Volker Turau, Christoph Weyer
Algorithmische Graphentheorie
De Gruyter Studium

Volker Turau, Christoph Weyer

Algorithmische Graphentheorie

4., erweiterte und überarbeitete Auflage

DE GRUYTER

Mathematics Subject Classification 2010
Primary: 68R10, 05C85; Secondary: 68W40, 90C35, 94C15, 68W25

Autoren
Prof. Dr. Volker Turau
Technische Universität Hamburg-Harburg
Schwarzenbergstr. 95
21073 Hamburg
turau@tu-harburg.de

Christoph Weyer
Technische Universität Hamburg-Harburg
Schwarzenbergstr. 95
21073 Hamburg
c.weyer@tu-harburg.de

ISBN 978-3-11-041727-2
e-ISBN (PDF) 978-3-11-041732-6
e-ISBN (EPUB) 978-3-11-042000-5

Library of Congress Cataloging-in-Publication Data
A CIP catalog record for this book has been applied for at the Library of Congress.

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de abrufbar.

Druck und Bindung: CPI books GmbH, Leck
♾ Gedruckt auf säurefreiem Papier
Printed in Germany

www.degruyter.com

Vorwort

In den 20 Jahren seit dem Erscheinen der ersten Auflage dieses Buches hat das Thema Algorithmische Graphentheorie nicht an Bedeutung verloren. Im Gegenteil, durch die zunehmende Durchdringung aller Lebensbereiche mit Informations- und Kommunikationstechnologien wird die Algorithmische Graphentheorie zu einem unverzichtbaren Baustein jeder Informatikausbildung. Dem wurde in der vorliegenden vierten Auflage Rechnung getragen durch das Hinzufügen eines neuen umfangreichen Kapitels über Entwurfsmethoden der Algorithmischen Graphentheorie. Die behandelten Entwurfsmethoden werden in den einzelnen Kapiteln konkret angewendet. Der Klasse der perfekten Graphen ist jetzt ein eigenes Kapitel gewidmet. Das Kapitel über approximative Algorithmen wurde neu strukturiert und erweitert. Die Neuauflage des Buches zeichnet sich durch eine wesentlich umfangreichere und detailliertere Bebilderung aus, viele Bilder sind erstmals in Farbe. Auch die Darstellung der Algorithmen durch Pseudocode wurde verbessert.

Wie schon bei den vorangegangenen Auflagen möchten wir den Lesern für ihre Anregungen danken. Unser besonderer Dank gilt Frau Kristina Ahlborn für ihre Unterstützung beim Korrekturlesen des Textes.

Die Lösungen der 280 Übungsaufgaben sind nicht mehr im Buch enthalten sondern können kostenfrei über die Web-Seite[1] des Verlags bezogen werden.

Hamburg, August 2015

Volker Turau
Christoph Weyer

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

1 http://dx.doi.org/10.1515/9783110417326-suppl

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 allejene 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 6 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 8 und 9 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 9 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 10 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 11 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.

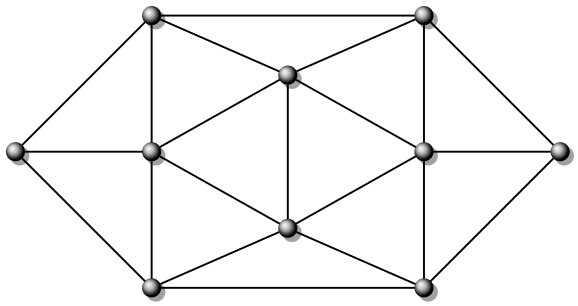
Ein unerfahrener Leser sollte zumindest die ersten vier Kapitel sequentiell lesen. Die restlichen Kapitel sind relativ unabhängig voneinander (mit Ausnahme von Kapitel 9, welches auf Kapitel 8 aufbaut). Das Kapitel über approximative Algorithmen enthält viele neuere Forschungsergebnisse und ist aus diesem 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 LaTeX 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

Inhalt



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 Anwendungsgebieten 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.

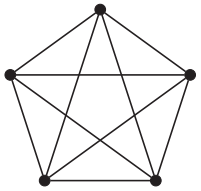
Im Folgenden werden sechs verschiedene Anwendungen diskutiert. Eine genaue Betrachtung der Aufgabenstellungen führt ganz natürlich auf eine graphische Beschreibung und damit auch auf den Begriff des Graphen; eine Definition wird bewusst erst im nächsten Kapitel vorgenommen. Die Beispiele dienen als Motivation 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

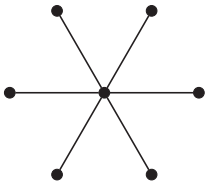
Ein Kommunikationsnetz ist ein durch Datenübertragungswege realisierter Verband mehrerer Rechner. Es unterstützt den Informationsaustausch zwischen Benutzern an verschiedenen Orten. Die *Verletzlichkeit* eines Kommunikationsnetzes ist durch die Anzahl von Leitungen oder Rechnern gekennzeichnet, die ausfallen 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 Verbindungslinien zwischen den Knoten darstellen: Die Knoten entsprechen den Stationen, die Verbindungslinien den Datenübertragungswegen. In Abbildung 1.1 sind vier Grundformen gebräuchlicher Netzwerke dargestellt.

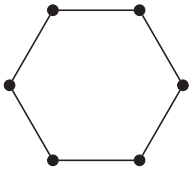
Abb. 1.1: Netzwerktopologien



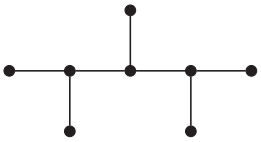
(a) Vermaschte Struktur



(b) Sternstruktur



(c) Ringstrutkur



(d) Busstruktur

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

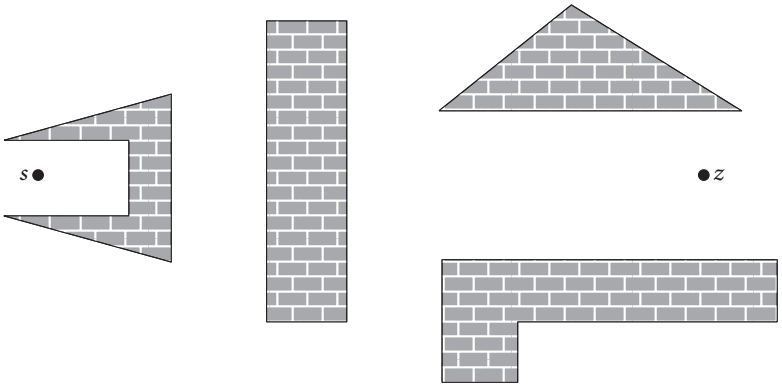
Analog kann man auch den Begriff *Knotenzusammenhang* eines Netzwerkes definieren. Er gibt die minimale Anzahl von Stationen an, deren Ausfall die Kommunikation der restlichen Stationen untereinander unterbrechen würde. Bei der Bus- und Sternstruktur ist diese Zahl gleich 1 und bei der Ringstruktur gleich 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 Verbindungszusammenhang und Knotenzusammenhang eines Netzwerkes werden in Kapitel 9 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.

Abb. 1.2: Hindernisse für einen Roboter



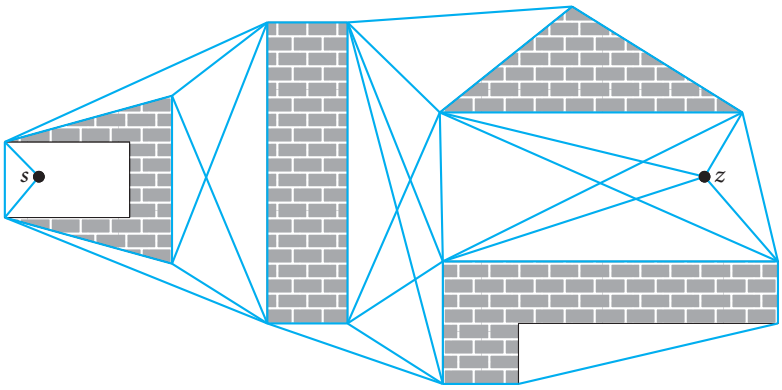
Eine stark idealisierte Version dieses komplizierten Problems erhält man, indem man die Bewegung auf eine Ebene beschränkt und den Roboter auf einen Punkt reduziert. Diese Ausgangslage kann auch in vielen Anwendungen durch eine geeignete Transformation geschaffen werden. Das Problem stellt sich nun folgendermaßen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

(1) Geradensegmente von s oder z zu einer konvexen Ecke eines Hindernisses, welche keine anderen Hindernisse schneiden
(2) Kanten von Hindernissen zwischen konvexen Ecken
(3) Geradensegmente zwischen konvexen Ecken von Hindernissen, welche keine anderen Hindernisse schneiden
(4) 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.

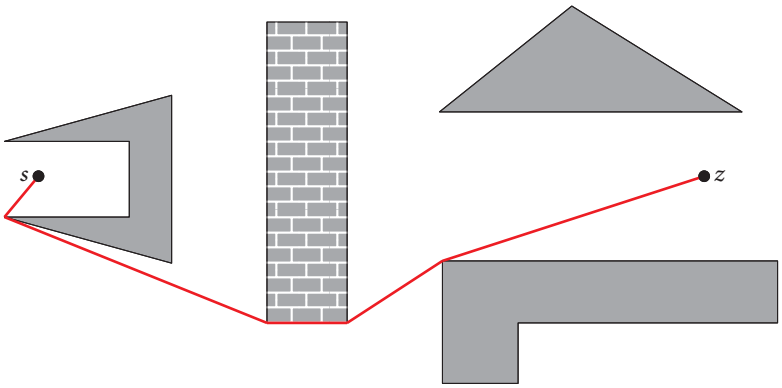
Abb. 1.3: Geradensegmente für den kürzesten Weg



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 blau eingezeichnet. Abbildung 1.4 zeigt den kürzesten Weg von s nach z. Verfahren zur Bestimmung von kürzesten Wegen in Graphen werden in Kapitel 10 behandelt.

Abb. 1.4: Der kürzeste Weg von s nach z



1.3 Optimale Umrüstzeiten für Fertigungszellen

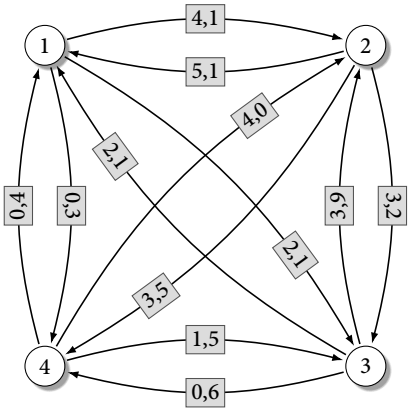
Die Maschinen einer Fertigungszelle in einer Produktionsanlage müssen für verschiedene Produktionsaufträge unterschiedlich ausgerüstet und eingestellt werden. Während dieser Umrüstzeit kann die Fertigungszelle nicht produktiv genutzt werden. Aus diesem Grund wird eine möglichst geringe Umrüstzeit angestrebt. 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 $P_1, \ldots, P_n$. 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 $P_{i_1}, \ldots, P_{i_n}$ 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 Umrüstvorgang durch einen

Abb. 1.5: Umrüstzeiten für vier Produktionsaufträge



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 Umrüstvorgänge 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 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 Wege. Eine Möglichkeit, den optimalen Weg zu finden, besteht darin, die Gesamtumrüstzeiten 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 11 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

Objektorientierte Programmiersprachen haben seit vielen Jahren prozedurale Sprachen in vielen Bereichen ersetzt. Java, C# und C++ sind Beispiele für solche Sprachen. In prozeduralen 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.

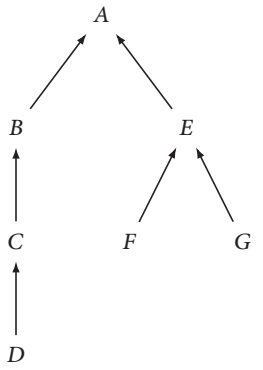
Die Auswahl der Implementierung einer Methode zur Laufzeit nennt man *Dispatching*. Konzeptionell geht man dabei wie folgt vor:

(1) Bestimmung der Klasse, zu der das Objekt gehört.
(2) Falls diese Klasse eine Implementierung für diese Methode zur Verfügung stellt, wird diese ausgeführt.
(3) Andernfalls werden in aufsteigender Reihenfolge die Oberklassen durchsucht und wie oben verfahren.
(4) 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 zeigt eine Klassenhierarchie bestehend aus den Klassen $A, B, \ldots, G$. Die Unterklassenrelation ist durch einen Pfeil von einer Klasse zu ihrer Oberklasse ge-

Abb. 1.6: Eine Klassenhierarchie



$A : m_1$
$B : m_1, m_2$
$C : m_3$
$D : m_3$

$E : m_1, m_5$
$F : m_1, m_4$
$G : m_4$

kennzeichnet. 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 $m \mid K$ dargestellt. Wird z. B. die Methode m_1 für eine Instanz der Klasse G aufgerufen, so ergibt sich direkt, dass $m_1 \mid E$ die zugehörige Implementierung ist.

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

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 Dispatching-Verfahren. Die effizienteste Möglichkeit, Dispatching durchzuführen, besteht darin, zur Übersetzungszeit eine globale *Dispatching-Tabelle* zu erzeugen. Die Dispatching-Tabelle für das oben angegebene Beispiel sieht wie folgt aus.

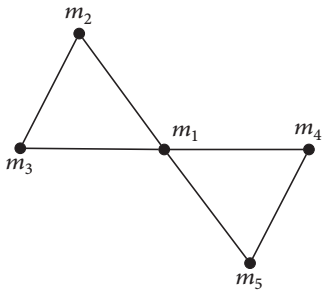
	A	B	C	D	E	F	G
m_1	$m_1 \mid A$	$m_1 \mid B$	$m_1 \mid B$	$m_1 \mid B$	$m_1 \mid E$	$m_1 \mid F$	$m_1 \mid E$
m_2		$m_2 \mid B$	$m_2 \mid B$	$m_2 \mid B$			
m_3			$m_3 \mid C$	$m_3 \mid D$			
m_4						$m_4 \mid F$	$m_4 \mid G$
m_5					$m_5 \mid E$	$m_5 \mid E$	$m_5 \mid E$

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 Dispatching-Tabelle 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 Dispatching-Tabelle leer. Diese Organisation garantiert, dass die Implementierung einer Methode in konstanter Zeit gefunden wird; es ist dabei genau ein Zugriff auf die Dispatching-Tabelle notwendig. Der große Nachteil einer solchen Dispatching-Tabelle ist der Platzverbrauch. Aus diesem Grund sind Dispatching-Tabellen in dieser Form praktisch nicht verwendbar.

Um den Speicheraufwand zu senken, macht man sich zunutze, dass die meisten Einträge der Dispatching-Tabelle 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 Dispatching-Tabelle eine gemeinsame Zeile verwenden. Um eine optimale Kompression zu finden, wird zunächst untersucht, welche Methoden sich nicht beeinflussen. Dazu wird ein so genannter *Konfliktgraph* gebildet. In diesem werden Methoden, welche sich beeinflussen, durch eine Kante verbunden. Abbildung 1.7 zeigt den Konfliktgraphen für das obige Beispiel.

Abb. 1.7: 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 Dispatching-Tabelle wird nun mittels den folgenden beiden Tabellen dargestellt. Die erste Tabelle gibt für jede Methode die entsprechende Zeile in der zweiten Tabelle an.

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

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.

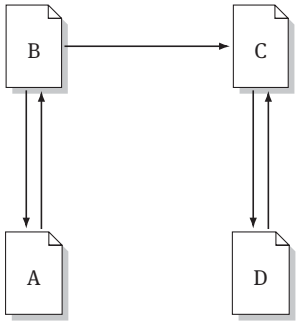
Das Problem bei dieser Vorgehensweise ist die Bestimmung einer Aufteilung der Methoden in eine minimale Anzahl von Teilmengen von sich nicht beeinflussenden Methoden. Dieses Problem wird ausführlich in den Kapiteln 6 und 11 behandelt.

1.5 Suchmaschinen

Suchmaschinen sind bei der Informationsrecherche im World Wide Web zu einem unverzichtbaren Werkzeug geworden. Diese erstellen und verwalten einen Index, der Stichwörter mit Web-Seiten verbindet. Eine Anfragesprache ermöglicht eine gezielte Suche in diesem Index. Der Index der Suchmaschine wird 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 1.8 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. Der Marktführer Google indexierte bereits 2008 mehr als eine Billion Web-Seiten.

Abb. 1.8: Vier Web-Seiten mit ihren Verweisen



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. Aus-

gehend von einer oder mehreren Startadressen durchsuchen Web-Roboter das Web und extrahieren nach verschiedenen Verfahren Wörter und fügen diese in den Index ein. Zur Verwaltung des Index 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 im Index 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.

Im ersten Schritt bestimmt eine Suchmaschine die Web-Seiten, welche dem Suchbegriff über den Index zugeordnet sind. 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 vermeintlichen 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 versuchen Autoren ihre Web-Seiten so zu gestalten, dass sie bei der Relevanzbeurteilung durch Suchmaschinen gut abschneiden. Beispielsweise werden zugkräftige Wörter mehrfach an wichtigen Stellen in den Dokumenten platziert, zum Teil sogar in Kommentaren. Mittlerweile sind die Suchverfahren aber so stark verfeinert worden, dass solche einfache Manipulationsversuche nicht mehr funktionieren.

Eine wichtige Idee besteht darin, die Anzahl der Verweise auf ein Dokument als ein grundsätzliches Kriterium in die Beurteilung der Relevanz einer Seite mit einzubeziehen. Ein Dokument ist dabei umso wichtiger, je mehr Dokumente auf es verweisen. Diese Idee wurde erstmals vom Suchdienst Google praktisch angewendet. Der so genannte *PageRank* basiert auf der Beobachtung, 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 Inhalt ein hoher Rang zugewiesen, wenn andere 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 L. Page und S. Brin entwickelte PageRank-Algorithmus ist sehr einfach nachvollziehbar. Für eine Web-Seite W bezeichnet $LC(W)$ die Anzahl der Verweise, welche von Seite W ausgehen, d. h., Verweise die im Text von W enthalten sind, und $PL(W)$ bezeichnet 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, umso weniger PageRank gibt sie an W weiter. Der gewichtete PageRank der Seiten $D \in PL(W)$ wird addiert. Dies hat zur Folge, dass jeder zusätzlich 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 einer andere verringert.

Die Eigenschaften des PageRank werden jetzt anhand des in Abbildung 1.8 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, wobei C den höchsten und A den niedrigsten PageRank erhält:

$$PR(A) = 0{,}71428 \quad PR(B) = 0{,}85714 \quad PR(C) = 1{,}2857 \quad PR(D) = 1{,}1428$$

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. Die nachfolgende Tabelle 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

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

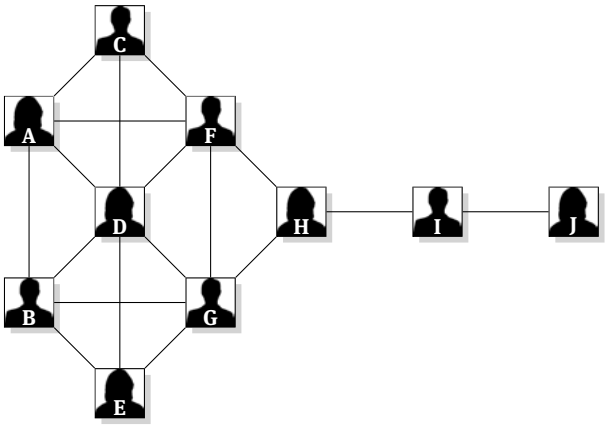
1.6 Analyse sozialer Netze

Der Begriff *Soziales Netzwerk* bezeichnet eine soziale Struktur, die zwischen menschlichen Akteuren mittels ihrer Interaktion entsteht. Der Begriff ist recht weit gefasst. Es ist nicht festgelegt, um welche Art von Interaktion es sich handelt und ob die Akteure Individuen, Gruppen oder Organisationen sind. Soziale Netzwerke lassen sich einfach als Graphen modellieren. Die Akteure bilden die Ecken und Interaktionen zwischen ihnen werden als Kanten dargestellt. Der US-amerikanische Psychologe S. Milgram untersuchte bereits in den 1960er Jahren soziale Netze. Er vertrat die Hypothese, dass jeder Mensch auf der Welt mit jedem anderen über eine überraschend kurze Kette von Bekanntschaftsbeziehungen verbunden ist. Im zugehörenden sozialen Netzwerk sind die Menschen die Ecken und die Bekanntschaft zweier Menschen wird durch eine Kante repräsentiert. Diese Art der Darstellung von sozialen Beziehungen wird auch *Soziogramm* genannt. Da es unmöglich ist diesen Graph explizit anzugeben, beschränkte sich Milgram auf empirische Untersuchungen (*Small world experiment*). Er interpretierte die Ergebnisse in der Art, dass Bürger der USA im Durchschnitt durch eine Kette von 6 Personen verbunden sind. Die Schlussfolgerungen aus seinen Untersuchungen sind wegen der geringen Datenlage allerdings umstritten.

Im Zeitalter des Internet erreichte die Erforschung sozialer Netze eine neue Dimension. Das World Wide Web bietet zahllose Plattformen, mit denen Benutzer ein persönliches Profil verwalten können. Des Weiteren können Beziehungen zu anderen Personen abgebildet werden. Allein in Deutschland nutzten im Jahr 2009 mehr als 10 Millionen Menschen solche Websites. Jede dieser Plattformen definiert ein soziales Netzwerk, wobei die Nutzer die Akteure sind. Die Kommunikationsmuster in Diskussionsforen oder Email-Listen können ebenfalls durch soziale Netzwerke repräsentiert werden. Abbildung 1.9 zeigt ein soziales Netzwerk mit 10 Akteuren, diese sind mit den Buchstaben A bis J gekennzeichnet. Hierbei sind zwei Personen durch eine Kante ver-

bunden, falls sie regelmäßig Emails austauschen. Dieses Beispiel wird wegen seiner Form auch *Kite-Netzwerk* genannt und wurde erstmals von D. Krackhard verwendet.

Abb. 1.9: Das Kite-Netzwerk, ein Beispiel für ein soziales Netzwerk



Der Zusammenhang zwischen der Struktur eines sozialen Netzwerkes auf der einen und dem Verhalten der darin eingebetteten Akteure und Gruppen auf der anderen Seite ist seit vielen Jahrzehnten Gegenstand sozialwissenschaftlicher Untersuchungen. Die hierbei verwendeten Methoden werden unter dem Begriff *Analytik sozialer Netzwerke* zusammengefasst. Im Folgenden werden wichtige Konzepte dieser Analytik und ihre graphentheoretische Interpretation vorgestellt.

Ein wichtiges Merkmal eines sozialen Netzwerkes ist seine *Dichte*. Sie bezeichnet das Verhältnis der Anzahl der Kanten in einem Soziogramm zur Zahl der möglichen Kanten. Diese Maßzahl verdeutlicht die Verbundenheit des sozialen Netzwerkes. Die Dichte des in Abbildung 1.9 dargestellten Netzwerks beträgt $18/45 = 0{,}4$. Die Dichte liegt immer im Intervall $[0, 1]$. Je näher die Dichte an 1 liegt, desto schneller kann sich beispielsweise eine Informationen im Netzwerk ausbreiten. Im Folgenden bezeichne n stets die Anzahl der Akteure und m die Anzahl der Kanten in einem sozialen Netzwerk. Die Dichte ist durch folgenden Ausdruck gegeben:

$$\frac{2m}{n(n-1)}.$$

Ein weiteres Mittel zur Untersuchung von Gesamtnetzwerken ist die *Cliquenanalyse*. Unter einer *Clique* versteht man eine Teilmenge der Ecken, die alle untereinander direkt verbunden sind. Eine Clique kennzeichnet eine Gruppe von Akteuren mit hoher Kohäsion. Die Cliquenanalyse unterteilt ein Gesamtnetzwerk in disjunkte Cliquen. Diese Aufteilung bringt Erkenntnisse über die Struktur und den inneren Zusammenhang eines sozialen Netzwerkes. Das in Abbildung 1.9 dargestellte Netzwerk kann beispielsweise in die disjunkten Cliquen $\{A, C, D, F\}$, $\{B, E, G\}$, $\{H, I\}$ und $\{J\}$ unterteilt werden. Es gibt keine Clique, die mehr als vier Akteure enthält.

Die Untersuchung eines sozialen Netzwerkes kann auch den Fokus auf einzelne Akteure richten. Dabei geht es um die Frage, wie zentral der Akteur ist bzw. welches Prestige er besitzt. Zur Messung von Zentralität (*Centrality*) wurden verschiedene Verfahren entwickelt, einige davon werden im Folgenden vorgestellt. Ein Akteur ist zentral im Sinne der *Degree-Centrality*, wenn er direkte Beziehungen zu möglichst vielen anderen Akteuren hat. Für einen Akteur, der mit $d(e)$ anderen Akteuren durch eine Kante verbunden ist, ist die Degree-Centrality $C_D(e)$ durch $d(e)/(n-1)$ gegeben. Diese Maßzahl kennzeichnet die Eingebundenheit eines Akteurs in einem Netzwerk. Je näher $C_D(e)$ an 1 liegt, desto höher ist das Prestige des Akteurs. In der folgenden Tabelle sind die Degree-Centrality Werte der 10 Akteure aus Abbildung 1.9 zusammengefasst. Akteur D hat mit $2/3 \approx 0{,}67$ den höchsten Wert.

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*
Degree	0,44	0,44	0,33	0,67	0,33	0,56	0,56	0,33	0,22	0,11
Betweenness	0,02	0,02	0	0,10	0	0,23	0,23	0,39	0,22	0
Closeness	1,89	1,89	2	1,67	2	1,56	1,56	1,67	2,33	3,22

Die *Betweenness-Centrality* ist ein Maß für die Bedeutung eines Akteurs für den Informationsaustausch. Ein Akteur hat signifikanten Einfluss auf den Informationsaustausch zwischen zwei anderen Akteuren, wenn er auf einem kürzesten Pfad zwischen diesen beiden Akteuren liegt. Ein Akteur gilt dann als zentral, wenn er auf möglichst vielen kürzesten Pfaden liegt. In Abbildung 1.9 haben die Akteure F, G und H zentrale Positionen. Die Betweenness-Centrality $C_B(e)$ eines Akteurs e kann wie folgt bestimmt werden:

$$C_B(e) = \frac{2}{(n-1)(n-2)} \sum_{s \neq e \neq z, s \neq z} \frac{\sigma_{sz}(e)}{\sigma_{sz}}.$$

Hierbei bezeichnet σ_{sz} die Anzahl der kürzesten Pfade von s nach z und $\sigma_{sz}(e)$ die Anzahl dieser Pfade, die e enthalten. Summiert wird über alle ungeordneten Paare s, z von Akteuren mit $s \neq e \neq z$ und $s \neq z$. Der Faktor $2/(n-1)(n-2)$ dient der Normierung des Wertes auf das Intervall $[0, 1]$. Je näher $C_B(e)$ an 1 ist, desto weitreichender sind die Kontrollmöglichkeiten, die dem Akteur e aufgrund seiner strategischen Position im Netzwerk zufallen. Dieses Maß geht implizit davon aus, dass Kommunikation immer entlang kürzester Wege erfolgt.

Zur Bestimmung von $C_B(H)$ bezüglich des in Abbildung 1.9 dargestellten Netzwerks beachte man, dass $\sigma_{IJ}(H) = 0$ und $\sigma_{sz}(H) = 0$ für $\{s, z\} \subset \{\text{A,B,C,D,E,F,G}\}$ gilt. Da H auf allen kürzesten Wegen zwischen Ecken in den Mengen {I,J} und {A,B,C,D,E,F,G} liegt, gilt $\sigma_{sI}(H) = \sigma_{sI}$ und $\sigma_{sJ}(H) = \sigma_{sJ}$ für $s \in \{\text{A,B,C,D,E,F,G}\}$. Somit ergibt sich $C_B(H) = 2/72 \cdot 14 \approx 0{,}39$, für alle anderen Akteure ist C_B kleiner. Die restlichen Werte sind in der obigen Tabelle zusammengefasst.

Bei der *Closeness-Centrality* C_C werden neben den direkten auch die indirekten Beziehungen betrachtet. Der Wert dieser Maßzahl für einen Akteur e entspricht den aufsummierten Längen der kürzesten Pfade, über die der betrachtete Akteur zu allen

anderen in Beziehung steht, dividiert durch $n-1$:

$$C_C(e) = \frac{\sum_{z \neq e} d(e,z)}{n-1}.$$

Summiert wird über alle Akteure $z \neq e$, welche von e aus erreichbar sind. Hierbei bezeichnet $d(e,z)$ die Anzahl der Kanten auf einem kürzesten Weg von e nach z. $C_C(e)$ ist ein Maß für die Nähe eines Akteurs e zu allen anderen. Je kleiner $C_C(e)$ ist, desto schneller kann der Akteur mit den restlichen Akteuren interagieren. Diese Maßzahl ist vor allem in der Bewertung der Kommunikation zwischen Akteuren von Bedeutung. In Abbildung 1.9 haben F und G mit jeweils $1,56$ die kleinste Closeness-Centrality (siehe vorherige Tabelle). Von diesen Akteuren verbreiten sich Informationen am schnellsten im gesamten Netzwerk aus.

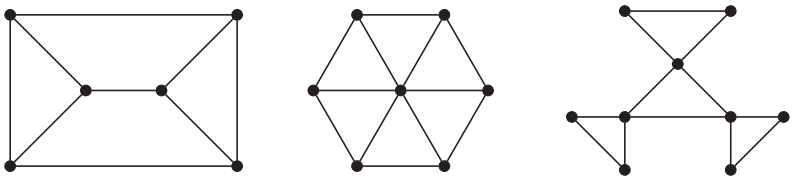
Cliquen werden in den Kapiteln 5 und 6 behandelt. Die Aufgaben 26 in Kapitel 11 und 5 in Kapitel 2 beschreiben Verfahren zur Bestimmung großer Cliquen. Effiziente Algorithmen zur Bestimmung der Betweenness- und der Closeness-Centrality werden in Abschnitt 10.5 und Aufgabe 36 in Kapitel 10 vorgestellt.

1.7 Literatur

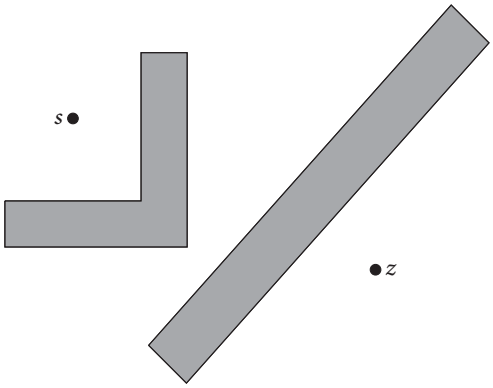
Verletzlichkeit von Kommunikationsnetzen ist ein Teilgebiet der *Zuverlässigkeitstheorie*. Eine ausführliche Darstellung der graphentheoretischen Aspekte beschreibt [69]. Eine Übersicht über *Wegeplanungsverfahren* in der Robotik findet man in [77]. Das Problem der optimalen Umrüstzeiten ist eng verwandt mit dem *Traveling-Salesman Problem*. Eine ausführliche Darstellung ist in [78] enthalten. Verfahren zur Optimierung von Methodendispatching in objektorientierten Programmiersprachen sind in [5] beschrieben. Information zu Algorithmen für Suchmaschinen findet man in [76]. Weitere praktische Anwendungen der Graphentheorie sind in [52, 53] beschrieben. Die Grundzüge der Analyse sozialer Netzwerke behandelt [16].

1.8 Aufgaben

1. 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.

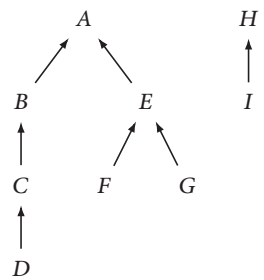


3. Im Folgenden sind die Umrüstzeiten T_{ij} für vier Produktionsaufträge für eine Fertigungszelle in Form einer Matrix gegeben.

$$\begin{pmatrix} 0 & 1{,}2 & 2 & 2{,}9 \\ 0{,}8 & 0 & 1{,}0 & 2{,}3 \\ 2 & 1{,}2 & 0 & 3{,}4 \\ 5{,}1 & 1{,}9 & 2{,}1 & 0 \end{pmatrix}$$

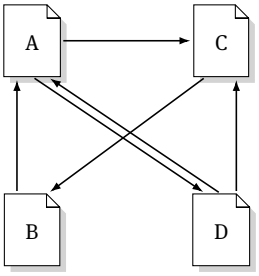
   Bestimmen Sie die optimale Reihenfolge der Produktionsaufträge, um die geringste Gesamtumrüstzeit zu erzielen. Die Fertigungszelle befindet sich anfangs im Zustand 1.

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

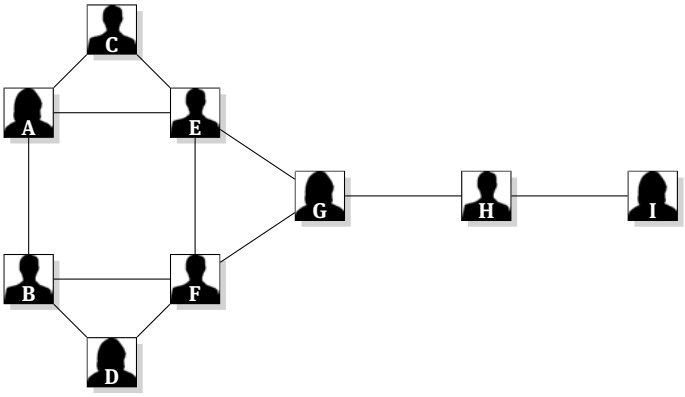


$A : m_1$
$B : m_1, m_4$
$C : m_1, m_3$
$D : m_1$
$E : m_2, m_3$
$F : m_1$
$G : m_1$
$H : m_4$
$I : m_5$

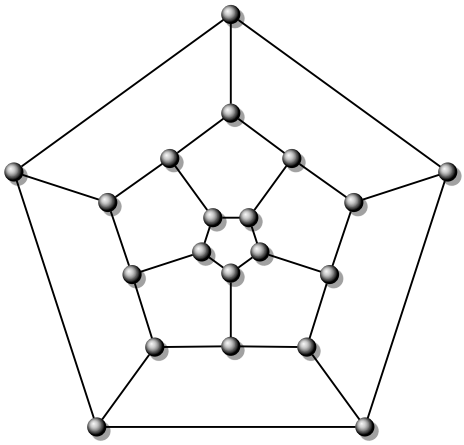
5. Es sei M eine Menge von Web-Seiten mit folgenden beiden Eigenschaften:
   (a) Jede Seite in M enthält mindestens einen Verweis und
   (b) außerhalb von M gibt es keine Seiten, welche auf Seiten in M verweisen.
   Beweisen Sie, daß die Summe der Werte des PageRank aller Seiten aus M gleich der Anzahl der Seiten in M ist.
6. Bestimmen Sie den PageRank aller Seiten der folgenden Hypertextstruktur ($d = 0,5$).



7. Bestimmen Sie die Dichte des folgenden sozialen Netzwerks. Berechnen Sie Degree-Centrality, Betweenness-Centrality und Closeness-Centrality für jeden Akteur.



8. Es sei G ein Graph, bei dem jede Ecke mit jeder anderen Ecken verbunden ist. Berechnen Sie für alle Ecken die Zentralitätsmaße: Degree-Centrality, Betweenness-Centrality und Closeness-Centrality.
9. Geben Sie ein Beispiel für ein soziales Netzwerk an, in dem es eine Ecke e mit $C_B(e) = 1$ gibt.



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. Die Beschreibung von Graphalgorithmen und deren Implementierung in einer objektorientierten Sprache werden ausführlich diskutiert. Die nachfolgenden Kapitel beschäftigen sich dann ausführlich mit weiterführenden Konzepten.

2.1 Grundlegende Definitionen

Ein *ungerichteter Graph* G 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* G besteht aus einer Menge E von Ecken und einer Menge K von *gerichteten Kanten*, die durch geordnete Paare von Ecken, den Anfangsecken und den Endecken, 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.

Abb. 2.1: Diagramme von Graphen

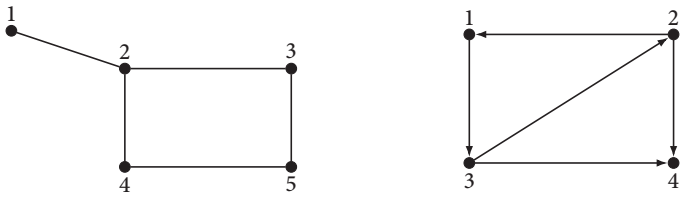


Abbildung 2.1 zeigt zwei Diagramme, die einen gerichteten und einen ungerichteten Graphen repräsentieren. Die Ecken- bzw. Kantenmenge des ungerichteten Graphen ist:

$$E = \{e_1, e_2, e_3, e_4, e_5\}$$
$$K = \{(e_1, e_2), (e_2, e_3), (e_3, e_5), (e_5, e_4), (e_2, e_4)\}$$

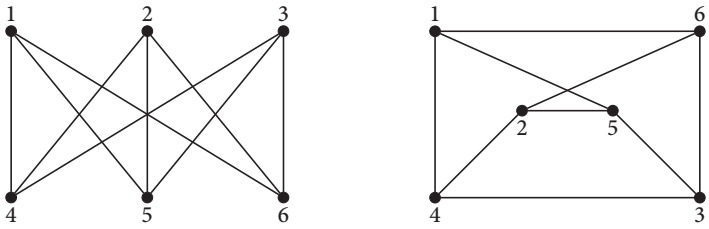
und die des gerichteten Graphen ist:

$$E = \{e_1, e_2, e_3, e_4\}$$
$$K = \{(e_1, e_3), (e_2, e_1), (e_2, e_4), (e_3, e_2), (e_3, e_4)\}$$

Häufig werden die Ecken mit den Zahlen von 1 bis n durchnummeriert. Diese Numerierung wird auch in den Abbildungen verwendet. Zur besseren Lesbarkeit werden die Ecken im Text mit e_1 bis e_n bezeichnet. Im Folgenden wird die Anzahl der Ecken immer mit n und die der Kanten immer mit m bezeichnet.

Eine geometrische Darstellung 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. Ferner müssen die Kanten auch nicht durch Strecken dargestellt werden, sondern es können beliebige Kurven verwendet werden.

Abb. 2.2: Verschiedene Diagramme für den gleichen Graphen



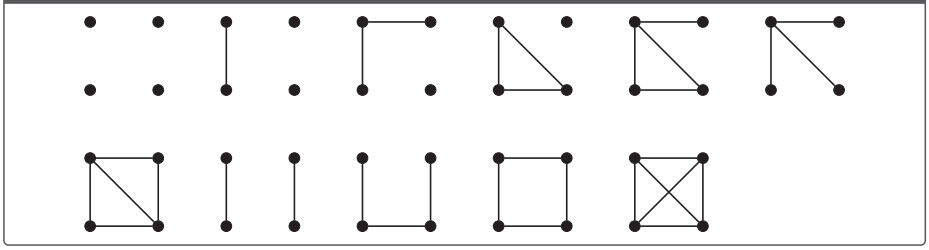
Die Definition eines Graphen schließt nicht aus, dass es zwischen zwei Ecken mehr als eine Kante gibt und dass es Kanten gibt, deren Anfangs- und Endecke identisch sind.

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.

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. Abbildung 2.3 zeigt alle elf ungerichteten Graphen, die genau vier Ecken haben.

Ein ungerichteter Graph heißt *vollständig*, wenn alle Ecken paarweise benachbart sind. Der vollständige Graph mit n Ecken wird mit K_n bezeichnet. Der letzte Graph in Abbildung 2.3 ist der Graph K_4.

Abb. 2.3: Die ungerichteten Graphen mit genau vier Ecken

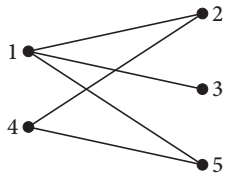


Zwei Ecken e und e' eines ungerichteten Graphen heißen *benachbart*, wenn es eine Kante von e nach e' gibt. Die Menge der *Nachbarn* einer Ecke e wird mit $N(e)$ bezeichnet. Für die Menge $N(e) \cup \{e\}$ wird die Bezeichnung $N[e]$ verwendet. Für eine Teilmenge $U \subseteq E$ bezeichnet $N_U(e)$ die Menge $N(e) \cap U$. Eine Ecke e eines ungerichteten Graphen ist *inzident* mit einer Kante k des Graphen, wenn e eine Endecke der Kante k ist.

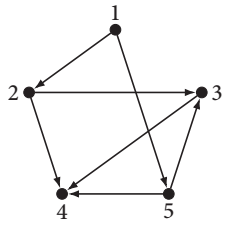
Der *Grad* $g(e)$ einer Ecke e in einem ungerichteten Graphen G ist die Anzahl der Kanten, die mit e inzident sind, d. h. $g(e) = |N(e)|$. Mit $\Delta(G)$ (oder kurz Δ) wird der maximale und mit $\delta(G)$ (oder kurz δ) der minimale Eckengrad von G bezeichnet. Es gilt $\Delta(K_n) = n - 1$.

In gerichteten Graphen werden die Nachbarn einer Ecke e in zwei Mengen aufgeteilt, die *Nachfolger* und die *Vorgänger* von e. Die Menge $N^+(e)$ bezeichnet alle Nachfolger, d. h. $N^+(e) = \{e' \mid (e, e') \in E\}$. Analog dazu bezeichnet $N^-(e) = \{e' \mid (e', e) \in E\}$ die Menge der Vorgänger von e. Des Weiteren unterscheidet man zwischen *Ausgangsgrad* $g^+(e)$ und dem *Eingangsgrad* $g^-(e)$ einer Ecke e. Es gilt $g^+(e) = |N^+(e)|$ und $g^-(e) = |N^-(e)|$. 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.

Abb. 2.4: Die Grade von Ecken



$g(e_1) = 3$ $\quad \Delta(G) = 3$
$g(e_2) = 2$ $\quad \delta(G) = 1$
$g(e_3) = 1$
$g(e_4) = 2$
$g(e_5) = 2$



$g^+(e_1) = 2$ $\quad g^-(e_1) = 0$
$g^+(e_2) = 2$ $\quad g^-(e_2) = 1$
$g^+(e_3) = 1$ $\quad g^-(e_3) = 2$
$g^+(e_4) = 0$ $\quad g^-(e_4) = 3$
$g^+(e_5) = 2$ $\quad g^-(e_5) = 1$

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

$$\sum_{e \in E} g(e) = 2m.$$

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

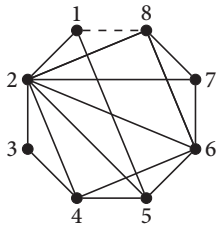
Eine Folge von Kanten $k_1, k_2, \ldots, k_z$ eines Graphen heißt *Kantenzug*, falls es eine Folge von Ecken $e_1, \ldots, e_z$ des Graphen gibt, so dass für jedes i zwischen 2 und z die Kante k_i gleich (e_{i-1}, e_i) ist. Zur Beschreibung eines Kantenzuges wird die Notation $<e_1, \ldots, e_z>$ verwendet. Die Anzahl der Kanten eines Kantenzug wird als die *Länge* des Kantenzuges bezeichnet. 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_1 die *Anfangsecke* und e_z die *Endecke* des Weges. Eine Ecke e kann man von der Ecke e' *erreichen*, falls es einen Weg von e nach e' gibt. Ist $e_1 = e_z$, 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. Vergleichen Sie dazu die Graphen in Abbildung 2.3.

Ein Weg heißt *einfacher Weg*, falls alle verwendeten Ecken bis auf Anfangs- und Endecke verschieden sind. Der *Umfang* eines ungerichteten Graphen ist definiert als die Länge eines geschlossenen einfachen Weges von geringster Länge. Enthält ein Graph keinen geschlossenen Weg so hat er den Umfang ∞. Der Umfang des ungerichteten Graphen aus Abbildung 2.4 ist 4.

Ein einfacher Weg eines ungerichteten Graphen, der alle Ecken des Graphen enthält, nennt man *Hamiltonschen Kreis*. Ein ungerichteter Graph heißt *Hamiltonscher*

Graph, wenn er einen Hamiltonschen Kreis besitzt. Der in Abbildung 2.5 dargestellte Graph besitzt keinen Hamiltonschen Kreis. Der Graph wird zu einem Hamiltonschen Graph, wenn eine Kante zwischen den Ecken e_1 und e_8 hinzugefügt wird.

Abb. 2.5: Ein Graph ohne Hamiltonschen Kreis



Der *Abstand* $d(e, e')$ zweier Ecken e, e' eines Graphen ist gleich der minimalen Anzahl von Kanten eines Weges mit Anfangsecke e und Endecke e'. Gibt es keinen solchen Weg, so ist $d(e, e') = \infty$. Es gilt $d(e, e) = 0$ für jede Ecke e eines Graphen. Der *Durchmesser* $D(G)$ eines Graphen G mit Eckenmenge E ist gleich dem größten Abstand zweier Ecken von G, d. h. $D(G) = \max\{d(e, e') \mid e, e' \in E\}$. Der ungerichtete Graph aus Abbildung 2.4 hat Durchmesser 3 und es gilt $d(e_4, e_3) = 3$.

Das *Komplement* eines ungerichteten Graphen G ist ein ungerichteter Graph $\overline{G}$ mit der gleichen Eckenmenge. Zwei Ecken sind in $\overline{G}$ dann und nur dann benachbart, wenn Sie nicht in G benachbart sind. Abbildung 2.6 zeigt einen ungerichteten Graphen und sein Komplement.

Abb. 2.6: Ein ungerichteter Graph und sein Komplement

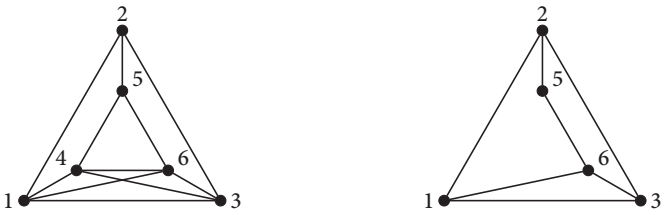
Sei G ein Graph mit Eckenmenge E und Kantenmenge K. Jeden Graph, dessen Eckenmenge eine Teilmenge von E und dessen Kantenmenge eine Teilmenge von K ist, nennt man *Untergraph* von G. Häufig werden Untergraphen betrachtet, welche alle Kanten enthalten, die inzident zu einer Ecke aus einer gegebenen Teilmenge U von E sind. Solche Untergraphen werden *induzierte Untergraphen* genannt und mit G_U bezeichnet. Formal ist G_U ein Graph mit Eckenmenge $U \subseteq E$ und der Kantenmenge

$$\{(e, e') \mid e, e' \in U \text{ und } (e, e') \in K\}.$$

Abbildung 2.7 zeigt einen Graphen mit sechs Ecken und den induzierten Untergraphen G_U für die Menge $U = \{e_1, e_2, e_3, e_5, e_6\}$. Man beachte, dass nicht jeder Untergraph ein induzierter Untergraph ist.

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.

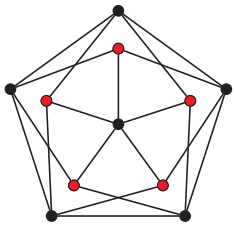
Abb. 2.7: Ein Graph und ein induzierter Untergraph



Ein Untergraph C eines ungerichteten Graphen G heißt *Clique*, falls C ein vollständiger Graph ist. Die Anzahl der Ecken einer maximalen Clique von G wird *Cliquenzahl* $\omega(G)$ von G genannt. Es gilt $\omega(K_n) = n - 1$. Der in Abbildung 2.7 links dargestellte Graph hat Cliquenzahl 4, eine entsprechende Clique wird von den Ecken e_1, e_3, e_4 und e_6 gebildet. Für den rechts dargestellten Graphen gilt $\omega(G) = 3$. Für den in Abbildung 2.8 abgebildeten Graphen gilt $\omega(G) = 2$.

Eine Teilmenge U der Eckenmenge eines ungerichteten Graphen G heißt *unabhängig*, falls kein Paar von Ecken aus U benachbart ist. Solche Eckenmengen werden kurz *unabhängige Mengen* genannt. Die in Abbildung 2.8 rot dargestellten Ecken bilden eine unabhängige Menge des dargestellten Graphen. Die *Unabhängigkeitszahl* $\alpha(G)$ von G ist die maximale Mächtigkeit einer unabhängigen Menge. Für den in Abbildung 2.8 abgebildeten Graphen ist $\alpha(G) = 5$. Es gilt $\alpha(K_n) = 1$. Ist U eine unabhängige Menge von G, dann ist U eine Clique von $\overline{G}$, d. h., es gilt $\alpha(G) = \omega(\overline{G})$.

Abb. 2.8: Ein Graph G mit $\alpha(G) = 5$ und $\omega(G) = 2$



2.2 Spezielle Graphen

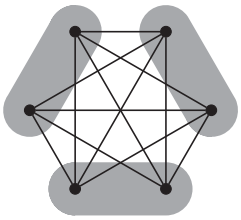
Ein ungerichteter Graph heißt *zusammenhängend*, falls es für jedes Paar e, e' von Ecken einen Weg von e nach e' gibt. Die Eckenmenge E eines ungerichteten Graphen kann in disjunkte Teilmengen $E_1, \ldots, E_z$ zerlegt werden, so dass die von den Mengen

E_i induzierten Untergraphen zusammenhängend sind. Dazu bilde man folgende Relation: $e, e' \in E$ sind äquivalent, falls es einen Weg von e nach e' 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 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. Ist $|E_1| = n_1$ und $|E_2| = n_2$, so wird der vollständig bipartite Graph mit K_{n_1,n_2} bezeichnet. Der Graph aus Abbildung 2.2 ist der Graph $K_{3,3}$. Der Graph $K_{1,n}$ wird *Sterngraph* genannt. Für einen bipartiten Graph G mit mindestens einer Kante gilt $\omega(G) = 2$.

Das Konzept von bipartiten Graphen kann verallgemeinert werden. Ein ungerichteter Graph G heißt *r-partit*, wenn die Eckenmenge E von G in r disjunkte Teilmengen $E_1, \ldots, E_r$ aufgeteilt werden kann, so dass die Endecken aller Kanten von G in verschiedenen Teilmengen E_i liegen. Die 2-partiten Graphen sind genau die bipartiten Graphen. Ein r-partiter Graph heißt *vollständig r-partit*, wenn alle Ecken aus verschiedenen Teilmengen E_i benachbart sind. Abbildung 2.9 zeigt einen vollständig 3-partiten Graphen. Die zu den Teilmengen E_i gehörenden Ecken sind jeweils grau hinterlegt.

Abb. 2.9: Ein vollständig 3-partiter Graph

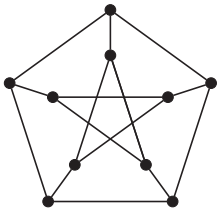


Ein Graph heißt *zyklisch*, falls er aus einem einfachen geschlossenen Weg besteht. Der zyklische Graph mit n Ecken wird mit C_n bezeichnet. C_n ist genau dann bipartit, wenn n gerade ist. Der vorletzte Graph in Abbildung 2.3 ist der Graph C_4. Es gilt $\omega(C_3) = 3$ und $\omega(C_n) = 2$ für $n > 3$. Ferner gilt $\alpha(C_{2n+1}) = n$ und $\alpha(C_{2n}) = n$.

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 2.10 haben alle den Grad 3. Dieser Graph ist nach dem Graphentheoretiker J. Petersen benannt.

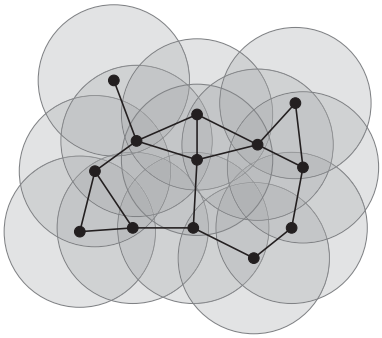
Graphen können auch auf Basis mathematischer Strukturen definiert werden. Dazu werden im Folgenden zwei Beispiele vorgestellt. Es sei P eine endliche Menge von Punkten der zweidimensionalen reellen Ebenen. Die Menge P definiert einen Graphen

Abb. 2.10: Der Petersen Graph



bei dem die Punkte den Ecken entsprechen. Zwei Ecken sind durch eine Kante verbunden, wenn der euklidische Abstand der zugehörenden Punkte kleiner oder gleich einer Konstanten a ist. Solche Graphen werden *Unit Disk Graphen* genannt. Sie werden zur Modellierung von drahtlosen Funknetzwerken verwendet. Dabei entsprechen die Ecken den Funkgeräten, welche alle die gleiche Sendereichweite S_R haben. Der daraus abgeleitete Unit Disk Graph mit der Konstanten $a = S_R$ zeigt, welche Funkgeräte direkt miteinander kommunizieren können. Abbildung 2.11 zeigt einen Unit Disk Graph. Anhand der Kreise kann leicht überprüft werden, ob zwei Ecken benachbart sind.

Abb. 2.11: Ein Unit Disk Graph



Im zweiten Beispiel entsprechen die Vektoren eines s-dimensionalen Vektorraums über dem Körper $GF[2]$ den Ecken eines ungerichteten Graphen. Jede der 2^s Ecken wird durch einen Vektor $(a_1, \ldots, a_s)$ mit $a_i \in \{0, 1\}$ repräsentiert. Ein *Hypercube* Q_s der Dimension s ist ein ungerichteter Graph bei dem zwei Ecken e, e' genau dann benachbart sind, wenn sich die zu e und e' gehörenden Vektoren an genau einer Position unterscheiden. Abbildung 2.12 zeigt den Hypercube der Dimension 3. Da die Vektoren benachbarter Ecken sich an genau einer Stelle unterscheiden, hat jede Ecke genau s Nachbarn, somit ist Q_s regulär mit $\Delta(Q_s) = s$.

Ein ungerichteter Graph, der keinen geschlossenen Weg enthält, heißt *Wald*. Die Zusammenhangskomponenten eines Waldes nennt man *Bäume*. In Bäumen gibt es zwischen je zwei Ecken des Graphen genau einen Weg. Abbildung 2.13 zeigt alle möglichen Bäume mit höchstens fünf Ecken.

Abb. 2.12: Der Hypercube Q_3

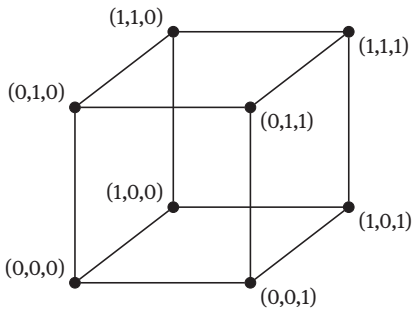
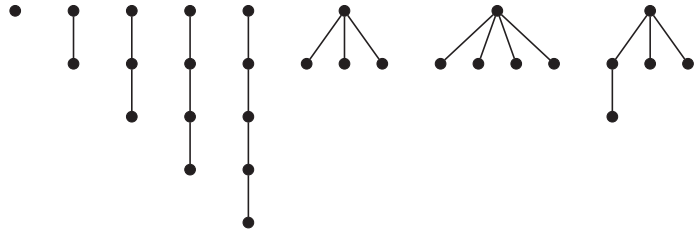


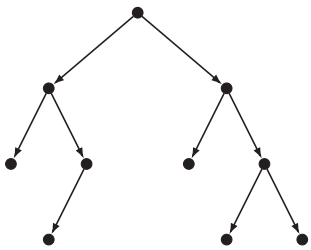
Abb. 2.13: Alle Bäume mit höchstens fünf Ecken



Ein *Wurzelbaum* ist ein Baum, bei dem eine Ecke als Wurzel ausgezeichnet ist. Von der Wurzel verschiedene Ecken mit Grad 1 nennt man Blätter, alle anderen Ecken nennt man *innere Ecken*. Wurzelbäume werden häufig als gerichtete Graphen interpretiert. Dazu werden alle Kanten weg von der Wurzel orientiert. In Darstellungen von Wurzelbäumen wird die Wurzel und die Orientierung der Kanten oft nicht explizit angegeben. Diese Information ist implizit dadurch gegeben, dass die Wurzel am höchsten liegt und die Kanten nach unten gerichtet sind.

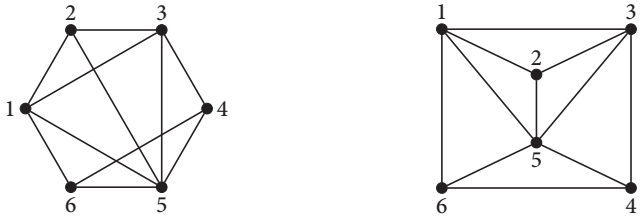
Im Folgenden werden Wurzelbäume in diesem Sinn oft als gerichtete Graphen interpretiert. Dann hat jede Ecke außer der Wurzel den Eingrad 1. Abbildung 2.14 zeigt einen gerichteten Baum, bei dem der Ausgrad jeder Ecke höchstens zwei ist. Solche Bäume nennt man *Binärbäume*.

Abb. 2.14: 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 2.15 zeigt links einen planaren Graphen und rechts eine kreuzungsfreie Darstellung dieses Graphen. Die Graphen $K_{3,3}$ und K_5 sind Beispiele für nicht planare Graphen. Sie besitzen keine kreuzungsfreien Darstellungen in der Ebene (vergleichen Sie 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 6.5 wird bewiesen, dass sie nicht planar sind.

Abb. 2.15: Ein planarer Graph und eine kreuzungsfreie Darstellung



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 erzeugt meist Algorithmen, die nicht sehr effizient sind. Der zweite Schritt besteht darin, den Algorithmus mittels einiger Standardtechniken zu verbessern. Kapitel 5 enthält eine Einführung in solche Techniken. Diese 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. Dadurch wird oft eine weitere Verbesserung der Laufzeit oder eine Senkung des Speicheraufwandes erreicht. Oft 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. Zum Abschluss wird eine mathematische Analyse der verwendeten Datenstrukturen vorgenommen. In diesem Buch wird versucht, die grundlegenden Techniken von Graphalgorithmen verständ-

lich 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.

Im Folgenden werden Datenstrukturen für Graphen bereitgestellt und eine Methodik zur Charakterisierung der Effizienz von Algorithmen beschrieben.

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. Dieser Schritt 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) Iteration über die Menge der Nachbarn einer Ecke.
(3) Iteration über die Menge der Nachfolger und Vorgänger einer Ecke in einem gerichteten Graphen.

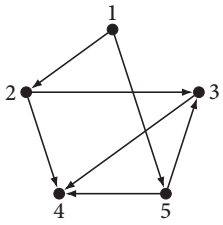
Neben der Effizienz der Operationen ist auch der Verbrauch an Speicherplatz ein wichtiges Kriterium für die Wahl einer Datenstruktur. Häufig besteht eine Wechselwirkung zwischen dem Speicherplatzverbrauch und der Effizienz der Operationen. Eine sehr effiziente Ausführung der Operationen geht oft zu Lasten 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 oft zu effizienteren Datenstrukturen. Im Folgenden werden wichtige 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 e_i nach e_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 2.16 zeigt einen gerichteten Graphen und seine Adjazenzmatrix.

Abb. 2.16: Ein gerichteter Graph und seine Adjazenzmatrix



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

Ist ein Diagonaleintrag der Adjazenzmatrix gleich 1, so bedeutet dies, dass eine Schlinge vorhanden ist. Um festzustellen, ob es eine Kante von e_i nach e_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 e_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.

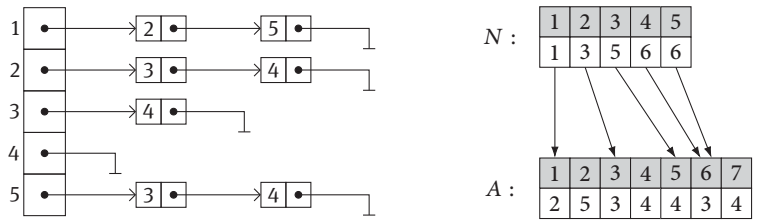
Für die Adjazenzmatrix werden unabhängig von der Anzahl der Kanten immer n^2 Speicherplätze benötigt. Für Graphen mit weniger Kanten sind dabei die meisten Einträge gleich 0. Da die Adjazenzmatrix eines ungerichteten Graphen symmetrisch ist, genügt es in diesem Fall, die Einträge oberhalb der Diagonale zu speichern. Dann benötigt man nur $n(n-1)/2$ Speicherplätze (vergleichen Sie Aufgabe 18).

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 e_i. Die Nachbarn selber werden als verkettete Listen dargestellt. Abbildung 2.17 zeigt links die entsprechende Darstellung des Graphen aus Abbildung 2.16.

Abb. 2.17: 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 e_1, dann die Nachbarn der Ecke e_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 e_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 17). Abbildung 2.17 zeigt rechts die entsprechende Darstellung des Graphen aus Abbildung 2.16.

Um festzustellen, ob es eine Kante von e_i nach e_j gibt, muss die Nachbarschaftsliste von Ecke e_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 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 $n(n-1)/2$ ist, verbraucht die Adjazenzliste weniger Platz als die Adjazenzmatrix. Bei dynamischen Anwendungen ist eine 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 2.18 zeigt die Kantenliste des Graphen aus Abbildung 2.16.

Abb. 2.18: Die Kantenliste eines gerichteten Graphen

K :

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

Um bei einem ungerichteten Graphen festzustellen, ob es eine Kante von e_i nach e_j gibt, muss die Kantenliste ganz 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 e_i nach e_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.

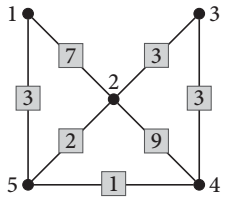
2.4.4 Bewertete Graphen

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 Anwendungen ist W häufig eine Teilmenge von $\mathbb{R}$ oder eine Menge von Wörtern. Abbildung 2.19 zeigt einen kantenbewerteten Graphen mit Wertemenge $W = \mathbb{N}$.

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* ge-

Abb. 2.19: Ein kantenbewerteter Graph mit seiner bewerteten Adjazenzmatrix



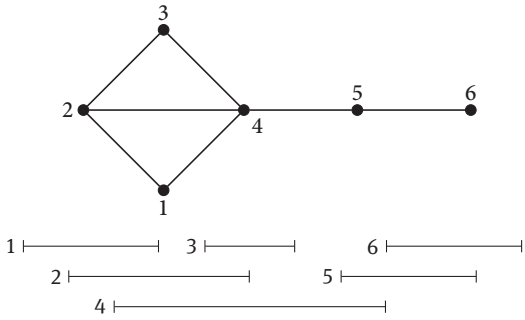
$$B = \begin{pmatrix} 0 & 7 & 0 & 0 & 3 \\ 7 & 0 & 3 & 9 & 2 \\ 0 & 3 & 0 & 3 & 0 \\ 0 & 9 & 3 & 0 & 1 \\ 3 & 2 & 0 & 1 & 0 \end{pmatrix}$$

bildet werden. Hierbei ist $B[i, j]$ die Bewertung der Kante von e_i nach e_j. Gibt es keine Kante von e_i nach e_j, so wird ein Wert, welcher nicht in W liegt, an der Stelle $B[i, j]$ abgelegt. Für nummerische Wertemengen kann dies z. B. ∞ oder die Zahl 0 sein, sofern $0 \notin W$. Abbildung 2.19 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 können strukturelle Eigenschaften eines Graphen für eine kompakte Speicherung genutzt werden. Dies gilt zum Beispiel für Intervallgraphen. Ein *Intervallgraph* ist durch eine endliche Menge $\mathcal{I}$ von abgeschlossenen Intervallen über der Menge $\mathbb{R}$ bestimmt. Jedes Intervall aus $\mathcal{I}$ entspricht einer Ecke und die zu den Intervallen $I_1, I_2 \in \mathcal{I}$ gehörenden Ecken sind benachbart, wenn die beiden Intervalle überlappen, d. h. $I_1 \cap I_2 \neq \varnothing$. Abbildung 2.20 zeigt ein Beispiel für einen Intervallgraph. Diese Klasse von Graphen wird in Abschnitt 7.6 näher betrachtet.

Abb. 2.20: Beispiel eines Intervallgraphen für eine Menge mit 6 Intervallen



Eine mögliche Darstellung von Intervallgraphen besteht darin, die Intervallgrenzen in einer $2 \times n$ Matrix abzuspeichern. Dazu werden lediglich $2n$ Speicherplätze benötigt. Um festzustellen, ob es eine Kante von e_i nach e_j gibt, muss lediglich überprüft

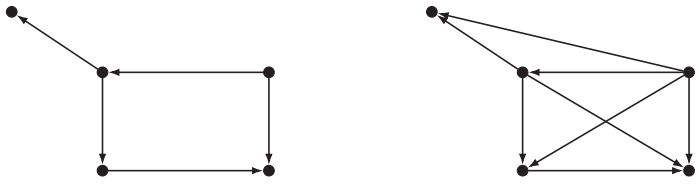
werden, ob sich die entsprechenden Intervalle überschneiden. Um die Nachbarn einer Ecke e_i zu bestimmen, muss ein Intervall mit allen anderen verglichen werden. Durch den Einsatz von Baumstrukturen, wie sie in Kapitel 3 dargestellt werden, erreicht man eine sehr kompakte Darstellung von Intervallgraphen, welche auch Nachbarschaftsabfragen effizient unterstützt. Die Bestimmung der Nachbarn einer Ecke erfordert dabei nicht die Betrachtung aller Intervalle.

2.5 Der transitive Abschluss eines Graphen

Dieser Abschnitt beschreibt einen ersten einfachen Graphalgorithmus. Zur Motivation dieses Algorithmus wird folgendes Problem betrachtet. 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_j wird direkt oder indirekt in die Datei D_i eingefügt, falls es einen Weg von D_i nach D_j 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 e' gibt, falls es in G einen Weg von e nach e' gibt, der aus mindestens einer Kante besteht. Abbildung 2.21 zeigt einen gerichteten Graphen und seinen transitiven Abschluss.

Abb. 2.21: 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 e_i und Endecke e_j, welche aus s Kanten bestehen.*

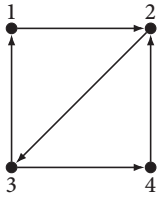
Beweis. Der Beweis erfolgt durch vollständige Induktion nach s. Für $s = 1$ ist die Aussage richtig, denn die Adjazenzmatrix enthält die angegebene Information über Kantenzüge der Länge 1. Die Aussage sei nun für $s > 1$ richtig. Nun sei $\tilde{a}_{il}$ der (i, l)-Eintrag von A^s und a_{lj} der (l, j)-Eintrag von A. Nach Induktionsvoraussetzung ist $\tilde{a}_{il}$ die Anzahl der Kantenzüge mit Anfangsecke e_i und Endecke e_l, welche aus s Kanten bestehen. Dann ist $\tilde{a}_{il}a_{lj}$ die Anzahl der Kantenzüge mit Anfangsecke e_i und Endecke e_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 2.22 zeigt einen gerichteten Graphen und die ersten 3 Potenzen der zugehörigen Adjazenzmatrix.

Abb. 2.22: Ein gerichteter Graph und die Potenzen seiner Adjazenzmatrix



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

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

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 Ecke e_j von Ecke e_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 2.22 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, $G_1, \ldots, G_n$ bezeichnet, wobei $G_0 = G$ ist. Der Übergang von G_l nach G_{l+1} erfolgt, indem für jedes Paar e_i, e_j von Ecken, für die es in G_l noch keine Kante gibt, eine Kante von e_i nach e_j hinzugefügt wird, falls es in G_l Kanten von e_i nach e_l und von e_l nach e_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 $l = 0, 1, \ldots, n$ gilt: In G_l gibt es genau dann eine Kante von e_i nach e_j, wenn es in G einen Weg von e_i nach e_j gibt, welcher nur Ecken aus $E_l = \{e_1, \ldots, e_l\}$ verwendet.*

Beweis. Der Beweis erfolgt durch vollständige Induktion nach l. Für $l = 0$ ist die Aussage richtig, denn $E_0 = \varnothing$. Die Aussage sei nun für $l > 0$ richtig. Gibt es in G_{l+1} eine Kante von e_i nach e_j, so gibt es nach Konstruktion auch in G einen Weg von e_i nach e_j, der nur Ecken aus E_{l+1} verwendet. Umgekehrt ist noch zu beweisen, dass es in G_{l+1} eine Kante von e_i nach e_j gibt, falls es in G einen Weg W gibt, welcher nur Ecken aus E_{l+1} verwendet. Verwendet W nur Ecken aus E_l, so ist die Aussage nach Induktionsvoraussetzung richtig. Verwendet W die Ecke e_{l+1}, so gibt es auch einen einfachen Weg W mit dieser Eigenschaft. Es sei W_1 der Teilweg von e_i nach e_{l+1} und W_2 der von e_{l+1} nach e_j. Nach Induktionsvoraussetzung gibt es somit in G_l Kanten von e_i nach e_{l+1} und von e_{l+1} nach e_j. Somit gibt es nach Konstruktion in G_{l+1} eine Kante von e_i nach e_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 2.23 zeigt die Prozedur `transAbschluss`, welche die Adjazenzmatrix