , so daß $\delta(U') \geq \delta(U)$ für alle induzierten Untergraphen U von G ist. Zum Beweis wird eine Reihenfolge der Ecken bestimmt, für die der Greedy-Algorithmus maximal $\delta(U') + 1$ Farben vergibt. Es sei e_n eine Ecke von minimalem Eckengrad in G . Sind e_n, \dots, e_{i+1} schon bestimmt, so wähle man aus $G \setminus \{e_n, \dots, e_{i+1}\}$ eine Ecke e_i mit minimalem Eckengrad. Für $i = 1, \dots, n$ sei G_i der von $\{e_1, \dots, e_i\}$ induzierte Untergraph. Dann gilt $\delta(G_i) = d_{G_i}(e_i)$. Wird der Greedy-Algorithmus auf G angewandt, wobei die Ecken in der Reihenfolge e_1, e_2, \dots betrachtet werden, so hat die jeweils aktuelle Ecke e_i schon $g_{G_i}(e_i)$ gefärbte Nachbarn. Insgesamt vergibt der Greedy-Algorithmus somit maximal $1 + \max\{g_{G_i}(e_i) \mid i = 1, \dots, n\}$ Farben. Sei j minimal, so daß $U' \subseteq G_j$ gilt. Da $U' \not\subseteq G_{j-1}$ ist, liegt e_j in U' . Nun gilt:

$$\begin{aligned} \delta(U') &\leq g_{U'}(e_j) \\ &\leq g_{G_j}(e_j) \\ &\leq \max\{g_{G_i}(e_i) \mid i = 1, \dots, n\} \\ &\leq \max\{\delta(G_i) \mid i = 1, \dots, n\} \\ &\leq \delta(U') \end{aligned}$$

Hieraus folgt die Aussage.

8. Es sei A die Menge der Ecken von G , deren Grad mindestens $\chi(G) - 1$ ist. Angenommen es gilt $|A| \leq \chi(G) - 1$. Dann färbe man die Ecken von G mit dem Greedy-Algorithmus, wobei zuerst die Ecken aus A betrachtet werden. Hierfür werden maximal $\chi(G) - 1$ Farben verwendet. Da die restlichen Ecken maximal den Eckengrad $\chi(G) - 2$ haben, vergibt der Greedy-Algorithmus insgesamt höchstens $\chi(G) - 1$ Farben. Dieser Widerspruch beweist $|A| \geq \chi(G)$.
9. Es sei e eine beliebige Ecke von G . Dann ist nach Voraussetzung $N(e)$ eine unabhängige Menge von G . Also gilt $\alpha(G) \geq \Delta(G)$. Somit folgt

$$(\alpha(G) + 1)^2 > \alpha(G)(\alpha(G) + 1) \geq \alpha(G)(\Delta(G) + 1) \geq \alpha(G)\chi(G) \geq n.$$

Dies beweist die Behauptung.

10. a) Für jede Ecke e von G gilt $\chi(G \setminus \{e\}) \geq \chi(G) - 1$. Da nach Voraussetzung $\chi(G \setminus \{e\}) < \chi(G)$ für jede Ecke gilt, muß $\chi(G \setminus \{e\}) = \chi(G) - 1$ sein.
- b) Ist G ein 2-kritischer Graph, so hat $G \setminus \{e\}$ keine Kanten. Daraus folgt $G = C_2$. Ein 3-kritischer Graph G ist nicht bipartit und enthält somit einen geschlossenen Weg W ungerader Länge. Alle Ecken von G müssen auf W liegen. Gäbe es Kanten, welche nicht auf W liegen, so würde es eine Ecke e geben, so daß $G \setminus \{e\}$ immer noch einen geschlossenen Weg ungerader Länge enthält. Da $\chi(G \setminus \{e\}) = 2$ ist, kann dies nicht sein. Somit ist G vom Typ C_{2i+1} mit $i \geq 1$.
- c) Es sei G ein kritischer Graph und B_i die Blöcke von G . Nach Aufgabe 6 gilt:

$$\chi(G) = \max \{\chi(B_i) \mid i = 1, \dots, s\}.$$

Angenommen es ist $s > 1$. Ohne Einschränkung kann man annehmen, daß $\chi(G) = \chi(B_1)$ ist. Dann gibt es eine Ecke e in $B_2 \setminus B_1$. Somit ist $\chi(G \setminus \{e\}) = \chi(B_1) = \chi(G)$. Dieser Widerspruch zeigt, daß $s = 1$ gilt, d.h. G ist zweifach zusammenhängend.

11. Da es zwischen den Ecken einer unabhängigen Menge keine Kanten gibt, gilt

$$m \leq n(n-1)/2 - \alpha(G)(\alpha(G)-1)/2$$

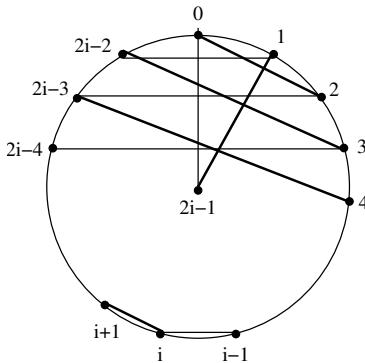
bzw. $2m \leq n^2 - \alpha(G)^2$. Hieraus folgt die Behauptung. Gilt $\alpha(G) = \sqrt{n^2 - 2m}$, so muß $\alpha(G) = n$ sein, d.h. G ist der leere Graph.

12. Für eine minimale Färbung muß es zu jedem Paar von Farben eine Kante geben, welche Ecken mit dieser Farbe verbindet. Somit gilt: $\chi(G)(\chi(G)-1)/2 \leq m$ bzw. $(\chi(G)-1/2)^2 \leq 2m + 1/4$. Hieraus folgt die Behauptung.
13. Die Intervalle, welche zu einer Farbklasse einer Färbung des Schnittgraphen gehören, überlappen sich paarweise nicht. Somit kann ein Arbeiter diese Arbeitsaufträge ausführen. Umgekehrt induziert eine Zuordnung von Arbeitern zu den Aufträgen eine Färbung von G . Somit ist $\chi(G)$ die minimale Anzahl von Arbeitern zur Ausführung der Aufträge. Auf der Menge der Intervalle wird eine

partielle Ordnung definiert: $T_i < T_j$, falls $b_i < a_j$. Ohne Einschränkung der Allgemeinheit kann man annehmen, daß die Intervalle in der Reihenfolge T_1, T_2, \dots gefärbt werden. Es seien $\{T_1, \dots, T_s\}$ die mit Farbe 1 gefärbten Intervalle und G' der von den restlichen Intervallen induzierte Graph. Im folgenden wird gezeigt, daß $\omega(G') < \omega(G)$ gilt. Daraus kann gefolgert werden, daß der Algorithmus maximal $\omega(G)$ Farben vergibt. Wegen $\omega(G) \leq \chi(G)$ bestimmt der Algorithmus somit eine minimale Färbung. Angenommen es gilt $\omega(G') = \omega(G)$. Dann gibt es in G' eine Clique C mit $\omega(G)$ Intervallen. Es sei I_C der Schnitt aller Intervalle aus C . Es ist $I_C \neq \emptyset$. Angenommen es gilt $T_i \cap I_C = \emptyset$ für alle $i = 1, \dots, s$. Sei j maximal mit $T_j < I_C$ (man beachte, daß $I_C < T_1$ nicht gelten kann). Somit gibt es in C ein Intervall I mit $T_j < I$ und es ist $I_C < T_{j+1}$. Dies steht im Widerspruch zur Wahl von T_j . Also gibt es ein Intervall $I \in \{T_1, \dots, T_s\}$ mit $I \cap I_C \neq \emptyset$. Da C eine maximale Clique ist, muß $I \in C$ sein. Somit liegt C nicht in G' . Dieser Widerspruch zeigt $\omega(G') < \omega(G)$.

14. $\chi(G) = 4$ und $\omega(G) = 2$ (Vergleichen Sie Aufgabe 27 für einen Beweis).
15. Die kantenchromatische Zahl der vier Graphen von links nach rechts und oben nach unten ist 3, 3, 3 und 4.
16. Da alle zu einer Ecke inzidenten Kanten verschiedene Farben haben müssen, gilt $\chi'(G) \geq \Delta(G)$.
17. Es sei Δ der maximale Eckengrad des bipartiten Graphen G . Die Aussage wird durch vollständige Induktion nach m bewiesen. Für $m = 1$ ist die Aussage wahr. Sei nun $m > 1$ und $k = (i, j)$ eine beliebige Kante von G . Nach Induktionsvoraussetzung können die Kanten des Graphen $G' = G \setminus \{k\}$ mit $\Delta(G')$ Farben gefärbt werden. Ist $\Delta(G') < \Delta$ oder gibt es eine Farbe, mit der keine der zu i und j inzidenten Kanten gefärbt ist, so ist die Aussage bewiesen. Andernfalls erreicht man durch Vertauschung von Farben, daß auch die Kante k mit einer schon verwendeten Farbe gefärbt werden kann. Zu jeder verwendeten Farbe f gibt es eine mit f gefärbte Kante, welche zu i oder j inzident ist. Da jedoch i und j maximal zu jeweils $\Delta - 1$ Kanten inzident sind, muß es Farben f_1 und f_2 geben, so daß eine Kante der Farbe f_1 zu i , aber keine Kante dieser Farbe zu j inzident ist. Ferner muß eine Kante der Farbe f_2 zu j , aber keine Kante dieser Farbe zu i inzident sein. Es sei $G(i, j)$ der von den mit f_1 und f_2 gefärbten Kanten gebildete Untergraph von G . Da die Färbung zulässig ist, haben die Ecken in $G(i, j)$ den Grad 1 oder 2, wobei i und j den Grad 1 haben. Die Zusammenhangskomponenten von $G(i, j)$ sind offene oder geschlossene Wege, deren Kanten abwechselnd mit f_1 und f_2 gefärbt sind. Es sei W ein Pfad, welcher mit der Ecke i anfängt. Da G bipartit ist, haben alle geschlossenen Wege gerade Länge. Somit hat W ungerade Länge (zusammen mit k entsteht ein geschlossener Weg). Also hat die letzte Kante von W die Farbe f_1 , d.h. j liegt nicht auf W . Vertauscht man die Farben f_1 und f_2 der Kanten in W , so entsteht wieder eine zulässige Färbung. Nun kann aber die Kante k mit der Farbe f_1 gefärbt werden.
18. Das Problem kann als Kantenfärbungsproblem formuliert werden. Jede Mannschaft entspricht einer Ecke in einem vollständigen Graphen. Die Anzahl der Far-

ben in einer minimalen Kantenfärbung entspricht der Anzahl der Tage des Wettbewerbes. Die Kanten mit gleicher Farbe entsprechen den Partien eines Spieltages. Um die minimale Dauer des Wettbewerbes zu bestimmen, muß $\chi'(K_n)$ bestimmt werden.



Zunächst wird gezeigt, daß $\chi'(K_{2i}) = 2i - 1$ ist. Dazu werden die Ecken von K_{2i} mit den Zahlen $\{0, \dots, 2i - 1\}$ nummeriert und eine graphische Darstellung von K_{2i} betrachtet, bei der die Ecken $\{0, \dots, 2i - 2\}$ gleichmäßig auf einem Kreis angeordnet sind und die Ecke $2i - 1$ im Mittelpunkt liegt. Folgende Kanten können mit Farbe 1 gefärbt werden: $(2i-1, 0), (1, 2i-2), (2, 2i-3), \dots, (i-1, i)$. In der graphischen Darstellung sieht man, daß alle Kanten bis auf die erste parallel sind und somit keine gemeinsame Ecke haben. Eine neue Farbklass hat man, indem man zu jeder Ecke jeder Kante, ausgenommen der ersten Ecke der ersten Kante, 1 modulo $2i - 1$ addiert. Die neue Farbklass besteht aus den Kanten

$$(2i-1, 1), (2, 0), (3, 2i-2), \dots, (i, i+1)$$

und ist durch fette Kanten dargestellt. Man erhält diese Kanten, indem man die Kanten der ersten Farbklass um den Mittelpunkt um den Winkel $2\pi/(2i-1)$ dreht. Auf diese Art erhält man $2i-1$ Farbklassen mit je i Kanten. Da alle Kanten gefärbt sind und alle Farbklassen die maximale Anzahl von Kanten enthalten, ist $\chi'(K_{2i}) = 2i - 1$.

Es bleibt noch $\chi'(K_{2i+1})$ zu bestimmen. Eine Farbklass in K_{2i+1} kann maximal i Kanten enthalten. Somit ist $\chi'(K_{2i+1}) \geq 2i + 1$. Da K_{2i+1} ein Untergraph von K_{2i+2} ist, induziert eine Kantenfärbung von K_{2i+2} mit $2i + 1$ Farben eine Kantenfärbung mit $2i + 1$ Farben auf K_{2i+1} . Somit ist $\chi'(K_{2i+1}) = 2i + 1$.

19. Für $n \geq 3$ gilt $\chi(G_n) = 4$. Der Aufruf `faerbung` $(G_n, 3)$ liefert **false** zurück. Jede der Ecken $1, \dots, n-1$ hat im Suchbaum mindestens den Eckengrad 2. Somit hat der Suchbaum $O(2^n)$ Ecken.
20. Die Änderungen an der Prozedur `faerbung` verbessern die absolute Laufzeit, so daß kleine Graphen in akzeptabler Zeit gefärbt werden können. Die Laufzeit wächst jedoch weiterhin exponentiell.
21. Auf dem ersten Niveau hat der Suchbaum $n-1$ Ecken und jede dieser Ecken hat wieder $n-1$ Nachfolger. Jeweils eine dieser Ecken hat keinen Nachfolger mehr. Somit sind auf dem dritten Niveau $(n-1)^2(n-2)$ Ecken. Dabei haben jeweils zwei Nachfolger einer Ecke auf Niveau drei keine Nachfolger etc. Somit sind auf dem vierten Niveau $(n-1)^2(n-2)(n-3)$ Ecken. Wiederholt man dieses Argument, so erhält man für die Gesamtanzahl der Ecken im Suchbaum:

$$(n-1) \sum_{s=0}^{n-1} \prod_{j=1}^s (n-j) = (n-1)(n-1)! \sum_{i=1}^n \frac{1}{(n-i)!}$$

Da die letzte Summe gegen e konvergiert, ist die Anzahl der Ecken im Suchbaum annährend $e(n-1)(n-1)!$.

22. Der Algorithmus verwaltet zwei Variablen **unten** und **oben**. Zu jedem Zeitpunkt gibt es eine Färbung mit **oben** Farben, aber keine mit **unten** Farben. Aus diesem Grund wird vorausgesetzt, daß der Graph nicht leer ist. In jedem Schritt wird das Intervall, in dem die chromatische Zahl liegt, halbiert.

```

function chromatischeZahl(Graph: G) : Integer;
var unten, oben, mitte : Integer;
begin
    unten := 1; oben := n;
    while unten < (oben - 1) do begin
        mitte := (oben - unten)/2;
        if färbung(G,mitte) then
            oben := mitte
        else
            unten := mitte;
    end;
    chromatischeZahl = oben;
end

```

23. a) Es sei I eine maximale unabhängige Menge von G . Ist $e \notin I$, so ist I eine maximale unabhängige Menge von $G \setminus \{e\}$. Ist $e \in I$, so ist $I \setminus \{e\}$ eine unabhängige Menge von $G \setminus (\{e\} \cup N(e))$. Wäre $I \setminus \{e\}$ keine maximale unabhängige Menge von $G \setminus (\{e\} \cup N(e))$, dann gäbe es eine unabhängige Menge I' von $G \setminus (\{e\} \cup N(e))$ mit $|I'| > |I| - 1$. Somit wäre auch $I' \cup \{e\}$ eine unabhängige Menge mit mehr als $|I| + 1$ Ecken. Hieraus folgt die Behauptung.
b) Es genügt, eine maximale unabhängige Menge für jede Zusammenhangskomponente zu bestimmen. Ist der maximale Eckengrad von G kleiner oder gleich 2, so sind die Zusammenhangskomponenten von G isolierte Ecken, Pfade oder geschlossene Wege. In jedem dieser Fälle läßt sich eine maximale unabhängige Menge in linearer Zeit bestimmen.
c) Die Korrektheit folgt direkt aus Teil a).
d) Bezeichne mit $s(n)$ die Anzahl der Schritte, die die Funktion **unabhängig** für Graphen mit n Ecken benötigt. Dann gilt für $n > 4$ folgende Beziehung:

$$s(n) \leq s(n-1) + s(n-4) + Cn \quad (C > 0)$$

Schätzt man $s(n)$ durch λa^n mit $a > 1$ ab und wählt a , so daß

$$a^n \approx a^{n-1} + a^{n-4} + o(1)$$

erfüllt ist, dann ist folgende Ungleichung zu lösen:

$$a^4 \geq a^3 + 1 + \epsilon$$

Der Wert $a \approx 1.39$ erfüllt diese Ungleichung. Somit ist $O(1.39^n)$ der Aufwand der Funktion **unabhängig**.

24. Zunächst wird die erste Ungleichung betrachtet. Es seien f_1 bzw. f_2 minimale Färbungen von G bzw. \overline{G} . Ordne jeder Ecke e aus G das Paar $(f_1(e), f_2(e))$ zu. Angenommen es gibt zwei Ecken $a \neq b$, welche das gleiche Paar zugeordnet bekommen. Dann ist $f_1(a) = f_1(b)$ und $f_2(a) = f_2(b)$. Somit können a und b weder in G noch in \overline{G} benachbart sein. Dieser Widerspruch beweist, daß $n \leq \chi(G)\chi(\overline{G})$ gilt. Hieraus folgt

$$4n \leq 4\chi(G)\chi(\overline{G}) \leq (\chi(G) + \chi(\overline{G}))^2$$

und somit $2\sqrt{n} \leq \chi(G)\chi(\overline{G})$. Für C_4 gilt $\chi(C_4) + \chi(\overline{C_4}) = 4 = 2\sqrt{4}$ und für $K_{s,s}$ mit $s > 2$ gilt $\chi(K_{s,s}) + \chi(\overline{K_{s,s}}) = 2 + s > 2\sqrt{2s} = 2\sqrt{n}$.

Der Beweis der zweiten Ungleichung wird durch vollständige Induktion nach n geführt. Für $n = 2$ ist die Aussage wahr. Sei nun $n > 2$, e eine Ecke von G und $G_e = G \setminus \{e\}$. Dann gilt $\chi(G_e) + 1 \geq \chi(G)$ und $\chi(\overline{G_e}) + 1 \geq \chi(\overline{G})$. Nach Induktionsvoraussetzung ist $\chi(G_e) + \chi(\overline{G_e}) \leq n$. Gilt sogar $\chi(G_e) + \chi(\overline{G_e}) < n$, so folgt die Aussage direkt. Somit bleibt noch der Fall $\chi(G_e) + \chi(\overline{G_e}) = n$. Ist $g(e) < \chi(G_e)$, so folgt $\chi(G) = \chi(G_e)$ und hieraus ergibt sich $\chi(G) + \chi(\overline{G}) \leq n+1$. Ist $g(e) \geq \chi(G_e)$, so gilt

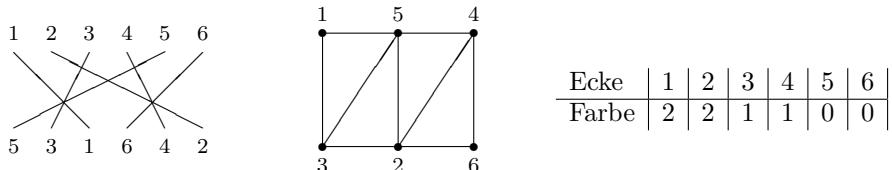
$$\bar{g}(e) = n - 1 - g(e) \leq n - 1 - \chi(G_e) = n - 1 - (n - \chi(\overline{G_e})) = \chi(\overline{G_e}) - 1$$

und somit $\bar{g}(e) < \chi(\overline{G_e})$. Also ist $\chi(\overline{G}) = \chi(\overline{G_e})$, woraus die Aussage folgt.

Für die Graphen $K_{s,s}$ gilt $\chi(K_{s,s}) + \chi(\overline{K_{s,s}}) = 2 + s < 2s + 1 = n + 1$ und für die Graphen K_n gilt $\chi(K_n) + \chi(\overline{K_n}) = n + 1$.

25. Es gilt $\chi(R_n) = 1 + \chi(C_n)$ für alle $n \geq 3$. Somit ist $\chi(R_{2n}) = 3$ und $\chi(R_{2n+1}) = 4$.
26. Da I_5 vollständig ist, gilt $\chi(I_5) = 5$. Für $n \geq 6$ gilt nach dem Satz von Brooks $\chi(I_n) \leq 4$. Da I_n einen geschlossenen Weg der Länge drei enthält, ist $\chi(I_n) \geq 3$. Ist $n \equiv 0(3)$, so können die Ecken im Uhrzeigersinn mit den Farben 1, 2, 3, 1, 2, 3, ... gefärbt werden. Somit ist $\chi(I_{3n}) = 3$ für $n \geq 2$. Ist $n \not\equiv 0(3)$, so legen die Farben von zwei auf C_n benachbarten Ecken eine Färbung fest. In diesem Fall kommt man nicht mit drei Farben aus, d.h. die chromatische Zahl ist 4.
27. Der Beweis wird durch vollständige Induktion nach n geführt. Für $n = 3$ ist die Aussage wahr. Sei nun $n > 3$. Es ist zu zeigen, daß G_n keinen geschlossen Weg der Länge drei enthält. Dazu beachte man zunächst, daß der von den neuen Ecken induzierte Untergraph diese Eigenschaft hat. Nach Induktionsvoraussetzung und Konstruktion bilden die Nachbarn der Ecken von G_{n-1} in G_n jeweils eine unabhängige Menge. Somit ist $\omega(G_n) = 2$. Eine minimale Färbung von G_{n-1} kann zu einer Färbung von G_n erweitert werden, indem die Ecke e' die gleiche Farbe wie e und die zusätzliche Ecke f eine neue Farbe bekommt. Somit gilt $\chi(G_n) \leq n$ nach Induktionsvoraussetzung. Angenommen es gibt eine Färbung von G_n , welche nur $n - 1$ Farben verwendet. Da Ecke f zu jeder neuen Ecke benachbart ist, ist keine dieser Ecken mit der gleichen Farbe gefärbt. Nun kann für den Untergraphen G_{n-1} von G_n eine Färbung mit $n - 2$ Farben erzeugt werden. Dazu bekommt jede Ecke e die Farbe von e' . Dies widerspricht der Induktionsvoraussetzung und somit ist $\chi(G_n) = n$.

28. Für die Graphen I_n gilt: $\chi'(I_{2n+1}) = 5$ und $\chi'(I_{2n}) = 4$. Für die Graphen R_n gilt: $\chi'(R_n) = n$.
29. Es werden maximal zwei weitere Farben benötigt. Man wende die Breitensuche auf B an und färbe die noch nicht gefärbten Ecken in ungeraden Niveaus mit der Farbe 2 und die in geraden Niveaus mit der Farbe 3.
30. Es wird ein Graph gebildet, in dem jede Radiostation einer Ecke entspricht und zwei Ecken verbunden sind, wenn die Entfernung der zugehörigen Stationen unter dem vorgegebenen Wert liegt. Radiostationen, welche in einer festen minimalen Färbung dieses Graphen die gleiche Farbe haben, können die gleiche Sendefrequenz benutzen.
31. Nein. Die Prozedur stützt sich bei der Färbung einer Ecke e mit $g(e) < 5$ wesentlich auf die Planarität des Graphen. Der Graph K_6 erfüllt die angegebene Bedingung, aber es gilt $\chi(K_6) > 5$.
32. $G^\pi = G_{\pi^{-1}}$.
33. Enthält jede Clique von G weniger als \sqrt{n} Ecken, so folgt aus dem letzten Lemma aus Abschnitt 5.5, daß $\chi(G) < \sqrt{n}$ ist. Somit gilt $n/\chi(G) > \sqrt{n}$. Aus dem gleichen Lemma folgt, daß G eine unabhängige Menge der Größe \sqrt{n} hat. Somit hat \bar{G} eine Clique der Größe \sqrt{n} .
34. Orientiert man jede Kante des Permutationsgraphen in Richtung der höheren Eckennummern, so entsteht eine kreisfreie transitive Orientierung. Die chromatische Zahl des Permutationsgraphen ist $h + 1$, wobei h die maximale Höhe einer Ecke ist. Aus Aufgabe 42 folgt nun, daß der Greedy-Algorithmus für Permutationsgraphen eine minimale Färbung erzeugt.
35. Definiere eine Permutation ρ durch $\rho(i) = \pi(n + 1 - i)$ für $i = 1, \dots, n$. Es gilt $\rho^{-1}(i) = n + 1 - \pi^{-1}(i)$. Ferner ist $\pi^{-1}(i) < \pi^{-1}(j)$ genau dann, wenn $\rho^{-1}(i) > \rho^{-1}(j)$ ist. Somit ist G_ρ das Komplement von G_π .
36. Im folgenden ist das Permutationsdiagramm, der Graph und die minimale Färbung dargestellt.



37. Eine Ecke j ist zu $\pi(1)$ benachbart, wenn π^{-1} die Reihenfolge von j und $\pi(1)$ vertauscht. Für $j \neq \pi(1)$ gilt $\pi^{-1}(j) > \pi^{-1}(\pi(1)) = 1$. Somit sind die Ecken $j = 1, \dots, \pi(1)-1$ zu $\pi(1)$ benachbart und es gilt $g(\pi(1)) = \pi(1)-1$. Eine Ecke j ist zu $\pi(n)$ benachbart, wenn π^{-1} die Reihenfolge von j und $\pi(n)$ vertauscht. Für $j \neq \pi(n)$ gilt $\pi^{-1}(j) < \pi^{-1}(\pi(n)) = n$. Somit sind die Ecken $j = n, \dots, \pi(1)+1$ zu $\pi(n)$ benachbart und es gilt $g(\pi(n)) = n - \pi(n)$.

38. Es sei $I = \{I_1, \dots, I_n\}$ eine Menge von Intervallen der Form $I_i = [a_i, b_i]$ und G der zugehörige Intervallgraph. Auf der Menge I kann eine partielle Ordnung eingeführt werden: $I_i < I_j$ genau dann, wenn $I_i \cap I_j = \emptyset$ und $b_i < a_j$ gilt. Das Komplement \overline{G} hat die Kantenmenge $K = \{(I_i, I_j) \mid I_i < I_j \text{ oder } I_i > I_j\}$. Wird jede Kante in K in Richtung des größeren Intervall ausgerichtet, so erhält man eine transitive Orientierung von \overline{G} .
39. Es sei G ein ungerichteter Graph, so daß G und \overline{G} planar sind. Bezeichne die Anzahl der Kanten von G mit m und die von \overline{G} mit \bar{m} . Dann gilt nach den Ergebnissen aus Abschnitt 5.4: $n(n-1)/2 = m + \bar{m} \leq 2(3n-6)$. Hieraus folgt $n(n-1)/(n-2) \leq 12$. Diese Ungleichung ist für $n > 10$ nicht erfüllt.
40. Der Graph ist 4-kritisch.
41. Es sei s die Anzahl der Farbklassen in einer nicht trivialen Färbung, welche genau eine Ecke enthalten. Diese bilden eine Clique. Somit ist $s \leq \chi(G)$. Da die restlichen Farbklassen mindestens zwei Farben enthalten, gilt $\chi_n(G) \leq s + (n-s)/2 \leq (n+\chi(G))/2$ bzw. $2(n-\chi_n(G)) \geq (n-\chi(G))$. Hieraus folgt die Behauptung. Der Greedy-Algorithmus erzeugt eine nicht triviale Färbung.
42. Es genügt folgende Aussage zu beweisen: Für jede Ecke i gilt: $h(i) + f(i) \leq h+1$. Hierbei bezeichnet $f(i)$ die vom Greedy-Algorithmus vergebene Farbnummer und $h(i)$ die Höhe der Ecke i . Der Beweis erfolgt durch vollständige Induktion nach der Farbnummer. Für Ecken i mit $f(i) = 1$ ist die Aussage klar. Sei nun i eine Ecke mit $f(i) > 1$. Da i nicht mit der Farbnummer $f(i)-1$ gefärbt wurde, muß es einen Nachbarn j von i mit $j < i$ und $f(j) = f(i)-1$ geben. Dann folgt $h(j) \geq h(i)+1$. Nach Induktionsvoraussetzung gilt nun:

$$h+1 \geq h(j) + f(j) \geq h(i) + 1 + f(i) - 1 = h(i) + f(i)$$

43. Man beachte zunächst, daß Färbungen der Graphen G^+ und G^- Färbungen auf G induzieren, welche genau soviel Farben verwenden. Somit ist $\chi(G^+) \geq \chi(G)$ und $\chi(G^-) \geq \chi(G)$. Es sei f eine minimale Färbung von G . Ist $f(a) = f(b)$, so induziert f eine Färbung auf G^+ . Somit gilt $\chi(G) \geq \chi(G^+)$ bzw. $\chi(G) = \chi(G^+)$. Ist hingegen $f(a) \neq f(b)$, so induziert f eine Färbung auf G^- . In diesem Fall gilt $\chi(G) \geq \chi(G^-)$ bzw. $\chi(G) = \chi(G^-)$. Hieraus folgt $\chi(G) = \min(\chi(G^+), \chi(G^-))$.

Die Gleichung führt direkt zu einem rekursiven Algorithmus zur Bestimmung einer minimalen Färbung. Ist G leer oder vollständig, so kann eine minimale Färbung direkt angegeben werden. Andernfalls werden zwei nicht benachbarte Ecken a, b ausgewählt und rekursiv minimale Färbungen für G^+ und G^- bestimmt. Die Färbung, welche die wenigsten Farben verwendet, wird zu einer Färbung auf G erweitert. Der durch dieses Verfahren aufgebauten Lösungsbaum heißt *Zykov Baum*.

B.6 Kapitel 6

1. Es sei f ein maximaler Fluß auf G und f' ein maximaler Fluß auf G' . Da f' ein zulässiger Fluß für G ist, gilt: $|f| \geq |f'|$. Es sei (X, \bar{X}) ein q-s-Schnitt von G'

mit $|f'| = \kappa_{G'}(X, \bar{X})$. Da keine Kante aus H in dem Schnitt (X, \bar{X}) liegt, gilt $|f'| = \kappa_G(X, \bar{X})$. Analog zu dem Beweis des Lemmas aus Abschnitt 6.1 folgt nun:

$$|f| = f(X, \bar{X}) - f(\bar{X}, X) \leq \kappa_G(X, \bar{X}) = |f'|$$

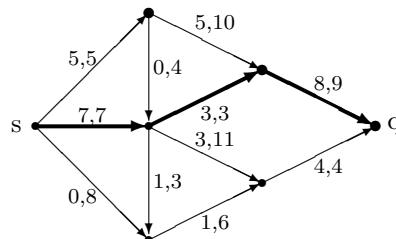
Man beachte hierzu die in Abschnitt 6.1 angegebene Definition von $|f|$. Hieraus folgt $|f'| = |f|$.

2. Es sei X die Menge der vier Ecken auf der linken Seite des Netzwerkes. Dann gilt $\kappa(X, \bar{X}) = 1 + \lambda + \lambda^2 = 2$. Somit ist der Fluß, welcher die drei mittleren waagrechten Kanten sättigt, maximal und hat den Wert 2. Wählt man die Erweiterungswege W_i wie in dem Hinweis angegeben, so ist für $i > 0$ jeweils eine der drei mittleren Kanten eine Rückwärtskante und die anderen beiden sind Vorwärtskanten. Die Wahl der Rückwärtskante ändert sich dabei zyklisch: jeweils die darüberliegende Kante ist im nächsten Erweiterungsweg die Rückwärtskante. Unter Ausnutzung der im Hinweis angegebenen Beziehung $\lambda^{i+2} = \lambda^i - \lambda^{i+1}$ folgt, daß die entstehenden Flüsse $|f_i|$ folgende Werte haben:

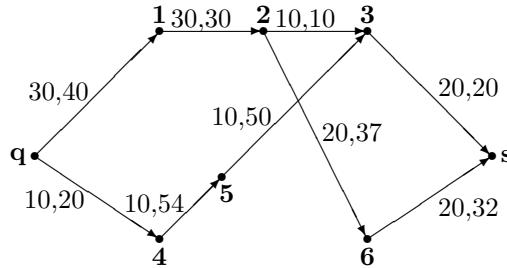
$$|f_i| = 1 + \sum_{j=2}^{i+1} \lambda^j = -\lambda + \sum_{j=0}^{i+1} \lambda^j < -\lambda + \lambda + 2 = 2$$

Ferner gilt $\lim_{i \rightarrow \infty} |f_i| = 2$. Fügt man noch eine zusätzliche Kante von q nach s mit Kapazität 1 ein, so ist der maximale Fluß dieses Netzwerkes gleich 3. Somit konvergiert die Folge der Flüsse f_i noch nicht einmal gegen den maximalen Fluß.

3. a) Der Wert von f ist 11.
- b) Die fett gezeichneten drei Kanten bilden einen Erweiterungsweg für f mit $f_\Delta = 1$. Der erhöhte Fluß hat den Wert 12 und ist unten dargestellt.
- c) Es sei X die Menge der drei fett gezeichneten Ecken. Dann ist $\kappa(\bar{X}, X) = 12$ und somit ist $\kappa(X, X)$ ein Schnitt mit minimaler Kapazität und der erhöhte Fluß ist maximal.
- d) Der unten dargestellte Fluß ist maximal und hat den Wert 12.

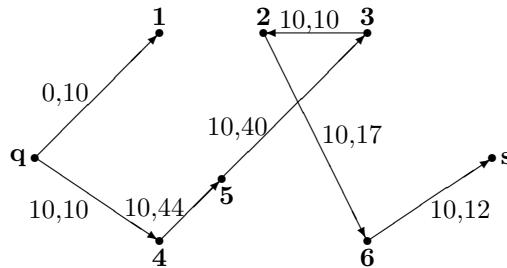


4. Das zu dem trivialen Fluß f_0 gehörende geschichtete Hilfsnetzwerk G'_{f_0} sieht wie folgt aus. Die Bewertungen der Kanten geben den blockierenden Fluß f_1 und die Kapazitäten an.



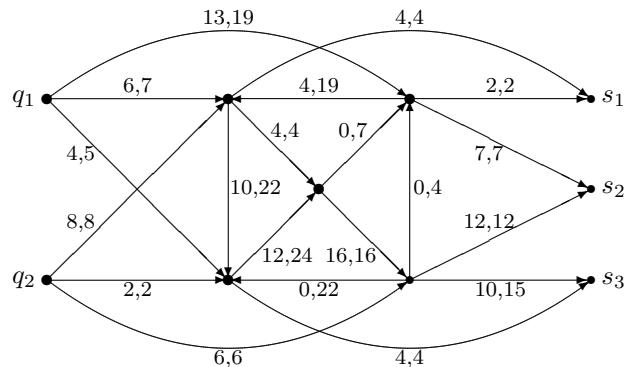
Ecke	q	1	2	3	4	5	6	s
Durchsatz	60	30	30	20	20	50	32	52

Bei der Bestimmung von f_1 hat zunächst Ecke 3 und danach Ecke 1 minimalen Durchsatz. Der blockierende Fluß f_1 hat den Wert 40. Das zu f_1 gehörende ge schichtete Hilfsnetzwerk C'_{f_1} sieht wie folgt aus.



Hieraus ergibt sich ein blockierender Fluß f_2 mit dem Wert 10 und somit ein maximaler Fluß f mit dem Wert 50. Im Gegensatz zum ursprünglichen Netzwerk wird die Prozedur **blockfluss** nur zweimal aufgerufen.

5. Die zusätzlichen Kanten müssen eine unbeschränkte Kapazität haben. Der unten dargestellte Fluß hat den Wert 39. Die fett dargestellten Ecken bilden einen Schnitt mit minimaler Kapazität.



6. Ist $X_1 \subseteq X_2$ oder $X_2 \subseteq X_1$, so ist die Aussage wahr. Im folgenden wird der Fall betrachtet, daß $X_1 \not\subseteq X_2 \not\subseteq X_1$ gilt. Dann sind folgende Mengen nicht leer und paarweise disjunkt:

$$A = X_1 \cap X_2, B = \overline{X_1} \cap X_2, C = X_1 \cap \overline{X_2}, D = \overline{X_1} \cap \overline{X_2}.$$

Ferner gilt $q \in A, s \in D, \overline{X_1 \cap X_2} = B \cup C \cup D, D = \overline{X_1 \cup X_2}$ und $X_1 \cup X_2 = A \cup B \cup C$. Da $(X_1, \overline{X_1})$ und $(X_2, \overline{X_2})$ minimale Schnitte sind, gilt:

$$\begin{aligned}\kappa(X_1, \overline{X_1}) &\leq \kappa(A, B \cup C \cup D) \\ \kappa(X_1, \overline{X_1}) &\leq \kappa(A \cup B \cup C, D) \\ \kappa(X_2, \overline{X_2}) &\leq \kappa(A, B \cup C \cup D) \\ \kappa(X_2, \overline{X_2}) &\leq \kappa(A \cup B \cup C, D)\end{aligned}$$

Wegen $X_1 = A \cup C$ und $\overline{X_1} = B \cup D$ folgt aus der ersten Ungleichung

$$\kappa(A, B) + \kappa(A, D) + \kappa(C, B) + \kappa(C, D) \leq \kappa(A, B) + \kappa(A, C) + \kappa(A, D)$$

bzw.

$$\kappa(C, B) + \kappa(C, D) \leq \kappa(A, C).$$

Aus den anderen drei Ungleichungen ergibt sich analog:

$$\begin{aligned}\kappa(A, B) + \kappa(C, B) &\leq \kappa(B, D) \\ \kappa(B, C) + \kappa(B, D) &\leq \kappa(A, B) \\ \kappa(A, C) + \kappa(B, C) &\leq \kappa(C, D)\end{aligned}$$

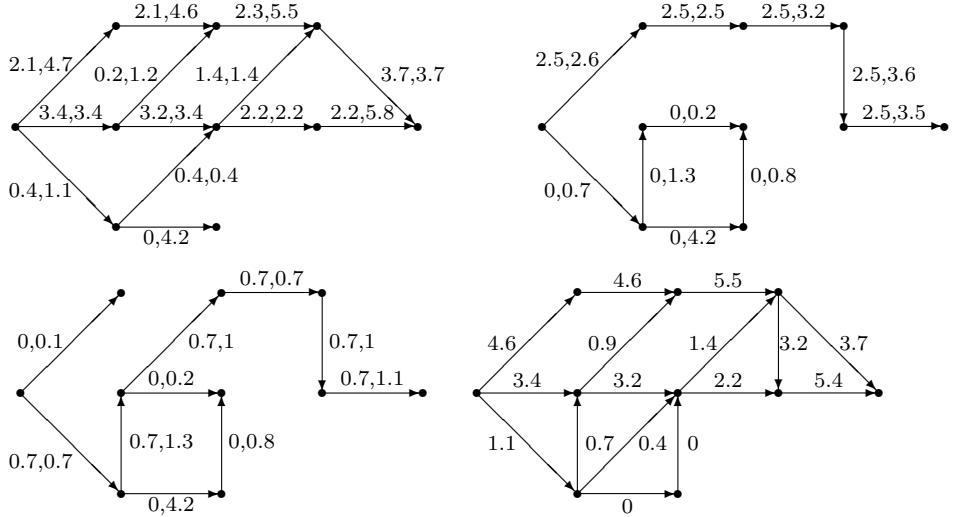
Addiert man die letzten vier Ungleichungen, so folgt $\kappa(C, B) = \kappa(B, C) = 0$, da alle Kapazitäten nicht negativ sind. Setzt man dies in die letzten vier Ungleichungen ein, so ergibt sich $\kappa(A, B) = \kappa(B, D)$ und $\kappa(A, C) = \kappa(C, D)$. Schließlich gilt:

$$\begin{aligned}\kappa(X_1, \overline{X_1}) &= \kappa(A, B) + \kappa(A, D) + \kappa(C, D) \\ &= \kappa(A, B) + \kappa(A, D) + \kappa(A, C) \\ &= \kappa(A, B \cup D \cup C) \\ &= \kappa(X_1 \cap X_2, \overline{X_1 \cap X_2})\end{aligned}$$

und

$$\begin{aligned}\kappa(X_1, \overline{X_1}) &= \kappa(A, B) + \kappa(A, D) + \kappa(C, D) \\ &= \kappa(B, D) + \kappa(A, D) + \kappa(C, D) \\ &= \kappa(B \cup A \cup C, D) \\ &= \kappa(X_1 \cup X_2, \overline{X_1 \cup X_2})\end{aligned}$$

7. Enthält ein Netzwerk einen geschlossenen Weg W , so daß $|f| < \kappa(k)$ für alle Kanten k von W gilt, so gibt es einen maximalen Fluß f' und eine Kante k auf W mit $f'(k) > |f'| = |f|$.
 8. Im folgenden sind die drei konstruierten geschichteten Hilfsnetzwerke und die dazugehörigen blockierenden Flüsse dargestellt. Die letzte Abbildung zeigt den maximalen Fluß mit Wert 9.1.



9. Die Prozedur `blockfFluss` findet unabhängig vom Wert von b einen blockierenden Fluss mit Wert 4. Dieser verwendet die beiden oberen Kanten. Ist $b = 0$, so ist der blockierende Fluss maximal. Ein maximaler Fluss hat den Wert $\min(4 + b, 10)$.
 10. Es sei f ein maximaler Fluss und W ein Weg von q nach s mit $f(k) > 0$ für alle Kanten k von W . Ferner sei k_0 eine Kante von W mit $f(k_0) \leq f(k)$ für alle Kanten k von W . Man entferne k_0 und setze $f'(k) = f(k) - f(k_0)$ für alle k auf W und ansonsten $f'(k) = f(k)$. Dieses Verfahren kann maximal m mal wiederholt werden. Die so entstehenden Wege bilden eine Folge von Erweiterungswegen, die zu einem maximalen Fluss führen. Leider ergibt sich hieraus kein neuer Algorithmus zur Bestimmung eines maximalen Flusses, da ein solcher Fluss schon für die Bestimmung der Wege benötigt wird.
 11. Der Algorithmus konstruiere die Flüsse f_0, f_1, \dots, f_k , wobei f_0 der triviale und f_k ein maximaler Fluss ist. Setze $\Delta_i = |f_{i+1}| - |f_i|$. Nach Aufgabe 11 kann mit maximal m Erweiterungswegen ein maximaler Fluss gefunden werden (mit f_i startend). Somit gilt $\Delta_i \geq (|f_{max}| - |f_i|)/m$. Hieraus folgt:

$$|f_{max}| - |f_i| \leq m\Delta_i = m(|f_{i+1}| - |f_i|) = m(|f_{i+1}| - |f_{max}|) + m(|f_{max}| - |f_i|)$$

Somit ergibt sich $|f_{max}| - |f_{i+1}| \leq (|f_{max}| - |f_i|)(1 - 1/m)$. Da f_0 der triviale Fluß ist, kann mittels vollständiger Induktion gezeigt werden, daß folgende Ungleichung

gilt:

$$|f_{max}| - |f_i| \leq |f_{max}|(1 - 1/m)^i$$

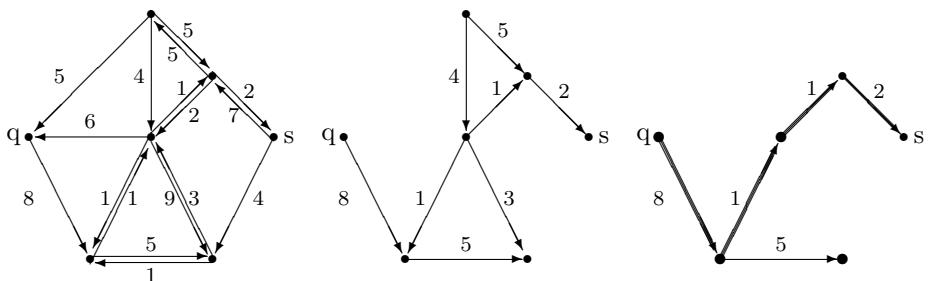
Da alle Kapazitäten ganzzahlig sind, folgt:

$$1 \leq |f_{max}| - |f_{k-1}| \leq |f_{max}|(1 - 1/m)^{k-1}$$

Unter Verwendung der Ungleichung $m(\log m - \log(m-1)) \geq 1$ zeigt man nun $k-1 \leq m \log |f_{max}|$. Hieraus folgt die Aussage.

12. Es sei f ein maximaler Fluß des Netzwerkes.

- a) Wird die Kapazität jeder Kante um C erhöht, so erhöht sich auch der Wert eines maximalen Flusses um mindestens C . Dazu wird der Wert von f auf den Kanten eines Weges W von q nach s jeweils um C erhöht. Gibt es noch andere Wege W' von q nach s , welche mit W keine gemeinsamen Kanten haben, so kann der Wert von f noch weiter erhöht werden.
 - b) Wird die Kapazität jeder Kante um den Faktor C erhöht, so setze man $f'(k) = Cf(k)$ für jede Kante k des Netzwerkes. Dann ist f' ein maximaler Fluß in dem neuen Netzwerk und es gilt $|f'| = C|f|$.
13. Die Kapazitäten der Kanten seien z_i/n_i für $i = 1, \dots, m$, wobei z_i und n_i positive ganze Zahlen sind. Ferner sei V das kleinste gemeinsame Vielfache der Nenner n_1, \dots, n_m . Dann ist $f_\Delta \geq 1/V$ für jeden Erweiterungsweg jedes Flusses f . Ist M der Wert eines maximalen Flusses, so endet der in Abschnitt 6.1 beschriebene Algorithmus spätestens nach MV Schritten.
14. Der Wert eines Schnittes ist in beiden Netzwerken gleich. Die Behauptung folgt aus dem Satz von Ford und Fulkerson.
15. Im folgenden ist zunächst der Graph G_f dargestellt. Daneben der Graph, welcher von den Vorrwärtskanten von G_f induziert wird. Da es in diesem Graph keinen Weg von q nach s gibt, ist f ein blockierender Fluß. Ganz rechts ist der Graph G'_f dargestellt. Die fett dargestellten Kanten bilden einen blockierenden Fluß h mit Wert 1. Der neue Fluß $f + h$ hat den Wert 12 und ist maximal. Dazu betrachte man die in G'_f fett gezeichnete Menge X von Ecken und zeige, daß $\kappa(X, \bar{X}) = 12$ ist.



16. Da f_1 und f_2 Flüsse auf G sind, gilt $f(k) \geq 0$ für jede Kante k von G und die Flußberhaltungsbedingung ist für f erfüllt. Somit ist f genau dann ein Fluß von G , wenn $f_1(k) + f_2(k) \leq \kappa(k)$ für jede Kante k von G gilt.
17. a) Wird die Kapazität einer Kante um 1 erhöht, so erhöht sich der maximale Fluß höchstens um 1. Da die Kapazitäten ganzzahlig sind, findet der Algorithmus von Edmonds und Karp in einem Schritt (mit f startend) einen maximalen Fluß. Somit ist der Aufwand $O(n + m)$.
- b) Wird die Kapazität einer Kante um 1 erniedrigt, so erniedrigt sich der maximale Fluß höchstens um 1. Ist $f(k) < \kappa(k)$, so folgt aus der Ganzzahligkeit der Kapazitäten, daß f auch in dem neuen Netzwerk ein maximaler Fluß ist. Ist $f(k) = \kappa(k)$, so findet man mit Aufwand $O(n + m)$ einen Weg W von q nach s , welcher die Kante k enthält und der Fluß durch jede Kante von W positiv ist. Bilde einen neuen Fluß f' durch $f'(k) = f(k) - 1$ für alle Kanten k auf W und $f'(k) = f(k)$ ansonsten. Dann ist f' ein zulässiger Fluß auf dem neuen Netzwerk mit $|f'| = |f| - 1$. Der Algorithmus von Edmonds und Karp findet in einem Schritt (mit f' startend) einen maximalen Fluß. Somit ist der Aufwand $O(n + m)$.
18. Die Prozeduren **erweiterevorwärts** und **und erweitererückwärts** können nicht direkt verwendet werden, sondern müssen noch angepaßt werden. Das folgende Programm ist eine Variante der Prozedur **erweitererückwärts**. Man beachte, daß am Ende der **while**-Anweisung die Flußberhaltungsbedingung für die entfernte Ecke i erfüllt ist (nur für $i \neq e$). Die Prozedur **erweiterevorwärts** muß entsprechend abgeändert werden.

```

var durchfluß : array[1..max] of Real;
W : warteschlange of Integer;
i,j : Integer;
m : Real;
begin
  Initialisiere durchfluss mit 0;
  durchfluß[e] := f_e;
  W.einfügen(e);
repeat
  i := W.entfernen;
  while durchfluß[i] ≠ 0 do begin
    sei k=(i,j) eine Kante mit f(k) > 0;
    m := min(f(k), durchfluß[i]);
    f(k) := f(k) - m;
    if j ≠ s then
      W.einfügen(j);
    durchfluß[j] := durchfluß[j] + m;
    durchfluß[i] := durchfluß[i] - m;
  end;
  until W = ∅;
end

```

19. a) Wegen $\alpha \in [0, 1]$ gilt für alle Kanten k :

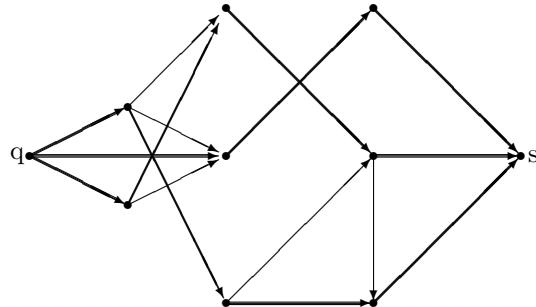
$$0 \leq f_\alpha(k) = \alpha f_1(k) + (1 - \alpha) f_2(k) \leq \alpha \kappa(k) + (1 - \alpha) \kappa(k) = \kappa(k)$$

b) Für jede Ecke $e \neq q, s$ von G gilt:

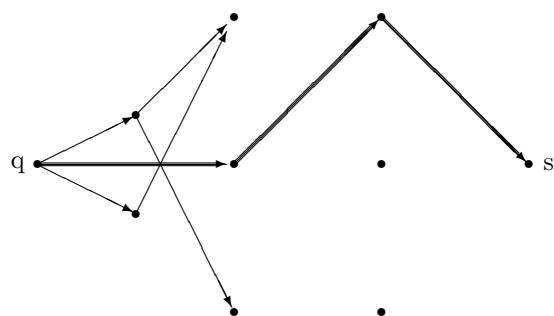
$$\begin{aligned}
 \sum_{k=(j,e) \in K} f_\alpha(k) &= \sum_{k=(j,e) \in K} \alpha f_1(k) + (1 - \alpha) f_2(k) \\
 &= \alpha \sum_{k=(j,e) \in K} f_1(k) + (1 - \alpha) \sum_{k=(j,e) \in K} f_2(k) \\
 &= \alpha \sum_{k=(e,j) \in K} f_1(k) + (1 - \alpha) \sum_{k=(e,j) \in K} f_2(k) \\
 &= \sum_{k=(e,j) \in K} \alpha f_1(k) + (1 - \alpha) f_2(k) \\
 &= \sum_{k=(e,j) \in K} f_\alpha(k)
 \end{aligned}$$

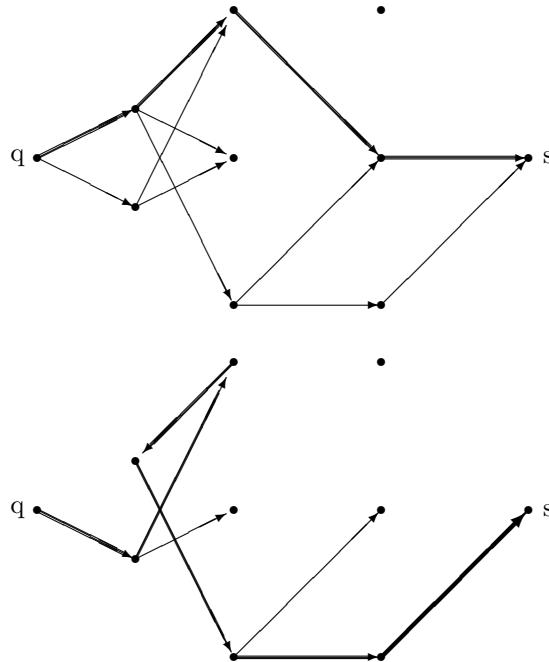
c) $|f_\alpha| = \alpha|f_1| + (1 - \alpha)|f_2|$.

20. Es sei k eine Kante eines minimalen Schnittes von G mit positiver Kapazität. Dann ist die Kapazität eines minimalen Schnittes von $G \setminus \{k\}$ echt kleiner als die eines minimalen Schnittes von G . Die Aussage folgt nun aus dem Satz von Ford und Fulkerson.
21. Die fett gezeichneten Kanten bilden einen maximalen Fluß.



Hierzu werden die im folgenden dargestellten drei blockierenden Flüsse gebildet. Dargestellt sind jeweils die geschichteten Hilfsnetzwerke und die blockierenden Flüsse (fette Kanten).





22. Der Beweis der Aussagen erfolgt analog zu den Beweisen in Abschnitt 6.5. Der Beweis für Aussage a) ergibt sich aus dem Beweis des Lemmas aus diesem Abschnitt und der für Aussage b) aus dem Beweis des nachfolgenden Satzes.

B.7 Kapitel 7

1. In Abschnitt 7.1 wurde gezeigt, daß es eine eindeutige Beziehung zwischen den Zuordnungen eines bipartiten Graphen G und den binären Flüssen des zugehörigen 0-1-Netzwerkes N_G gibt. Es sei Z eine Zuordnung eines bipartiten Graphen G . Ein Weg W , welcher zwei nicht zugeordnete Ecken verbindet, heißt Erweiterungsweg für Z , falls jede zweite Kante von W in Z liegt. Die Erweiterungswägen für Z in G entsprechen eindeutig den Erweiterungswägen bezüglich f_Z in N_G . Der Begriff des Erweiterungsweges läßt sich auch auf beliebige ungerichtete Graphen übertragen. Es gilt folgender Satz.

Satz. Eine Zuordnung Z in einem ungerichteten Graphen G ist genau dann maximal, wenn es in G für Z keinen Erweiterungsweg gibt.

Beweis. Es sei Z eine Zuordnung und W ein Erweiterungsweg für Z . Ferner sei Z' die symmetrische Differenz von Z und W (Z' besteht aus den Kanten, die entweder nur in Z oder nur in W liegen, aber nicht in beiden Mengen). Dann ist Z' eine Zuordnung für G und es gilt $|Z'| = |Z| + 1$. Somit kann es für eine maximale Zuordnung keinen Erweiterungsweg geben.

Es sei nun Z eine Zuordnung, welche nicht maximal ist, und Z' eine maximale Zuordnung. Bezeichne mit G' den von den Kanten aus $Z \cup Z'$ induzierten Untergraphen von G . Für jede Ecke e von G' ist $g(e) \leq 2$. Ist $g(e) = 2$, so ist e zu genau einer Kante aus Z und einer Kante aus Z' inzident. Somit können die Zusammenhangskomponenten von G' in zwei Gruppen aufgeteilt werden: geschlossene und einfache Wege. Auf diesen Wegen liegen abwechselnd Kanten aus Z und Z' . Da geschlossene Wege jeweils gleichviel Kanten aus Z und Z' enthalten und es in Z' mehr Kanten als in Z gibt, muß es in G' einen Weg geben, welcher zwei bezüglich Z nicht zugeordnete Ecken verbindet. Dies ist ein Erweiterungsweg für Z in G .

■

2. Mit einem Greedy-Verfahren wird eine nicht erweiterbare Zuordnung Z bestimmt (Aufwand $O(n + m)$). Ist Z noch keine vollständige Zuordnung, so ist $|Z| < 2n$. Dann gibt es Ecken e und f von G , welche zu keiner Kante aus Z inzident sind. Da Z nicht erweiterbar ist, gibt es für jeden Nachbarn j von e oder f genau eine zu j inzidente Kante aus Z . Wegen $|Z| < 2n$ und $g(e) + g(f) \geq 2n$ gibt es eine Kante $k = (j, j') \in Z$, so daß (e, j) und (j', f) Kanten in G sind. Somit ist auch

$$Z' = Z \setminus \{k\} \cup \{(e, j), (j', f)\}$$

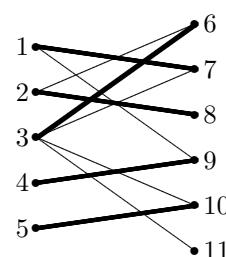
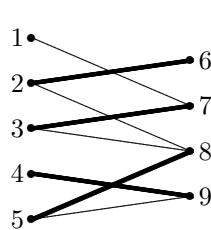
eine Zuordnung von G . Dieses Verfahren wiederholt man bis eine vollständige Zuordnung vorliegt. Der Aufwand des folgenden Algorithmus ist $O(n + m)$.

```

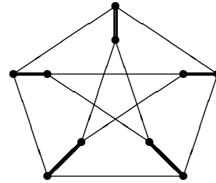
var L : array[1..max] of Integer;
Bestimme mit einem Greedy-Verfahren nicht erweiterbare Zuordnung Z;
Initialisiere L mit 0;
for alle nicht zugeordneten Ecken x,y do begin
    for jeden Nachbar j von x do begin
        sei k = (j,j') ∈ Z;
        L[j'] = j;
    end
    suche Nachbarn j von y mit L[j] ≠ 0;
    setze Z := Z \ {(j,L[j])} ∪ {(x,L[j]), (j,y)};
    for jeden Nachbar j von x do begin
        sei k = (j,j') ∈ Z;
        L[j'] = 0;
    end
end

```

3. Die fett gezeichneten Kanten bilden jeweils eine maximale Zuordnung.



4. Die fett gezeichneten Kanten bilden eine vollständige Zuordnung.



5. Wegen $\alpha(G) = 2$ besteht jede Farbklasse einer Färbung von G aus maximal zwei Ecken. Es sei a die Anzahl der Farbklassen mit zwei Ecken einer minimalen Färbung von G . Dann ist $\chi(G) = a + (n - 2a) = n - a$. Die Farbklassen mit zwei Ecken induzieren eine Zuordnung auf \overline{G} mit a Kanten. Somit gilt $a \leq z$ und $\chi(G) \geq n - z$. Jede maximale Zuordnung auf \overline{G} induziert eine Färbung mit $n - z$ Farben auf G . Somit gilt $\chi(G) \leq n - z$.
6. Es sei $T \subseteq E_1$. Ferner sei K_1 die Menge der Kanten von G , welche zu einer Ecke aus T inzident sind, und K_2 die Menge der Kanten von G , welche zu einer Ecke aus $N(T)$ inzident sind. Es gilt $K_1 \subseteq K_2$. Aus der Voraussetzung folgt $|T|k \leq |K_1|$ und $|N(T)|k \geq |K_2|$. Somit gilt $|N(T)| \geq |T|$ für alle Teilmengen T von E_1 . Die Aussage folgt nun aus dem Satz von Hall.
7. Es sei U_1 bzw. U die Menge der Ecken, welche zu keiner Kante aus Z_1 bzw. Z inzident sind. Dann sind U_1 und U unabhängige Mengen mit $|U_1| = n - 2|Z_1|$ und $|U| = n - 2|Z|$. Eine maximale unabhängige Menge von G hat maximal $|Z| + |U|$ Ecken. Somit gilt:

$$n - 2|Z_1| = |U_1| \leq \alpha(G) \leq |Z| + |U| = n - |Z|$$

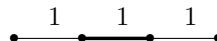
Hieraus folgt $|Z| \leq 2|Z_1|$.

8. Nach dem Satz von Hall hat G genau dann keine Zuordnung Z mit $|Z| = |E_1|$, wenn es eine Teilmenge D von E_1 mit $|D| > |N(D)|$ gibt. Zunächst wird ein maximaler binärer Fluß f auf dem zu G gehörenden 0-1-Netzwerk N_G bestimmt. Gilt $|f| = |E_1|$, so existiert keine Teilmenge D von E_1 mit $|D| > |N(D)|$. Gilt $|f| < |E_1|$, so wird die Menge X aller Ecken aus N_G bestimmt, welche von q über Erweiterungswege bezüglich f erreichbar sind. Es sei $D = X \cap E_1$. Wie im Beweis des Satzes von Hall in Abschnitt 7.1 zeigt man $|f| = |E_1 \setminus X| + |N(D)|$. Wegen $|E_1| = |E_1 \setminus X| + |D|$ folgt nun $|D| > |N(D)|$. Dieser Algorithmus hat eine Laufzeit von $O(\sqrt{nm})$.
9. Es sei G ein kantenbewerteter, vollständig bipartiter Graph und e eine beliebige Ecke von G . Es sei G' eine Kopie von G , in der die Bewertungen aller zu e inzidenten Kanten um b geändert sind. Dann ist jede vollständige Zuordnung von G auch eine vollständige Zuordnung von G' (und umgekehrt). Die Kosten der beiden Bewertungen unterscheiden sich um b . Es sei $B = b_1 + \dots + b_{|E_1|}$, wobei $b_i = \min\{\text{kosten}(k) \mid k \text{ zu } e_i \in E_1 \text{ inzidente Kante}\}$ ist. Ist die maximale Zuordnung für G_0 auch für G vollständig, so hat man eine vollständige Zuordnung für G mit minimalen Kosten B . Andernfalls bestimme man, wie in Aufgabe 8 beschrieben,

eine Teilmenge X von E_1 mit $|N_{G_0}(X)| < |X|$ und verändere den Graphen G_0 . Man beachte, daß G_0 mindestens eine zusätzliche Kante besitzt. Ist die maximale Zuordnung für G_0 auch für G vollständig, so hat man eine vollständige Zuordnung für G mit minimalen Kosten $B + (|X| - |N_{G_0}(X)|)b_{min}$. Andernfalls wird der letzte Schritt wiederholt. Da in jedem Schritt G_0 mindestens eine zusätzliche Kante bekommt, sind maximal $|E_1|(|E_1| - 1) < n^2$ Durchgänge notwendig.

10. Es sei G ein kantenbewerteter bipartiter Graph und T die Summe der Bewertungen aller Kanten von G . Durch Hinzufügen neuer Ecken und Kanten mit Bewertung T erhält man einen Graph G' mit den in Aufgabe 9 angegebenen Eigenschaften. Jede Zuordnung von G kann zu einer Zuordnung von G' ergänzt werden. Die Wahl von T bedingt, daß aus einer Zuordnung mit minimalen Kosten für G eine vollständige Zuordnung mit minimalen Kosten für G' entsteht. Es sei Z' eine vollständige Zuordnung von G' mit minimalen Kosten. Entfernt man alle Kanten mit Bewertung T aus Z' , so erhält man eine Zuordnung Z von G . Dies ist eine maximale Zuordnung mit minimalen Kosten. Der in Aufgabe 9 angegebene Algorithmus kann also zur Bestimmung maximaler Zuordnungen mit minimalen Kosten in beliebigen kantenbewerteten bipartiten Graphen verwendet werden.

Eine nicht erweiterbare Zuordnung mit minimalen Kosten ist nicht notwendigerweise eine maximale Zuordnung wie der folgende Graph zeigt. Die fett gezeichnete Kante bildet eine nicht erweiterbare Zuordnung mit minimalen Kosten. Diese Zuordnung ist aber nicht maximal.



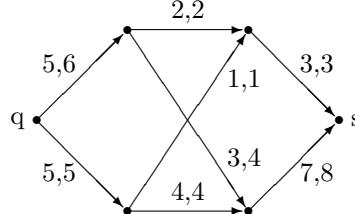
Der Zusammenhang zwischen maximalen Zuordnungen mit minimalen Kosten in kantenbewerteten, vollständig bipartiten Graphen vom Typ $G_{n,n}$ und maximalen Zuordnungen in beliebigen kantenbewerteten bipartiten Graphen ist nicht auf nicht erweiterbare Zuordnungen übertragbar. Somit kann der angegebene Algorithmus nicht zur Bestimmung nicht erweiterbarer Zuordnungen mit minimalen Kosten verwendet werden.

11. Es sei G' das angegebene neue Netzwerk. Jedem q - s -Fluß f auf G entspricht ein s - q -Fluß f' auf G' und umgekehrt. Dazu definiert man

$$f((a, b)) = -f'((b, a))$$

für alle Kanten (a, b) . Man beweist direkt, daß f' genau dann zulässig für G' ist, wenn f zulässig für G ist. Ferner ist f genau dann minimal zulässig, wenn f' maximal zulässig ist. Somit genügt es einen maximalen Fluß für das Netzwerk G' zu konstruieren. Hierzu konstruiert man zunächst wie in Abschnitt 7.2 angegeben einen zulässigen Fluß für G , dieser induziert einen zulässigen Fluß auf G' . Mit Hilfe von Erweiterungswegen kann dieser zu einem zulässigen maximalen Fluß für G' erweitert werden. Dieser entspricht dann einen minimalen zulässigen Fluß auf G .

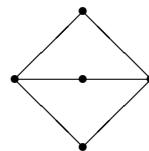
12. Der Fluß mit Wert 2 auf der Kante $(1, 2)$ und dem Wert 1 auf allen anderen Kanten hat unter den zulässigen Flüssen des Netzwerkes aus Abbildung 7.6 den kleinsten Wert.
13. Die Bewertungen der Kanten geben den Wert eines minimalen und eines maximalen Flusses an. Die Werte der Füsse sind 10 und 11.



14. Genau dann gibt es einen zulässigen Fluß auf G , wenn jede Kante von G auf einem Weg von q nach s liegt.
15. In Aufgabe 11 wurde aus einem Netzwerk G mit oberen und unteren Kapazitätsgrenzen ein neues Netzwerk G' konstruiert. Der Wert eines minimalen zulässigen q - s -Flusses f_{MIN} auf G ist gleich dem negativen Wert eines maximal zulässigen s - q -Flusses f_{MAX} auf G' . Ist (X, \bar{X}) ein q - s -Schnitt von G , so ist (\bar{X}, X) ein s - q -Schnitt von G' und es gilt $\kappa_G(X, \bar{X}) = -\kappa_{G'}(\bar{X}, X)$. Somit gilt:

$$\begin{aligned} |f_{MIN}| &= -|f_{MAX}| \\ &= -\min\{\kappa_{G'}(X, \bar{X}) \mid (X, \bar{X}) \text{ } s\text{-}q\text{-Schnitt von } G'\} \\ &= -\min\{-\kappa_G(\bar{X}, X) \mid (\bar{X}, X) \text{ } q\text{-}s\text{-Schnitt von } G\} \\ &= -\min\{\kappa_G(X, \bar{X}) \mid (\bar{X}, X) \text{ } q\text{-}s\text{-Schnitt von } G\} \\ &= -\min\{\kappa_G(\bar{X}, X) \mid (X, \bar{X}) \text{ } q\text{-}s\text{-Schnitt von } G\} \end{aligned}$$

16. Der Beweis aus Abschnitt 7.4 für die angegebene Aussage über den Kantenzusammenhang kann nicht auf den Eckenzusammenhang übertragen werden. In einem Graph G kann es eine Ecke a geben, welche in jeder trennenden Eckenmenge jedes Paares e, f von Ecken mit $Z^e(e, f) = Z^e(G)$ liegt. Dies ist z.B. gegeben, wenn eine Ecke a zu jeder anderen Ecke benachbart ist. Betrachten Sie z.B. die Ecken mit Grad 3 in folgendem Graph:



17. Es sei T eine minimale trennende Kantenmenge für a, b . Der Graph $G \setminus T$ ist nicht mehr zusammenhängend und a und b liegen in verschiedenen Zusammenhangskomponenten. Dann liegt c entweder in der gleichen Zusammenhangskomponente wie a oder b oder in einer anderen Zusammenhangskomponente. Auf jeden

Fall ist T eine trennende Eckenmenge für a, c oder b, c . Somit gilt $Z^k(a, b) \geq \min\{Z^k(a, c), Z^k(c, b)\}$.

18. Es sei E die Eckenmenge von G und G' der vollständige Graph mit Eckenmenge E . Jede Kante (a, b) von G' wird mit $|f_{ab}|$ bewertet (man beachte, daß $|f_{ab}| = |f_{ba}|$ gilt, dies gilt nicht für unsymmetrische Netzwerke). Es sei B ein maximal aufspannender Baum von G' und x, y beliebige Ecken aus E . Für die Kanten $k_i = (a_i, b_i)$ eines Weges W von x nach y in B gilt somit:

$$|f_{xy}| \leq \min\{|f_{a_i b_i}| \mid (a_i, b_i) \in W\}$$

In Abschnitt 7.4 wurde bewiesen, daß $|f_{ab}| = Z^k(a, b)$ für jedes Paar von Ecken a, b aus G gilt. Aus Aufgabe 17 folgt somit $|f_{xy}| = \min\{|f_{a_i b_i}| \mid (a_i, b_i) \in W\}$. Zu jedem Paar von Ecken a, b von G gibt es also eine Kante (x, y) von B mit $|f_{ab}| = |f_{xy}|$. Da B $n - 1$ und G' $n(n - 1)/2$ Kanten hat, muß es mindestens $n/2$ Eckenpaare geben, so daß die Werte der entsprechenden maximalen Flüsse alle gleich sind.

19. Es sei N_G das zu G gehörende 0-1-Netzwerk und f ein binärer Fluß auf N_G . Ferner sei W ein einfacher Erweiterungsweg von N_G bezüglich f und f' der durch W entstandene erhöhte Fluß. Die spezielle Struktur von N_G bwirkt, daß sich auf W Vorwärts- und Rückwärtskanten abwechseln. Des weiteren beginnt und endet W mit einer Vorwärtskante. Für jede Ecke e sei $f(e)$ bzw. $f'(e)$ der Fluß durch die Ecke e bezüglich f bzw. f' . Es gilt $f'(e) \geq f(e)$ für jede Ecke e von G .

Die Kanten aus Z induzieren einen binären Fluß f_Z auf N_G (hierzu vergleiche man den Beweis des Lemmas aus Abschnitt 7.1). Es sei f ein binärer maximaler Fluß von N_G , welcher durch eine Folge von Erweiterungswegen aus f_Z entsteht. Dann gilt $f(e) \geq f_Z(e)$ für jede Ecke e von G . Hieraus folgt die zu beweisende Aussage.

20. Sowohl die Ecken- als auch Kantenzusammenhangszahl ist gleich 3.
21. Für beide Graphen gilt $Z^e(G) = 4$.
22. Es gilt $Z^e(G) = Z^k(G) = 3$.
23. Es sei G ein 2-fach kantenzusammenhängender Graph und X die Menge der Ecken, welche in G' von der Startecke s aus erreichbar sind. Ferner sei Y die Menge der restlichen Ecken. Angenommen Y ist nicht leer. Es sei $e \in Y$ die Ecke mit der kleinsten Tiefensuchenummer. Da G zusammenhängend ist, sind alle Ecken aus Y im Tiefensuchebaum Nachfolger von e . Es sei (a, b) eine Kante aus G mit $a \in X$ und $b \in Y$. Im folgenden wird gezeigt, daß es nur eine solche Kante gibt. Dies steht aber im Widerspruch zu $Z^k(G) \geq 2$. Somit ist Y leer. Da b nicht in X liegt, ist (b, a) eine Kante in G' . Wäre (a, b) in G eine Rückwärtskante, so wäre $TSN[b] < TSN[a]$, d.h. a wäre Nachfolger von b im Tiefensuchebaum. Dann wäre aber b von a in G' erreichbar und somit auch b von s . Somit muß (a, b) in G eine Baumkante sein. Da b im Tiefensuchebaum ein Nachfolger von e ist, muß $b = e$ sein und a der Vorgänger von e im Tiefensuchebaum sein. Also ist (a, b) die einzige Kante in G , die Ecken aus X und Y verbindet.

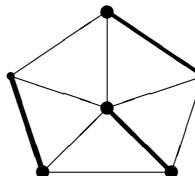
Sei nun umgekehrt jede Ecke in G' von der Startecke s aus erreichbar. Dann ist G zusammenhängend. Angenommen $Z^k(G) = 1$ und $\{k\}$ ist eine trennende Kantenmenge für G . Dann muß $k = (a, b)$ in G eine Baumkante sein. Also ist b in G' nicht von s aus erreichbar.

Der Aufwand ist $O(m)$, da die Tiefensuche zweimal angewendet wird.

24. Es sei R das Rechteck, bei dem die entfernten Felder in diagonal gegenüberliegenden Ecken liegen. Eine Seite von R hat gerade und die andere ungerade Länge. Die Fläche des Schachbrettes außerhalb von R kann in 4 Rechtecke aufgeteilt werden. Mindestens eine Seite jedes Rechteckes hat gerade Länge (eventuell 0). Dieser Teil des Schachbrettes kann somit mit Rechtecken der Größe 1×2 vollständig abgedeckt werden. R ohne die beiden Felder kann in 3 Rechtecke aufgeteilt werden, bei denen jeweils eine Seite gerade Länge hat (eventuell 0). Somit gibt es auch eine Überdeckung für R .

Die schwarzen und weißen Felder des Schachbrettes induzieren einen bipartiten Graphen. Eine vollständige Überdeckung mit Recken entspricht einer vollständigen Zuordnung dieses Graphen. Da zwei diagonal gegenüberliegende Felder die gleiche Farbe haben, besitzt der Restgraph keine vollständige Zuordnung. Somit gibt es auch keine vollständige Überdeckung der Restfläche.

25. a) Zu jeder Kante von M muß eine Ecke aus U inzident sein. Da M eine Zuordnung ist, sind diese Ecken alle verschieden. Somit ist $|M| \leq |U|$.
 b) Die 4 fett gezeichneten Ecken bilden eine minimale Eckenüberdeckung und die 3 fett gezeichneten Kanten bilden eine maximale Zuordnung.



- c) Der Graph G wird zu einem ungerichteten Graphen G' mit zwei zusätzlichen Ecken a und b erweitert. Hierbei ist a zu jeder Ecke aus E_1 und b zu jeder Ecke aus E_2 inzident. Nach dem Satz von Menger gilt $Z^e(a, b) = W^e(a, b)$ für G' . Die Ecken aus einer minimalen trennenden Eckenmenge für a, b bilden eine Eckenüberdeckung. Des Weiteren ist $|W^e(a, b)|$ gleich der Anzahl der Kanten in einer maximalen Zuordnung von G , d.h. $|U| \leq |M|$. Die Aussage folgt nun aus Teil a) dieser Aufgabe.
 d) Gibt es eine vollständige Zuordnung von G , so gilt $|E_1| = |E_2|$ und eine Eckenüberdeckung muß mindestens $|E_1| = (|E_1| + |E_2|)/2$ Ecken enthalten. Für die umgekehrte Beweisrichtung beachte man, daß sowohl E_1 als auch E_2 Eckenüberdeckungen von G sind. Da beide deshalb mindestens $(|E_1| + |E_2|)/2$ Ecken enthalten, folgt daraus $|E_1| = |E_2|$. Nun folgt aus dem Satz von König-Egerváry, daß G eine vollständige Zuordnung besitzt.
 e) Die Cliquenpartitionszahl $\theta(G)$ eines Graphen G ist die minimale Anzahl von Mengen einer Partition der Eckenmenge von G , bei der die von den

Teilmengen induzierten Graphen vollständig sind. Es ist $\theta(G) = \chi(\overline{G})$. Da $\omega(\overline{G}) = \alpha(G)$ ist, genügt es zu zeigen, daß für bipartite Graphen $\theta(G) = \alpha(G)$ ist.

Ist e eine isolierte Ecke von G , so liegt diese in jeder maximalen unabhängigen Menge und $\{e\}$ ist in jeder Cliquenpartition enthalten. Somit kann angenommen werden, daß G keine isolierten Ecken besitzt.

Nach dem Satz von König-Egerváry ist die Anzahl u der Ecken in einer minimalen Eckenüberdeckung von G gleich der Anzahl s der Kanten in einer maximalen Zuordnung. Da $n - \alpha(G) = u$ ist, genügt es zu zeigen, daß $n - \theta(G) = s$ ist. Hieraus folgt, daß $\theta(G) = \alpha(G)$ ist.

Es sei zunächst M eine maximale Zuordnung mit s Kanten. Dann erhält man leicht eine Cliquenpartition mit $s + n - 2s = n - s$ Mengen. Somit gilt $\theta(G) \leq n - s$. Es sei nun M' die von den zwei-elementigen Mengen einer minimalen Cliquenpartition gebildete Zuordnung. Ferner wähle man zu jeder ein-elementigen Menge eine zu dieser Ecke inzidente Kante. Die so entstandene Kantenmenge W bildet einen Wald mit $\theta(G)$ Kanten und n Ecken. Nach den Ergebnissen aus Abschnitt 3.1 besteht W aus $n - \theta(G)$ Zusammenhangskomponenten. Da die Anzahl der Zusammenhangskomponenten durch s beschränkt ist, gilt $n - \theta(G) \leq s$. Hieraus folgt $n - \theta(G) = s$.

- f) Eine Ecke e in einer minimalen Eckenüberdeckung von G überdeckt maximal n Kanten. Hieraus folgt, daß eine minimale Eckenüberdeckung aus mindestens s Ecken bestehen muss. Die Aussage folgt nun aus dem Satz von König-Egerváry.
26. Nach den Ergebnissen aus Abschnitt 5.4 gilt $\delta(G) \leq 5$. Nach dem ersten Satz in Abschnitt 7.4 gilt dann $Z^e(G) \leq \delta(G) \leq 5$.
27. Für die Graphen I_n mit $n \geq 5$ gilt $Z^k(I_n) = Z^e(I_n) = 4$.
28. Sowohl die Ecken- als auch Kantenzusammenhangszahl ist gleich 2.
29. Es seien a, b Ecken mit $Z^e(a, b) = 1$ und $\{x\}$ eine trennende Eckenmenge für a, b . Der Graph $G \setminus \{x\}$ zerfällt in die Zusammenhangskomponenten Z_1, \dots, Z_s mit $s > 1$. Es sei K_i die Menge der Kanten, welche x mit Ecken aus Z_i verbinden. Diese Mengen sind trennende Kantenmengen für a, b . Da x den Grad g hat, gibt es ein K_i mit $|K_i| \leq g/2$. Somit gilt $Z^k(G) \leq |K_i| \leq g/2$.
30. Das Tiefesucheverfahren legt die Reihenfolge, in der die Nachbarn einer Ecke besucht werden, nicht fest. Es wird folgende Reihenfolge betrachtet: für jede Ecke wird zunächst der unbesuchte Nachbar betrachtet, welcher über eine Kante aus Z erreichbar ist (falls vorhanden). Es sei B der hierdurch entstehende Tiefensuchebaum und (x, y) eine beliebige Kante aus Z . Ohne Einschränkung der Allgemeinheit kann angenommen werden, daß x vor y besucht wurde. Somit wurde beim Besuch von x der Nachbar y zuerst betrachtet. Da zu diesem Zeitpunkt y noch nicht besucht wurde, wird (x, y) in B eingefügt. Somit sind alle Kanten aus Z auch in B enthalten.

31. Es sei i eine Ecke von B , welche zu keiner Kante aus Z inzident ist. Somit wurde i im Verlauf des Algorithmus nicht markiert. Beim Verlassen von i durch die Tiefensuche muß somit der Vorgänger j von i schon markiert sein. Da die Tiefensuche in diesem Moment j noch nicht verlassen hat, muß es einen Nachfolger s von j geben, so daß die Kante (j, s) in Z liegt.

Angenommen es gibt in B einen Erweiterungsweg $W = \{e_1, \dots, e_s\}$ bezüglich Z (vergleichen Sie Aufgabe 9). Nach Konstruktion von Z ist $s \geq 4$. Ohne Einschränkung der Allgemeinheit kann angenommen werden, daß e_2 der Vorgänger von e_1 im Tiefensuchebaum ist (sonst ist e_{s-1} ein Vorgänger von e_s). Nach der oben bewiesenen Eigenschaft muß e_3 ein Nachfolger von e_2 sein. Somit ist e_s, e_{s-1}, \dots, e_2 ein gerichteter Weg im Tiefensuchebaum. Analog zu oben zeigt man, daß e_{s-2} ein Nachfolger von e_{s-1} ist. Dieser Widerspruch zeigt, daß es bezüglich Z keinen Erweiterungsweg gibt. Somit ist Z eine maximale Zuordnung von B .

32. Es sei $M \in \mathcal{U}$ mit $f(M) = \max\{f(U) \mid U \in \mathcal{U}\}$. Wird eine Kante aus M entfernt, so entsteht eine isolierte Ecke (da die Anzahl der Ecken unverändert bleibt, würde andernfalls $f(M') > f(M)$ für den neuen Graphen M' gelten). Somit enthält M keinen geschlossenen Weg und M ist ein Wald. Aus dem gleichen Grund gibt es in M auch keinen Weg der Länge 4. Daraus folgt, daß jede Zusammenhangskomponente von M ein Sterngraph ist. Es seien M_1, \dots, M_s die Zusammenhangskomponenten von M . Bezeichnet man mit k die Anzahl der Kanten in M , so enthält M genau $k+s$ Ecken und es gilt $f(M) = (k+s-1)/k$. Angenommen $s > 1$. Nun wird eine neuer Untergraph M' von G gebildet. M' enthält aus jeder Zusammenhangskomponente M_i genau eine Kante. Dann gilt $M' \in \mathcal{U}$ und $f(M') = (2s-1)/s$. Aus $f(M) \geq f(M')$ folgt nun $s \geq k$, d.h. die Kanten in M bilden eine Zuordnung. Aus der Maximalität von $f(M)$ folgt, daß diese Zuordnung maximal ist. Andernfalls ist $s = 1$ und man sieht direkt, daß G bis auf isolierte Ecken ein Sterngraph ist.

33. a) Jede Ecke muß zu mindestens einer Ecke einer anderen Farbklasse benachbart sein, andernfalls würde es eine zweite Färbung von G geben.
- b) Es seien F_1 und F_2 zwei Farbklassen. Angenommen der von $F_1 \cup F_2$ induzierte Untergraph U ist nicht zusammenhängend. Es seien U_1 und U_2 Zusammenhangskomponenten von U . Da jede Ecke zu mindestens einer Ecke einer anderen Farbklasse benachbart ist, enthalten U_1 und U_2 Ecken unterschiedlicher Farben. Vertauscht man die beiden Farben der Ecken in U_1 , so gelangt man zu einer zweiten Färbung von G . Dieser Widerspruch zeigt die Behauptung.
- c) Angenommen es gibt eine Menge M mit $c - 2$ Ecken, so daß der durch das Entfernen dieser Ecken entstehende Graph G' nicht mehr zusammenhängend ist. Dann gibt es zwei Farben f_1 und f_2 , mit denen keine der Ecken in M gefärbt ist. Es seien e_1 bzw. e_2 Ecken, die mit f_1 bzw. f_2 gefärbt sind. Nach Teil b) gibt es zwischen diesen Ecken einen Pfad P , dessen Ecken nur mit f_1 und f_2 gefärbt sind. Somit liegen alle mit f_1 bzw. f_2 gefärbten Ecken in einer einzigen Zusammenhangskomponente G_1 von G' . Ändert man nun die Färbung einer Ecke in $G' \setminus G_1$ in f_1 um, so erhält man eine zweite Färbung von G . Dieser Widerspruch zeigt die Behauptung.

34. Es sei Z eine Zuordnung von G mit t Kanten, Z_1 die Menge der Ecken aus E_1 , welche zu Kanten aus Z inzident sind, und $T \subseteq E_1$. Dann ist

$$|E_1| \geq |T \cup Z_1| = |T| + |Z_1| - |T \cap Z_1|.$$

Wegen $|N(T)| \geq |T \cap Z_1|$ und $|Z_1| = t$ folgt:

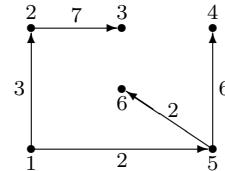
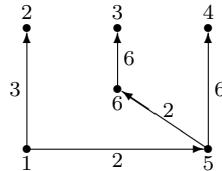
$$|N(T)| \geq |T| + t - |E_1|$$

Sei nun umgekehrt $|N(T)| \geq |T| + t - |E_1|$ für jede Teilmenge T von E_1 . Man betrachte das zugehörige Netzwerk N_G und einen maximalen binären Fluß f . Wie in dem Beweis des Satzes von Hall zeigt man nun $|f| = \kappa(X, \bar{X}) = |E_1 \setminus X| + |N(X \cap E_1)|$. Nach Voraussetzung gilt nun $|f| \geq |E_1 \setminus X| + |X \cap E_1| + t - |E_1| = t$. Sei nun Z die Menge aller Kanten aus G , für die der Fluß f durch die entsprechende Kante in N_G gerade 1 ist. Da jede Ecke aus E_1 in N_G den Eingrad 1 und jede Ecke aus E_2 in N_G den Ausgrad 1 hat, folgt aus der Flußerhaltungsbedingung, daß Z eine Zuordnung mit $|f| \geq t$ Kanten ist.

B.8 Kapitel 8

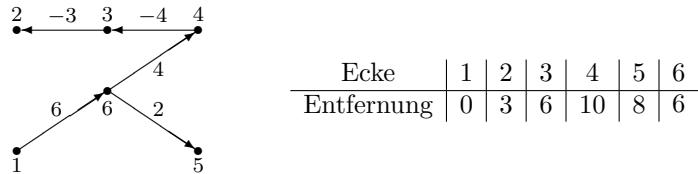
1. Es sei S die Menge der von der Startecke s aus erreichbaren Ecken und G_S der von S induzierte Untergraph von G . Nach Voraussetzung hat G_S die Eigenschaft (*). Eine Anwendung des Algorithmus von Moore und Ford auf G verwendet nur Ecken und Kanten aus G_S , d.h. beide Anwendungen sind identisch. Somit arbeitet der Algorithmus auch auf G korrekt.

2.



3. Der Graph besitzt keine geschlossenen Wege. Mit dem in Aufgabe 8 beschriebenen Verfahren können die kürzesten Wege in linearer Zeit bestimmt werden.
4. a) Der kürzeste Weg von e durch eine vorgegebene dritte Ecke v nach f besteht aus dem kürzesten Weg von e nach v gefolgt von dem kürzesten Weg von v nach f .
- b) Für $i, j = 1, \dots, l$ bestimme die kürzesten Wege von e nach e_i , e_i nach e_j und von e_i nach f (insgesamt $l^2 + l$ Wege). Danach bestimme man unter allen $l!$ Reihenfolgen der Ecken e_1, \dots, e_l den kürzesten der Wege $e \rightarrow e_{i_1} \rightarrow \dots \rightarrow e_{i_l} \rightarrow f$. Hierfür können die in Abschnitt 9.6 beschriebenen Verfahren wie *Branch-and-bound* oder *dynamisches Programmieren* verwendet werden.
5. Es sei B ein minimal aufspannender Baum von G , (e, f) eine der $n - 1$ Kanten von B und W ein kürzester Weg von e nach f in G . Wäre die Länge von W kleiner als die Länge von (e, f) , so würde dies zu einem aufspannenden Baum mit geringeren Kosten als B führen. Somit ist (e, f) der kürzeste Weg von e nach f in G .

6. Die folgende Abbildung zeigt den entstehenden kW-Baum und die kürzesten Entfernungen zur Startecke.



7. Die Prozedur `kürzesteWege` aus Abbildung 8.5 muß wie folgt geändert werden.

```

while W ≠ ∅ do begin
    i := W.entfernen;
    if not W.enthalten(Vorgänger[i]) then
        for jeden Nachfolger j von i do
            verkürzeW(i,j);
    end

```

Es sei i die erste Ecke in der Warteschlange und j der momentane Vorgänger von i im kW-Baum. Dann wurde j nach i in die Warteschlange eingefügt, d.h. der aktuelle Wert von $D[j]$ ist kleiner als zu dem Zeitpunkt, als $D[i]$ zuletzt aktualisiert wurde. Somit ist der aktuelle Wert von $D[i]$ zu hoch. Spätestens bei der Abarbeitung von j wird der Wert von $D[i]$ erniedrigt und i erneut in die Warteschlange eingefügt. Somit ist es überflüssig, Ecke i vor ihrem aktuellen Vorgänger j zu bearbeiten. Die Korrektheit dieser Variante des Algorithmus von Moore und Ford ergibt sich aus dieser Beobachtung und der Korrektheit der Orginalversion. Diese Variante hat in vielen Fällen eine geringere Laufzeit. Die *worst case* Komplexität bleibt aber unverändert. Dazu betrachte man die in der Lösung von Aufgabe 24 angegebene Folge von Graphen. Für diese ergibt sich keine Verbesserung, d.h. die Anzahl der Durchläufe der `while`-Schleife ist weiterhin $O(n^2)$.

8. Die Ecken eines gerichteten, kreisfreien Graphen seien mit ihren topologischen Sortierungsnummern numeriert. Der folgende Algorithmus bestimmt einen kW-Baum für eine beliebige Ecke.

```

procedure kWbaum (G : K-G-Graph; start : Integer);
var i,j : Integer;
begin
    initkW(start);
    for i := start to n-1 do
        for jeden Nachbar j von i do
            verkürze(i,j);
    end

```

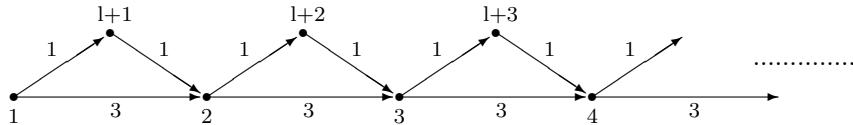
Ohne Einschränkung der Allgemeinheit kann $start = 1$ angenommen werden. Die Korrektheit dieser Prozedur ergibt sich aus folgender Aussage: Nach dem Ende des i -ten Durchlaufs gilt $D[j] = d(start, j)$ für $j = 1, \dots, i + 1$. Diese Aussage wird mit Hilfe von vollständiger Induktion nach i bewiesen. Zu jedem Zeitpunkt gilt $D[j] \geq d(start, j)$. Da Ecke 2 nur einen Vorgänger hat, gilt nach dem ersten

Durchlauf $D[j] = d(start, j)$ für $j = 1, 2$. Sei nun $i > 1$. Nach dem $i - 1$ -ten Durchgang werden die Werte für $D[1], \dots, D[i]$ nicht mehr geändert. Es muß also nur gezeigt werden, daß nach dem i -ten Durchgang $D[i + 1] = d(start, i + 1)$ gilt. Es sei j der Vorgänger von $i + 1$ auf einem kürzesten Weg von 1 nach $i + 1$. Dann gilt $j < i + 1$ bzw. $j \leq i$ und somit war $D[j] = d(start, j)$ vor dem j -ten Durchgang. Nach dem j -ten Durchgang gilt:

$$D[i + 1] \leq D[j] + B[j, i + 1] = d(start, i + 1)$$

Wegen $d(start, i + 1) \leq D[i + 1]$ gilt also $D[i + 1] = d(start, i + 1)$ schon nach dem j -ten und somit erst recht nach dem i -ten Durchgang.

9. Es sei n eine ungerade Zahl und $l = (n + 1)/2$. Der unten dargestellte kantenbewertete gerichtete Graph mit n Ecken und $3(n - 1)/2$ Kanten ist kreisfrei. Bei der Anwendung des Algorithmus von Moore und Ford auf diesen Graph werden die Nachfolger jeder Ecke in der Reihenfolge aufsteigender Eckennummern betrachtet. Für $j = 1, \dots, l - 1$ gilt: Die Ecken j und $l + j$ werden j -mal in die Warteschlange eingefügt. Somit ist die Laufzeit des Algorithmus für diesen Graph $O(n^2)$.



10. Die folgende Tabelle zeigt die Werte von B, Min, D und Vorgänger für die einzelnen Schritte.

B	Min	D								Vorgänger							
		1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
3		∞	∞	0	∞	∞	∞	∞	∞	0	0	0	0	0	0	0	0
2,4	3	∞	3	0	2	∞	∞	∞	∞	0	3	0	3	0	0	0	0
2,1,5	4	6	3	0	2	10	∞	∞	∞	4	3	0	3	4	0	0	0
1,5,8	2	4	3	0	2	10	∞	∞	7	2	3	0	3	4	0	0	2
5,8	1	4	3	0	2	7	∞	∞	7	2	3	0	3	1	0	0	2
8,6,7	5	4	3	0	2	7	13	12	7	2	3	0	3	1	5	5	2
6,7	8	4	3	0	2	7	13	12	7	2	3	0	3	1	5	5	2
6	7	4	3	0	2	7	13	12	7	2	3	0	3	1	5	5	2
	6	4	3	0	2	7	13	12	7	2	3	0	3	1	5	5	2

11. Der folgende Algorithmus basiert auf der Tiefensuche und hat eine Laufzeit von $O(m)$. Eine neue Ecke i wird nur dann besucht, wenn Sie nicht schon als besucht markiert ist und wenn die Entfernung im Tiefensuchebaum von der Startecke e mit $d(e, i)$ übereinstimmt. Zum Beweis der Korrektheit der Prozedur kWBaum genügt es zu zeigen, daß alle Ecken erreicht werden. Es sei v eine beliebige Ecke von G und W der Weg von der Startecke zu v in einem beliebigen kW-Baum (da G die Eigenschaft (*) hat, muß es einen kW-Baum geben). Angenommen v ist nicht in dem erzeugten Baum B enthalten. Es sei j die erste Ecke auf W , welche

nicht in B ist. Dann ist der Vorgänger i von j auf W in B enthalten und es gilt $d(e, j) = d(e, i) + b(i, j)$. Beim Besuch von i durch **aufbauen** wird dann j besucht. Dieser Widerspruch zeigt, daß B ein kW-Baum ist.

```

var Besucht : array[1..max] of Boolean;
      Vorgänger : array[1..max] of Integer;
procedure aufbauen(i,e : Integer);
var j : Integer;
begin
    Besucht[i] := true;
    for jeden Nachbar j von i do
        if Besucht[j] = false and d(e,j) = d(e,i) + b(i,j) then begin
            Vorgänger[j] := i;
            aufbauen(j,e);
        end
    end
procedure kWBaum(G : B-Graph; e : Integer);
begin
    Initialisiere Besucht mit false und Vorgänger mit 0;
    aufbauen(e,e);
end

```

12. Da G die Eigenschaft (*) erfüllt, gilt das Optimalitätsprinzip. Dann ist der Teilweg W_1 von e_1 nach f auf W ein kürzester Weg von e_1 nach f . Für W_1 gilt wieder das Optimalitätsprinzip, somit gilt $L(\overline{W}) = d(e_1, f_1)$, d.h. \overline{W} ist ein kürzester Weg.
13. Es seien a und b positive reelle Zahlen. Dann gilt: $ab = 1/e^{-\log a - \log b}$. Da die e -Funktion monoton steigend ist, ist das Produkt ab um so größer, um so kleiner $-\log a - \log b$ ist. Für $a \in [0, 1]$ ist $-\log a > 0$. Ändert man die Bewertung jeder Kante von b_{ij} auf $-\log b_{ij}$, so kann der Algorithmus von Dijkstra angewendet werden. Dann entsprechen die kürzesten Wege bezüglich der neuen Bewertung den Wegen, bei den das Produkt der einzelnen Wahrscheinlichkeiten maximal ist. Der Übertragungsweg von Sender 1 zum Empfänger 5 mit der größten Wahrscheinlichkeit einer korrekten Übertragung ist 1, 8, 9, 3, 4, 5. Die Übertragungswahrscheinlichkeit ist 0.40824.
14. Während der Ausführung des Algorithmus von Dijkstra gilt:

$$W = \{D[i] \mid i \in B\} \subseteq \{s, s+1, s+2, \dots, s+C\}$$

wobei $s = \min W$ ist. Diese Eigenschaft ist zu Beginn des Algorithmus erfüllt und bleibt auch erhalten, wenn eine Ecke i mit minimalen Wert entfernt wird. Danach werden die Nachbarn j von i bearbeitet und es gilt $D[i] \leq D[i] + B[i, j]$ bzw. $s \leq D[j] \leq s + C$. Also bleibt die obige Eigenschaft stets erhalten. Zu jedem Zeitpunkt gilt:

$$\{D[i] \bmod (C+1) \mid i \in B\} \subseteq \{0, \dots, C\}$$

Dies macht man sich bei der Speicherung der Werte $D[i]$ zunutze. Es werden $C+1$ Fächer vorgesehen. In das i -te Fach werden die Werte abgespeichert, welche

modulo $C + 1$ gleich i sind. Ein Fach wird durch eine doppelt verkettete Liste implementiert. In einem Feld FA der Länge $C + 1$ werden die Zeiger auf die Fächer verwaltet. Damit das Ändern des Wertes einer Ecke in konstanter Zeit erfolgen kann, wird für jede Ecke ein Zeiger zu dem zugehörigen Element im entsprechenden Fach abgespeichert. Hierzu dient das Feld E der Länge n . Folgende Datenstrukturen werden verwendet.

```

Fach = record
    vorgänger, nachfolger : zeiger Fach;
    nummer : Integer;
end;
Ecke = record
    wertZeiger : zeiger Fach;
    wert : Integer;
end;
E : array[1..n] of Ecke;
FA : array[0..C] of zeiger Fach;
```

Mit Hilfe dieser Datenstrukturen kann das Einfügen einer neuen Ecke in konstanter Zeit realisiert werden. Jede Ecke wird genau einmal betrachtet. Da die Bearbeitung eines Nachbarn in konstanter Zeit erfolgt, ist der Gesamtaufwand für alle Änderungsoperationen $O(m)$. Es bleibt noch zu zeigen, wie die Ecken mit minimalem Wert gefunden werden.

Die einfachste Variante verwaltet den Index \min des Faches mit dem kleinsten Wert. Da die minimalen Werte im Laufe des Algorithmus immer größer werden, kann dieser Index leicht aktualisiert werden:

```

while FA[min] = leere Liste do
    min := min + 1 mod (C+1);
```

Der Aufwand hierfür ist $O(L)$, wobei L die Länge des längsten aller kürzesten Wege ist. Wegen $L \leq (n - 1)C$ ist der Gesamtaufwand für den Algorithmus von Dijkstra $O(m + nC)$. Man beachte, daß der Specheraufwand jetzt auch von C abhängt.

Eine andere Variante besteht darin, die Indices des Feldes FA , welche nichtleere Listen enthalten, in einem Heap zu verwalten. Dann sind zu jedem Zeitpunkt maximal $C + 1$ Elemente im Heap. Jede einzelne Heap-Operation hat den Aufwand $O(\log C)$. Der Gesamtaufwand für den Algorithmus von Dijkstra ist bei dieser Realisierung $O((m + n) \log C)$.

15. Ohne Einschränkung der Allgemeinheit kann man annehmen, daß alle Kanten die Bewertung 1 haben (vergleichen Sie Aufgabe 23). Eine in Abschnitt 8.2 bewiesene Folgerung des Optimalitätsprinzip ist: Für alle Kanten (e, f) gilt:

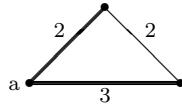
$$d(s, f) \leq d(s, e) + B[e, f]$$

Die Aussage der Aufgabe folgt nun aus Aufgabe 31 in Kapitel 4.

16. Die folgende Tabelle zeigt die Werte von D und Vorgänger für einen kW-Baum mit Startecke 1.

Startecke	D					Vorgänger				
	1	2	3	4	5	1	2	3	4	5
1	0	3	6	3	2	0	1	2	3	4

17. Der Korrektheitsbeweis für diese Version des Algorithmus von Moore und Ford folgt dem Orginalbeweis aus Abschnitt 8.3. Untersuchungen haben gezeigt, daß diese Variante in vielen Fällen eine geringere Laufzeit aufweist. Wendet man den neuen Algorithmus auf die in der Lösung von Aufgabe 24 beschriebenen Graphen an, so stellt man allerdings fest, daß die Anzahl der Ausführungen der `while`-Schleife unverändert ist.
18. Man vergleiche hierzu die Lösung von Aufgabe 14. Sind die Kantenbewertungen reelle Zahlen, so haben die Ecken innerhalb eines Faches nicht unbedingt die gleichen Werte. Dies erhöht die Zugriffszeit und die in Aufgabe 14 angegebenen Laufzeiten werden nicht erreicht. Entscheidend ist die Verteilung der Ecken auf die Fächer. Je gleichmäßiger die Verteilung ist, desto größer ist der Vorteil von Bucket-Sort.
19. Es wird ein neuer Graph G' mit gleicher Ecken- und Kantenmenge gebildet. Die Kanten (e, f) von G' tragen die Bewertung $B'[e, f] = B[e, f] + (B[e] + B[f])/2$. Genau dann ist W ein kürzester Weg von s nach z in G' bezüglich B' , wenn W ein kürzester Weg von s nach z in G bezüglich B ist. Die Länge von W in G ist gleich $L'(W) - (B[s] + B[z])/2$, wobei $L'(W)$ die Länge von W in G' ist.
20. Die fett gezeichneten Kanten bilden einen kW-Baum für Ecke a , aber keinen minimal aufspannenden Baum.



21. a) Distanz- und Vorgängermatrix des Graphen aus Übungsaufgabe 6:

$$D = \begin{pmatrix} 0 & 3 & 6 & 10 & 8 & 6 \\ \infty & 0 & \infty & \infty & \infty & \infty \\ \infty & -3 & 0 & 5 & 3 & 1 \\ \infty & -7 & -4 & 0 & -1 & -3 \\ \infty & -5 & -2 & 2 & 0 & -1 \\ \infty & -3 & 0 & 4 & 2 & 0 \end{pmatrix} \quad V = \begin{pmatrix} 0 & 3 & 4 & 6 & 6 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 5 & 6 & 3 \\ 0 & 3 & 4 & 0 & 6 & 3 \\ 0 & 3 & 4 & 5 & 0 & 3 \\ 0 & 2 & 4 & 6 & 6 & 0 \end{pmatrix}$$

- b) Die Werte für die Kantenbewertung B' sind: $b_{12} = 11$, $b_{15} = 10$, $b_{16} = 9$, $b_{32} = 0$, $b_{36} = 0$, $b_{43} = 0$, $b_{54} = 1$, $b_{64} = 1$, $b_{65} = 0$.
22. Zunächst wird der Abstand zwischen allen Paaren von Orten bestimmt. Danach wird für jeden Ort die weiteste Entfernung bestimmt (größter Wert in der entsprechenden Zeile der Distanzmatrix) und der Ort mit der kleinsten weitesten

Entfernung ausgewählt. In diesem Fall ist es Ort 2 mit einer maximalen Entfernung von 180. Im folgenden ist die Distanzmatrix dargestellt. Die weitesten Wegstrecken für die einzelnen Orte sind eingerahmt.

$$D = \begin{pmatrix} 0 & 100 & 250 & 280 & 140 & \boxed{260} \\ 100 & 0 & 150 & \boxed{180} & 40 & 160 \\ \boxed{250} & 150 & 0 & 30 & 190 & 70 \\ \boxed{280} & 180 & 30 & 0 & 220 & 100 \\ 140 & 40 & 190 & \boxed{220} & 0 & 120 \\ \boxed{260} & 160 & 70 & 100 & 120 & 0 \end{pmatrix}$$

23. Es sei W ein Weg bestehend aus l Kanten, $L(W)$ bzw. $L'(W)$ bezeichne die Länge von W vor bzw. nach der Erhöhung um C . Im ersten Fall gilt $L'(W) = L(W) + lC$ und im zweiten Fall $L'(W) = CL(W)$. Im zweiten Fall bleiben die kürzesten Wege die gleichen.
24. Der Graph hat die Eigenschaft (*): Jeder geschlossene Weg hat mindestens eine Kante mit Bewertung 64, da alle Kanten (i, j) mit $i < j$ die Bewertung 64 haben. Der kW-Baum mit Startecke 1 ist der Weg 1, 6, 5, 4, 3, 2 und die Längen der kürzesten Wege sind 64, 47, 38, 33, 30. Die **while**-Schleife wird 16 mal ausgeführt.
25. Die Werte für die Kantenbewertung B' sind: $b_{12} = 0$, $b_{14} = 4$, $b_{23} = 2$, $b_{31} = 1$, $b_{34} = 0$, $b_{45} = 0$, $b_{51} = 0$.
26. Der Graph hat nicht die Eigenschaft (*), der geschlossene Weg 6, 5, 3 hat die Länge -1 . Wendet man die Prozedur **floyd** an, so wird der Diagonaleintrag $D[6, 6]$ im vorletzten Durchgang negativ.
27. Mit Hilfe der Tiefensuche wird für jede Ecke u die Länge $d_B(e, u)$ des kürzesten Weges von e nach u in B bestimmt (Aufwand $O(n)$). Nach den Ergebnissen aus Abschnitt 8.2 ist B genau dann ein kW-Baum, wenn für jede Kante (u, v) von G die Ungleichungen $d_B(e, u) + B[u, v] \geq d_B(e, v)$ und $d_B(e, v) + B[u, v] \geq d_B(e, u)$ erfüllt sind. Dies kann mit Aufwand $O(m)$ überprüft werden.
28. Falls es einen geschlossenen Weg W mit negativer Länge gibt, so gibt es eine Ecke **start** auf W , so daß alle in **start** startenden Teilwege von W negative Länge haben. Die Prozedur **negativerKreis** überprüft die Existenz eines solchen geschlossenen Weges für die Ecke **start**. Wird ein solcher Weg gefunden, so wird das Programm beendet. Der Graph hat dann nicht die Eigenschaft (*). Im Hauptprogramm wird diese Prozedur für jede Ecke aufgerufen. Gibt es für keine Ecke des Graphen einen solchen Weg, so erfüllt der Graph die Eigenschaft (*).

```

for jede Ecke i do
    negativerKreis(i, i, 0);
    Exit('Graph hat die Eigenschaft (*)')

procedure negativerKreis(start, akt, länge : Integer);
var j : Integer;
begin

```

```

for jeden Nachfolger j von akt do
  if läng + bew(akt,j) < 0 then
    if j = start then
      Exit('Graph hat nicht die Eigenschaft (*)')
    else
      negativerKreis(start, j, läng + bew(akt,j));
  end

```

29. Die folgende Tabelle zeigt die Werte von D und Vorgänger für einen IW-Baum (*längste Wege Baum*) mit Startecke 1.

Startecke	D						Vorgänger					
	1	2	3	4	5	6	1	2	3	4	5	6
	0	1	-9	-3	3	-7	0	1	4	2	4	3

30. Die letzte **if**-Anweisung im Programm aus Abbildung 8.23 muß wie folgt abgeändert werden.

```

if max{D[i,j],D[j,k]} < ∞ and D[i,k] > min{D[i,j],D[j,k]} then begin
  D[i,k] := min{D[i,j],D[j,k]};
  V[i,k] := V[j,k];
end

```

31. Da f_1 und f_2 konsistent sind, gilt $0 \leq f_i(u) \leq B[u, v] + f_i(v)$ für $i = 1, 2$ und jede Kante (u, v) . Multipliziert man die erste Ungleichung mit a und die zweite mit b und addiert diese dann, so erhält man:

$$0 \leq af_1(u) + bf_2(u) \leq (a+b)B[u, v] + af_1(v) + bf_2(v).$$

Dividiert man diese Ungleichung durch $a+b$, so folgt die Konsistenz von $(af_1 + bf_2)/(a+b)$.

32. Für einen beliebigen Brettzustand b und einen Zielzustand z sei

$f_1(b)$ die Anzahl der Plättchen, welche in b eine andere Position als in z haben;
 $f_2(b)$ die Summe der für jedes Plättchen von b (ohne Beachtung der anderen Plättchen) mindestens notwendigen Verschiebungen, um die Position in z zu erreichen.

Für die in Abbildung 4.30 dargestellte Startstellung s gilt $f_1(s) = 8$ und $f_2(s) = 13$. Sowohl f_1 als auch f_2 sind zulässig. Beide Schätzfunktionen können mit Hilfe von Metriken auf dem Raum $Z \times Z$ definiert werden: für a, b aus $Z \times Z$ sei $d_1(a, b) = 0$, falls $a = b$, und 1, falls $a \neq b$ und $d_2(a, b) = |a_x - b_x| + |a_y - b_y|$. Bezeichnet man mit $P_a(i)$ die Position von Plättchen i im Zustand a , so gilt

$$f_i(a) = d_i(P_a(1), P_z(1)) + \dots + d_i(P_a(8), P_z(8))$$

für $i = 1, 2$. Aus der Dreiecksungleichung für d_1 und d_2 folgt:

$$f_i(a) - f_i(b) = \sum_{j=1}^8 d_i(P_a(j), P_z(j)) - d_i(P_b(j), P_z(j)) \leq \sum_{j=1}^8 d_i(P_a(j), P_b(j))$$

Da die rechte Seite dieser Ungleichung höchstens so groß ist, wie die minimale Anzahl von Verschiebungen, um Zustand b von Zustand a aus zu erreichen, sind f_1 und f_2 konsistent. Wegen $f_1(a) \leq f_2(a)$ wird das Ziel mit Hilfe der zweiten Schätzfunktion im Allgemeinen schneller gefunden.

33. Startend mit dem trivialen Fluß f_0 werden mit Hilfe von Erweiterungswegen minimaler Kosten neue kostenminimale Flüsse f_i bestimmt. Dazu beachte man, daß der triviale Fluß ein Fluß mit Wert 0 und minimalen Kosten ist. Einen Erweiterungsweg mit minimalen Kosten findet man mit dem Algorithmus von Moore und Ford. Dieser Algorithmus ist anwendbar, da es in den Graphen G_{f_i} nach Teil a) des Satzes aus Abschnitt 6.6 keine geschlossenen Wege negativer Länge gibt. Das Verfahren wird beendet, sobald Ecke s in G_{f_i} nicht mehr von q aus erreichbar ist. Dann liegt ein maximaler Fluß mit minimalen Kosten vor. Da alle Kapazitäten ganzzahlig sind, gilt $|f_{i+1}| \geq |f_i| + 1$, d.h. der Algorithmus terminiert nach maximal $|f_{max}|$ Durchgängen. Unter Verwendung des Algorithmus von Moore und Ford ergibt sich eine Laufzeit von $O(|f_{max}|nm)$.
34. Der in Abbildung 8.8 dargestellte Algorithmus von Dijkstra expandiert immer die Ecke aus dem kW-Baum, welche aktuell den kürzesten Abstand zur Startecke hat. Zur Lösung der gestellten Aufgabe muß die Ordnung auf den Ecken des kW-Baumes neu definiert werden. Ist die Entfernung zweier Ecken zur Startecke gleich, so wird die Ecke, bei der der Weg von der Startecke weniger Kanten enthält, als kleiner angesehen. Hierzu wird neben der eigentlichen Entfernung zur Startecke auch die Anzahl der Kanten des Weges im kW-Baum in einem Feld `kantenAnzahl` gespeichert. Dieses Feld wird mit ∞ initialisiert. Die Prozedur `verkürze` muß wie folgt geändert werden.

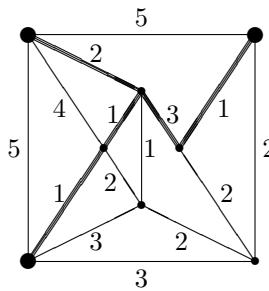
```

procedure verkürze (i,j : Integer);
begin
  if D[i] + B[i,j] < D[j] then begin
    D[j] := D[i] + B[i,j];
    Vorgänger[j] := i;
    kantenAnzahl[j] := kantenAnzahl[i] + 1;
  end
  else
    if D[i] + B[i,j] = D[j] and
        kantenAnzahl[j] > kantenAnzahl[i] + 1 then begin
      Vorgänger[j] := i;
      kantenAnzahl[j] := kantenAnzahl[i] + 1;
    end
  end
end

```

Die Implementierung des Algorithmus von Dijkstra wird an zwei Stellen geändert. Die Auswahl einer Ecke aus B erfolgt gemäß der neuen Ordnung und beim Einfügen einer Ecke in B wird das Feld `kantenAnzahl` auf 1 gesetzt.

35. Die fett dargestellten Kanten bilden einen minimalen Steinerbaum mit 3 Steiner-ecken und Kosten 8.



B.9 Kapitel 9

1. a) Ohne Einschränkung der Allgemeinheit kann angenommen werden, daß nach der Zuordnung der Programme zu Prozessoren Prozessor P_1 die höchste Last hat. Bezeichne mit T die Gesamtaufzeit aller P_1 zugeordneter Programme. Das zuletzt an P_1 zugeordnete Programm habe die Laufzeit t_j . Da Programm j als letztes zugeordnet wurde, hat jeder Prozessor mindestens eine Laufzeit von $T - t_j$. Hieraus folgt:

$$\sum_{i=1}^n t_i \geq m(T - t_j) + t_j$$

Ferner ist $A(n) = T$ und somit gilt

$$OPT(n) \geq \frac{\sum_{i=1}^n t_i}{m} \geq (T - t_j) + \frac{t_j}{m} = A(n) - (1 - \frac{1}{m})t_j.$$

Wegen $t_j \leq OPT(n)$ folgt

$$A(n) \leq OPT(n) + (1 - \frac{1}{m})t_j \leq OPT(n)(2 - \frac{1}{m}).$$

Hieraus ergibt sich die Behauptung.

- b) Die Lasten der einzelnen Prozessoren werden in einem Heap verwaltet. In jedem Schritt wird der Prozessor mit der geringsten Last ausgewählt und die Last dieses Prozessors wird um die Laufzeit des aktuellen Programmes erhöht. Diese Operation hat den Aufwand $O(\log m)$. Für alle n Programme ergibt sich zusammen der Aufwand $O(n \log m)$.
- c) Es sei $n = m(m - 1) + 1$, die ersten $n - 1$ Programme haben die Laufzeit 1 und das letzte Programm die Laufzeit m . Hieraus folgt $A(n) = 2m - 1$ und $OPT(n) = m$. Somit ist $\mathcal{W}_A(n) \leq 2 - \frac{1}{m}$.
- d) Da das zuletzt zugeordnete Programm die kürzeste Laufzeit hat, gilt

$$t_j \leq \frac{\sum_{i=1}^n t_i}{n} \leq \frac{m}{n} OPT(n).$$

Hieraus folgt

$$\mathcal{W}_A(n) \leq 1 + \frac{m}{n}(1 - \frac{1}{m}).$$

Wegen $n > m$ wird also der Wirkungsgrad verbessert. Mit einer genaueren Analyse kann gezeigt werden, daß $\mathcal{W}_A(n) \leq \frac{4}{3} - \frac{1}{3m}$ gilt.

2. Aus den $(2n)!$ möglichen Reihenfolgen sind die zu bestimmen, die zu einer 2-Färbung führen. Im folgenden werden die Ecken auf der linken Seite mit E_1 und auf der rechten Seite mit E_2 bezeichnet. Sind die ersten beiden Ecken aus E_1 , so bekommen diese die Farbe 1. Daraufhin kann keine der Ecken aus E_2 die Farbe 1 mehr bekommen. Somit wird in diesem Fall eine 2-Färbung erzeugt. Die gleiche Aussage gilt für E_2 . Dies sind insgesamt $2n(n-1)(2n-2)!$ verschiedene Reihenfolgen. Sind die ersten beiden Ecken durch eine Kante verbunden und die dritte Ecke entweder aus der gleichen Menge wie die erste Ecke oder mit der ersten Ecke verbunden, so wird ebenfalls eine 2-Färbung erzeugt. Dies sind noch einmal $2n(n-1)(2n-3)(2n-3)!$ verschiedene Reihenfolgen. In allen anderen Fällen bekommen eine Ecke aus E_1 und eine Ecke aus E_2 die gleiche Farbe und dadurch werden mindestens 3 Farben vergeben. Somit ist die Wahrscheinlichkeit, daß der Greedy-Algorithmus eine 2-Färbung erzeugt, gleich

$$\frac{2n(n-1)(2n-2)! + 2n(n-1)(2n-3)(2n-3)!}{(2n)!} = \frac{4n-5}{4n-2}.$$

Die Wahrscheinlichkeit, daß der Algorithmus für diesen Graphen eine optimale Färbung erzielt, strebt also mit wachsendem n gegen 1. Für $n = 20$ liegt diese Wahrscheinlichkeit schon über 96%.

3. Numeriere die Ecken des Graphen im Uhrzeigersinn beginnend bei der Ecke oben links mit der Nummer 1. Für die Reihenfolge 1, 4, 6, 3, 2, 5 vergibt der Greedy-Algorithmus vier Farben und für die Reihenfolge 1, 5, 3, 6, 2, 4 zwei Farben.

4.

Ecke	1	2	3	4	5	6
Greedy-Algorithmus	1	2	1	3	2	4
Johnson Algorithmus	1	1	2	2	3	4
Optimale Färbung	1	2	3	3	2	1

5. Mittels vollständiger Induktion nach i wird gezeigt, daß Ecken aus $F_{\pi(i)}$ eine Farbe mit der Nummer i oder kleiner bekommen. Hieraus folgt dann direkt die Behauptung. Die Ecken aus $F_{\pi(i)}$ haben bezüglich f alle die gleiche Farbe, d.h. sie sind nicht benachbart. Somit bekommen die Ecken aus $F_{\pi(1)}$ alle die Farbe 1 zugeordnet. Angenommen eine Ecke a aus $F_{\pi(i)}$ bekommt die Farbe $i+1$ zugeordnet. Somit muss a zu einer Ecke b mit Farbe i benachbart sein. Nach Induktionsvoraussetzung kann b nicht in $F_{\pi(1)}, \dots, F_{\pi(i-1)}$ liegen. Also liegen a und b in $F_{\pi(i)}$. Dies bedeutet aber, daß a und b nicht benachbart sind. Dieser Widerspruch zeigt die Behauptung.

Wendet man den Greedy-Algorithmus in der angegebenen Reihenfolge auf die Ecken des Graphen aus Aufgabe 4 an, so erhält man eine Färbung mit vier Farben.

Ordnet man die Ecken nach absteigenden Farbklassen um, d.h. man betrachtet die Ecken in der Reihenfolge $(6, 4, 5, 2, 3, 1)$, dann erhält man hingegen eine Färbung mit drei Farben.

Es sei G ein bipartiter Graph mit Eckenmenge $E = \{a_1, \dots, a_n\} \cup \{b_1, \dots, b_n\}$ und Kantenmenge $K = \{(a_i, b_j) \mid 1 \leq i, j \leq n, i \neq j\}$. Im ungünstigsten Fall ordnet der Greedy-Algorithmus für jedes i den Ecken a_i und b_i die gleiche Farbe zu. Dann vergibt der Algorithmus n Farben, obwohl 2 ausreichen. In diesem Fall wird für jede Permutation der Farbklassen ebenfalls eine Färbung mit n Farben erzeugt.

6. a) Der Greedy-Algorithmus betrachtet bei der Bestimmung der Farbe einer Ecke e die schon gefärbten Nachbarn und wählt dann die kleinste Farbnummer aus, welche noch nicht für diese Nachbarn verwendet wurde. Zur Färbung der i -ten Ecke e_i werden $\min(i - 1, g(e_i))$ Nachbarn betrachtet, d.h. die Farbnummer von e_i ist maximal $\min(i, g(e_i) + 1)$. Eine Obergrenze für die Anzahl der durch den Greedy-Algorithmus vergebenen Farben ist:

$$\max \{\min(i, g(e_i) + 1) \mid i = 1, \dots, n\}$$

- b) Die Ecken der folgenden Menge liegen innerhalb der ersten k Farbklassen:

$$\{e_i \mid g(e_i) < k\} \cup \{e_1, \dots, e_k\}$$

Es gilt also, diese Menge für festes k möglichst groß zu machen. Dies erreicht man, indem Ecken mit kleinem Eckengrad über die erste Teilmenge und Ecken mit großem Eckengrad über die zweite Teilmenge abgedeckt werden. Hierzu betrachtet man beispielsweise die Ecken in der Reihenfolge absteigender Eckengrade.

- c) Alle Ecken des Graphen H_i haben den Eckengrad i . Betrachtet man die Ecken der Graphen H_i in der Reihenfolge absteigender Eckennummern (d.h. $2i - 1, 2i - 2, \dots, 1$), so vergibt der Greedy-Algorithmus $i + 1 = n/2$ Farben. Somit gilt für den Wirkungsgrad: $\mathcal{W}_A(n) \geq n/4$ und es ist $\mathcal{W}_A^\infty = \infty$.
7. a) Es sei G ein Graph, der entsteht, wenn die Wurzel eines Baumes der Höhe 1 und $s + 1$ Ecken mit einer beliebigen Ecke des vollständigen Graphen K_s verbunden wird. G hat $2s + 1$ Ecken und $\omega(G) = s$. Die Wurzel des Baumes hat in G maximalen Eckengrad. Entfernt man diese und alle nicht zu ihr benachbarten Ecken, so verbleiben $s + 1$ isolierte Ecken. Somit erzeugt A_1 eine Clique der Größe 2 und es gilt $OPT(G)/A_1(G) = s/2 = (n - 1)/4$ bzw. $\mathcal{W}_{A_1}^\infty = \infty$.
- b) Es sei G ein Graph G , der entsteht, wenn eine beliebige Ecke aus dem vollständigen Graphen K_s mit einer beliebigen Ecke aus dem vollständig bipartiten Graphen $K_{s,s}$ verbunden wird. G hat $3s$ Ecken und $\omega(G) = s$. In K_s gibt es $s - 1$ Ecken mit Eckengrad $s - 1$, diese werden von A_2 als Erstes aus G entfernt. Somit erzeugt A_2 eine Clique der Größe 2 und es gilt $OPT(G)/A_2(G) = s/2 = n/6$ bzw. $\mathcal{W}_{A_2}^\infty = \infty$.

8. In Kapitel 5 wurde gezeigt, daß die Größe einer maximalen Clique $\omega(G)$ gleich der Größe einer maximalen unabhängigen Menge $\alpha(\bar{G})$ des Komplementes des Graphen ist. Somit ergibt sich aus jedem approximativen Algorithmus für das Optimierungsproblem von Cliques direkt ein approximativer Algorithmus für das Optimierungsproblem der unabhängigen Menge mit gleichem Wirkungsgrad. Die in der Aufgabe gemachte Aussage folgt nun direkt aus dem letzten Satz aus Abschnitt 9.4.
9. Ein Erweiterungsweg bezüglich einer Zuordnung Z ist ein Weg, der bei einer nicht zugeordneten Ecke beginnt und endet und bei dem jede zweite Kante nicht in Z enthalten ist. Mit Hilfe eines Erweiterungsweges W kann eine gegebene Zuordnung vergrößert werden. Hierzu werden aus Z alle Kanten entfernt, welche auf W liegen. Die restlichen Kanten von W werden in Z eingefügt. Die so entstandene Zuordnung enthält eine Kante mehr als die ursprüngliche Zuordnung. Zum Beweis der Aussage in der Aufgabe wird der Graph G' betrachtet. Da die Kanten von G' aus zwei Zuordnungen stammen, ist der Eckengrad jeder Ecke von G' kleiner oder gleich 2. Es sei H eine Zusammenhangskomponente von G' . H ist entweder eine isolierte Ecke, ein einfacher, geschlossener Weg mit der gleichen Anzahl von Kanten aus Z_1 und Z_2 oder ein offener einfacher Weg, dessen Kanten abwechselnd aus Z_1 und Z_2 stammen. Komponenten vom letzten Typ mit einer ungeraden Anzahl von Kanten sind entweder Erweiterungswäge bezüglich Z_1 oder Z_2 (je nachdem ob mehr Kanten aus Z_2 oder Z_1 stammen). Da alle Kanten aus Z_1 und Z_2 auf die Zusammenhangskomponenten verteilt sind, muss es $|Z_2| - |Z_1|$ Komponenten vom letzten Typ geben, bei denen die Mehrzahl der Kanten aus Z_2 kommt. Dies sind alles eckendisjunkte Erweiterungswäge bezüglich Z_1 .
10. Die Funktion **greedy-Zuordnung** erzeugt offensichtlich eine nicht erweiterbare Zuordnung Z von G . Es sei M eine maximale Zuordnung von G . Dann gibt es nach der letzten Aufgabe $|M| - |Z|$ eckendisjunkte Erweiterungswäge bezüglich Z . Nach Konstruktion enthält jeder dieser Erweiterungswäge mindestens zwei Kanten aus M . Da die Wege eckendisjunkt sind, ist $|M| - |Z| \leq |M|/2$ bzw. $2|Z| \geq |M|$. Hieraus folgt, daß der Wirkungsgrad des angegebenen Algorithmus kleiner oder gleich 2 ist.

Es sei l eine gerade Zahl und G_l ein bipartiter Graph mit Eckenmenge

$$E = \{a_1, \dots, a_l\} \cup \{b_1, \dots, b_l\}$$

und Kantenmenge

$$\begin{aligned} K = & \{(a_i, b_i) \mid i = 1, \dots, l\} \cup \{(a_i, b_{i+1}) \mid i = 1, 3, 5, \dots, l-1\} \\ & \cup \{(a_i, b_{i-1}) \mid i = 3, 5, \dots, l-1\}. \end{aligned}$$

Die Kanten $\{(a_i, b_i) \mid i = 1, \dots, l\}$ bilden eine maximale Zuordnung mit l Kanten. Die Kanten $\{(a_i, b_{i+1}) \mid i = 1, 3, 5, \dots, l-1\}$ bilden eine Zuordnung, die durch die Funktion **greedy-Zuordnung** bestimmt wurden. Somit gilt

$$OPT(G_l)/A(G_l) = l/2l = 2$$

und der asymptotische Wirkungsgrad ist 2.

In der folgenden Implementierung von **greedy-Zuordnung** werden die Nachbarn jeder Ecke genau einmal betrachtet, deshalb ist die worst case Laufzeit gleich $O(n + m)$. Die Endenken der Kanten der Zuordnung können in linearer Zeit aus dem Feld **Zuordnung** entnommen werden.

```

Zuordnung : array[1..max] of Integer;
procedure greedy-Zuordnung(G : Graph);
var
    i,j : Integer;
begin
    Initialisiere Zuordnung mit 0;
    for jede Ecke i do
        if Zuordnung[i] = 0 then
            for jeden Nachbar i von j do
                if Zuordnung[j] = 0 then begin
                    Zuordnung[i] := j; Zuordnung[j] := i;
                    break;
                end
            end
    end
end

```

11. Der Algorithmus aus der letzten Aufgabe hat für das beschriebene Problem die worst-case Laufzeit $O(n + \bar{m})$, wobei \bar{m} die Anzahl der Kanten von \bar{G} ist. Der folgende Algorithmus ist ebenfalls ein Greedy-Algorithmus, d.h. die Aussage über den Wirkungsgrad bleibt gültig. Er verwendet eine verkettete Liste zur effizienten Iteration über die Menge der noch nicht zugeordneten Ecken.

```

Zuordnung : array[1..max] of Integer;
procedure greedy-Zuordnung(G : Graph);
var
    nachbarn : array[1..max] of Boolean;
    unbedeckt Liste;
    i,j : Integer;
begin
    Initialisiere Zuordnung mit 0;
    Initialisiere nachbarn mit false;
    Füge alle Ecken von G in unbedeckt ein;
    while L ≠ ∅ begin
        i = unbedeckt.entreneKopf();
        for jeden Nachbar j von i do
            nachbarn[j] := true;
        for jede Ecke j in L do
            if nachbarn[j] = true begin
                unbedeckt.entrene(j);
                Zuordnung[i] := j; Zuordnung[j] := i;
                break;
            end
        for jeden Nachbar j von i do
            nachbarn[j] := false;
    end
end

```

Für die Analyse der Laufzeit beachte man, daß jede der drei **for**-Schleifen innerhalb der **while**-Schleife den Aufwand $O(g(i))$ hat. Hieraus ergibt sich, daß der Algorithmus lineare Laufzeit $O(n + m)$ hat.

12. Nach Aufruf der Prozedur aus der letzten Aufgabe werden die z Kanten der Zuordnung Z der Reihe nach betrachtet. Gibt es für die Enden einer Kante (e, f) nicht zugeordnete Nachbarn e' und f' mit $e \neq f$, so wird (e, f) aus Z entfernt und die beiden neuen Kanten (e', e) und (f', f) werden in Z eingefügt. Man überzeugt sich schnell, daß nach Betrachtung der z Kanten jeder Erweiterungsweg bezüglich Z aus mindestens fünf Kanten besteht (drei davon gehören nicht zu Z). Die Laufzeit des Verfahrens bleibt weiterhin linear.

Es sei M eine maximale Zuordnung von G . Dann gibt es nach der vorletzten Aufgabe $|M| - |Z|$ eckendisjunkte Erweiterungswege bezüglich Z . Nach Konstruktion enthält jeder dieser Erweiterungswege mindestens drei Kanten aus M . Da die Wege eckendisjunkt sind, ist $|M| - |Z| \leq |M|/3$ bzw. $3|Z| \geq 2|M|$. Somit ist der Wirkungsgrad des Algorithmus kleiner oder gleich $3/2$.

Es sei l eine durch 3 teilbare Zahl und G_l ein bipartiter Graph mit Eckenmenge

$$E = \{a_1, \dots, a_l\} \cup \{b_1, \dots, b_l\}$$

und Kantenmenge

$$\begin{aligned} K = & \{(a_i, b_i) \mid i = 1, \dots, l\} \cup \{(a_i, b_{i+1}) \mid 1 \leq i < l, i \text{ nicht durch 3 teilbar}\} \\ & \cup \{(b_i, a_{i+1}) \mid 1 \leq i < l, i \text{ durch 3 teilbar}\}. \end{aligned}$$

Die Kanten $\{(a_i, b_i) \mid i = 1, \dots, l\}$ bilden eine maximale Zuordnung mit l Kanten. Die Kanten $\{(a_i, b_{i+1}) \mid 1 \leq i < l, i \text{ nicht durch 3 teilbar}\}$ bilden eine Zuordnung, die durch den Algorithmus bestimmt wurde. Somit gilt

$$OPT(G_l)/A(G_l) = l/(2/3)l = 3/2$$

und der asymptotische Wirkungsgrad ist gleich $3/2$.

13. a) Die Ecken einer Eckenüberdeckung bilden auch eine dominierende Menge, aber nicht umgekehrt. Es sei W_n der Windmühlengraph mit n Ecken (Windmühlengraphen werden auf Seite 349 eingeführt). Die zentrale Ecke bildet eine minimale dominierende Menge von Ecken von W_n , eine minimale Eckenüberdeckung hat dagegen $(n - 1)/2$ Ecken.
- b) Jede Ecke aus $E \setminus E'$ ist zu einer Ecke aus E' benachbart. Mit Hilfe der in Abschnitt 2.8 eingeführten Datenstrukturen kann der Algorithmus so implementiert werden, daß er linearen Zeitaufwand hat.
- c) Im folgenden werden die Ecken S_1, \dots, S_r Spaltenecken und die restlichen Ecken $S_{r+1}, \dots, S_{r+l!}$ Zeilenecken genannt. Sowohl die Zeilen- als auch die Spaltenecken bilden unabhängige Mengen, d.h. die Graphen G_l sind bipartit. Es sei U die Menge der ersten $l!$ Spaltenecken und aller Zeilenecken. Der von U induzierte Untergraph besteht aus $l!$ Zusammenhangskomponenten, jede

besteht aus genau einer Kante. Hieraus folgt, daß die Menge der $l!$ Zeilenecken eine minimale dominierende Menge von Ecken für G_l bildet.

Jede Zeilenecke hat den Eckengrad l und die von der j -ten Spalte induzierten Spaltenecken haben den Eckengrad j . Der angegebene Algorithmus wählt zunächst die von der letzten Spalte induzierten Spaltenecken. Danach verbleiben noch die restlichen Spaltenecken als isolierte Ecken. Der Algorithmus bestimmt somit die Menge der r Spaltenecken als dominierende Menge. Der Wirkungsgrad des Algorithmus ist damit mindestens

$$\frac{r}{l!} = \sum_{j=1}^l \frac{1}{j}.$$

Da die harmonische Reihe divergiert, ist der asymptotische Wirkungsgrad unendlich.

14. Für einen vollständig k -partiten Graphen G gilt $\chi(G) = k$. Nachdem der Greedy-Algorithmus der ersten Ecke einer Teilmenge E_i die Farbe f_i gegeben hat, kann diese Farbe an keine Ecke einer anderen Teilmenge E_j mehr vergeben werden. Die restlichen Ecken von E_i werden jedoch unabhängig von ihrer Reihenfolge mit dieser Farbe gefärbt. Somit vergibt der Algorithmus genau $k = \chi(G)$ Farben. Der Greedy-Algorithmus färbt die Graphen C_n mit ungeradem n für jede Reihenfolge der Ecken mit 3 Farben, d.h. mit der minimalen Anzahl von Farben. Die Graphen C_n sind jedoch nicht vollständig k -partit.
15. Der Algorithmus A_2 bestimmt die chromatische Zahl von G_X mittels einer maximalen Zuordnung $|Z|$ des Komplements von G_X . Es gilt $\chi(G_X) = |X| - |Z|$. Verwendet man anstatt Z eine Zuordnung Z' mit mindestens $2/3|Z|$ Kanten, so wird eine Färbung von G_X mit höchstens $|X| - 2/3|Z| = |X|/3 + 2/3\chi(G_X)$ Farben verwendet. Hieraus kann der Wirkungsgrad des neuen Algorithmus A'_2 bestimmt werden:

$$\begin{aligned} A'_2(G) &\geq n - (|X|/3 + 2/3\chi(G_X)) - \frac{(n - |X|)}{3} \\ &= \frac{2}{3}n - \frac{2}{3}\chi(G_X) \\ &\geq \frac{2}{3}(n - \chi(G)) \\ &= \frac{2}{3}OPT(G) \end{aligned}$$

16. In konstanter Zeit kann festgestellt werden, ob es sich um den vollständigen Graphen mit vier Ecken handelt. In diesem Fall ist $\chi(G) = 4$. Andernfalls folgt aus dem Satz von Brooks, daß $\chi(G) \leq 3$ gilt. Mit linearem Aufwand kann geprüft werden, ob ein bipartiter Graph vorliegt, d.h. ob $\chi(G) = 2$ ist. Ist G nicht bipartit, dann besteht G aus genau einer Ecke oder es gilt $\chi(G) = 3$.
17. a) Für $n = 3$ ist die Aussage trivialerweise richtig. Sei nun $n > 3$. Falls G Hamiltonsch ist, so gilt dies auch für G' . Sei nun umgekehrt G' Hamiltonsch

und H ein Hamiltonscher Kreis von G' . Liegt die Kante (e, f) nicht auf H , dann ist auch H ein Hamiltonscher Kreis für G . Andernfalls ergibt sich aus H ein einfacher Weg $e = e_1, e_2, \dots, e_n = f$ in G , auf dem alle Ecken von G liegen. Im folgenden werden die Mengen

$$A = \{e_i \mid (f, e_{i-1}) \text{ ist Kante in } G \text{ und } 3 \leq i \leq n-1\}$$

und

$$B = \{e_i \mid (e, e_i) \text{ ist Kante in } G \text{ und } 3 \leq i \leq n-1\}$$

betrachtet. Es gilt $\{e_1, e_2, e_n\} \cap A = \emptyset$, $\{e_1, e_2, e_n\} \cap B = \emptyset$, $|A| \geq |N(f)| - 1$ und $|B| = |N(e)| - 1$. Nun folgt aus der Voraussetzung:

$$n - 3 \geq |A \cup B| = |A| + |B| - |A \cap B| \geq n - 2 - |A \cap B|$$

Somit ist $|A \cap B| > 0$, d.h. es gibt eine Ecke e_i , so daß (e, e_i) und (f, e_{i-1}) Kanten in G sind. Dann ist $e_1, e_2, \dots, e_{i-1}, e_n, e_{n-1}, \dots, e_i, e_1$ ein Hamiltonscher Kreis in G .

- b) (i) Da vollständige Graphen Hamiltonsch sind, kann die Aussage leicht mittels Teil a) bewiesen werden.
 - (ii) Folgt direkt aus (i).
18. Der Petersen-Graph besteht aus zwei Kopien von C_5 , die Ecken dieser beiden Graphen sind durch fünf *Speichen* verbunden (siehe Abbildung 2.7) auf Seite 23. Man überlegt sich schnell, daß in einem Hamiltonschen Kreis entweder genau zwei oder genau vier Speichen vorkommen müssen. Im ersten Fall müßten dann jeweils vier Kanten jeder Kopie des zyklischen Graphen C_5 auf dem Hamiltonschen Kreis vorkommen. Dies geht jedoch nicht. Im zweiten Fall müßten von der einen Kopie zwei und von der anderen Kopie drei Kanten auf dem Hamiltonschen Kreis liegen. Man sieht sofort, daß auch dies unmöglich ist.
19. a) Ein 1-Baum existiert nur, falls der von den Ecken $2, \dots, n$ induzierte Untergraph G' zusammenhängend und $g(1) \geq 2$ ist. Es sei B ein minimal aufspannender Baum von G' und K_1 die Menge der zu Ecke 1 inzidenten Kanten. Die beiden Kanten aus K_1 mit den geringsten Bewertungen bilden zusammen mit den Kanten aus B einen minimalen 1-Baum. Der Aufwand des Algorithmus hängt vom Aufwand der Bestimmung eines minimal aufspannenden Baumes für G' ab. Hierzu vergleiche man die in Kapitel 3 vorgestellten Algorithmen.
- b) Man beachte, daß W ein 1-Baum ist.
20. Es seien W_1 und W_2 wie in der Aufgabe beschrieben. Gibt es eine Ecke e , welche mehr als einmal auf W_1 vorkommt, dann gibt es Kanten (e_1, e) und (e, e_2) , welche nacheinander auf W_1 vorkommen. Ersetzt man diese beiden Kanten durch die Kante (e_1, e_2) , so folgt aus der Dreiecksungleichung und der Minimalität von W_1 , daß der neu entstandene Weg die gleiche Länge wie W_1 hat. Auf diese Art kann ein geschlossener einfacher Weg W'_1 mit gleicher Länge wie W_1 erzeugt werden, auf dem alle Ecken von G liegen. Hieraus folgt:

$$L(W_2) \geq L(W_1) = L(W'_1) \geq L(W_2)$$

Dies zeigt die Behauptung.

21. Es sei L_i die Summe der Längen der Kanten, welche bis zum i -ten Schritt ausgewählt und in den Weg W_i eingefügt werden. Es sei $W_i = e_1, \dots, e_i, e_1$ der konstruierte Weg. Mittels vollständiger Induktion nach i wird bewiesen, daß $L(W_i) \leq 2L_i$ für $i = 2, \dots, n$ gilt. Für $i = 2$ ist die Aussage klar. Sei nun $i > 2$ und (e_s, f) die neu ausgewählte Kante. Da G die Dreiecksungleichung erfüllt, gilt

$$d(e_{s-1}, f) \leq d(e_{s-1}, e_s) + d(e_s, f)$$

bzw.

$$d(e_{s-1}, f) + d(e_s, f) - d(e_{s-1}, e_s) \leq 2d(e_s, f).$$

Mit Hilfe der Induktionsvoraussetzung folgt nun

$$\begin{aligned} L(W_i) &= L(W_{i-1}) + d(e_{s-1}, f) + d(e_s, f) - d(e_{s-1}, e_s) \\ &\leq 2L_{i-1} + 2d(e_s, f) \\ &= 2L_i. \end{aligned}$$

(Ist $s = 1$, so ersetze man in diesen Ungleichungen e_{s-1} durch e_i). Man beachte, daß die ausgewählten Kanten genau die Kanten sind, welche auch der Algorithmus von Prim auswählt. Somit ist $L(W_n) \leq 2K < 2OPT(G)$, wobei K die Kosten eines minimal aufspannenden Baumes von G sind. Hieraus folgt die Aussage über den Wirkungsgrad. Der Algorithmus hat die gleiche Laufzeit wie der Algorithmus von Prim.

22. Auf W liegen für jede Ecke e genau zwei verschiedene zu e inzidente Kanten. Diese haben zusammen mindestens die Länge $m(e)$. Da jede Kante auf W zu genau zwei Ecken inzident ist, gilt:

$$\frac{1}{2} \sum_{e \in E} m(e) \leq L(W)$$

23. Der dargestellte Algorithmus folgt im Prinzip dem zweiten Teil des Beweises des Satzes über die Charakterisierung von Eulerschen Graphen auf Seite 322. In diesem Beweis werden sukzessive geschlossene Kantenzüge bestimmt und miteinander verschmolzen. Die rekursive Prozedur `euler` verschränkt die Suchen nach geschlossenen Kantenzügen. Ausgehend von einer Startecke wird ein geschlossener Kantenzug bestimmt und die Kanten werden als besucht markiert. Die Eckengrade des verbleibenden Graphen sind weiterhin gerade. Die Suche wird bei der zuletzt besuchten Ecke fortgesetzt, die zu noch nicht markierten Kanten inzident ist. Dies wird durch die Rekursion umgesetzt. Da die Ausgabe der Ecken am Ende der Prozedur erfolgt, wird auch das korrekte Verschmelzen der Kantenzüge garantiert. Zwei nacheinander ausgegebene Ecken sind benachbart. Da G zusammenhängend ist, besucht die Prozedur auf jeden Fall alle Kanten. Der Korrektheitsbeweis unterscheidet zwei Fälle.

Fall 1: Bis auf den letzten Aufruf bewirkt jeder Aufruf von `euler` genau einen weiteren Aufruf der Prozedur. Die Aufrufhierarchie sei `euler(i1)` bis `euler(is)`, wobei `euler(ik)` nur `euler(ik+1)` aufruft. Dann ist $i_1 = i_s$, $s = n+1$ und die Ausgabe lautet: $i_1, i_n, i_{n-1}, \dots, i_1$. Dies ist notwendigerweise ein Eulerscher Kreis.

Fall 2: Es gibt eine Aufruffolge `euler(i0)` bis `euler(is)`, so daß `euler(ik)` für $k = 1, \dots, s-1$ nur `euler(ik+1)` aufruft, `euler(is)` keinen weiteren Aufruf macht und `euler(i1)` ist der letzte, aber nicht der erste Aufruf von `euler(i0)`. Dann ist $i_0 = i_s$ und es wird $i_s, i_{s-1}, \dots, i_1, i_s$ ausgegeben. Dies ist ein geschlossener Kantenzug. Läßt man die Kanten dieses Kreises aus G weg, so ist die Eckengradbedingung immer noch erfüllt. Per Induktion produziert die Prozedur für diesen Graphen einen Eulerschen Kreis, hierbei werden die Kanten in der gleichen Reihenfolge wie im Orginalgraph besucht. Die Zusammenfassung der beiden Kantenzüge ergibt einen Eulerschen Kreis für den Ausgangsgraphen, dieser ist auch identisch mit der Ausgabe des Gesamtverfahrens.

Der entscheidende Punkt bei der Bestimmung des Aufwandes der Prozedur ist die Umsetzung der Markierungen der Kanten und die Überprüfung der Kantenummarkierungen. Kann dies in konstanter Zeit erfolgen, so ist der Gesamtaufwand $O(m)$. Für die Verwaltung der Markierungen wird ein boolsches Feld `Markierung` der Länge m verwendet, dieses Feld wird mit `false` initialisiert. Weiterhin wird die Adjazenzliste erweitert. Für jede Ecke wird eine Liste von Records vom Typ `Eintrag` mit zwei Komponenten angelegt: Eckenummer des Nachbarn und Index der Kante im Feld `Markierung`. Die Initialisierung dieser neuen Adjazenzliste erfolgt mit Hilfe der normalen Adjazenzliste mit Aufwand $O(m)$:

```
i,j,kantenummer : Integer;
kantenummer := 1:
for jede Ecke i do
    for jeden Nachbar i von j do
        if i < j then begin
            nachbarn(i).anhängen(Eintrag(j, kantenummer));
            nachbarn(j).anhängen(Eintrag(i, kantenummer));
            kantenummer := kantenummer + 1;
        end
```

Die Umsetzung der `if`-Anweisung innerhalb der Prozedur `euler` ist nun sehr einfach. Über die Adjazenzliste erhält man die Kantenummer einer Kante und über das Feld `Markierung` erfährt man, ob die Kante schon markiert ist (unabhängig von der Durchlaufrichtung). Die Markierung einer durch die Kantenummer gegebenen Kante erfolgt in konstanter Zeit.

24. Die Funktion `kanten-Färbung` gibt die Anzahl A der vergebenen Farben zurück. Der Aufwand der Funktion ist gleich $O(A(n + m))$.

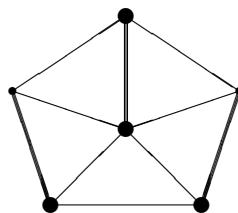
```
function kanten-Färbung(G : Graph) : Integer;
var
    Zuordnung : array[1..max] of Integer;
    i,j,farbe : Integer;
```

```

begin
    farbe := 0;
    Initialisiere Zuordnung mit 0;
    while G enthält noch Kanten do begin
        farbe := farbe + 1;
        for jede Ecke i do
            if Zuordnung[i] < farbe then
                for jeden Nachbar j von i do
                    if Zuordnung[j] < farbe then begin
                        Zuordnung[i] := farbe;
                        Zuordnung[j] := farbe;
                        färbe Kante (i,j) mit Farbe farbe;
                        entferne Kante (i,j);
                        break;
                    end
                end
            kanten-Färbung := farbe;
        end
    end

```

25. Zur Bestimmung des Wirkungsgrades des dargestellten Algorithmus werden die in Abschnitt 9.4 konstruierten Graphen G_r betrachtet. Sie haben eine minimale Eckenüberdeckung mit r Ecken. Die Konstruktion der Graphen bewirkt, daß der Algorithmus der Reihe nach die Ecken aus R_r, R_{r-1}, \dots, R_1 auswählt, d.h. die Menge R wird als Eckenüberdeckung konstruiert. Wie in Abschnitt 9.4 gezeigt, gilt auch für diesen Algorithmus $\mathcal{W}_A(n) = O(\log n)$ und $\mathcal{W}_A^\infty = \infty$. Mit Hilfe der in Abschnitt 2.8 entwickelten Datenstrukturen kann der Algorithmus mit Aufwand $O(n + m)$ umgesetzt werden.
26. Eine minimale Eckenüberdeckung des folgenden Graphen enthält vier Ecken (fett dargestellt), eine maximale Zuordnung enthält aber nur drei Kanten (fett dargestellt).



27. Der Algorithmus verwaltet ein Feld **Überdeckung**, in dem alle zur Überdeckung gehörenden Ecken markiert werden. Die Ecken des Baumes werden gemäß der Tiefensuche besucht. Bei diesem Algorithmus gehören Blätter nicht zur konstruierten Überdeckung. Die Funktion **eckenüberdeckung(i, vorgänger)** zeigt beim Verlassen einer Ecke an, ob die Kante $(i, \text{vorgänger})$ noch überdeckt werden muss. Der Funktionsaufruf **eckenüberdeckung(start, 0)** mit einer beliebigen Ecke **start** bestimmt eine minimale Eckenüberdeckung in linearer Zeit $O(n + m)$.

```

Überdeckung : array[1..max] of Boolean;
function eckenüberdeckung(i : int, int vorgänger) : Boolean;
var

```

```

j : Integer;
begin
  Überdeckung[i] := false;
  for jeden Nachbar j von i do
    if j ≠ vorgänger then do
      Überdeckung[i] := eckenüberdeckung(j,i) or Überdeckung[i]
      eckenüberdeckung = not Überdeckung[i];
  end
end

```

Die Korrektheit des Verfahrens wird durch vollständige Induktion nach n , der Anzahl der Ecken des Baumes B , geführt. Für $n = 2$ erzeugt der Algorithmus offensichtlich eine minimale Eckenüberdeckung mit einer Ecke. Sei nun $n > 2$. Es sei b ein Blatt und v der Vorgänger von b im Tiefensuchebaum. Hat v keinen weiteren Nachfolger, so bestimmt der Algorithmus nach Induktionsvoraussetzung eine minimale Eckenüberdeckung U für den Baum $B \setminus \{v, b\}$. Somit ist $U \cup \{v\}$ eine minimale Eckenüberdeckung von B . Wie man leicht sieht, ist dies aber genau die Eckenüberdeckung, welche der Algorithmus für B produziert. Gibt es in B kein Blatt mit dieser Eigenschaft, so muß es in B ein Blatt b geben, so daß der Vorgänger v ein weiteres Blatt als Nachfolger hat. Nach Induktionsvoraussetzung bestimmt der Algorithmus für den Baum $B \setminus \{b\}$ eine minimale Eckenüberdeckung U . Wie man wieder leicht sieht, ist dies genau die Eckenüberdeckung, welche der Algorithmus für B produziert.

28. Da nach Aufgabe 33 aus Kapitel 4 auf Seite 129 die Blätter eines Tiefensuchebaumes eine unabhängige Menge sind, bildet die Menge der inneren Ecken eine Eckenüberdeckung. Mittels des in Aufgabe 31 in Kapitel 7 beschriebenen Algorithmus kann eine maximale Zuordnung Z des Baumes bestimmt werden. Man überzeugt sich leicht, daß die erste Ecke, die der Algorithmus besucht, durch Z überdeckt wird. Ferner wird auch jede innere Ecke außer der Wurzel durch Z überdeckt. Ist k die Anzahl der Kanten in Z und $e = |E'|$, so gilt $2k \geq e$. Zur Überdeckung der Kanten aus Z werden mindestens k Ecken benötigt. Hieraus folgt die Behauptung.
29. Ist e eine Ecke von G und T eine beliebige Teilmenge der Ecken von G , so wird mit $N_T(e)$ im folgenden die Menge der in T liegenden Nachbarn von e bezeichnet. Es sei e eine Ecke aus X mit $N_X(e) > N_{\bar{X}}(e)$. Wird nun e von X nach \bar{X} verschoben, so erhält der Schnitt (X, \bar{X}) genau $N_{\bar{X}}(e) - N_X(e) > 0$ zusätzliche Kanten. Auch bei der Verschiebung einer Ecke e aus \bar{X} mit $N_{\bar{X}}(e) > N_X(e)$ nach X wird die Anzahl der Kanten im Schnitt erhöht. Da ein Schnitt maximal m Kanten enthält, endet das Verfahren spätestens nach m Schritten. Dann gilt $N_X(e) \leq N_{\bar{X}}(e)$ für alle Ecken e aus X , und $N_{\bar{X}}(e) \leq N_X(e)$ für alle Ecken aus \bar{X} . Nun kann der Wirkungsgrad leicht bestimmt werden. Es ist

$$2A(G) = \sum_{e \in X} N_{\bar{X}}(e) + \sum_{e \in \bar{X}} N_X(e).$$

Hieraus folgt

$$\begin{aligned}
 OPT(G) &\leq m \\
 &= \frac{1}{2} \left(\sum_{e \in X} (N_X(e) + N_{\bar{X}}(e)) + \sum_{e \in \bar{X}} (N_X(e) + N_{\bar{X}}(e)) \right) \\
 &\leq \sum_{e \in X} N_{\bar{X}}(e) + \sum_{e \in \bar{X}} N_X(e) \\
 &= 2A(G).
 \end{aligned}$$

30. Das betrachtete Entscheidungsproblem liegt offenbar in \mathcal{NP} . Es wird ein Graph G mit Eckenmenge $E = X \cup Y \cup Z$ konstruiert. Zwei Ecken $a, b \in E$ sind genau dann benachbart, wenn es kein Element $m \in M$ gibt, welches a und b gleichzeitig als Komponenten enthält. Der Graph G hat $3q$ Ecken und lässt sich in polynomialer Zeit konstruieren. Ist $(x, y, z) \in M$, dann bilden x, y, z in G eine unabhängige Menge. Die von den Mengen X, Y und Z induzierten Untergraphen sind vollständig, somit ist $\chi(G) \geq q$. Es sei A eine vier-elementige Teilmenge von E . Dann muß es in A zwei Elemente geben, welche in der gleichen der drei Cliquen liegen. Somit ist $\alpha(G) \leq 3$. Im folgenden wird bewiesen, daß M genau dann eine 3-dimensionale Zuordnung besitzt, wenn $\chi(G) = q$ gilt. Hieraus folgt unmittelbar die in der Aufgabe gemachte Aussage. Ist $\chi(G) = q$, so besteht G aus genau q Farbklassen mit je drei Elementen. Es sei $\{a, b, c\}$ mit $a \in X, b \in Y$ und $c \in Z$ eine solche Farbklasse. Nach Konstruktion von G gibt es $x \in X, y \in Y$ und $z \in Z$, so daß $(x, b, c), (a, y, c), (a, b, z)$ in M liegen. Wegen der paarweisen Konsistenz muss dann auch (a, b, c) in M liegen. Hieraus folgt, daß die aus den Farbklassen gebildeten Elemente aus M eine 3-dimensionale Zuordnung bilden. Umgekehrt gilt, daß jede 3-dimensionale Zuordnung auf G eine disjunkte Überdeckung mit 3-elementigen unabhängigen Mengen induziert, d.h. es gilt $\chi(G) = q$.
31. Der erste Teil des Algorithmus entspricht der Funktion **greedy-Zuordnung** aus Aufgabe 10. Besteht die erzeugte Zuordnung Z aus m' Kanten, so gilt $|I_1| = n - 2m'$. Wegen $m' < n/2$ gibt es eine Zahl $\gamma \in [0, 1]$ mit $m' = n(1 - \gamma)/2$ bzw. $|I_1| = n\gamma$. Nun bildet $|Z| + |I_1|$ eine obere Abschätzung für $\alpha(G)$, hieraus folgt:

$$\alpha(G) \leq \frac{n}{2}(1 + \gamma) \quad (\text{B.1})$$

Nun wird der zweite Teil des Algorithmus betrachtet. Im ersten Durchlauf der **while**-Schleife werden $\delta + 1$ Ecken entfernt. In jedem weiteren Schritt werden höchstens Δ Ecken aus E entfernt. Dazu beachte man, daß es wegen des Zusammenhangs von G mindestens eine Ecke im Restgraphen geben muss, der Grad kleiner oder gleich $\Delta - 1$ ist. Somit gilt

$$|I_2| \geq \frac{n - (\delta + 1)}{\Delta} + 1. \quad (\text{B.2})$$

Aus Gleichung (B.1) und $|I_1| = n\gamma$ folgt

$$\frac{\alpha(G)}{|I_1|} \leq \frac{\gamma + 1}{2\gamma}. \quad (\text{B.3})$$

Aus Gleichung (B.1) und (B.2) folgt

$$\frac{\alpha(G)}{|I_2|} \leq \frac{\frac{n}{2}(1+\gamma)}{\frac{n-(\delta-1)+\Delta}{\Delta}} = \frac{\Delta n(1+\gamma)}{2(n-(\delta+1)+\Delta)} \leq \frac{\Delta n(1+\gamma)}{2(n-1)}. \quad (\text{B.4})$$

Für die folgende Analyse beachte man, daß für $\gamma \in [0, 1]$ die Funktion $(\gamma+1)/2\gamma$ monoton fallend und die Funktion $\Delta n(1+\gamma)/2(n-1)$ monoton steigend ist. Für die Bestimmung des Wirkungsgrades des Algorithmus ist der Schnittpunkt $S = (n-1)/\Delta n$ der beiden Funktionen von Bedeutung. Für $\gamma \leq S$ ist $|I_2| \geq |I_1|$ und für $\gamma \geq S$ ist $|I_1| \geq |I_2|$. Aus den Gleichungen (B.3) bzw. (B.4) ergibt sich die angegebene Schranke für den Wirkungsgrad.

32. Es sei d_i der Eckengrad der Ecke im Restgraphen, welche in Durchlauf i entfernt wurde. Dann gilt folgende Beziehung zwischen n und den d_i :

$$n = \sum_{i=1}^{|I_2|} (d_i + 1)$$

Da jedesmal die Ecke mit dem kleinsten Eckengrad ausgewählt wird, haben im i -ten Schritt alle entfernten Ecken mindestens den Eckengrad d_i . Somit werden im i -ten Schritt mindestens $d_i(d_i + 1)/2$ Kanten entfernt. Hieraus folgt:

$$\frac{\bar{d}n}{2} = |E| \geq \sum_{i=1}^{|I_2|} d_i(d_i + 1)/2$$

Addiert man das zweifache der letzten Gleichung zu der ersten Gleichung, so erhält man mittels der Cauchy-Schwartz Ungleichung und der ersten Gleichung:

$$(\bar{d} + 1)n \geq \sum_{i=1}^{|I_2|} (d_i + 1)^2 \geq \frac{1}{|I_2|} \left(\sum_{i=1}^{|I_2|} (d_i + 1) \right)^2 \geq \frac{n^2}{|I_2|}$$

Hieraus ergibt sich sofort die Behauptung.

33. Der Beweis erfolgt mittels vollständiger Induktion nach der Anzahl der Ecken des Graphen. Für $n = 1$ ist die Behauptung trivial. Es sei nun G ein Graph mit $n > 1$ Ecken. Die Funktion **unabhängigeMenge** bestimmt eine unabhängige Menge U mit $f(n)$ Ecken. Es sei $G' = G \setminus U$. Nach Induktionsvoraussetzung erzeugt **färbung** für G' eine Färbung mit höchstens

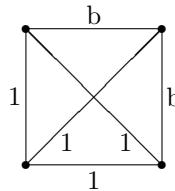
$$\sum_{i=1}^{n-f(n)} \frac{1}{f(i)}$$

Farben und für G wird genau eine zusätzliche Farbe benötigt. Da f monoton steigend ist, gilt

$$\sum_{i=n-f(n)+1}^n \frac{1}{f(i)} \geq \sum_{i=n-f(n)+1}^n \frac{1}{f(n)} = 1.$$

Hieraus folgt die Behauptung.

34. Erfüllen die Bewertungen der Kanten die Dreiecksungleichung, so gilt $L \leq 2K$. Im allgemeinen gibt es aber keine solche Konstante, wie der der folgende Graph zeigt, hierbei ist L von b abhängig und K von b unabhängig.



35. Ein polynomialer Algorithmus für das Optimierungsproblem löst natürlich auch das zugehörige Entscheidungsproblem. Es sei nun A ein polynomialer Algorithmus für das Entscheidungsproblem. In einer ersten Phase werden die Kosten C für eine optimale Lösung berechnet, die Bestimmung der optimalen Rundreise erfolgt dann in der zweiten Phase. Ist l die Länge der Kodierung einer Instanz des Problems des Handlungsreisenden, so liegt C zwischen 0 und 2^l . Mittels binärer Suche in Kombination mit Algorithmus A kann C in höchstens n Schritten bestimmt werden. In der zweiten Phase wird Algorithmus A für jede Kante einmal aufgerufen. Hierzu werden die Kosten der betrachteten Kante auf $C + 1$ gesetzt. Ergibt die Anwendung von A , daß dieser Graph keinen Hamiltonschen Kreis mit Kosten C besitzt, so wird die Bewertung der Kante wieder auf den ursprünglichen Wert zurückgesetzt, andernfalls wird die Kante entfernt. Am Ende dieser Phase bilden die verbliebenen Kanten einen Hamiltonschen Kreis mit Kosten C und das Optimierungsproblem ist gelöst. Insgesamt wurde Algorithmus A höchstens $n + m$ -mal aufgerufen, d.h. das Verfahren hat polynomialen Aufwand.
36. Das betrachtete Entscheidungsproblem liegt offenbar in \mathcal{NP} . Für einen gegebenen zusammenhängenden Graphen G wird wie im Hinweis zur Aufgabe ein neuer Graphen G' konstruiert. Es sei

$$S = \{x_{ef} \mid (e, f) \text{ ist Kante von } G\} \cup \{e_n\}.$$

Es sei s_{stei} das Gewicht eines minimalen Steiner Baums von G' für S und s_{deck} die Anzahl der Ecken in einer minimalen Eckenüberdeckung für G . Im folgenden wird bewiesen, daß $s_{stei} = s_{deck} + m$ gilt. Hieraus folgt unmittelbar die in der Aufgabe gemachte Aussage.

Es sei M eine minimale Eckenüberdeckung von G . Dann ist der von $S \cup M$ induzierte Untergraph U von G' zusammenhängend. Hierzu beachte man, daß e_n zu jeder Ecke aus M benachbart ist und jede Ecke aus $S \setminus \{e_n\}$ zu einer Ecke aus M benachbart sein muß (M ist eine Eckenüberdeckung). Da alle Kanten die Bewertung 1 haben, hat ein aufspannender Baum von U das Gewicht $|S \cup M| = s_{deck} + m$. Hieraus folgt $s_{stei} \leq s_{deck} + m$. Sei nun T ein minimaler Steiner Baum von G' für S und S' die Menge der Steiner Ecken von T . Nach Konstruktion von G' bildet S' eine Eckenüberdeckung von G : Ist $k = (e, f)$ eine Kante von G , so gibt es in T einen Weg von e_n nach x_{ef} , dieser verwendet eine der Ecken e und f , d.h. e oder f liegt in S' . Hieraus folgt:

$$s_{deck} \leq |S'| = s_{stei} + 1 - |S| = s_{stei} - m$$

37. Der konstruierte Baum T^* ist sicherlich ein Steiner Baum für S . Es gilt:

$$\text{kosten}(T^*) \leq \text{kosten}(T'') \leq \text{kosten}(G'') \leq \text{kosten}(T')$$

Es sei T_{OPT} ein minimaler Steiner Baum für S . Mit Hilfe der Tiefensuche, angewendet auf T_{OPT} , kann ein geschlossener Kantenzug W in G erzeugt werden, der jede Kante von T_{OPT} genau zweimal enthält. Dazu wird sowohl beim Erreichen als auch beim Verlassen einer Ecke die entsprechende Kante in W eingefügt. Somit gilt $\text{kosten}(W) = 2 \cdot \text{kosten}(T_{OPT})$. Es sei $s_1 \in S$ ein Blatt von T_{OPT} . Nun durchläuft man W beginnend bei s_1 . Dabei trifft man auf die restlichen Ecken von S . Bezeichne mit $s_2, \dots, s_{|S|}$ die restlichen Ecken von S in der Reihenfolge, in der ihnen zum ersten Mal auf W begegnet wird. Da G' vollständig ist, bilden die Kanten $\{(s_i, s_{i+1}) \mid 1 \leq i \leq |S|-1\}$ einen aufspannenden Baum B für G' . Aus der Dreiecksgleichung für G' folgt $\text{kosten}(B) \leq \text{kosten}(W)$. Da T' ein minimal aufspannender Baum für G' ist, gilt $\text{kosten}(T') \leq \text{kosten}(B)$. Zusammenfassend erhält man

$$\text{kosten}(T^*) \leq \text{kosten}(T') \leq \text{kosten}(B) \leq \text{kosten}(W) = 2 \cdot \text{kosten}(T_{OPT}).$$

Mit Hilfe der Algorithmen von Dijkstra und Prim wird eine Laufzeit von $O(|S|n^2)$ erzielt.

38. Der Algorithmus bestimmt den links dargestellten minimal aufspannenden Baum T' von G' . Da alle Kanten von T' auch Kanten von G sind, ist $G'' = T' = T''$. Der resultierende Steinerbaum T^* mit Gewicht 9 ist rechts dargestellt. Ein minimaler Steinerbaum hat Gewicht 8.



39. Ein polynomialer Algorithmus für das Optimierungsproblem löst natürlich auch das zugehörige Entscheidungsproblem. Es sei

```
function eckenüberdeckung(G : Graph, wert : Integer) : Boolean;
```

eine Funktion, welche mit polynomialem Aufwand entscheidet, ob G eine Eckenüberdeckung mit $wert$ Ecken hat. Mit maximal OPT Aufrufen dieser Funktion kann die Größe OPT der kleinsten Eckenüberdeckung von G bestimmt werden. Es sei nun (e, f) eine Kante des Graphen G . Dann liegt e oder f in einer minimalen Eckenüberdeckung. Es sei U_e bzw. U_f eine minimale Eckenüberdeckung des Graphen, der sich aus G ergibt, wenn alle zu e bzw. zu f inzidenten Kanten entfernt werden. Ist $|U_e| = OPT - 1$, so ist $U_e \cup \{e\}$ eine minimale Eckenüberdeckung von G , andernfalls $U_f \cup \{f\}$. Der beschriebene Algorithmus wird durch den Aufruf `minüberdeckung(G, OPT)` der folgenden rekursiven Funktion umgesetzt:

```
function minüberdeckung(G : Graph, wert Integer) : set of Kanten;
begin
    if G hat keine Kanten then
        minüberdeckung := ∅;
    else begin
        wähle beliebige Kante (e,f) aus G;
         $G_e := G$  ohne die zu e inzidenten Kanten;
        if eckenüberdeckung( $G_e$ , wert-1) then
            minüberdeckung := {e}  $\cup$  minüberdeckung( $G_e$ , wert-1);
        else begin
             $G_f := G$  ohne die zu f inzidenten Kanten;
            minüberdeckung := {f}  $\cup$  minüberdeckung( $G_f$ , wert-1);
        end
    end
end
```

Insgesamt wird diese Funktion höchstens n -mal aufgerufen. Damit wird auch die Funktion eckenüberdeckung höchstens n -mal aufgerufen, d.h., es liegt ein Algorithmus mit polynomialem Aufwand vor.

Literaturverzeichnis

- [1] Aho, A.V., Hopcroft, J.E. und Ullman, J.D., “Data Structures and Algorithms”, *Addison-Wesley*, 1983.
- [2] Aho, A.V., Sethi, R. und Ullman, J.D., “Compilerbau”, *Addison-Wesley*, 1988.
- [3] Ahuja, R.K., Mehlhorn, K., Orlin J.B. und Tarjan, R.E., “Faster Algorithms for the Shortest Path Problem”, *Technical Report CS-TR-154-88* Dept. of Comp. Science, Princeton Univ., 1988.
- [4] Ahuja, R.K., Magnanti, T.L. und Orlin J.B., “Network flows: theory, algorithms and applications”, *Prentice Hall, Englewood Cliffs, N.J.*, 1993.
- [5] Alt, H., Blum, N., Mehlhorn, K. und Paul, M., “Computing a maximum cardinality matching in a bipartite graph in time $O(n^{1.5} \sqrt{m/\log n})$ ”, *Information Processing Letters 37*, 237–240, 1991.
- [6] André, P. und Royer, J.-C., “Optimizing method search with lookup caches and incremental coloring”, *OOPSLA Conference Proceedings '92*, 110–126, 1992.
- [7] Appel, K. und Haken, W., “Every planar map is four colorable”, *Bulletin of the American Mathematical Society*, Vol. 82, 711–712, 1976.
- [8] Appel, K. und Haken, W., “Every planar map is four colorable”, *Illinois Journal of Mathematics*, Vol. 21, 421–567, 1977.
- [9] Arora, S. und Safra, S., “Probabilistic checking of proofs”, *Proc. 33rd IEEE Symp. on the Foundations of Computer Science*, 2–13, 1992.
- [10] Arora, S., Lund, C., Motwani, R., Sudan, M. und Szegedy, M., “Proof verification and hardness of approximation problems”, *Proc. 33rd IEEE Symp. on the Foundations of Computer Science*, 14–23, 1992.
- [11] Bell, T., Cleary, J. und Witten, I., “Text Compression”, *Prentice Hall*, 1990.
- [12] Bellman, R.E., “On a routing problem”, *Quart. Appl. Math. 16*, 87–90, 1958.
- [13] Berger, B. und Rompel, J., “A Better Performance Guarantee for Approximate Graph Coloring”, *Algorithmica 5*, 459–466, 1990.
- [14] Blum, A., “New Approximation Algorithms for Graph Coloring”, *Journal of the ACM*, 470–516, 1994.
- [15] Braers, D., “Die Bestimmung kürzester Pfade in Graphen und passende Datenstrukturen”, *Computing 8*, 171–181, 1971.

- [16] Brin, S. und Page, L., "The Anatomy of a Large-Scale Hypertextual Web Search Engine", Proc. 7th Int'l World Wide Web Conf., *ACM Press*, New York, 107–117, 1998.
- [17] Brooks, R.L., "On Colouring the Nodes of a Network", *Proc. Cambridge Phil. Soc.*, Vol. 37, 194–197, 1941.
- [18] Buckley, C., "Path-Planning Methods for Robot Motion", in: Rembold, U., *Robot Technology and Applications*, Marcel Dekker, 1990.
- [19] Chaitin, G.J., "Register allocation und spilling via graph colouring", *ACM SIGPLAN Notices* 17:G, 201–207, 1982.
- [20] Cherkassky, B.V., Goldberg, A.V. und Radzik, T., "Shortest Paths Algorithms: Theory and Experimental Evaluation", *Stanford University*, Technical Report, 1993.
- [21] Cherkassky, B.V. und Goldberg, A.V., "On implementing push-relabel method for the maximum flow problem", *Stanford University*, Technical Report, 1993.
- [22] Chiba, N., Nishizeki, T. und Saito, N., "A linear Algorithm for five-coloring a planar Graph", *Lecture Notes in Computer Science*, 108, *Graph Theory and Algorithms*, Springer Verlag, 9–19, 1980.
- [23] Christofides, N., "Worst-case analysis of a new heuristic for the traveling salesman problem", *Technical Report*, Graduate School of Industrial Administration, Carnegie-Mellon University, 1976.
- [24] Cormen T.H., Leiserson C.E. und Rivets R.L., "Introduction to Algorithms", *MIT Press*, 1990.
- [25] Cornuejols, G. und Nemhauser, G.L., "Tight Bounds for Christofides' Traveling Salesman Heuristic", *Mathematical Programming* 14, 116–121, 1978.
- [26] Culberson, J.C., "Iterated Greedy Graph Coloring und the Difficulty Landscape", *Technical Report TR 92-07*, University of Alberta, Canada, June 1992 (Zu beziehen über <http://web.cs.ualberta.ca/~joe/>).
- [27] Dantzig, G.B. und Fulkerson, D.R., "On the Max-Flow Min-Cut Theorem of Networks", In: *Linear Inequalities and Related Systems*, Annals of Math. Study 38, Princeton University Press, 1956, 215–221.
- [28] Dial, R.B., "Algorithm 360: Shortest Path Forest with Topological Ordering", *Comm. ACM* 12, 632–633, 1969.
- [29] Dijkstra, E.W., "A note on two problems in connexion with graphs", *Numerische Mathematik* 1, 269–271, 1959.
- [30] Center for Discrete Mathematics and Theoretical Computer Science, "Clique and coloring Problems: A brief introduction with project ideas", *Technical Report*, New Brunswick, 1992 (Zu beziehen per ftp von [dimacs.rutgers.edu](ftp://dimacs.rutgers.edu)).

- [31] Dinic, E.A., "Algorithms for solution of a problem of maximum flow in networks with power estimation", *Soviet Math. Dokl.* 11, 1277–1280, 1970.
- [32] Domschke, W., "Logistik: Rundreisen und Touren", *Oldenbourg Verlag*, 1989.
- [33] Domschke, W., "Logistik: Transport", *Oldenbourg Verlag*, 1989.
- [34] Domschke, W., "Zwei Verfahren zur Suche negativer Zyklen in bewerteten Digraphen", *Computing* 11, 125–136, 1973.
- [35] Edmonds, J. und Karp, R.M., "Theoretical improvements in algorithmic efficiency for network flow problems", *Journal of the ACM* 19 (2), 248–264, 1972.
- [36] Even, S., Pnueli, A. und Lempel, A., "Permutation Graphs and Transitive Graphs", *Journal of the ACM*, 400–410, 1972.
- [37] Even, S. und Tarjan, R.E., "Network Flow and Testing Graph Connectivity", *SIAM Journal on Computing* 4, 507–518, 1975.
- [38] Even, S., "Algorithm for Determining whether the Connectivity of a Graph ist at least k", *SIAM Journal on Computing* 4, 393–396, 1977.
- [39] Florian, M. und Robert, P., "A direct search method to locate negative cycles in a graph", *Manag. Sci.* 17, 307–310, 1971.
- [40] Floyd, R.W., "Algorithm 245: treesort 3", *Comm. ACM* 7:12, 701, 1964.
- [41] Floyd, R.W., "Algorithm 97: shortest path", *Comm. ACM* 5:6, 345, 1962.
- [42] Ford, L.R. und Fulkerson, D.R., "Maximal flow through a network", *Canad. J. Math.* 8, 399–404, 1956.
- [43] Ford, L.R., "Network Flow Theory", *Rand Corporation, Santa Monica, Calif.*, 293, 1956
- [44] Fredman, M. und Tarjan, R.E., "Fibonacci heaps and their use in improved network optimization algorithms", *Journal of the ACM* 34 (3), 596–615, 1987.
- [45] Fredrickson, G.N., "Fast algorithms for shortest paths in planar graphs, with applications", *SIAM Journal on Computing* 16, 1004–1022, 1987.
- [46] Garey, M.L. und Johnson, D.S., "Computers and Intractability — A guide to the theory of NP-Completeness", *W.H. Freeman And Company*, 1979.
- [47] Gilmore, P. und Hoffman, A., "A Characterization of Comparability Graphs and of Interval Graphs", *Canad. J. Math.* 16, 539–548, 1964.
- [48] Goldberg, A.V., "A new max-flow algorithm", *Tech. Memorandum MIT/LCS/TM-291*, Lab. for Comp. Science, MIT, 1985.
- [49] Goldberg, A.V. und Tarjan, R.E., "A New Approach to the Maximum Flow Problem", *Journal of the ACM* 35, 921–940, 1988.

- [50] Goldberg, A.V., Tardos, E. und Tarjan, R.E., "Network flow algorithms", *Technical Report STAN-CS-89-1252*, Dept. of Comp. Science, Stanford Univ., 1989.
- [51] Goldberg, A.V. und Radzik, T., "A Heuristic Improvement of the Bellman-Ford Algorithm", *Applied Math. Let.* 6, 3–6, 1993.
- [52] Goldreich, O., Micali, S. und Wigderson A., "Proofs that yield nothing but their validity, and a methodology of cryptographic protocol design", *Proc. 27th IEEE Symp. on the Foundations of Computer Science*, 174–187, 1986.
- [53] Grimmett, G.R. und McDiarmid, C.J.H., "On coloring random graphs", *Math. Proc. Cambridge Philos. Soc.* 77, 313–324, 1975.
- [54] Grötschel, M. und Lovász, L., "Combinatorial Optimization: A Survey", *DIMACS Technical Report 93-29*, Dept. of Comp. Science, Princeton Univ., 1993.
- [55] Hall, P., "On Representatives of Subsets", *J. London Math. Soc.*, Vol 10, 1935, 26–30.
- [56] Halldórsson, M.M., "A still better performance guarantee for approximate graph coloring", *Information Processing Letters* 45, 19–23, 1993.
- [57] Halldórsson, M.M., "Approximating Set Cover via Local Improvements", *JAIST Technical Report IS-RR-95-0002F*, 1995.
- [58] Harary, F., Hedetniemi, S. und Robinson, R.W., "Uniquely colorable graphs", *Journal Combinatorial Theory* 6, 264–270, 1969.
- [59] Harary, F., "Graphentheorie", *Oldenbourg Verlag*, 1974.
- [60] Hart, P., Nilsson, N. und Raphael, B., "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", *IEEE Trans. Sys. Sci. Cybern.* 2, 100–107, 1968.
- [61] Hassin, R. und Lahav, S., "Maximizing the number of unused colors in the vertex coloring problem", *Information Processing Letters* 52, 87–90, 1989.
- [62] Hayes, B., "Graph Theory in Practice: Part I", *American Scientist* 88(1), 9–13, 2000.
- [63] Hayes, B., "Graph Theory in Practice: Part II", *American Scientist* 88(2), 104–109, 2000.
- [64] Heesch, H., "Untersuchungen zum Vierfarbenproblem", *Hochschulskriptum 810/a/b*, Bibliographisches Institut, Mannheim, 1969.
- [65] Hochbaum, D.S., "Approximation Algorithms for NP-Hard Problems", *PWS Publishing Company*, 1997.
- [66] Holyer, I., "The NP-completeness of edge coloring", *SIAM Journal on Computing* 10, 718–720, 1981.

- [67] Hopcroft, J.E. und Tarjan, R.E., "Efficient planarity testing", *Journal of the ACM* 21(4), 549–568, 1974.
- [68] Hopcroft, J.E. und Karp, R.M., "An $n^{5/2}$ Algorithm for Maximum Matching in Bipartite Graphs", *SIAM Journal on Computing* 2:4, 225–231, 1973.
- [69] Hopcroft, J.E. und Tarjan, R.E., "Dividing a graph into triconnected components", *SIAM Journal on Computing* 2, 135–158, 1973.
- [70] Huffman, D.A., "A method for the construction of minimum-redundancy codes", *Proc. IRE* 40, 1098–1101, 1952.
- [71] Jensen, T.R. und Toft, B., "Graph Coloring Problems", *Wiley Interscience*, 1995.
- [72] Johnson, D.S., "Worst-case behaviour of graph-colouring algorithms", *Proc. 5th South-Eastern Conf. on Combinatorics, Graph Theory and Computing, Utilitas Mathematica Publishing*, Winnipeg, 513–527, 1974.
- [73] Johnson, D.S., "Approximation Algorithms for Combinatorial Problems", *Journal of Computer and System Sciences* 9, 256–278, 1974.
- [74] Johnson, D.S. und McGeoch, C.C., "Network Flows and Matching: First DIMACS Implementation Challenge", *AMS*, 1993.
- [75] Johnson, D.S., "The tale of the second prover", *Journal of Algorithms* 13, 502–524, 1992.
- [76] Jungnickel, D., "Graphen, Netzwerke und Algorithmen", *Wissenschaftsverlag, Mannheim*, 1994.
- [77] Kou, L., Markowsky, G. und Berman, L., "A fast algorithm for Steiner trees," *Acta Informatica*, Vol. 15, 141–145, 1981.
- [78] Knuth, D.E., "The Art of Computer Programming Vol. III: Sorting and Searching", *Addison-Wesley, Reading, Mass.*, 1973.
- [79] Knuth, D.E., "The Stanford GraphBase", *Addison-Wesley, Reading, Mass.*, 1993.
- [80] Kohlas, J., "Zuverlässigkeit und Verfügbarkeit", *Teubner Studienbücher*, 1987.
- [81] Korf, R.E., "Depth-first iterative-deepening: An optimal admissible tree search", *Artificial Intelligence*, Vol. 27, No. 1, 97–109, 1985.
- [82] Korf, R.E., "Optimal Path-Finding Algorithms", in: Search in Artificial Intelligence, Editoren: Kumal, L. und Kumar, V., *Springer Verlag*, 1988.
- [83] Kozen, D.C., "The Design and Analysis of Algorithms", *Springer Verlag*, 1992.
- [84] Kruskal, J., "On the shortest spanning subtree of a graph and the traveling salesman problem", *Proc. AMS* 7:1, 48–50, 1956.
- [85] Kubale, M. und Jackowski, B., "A generalized implicit enumeration algorithm for graph coloring", *Comm. ACM* 28, 412–418, 1985.

- [86] Kučera, L., "The Greedy Coloring is a Bad Probabilistic Algorithm", *Journal of Algorithms* 12, 674–684, 1991.
- [87] Lawler, E.L., Lenstra, J.K., Rinnoy Kan, A.H.G. und Shmoys, D.B., "The Traveling Salesman Problem", *John Wiley & Sons*, 1985.
- [88] van Leeuwen, J., "Graph Algorithms", in: *Handbook of Theoretical Computer Science, Volume A*, Elsevier, 1990.
- [89] Lin, S. und Kernighan, B.W., "An effective heuristic for the traveling salesman problem", *Operations Research* 21, 498–516, 1973.
- [90] Lund, C. und Yannakakis, M., "On the hardness of approximating minimization problems", *Proc. of the 25th Annual ACM Symposium on Theory of Computing*, 286–293, 1993.
- [91] Malhotra, V.M., Pramodh Kumar, M. und Mahaswari, S.N., "An $O(|V|^3)$ algorithm for finding maximum flows in networks", *Information Processing Letters* 7, 277–278, 1978.
- [92] Manber, U., "Introduction to Algorithms", Addison-Wesley, 1989.
- [93] Manzini, G., "BIDA*: an improved perimeter search algorithm", *Artificial Intelligence* 75, 347–360, 1995.
- [94] Mehlhorn, K., "Data Structures and Algorithms", Volume 1–3, Springer-Verlag, 1984.
- [95] Mycielski, J., "Sur le coloriage des Graphes", *Colloq. Math.* 3, 161–162, 1955.
- [96] Menger, V., "Zur allgemeinen Kurventheorie", *Fund. Math.* 10, 26–30, 1927.
- [97] Micali, S. und Vazirani, V.V., "An $O(\sqrt{|V||E|})$ Algorithm for Finding Matching in General Graphs", *Proc. 21st Ann. IEEE Symp. Foundations of Computing Science*, Syracuse, 17–27, 1980.
- [98] Moore, E.F., "The Shortest Path through a Maze", *Proc. Int. Symp. on Theory of Switching, Part II*, 285–292, 1959.
- [99] Munro, J.I., "Efficient determination of the transitive closure of a directed graph", *Information Processing Letters* 1, 56–58, 1971.
- [100] Nuutila, E. und Soisalon-Soininen, E., "On finding the strongly connected components in a directed graph", *Information Processing Letters* 49, 9–14, 1994.
- [101] Ottmann, T. und Widmayer, P., "Algorithmen und Datenstrukturen", Wissenschaftsverlag, 1993.
- [102] Page, L., Brin, S., Motwani, R. und Winograd, T., "The PageRank Citation Ranking: Bringing Order to the Web", Stanford Digital Library Technologies Project, 1998.

- [103] Papadimitriou, C.H., "Computational Complexity", *Addison-Wesley*, 1994.
- [104] Paschos, V.T. "A $(\Delta/2)$ -approximation algorithm for the maximum independent set problem", *Information Processing Letters* 44, 11–13, 1992.
- [105] Pnueli, A., Lempel, A. und Even, S., "Transitive Orientation of Graphs and Identification of Permutation graphs", *Canad. J. Math.*, 160–175, 1971.
- [106] Prim, R., "Shortest connection networks and some generalization", *Bell System Technical J.* 36, 1389–1401, 1957.
- [107] Reinelt, G., "The Traveling Salesman", *Lecture Notes in Computer Science* 840, Springer Verlag, 1994.
- [108] Rhee, C., Liang, Y.L., Dhall, S.K. und Lakshmivarahan S., "Efficient algorithms for finding depth-first and breadth-first search trees in permutation graphs", *Information Processing Letters* 49, 45–50, 1994.
- [109] Robertson, N., Sanders, D., Seymour, P. und Thomas, R., "A new proof of the four-colour theorem", *Electronic Research Announcements Of The AMS*, Vol. 2, Number 1, 1996.
- [110] Rosenkrantz, D.J., Stearns, R.E. und Lewis, P.M., "An analysis of several heuristics for the traveling salesman problem", *SIAM Journal Computing* 6, 563–581, 1977.
- [111] Roy, B., "Transitivité et connexité", *C.R. Acad. Sci. Paris* 249, 216–218, 1959.
- [112] Saaty, T.L. und Kainen, P.C., "The four-color problem, Assaults and Conquest", *Dover Publications*, 1986.
- [113] Savage, C., "Depth-First Search and the Vertex Cover Problem", *Information Processing Letters* 14, 233–235, 1982.
- [114] Simon, K., "Effiziente Algorithmen für perfekte Graphen", *B.G. Teubner*, 1992.
- [115] Spirakis, P. und Tsakalidis, A., "A very fast, practical algorithm for finding a negative cycle in a digraph", *Proc. of 13th ICALP*, 397–406, 1986.
- [116] Stoer, M. und Wagner F., "A Simple Min Cut Algorithm", *Algorithms, Proc. Sec. Annual European Symposium*, Springer Lecture Notes in Comp. Science 855, 141–147, 1994.
- [117] Tarjan, R.E., "Depth first search and linear graph algorithms", *SIAM Journal Computing* 1, 146–160, 1972.
- [118] Thomas, R., "An Update on the Four-Color Theorem", *Notices of the American Mathematical Society*, Volume 45, Number 7, August 1998.
- [119] Turner, J.S., "Almost all k-colorable graphs are easy to color", *Journal of Algorithms*, 9, 63–82, 1988.

- [120] Vizing, V.G., “Über die Abschätzung der chromatischen Klasse eines p-Graphen”, *Diskret. Analiz.* 3, 25–30, 1964.
- [121] Warshall, S.A., “A theorem in Boolean matrices”, *Journal of the ACM* 9, 11–12, 1962.
- [122] Whitney, M., “Congruent graphs and the connectivity of graphs”, *Amer. J. Math.* 54, 150–168, 1932.
- [123] Wilf, H.S., “Backtrack: An O(1) expected time algorithm for the graph coloring problem”, *Information Processing Letters* 18, 119–121, 1984.
- [124] Williams, J., “Algorithm 232 : Heapsort”, *Comm. ACM* 7:6, 347–348, 1964.
- [125] Winter, P., “Steiner tree problem in networks: a survey”, *Networks* 17, 129–167, 1987.
- [126] Wirth, N., “Algorithmen und Datenstrukturen”, *Teubner Studienbücher*, 1975.
- [127] Zelikovski, A.Z., “An 11/6-approximation algorithm for the network Steiner problem”, *Algorithmica* 9, 463–470, 1993.

Index

Symbolen

- I_n , 238
- $N(e)$, 19
- $OPT(a)$, 296
- $W^e(a, b)$, 215
- $W^k(a, b)$, 223
- $Z^e(a, b)$, 214
- $Z^k(G)$, 222
- $\Delta\text{-TSP}$, 319, 332
- $\Delta(G)$, 19
- $\alpha(G)$, 137
- \mathcal{NP} (non-deterministic polynomial), 291
- \mathcal{NP} -complete, 292
- \mathcal{NP} -vollständig, 292
- \mathcal{NPC} , 292
- \mathcal{P} , 291
- $\chi'(G)$, 160
- $\chi(G)$, 132
- $\chi_n(G)$, 163
- $\delta(G)$, 19
- $\kappa(X, \overline{X})$, 168
- κ_o , 209
- κ_u , 209
- $\kappa(k)$, 166
- $\omega(G)$, 133
- \propto , 292
- $a \not\sim b$, 214
- $d(e, f)$, 20
- f_Δ , 170
- $g(e)$, 19
- \mathcal{W}_A , 299
- $\mathcal{W}_A(n)$, 299
- \mathcal{W}_A^∞ , 299
- $\mathcal{W}_{MIN}(P)$, 303
- 0-1-Netzwerk, 190, 202, 205, 292
- 1-Baum, 331
 - minimaler, 331
- 8-Zusammenhang, 108

A

- A^* -Algorithmus, 259, 288
- Ableitungsbaum, 54
- Abstand, 20, 242
- Adjazenzliste, 26
- Adjazenzmatrix, 25
 - bewertete, 28
- Ahuja, R.K., 282
- Algorithmus
 - A^* , 259, 282
 - approximativer, 138, 295, 296
 - ausgabesensitiver, 40
 - Dijkstra, 253, 285, 286, 288, 291
 - Dinic, 181, 190, 206, 213, 216, 218, 225, 226
 - Edmonds und Karp, 174, 176, 213
 - effizienter, 37
 - exponentieller, 290
 - Floyd, 276
 - Greedy, 296, 300, 306, 311, 325–327
 - greedy, 40
 - Huffman, 61
 - iterativer A^* , 264
 - Johnson, 306
 - Kruskal, 75, 77
 - Moore und Ford, 248, 251, 272, 282–285, 287
 - Nächster-Nachbar, 319
 - Preflow-Push, 197
 - Prim, 80, 227, 323, 406
 - probabilistischer, 295
 - Simplex, 290
 - Turner, 295
- Alt, H., 233
- Anfangsecke, 20
- Appel, K., 145, 156
- Arora, S., 305
- Ausgangsgrad, 19
- average case Komplexität, 36

B

Backtracking, 294, 325
 Backtracking-Verfahren, 141
 Baum, 22, 51, 53
 aufspannender, 52, 129, 293
 binärer, 22
 geordneter, 55
 minimal aufspannender, 74
 Baumkante, 92, 106
 Bellman, R.E., 282
 Bellmansche Gleichungen, 246
 benachbart, 19
 Bildverarbeitung, 108
 Binärbaum, 22, 56
 Birkhoffsscher Diamant, 337
 Blatt, 53
 Block, 112, 128
 Blockgraph, 112
 Brücke, 46, 356
 Branch-and-bound, 294, 317, 389
 breadth-first-search, 116
 Breitensuche, 87, 116, 174, 180, 184
 Breitensuchebaum, 116, 174
 Breitensuchenummer, 116
 Brin, S., 12, 13
 Brooks, R.L., 135, 156
 Bucket-Sort, 285
 Busstruktur, 2

C

C_n , 22
 c-Färbung, 131
 c-Kantenfärbung, 160
 Chaitin, G.J., 157
 Cherkassky, B.V., 197, 282
 Christofides, N., 321, 325
 chromatische Zahl, 132, 298
 Clique, 41, 133
 maximale, 327
 Cliquenzahl, 133
 Cobham, A., 290
 Cook, S.A., 293
 Culberson, J.C., 325

D

DAG, 94
 Dame, 258

Dantzig, G.B., 233

Datenübertragungsnetzwerk, 284
 Datenkompression, 59
 De Morgan, A., 145
 depth-first-search, 88
 Dijkstra, E.W., 253, 282
 Dinic, E.A., 181, 196
 directed acyclic graph, 94
 Directory, 54
 Dispatching, 7
 Dispatchtabelle, 8
 Distanzmatrix, 272
 divide and conquer, 24
 dominierende Menge, 329
 Dreieckssehne, 150
 Dreiecksungleichung, 318
 Durchsatz, 84, 186
 dynamisches Programmieren, 24, 317, 389

E

Ecke, 17
 aktive, 228
 innere, 22, 53
 isolierte, 19
 trennende, 110
 verwendbare, 246
 Eckenüberdeckung, 237, 300, 333
 minimale, 300, 333, 336
 eckendisjunkt, 215
 Eckenmenge
 minimal trennende, 215
 trennende, 214
 unabhängige, 137
 Eckenzusammenhangszahl, 216
 Edmonds, J., 174, 196, 282, 290
 Eigenschaft (*), 243
 Eingangsgrad, 19
 Endecke, 20
 Entscheidungsproblem, 290
 erreichbar, 21
 Erreichbarkeitsbaum, 88
 Erreichbarkeitsmatrix, 32
 Erweiterungskreis, 194
 Erweiterungsweg, 169, 172, 207, 234
 Euler, L., 321
 Eulersche Polyederformel, 145
 Eulerscher Graph, 332

Even, S., 157, 233

F

Färbbarkeit, 294

Färbung, 131, 293, 298, 305

einheitige, 239, 338

minimale, 132, 331

nicht triviale, 163

Fehler

absoluter, 297

Fertigungszelle, 5

Fibonacci-Heap, 81, 256, 272, 282

Floyd, R., 82, 276, 282

Fluß, 167

binärer, 191

blockierender, 181

kostenminimaler, 194

maximaler, 167

trivialer, 167

Wert, 167

zulässiger, 209

Ford, L.R., 196, 210, 233, 248, 252, 282

formale Sprache, 291

Fredman, M., 82

Fulkerson, D.R., 196, 210, 233

G

Güte, 297

Garey, A.R., 290, 325

geschichtetes Hilfsnetzwerk, 181, 191

Gilmore, P., 151

Gittergraph, 157

Goldberg, A.V., 197, 282

Grad, 19

Graph

k -partiter, 330

azyklischer, 94

bewerteter, 28

bipartiter, 22, 132, 204

c-kritischer, 158

eckenbewerteter, 28

Eulerscher, 321, 332

gerichtet, 18

Hamiltonscher, 315, 331

invertierter, 127

kantenbewerteter, 28

kreisfreier, 94

kritischer, 158

minimaler, 337

perfekter, 152

Petersen, 22

plättbarer, 23

planarer, 23, 124, 238, 282

regulärer, 22

schlichter, 18

transitiv orientierbarer, 150

transitiver, 150

ungerichtet, 17

vollständig bipartiter, 22

vollständiger, 22

vollständig k -partiter, 330

zufälliger, 44

zusammenhängender, 21

zyklischer, 22

greedy-Techniken, 24, 40

H

Höhe, 54, 149

Haken, W., 145, 156

Hall, P., 207, 233

Halldórsson, M.M., 314, 325

Hamiltonscher Kreis, 293, 295, 315

Handlungsreisendenproblem, 315, 335

Harary, F., 45, 338

Hassin, R., 312

Hayes, B., 13

Heap, 67, 255, 272

heapsort, 67

Heawood, P., 147

Hedetniemi, S., 338

Heesch, H., 145, 156

Herz, 46

Heuristik, 138, 295

hierarchisches Dateisystem, 54

Hoffmann, A., 151

Holyer, I., 325

Hopcroft, J.E., 124, 233

Huffman, D.A., 82

Hyperwürfel, 158

I

implizite Darstellung, 28

include-Mechanismus, 30

induzierter Untergraph, 21

Internet, 238
 Intervallgraph, 28, 163
 intractable, 290
 inzident, 19
 Inzidenzmatrix, 47

J

Jackowski, B., 156
 Johnson, D.S., 290, 306, 325, 326

K

Königsberger Brückenproblem, 131, 322
 künstliche Intelligenz, 108, 121, 258
 kürzester-Wege-Baum, 244
 Kante, 17
 gerichtete, 18
 kritische, 176
 Kantenbewertung, 241
 konstante, 252
 negative, 242
 nichtnegative, 253
 kantenchromatische Zahl, 160, 299, 332
 Kantenfärbung, 160, 299, 332
 Kantengraph, 338
 Kantenliste, 27
 Kantenmenge
 minimale trennende, 222
 trennende, 222
 Kantenzug, 20
 Länge, 242, 288
 minimaler Länge, 316
 Kantenzusammenhang, 222
 Kantenzusammenhangszahl, 222, 237
 Kapazität, 165
 Karp, R.M., 174, 196, 233, 282
 Kempe, A., 145, 147
 Kernighan, B.W., 326
 Knotenzusammenhang, 3
 Knuth, D., 45
 Kohäsion, 3
 Komplement, 21
 Komplexitätstheorie, 35
 Konfiguration, 337
 reduzible, 337
 Konfliktgraph, 9, 138
 Kosten, 74
 Kruskal, J.B., 75

Kubale, M., 156
 kW-Baum, 244

L

Lahar, A., 312
 Landausches Symbol O, 36
 late binding, 7
 Lawler, E.L., 325
 Lempel, A., 157
 Lenstra, J.K., 325
 Lewis, P.M., 325
 lexikographische Ordnung, 56
 Lin, S., 326
 lineare Programmierung, 290
 Link, 10
 Lovasz, L., 156
 Lund, C., 310, 325

M

Maheswari, S.N., 181
 Malhotra, V.M., 181
 Manzini, G., 282
 Markierungsalgorithmus, 88
 Matching, 204
 Matroid, 45
 Maximierungsproblem, 295
 Menger, V., 233
 Minimierungsproblem, 295
 mittlere Codewortlänge, 61
 Moore, E.F., 248, 252, 282
 multiple inheritance, 7
 Mycielski, J., 133, 157

N

Nachbar, 19
 Nachbarschaftsgraph, 108
 Nachfolger, 21, 53
 Netzwerk, 167
 Kosten, 194
 obere Kapazitäten, 209
 symmetrisches, 223, 236
 untere Kapazitäten, 209
 Niveau, 54, 116

O

objektorientierte Programmiersprache, 6
 Operations Research, 203, 208

Optimalitätsprinzip, 245, 317
Optimierungsproblem, 290
Ordnung, 36
Orientierung, 148
 kreisfrei, 149

P

Page, L., 12, 13
PageRank, 11, 13
Partition, 167
Permutationsdiagramm, 152
Permutationsgraph, 152
Petersen, J., 22
Petersen-Graph, 22, 23, 234, 236, 331
Platonische Körper, 337
Pnueli, A., 157
polynomiale Transformation, 292
Präfix-Codes, 60
Pramodh Kumar, M., 181
Prim, R., 80, 227, 253
Prioritäten, 71
Produkt von Graphen, 304
Public-Key Kryptosysteme, 140
Puzzle, 121

Q

q-s-Fluß, 166
q-s-Netzwerk, 165, 166
q-s-Schnitt, 167
Qualitätsgarantie
 absolute, 297
 relative, 299
Quelle, 165
Querkante, 92
Querschnitt, 84
Quicksort, 36

R

Rückwärtskante, 92, 107, 169, 213
Radzik, T., 282
Registerinterferenzgraph, 139
Reinelt, G., 326
Rekursion
 direkte, 96
 indirekte, 96
Ringstruktur, 2
Rinnoy Kan, A.H.G., 325

Robertson, N., 145
Robertson, N., 156
Robinson, R.W., 338
Robotik, 3, 262, 266
Rosenkrantz, D.J., 325
Routing-Problem, 257
Routingtabelle, 257
Roy, B., 33
RSA-System, 140

S

Safra, S., 305
Sanders, D., 145, 156
Satisfiability-Problem SAT, 293
Satz
 Ford und Fulkerson, 170, 213
 Hall, 207, 233, 389
 König-Egerváry, 238
 Menger, 215, 224, 233
 Whitney, 218, 225, 233
Schätzfunktion
 konsistente, 261, 288
 monotone, 268
 zulässige, 259
Schach, 237, 258
Scheduler, 72
Scheduling, 326
Schlüssel, 56
Schnitt, 2, 167
 maximaler, 333
 minimaler, 3, 226
Schnittgraphen, 159
Semiring, 282
Senke, 48, 165
Seymour, P., 145, 156
Shmoys, D.B., 325
Simplex-Algorithmus, 290
single inheritance, 7
Startecke, 244
Stearns, R.E., 325
Steiner Baum, 278, 335
 minimaler, 278
Steiner Ecke, 278
Sterngraph, 239
Sternstruktur, 2
Stoer, M., 227, 233
Strukturgraph, 99, 103

Suchbaum, 55
 binärer, 56
 höhenbalancierter, 59
 Suchbaumbedingung, 56
 Suche
 bidirektional, 271
 Suchmaschine, 10
 Symboltabelle, 55

T

Tait, P.G., 145
 Tarjan, R.E., 82, 124, 233
 Teilbaum mit Wurzel e , 54
 Thomas, R., 145, 156
 Tiefensuche, 87, 88, 125
 beschränkte, 123
 iterative, 123
 Tiefensuchenummer, 89
 Tiefensuchewald, 92
 topologische Sortierung, 94, 96, 99, 105,
 126, 127
 transitive Reduktion, 103
 transitiver Abschluß, 30, 31, 93, 103
 Transportproblem, 195
 Traveling-Salesman Problem, 13, 293, 315
 TSP, 316, 325
 Turing Maschine, 291
 Turner, J., 295, 325

U

Umkreissuche, 267
 unabhängige Eckenmenge, 45, 293, 304
 maximale, 333
 Unabhängigkeitszahl, 137, 297
 Untergraph, 21

V

Verbindungszusammenhang, 3
 Verletzlichkeit, 2, 214
 vermaschte Struktur, 2
 Verzeichnis, 54
 Vier-Farben-Problem, 131, 145
 Vizing, V.G., 299
 Vorgänger, 21, 53
 Vorgängermatrix, 272
 Vorrang-Warteschlange, 72
 Vorwärtskante, 92, 169, 213

W

Wagner, F., 227, 233
 Wald, 51
 Warshall, S.A., 33, 276, 282
 Web-Roboter, 10
 Weg, 20
 einfacher, 20
 geschlossener, 20
 kürzester, 3, 242, 291
 längster, 243, 292
 minimaler Länge, 316, 321
 offener, 20
 optimaler, 241
 Wegeplanungsverfahren, 13
 Whitney, H., 218
 Whitney, M., 233
 Williams, J., 82
 Windmühlengraph, 349
 Wirkungsgrad, 299
 asymptotischer, 299
 worst case Komplexität, 36
 Wurzel, 99
 Wurzelbaum, 22

Y

Yannakakis, M., 310, 325

Z

zero-knowledge Protokolle, 141
 Zuordnung, 204, 328, 331
 3-dimensionale, 333
 maximale, 204, 234, 292, 312
 minimale Kosten, 235, 323
 nicht erweiterbare, 204
 vollständige, 204
 zusammenhängend
 z -fach, 217
 quasi stark, 83
 stark, 98
 zweifach, 110
 Zusammenhangskomponente, 22, 107
 starke, 98
 Wurzel, 99
 Zusammenhangszahl, 216
 Zuverlässigkeitstheorie, 13
 zyklische Adreßkette, 78
 Zykov Baum, 372