

SEBASTIAN DÖRN

Programmieren für Ingenieure und Natur- wissenschaftler

Algorithmen und Programmiertechniken



eXamen.press



Springer Vieweg

eXamen.press

eXamen.press ist eine Reihe, die Theorie und Praxis aus allen Bereichen der Informatik für die Hochschulausbildung vermittelt.

Sebastian Dörn

Programmieren für Ingenieure und Naturwissenschaftler

Algorithmen und Programmiertechniken



Springer Vieweg

Sebastian Dörn
Hochschulcampus Tuttlingen
Hochschule Furtwangen
Tuttlingen, Deutschland

ISSN 1614-5216

eXamen.press

ISBN 978-3-662-54175-3

DOI 10.1007/978-3-662-54176-0

ISBN 978-3-662-54176-0 (eBook)

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

Springer Vieweg

© Springer-Verlag GmbH Deutschland 2017

Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung, die nicht ausdrücklich vom Urheberrechtsgesetz zugelassen ist, bedarf der vorherigen Zustimmung des Verlags. Das gilt insbesondere für Vervielfältigungen, Bearbeitungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften. Der Verlag, die Autoren und die Herausgeber gehen davon aus, dass die Angaben und Informationen in diesem Werk zum Zeitpunkt der Veröffentlichung vollständig und korrekt sind. Weder der Verlag noch die Autoren oder die Herausgeber übernehmen, ausdrücklich oder implizit, Gewähr für den Inhalt des Werkes, etwaige Fehler oder Äußerungen. Der Verlag bleibt im Hinblick auf geografische Zuordnungen und Gebietsbezeichnungen in veröffentlichten Karten und Institutionsadressen neutral.

Gedruckt auf säurefreiem und chlorfrei gebleichtem Papier

Springer Vieweg ist Teil von Springer Nature

Die eingetragene Gesellschaft ist Springer-Verlag GmbH Deutschland

Die Anschrift der Gesellschaft ist: Heidelberger Platz 3, 14197 Berlin, Germany

Vorwort

In diesem dreibändigen Werk **Programmieren für Ingenieure und Naturwissenschaftler** werden die Methoden der Programmierung und Algorithmen von ihren Grundlagen bis zu leistungsfähigen Verfahren aus dem Bereich der künstlichen Intelligenz mit einer umfangreichen Auswahl von technisch-naturwissenschaftlichen Anwendungen vorgestellt. In den Ingenieur- und Naturwissenschaften haben diese Verfahren heute durch die zunehmende Digitalisierung einen sehr großen Anwendungsbereich.

Das Ziel dieser Buchreihe ist es, Studierenden der Ingenieur- oder Naturwissenschaften die Programmierung als Schlüsselqualifikation mit zahlreichen Anwendungsmöglichkeiten vorzustellen. Ein großer Wert wird dabei auf eine praxisorientierte und verständliche Darstellung der Arbeitsweise von Algorithmen mithilfe von Schritt-für-Schritt-Anleitungen gelegt. Alle vorgestellten Algorithmen werden von den Grundprinzipien bis zu den Implementierungsdetails genau besprochen. Die Umsetzung von Programmierkonzepten und algorithmischen Verfahren erfolgt dabei in der Programmiersprache **Java**.

Auswahl der Inhalte des zweiten Bands

Aufbauend auf den im ersten Band dargestellten Grundkonzepten von Java, Methoden zum Entwurf von Algorithmen, numerischen Verfahren und diversen Anwendungen aus dem Bereich der Ingenieur- und Naturwissenschaften werden im zweiten Band die folgenden Inhalte behandelt:

- **Anwendungsbereiche:** Operations Research, medizinische Informatik, Automatisierungstechnik.
- **Objektorientierte Entwurfsmuster:** objektorientierte Analyse und Konzepte, Strukturmuster, Verhaltensmuster, Erzeugungsmuster.
- **Zentrale Konzepte:** Programmierkonzepte (Exception, Multithreading, Pakete), Datenbankzugriffe (SQL, XML, Excel), Datenstrukturen (verkettete Listen, Stapel, Warteschlangen, Baumstrukturen, Graphen).
- **Suchen:** Breiten- und Tiefensuche, Dijkstra-Algorithmus, gierige Suche, A*-Suche, Simulated Annealing, genetische Algorithmen.

- **Graphen und Netzwerke:** grundlegende Begriffe, Graphen und Matrizen, minimale aufspannende Bäume, Flüsse in Netzwerken, Tourenprobleme, spezielle Knotenmengen, Färbungen von Knoten.
- **Automaten:** deterministische Automaten, deterministische Ein-/Ausgabe-Automaten, nicht deterministische Automaten und reguläre Sprachen, Grammatiken.
- **Bildverarbeitung:** allgemeine Grundlagen, Bildfilterung, Registrierung, Segmentierung.

Danksagung

Für die Erstellung der Abbildungen zu den einzelnen Kapiteln bin ich Sonja Jenisch, Aline Winter und David Schulz sehr dankbar. Für wertvolle Hinweise und Verbesserungsvorschläge gilt mein Dank an Martina Warmer, Sonja Jenisch, Johannes Jaeger, Lilli Reiner, Martin Haimerl und Mike Fornefett.

Hinweise und Anregungen

Hinweise und Verbesserungsvorschläge sind von allen Seiten sehr willkommen, um eine kontinuierliche Verbesserung dieser Lehrbuchreihe zu erreichen: per Email an sebastian.doern@hs-furtwangen.de.

Tuttlingen, Januar 2017

Sebastian Dörn

Inhaltsverzeichnis

- 1 Anwendungen** 1
 - 1.1 Operations Research 1
 - 1.2 Medizinische Informatik 5
 - 1.3 Automatisierungstechnik 9
 - Literaturhinweise 13
- 2 Objektorientierte Entwurfsmuster** 15
 - 2.1 Einführendes Beispiel 16
 - 2.2 Objektorientierte Analyse 20
 - 2.3 Objektorientierte Konzepte 22
 - 2.3.1 Innere Klassen 23
 - 2.3.2 Polymorphie 27
 - 2.3.3 Abstrakte Klassen 27
 - 2.3.4 Interface 31
 - 2.4 Objektorientierte Muster 39
 - 2.5 Strukturmuster 40
 - 2.5.1 Adapter 40
 - 2.5.2 Fassade 44
 - 2.5.3 Dekorierer 47
 - 2.6 Verhaltensmuster 52
 - 2.6.1 Schablone 52
 - 2.6.2 Strategie 55
 - 2.6.3 Beobachter 58
 - 2.7 Erzeugungsmuster 61
 - 2.7.1 Singleton 62
 - 2.7.2 Fabrik 64
 - 2.8 Übungsaufgaben 67
 - Literaturhinweise 69

3	Zentrale Konzepte	71
3.1	Programmierkonzepte	71
3.1.1	Exception	71
3.1.2	Multithreading	77
3.1.3	Pakete	85
3.2	Datenbankzugriffe	87
3.2.1	SQL	87
3.2.2	XML	94
3.2.3	Excel	98
3.3	Datenstrukturen	101
3.3.1	Listen, Mengen und Hash-Tabellen	101
3.3.2	Verkettete Listen	107
3.3.3	Stapel und Warteschlangen	111
3.3.4	Baumstrukturen	114
3.3.5	Graphen	119
3.4	Übungsaufgaben	122
	Literaturhinweise	123
4	Suchen	125
4.1	Allgemeine Grundlagen	126
4.2	Breiten- und Tiefensuche	129
4.2.1	Breitensuche	129
4.2.2	Tiefensuche	132
4.3	Dijkstra-Suche	134
4.3.1	Einführendes Beispiel	134
4.3.2	Problemstellung	135
4.3.3	Grundlegende Lösungsprinzipien	136
4.3.4	Algorithmus und Implementierung	139
4.3.5	Anwendungen	141
4.4	Gierige Suche	143
4.4.1	Einführendes Beispiel	143
4.4.2	Problemstellung	144
4.4.3	Grundlegende Lösungsprinzipien	144
4.4.4	Algorithmus und Implementierung	145
4.4.5	Anwendungen	147
4.5	A*-Suche	147
4.5.1	Einführendes Beispiel	147
4.5.2	Problemstellung	147
4.5.3	Grundlegende Lösungsprinzipien	148
4.5.4	Algorithmus und Implementierung	149
4.5.5	Anwendungen	152

4.6	Simulated Annealing	155
4.6.1	Einführendes Beispiel	155
4.6.2	Problemstellung	157
4.6.3	Grundlegende Lösungsprinzipien	158
4.6.4	Algorithmus und Implementierung	160
4.6.5	Anwendungen	166
4.7	Genetische Algorithmen	168
4.7.1	Grundlegende Begriffe	168
4.7.2	Problemstellung	168
4.7.3	Grundlegende Lösungsprinzipien	169
4.7.4	Algorithmus und Implementierung	171
4.7.5	Anwendungen	177
4.8	Übungsaufgaben	179
	Literaturhinweise	181
5	Graphen und Netze	183
5.1	Grundlegende Begriffe	184
5.1.1	Graphen und Untergraphen	184
5.1.2	Kreise und Wege	187
5.1.3	Spezielle Graphen	188
5.1.4	Graphoperationen	190
5.2	Graphen und Matrizen	190
5.2.1	Matrizendarstellung	190
5.2.2	Listendarstellung	192
5.2.3	Abstandsmatrix	193
5.2.4	Gerüste	194
5.3	Minimal aufspannende Bäume	196
5.3.1	Einführendes Beispiel	196
5.3.2	Problemstellung	198
5.3.3	Grundlegende Lösungsprinzipien	198
5.3.4	Algorithmus und Implementierung	200
5.3.5	Anwendungen	202
5.4	Flüsse in Netzen	203
5.4.1	Einführendes Beispiel	203
5.4.2	Problemstellung	204
5.4.3	Grundlegende Lösungsprinzipien	204
5.4.4	Algorithmus und Implementierung	208
5.4.5	Anwendungen	210
5.5	Tourenprobleme	213
5.5.1	Einführendes Beispiel	213
5.5.2	Problemstellung	214

5.5.3	Grundlegende Lösungsprinzipien	215
5.5.4	Algorithmus und Implementierung	217
5.5.5	Anwendungen	218
5.6	Spezielle Knotenmengen	220
5.6.1	Einführendes Beispiel	220
5.6.2	Problemstellung	221
5.6.3	Grundlegende Lösungsprinzipien	223
5.6.4	Algorithmus und Implementierung	226
5.6.5	Anwendungen	230
5.7	Färbungen von Knoten	230
5.7.1	Einführendes Beispiel	230
5.7.2	Problemstellung	231
5.7.3	Grundlegendes Lösungsprinzipien	233
5.7.4	Algorithmus und Implementierung	234
5.7.5	Anwendung	235
5.8	Übungsaufgaben	237
	Literaturhinweise	239
6	Automaten	241
6.1	Deterministische Automaten	241
6.1.1	Einführende Beispiele	241
6.1.2	Grundlegende Begriffe	244
6.1.3	Problemstellung	249
6.1.4	Grundlegende Lösungsprinzipien	250
6.1.5	Algorithmus und Implementierung	254
6.1.6	Anwendungen	259
6.2	Deterministische E/A-Automaten	265
6.2.1	Einführendes Beispiel	265
6.2.2	Grundlegende Begriffe	265
6.2.3	Problemstellung	267
6.2.4	Grundlegende Lösungsprinzipien	267
6.2.5	Algorithmus und Implementierung	268
6.2.6	Anwendungen	270
6.3	Nicht deterministische Automaten und reguläre Ausdrücke	273
6.3.1	Einführende Beispiele	273
6.3.2	Grundlegende Begriffe	273
6.3.3	Problemstellung	277
6.3.4	Grundlegende Lösungsprinzipien	277
6.3.5	Algorithmus und Implementierung	282
6.3.6	Anwendungen	286

6.4	Grammatiken	288
6.4.1	Einführendes Beispiel	288
6.4.2	Grundlegende Begriffe	289
6.4.3	Problemstellung	292
6.4.4	Grundlegende Lösungsprinzipien	293
6.4.5	Algorithmus und Implementierung	296
6.4.6	Anwendungen	301
6.5	Übungsaufgaben	304
	Literaturhinweise	305
7	Bildverarbeitung	307
7.1	Allgemeine Grundlagen	308
7.2	Bildfilterung	313
7.2.1	Problemstellung	313
7.2.2	Grundlegende Lösungsprinzipien	314
7.2.3	Algorithmus und Implementierung	321
7.2.4	Anwendungen	326
7.3	Registrierung	327
7.3.1	Problemstellung	328
7.3.2	Grundlegende Lösungsprinzipien	329
7.3.3	Algorithmus und Implementierung	333
7.3.4	Anwendungen	336
7.4	Segmentierung	337
7.4.1	Problemstellung	337
7.4.2	Grundlegende Lösungsprinzipien	338
7.4.3	Algorithmus und Implementierung	343
7.4.4	Anwendungen	347
7.5	Spezielle Anwendung	348
7.6	Übungsaufgaben	352
	Literaturhinweise	357
	Anhang	359
	Sachverzeichnis	369

Inhaltsverzeichnis des Bandes Programmieren für Ingenieure und Naturwissenschaftler – Grundlagen

- 1 Grundbegriffe der Programmierung**
- 2 Strukturelle Programmierung**
- 3 Entwicklung von Computerprogrammen**
- 4 Numerische Algorithmen**
- 5 Entwurfsmuster von Algorithmen**
- 6 Objektorientierte Programmierung**
- 7 Graphische Benutzeroberflächen**
- 8 Technische und naturwissenschaftliche Anwendungen**

Die in diesem Buch beschriebenen Programmiertechniken und algorithmischen Verfahren, in Form von Suchtechniken, Graphen, Automaten und Bildverarbeitung kommen in den verschiedensten Bereichen, in den Natur- und Ingenieurwissenschaften vor. In diesem Kapitel stellen wir überblickartig einige dieser Gebiete genauer vor. Konkret sind das die Fachdisziplinen des Operations Research, der medizinischen Informatik und der Automatisierungstechnik.

1.1 Operations Research

Operations Research ist ein interdisziplinäres Wissensgebiet zwischen angewandter Mathematik, Informatik und Wirtschaftswissenschaften. Unter dem Begriff Operations Research wird die Entwicklung und Anwendung von mathematischen Modellen und Methoden zur Entscheidungsfindung verstanden. Die Methoden des Operations Research werden heute in vielen unterschiedlichen Bereichen in den Ingenieurwissenschaften, in den Wirtschaftswissenschaften, in der Wirtschaftsinformatik und auch in den Naturwissenschaften verwendet.

Das Prinzip des Handelns lässt sich als einen Prozess mit den vier Phasen Planung, Entscheidung, Durchführung und Kontrolle beschreiben. Unter der Planung versteht man die systematische Vorgehensweise zur Analyse und Lösung von technischen, wirtschaftlichen oder naturwissenschaftlichen Problemstellungen. Die Entscheidungsfindung ist eng mit der Planungsphase verbunden, da im Laufe der einzelnen Teilprozesse der Planung zahlreiche Entscheidungen zu treffen sind. Die Durchführung beschreibt die Umsetzung der einzelnen Planungs- bzw. Entscheidungsschritte, die am Ende mit der Kontrolle des gesetzten Ziels verbunden ist.

Historische Entwicklung und Perspektiven Ursprünglich stammt der Begriff des Operations Research aus dem Militärwesen. Im Jahre 1937 gab es in Großbritannien eine Gruppe von Wissenschaftlern, die sich mit dem optimalen Aufbau eines Radarüberwachungssystems zur Abwehr feindlicher Flugzeuge beschäftigte. Zu Beginn des 2. Weltkrieges wurde die Abteilung „Operational Research“ in der britischen und amerikanischen Armee eingerichtet. Zu den zentralen Aufgaben gehörte, eine optimale Strategie im Luftkampf und eine bestmögliche Zusammensetzung von Schiffskonvois mit Begleitschutz vor deutschen U-Booten zu ermitteln.

Durch die Erfolge dieser Disziplin im 2. Weltkrieg wurde das Prinzip des Operations Research auch auf andere Disziplinen in Wirtschaft, Technik und Gesellschaft übertragen. Das grundlegende Ziel ist, einen Planungs- und Entscheidungsprozess zu modellieren, sodass ein gewünschtes Ergebnis mit den geringsten Kosten erreicht wird.

Durch den Fortschritt in der Informatik entwickelte sich das Gebiet des Operations Research sehr schnell. Heute überwiegen vor allem ökonomische und ingenieurwissenschaftliche Anwendungen, insbesondere in den Bereichen der Logistik und Produktion. In der Industrie gewinnt dieses Fachgebiet durch das Aufkommen der sogenannten Industrie 4.0 einen enorm breiten Anwendungsbereich. Der Begriff Industrie 4.0 beschreibt die vollständige Durchdringung der industriellen Produkte und Dienstleistungen von Software, mit einer gleichzeitigen Vernetzung und Informationsverarbeitung, mit dem Ziel der Ausschöpfung aller Optimierungspotenziale.

Seit einiger Zeit gewinnt die Disziplin des Operations Research auch in den Naturwissenschaften und der Medizin an Bedeutung. Der Grund dafür ist die fortschreitende Digitalisierung und die damit verbundene mathematische Modellbildung der relevanten Entscheidungsprobleme, die aufgrund der riesigen Datenmengen einer immer größer werdenden Komplexität unterliegen. Methoden des Operations Research kommen hier für vielfältigste Anwendungen zum Einsatz wie beispielsweise im Rahmen der Entscheidungsfindung zur Behandlung von Patienten, unter dem Einsatz optimaler medizinischer Therapien.

Bei der Lösung praktischer Probleme auf diesem Gebiet sind Fachleute verschiedenster Disziplinen gefragt. Interdisziplinäre Teams aus Mathematikern zur mathematischen Modellbildung, Informatikern für die Entwicklung leistungsfähiger Algorithmen und Natur- bzw. Wirtschaftswissenschaftler des jeweiligen praktischen Anwendungsgebietes sind der Schlüssel für einen erfolgreichen Planungs- und Entscheidungsprozess. Operations Research bietet in der heutigen Zeit durch seine vielfältigsten Anwendungsbereiche in unterschiedlichsten Bereichen ein hochinteressantes Tätigkeitsfeld für Ingenieure und Naturwissenschaftler, die sich mit den Methodiken in dieser Wissensdisziplin auskennen.

Grundlegender Ablauf Operations Research befasst sich mit der Analyse von praxisrelevanten Problemstellungen im Rahmen eines Planungsprozesses. Die optimale Entscheidungsfindung wird auf Basis von mathematischen Modellen und dazugehörigen algorithmischen Lösungsverfahren durchgeführt. Die Hauptaufgaben bestehen in der Entwicklung

eines Optimierungsmodells für ein Entscheidungsproblem sowie in der Konstruktion und Anwendung eines passenden Algorithmus zur Lösung dieses Problems.

Die einzelnen Schritte im Operations Research können wie folgt gegliedert werden:

1. **Analyse des Problems:** In einem technischen, wirtschaftlichen oder naturwissenschaftlichen Bereich wird ein gewisser Entscheidungsbedarf erkannt.
2. **Bestimmung von Zielen:** Eine Menge von Zielen und Randbedingungen im zugehörigen Anwendungsbereich wird definiert.
3. **Aufstellung eines mathematischen Modells:** Das vorliegende konkrete, reale System bzw. der Prozess wird durch ein vereinfachtes Abbild in Form eines mathematischen Modells beschrieben.
4. **Beschaffung der Daten:** Für das mathematische Modell müssen geeignete Datenmengen beschafft werden.
5. **Bestimmung der Lösung:** Mithilfe eines geeigneten Algorithmus wird das aufgestellte mathematische Modell unter Verwendung aller vorhandenen Daten optimal gelöst.
6. **Bewertung der Lösung:** Die erhaltene Lösung wird in Bezug auf deren praktische Verwendung und Plausibilität analysiert und bewertet.

Je nach Zielerreichungsgrad müssen diese einzelnen Schritte gegebenenfalls mehrfach durchlaufen werden. Die zugehörigen mathematischen Modelle hängen von den jeweiligen Anwendungsgebieten ab, die wir im Folgenden kurz vorstellen.

Lineare und nicht lineare Optimierung Die Modelle der linearen Optimierung bestehen aus einer oder mehreren linearen Zielfunktionen und einer Vielzahl von linearen Nebenbedingungen in Form von Gleichungen bzw. Ungleichungen mit reellen bzw. positiven Variablen. Bei nicht linearen Modellen kommt entweder eine nicht lineare Zielfunktion bzw. mindestens eine nicht lineare Nebenbedingung vor. Die lineare Optimierung besitzt mit dem Simplex-Algorithmus ein effizientes Standardlösungsverfahren. Für nicht lineare Optimierungsprobleme gibt es hingegen eine ganze Reihe unterschiedlichster problem-spezifischer Lösungstheorien. Anwendungsbereiche findet die lineare bzw. nicht lineare Optimierung in den folgenden Gebieten:

- Aufstellung von optimalen Produktionsplänen in der industriellen Fertigung,
- Modellierung von Reihenfolgeproblemen in Form von Maschinenbelegungsplänen bzw. Auslieferungsplänen,
- Aufstellung von Zuordnungsplänen von Arbeitskräften bzgl. Tätigkeiten oder Maschinen bzgl. Ressourcen,
- Erstellung einer optimalen Lagerhaltung in Bezug auf Bestellmenge, Bestellzeitpunkt und Lagermenge.

Graphentheorie und Netzplantechnik Mit Hilfsmitteln der Graphentheorie lassen sich viele Organisationsstrukturen oder Projektablaufe grafisch anschaulich darstellen und analysieren. Die Netzplantechnik ist eine in der Praxis am häufigsten eingesetzte Methode zur

Planung, die der Überwachung und Kontrolle von betrieblichen Abläufen und Projekten dient. Anwendungsbereiche besitzen die Methoden der Graphentheorie und Netzplantechnik in den folgenden Gebieten:

- Struktur- und Zeitplanung von umfangreichen und komplexen Projekten.
- Modellierung von Transportproblemen im Bereich der Logistik oder der Fertigungsplanung.
- Aufstellung von Zuordnungsplänen von Arbeitskräften bzgl. Tätigkeiten oder Maschinen bzgl. Ressourcen.
- Unterstützung von Managemententscheidungen im Bezug auf Standortplanung oder Mitarbeiterinsatz.

Kombinatorische Optimierung Viele kombinatorische Optimierungsprobleme lassen sich mathematisch als ganzzahlige oder binäre Optimierungsprobleme formulieren. Bei der ganzzahligen Optimierung dürfen die Variablen nur ganze Zahlen oder Binärzahlen annehmen. Diese Modelle spielen vor allem bei zahlreichen Zuordnungsproblemen eine große Rolle. Kombinatorische Optimierungsprobleme besitzen in der Regel keine effizienten Lösungsverfahren, sondern können nur mit heuristischen Suchverfahren bearbeitet werden. Das Einsatzgebiet der kombinatorischen Optimierung umfasst die folgenden Anwendungsmöglichkeiten:

- Aufstellung von Zuordnungsplänen von Maschinen zu Plätzen mit minimalen Transportkosten,
- Modellierung von Maschinenbelegungsplänen unter zahlreichen Nebenbedingungen,
- strukturelle Planung von Gruppierungsproblemen für ähnliche Aufträge oder Kunden,
- Planung einer optimalen Auswahl bei Transport- oder Produktionsproblemen.

Warteschlangentheorie Die Warteschlangentheorie dient der Untersuchung des Abfertungsverhaltens von Service- und Bedienungsstationen. Ein Beispiel ist die Produktionsplanung mit der Untersuchung von Warteschlangen vor Fertigungsmaschinen. Das Ziel ist, die Maschinenkapazitäten zu optimieren und hohe Kapitalbindungskosten zu vermeiden. Der Einsatz von Warteschlangenmodellen bietet sich in den folgenden Bereichen an:

- Modellierung von Warteschlangenproblemen bei Bank- oder Behördenschaltern,
- Entwurf eines effizienten Verteilungssystems von Produktionsmengen,
- Entwicklung von Softwaresystemen für ein optimales Lieferkettenmanagement,
- Unterstützung von Managemententscheidungen in Bezug auf Preis- und Kapazitätssteuerung.

In diesem Band werden die beiden zentralen Teilgebiete der Graphentheorie- und Netzplantechnik sowie Verfahren zur Lösung kombinatorischer Optimierungsprobleme in Form von (heuristischen) Suchalgorithmen behandelt. Warteschlangenprobleme werden ausführlich im 3. Band im Rahmen von stochastischen Automaten vorgestellt.

1.2 Medizinische Informatik

Die Verwendung von Konzepten und Methoden der Informatik bekommt in der Medizintechnik eine immer größere Bedeutung. Ohne systematische Informationserfassung und Informationsverarbeitung ist die moderne Medizin von heute undenkbar. Beispiele sind Verfahren der digitalen Bildverarbeitung, Datenbanksysteme zur Verwaltung von Patientendaten oder digitale Diagnose- bzw. Therapieunterstützungssysteme.

Die medizinische Informatik hat sich seit den 1970er-Jahren als eigenständige Fachdisziplin herausgebildet. Dieses Fachgebiet umfasst unter anderem die Gestaltung von informationsverarbeitenden Systemen in der Medizin und im Gesundheitswesen bzw. in der Entwicklung von Softwareanwendungen für medizinische Geräte im Bereich der Diagnose und Therapie. Die in diesen Geräten erzeugten Informationen sind mittlerweile so umfangreich, dass sie nur mit einer automatisierten Analyse durch eine algorithmische Auswertung einen praktisch verwendbaren Nutzen finden.

Die einzelnen Teilgebiete der medizinischen Informatik sind sehr umfangreich und umfassen im Wesentlichen die Bereiche medizinische Dokumentation und Qualitätsmanagement, medizinische Informations-, Kommunikationssysteme und Datenverarbeitungssysteme, medizinische Bild- und Signalverarbeitung, medizinische Experten- und KI-Systeme, medizinische Robotik und computerunterstützte Chirurgie, medizinische Statistik und Datenanalyse, Modellbildung biologischer Prozesse und Bioinformatik.

Die Ziele der medizinischen Informatik im Bereich der digitalisierten Anwendungen sind die Folgenden:

- neue Therapie- und Unterstützungssysteme für Ärzte und Patienten,
- Erhöhung der Effizienz im Bereich der Patientenversorgung,
- Verbesserung der Leistungsfähigkeit des Gesundheitssystems,
- Integration von Medizinprodukten in den allgemeinen Versorgungszyklus.

Mit der Verschmelzung der Medizintechnik- und der IT-Branche werden neuartige Innovationen im Produkt-, Prozess oder Dienstleistungsbereich ermöglicht. Die medizinische Informatik ist dabei eine Schlüsseldisziplin mit einem sehr starken interdisziplinären Charakter:

- Informatik (z. B. Softwaretechnik, Algorithmik, Datenbanken)
- Mathematik (z. B. Modellbildung und Simulation, Statistik)
- Ingenieurwissenschaften (z. B. Mechatronik, Robotik)
- Naturwissenschaften (z. B. humanbiologische Prozesse, Strömungsmodelle)
- Wirtschafts- und Rechtswissenschaften (z. B. Budgetierung, Medizinprodukterecht)

Historische Entwicklung Die Geschichte der medizinischen Informatik, zu der auch die medizinische Dokumentation gehört, ist prinzipiell so alt wie die der Medizin. Natürlich findet der eigentliche Einsatz der Rechentechnik erst in den letzten 50 Jahren statt. Wir stellen einige der wichtigsten historischen Entwicklungen auf diesem Gebiet vor:

1700 v. Chr.	In der <i>Papyrus Smith</i> sind die ersten Zeugnisse medizinischer Dokumentationen zur chirurgischen Wundbehandlung der alten Ägypter beschrieben
400 v. Chr.	Hippokrates warb für die Schreibung von Krankengeschichten in der Medizinschule auf der Insel Kos
11. Jh.	Konstantin von Afrika übersetzte die medizinischen Schriften aus dem Arabischen und gründete die Medizinschule von Salerno
16. Jh.	Nürnberger Stadtarzt Johannes Magenbuch dokumentiert die ersten Krankengeschichten seiner Patienten, darunter auch Martin Luther
19. Jh.	In Krankenhäusern existieren Aufnahmebücher und Krankengeschichten der Patienten, mit denen Krankenhausstatistiken angefertigt wurden
1887	Anfertigung der ersten Elektrokardiogramm (EKG)-Auswertungen
Um 1894	Einsatz von Lochkartenmaschinen zur Erstellung von Medizinalstatistiken
Um 1930	Erstellung von ersten medizinischen Statistiken in Krankenhäusern
Um 1956	Verwendung erster Krankenhausinformationssysteme und Speicherung von Patientendaten auf Maschinenlochkarten
Um 1970	Einsatz von Computern zur Speicherung von Patientendaten in Krankenhäusern sowie zur Auswertung von Biosignalen und Bildauswertung
Um 1997	Entwicklung des sogenannten E-Health als Sammelbegriff für den Einsatz digitaler Technologien im Gesundheitswesen (z. B. elektronische Gesundheitsakte, Telemedizindienste, Gesundheitsportale oder Onlineapotheken)
Um 2007	Fortschreitendes Aufkommen von mobilen Anwendungen im Gesundheitswesen zur persönlichen und dezentralen Gesundheitsfürsorge im Bereich der Überwachung, Diagnose oder Beratung
Um 2013	Google investiert massiv im Gesundheitsbereich (z. B. Cloud-Dienst für DNA-Daten, Erforschung von altersbedingten Krankheiten, Kontaktlinse zur Messung des Blutzuckers durch Augenflüssigkeit oder Nanopartikel zur Suche nach Krankheiten im Körper)

Wir stellen im Folgenden einige der zentralen Aufgabenfelder im Bereich der medizinischen Informatik genauer vor.

Medizinische Informationssysteme und Dokumentation Medizinische Informationssysteme umfassen die Gesamtheit aller informationsverarbeitenden Systeme zur Erfassung, Bearbeitung und Weitergabe medizinischer und administrativer Daten. In Krankenhäusern finden sogenannte Krankenhausinformationssysteme (KIS) eine vielfältige Anwendung. Eine Arztpraxissoftware stellt das Äquivalent eines Krankenhausinformationssystems zur Verwaltung, Organisation und Betrieb von Arztpraxen. Das Aufgabengebiet dieser Softwaresysteme besteht in der Optimierung von Arbeitsabläufen und in der Einsparung von Kosten.

Die medizinische Dokumentation beschreibt die Merkmale von Krankheitsbildern und deren entsprechende Behandlungsmethoden. Zur medizinischen Dokumentation gehören unter anderem Diagnosedaten, Labordaten und Bildmaterial. Diese Art der Dokumenta-

tion dient unter anderem auch zur Nachvollziehbarkeit von Abläufen in medizinischen Einrichtungen und damit gleichzeitig zur juristischen Absicherung von Ärzten und Pflegekräften.

Neben der Gewinnung neuer Informationen über Krankheitsbilder und der Behandlungsmethoden zur Verbesserung von Therapien sind noch die folgenden Ziele und Aufgaben von Bedeutung:

- Organisation eines reibungslosen Ablaufs in Krankenhäusern und Arztpraxen,
- optimales Qualitätsmanagement in der medizinischen Versorgung,
- Verwaltung von Patientendaten, Korrespondenzen, Bildmaterial, Diagnosen und Laborwerten,
- logistische Unterstützung zu Materialbestellungen und des Verbrauchsflusses,
- Verbesserung der Kommunikation zwischen den Mitarbeitern,
- Buchhaltungssysteme mit Statistik-, Such- und Auswertungsfunktionen,
- Abrechnungssysteme mit Krankenkassen, Erstellung von Rechnungen, Formularwesen.

Medizinische Bildverarbeitung In der medizinischen Bildverarbeitung werden mithilfe von algorithmischen Verfahren aus medizinischen Bilddaten Informationen extrahiert, die für die medizinische Diagnostik eine gewisse Relevanz besitzen. In der Radiologie und inneren Medizin wird eine große Menge von bildgebenden Verfahren eingesetzt wie beispielsweise Röntgen, Computertomografie (CT) oder Magnetresonanztomografie (MRT). In der Endoskopie bzw. Dermatoskopie werden durch digitale Kameras Bilder aufgenommen und anschließend analysiert. Bei diesen Anwendungen spielen die Bildverbesserung, Bildrekonstruktion, Bildanalyse, Segmentierung und die 3-D-Visualisierung eine bedeutende Rolle.

Diese Form der rechnergestützten bildbasierten Diagnose und Therapie für IT-gestützte Expertensysteme bietet eine großartige Perspektive zur besseren und schnelleren Diagnose und Behandlung von Krankheiten. Beispielsweise benutzen Mediziner Softwaresysteme zur Analyse von Bilddatensätzen auf Krebsherden zur Steigerung der erfolgreichen Behandlungsquote durch den automatischen Vergleich mit früheren Fällen als auch zur Verringerung der hohen Arbeitsbelastung. Weitere Anwendungsbereiche liegen in dem Austausch von medizinischen Bilddaten und Dokumenten über digitale Plattformen, um den behandelnden Ärzten jederzeit alle Informationen zur Verfügung zu stellen.

Digitalisierung und Big Data Die Digitalisierung ist heute einer der größten Trends und Treiber für eine Vielzahl von Innovationen, insbesondere im Bereich der mobilen Anwendungen. Unter Digitalisierung versteht man im Allgemeinen die Überführung analoger Größen in diskrete Werte, um diese anschließend zu speichern und mithilfe von Algorithmen zu analysieren. Heute werden Patienten in der Regel in getrennten Systemen behandelt, die untereinander nicht kompatibel sind, sodass kein effizienter Informationsfluss möglich ist.

Für das Gesundheitssystem bietet die Digitalisierung eine große Anzahl von neuen Möglichkeiten, mit denen sich die Patientenversorgung und die Leistungsfähigkeit im Gesundheitssystem erheblich effizienter gestalten lässt. Damit lassen sich schnellere und flexiblere medizinische Geräte, Kommunikationstechnologien, Informations- und Datenmanagementsysteme für medizinische Einrichtungen oder Patienten entwickeln. Beispiele für digitale Systeme, die mithilfe leistungsfähiger und intelligenter Softwarekomponenten arbeiten, sind die folgenden:

- **Digitale medizinische Geräte:** Gesundheits-Monitoring- und Assistenzsysteme für die Überwachung des Gesundheitszustandes und zur Förderung der Therapietreue und der Prävention.
- **Onlinedienste zu Konsultation, Diagnostik und Therapie:** Erweiterung von medizinischen Angeboten durch Onlinesprechstunden und Patienten-Apps für verschiedene Arten von Erkrankungen im Rahmen einer schnellen fachärztlichen Behandlung, insbesondere in ländlichen Regionen.
- **Digitale Datenplattformen:** Austausch, Rückverfolgung und Dokumentation von medizinischen Daten und klinischem Equipment über digitale Plattformen liefert die Grundvoraussetzung für eine effiziente Behandlung.
- **Digitale Versorgungs- und Fertigungsketten:** Verbindung digitalgestützter Therapien mit digitalen Fertigungsketten zur optimalen Organisation von Abläufen über Sektorgrenzen hinweg, um damit den gesamten Lebenszyklus von Medizinprodukten abzubilden.
- **Vorhersage von Infektionskrankheiten:** Nutzung von riesigen Datenbeständen aus Suchmaschinen und Arztpraxen, um die Behörden auf Epidemien vorzuwarnen und Kosten im Gesundheitssystem einzusparen.

Große Datenmengen (Big Data) besitzen wertvolle Informationen über Krankheitsverläufe und Nebenwirkungen, die gezielt zur Entwicklung neuer medizintechnischer Produkte und Therapien einsetzbar sind. Viele biologische Phänomene lassen sich nicht auf einfache allgemeingültige Zusammenhänge zurückführen, sodass die klassischen Methoden aus klinischen Studien der biomedizinischen Komplexität kaum gewachsen sind.

Medizinische Experten- und KI-Systeme Medizinische Expertensysteme sind komplexe, wissensbasierte Softwarepakete, die Expertenwissen aus einem speziellen Gebiet speichern, verwalten und mit Methodiken aus der künstlichen Intelligenz auswerten. Der Nutzen von medizinischen Expertensystemen beruht auf der schnelleren Diagnose von Krankheiten, um so zielgerichtete Therapien zu verordnen. Krankenhäuser und Krankenkassen können damit erhebliche Kosten sparen, indem sie ähnliche Krankheitsbilder miteinander vergleichen, die optimalen Behandlungsabläufe auswählen, um so Falschdiagnosen so weit wie möglich zu vermeiden. Damit ist das gesamte verfügbare medizinische Wissen effizient nutzbar, um so mithilfe von Algorithmen nach Mustern zu suchen. Weitere Anwendungen dieser Systeme im Bereich der Medizin sind die folgenden:

- Vorhersage der Wahrscheinlichkeit für eine erfolgreiche Therapie,
- Prognose der Anzahl der Tage, die ein Patient im Krankenhaus verbringt,
- Vorhersage einer Frühgeburt durch spezielle Biomarker im Blut,
- Vorhersage der Komplikationswahrscheinlichkeit einer medizinischen Therapie,
- Entdeckung von Abrechnungsfehlern,
- Prognose von Gesundheitsrisiken.

In diesem Band stellen wir mit der Bildverarbeitung ein wesentliches Teilgebiet der medizinischen Informatik vor. Die vorgestellten objektorientierten Entwurfsmuster lassen sich zudem sehr effizient für die Entwicklung komplexer Softwareprodukte wie medizinische Anwendungs-, Dokumentations- und Informationssysteme verwenden. Die Automatenmodelle stellen zudem eine einfache Form von Dialogsystemen vor, die ebenso für zahlreiche medizinische Aufgabengebiete verwendbar sind. Der Anwendungsbereich der Digitalisierung und die Methodiken der künstlichen Intelligenz werden sehr ausführlich im 3. Band dieser Buchreihe vorgestellt. An dieser Stelle beschränken wir uns zunächst auf ausgewählte heuristische Suchmethodiken, die für viele dieser Systeme von essenzieller Bedeutung sind.

1.3 Automatisierungstechnik

Die Automatisierungstechnik beschäftigt sich mit der Überwachung und Steuerung technischer Systeme, sodass diese selbstständig ihre zu erfüllenden Aufgaben erkennen und unter Beachtung aller Sicherheitsanforderungen ausführen. Die Kernbereiche der Automatisierungstechnik sind mathematische Modelle, mit denen sich die technischen Systeme beschreiben und bezüglich deren Eigenschaften und Verhaltensweisen analysieren lassen. Das gesamte technische System lässt sich auf dieser Grundlage mithilfe von algorithmischen Verfahren steuern.

Bei der Überwachung werden die wichtigsten Prozessgrößen eines Systems von Sensoren gemessen und von Algorithmen ausgewertet, um damit die nicht messbaren Größen vorherzusagen, eventuelle Fehler frühzeitig zu erkennen und eine Fehlerdiagnose zu erstellen. Bei der Steuerung wird das Verhalten des Systems so beeinflusst, dass die darin ablaufenden Prozesse trotz gewisser Störungen auf den vordefinierten Kennwerten gehalten werden.

Die Automatisierungstechnik ist eine interdisziplinäre Wissenschaft zwischen Maschinenbau und IT, mit einem immer stärkeren Einfluss aus der Informatik in Form von Programmiertechniken (z. B. Objektorientierung, Echtzeitprogrammierung), theoretischer Informatik (z. B. Modellstrukturen, Analysetechniken), Algorithmik (z. B. evolutionäre Verfahren, Randomisierung), Datenbanken (z. B. Speicherstrukturen, Big Data) und Netzwerken (z. B. Kommunikationsprotokolle).

Zentrale Ziele Die Automatisierungstechnik ist für eine leistungsfähige Industrie mit modernen Produktionsbetrieben von zentraler Bedeutung. Die wesentlichen Ziele für deren Einsatz sind unter anderem:

- Ersetzung des Menschen durch eine Automatisierungseinrichtung,
- Verkürzung der Durchlaufzeit von Produktionsprozessen,
- Erhöhung der Zuverlässigkeit, Sicherheit und Qualität,
- Erhöhung der Flexibilität des Fertigungsprozesses,
- Erhöhung des Auslastungsgrads der Produktionseinrichtungen,
- Einsparung von Personal, Material, Energie und Lagerkosten,
- Verbesserung der Termintreue und Lieferfähigkeit.

Historische Entwicklung Die Geschichte der Automatisierung ist von großen Durchbrüchen und Entwicklungen aus verschiedensten Wissenszweigen der Technik geprägt. Einige der bedeutendsten Meilensteine auf diesem Gebiet sind die folgenden:

1785	Edmond Cartwright erfand den ersten automatisierten Webstuhl, der zum erstmaligen Verlust von Arbeitsplätzen durch die Automatisierung führte
1788	Weiterentwicklung der Dampfmaschine durch James Watt in Form von mechanischen Reglern und Sicherheitsventilen, die zum Beginn der industriellen Revolution führte
1833	Samuel Morse entwickelte den ersten elektromagnetischen Schreibtelegraphen, der mit dem ersten Fernschreiber zu einem Durchbruch in der Kommunikation führte
1941	Konrad Zuse entwickelte den Z3, den ersten frei programmierbaren Computer, und damit die Basis für die Automatisierung von Rechenprozessen
1947	Norbert Wiener prägte den Begriff der Kybernetik – die Wissenschaft der Steuerung und Regelung, die Rückkoppelungsmechanismen in verschiedenen Arten von Maschinen, Organismen oder sozialen Organisationen untersucht
1953	John W. Backus entwickelte bei IBM die erste höhere Programmiersprache Fortran, die für numerische Berechnungen in Wissenschaft und Technik große Bedeutung erlangte
1965	Das Stanford Research Institut entwickelte mit Shakey den ersten teilautonomen Roboter
1969	Richard Morley stellte die erste speicherprogrammierbare Steuerung (SPS) zur Steuerung bzw. Regelung einer Maschine auf Basis der Programmierung vor
1970	Die Firma Texas Instruments entwickelte den ersten Mikroprozessor, der die Grundlage für die schnelle Entwicklung der Computertechnik darstellte
1977	Einführung des europäischen EAN-Barcode-Systems (European Article Number), das mithilfe von Scannern gelesen wird und das Fundament für die automatische Warensteuerung und Logistik ist

- 1995 Durch die USA wird das erste voll funktionsfähige satellitengestützte Ortungssystem (GPS) in Betrieb genommen, das heute die Grundlage für das autonome Fahren darstellt
- 1998 Die Firma iRobots entwickelt mit dem PackBot einen kettengetriebenen Roboter, der unter anderem Treppen erklimmen kann
- Um 2006 Die Entwicklung von sogenannten Deep-Learning-Anwendungen führt in vielen Industriezweigen zusammen mit großen Datenmengen zu einer verstärkten Automatisierung von strukturierten Prozessen
- Um 2011 Der Begriff Industrie 4.0 prägt die Entwicklungen von automatisierten Produktionseinrichtungen in Form von intelligenten Softwaresystemen für die Kommunikation zwischen Maschinen, Produktionsmitteln und Produkten.

Zentrale Anwendungsbereiche Die Automatisierungstechnik ist heute eine der wichtigsten Gebiete der Ingenieurwissenschaften mit einer Vielzahl von Anwendungsbereichen. Beispiele für die Automatisierung sind die Steuerung von Produktionsanlagen, die Unterstützung des Autofahrers durch Fahrerassistenzsysteme oder auch die automatische Regulierung von Klimaanlage. Die einzelnen Bereiche lassen sich dabei wie folgt unterteilen:

- **Prozessautomatisierung:** Realisierung von verfahrenstechnischen Prozessen, Steuerung von Batch-Prozessen, Prozessleittechnik mit Mensch-Maschine-Kommunikation;
- **Produktionsautomatisierung:** Überwachung und Steuerung der Arbeitsabläufe von Werkzeugmaschinen und Robotern in Fertigungs- und Logistiksystemen;
- **Fahrzeugautomatisierung:** Überwachung und Steuerung von Motoren, Fahrzeugkomponenten oder Fahrerassistenzsystemen;
- **Gebäudeautomatisierung:** Steuerung von Fahrstühlen, Regulierung der Raumtemperatur, Überwachung der Gebäudesicherheit;
- **Informationstechnik:** Informations- und Datenverarbeitung, Übertragung von Daten in Netzwerken.

Allgemeine Prinzipien der Automatisierung Ein automatisiertes System besteht im Allgemeinen aus den folgenden zwei Komponenten:

1. **automatisierender Prozess:** Umwandlung von Energie oder Materie und Übertragung von Prozessdaten an die Automatisierungseinrichtung;
2. **Automatisierungseinrichtung:** Einwirkung auf den Prozess durch Stellgrößen im Regelkreislauf.

Das zentralste Prinzip der Automatisierungstechnik ist die Schaffung von Rückkopplung, bei dem sich das gesteuerte System und das steuernde System gegenseitig beeinflussen.

Die Bearbeitung einer Automatisierungsaufgabe erfolgt dabei durch die folgenden grundlegenden Schritte:

1. **Aufstellung des Modells:** Das Verhalten des zu automatisierenden Systems wird durch ein mathematisches Modell beschrieben.
2. **Vorhersage des Systemverhaltens:** Die wichtigen Eigenschaften des zu automatisierenden Systems werden mithilfe des Modells ermittelt.
3. **Entwurf von Algorithmen:** Für die zu lösende Automatisierungsaufgabe werden passende algorithmische Verfahren in Form von Reglern oder Diagnosetools entwickelt.
4. **Verifizierung des Modells:** Im Rahmen von Simulationen wird die Automatisierungseinrichtung am Modell getestet und überprüft, ob sich das gesamte System so verhält wie vorgeschrieben.
5. **Implementierung:** Die Automatisierungseinrichtung wird in dem zugehörigen technischen Gerät umgesetzt.

Methoden der Automatisierungstechnik Im Bereich der Automatisierungstechnik verwendet man je nach Anwendung sowohl kontinuierliche als auch ereignisdiskrete Modelle. Im ersten Fall spricht man dabei von einer kontinuierlichen und im zweiten Fall von einer diskreten Steuerung. Die Methodiken zur Automatisierung von technischen Systemen hängen grundlegend von deren abstrakten Modellen ab. Zentrale Modellformen oder Lösungsmethodiken sind dabei die folgenden:

- **Systemidentifikation:** Modellbildung auf der Grundlage von Bilanzgleichungen und physikalischen Erhaltungssätzen, in Form einer Beziehung zwischen Ein- und Ausgangsgrößen durch ein beschreibendes Differenzialgleichungssystem. Für lineare, zeitinvariante Systeme erfolgt die Überführung mithilfe der Laplace-Transformation und die Ermittlung der Übertragungsfunktion. Anschließend wird das System mit definierten Testsignalen angeregt, um so die Abhängigkeit der Ausgangsgröße von den Eingangsgrößen des Systems zu bestimmen und die Parameter des Modells anzupassen.
- **Endliche Automaten:** Modell eines digitalen zeitdiskreten Systems in Form eines gerichteten Graphen, dessen Knoten die Zustände und dessen gerichtete Kanten die möglichen Übergänge darstellen. Mit Automaten kann beispielsweise das Verhalten von gesteuerten diskreten Systemen beschrieben werden oder sie dienen zur Definition formaler Sprachen (z. B. Programmiersprache, natürliche Sprache).
- **Markov-Modelle:** stochastische Form eines Automaten, bei dem Zustandsübergänge mit Wahrscheinlichkeiten modelliert werden. Mit Markov-Modellen lassen sich Prognosen über die zukünftige Entwicklung eines Systems oder Prozesses erstellen, ohne dabei dessen gesamte Vorgeschichte zu kennen.
- **Neuronale Netze:** informationsverarbeitende Systeme, die aus einer großen Anzahl von Einheiten (Neuronen) bestehen, die Informationen durch Aktivierung der einzelnen Neuronen über gewichtete Verbindungen verschicken. Neuronale Netze haben heu-

te einen sehr breiten Anwendungsbereich bei der Mustererkennung, zur Steuerung von Robotern oder bei der Entwicklung von autonomen Systemen.

- **Fuzzylogik:** Verallgemeinerung der zweiwertigen booleschen Logik zur Modellierung von Unsicherheit durch umgangssprachliche Beschreibungen. Die sogenannten Fuzzyregler werden in der Automatisierungstechnik und in anderen Bereichen der Regelungstechnik als einfache Alternative für konventionelle Regler eingesetzt.
- **Petri-Netze:** Modellform für verteilte diskrete Systeme auf Basis endlicher Automaten zur vereinfachten Modellierung von unabhängigen Zustandsübergängen in parallelen Prozessen. Die Petri-Netze finden neben der Informatik und Automatisierungstechnik auch in der Logistik und in Bereichen des Operations Research eine Anwendung.
- **Evolutionäre Algorithmen:** Methodik, die auf den Anpassungsstrategien der Natur basiert, bei denen das aus der Evolutionslehre basierende Gesetz des „survival of the fittest“ in ein algorithmisches Verfahren übersetzt wird. Diese Art von Algorithmen besitzt bei einer Vielzahl von Optimierungsproblemen wie der Optimierung von Baugruppen oder im Bereich der Versuchsplanung umfassende Einsatzmöglichkeiten.
- **Bildverarbeitung:** Disziplin zur Aufbereitung, Analyse, Klassifizierung und Interpretation von visuellen Informationen in Form von Bilddaten. Die Bildverarbeitung ist eine Schlüsseltechnologie in der Automatisierungstechnik zur Klassifizierung von Objekten im Bereich der Qualitätsprüfung (z. B. Oberflächenkontrolle, Vollständigkeitskontrolle) oder zur Lagebestimmung und Vermessung von Objekten.

Graphen, Automaten, evolutionäre Algorithmen und die Methodiken der Bildverarbeitung stellen wir in diesem Buch mit ihren zugehörigen Anwendungsbereichen detailliert vor. Neuronale Netze und Markov-Modelle sind Gegenstand des 3. Bandes dieser Buchreihe.

Literaturhinweise

1. Domschke, W., Drexl, A. (2011). *Operations Research*. Springer.
2. Neumann, K., Morlock, M. (2004). *Operations Research*. Hanser.
3. Werners, B. (2008). *Grundlagen des Operations Research*. Springer.
4. Thonemann, U. (2010). *Operations Management*. Pearson.
5. Lehmann, T.M. (2005). *Handbuch der Medizinischen Informatik*. Hanser.
6. Lipinski, H.G. (1999). *Einführung in die medizintechnische Informatik*. Oldenbourg.
7. Dugas, M. (2003). *Medizinische Informatik und Bioinformatik*. Springer.
8. Zauner, M., Schrempf, A. (2009). *Informatik in der Medizintechnik*. Springer.
9. Handels, H. (2009). *Medizinische Bildverarbeitung*. Vieweg.
10. Husar, P. (2010). *Biosignalverarbeitung*. Springer.
11. Paetz, J. (2005). *Soft Computing in der Bioinformatik*. Springer.
12. Merkl, R., Waack, S. (2009). *Bioinformatik Interaktiv*. Wiley-Blackwell.
13. Schuster, R. (2009). *Biomathematik*. Vieweg&Teubner.
14. Wintermantel, E. (2010). *Medizintechnik*. Springer.
15. Lunze, J. (2012). *Automatisierungstechnik*. Oldenbourg.

16. Langmann, R. (2010). *Taschenbuch der Automatisierung*. Hanser.
17. Heinrich, B., Linke, P., Glöckler, M. (2015). *Grundlagen Automatisierung – Sensorik, Regelung, Steuerung*. Springer.
18. Gevatter, H-J., Grünhaupt, U. (2006). *Handbuch der Mess- und Automatisierungstechnik in der Produktion*. Springer.
19. Jakoby, W. (2013). *Automatisierungstechnik. Algorithmen und Programme*. Springer.

Im 1. Band *Grundlagen* haben wir bereits Entwurfsmuster für die Entwicklung von Algorithmen kennengelernt. Das Ziel bestand darin, für bestimmte Problemklassen allgemeine Muster zu konstruieren, um diese geeignet für ein konkretes Problem anzuwenden. Diese Muster sind ein abstrakter Programmrahmen, der dann für das jeweilige Problem nur noch ausgefüllt werden muss. Wir haben diese allgemeinen Lösungsmethoden zunächst vorgestellt und sie anschließend an einem einfachen Beispiel gezeigt.

Im Bereich der objektorientierten Programmierung gehen wir nun auf ähnliche Weise vor. In der Softwareentwicklung gibt es für viele Entwicklungsschritte spezielle Muster zum Entwurf und der Implementierung von Programmen. Diese Entwurfsmuster sind bewährte Lösungsvorschläge für bestimmte Problemstellungen, die bereits in vielen Anwendungen erfolgreich verwendet wurden.

Ähnlich wie algorithmische Muster sind auch die objektorientierten Entwurfsmuster nicht auf eine bestimmte Programmiersprache beschränkt. Beim Erlernen der allgemeinen Prinzipien ist es jedoch von großem Vorteil, diese Muster an einem konkreten Beispiel zu implementieren. Genau diesen Weg werden wir in diesem Kapitel mithilfe der Programmiersprache Java verfolgen.

Das Ziel von objektorientierten Entwurfsmustern für den Softwareentwurf ist, bereits gewonnene Erkenntnisse wiederverwendbar zu machen, die Flexibilität der Softwarearchitektur zu erhöhen sowie einen verständlichen, erweiterbaren und wartungsfreundlichen Code zu schreiben. Durch das hohe Abstraktionsniveau dieser Entwurfsmuster lässt sich damit auch die Entwicklung großer Softwarepakete zielsicher und effizient planen.

In diesem Kapitel werden wir zunächst in einem einführenden Beispiel die Grundprinzipien der objektorientierten Programmierung wiederholen. Im Anschluss daran stellen wir zentrale und weiterführende Konzepte der objektorientierten Programmierung in Java vor. Die objektorientierten Entwurfsmuster beruhen vor allem auf den zwei Konzepten der Vererbung und der Polymorphie. Mit diesen Konzepten stellen wir die zentralen Entwurfsmuster objektorientierter Programmierung mithilfe von Beispielen in Java vor.

2.1 Einführendes Beispiel

Als einführendes Beispiel betrachten wir eine Aufgabenstellung zur Klassifizierung von Produkten in Form von Schrauben.

Ein Hersteller von Schrauben will seine Produkte nach dem folgenden Schema einordnen:

- Schrauben mit einem Durchmesser bis zu 3 mm und einer Länge bis zu 20 mm haben den Preis EUR 0,30.
- Schrauben mit einem Durchmesser zwischen 3 und 5 mm und einer Länge zwischen 20 und 30 mm haben den Preis EUR 0,40.
- Schrauben mit einem Durchmesser zwischen 5 und 6 mm und einer Länge zwischen 20 und 30 mm haben den Preis EUR 0,60.
- Schrauben mit einem Durchmesser zwischen 6 und 15 mm und einer Länge zwischen 30 und 50 mm haben den Preis EUR 0,80.
- Schrauben mit einem Durchmesser zwischen 15 und 20 mm und einer Länge zwischen 30 und 50 mm haben den Preis EUR 0,90.

Die Aufgabe besteht darin ein Programm zu erstellen, das den richtigen Preis einer Schraube ermittelt, wenn Durchmesser und Länge eingegeben werden. Sollte eine Schraube keiner der oben beschriebenen Kategorien angehören, soll die Meldung „Unbekannter Schraubentyp“ ausgegeben werden.

Wir stellen im Folgenden zwei Varianten zur Lösung dieser Problemstellung vor. Der erste Fall ist die herkömmliche strukturierte Programmierung, die wir bereits in vielen Fällen erfolgreich angewandt haben. Mit diesem Programmierparadigma können wir schnell eine Lösung für eine gegebene Problemstellung finden. Wesentlicher Nachteil dieses Ansatzes ist es, dass die Struktur der Daten bekannt sein muss und kein logischer Zusammenhang zwischen Daten und den darauf anwendbaren Operationen besteht. Das Ergebnis ist in vielen Fällen ein starrer, unflexibler und schwer erweiterbarer Code.

Im zweiten Fall verwenden wir einen objektorientierten Ansatz, bei dem zusammengehörige Daten und die darauf arbeitende Programmlogik zu einer Einheit zusammengefasst werden. Dieser Programmcode verlangt auf den ersten Blick vom Programmierer etwas mehr Arbeit ab, vor allem da ein Konzept von mehreren Klassen erstellt werden muss. Der große Vorteil davon ist, dass wir einen gut erweiterbaren, modularen und für andere ähnliche Problemstellungen wiederverwendbaren Programmcode erhalten.

Strukturierte Programmierung

Die einfachste und kürzeste Implementierung erfolgt über eine Methode `bestimmePreis` mithilfe von `if-else`-Konstrukten:

```

public class Schrauben_Strukturiert
{
    // Bestimmung des Preises der Schrauben
    public static void bestimmePreis(double d, double l)
    {
        if (d<=3 && l<=20)
            System.out.printf("Der Preis beträgt 0.3 Euro.");
        else if (d>3 && d<=5 && l> 20 && l<=30)
            System.out.printf("Der Preis beträgt 0.4 Euro.");
        else if (d>5 && d<=6 && l>20 && l<=30)
            System.out.printf("Der Preis beträgt 0.6 Euro.");
        else if (d>6 && d<=15 && l>30 && l<=50)
            System.out.printf("Der Preis beträgt 0.8 Euro.");
        else if (d>15 && d<=20 && l>30 && l<=50)
            System.out.printf("Der Preis beträgt 0.9 Euro.");
        else
            System.out.println("Unbekannter Typ");
    }

    public static void main(String[] args)
    {
        // -----
        // --- 1. Durchmesser
        double durchmesser = 5.5;

        // --- 2. Laenge
        double laenge = 23;

        // -----
        bestimmePreis(durchmesser, laenge);
    }
}

```

Dieser Programmtext ist zwar schnell geschrieben, hat jedoch, wie wir gleich sehen werden, zahlreiche Nachteile.

Objektorientierte Programmierung

In der realen Welt gibt es eine Preisgruppe für die Schraubentypen. Diese Preisgruppe bilden wir nun mit einer Klasse `Preisgruppe` nach. Die Attribute sind neben dem Preis die Unter- und Obergrenzen für den Durchmesser und die Länge.

```

public class Preisgruppe
{
    double uD, oD; // unterer und oberer Durchmesser
    double uL, oL; // untere und obere Länge
    double preis;  // Preis
}

```

```

// Konstruktor zur Initialisierung einer Preisgruppe
public Preisgruppe(double uD, double oD, double ul, double ol, double preis)
{
    this.uD = uD;
    this.oD = oD;
    this.ul = ul;
    this.ol = ol;
    this.preis = preis;
}

// Prüfung des Preises des Produktes mit Durchmesser d und Laenge l
public boolean pruefePreis(double d, double l)

// Rueckgabe des Preises
public double getPreis()
}

```

Die Prüfung, ob ein gegebener Durchmesser und eine gegebene Länge zu einer Preisgruppe gehören, erfolgt mit der Methode `pruefePreis`. Diese bekommt die zwei Parameter des zu testenden Durchmessers und der zu testenden Länge übergeben. Anschließend vergleicht die Methode diese beiden Parameter mit ihren eigenen Attributen.

```

// Prüfung des Preises des Produktes mit Durchmesser d und Laenge l
public boolean pruefePreis(double d, double l)
{
    if (d > uD && d <= oD && l > ul && l <= ol)
        return true;
    else
        return false;
}

```

Diese Methode muss nur ein einziges Mal geschrieben werden, da einzelne Preisgruppenobjekte mit unterschiedlichen Parametern erzeugt worden sind. Auf die verschachtelten `if-else`-Schleifen können wir damit verzichten. Liegt die übergebene Größe im Bereich der Preisgruppe, dann gibt die Methode `true` zurück, andernfalls `false`.

Die Methode `getPreis` gibt den zugehörigen Preis als `double`-Wert zurück. Änderungen wie ein Steueraufschlag können leicht in dieser Methode noch ergänzt werden.

```

public double getPreis()
{
    return preis;
}

```

In der Klasse `TestSchrauben` definieren wir nun die 5 Preisgruppenobjekte in Form eines Feldes vom Typ `Preisgruppe`. Anschließend gehen wir der Reihe nach durch alle Preisgruppen durch und vergleichen, ob der zu testende Durchmesser und die zu testende Länge vorhanden sind. Am Ende wird das Ergebnis ausgegeben.

```

public class Schrauben
{
    public static void main(String[] args)
    {
        // -----
        // --- 1. Durchmesser
        double durchmesser = 5.5;

        // --- 2. Laenge
        double laenge = 23;

        // -----
        // --- 1. Definition der Preisgruppen
        Preisgruppe pg[] = new Preisgruppe[5];
        pg[0] = new Preisgruppe(0, 3, 0, 20, 0.30);
        pg[1] = new Preisgruppe(3, 5, 20, 30, 0.40);
        pg[2] = new Preisgruppe(5, 6, 20, 30, 0.60);
        pg[3] = new Preisgruppe(6, 15, 30, 50, 0.80);
        pg[4] = new Preisgruppe(15, 20, 30, 50, 0.90);

        // --- 2. Bestimmung des aktuellen Preises
        double preis = -1;
        for (int i = 0; i < pg.length; i++)
            if (pg[i].pruefePreis(durchmesser, laenge))
            {
                preis = pg[i].getPreis();
                break;
            }

        // --- 3. Ausgabe
        if (preis >= 0)
            System.out.printf("Der Preis beträgt %1.2f Euro.", preis);
        else
            System.out.printf("Der Preis ist unbekannt.");
    }
}

```

Ausgabe

Der Preis beträgt 0,60 Euro.

Die objektorientierte Implementierung ist auf den ersten Blick etwas umfangreicher, dafür besitzt sie zahlreiche Vorteile. Die beiden Klassen sind übersichtlicher und gut nachvollziehbar. Änderungen wie beispielsweise durch Hinzufügen oder Weglassen von Gleichheitszeichen in der `if`-Abfrage benötigen nur 1 Änderung, statt 5 wie in der obigen Implementierung. Damit können problemlos neue Preisgruppen ergänzt werden, ohne Programmcode gegebenenfalls wieder fehlerhaft zu machen. Aufgrund der Tatsache, dass das Programm in zwei Teilklassen zerlegt ist, kann die Klasse `Preisgruppe` für andere Produkte leicht wiederverwendet werden.

2.2 Objektorientierte Analyse

Die Grundlage der objektorientierten Programmierung eines Softwaresystems ist die Modellierung der Aufgabenstellung durch kooperierende Objekte. Objekte sind besondere Daten- und Programmstrukturen, die Eigenschaften (Attribute) und Verhaltensweisen (Methoden) besitzen. Bei der objektorientierten Analyse sind die zu modellierenden Objekte zu finden, zu organisieren und zu beschreiben. Die einzelnen Schritte lassen sich in der folgenden Reihenfolge darstellen:

1. Finden der Objekte

In der gegebenen Problemstellung des zu modellierenden Softwaresystems sind die darin enthaltenden Objekte zu finden. Diese Objekte beschreiben eine Gruppe von interagierenden Elementen, um damit reale Objekte wie Autos, Kunden, Aufträge oder Artikel direkt in Software zu modellieren.

2. Organisation der Objekte

Bei einer großen Anzahl von beteiligten Objekten werden die zusammengehörigen Objekte in Gruppen zusammengesetzt. Diese Zusammenstellung ergibt sich aus den Beziehungen der einzelnen Objekte zueinander. Beispielsweise können Objekte andere Objekte enthalten (z. B. Bestellsystem enthält Artikel, Maschine enthält Komponenten). Die Gruppenzusammengehörigkeit ist umso größer, je mehr Beziehungen zwischen einzelnen Objekten bestehen.

3. Interaktion der Objekte

Die Interaktion bzw. Assoziation zweier Objekte beschreibt die Beziehung zwischen zwei Objekten. Die Aggregation beschreibt die Zusammensetzung eines Objekts aus anderen Objekten. Die Komposition ist ein Spezialfall einer Aggregation, bei der Abhängigkeiten zwischen den Objekten in der Form bestehen, dass ein beschriebenes Objekt nur durch gewisse Teilobjekte existiert (z. B. Maschine besteht aus Teilen, Container besteht aus Behältern, Behälter besteht aus Gegenständen).

4. Beschreibung der Attribute der Objekte

Die Attribute sind die individuellen Eigenschaften zur Definition des aktuellen Zustands des Objekts. Das Attribut ist ein Datenelement einer Klasse, das in allen Objekten vorhanden ist (z. B. Farbe eines Autos, Name eines Artikels, Länge eines Behälters).

5. Beschreibung des Verhaltens der Objekte

Das Verhalten eines Objekts wird durch Methoden innerhalb einer Klasse definiert, die jeweils auf einem Objekt dieser Klasse operieren. Die Methoden definieren dabei eine Aufgabe in Form von Operationen auf der Instanz einer Klasse (z. B. Algorithmus steuert Maschine, Kalkulation berechnet Preis eines Artikels).

Das Ergebnis dieser Analyse ist die obige Beschreibung sowie die Erstellung von Abbildungen mit den einzelnen Klassen (Kästchen) und deren Beziehung untereinander (Linien und Text).

Beispiel 2.1 Wir modellieren eine Software für die Produktionsplanung einer Firma. Das Softwaresystem besteht aus einer Eingabemaske, bei dem der Kunde die Auftragsdaten und die Produktionsplanungsdaten eingeben muss. Die Auftragsdaten bestehen aus den einzelnen Artikeln und die Produktionsplanungsdaten aus der Produktionsmaschine und deren Konfiguration.

1. **Finden der Objekte:** Eingabemaske, Kunde, Auftrag, Artikel, Produktionsplanung, Produktionsmaschine, Konfiguration
2. **Organisation der Objekte:** Die Objekte werden in 4 Gruppen aufgeteilt:
 - (a) Eingabemaske
 - (b) Kunde
 - (c) Auftrag, Artikel
 - (d) Produktionsplanung, Produktionsmaschine, Konfiguration
3. **Interaktion der Objekte:**
 - Kunde, Auftrag und Produktionsplanung werden über die Eingabemaske eingegeben.
 - Artikel gehört zu dem Auftrag
 - Produktionsmaschine ist Produktionsplanung zugeordnet
 - Konfiguration wird von Produktionsplanung ausgewählt
4. **Beschreibung der Attribute der Objekte:**
 - Kunde: Name, Adresse, Kundennummer
 - Artikel: Nummer, Name, Preis, Anzahl
 - Auftrag: Array von Artikeln
 - Konfiguration: Maschinenparameter
 - Produktionsmaschine: Maschinenbezeichner, Laufzeit
 - Produktionsplanung: Produktionsmaschine, Konfiguration
5. **Beschreibung des Verhaltens der Objekte:**
 - Kunde: getKundennummer, getKunde, ...
 - Auftrag: getAuftragswert, getAnzahlauftraege, ...
 - Artikel: getArtikelnummer, getArtikelanzahl, ...
 - Produktionsplanung: getKonfiguration, ...
 - Konfiguration: getParameter, setParameter, ...
 - Produktionsmaschine: getMaschinenbezeichner, ...

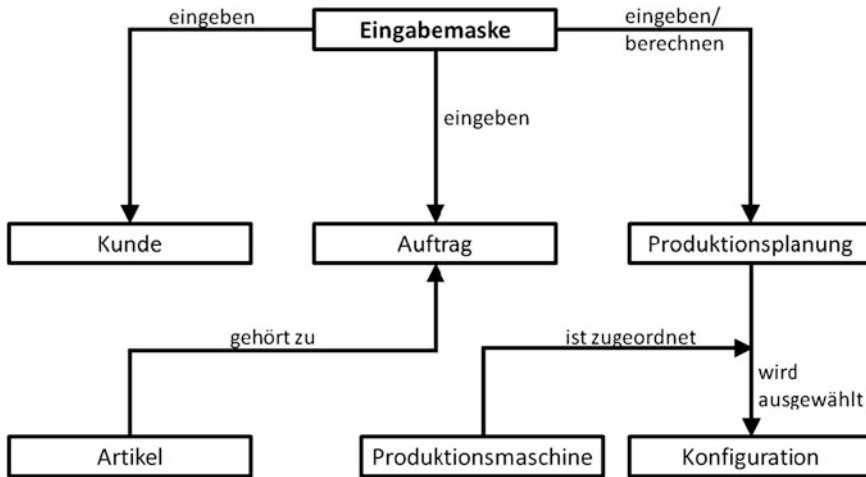


Abb. 2.1 Objektorientierte Analyse einer Planungssoftware

Die Abb. 2.1 zeigt ein Übersichtsdiagramm der objektorientierten Analyse dieser Planungssoftware.

2.3 Objektorientierte Konzepte

Die objektorientierte Programmierung ist sehr gut geeignet, umfangreiche Programme in eine Gruppe von interagierenden Objekten aufzuteilen, um damit eine ganze Reihe von wichtigen Prinzipien der Programmierung zu erfüllen:

- **Flexibilität:** Klassen sind nicht starr auf ein vorliegendes Problem angepasst, sodass Änderungen problemlos möglich sind.
- **Wiederverwendbarkeit:** Einzelne Module können in neuen Anwendungen weiterverwendet werden, sodass die Kosten für die Neuentwicklung neuer Systeme sinken.
- **Erweiterbarkeit:** System kann gut um neue Funktionalität erweitert werden, um beispielsweise neue Bibliotheksklassen optimal hinzuzufügen.
- **Veränderbarkeit:** Veränderungen an einer bestimmten Stelle wirken sich nicht auf andere Teile des Systems aus.
- **Wartbarkeit:** System besteht nur aus wenigen Abhängigkeiten, sodass es über die Zeit kostengünstig weiterentwickelbar ist.
- **Redundanzfreiheit:** System besitzt keine mehrfache Verwendung von identischem oder ähnlichem Code.
- **Verständlichkeit:** System ist logisch durch einen gut lesbaren Code aufgebaut, damit Änderungen schnell und sicher umgesetzt werden können.

Das Ziel der objektorientierten Programmierung ist, die Abhängigkeiten der verschiedenen Module zu minimieren, die Korrektheit sicherzustellen sowie ein verständliches und gut dokumentiertes Programmpaket zu schaffen. Alle in diesem Kapitel vorgestellten Konzepte sind dazu da, diese Prinzipien in die Praxis umzusetzen.

Im Rahmen der objektorientierten Programmierung bezeichnet der Begriff des Refactoring die Veränderung des Programmcodes, ohne die Funktionalität zu erweitern. Das Refactoring eines Codes wird durchgeführt, um die Erweiterbarkeit und die Wartbarkeit sicherzustellen. Bei größeren Programmpaketen ist es oftmals sehr sinnvoll, in gewissen Entwicklungsschritten suboptimale Codeteile zu identifizieren, aufzuräumen und für eventuell weitere Erweiterungen vorzubereiten.

2.3.1 Innere Klassen

In Java ist es möglich eine Klasse in eine andere Klasse mit hineinzunehmen, um beispielsweise nur lokale Typdeklarationen zu definieren, die keine weitere Sichtbarkeit benötigen. Wir stellen nun einige verschiedene Varianten zur Definition von inneren Klassen vor. Innere Klassen sind oft sehr nützlich für den Entwurf von Hilfsdatenstrukturen, die nur in speziellen Klassen benötigt werden. Ebenso finden Sie in einigen Entwurfsmustern der Softwaretechnik Anwendung.

Mitgliedsklasse Eine Mitgliedsklasse einer äußeren Klasse ist ähnlich einem Attribut und wird wie folgt definiert:

```
class AeussereKlasse
{
    class InnereKlasse
    {
        ...
    }
}
```

Um ein Objekt der inneren Klasse zu erzeugen, muss ein Objekt der äußeren Klasse existieren:

```
AeussereKlasse out = new AeussereKlasse();
InnereKlasse in = out.new InnereKlasse();
```

Zum Zugriff von der inneren Klasse auf eine Variable `var` der äußeren Klasse, die von inneren Klassen überdeckt wird, verwenden wir den Befehl `AeussereKlasse.this.var`.

Beispiel 2.2 Wir definieren eine äußere und eine innere Klasse mit einer Methode `ausgabe`:

```
public class AeussereKlasse
{
    String name = "Aussen";

    class InnereKlasse
    {
        String name = "Innen";
        public void ausgabe()
        {
            System.out.println(name);
            System.out.println(AeussereKlasse.this.name);
        }
    }

    public static void main( String[] args )
    {
        AeussereKlasse out = new AeussereKlasse();
        InnereKlasse in = out.new InnereKlasse();
        in.ausgabe();
    }
}
```

Ausgabe

```
Innen
Aussen
```

Allgemeine Erklärung

Zuerst wird ein Objekt `out` der äußeren Klasse `AeussereKlasse` angelegt. Anschließend kann ein Objekt `in` der inneren Klasse `InnereKlasse` mithilfe der Variable `out` definiert werden. Mithilfe der Objektvariable `in` kann dann auf die Methode `ausgabe` der inneren Klasse zugegriffen werden.

Statische innere Klasse Bei einer statischen inneren Klasse wird eine innere Klasse als eine Art statische Eigenschaft in der äußeren Klasse definiert. Der Unterschied zur Mitgliedsklasse ist das Schlüsselwort `static`:

```
class AeussereKlasse
{
    static class InnereKlasse
    {
        ...
    }
}
```

Der Zugriff von außen auf eine innere Klasse erfolgt durch den Aufruf:

```
InnereKlasse in = new AeussereKlasse.InnereKlasse();
```

Im Unterschied zu Mitgliedsklassen existieren statische innere Klassen auch ohne Objekt der äußeren Klasse.

Beispiel 2.3 Wir definieren eine äußere und eine innere Klasse um den Zugriff auf Klassenvariablen zu erläutern:

```
public class AeussereKlasse
{
    static int zahl = 10;
    static String name = "Anton";
    double var = 1.23;

    static class InnereKlasse
    {
        public void ausgabe()
        {
            System.out.println(zahl);
            System.out.println(name);
        }
    }

    public static void main( String[] args )
    {
        InnereKlasse in = new AeussereKlasse.InnereKlasse();
        in.ausgabe();
    }
}
```

Ausgabe

```
10
Anton
```

Allgemeine Erklärung

Die statische innere Klasse `InnereKlasse` kann nur auf statische Objekte der äußeren Klasse `AeussereKlasse` zugreifen, also auf die Variablen `zahl` und `name`, nicht jedoch auf die Variable `var`.

Lokale Klassen Lokale Klassen sind spezielle innere Klassen, die in den Anweisungsblöcken von Konstruktoren oder Methoden gesetzt werden.

```
class AeussereKlasse
{
    AeussereKlasse()
    {
        class InnereKlasse
        {
            ...
        }
    }
}
```

Beispiel 2.4 Wir definieren eine äußere und eine innere Klasse im Konstruktor der äußeren Klasse:

```
public class AeussereKlasse
{
    String name = "Aussen";

    public AeussereKlasse()
    {
        class InnereKlasse
        {
            String name = "Innen";
            InnereKlasse()
            {
                System.out.println(name);
            }
        }
        new InnereKlasse();
    }

    public static void main( String[] args )
    {
        AeussereKlasse out = new AeussereKlasse();
        System.out.println(out.name);
    }
}
```

Ausgabe

```
Innen
Aussen
```

Allgemeine Erklärung

Durch das Anlegen des Objekts `out` wird der Konstruktor `AeussereKlasse` aufgerufen. In dieser Klasse ist die innere Klasse `InnereKlasse` definiert. Ein Objekt der inneren Klasse wird durch den Befehl `new InnereKlasse()` erzeugt. Damit wird der Konstruktor `InnereKlasse` aufgerufen und der Wert `Innen` der Variable `name` gegeben. Im Anschluss daran wird der Ausgabebefehl in der `main` aufgerufen und der Wert `Aussen` der Variable `name` ausgegeben.

2.3.2 Polymorphie

Der Begriff *Polymorphie* beschreibt die Vielgestaltigkeit, d. h., dass verschiedene Objekte auf die gleiche Anweisung (z. B. Aufruf einer Methode) unterschiedlich reagieren können. Ein Beispiel ist der „+“-Operator in Java, der für die beiden Objekte Zeichenketten (Verkettung zweier Zeichen) und Zahlen (Addition zweier Zahlen) unterschiedliche Bedeutung hat. Mittels Polymorphie kann ebenso eine Methode `zeichnen` für unterschiedliche geometrische Objekte wie beispielsweise ein Kreis oder ein Rechteck in verschiedener Art und Weise implementiert werden.

In Java besteht die Polymorphie auf den folgenden Prinzipien:

- **Überladen:** Mehrfache Verwendung eines Methodennamens innerhalb einer Klassendefinition mit unterschiedlichen Anzahlen und Typen von Parametern, aber mit gleichem Ergebnistyp.
- **Überschreibung:** Methode der Unterklasse ist in Bezug auf den Namen, den Ereignistyp und die Anzahl der Parameter identisch zu einer Methode der Oberklasse. Damit wird die vererbte Methode durch eine neue Implementierung überschrieben.

Mit der Verwendung von Polymorphie können sich damit hinter einem Namen verschiedene Umsetzungskonzepte verbergen. Die Umsetzung Polymorphie wird vor allem dadurch erreicht, dass Methoden innerhalb einer Klassenhierarchie überschrieben werden.

Die Eigenschaft der Polymorphie im Bereich der objektorientierten Programmierung tritt immer im Zusammenhang mit Vererbung und Interfaces auf. Im Folgenden werden wir dieses Konzept in abstrakte Klassen und Interfaces in Java umsetzen.

2.3.3 Abstrakte Klassen

In vielen Fällen soll eine Klasse nicht sofort programmiert werden, beispielsweise dann, wenn die Oberklasse nur Methoden für die Unterklasse definieren soll, von denen aber noch nicht bekannt ist, wie diese genau implementiert werden sollen. Abstrakte Klassen und Methoden werden in diesem Fall verwendet um unterschiedliche Implementierungen für verschiedene Objekte zu definieren. Eine *abstrakte Methode* definiert die Signatur (Schnittstelle) und enthält nur die Deklaration der Methode, nicht aber ihre konkrete Implementierung. Abstrakte Methoden werden in Java mit dem Attribut `abstract` gekennzeichnet. Anstelle der geschweiften Klammern mit den auszuführenden Anweisungen steht nur ein Semikolon:

```
public abstract rueckgabetyyp methodenname();
```

Abstrakte Methoden zeigen, dass die Oberklasse keine Information über die Implementierung besitzt und dass die Unterklasse die Aufgabe der konkreten Implementierung übernehmen muss.

Eine *abstrakte Klasse* ist eine Klasse, die mindestens eine abstrakte Methode enthält. In diesem Fall muss diese Klasse ebenfalls mit dem Schlüsselwort `abstract` bezeichnet werden, da ansonsten ein Objekt dieser Klasse angelegt werden kann und die abstrakte Methode aufrufbar ist, ohne jedoch definierte Inhalte zu besitzen.

```
abstract class AbstrakteKlasse
{
    ...
    public abstract rueckgabetyyp methodenname();
}
```

Abstrakte Klassen können nur mit dem Befehl `extends` abgeleitet werden, d. h., das Erzeugen von Objekten ist nicht erlaubt. In vielen Anwendungen ist das sehr sinnvoll, insbesondere dann, wenn eine Klasse nur als Oberklasse in einer Vererbungshierarchie existieren soll. Beispielsweise gibt es in der realen Welt keine allgemeinen unspezifizierten Objekte wie Gegenstände, sondern immer spezielle Unterarten wie Autos, Tische, Bücher oder anderes. Alle anderen Eigenschaften von normalen Klassen gelten auch für abstrakte Klassen.

Die abstrakten Klassen werden bei der Vererbung verwendet. Die sogenannte *konkrete Klasse* erhalten wir durch Ableitung der abstrakten Klasse mit dem Schlüsselwort `extends` und durch Implementierung des fehlenden Methodenrumpfes.

```
class KonkreteKlasse extends AbstrakteKlasse
{
    ..
    public rueckgabetyyp methodenname()
    {
        // Implementierung der Methode
    }
}
```

Durch Implementierung der abstrakten Methode wird die Klasse konkret und kann wie üblich aufgerufen werden. Es gilt damit auch die bekannte „Ist-eine-Art-von-Beziehung“ der Vererbung, sodass man Folgendes schreiben kann:

```
Abstrakte Klasse a = new KonkreteKlasse;
```

Da anstelle eines Objekts auch ein Objekt der Unterklasse auftauchen kann, spricht man von *Substitution* bzw. vom *Liskov'schem Substitutionsprinzip*. Eine abstrakte Klasse kann entweder ausschließlich abstrakte Methoden enthalten oder auch konkrete Implementierungen.

Für das Verständnis der Reihenfolge der Ausführungen der einzelnen Anweisungen betrachten wir folgendes kleines Demonstrationsbeispiel:

```
abstract class AbstrakteKlasse
{
    int a;
    public AbstrakteKlasse()
    {
        System.out.println("Aufruf des Konstruktors der abstrakten Klasse.");
        methode(a);
    }
    public abstract void methode(int a);
}

class KonkreteKlasse extends AbstrakteKlasse
{
    int a = 1;
    public KonkreteKlasse()
    {
        System.out.println("Aufruf des Konstruktors der konkreten Klasse.");
        methode(a);
    }
    public void methode(int a)
    {
        System.out.println("a = " + a);
    }
}

public class TestAbstrakteKlasse
{
    public static void main(String[] args)
    {
        AbstrakteKlasse aK = new KonkreteKlasse();
        aK.methode(2);
    }
}
```

Ausgabe

Aufruf des Konstruktors der abstrakten Klasse.

a = 0

Aufruf des Konstruktors der konkreten Klasse.

a = 1

a = 2

Allgemeine Erklärung

Die Reihenfolge der Initialisierung der Variablen ist die folgende:

1. Aufruf des Konstruktors `AbstrakteKlasse` der abstrakten Oberklassen,
2. Ausführung des Rumpfes des Konstruktors,
3. Initialisierung der konkreten Klasse,
4. Aufruf des Konstruktors `KonkreteKlasse` der konkreten Klasse,
5. Ausführung des Rumpfes des eigenen Konstruktors.

Beispiel 2.5 Wir betrachten eine Anwendung zur Berechnung des Preises unterschiedlicher Produkte einer Firma. Dazu definieren wir zunächst eine abstrakte Basisklasse `Produkt` mit den drei Instanzvariablen `prodnr` für die Produktnummer, `name` für den Produktnamen und `preis` für den Preis des Produktes sowie mit der abstrakten Methode `berechnePreis`:

```
abstract class Produkt
{
    int prodnr; // Produktnummer
    String name; // Produktname
    int preis; // Produktpreis

    public Produkt()
    {
    }

    public abstract double berechnePreis();
}
```

In dieser Klasse können noch weitere Methoden implementiert werden, um die Instanzvariablen zu manipulieren. Die Klasse `Produkt` verwenden wir nun als Basisklasse für spezialisierte Unterklassen, um gewisse Produkte, in diesem Fall `Produkt A` oder `Produkt B`, in der Preisgestaltung individuell zu gestalten. Dafür wird die abstrakte Methode `berechnePreis` in den abgeleiteten Klassen mit dem gewünschten Inhalt gefüllt.

```
class Produkt_A extends Produkt
{
    double preisMaterial;
    double preisStunde;
    double anzahlStunden;
    final double mwst = 1.19;

    public Produkt_A(double preisMaterial, double preisStunde, double anzahlStunden)
    {
        this.preisMaterial = preisMaterial;
        this.preisStunde = preisStunde;
        this.anzahlStunden = anzahlStunden;
    }

    public double berechnePreis()
    {
        return mwst * (preisMaterial + preisStunde*anzahlStunden);
    }
}

class Produkt_B extends Produkt
{
    double preisMaterial;
    double preisArbeit;
    final double mwst = 1.07;
```

```

public Produkt_B(double preisMaterial, double preisArbeit)
{
    this.preisMaterial = preisMaterial;
    this.preisArbeit = preisArbeit;
}

public double berechnePreis()
{
    return mwst * (preisMaterial + preisArbeit);
}
}

```

Der Aufruf von `berechnePreis` in der folgenden Klasse `Preisberechnung` führt zum jeweiligen konkreten Objekt des Arrayelements `prod` vom Typ `Produkt_A` oder `Produkt_B` die passende Berechnung aus.

```

public class Preisberechnung
{
    public static void main(String[] args)
    {
        Produkt prod[] = new Produkt[3];
        prod[0] = new Produkt_A(100.34, 39.00, 3);
        prod[1] = new Produkt_B(59.99, 50);
        prod[2] = new Produkt_A(59.99, 50.00, 4.6);

        double summe = 0.0;
        for(Produkt p : prod)
            summe = summe + p.berechnePreis();
        System.out.printf("Preis aller Produkte = %1.2f Euro", summe);
    }
}

```

Ausgabe

```
Preis aller Produkte = 721,41 Euro
```

Durch die abstrakte Klasse `Produkt` können beliebig viele verschiedene Produkte erzeugt werden. Alle diese Produkte können in einem gemeinsamen Feld vom Typ der abstrakten Klasse `Produkt` gespeichert werden. Erst durch das Anlegen konkreter Produkte wird der jeweilige Produkttyp festgelegt. In vielen praktischen Anwendungen hat dies enorme Vorteile, da man durch die abstrakte Klasse `Produkt` einen allgemeinen Datentyp besitzt, der beispielsweise an andere Konstruktoren oder Methoden übergeben werden kann.

2.3.4 Interface

Die Verwendung eines Interface (Schnittstelle) ist ein zentraler Bestandteil zahlreicher Entwurfsmuster, die wir im Anschluss vorstellen wollen. Interfaces werden immer dann

verwendet, wenn Eigenschaften einer Klasse beschrieben werden sollen, die nicht direkt in einer Vererbungshierarchie abbildbar sind.

Mithilfe des Interface ist es möglich, dass Klassen eine oder auch mehrere Schnittstellendefinitionen erben, um damit eine Art von Mehrfachvererbung von Klassen in Java darzustellen. Mit Interfaces werden funktionale Eigenschaften durch Klassen ergänzt, sodass die konkrete Implementierung einfach geändert werden kann.

Abstrakte Klassen haben im Gegensatz dazu den Nachteil, dass sie alle Erweiterungen an alle erbenenden Objekte übernehmen müssen. Schnittstellen und abstrakte Klassen trennen in der objektorientierten Programmierung das „Was“ vom „Wie“. Abstrakte Methoden und Schnittstellen liefern Informationen über das „Was“, aber erst die konkrete Implementierung liefert die Umsetzung, das „Wie“.

Definition eines Interface Ein Interface ist eine spezielle Form einer Klasse, die nur Methoden deklariert und Konstanten enthält, ohne die Beschreibung der genauen Funktionalität. Ein Interface wird mit dem Bezeichner `interface` deklariert. Alle Methoden eines Interface sind damit abstrakt, öffentlich und definieren die Schnittstelle für den Zugriff:

```
public interface InterfaceName
{
    public rueckgabetyyp methode1();
    public rueckgabetyyp methode2();
}
```

Die vom Interface definierten Methoden geben wie die abstrakten Methoden niemals eine Implementierung an. Ein Interface darf auch im Gegensatz zu einer abstrakten Klasse keinen Konstruktor besitzen. Der Name einer Schnittstelle endet in Java oft auf „-able“ wie beispielsweise `Runnable`.

Implementierung eines Interface Eine Klasse implementiert ein Interface in der Form, dass die `class`-Anweisung um eine `implements`-Klausel hinter dem Namen des zu implementierenden Interface angegeben wird:

```
public class Klassenname implements InterfaceName
{
    public rueckgabetyyp methode1();
    {
        // Implementierung der Methode
    }
    public rueckgabetyyp methode2();
    {
        // Implementierung der Methode
    }
}
```

Da in Schnittstellen deklarierte Methoden immer `public` sind, müssen auch die implementierten Methoden in den Klassen öffentlich sein.

An dieser Stelle ist die Ausdrucksweise wichtig: Klassen werden vererbt und Schnittstellen implementiert. Ein großer Fehler ist es Schnittstellen später zu ändern, wenn schon viele Klassen die Schnittstelle implementiert haben. Bereits wenn nur eine neue Operation hinzukommt oder wenn ein Typ einer Variable geändert wird, sind alle implementierten Klassen fehlerhaft. Wenn jedoch trotzdem Schnittstellen erweitert werden sollen, empfiehlt es sich eine neue Schnittstelle mit weiteren Operationen einzuführen, die die alte Schnittstelle erweitert. In Java wird dies beispielsweise durch Hinzufügen einer 2 an den alten Schnittstellennamen erledigt.

Ein Interface kommt immer dann zur Anwendung, wenn Eigenschaften einer Klasse beschrieben werden sollen, die nicht direkt in der normalen Vererbungshierarchie abbildbar sind. Auch beim Interface kommt wieder das Substitutionsprinzip zum Tragen, d. h., dass anstelle eines Objekts auch ein Objekt der Unterklasse verwendet werden kann:

```
Interfacename obj = new Klassenname();
```

Eng verbunden ist damit auch wieder das Prinzip der Polymorphie, also die Tatsache, dass verschiedene Arten von Objekten auf die gleichen Methoden unterschiedlich reagieren können. Im nachfolgenden Beispiel werden wir dieses Prinzip verdeutlichen.

Beispiel 2.6 Wir betrachten als Beispiel das folgende Interface `IFlaeche`:

```
public interface IFlaeche
{
    public double getLaenge();
    public double getBreite();
}
```

Wir wollen dieses Interface zur Berechnung der Größe unterschiedlicher Objekte wie Häuser, Tische usw. verwenden. Durch die Verwendung eines Interface verhindern wir eine sehr unnatürliche Ableitungshierarchie, da die Methoden unabhängig von der eigenen Vererbungslinie erzeugt werden:

```
class Haus implements IFlaeche
{
    private double laenge;
    private double breite;

    public Haus(double laenge, double breite)
    {
        this.laenge = laenge;
        this.breite = breite;
    }
}
```

```

    public double getLaenge()
    {
        return this.laenge;
    }

    public double getBreite()
    {
        return this.breite;
    }
}

class Tisch implements IFlaeche
{
    private double laenge;
    private double breite;

    public double getLaenge()
    {
        return 2.00;
    }

    public double getBreite()
    {
        return 1.50;
    }
}

```

In der main-Methode werden Objekte der beiden Klassen Haus und Tisch angelegt. Anschließend wird mit der statischen Methode `grundFlaeche`, mit dem beiderseitigen kompatiblen Übergabetyp `IFlaeche` die zugehörige Grundfläche berechnet:

```

public class Flaeche
{
    public static double grundFlaeche(IFlaeche fl)
    {
        return fl.getLaenge() + fl.getBreite();
    }

    public static void main(String[] args)
    {
        Haus haus = new Haus(15.04, 10.08);
        Tisch tisch = new Tisch();

        System.out.printf("Haus hat eine Fläche von %1.2f qm.\n", grundFlaeche(haus));
        System.out.printf("Tisch hat eine Fläche von %1.2f qm.\n", grundFlaeche(tisch));
    }
}

```

Ausgabe

Haus hat eine Fläche von 25,12 qm.
Tisch hat eine Fläche von 3,50 qm.

Der große Vorteil eines Interface ist, dass wir der Methode `grundFlaeche` der Klasse `Flaeche` nicht einen konkreten Datentyp wie `Haus` oder `Tisch` übergeben müssen, sondern nur das dazugehörige Interface `IFlaeche`. In Abhängigkeit des jeweiligen konkreten Typs, also `Haus` oder `Flaeche`, wird dann die zugehörige Grundfläche mithilfe der Methoden `laenge` und `breite` der jeweiligen konkreten Klasse berechnet.

Bemerkung 2.1

1. Interfaces sollten verwendet werden, wenn Eigenschaften einer Klasse beschrieben werden, die nicht direkt in der normalen Vererbungshierarchie stehen.
2. Eine Klasse kann auch mehrere Interfaces implementieren, wenn diese sämtliche definierten Methoden aller Interfaces implementieren. Die Namen der Interfaces werden durch Kommas getrennt hinter das Schlüsselwort `implements` geschrieben:

```
public class Unterklasse extends Oberklasse implements  
    Interface1, Interface2, ...
```
3. Mit jedem implementierten Interface wird eine Klasse zu dem dadurch definierten Datentyp kompatibel. Eine Klasse, die n Interfaces implementiert, ist dann zu $n + 1$ Datentypen kompatibel.
4. Ein Interface kann selbst auch abgeleitet werden. Die implementierende Klasse muss dann alle Methoden von allen übergeordneten Interfaces implementieren.
5. Die Gemeinsamkeit zwischen abstrakten Klassen und Interfaces ist, dass beide zur Definition von Variablen und Signaturen von Methoden verwendbar sind.

Interface Comparable Ein wichtiges Interface zum paarweisen Vergleich von Objekten ist das Interface `Comparable` des Pakets `java.lang`:

```
public interface Comparable  
{  
    public int compareTo(Object o);  
}
```

Für die Benutzung des Interface `Comparable` können beliebige Arten von Objekten (z. B. Strings) verglichen werden, solange alle das Interface `Comparable` implementieren. Die Methode `compareTo` liefert genau dann einen Wert kleiner 0, wenn das Objekt „kleiner“, größer 0, wenn es „größer“, und gleich 0, wenn es „gleich“ dem als Argument übergebenen Objekt ist. Mithilfe von `Comparable` kann die Reihenfolge der Objekte einer Klasse ermittelt werden.

Beispiel 2.7 Wir verwenden das Interface `Comparable` zur Bestimmung des kleinsten Elements in einer Menge von Zeichenketten. Bei Objekten vom Typ `String` ist die Methode `compareTo` durch die Reihenfolge der Anfangsbuchstaben der Zeichenkette im Alphabet implementiert.

```

public class TestComparable
{
    public static Object getKleinstes(Comparable obj[])
    {
        Object minimum = obj[0];
        for (Comparable o : obj)
        {
            if (o.compareTo(minimum) < 0)
                minimum = o;
        }
        return minimum;
    }

    public static void main(String[] args)
    {
        Comparable obj[] = {"Fritz", "Paul", "Maria", "Peter", "Ute"};
        System.out.println("Kleinstes Element: " + (String) getKleinstes(obj));
    }
}

```

Ausgabe

Kleinstes Element: Fritz

Durch die Implementierung des Interface `Comparable` in eigenen Klassen können beliebige Objekte verglichen werden, solange die konkrete Ausgestaltung der Methode `compareTo` implementiert ist.

Konstanten in Interfaces Interfaces können Konstanten definieren, die anschließend an alle implementierten Interfaces weitervererbt werden. Insbesondere wenn viele Programme sehr viele Konstanten benötigen, können diese bequem in ein Interface ausgelagert werden. Jede Klasse, die diese Konstanten benötigt, implementiert dann dieses Interface.

```

interface IKonstanten
{
    public static final konst1 = 1.2345;
    public static final konst2 = 3.4567;
    ...
}

```

Durch die Implementierung des Interface können alle Konstanten ohne vorangestellten Interfacenamen verwendet werden.

Statischer Import In Java ist es möglich, statische Teile aus Klassen und Interfaces durch den statischen Import mit dem Befehl `static import` zu verwenden. Beispielsweise kann die Klasse `java.lang.Math` durch den folgenden Befehl statisch importiert werden:

```
import static java.lang.Math.*;
```

Die enthaltenen Methoden können nach dem statischen Import der Klasse ohne Verwendung des Klassennamens `Math` verwendet werden:

```
System.out.println(Math.sqrt(5));  
System.out.println(Math.cos(1.98)); }
```

Nachbildung von Funktionszeigern Eine sehr nützliche Anwendung von Interfaces ist das Prinzip, dass eine Funktion als Argument an andere Funktionen übergeben werden kann. Die Schritte sehen hierzu wie folgt aus:

1. Definition eines Interface mit der Deklaration einer Methode `methode` des gewünschten Typs,
2. Implementierung des Interface von unterschiedlichen Klassen, welche die konkrete Implementierung der Methode `methode` definieren,
3. Anwendung der Klassen durch Instanzieren eines Objekts dieser Klasse mit der Übergabe an die jeweilige Methode.

Beispiel 2.8 Wir wollen eine Wertetabelle für beliebige Funktionen ausgeben. Dazu definieren wir zunächst ein Interface `Funktion`, das eine Methode `berechne` deklariert:

```
public interface Funktion  
{  
    public double berechne(double wert);  
}
```

Anschließend verwenden wir dieses Interface in verschiedenen Klassen:

```
class Exp implements Funktion  
{  
    public double berechne(double wert)  
    {  
        return Math.exp(wert);  
    }  
    public String toString()  
    {  
        return "Exponentialfunktion";  
    }  
}  
  
class Sqrt implements Funktion  
{  
    public double berechne(double wert)  
    {  
        return Math.sqrt(wert);  
    }  
    public String toString()  
    {  
        return "Wurzelfunktion";  
    }  
}
```

Die Verwendung der obigen Klassen erfolgt durch Aufruf einer Methode `schreibeTabelle` mit dem Übergabeobjekt vom Typ `Funktion`:

```
public class WerteTabelle
{
    public static void schreibeTabelle(Funktion fkt)
    {
        System.out.println(fkt.toString() + ": ");
        for(double x = 0.0; x <= 5.0; x++)
            System.out.printf(" %f %f\n", x, fkt.berechne(x));
    }

    public static void main(String[] args)
    {
        schreibeTabelle(new Exp());
        schreibeTabelle(new Sqrt());
    }
}
```

Ausgabe

Exponentialfunktion:

```
0,000000 1,000000
1,000000 2,718282
2,000000 7,389056
3,000000 20,085537
4,000000 54,598150
5,000000 148,413159
```

Wurzelfunktion:

```
0,000000 0,000000
1,000000 1,000000
2,000000 1,414214
3,000000 1,732051
4,000000 2,000000
5,000000 2,236068
```

Allgemeine Erklärung

In der `main`-Methode werden die zwei unterschiedlichen Funktionsklassen aufgerufen und mit der Methode `schreibeTabelle` die zugehörige Wertetabelle ausgegeben.

Abstrakte Klassen vs. Schnittstellen Abstrakte Klassen und Schnittstellen haben viele Gemeinsamkeiten und auch Unterschiede. Die Gemeinsamkeit liegt in der Deklaration von Operationen, deren konkrete Implementierung erst später erfolgt. Die Unterschiede sind, dass Klassen beliebig viele Schnittstellen implementieren können, aber nur eine Klasse durch Vererbung erweiterbar ist. Die Vererbung ist über ein „Ist-ein-Verhältnis“ und ein Interface über ein „Hat-ein-Verhältnis“ darstellbar.

Ein vererbtes Objekt ist auch ein Objekt der Oberklasse mit allen vererbten Attributen und Eigenschaften. Ein implementiertes Objekt bei einer Schnittstelle besitzt nur den

Zugriff auf andere Objekte, von denen es nur die Schnittstelle, aber nicht deren interne Arbeitsweise kennt. Die nachträglichen Änderungen an den Schnittstellen sind komplizierter, während es bei abstrakten Klassen zu keiner Anpassung an den Unterklassen kommt.

Eine prinzipielle Empfehlung ist, Vererbung vor Schnittstellenimplementierung zu verwenden. Schnittstellen sollten dann benutzt werden, wenn Objekte aus verschiedenen Klassenhierarchien stammen oder eine gewisse Querschnittsfunktionalität implementiert werden soll. Schnittstellen haben eine ganze Reihe von Nachteilen, die der Entwickler eines Softwaresystems kennen und beachten sollte. Bei der Verwendung von Schnittstellen entstehen Inkompatibilitäten bei der Erweiterung und Veränderung von Schnittstellen in gewissen Bibliotheken. In diesem Fall erhalten Nutzer bei diesen geänderten Schnittstellen eine Fehlermeldung. Ein weiterer Nachteil ist das Auftreten von Redundanzen im Programmcode, da Schnittstellen keine Funktionalität implementieren. Abstrakte Klassen besitzen weiterhin den Vorteil, dass man aus ihnen leicht eine gewöhnliche Klasse machen kann, wenn man eine zusätzliche Funktionalität hinzufügt.

2.4 Objektorientierte Muster

Bei Entwurfsmustern wird das Prinzip einer früheren grundlegenden Lösung auf viele Probleme ähnlicher Art übertragen. *Entwurfsmuster (Design Patterns)* sind bewährte Muster oder Lösungen für nicht triviale Probleme, die immer wieder auf eine bestimmte Art auftreten.

Ein Beispiel ist das Entwurfsmuster des Adapters. Wenn man ein elektrisches Gerät mit einem deutschen Stecker an eine amerikanische Steckdose anschließen will, benötigt dieses einen passenden Adapter. Das gleiche Prinzip lässt sich nun auf Software übertragen. Eine vorhandene Software besteht aus einer großen Anzahl unterschiedlicher Klassen. Leider passt die Schnittstelle der einen Klasse nicht zu der Schnittstelle der anderen. Mithilfe des Entwurfsmusters des Adapters werden zwei Schnittstellen so verbunden, dass zwei Objekte Nachrichten austauschen können.

Das Entwurfsmuster ist ein von einer Programmiersprache unabhängiger Lösungsansatz, welcher das Problem, die Lösungsprinzipien und die Eigenschaften (Vor- und Nachteile) des Ansatzes beschreibt. Die genaue Implementierung bleibt hingegen dem Nutzer des Entwurfsmusters überlassen. Die Wiederverwendbarkeit und das gut erprobte Lösungsmuster spielen dabei eine wichtige Rolle zur Lösung von konkreten und komplexen Problemstellungen.

In dem Buch *Design Patterns. Elements of Reusable Object-Oriented Software* haben die vier Autoren Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides, die sogenannte Gang of Four – GoF (Viererbande), die grundlegendsten 23 Muster beschrieben. Diese Muster sind dadurch zum allgemeinen Sprachgebrauch der Programmierer geworden, mit denen sich viele Probleme abstrakt modellieren lassen.

Entwurfsmuster werden in die folgenden drei Kategorien unterteilt:

- **Erzeugungsmuster** beschreiben, wie Objekte erzeugt werden. In Java werden Objekte mit dem `new`-Operator erzeugt. In manchen Anwendungen will man jedoch aus verschiedenen Gründen (z. B. Flexibilität, Verhinderung der Objektanlegung von außen) die Objekterzeugung in einer Methode `getInstance` auslagern.
- **Verhaltensmuster** beschreiben, wie Klassen und Objekte zusammenarbeiten. Das Verhaltensmuster hat die Aufgabe, einen Prozess zu entwickeln, sodass sich beispielsweise zwei Objekte gegenseitig über ihren Zustand informieren.
- **Strukturmuster** beschreiben, wie Klassen zusammengesetzt werden. Strukturmuster haben die Aufgabe, die einzelnen Teile geeignet zusammenzubringen, um so größere Einheiten zu bilden.

Der erste Schritt bei der Verwendung eines Entwurfsmuster ist die Klassifizierung des Problems in die drei gegebenen Musterkategorien. Anschließend muss aus der Menge der vorhandenen Muster der jeweiligen Kategorie das passende ausgewählt werden. Jedes Entwurfsmuster besteht dabei aus einem zu lösenden Problem, einem abstrakten Lösungsprinzip sowie gewissen Implementierungsdetails. Bei der Anwendung des Musters auf die konkrete Anwendung werden die Implementierungsdetails umgesetzt und gegebenenfalls an die Problemstellung angepasst. Eine sorgfältige Dokumentation und Bezeichnung der entwickelten Klassen und Methoden ist von großer Bedeutung.

Bei der Verwendung von Entwurfsmustern ist ein wesentlicher Punkt, dass diese keinen Selbstzweck besitzen. Ein Muster sollte nur dann verwendet werden, wenn es tatsächlich sinnvoll ist und auch einen gewissen Mehrwert bringt. Der Hauptarbeitsschritt ist die Auswahl eines passenden Musters zur vorhandenen Anwendung. Anschließend muss geprüft werden, welche Konsequenzen sich aus der Anwendung des Musters ergeben.

Die meisten Muster benötigen die Entwicklung von zusätzlichen Klassen. Keinesfalls sollte man viele verschiedene Muster in einer relativ kleinen Anwendung verwenden, da man damit ein unnötig aufgeblähtes und schlecht wartbares System erhält.

Wir stellen in den folgenden Abschnitten einige oft verwendete Entwurfsmuster aus den oben beschriebenen drei Kategorien der Erzeugungsmuster, Verhaltensmuster und Strukturmuster vor. Für jedes der ausgewählten Muster betrachten wir ein konkretes übersichtliches Implementierungsbeispiel.

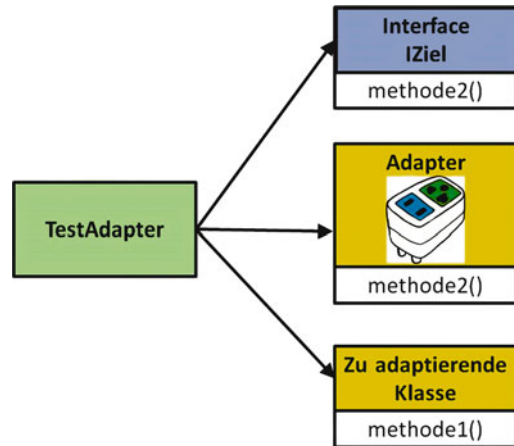
2.5 Strukturmuster

Die Strukturmuster beschreiben die Art und Weise, wie Klassen zusammengesetzt werden, um aus einzelnen Teile eine komplexe Konstruktion zu bilden.

2.5.1 Adapter

Auf Reisen sind Adapter ein sehr nützliches Hilfsmittel, um ein elektronisches Gerät mit deutschem Stecker an eine ausländische Steckdose anzuschließen. In der Softwaretechnik

Abb. 2.2 Strukturmuster des Adapters



passt das Adaptermuster (Adapter Pattern) eine vorhandene unpassende Schnittstelle an die gewünschte Form des Anwenders an. Mit diesem Entwurfsmuster ist es möglich, dass verschiedene Klassen zusammenarbeiten, die ansonsten wegen inkompatibler Schnittstellen nicht dazu in der Lage wären.

Problem Vorhandene Klassen, deren Schnittstellen nicht zusammenpassen.

Ziel Anpassung der inkompatiblen Schnittstelle der vorhandenen Klasse an die gewünschte Form mittels eines Adapters.

Lösungsprinzip Die bereits existierende Klasse, die den gewünschten Anforderungen entspricht, aber eine unpassende Schnittstelle für den Nutzer besitzt, wird mithilfe einer Adapterklasse angepasst. Dieser Adapter ergänzt die bestehende Klasse um eine Art von Übersetzung der bereits existierenden Methoden. Damit kann die vorhandene Klasse ohne weitere Änderungen verwendet werden.

Implementierung Vorhanden ist eine bereits existierende Klasse `ZuAdaptierende Klasse` die bereits die gewünschte Funktionalität des Anwenders mit einer unpassenden Schnittstelle bereitstellt (siehe Abb. 2.2). Die Implementierung besteht aus der Definition eines Interface `IZiel`, welche die vorgegebene Aufrufschnittstelle der Anwendung `TestAdapter` definiert. Die neue Klasse `Adapter` hat die Aufgabe, die Anpassung an die benötigte Schnittstelle der Anwendung vorzunehmen. Ein Objekt der Klasse `Adapter` übernimmt dann die Verbindung zwischen dem Anwender und einem Objekt der zu adaptierenden Klasse.

Beispiel Die Aufgabe besteht in dem Auslesen von Artikeldaten aus einer csv-Datei. Die Klasse `ZuAdaptierendeKlasse` zum Auslesen der Daten ist bereits implemen-

tiert worden, jedoch passt die Aufrufschnittstelle nicht zur neuen Anwendung. Um die Klasse zur Anwendung kompatibel zu machen, wird eine Klasse `Adapter` geschrieben.

Wir verwenden hierzu die folgenden Komponenten:

- Klasse `Artikel`: Hilfsklasse zur Speicherung des Namens und der Nummer der Artikeldaten.

```
class Artikel
{
    private String name;
    private String nummer;

    public Artikel(String name, String nummer)
    {
        this.name = name;
        this.nummer = nummer;
    }

    public void ausgabe()
    {
        System.out.println("Artikel " + name + " mit Nummer " + nummer);
    }
}
```

- Klasse `ZuAdaptierendeKlasse`: adaptierende Klasse, die eine csv-Datei mit Artikeldaten aus einer Datei einliest.

```
class ZuAdaptierendeKlasse
{
    public Vector<String []> leseDatei(String file) throws FileNotFoundException
    {
        Vector<String []> artikel = new Vector<String []>();
        File datei = new File(file);
        Scanner in = new Scanner(datei);
        while (in.hasNextLine())
        {
            String string = in.nextLine();
            String name[] = string.split(";");
            if (name.length >= 2)
                artikel.add(new String [] {name[0], name[1]});
        }
        in.close();

        return artikel;
    }
}
```

Die Methode `leseDatei` gibt ein Array vom Typ `Vector` mit Elementen vom Typ `String` zurück. Diese Implementierung ist jedoch inkompatibel, da ein Array vom Typ `Vector` mit Elementen vom Typ `Artikel` vom Anwender gefordert ist.

- Interface `IZiel`: Interface ist die Schnittstelle aller Klassen, die Artikeldaten einlesen können.

```
public interface IZiel
{
    public Vector<Artikel> leseArtikel();
}
```

Die Methode `leseArtikel` hat die Aufgabe, ein Array vom Typ `Vector` mit Elementen vom Typ `Artikel` zurückzugeben.

- Klasse `Adapter`: Adapter für die Klasse `ZuAdaptierendeKlasse`, die durch Implementierung des Interface `IZiel` erstellt wird.

```
class Adapter implements IZiel
{
    private String file;

    public Adapter(String file)
    {
        this.file = file;
    }

    public Vector<Artikel> leseArtikel()
    {
        ZuAdaptierendeKlasse zA = new ZuAdaptierendeKlasse();
        Vector<String []> artikel = zA.leseDatei(file);

        Vector<Artikel> artikelVector = new Vector<Artikel>();
        for(String name[] : artikel)
            artikelVector.add (new Artikel(name[0], name[1]));

        return artikelVector;
    }
}
```

Das Einlesen der Artikeldaten erfolgt durch die vorhandene Klasse `ZuAdaptierendeKlasse`. Danach werden die erhaltenen Daten so aufbereitet, dass sie auf die Schnittstelle des Anwenders passen.

- Klasse `TestAdapter`: Anwender, der eine csv-Datei mit Artikeldaten mithilfe der Klasse `Adapter` einliest und diese auf der Konsole ausgibt.

```
public class TestAdapter
{
    public static void main (String[] args)
    {
        IZiel adapter = new Adapter("Artikel.csv");
        Vector<Artikel> artikel = adapter.leseArtikel();
        for (Artikel a : artikel)
            a.ausgabe();
    }
}
```

Ausgabe

Artikel Schraube mit Nummer HS1234b

Artikel Mutter mit Nummer TI3456c

Anwendungen

- Zusammensetzung von verschiedenen Klassen mit unterschiedlichen Schnittstellen,
- Anpassung von Klassen einer Klassenbibliothek an eine Schnittstelle,
- Übertragung von grafischen Oberflächen auf eine neue Plattform.

Bewertung

- **Vorteile:** effiziente Kommunikation zwischen zwei unabhängigen Softwarekomponenten, Erweiterung um weitere Funktionalitäten, Anpassen von vorhandenen Klassen;
- **Nachteile:** höherer Aufwand durch zusätzliche Zwischenschritte bei komplexen Adaptern, schlechte Wiederverwendbarkeit der Adapter.

2.5.2 Fassade

Viele technische Produkte sind heute sehr komplex und teilweise schwierig zu bedienen. Viele Hersteller erleichtern ihren Kunden den Zugriff auf das System, indem sie häufig verwendete Komponenten unter einer neuen Funktion zusammenfassen. Damit erzeugen sie eine sogenannte Fassade (Facade Pattern), damit der Anwender sich nicht mehr mit den konkreten Details beschäftigen muss. Die Fassade erledigt die konkreten Schritte der Subsysteme und bietet eine einheitliche und einfachere Schnittstelle für die Systeme an. Weiterhin besteht jederzeit die Möglichkeit auf die einzelnen Komponenten des Systems ohne die Fassade zuzugreifen.

Auch viele komplizierte Softwaresysteme bestehen meist aus mehreren Subsystemen, die miteinander interagieren. Eine Zerlegung eines Systems in Subsysteme dient der Übersichtlichkeit, der besseren Wartbarkeit und auch dem schnelleren Zugriff auf Funktionen des Systems. Der Zugriff auf das System wird durch das Fassadenmuster für den Anwender deutlich vereinfacht, in dem eine gewisse Funktionalität durch eine vereinfachte Schnittstellen zur Verfügung gestellt wird.

Problem Kompliziertes System, das aus einer Menge von Schnittstellen besteht.

Ziel Definition einer vereinfachten Schnittstelle zum Zugriff auf die Klassen eines Subsystems.

Lösungsprinzip Die neue und vereinfachte Schnittstelle zwischen dem Anwender und den Klassen eines Subsystems wird in einer Fassadenklasse bereitgestellt. Die Fassa-



Abb. 2.3 Strukturmuster der Fassade

de versteckt die einzelnen Details der internen Struktur eines Subsystems. Damit ist es möglich, den Aufbau eines Subsystems zu ändern, ohne dass der Anwender Änderungen vornehmen muss.

Implementierung Vorhanden ist eine Menge von verschiedenen Subsystemen mit den Klassen `Subsystem1`, `Subsystem2` usw. Implementiert wird eine Klasse `Fassade`, welche die Schnittstelle zu den Subsystemen herstellt (siehe Abb. 2.3). Der Anwender `TestFassade` legt ein Objekt der Klasse `Fassade` an und ruft anschließend verschiedene Methoden dieser Klasse auf. Dieser Aufruf wird an die Objekte der Subsysteme weitergeleitet. Die Resultate der Subsystemklasse werden auf dem umgekehrten Weg an den Anwender zurückgeliefert.

Beispiel Die Aufgabe ist die Verwaltung von Lagern mittels einer Fassade vorzunehmen. Zur Übersichtlichkeit verwenden wir hier nur zwei Lager, es ist jedoch eine beliebige Anzahl von Lagern möglich. Diese Lager stellen die Subsystemklassen dar. Mit der Klasse `Fassade` wird eine Schnittstelle implementiert, bei der in allen Lagern immer gleich viele Teile entnommen bzw. hinzugefügt werden. Ebenso wäre eine andere Definition einer Fassade möglich, die dann in einer zweiten Fassadenklasse implementiert werden kann.

Wir verwenden hierzu die folgenden vier Klassen:

- Klasse `Lager1`: Repräsentation der ersten Subsystemklasse:

```

class Lager1
{
    int anzahl = 0;

    public void aufstocken(int n)
    {
        anzahl = anzahl + n;
        System.out.println("Das Lager 1 wurde um " + anzahl + " Stück aufgestockt.");
    }
    public void entnehmen(int n)
    {
        anzahl = anzahl - n;
        System.out.println("Dem Lager 1 wurden " + anzahl + " Stück entommen.");
    }
}
  
```

- Klasse Lager2: Repräsentation der zweiten Subsystemklasse:

```
class Lager2
{
    int anzahl = 0;

    public void aufstocken(int n)
    {
        anzahl = anzahl + n;
        System.out.println("Das Lager 2 wurde um " + anzahl + " Stück aufgestockt.");
    }
    public void entnehmen(int n)
    {
        anzahl = anzahl - n;
        System.out.println("Dem Lager 2 wurden " + anzahl + " Stück entommen.");
    }
}
```

- Klasse Fassade: Darstellung der Fassade, die die gemeinsame Aufstockung und Entnahme von Lagerteilen aus allen zwei Lagern mit den Methoden lagerAufstocken und lagerEntnehmen vornimmt.

```
class Fassade
{
    private Lager1 lager1;
    private Lager2 lager2;

    public Fassade()
    {
        lager1 = new Lager1();
        lager2 = new Lager2();
    }

    public void lagerAufstocken(int anzahl)
    {
        System.out.println("Der Bestand aller Lager wird aufgestockt:");
        lager1.aufstocken(anzahl);
        lager2.aufstocken(anzahl);
    }
    public void lagerEntnehmen(int anzahl)
    {
        System.out.println("Zur Produktion werden allen Lagern Teile entnommen:");
        lager1.entnehmen(anzahl);
        lager2.entnehmen(anzahl);
    }
}
```

- Klasse TestFassade: Anwender, der ein Objekt der Klasse Fassade anlegt und damit die Lager mit den von der Fassade bereitgestellten Methoden verwaltet.

```
public class TestFassade
{
    public static void main (String[] args)
    {
        Fassade fassade = new Fassade();
        fassade.lagerAufstocken(20);
        fassade.lagerEntnehmen(10);
        fassade.lagerAufstocken(5);
    }
}
```

Ausgabe

Der Bestand aller Lager wird aufgestockt:
Das Lager 1 wurde um 20 Stück aufgestockt.
Das Lager 2 wurde um 20 Stück aufgestockt.
Zur Produktion werden allen Lagern Teile entnommen:
Dem Lager 1 wurden 10 Stück entnommen.
Dem Lager 2 wurden 10 Stück entnommen.
Der Bestand aller Lager wird aufgestockt:
Das Lager 1 wurde um 15 Stück aufgestockt.
Das Lager 2 wurde um 15 Stück aufgestockt.

Anwendungen

- Definition einer einfacheren Schnittstelle für Softwaresysteme,
- Verbergung der inneren Details von komplexen Subsystemen,
- Erstellung von vereinfachten Oberflächenelementen (z. B. `JOptionPane`).

Bewertung

- **Vorteile:** vereinfachter Zugriff auf Funktionen von Subsystemen, Erleichterung des Austauschs von Subsystem und Anwender, Weiterentwickelbarkeit der Systeme ohne Auswirkung auf Anwender, lose Kopplung der Systeme.
- **Nachteile:** Einschränkung der Funktionalität der Subklasse. Änderungen der Schnittstellen des Subsystems müssen auch in der Fassade berücksichtigt werden.

2.5.3 Dekorierer

Viele Produkte wie Häuser, Autos usw. bestehen heute aus wenigen Grundmodellen mit zahlreichen Ausstattungsvarianten. Der Dekorierer (Decorator Pattern) hat die Aufgabe, gewisse Objekte um Aussehen oder Zuständigkeiten zu erweitern. Dabei soll die Erweiterung das Objekt nicht durch Unterklassenbildung verändern. Beim Dekorierermuster

kann während der Laufzeit eine Zusatzfunktionalität zu einem existierenden Objekt ergänzt werden, d. h., ein Objekt wird um eine Funktionalität dekoriert.

Problem Fehlende Funktionalität für ein vorhandenes Objekt.

Ziel Hinzufügung einer zusätzlichen Funktionalität zu einem vorhandenen Objekt während der Laufzeit.

Lösungsprinzip Vorhanden sind eine Menge zu erweiternder Komponenten und eine Menge von verschiedenen Dekorierern. Ein Dekorierer hat dabei die Aufgabe, zu der existierenden Komponente eine Zusatzfunktionalität in Form eines dekorierenden Objekts hinzuzufügen. Dem Anwender wird dann eine einheitliche Schnittstelle für die Funktionalität zur Verfügung gestellt. Hierbei merkt das zu dekorierende Objekt gar nicht, dass es dekoriert wird.

Implementierung Das Interface `IKomponente` ist die Basis aller konkreten Komponenten `KonkreteKomponente1`, `KonkreteKomponente2` usw. und der abstrakten Klasse `AbstrakterDekorierer` zum Dekorieren von Komponenten (siehe Abb. 2.4). Von der abstrakten Klasse `AbstrakterDekorierer` werden verschiedene Subklassen `Dekorierer1`, `Dekorierer2` usw. abgeleitet, um gewisse Zusatzfunktionalitäten bei Objekten zu dekorieren. Für eine konkrete Anwendung definiert der Anwender `TestDekorierer` ein Objekt dieser Subklassen und übergibt die zugehörige zu dekorierende Komponente der Klasse `Komponente`. Das Objekt der Klasse `Komponente` gibt das Ergebnis durch das Objekt der Klasse `Dekorierer1`, `Dekorierer2` usw. an den Anwender zurück.

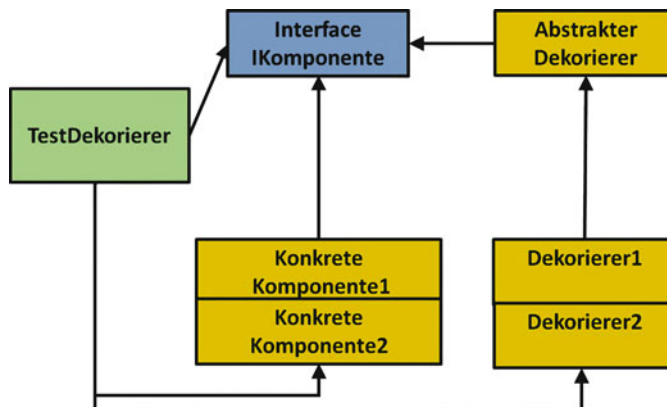


Abb. 2.4 Strukturmuster des Dekorieres

Beispiel Die Aufgabe besteht darin, gewisse Produkte (Komponenten) mit verschiedenen Ausstattungen (Dekorierer) zu dekorieren. Damit kann eine ganze Reihe von verschiedenen Produkten erzeugt werden, für die jeweils keine eigene Klasse geschrieben werden muss.

Wir verwenden hierzu die folgenden Komponenten:

- Interface `IProdukt`: Das Interface mit den beiden Methoden `getPreis` und `getMerkmale` dient zum Aufsummieren des Preises der Ausstattung und zur Ausgabe von Informationen.

```
interface IProdukt
{
    public double getPreis();
    public void getMerkmale();
}
```

- Klasse `Produkt1` und `Produkt2`: Repräsentierung der zu dekorierenden konkreten Komponente mit den vorhandenen Grundkosten.

```
class Produkt1 implements IProdukt
{
    public double getPreis()
    {
        return 320;
    }
    public void getMerkmale()
    {
        System.out.print("Produkt 1");
    }
}
```

```
class Produkt2 implements IProdukt
{
    public double getPreis()
    {
        return 150;
    }
    public void getMerkmale()
    {
        System.out.print("Produkt 2");
    }
}
```

- Klasse `Ausstattung`: abstrakter Dekorierer zur Definition der Instanzvariablen `produkt` der Klasse `Produkt`.

```
abstract class Ausstattung implements IProdukt
{
    protected IProdukt produkt;

    public Ausstattung (IProdukt produkt)
    {
        this.produkt = produkt;
    }
}
```

- Klasse Ausstattung1 und Ausstattung2: konkreter Dekorierer, der von der abstrakten Klasse Ausstattung abgeleitet wird und die konkreten Methoden implementiert.

```
class Ausstattung1 extends Ausstattung
{
    public Ausstattung1(IProdukt produkt)
    {
        super(produkt);
    }

    public double getPreis()
    {
        return produkt.getPreis() + 15.50;
    }
    public void getMerkmale()
    {
        produkt.getMerkmale();
        System.out.print(" mit Ausstattung1");
    }
}
```

```
class Ausstattung2 extends Ausstattung
{
    public Ausstattung2(IProdukt produkt)
    {
        super(produkt);
    }

    public double getPreis()
    {
        return produkt.getPreis() + 50;
    }
    public void getMerkmale()
    {
        produkt.getMerkmale();
        System.out.print(" mit Ausstattung2");
    }
}
```

- Klasse `TestDekorierer`: Definition verschiedener Produkte mit unterschiedlicher Ausstattung.

```
public class TestDekorierer
{
    public static void main(String[] args)
    {
        IProdukt produkt = new Produkt1();

        produkt = new Ausstattung1(produkt);
        produkt.getMerkmale();
        System.out.println("; Preis " + produkt.getPreis() + " Euro");

        produkt = new Ausstattung2(produkt);
        produkt.getMerkmale();
        System.out.println("; Preis " + produkt.getPreis() + " Euro");

        produkt = new Ausstattung1(new Produkt2());
        produkt.getMerkmale();
        System.out.println("; Preis " + produkt.getPreis() + " Euro");
    }
}
```

Ausgabe

Produkt 1 mit Ausstattung1; Preis 335.50 Euro

Produkt 1 mit Ausstattung1 mit Ausstattung2; Preis 385.50 Euro

Produkt 2 mit Ausstattung1; Preis 165.50 Euro

Anwendungen

- Hinzufügen von Zusatzfunktionalitäten für Klassen, ohne die Erzeugung neuer Unterklassen,
- Austausch von Funktionalitäten durch Dekorierer während der Laufzeit,
- Erweiterung grafischer Oberflächen durch gewisse Zusatzfunktionen,
- Erstellung umfangreicher Klassenbibliotheken mit ähnlichen Funktionalitäten.

Bewertung

- **Vorteile:** stabiles und flexibles System, dynamische Erweiterung von Objekten zur Laufzeit, Erweiterung von Klassen einer Vererbungshierarchie, Kombination unterschiedlicher Dekorierer für beliebige Funktionalitäten.
- **Nachteile:** höherer Aufwand durch zahlreiche Delegationsmethoden. Fehlersuche in kombinierten Dekorierern ist oft schwierig.

2.6 Verhaltensmuster

Verhaltensmuster beschreiben, wie Klassen und Objekte zusammenarbeiten, sodass sich beispielsweise zwei Objekte gegenseitig über ihren Zustand informieren.

2.6.1 Schablone

Eine Schablone ist ein ausgeschnittenes Muster zur Herstellung gleichgestaltiger Dinge aus speziellen Materialien. Die Dinge können anschließend bearbeitet werden und mit Verzierungen versehen werden. In der Softwaretechnik wird unter der Schablonenmethode (Template Method Pattern) ein Entwurfsmuster verstanden, bei dem das Grundgerüst eines Algorithmus mit den einzelnen Operationen definiert ist. Die konkreten Umsetzungsschritte werden in speziellen Unterklassen implementiert. Die Schablonenmethode ist eine der wichtigsten Entwurfsmuster für Algorithmen, die wir in diesem Buch an vielen verschiedenen Stellen verwenden werden.

Problem Ein Algorithmus soll in verschiedenen Ausprägungen implementiert werden, sodass Detailschritte leicht auswechselbar sind.

Ziel In einer Basisklasse wird die allgemeine Struktur eines Algorithmus festgelegt, und die variablen Teile werden in diversen Unterklassen implementiert.

Lösungsprinzip Eine Schablonenmethode definiert nur das Gerüst eines Algorithmus und verwendet dazu abstrakte Einschubmethoden. Diese Einschubmethoden werden in unterschiedlichen, von der Basisklasse unabhängigen Unterklassen implementiert. Diese nicht öffentlichen Einschubmethoden werden nur von der Schablonenmethode selbst aufgerufen und sind speziell für den jeweiligen Algorithmus geschrieben. Damit realisiert jede Unterklasse eine andere Variante des Basisalgorithmus.

Implementierung Eine abstrakte Klasse `AbstrakteKlasse` definiert unter Verwendung einer Schablonenmethode `schablonenmethode` und verschiedener abstrakter Einschubmethoden `einschubmethode1`, `einschubmethode2` usw. das Gerüst zur Festlegung der groben Schritte eines Algorithmus (siehe Abb. 2.5). Die Schablonenmethode `schablonenmethode` enthält dabei so viel identischen Code wie möglich. Eine Reihe von verschiedenen konkreten Subklassen `KonkreteKlasse1`, `KonkreteKlasse2` usw. legt dann die spezifischen Schritte des Algorithmus in den konkreten Einschubmethoden fest.

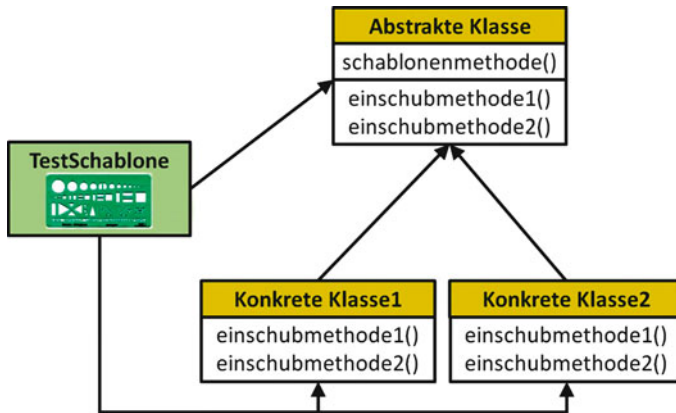


Abb. 2.5 Verhaltensmuster der Schablone

Beispiel Wir demonstrieren die Schablonenmethode an einem sehr einfachen Grundgerüst mit zwei konkreten Klassen. In den nachfolgenden Kapiteln werden wir noch zahlreiche weitere konkrete Beispiele dazu vorstellen.

Wir verwenden hierzu die folgenden vier Klassen:

- Klasse `AbstrakteKlasse`: abstrakte Klasse mit Schablonenmethode `schablonenmethode` und abstrakten Schablonenmethoden `einschubmethode1` und `einschubmethode2`.

```

abstract class AbstrakteKlasse
{
    public void schablonenmethode()
    {
        System.out.println("Grundgerüst des Algorithmus");
        einschubmethode1();
        einschubmethode2();
    }

    abstract protected void einschubmethode1();
    abstract protected void einschubmethode2();
}
  
```

- Klasse `KonkreteKlasse1` und `KonkreteKlasse2`: Subklassen mit den jeweiligen konkreten Einschubmethoden.

```

class KonkreteKlasse1 extends AbstrakteKlasse
{
    protected void einschubmethode1()
    {
        System.out.println("Spezieller Teil 1 des Algorithmus 1");
    }
    protected void einschubmethode2()
    {
        System.out.println("Spezieller Teil 2 des Algorithmus 1");
    }
}

class KonkreteKlasse2 extends AbstrakteKlasse
{
    protected void einschubmethode1()
    {
        System.out.println("Spezieller Teil 1 des Algorithmus 2");
    }
    protected void einschubmethode2()
    {
        System.out.println("Spezieller Teil 2 des Algorithmus 2");
    }
}

```

- Klasse TestSchablone: Anwender ruft die Schablonenmethode schablonenmethode auf, die dann die jeweiligen konkreten Einschubmethoden verwendet.

```

public class TestSchablone
{
    public static void main (String args[])
    {
        KonkreteKlasse1 algo1 = new KonkreteKlasse1();
        algo1.schablonenmethode();

        KonkreteKlasse2 algo2 = new KonkreteKlasse2();
        algo2.schablonenmethode();
    }
}

```

Ausgabe

```

Grundgerüst des Algorithmus
Spezieller Teil 1 des Algorithmus 1
Spezieller Teil 2 des Algorithmus 1
Grundgerüst des Algorithmus
Spezieller Teil 1 des Algorithmus 2
Spezieller Teil 2 des Algorithmus 2

```

Anwendungen

- Implementierung von verschiedenen Arten von Algorithmen, bei denen mehrere Arbeitsschritte gleich sind.

Bewertung

- **Vorteile:** keine Abhängigkeit der Basisklasse von ihren Unterklassen. Algorithmus kann im Groben ohne die Details festgelegt werden. Sehr gute Wiederverwendung.
- **Nachteile:** ohne.

2.6.2 Strategie

Im Allgemeinen versteht man unter einer Strategie eine geplante Verhaltensweise einer Person oder eines Unternehmens zur Erreichung seiner oder ihrer Ziele. In der Softwaretechnik versteht man unter dem Entwurfsmuster Strategie (Strategy Pattern) eine Familie von austauschbaren Algorithmen, die je nach der gewünschten Verhaltensweise einsetzbar ist. Damit soll das Verhalten eines Objekts (Kontextobjekt) flexibel an die zu nutzende Strategie angepasst werden. Dieses Kontextobjekt darf dabei nicht von einer speziellen Implementierung des Algorithmus abhängen.

Problem Starre Handhabung der Verhaltensweise eines Algorithmus für gleiche Objekte.

Ziel Austausch eines Algorithmus in Form einer Kapsel.

Lösungsprinzip Jeder einzelne Algorithmus wird als Strategie in einer eigenen Klasse implementiert, um ihn als Kapsel austauschbar zu machen. Der benötigte Algorithmus kann dann vom Anwender des Kontextobjekts gezielt ausgewählt werden.

Implementierung Für die Austauschbarkeit der verschiedenen Algorithmen wird ein Interface `IStrategie` verwendet (siehe Abb. 2.6). Die Familie von austauschbaren Algorithmen verwendet das Interface `IStrategie` und implementiert die verschiedenen Strategien in den Klassen `KonkreteStrategie1`, `KonkreteStrategie2`

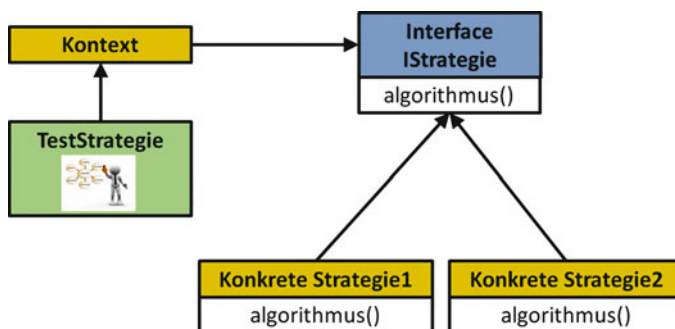


Abb. 2.6 Verhaltensmuster der Strategie

usw. Die Klasse `Kontext` implementiert das Kontextobjekt durch Verwendung des Interface `IStrategie`, in dem diese Klasse eine Referenz vom Typ des Interface `IStrategie` enthält. Der Anwender `TestStrategie` setzt dann mit einer Methode `setzeStrategie` die gewählte Strategie für das Kontextobjekt und ruft dann die jeweilige Methode des Objektes der Klasse `Kontext` zur Ausführung der gewählten Strategie auf.

Beispiel Die Aufgabe besteht darin eine Klasse zum Speichern von Personendaten zu definieren, und dann die Personendaten je nach gewählter Strategie in verschiedenen Formaten auszugeben.

Wir verwenden hierzu die folgenden Komponenten:

- Interface `IStrategie`: Interface dient als Schnittstelle für alle konkreten Strategien durch Definition der Methode `algorithmus`, in der dann die jeweilige konkrete Strategie implementiert wird.

```
interface IStrategie
{
    public void algorithmus(String name, String vorname, int jahr);
}
```

- Klasse `Person`: Definition des Kontextobjekts zur Speicherung der Personendaten. In der Methode `setzeStrategie` wird vom Anwender die gewünschte Strategie gesetzt. Die Methode `anwendenStrategie` ruft dann den Algorithmus dieser zugehörigen Strategie auf.

```
class Person
{
    private IStrategie strategie = null;
    private String name, vorname;
    private int jahr;

    public Person(String name, String vorname, int jahr)
    {
        this.name = name;
        this.vorname = vorname;
        this.jahr = jahr;
    }
    public void setzeStrategie(IStrategie strategie)
    {
        this.strategie = strategie;
    }
    public void anwendenStrategie()
    {
        strategie.algorithmus(name, vorname, jahr);
    }
}
```

- Klasse `KonkreteStrategie1` und `KonkreteStrategie2`: Die beiden Klassen definieren jeweils ein Ausgabeformat für die Personendaten.

```
class KonkreteStrategie1 implements IStrategie
{
    public void algorithmus(String name, String vorname, int jahr)
    {
        System.out.println(vorname + " " + name + " (geb. " + jahr + ")");
    }
}

class KonkreteStrategie2 implements IStrategie
{
    public void algorithmus(String name, String vorname, int jahr)
    {
        System.out.println(name + ", " + vorname + ", Jg. " + jahr);
    }
}
```

- Klasse `TestStrategie`: Erzeugung einer Instanz der Klasse `Person` und setzen der Formatierungsstrategie mit der Methode `setzeStrategie`.

```
public class TestStrategie
{
    public static void main (String[] args)
    {
        Person person = new Person("Schmidt", "Anton", 1965);
        person.setzeStrategie(new KonkreteStrategie1());
        person.anwendenStrategie();
        person.setzeStrategie(new KonkreteStrategie2());
        person.anwendenStrategie();
    }
}
```

Ausgabe

Anton Schmidt (geb. 1965)
Schmidt, Anton, Jg. 1965

Anwendungen

- Implementierung einer Anwendung, die über verschiedene Algorithmen besteht,
- Auswahl eines Layoutmanagers einer grafischen Benutzeroberfläche.

Bewertung

- **Vorteile:** flexible Einsatzmöglichkeit bei der Verwendung unterschiedlicher Algorithmen, gute Wiederverwendbarkeit der Algorithmen, sehr gute Übersichtlichkeit des Programmcodes.

- **Nachteile:** erhöhter Aufwand zur Kommunikation zwischen Kontext und Strategie. Anwendung muss die unterschiedlichen Strategien eines Kontextobjekts kennen.

2.6.3 Beobachter

Bei dem Muster des Beobachters interessieren wir uns für die zielgerichtete Verteilung von Informationen. Ein Beispiel ist ein Newsletter, bei dem man sich anmeldet, um eine neue Wohnung oder einen Job zu finden. Anschließend bekommt man genau die Informationen zugeschickt, die einen interessieren. Das Muster des Beobachters (Observer Pattern) hat bei einem beobachtbaren Objekt die Aufgabe, die abhängigen Objekte (Beobachter) von einer Änderung seines Zustands zu informieren, um so eine automatische Aktualisierung durchzuführen.

Problem Benachrichtigung aller Beobachter bei Zustandsänderung des beobachteten Objekts.

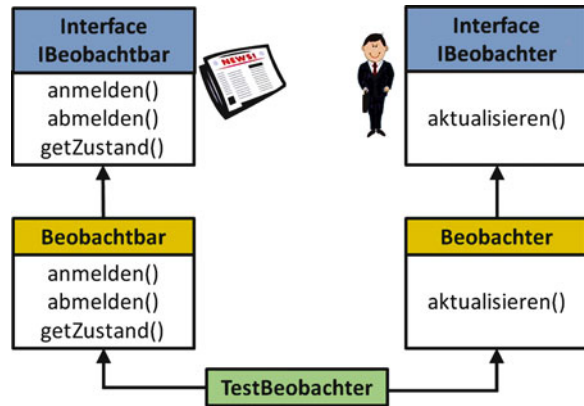
Ziel Die Menge aller Beobachter, die bei einer Zustandsänderung des beobachtbaren Objekts benachrichtigt werden, soll dynamisch veränderbar sein, d.h., Objekte können hinzukommen und zu einem späteren Zeitpunkt auch wieder entfernt werden.

Lösungsprinzip Die Struktur des Beobachters besteht aus einem beobachtbaren Objekt mit gewissen Daten und mehreren Beobachtern, die auf Änderungen der Daten reagieren müssen. Dabei sind die Beobachter erst zur Laufzeit nach ihrer Anmeldung beim beobachtbaren Objekt bekannt. Anschließend erhält der Beobachter so lange bei jeder Zustandsänderung des zu beobachtenden Objekts eine Nachricht, bis er sich selbst wieder abmeldet. Der Beobachter ist dann selbst für das Abfragen des neuen Zustands verantwortlich.

Implementierung Für das beobachtbare Objekt der Klasse `Beobachtbar` und den Beobachter der Klasse `Beobachter` wird jeweils ein Interface `IBeobachtbar` bzw. `IBeobachter` erstellt (siehe Abb. 2.7). Das Interface `IBeobachtbar` für das beobachtbare Objekt enthält die Operationen für das An- und Abmelden, `anmelden` und `abmelden` der Beobachter sowie die abstrakte Methode `getZustand` zum Abruf der geänderten Daten.

Das Interface `IBeobachter` des Beobachters enthält die Methode `aktualisieren` zur Mitteilung einer Zustandsänderung. Beim Aufruf der Methode `aktualisieren` übergibt ein beobachtbares Objekt eine Referenz auf sich selbst. Damit kann der aufgerufene Beobachter mittels dieser Referenz die Methode `getZustand` bei dem beobachteten Objekt aufrufen und sich den neuen Zustand abholen. Diese Referenz ist notwendig, damit ein Beobachter, der mehrere Objekte gleichzeitig beobachtet, weiß, welches dieser Objekte seinen Zustand geändert hat.

Abb. 2.7 Verhaltensmuster des Beobachters



Ein beobachtbares Objekt besitzt eine Liste von Referenzen auf Objekte. Wenn sich ein Beobachter anmeldet, so wird ein neuer Eintrag hinzugefügt. Bei der Abmeldung wird der Eintrag wieder aus der Liste entfernt.

Falls sich die Daten eines beobachtbaren Objekts geändert haben, ruft dieses seine nicht öffentliche Methode `informiere` auf, um alle angemeldeten Beobachter durch den Aufruf ihrer Methode `aktualisieren` zu benachrichtigen und dabei eine Referenz auf sich selbst zu übergeben. Ein Beobachter holt sich mithilfe der Methode `getZustand` vom beobachtbaren Objekt die Information über den neuen Zustand ab.

Beispiel Die Aufgabe besteht darin, einen Newsletter zu implementieren, der seine Abonnenten verwaltet und diese benachrichtigt, wenn neue Informationen vorliegen.

Wir verwenden hierzu die folgenden Komponenten:

- **Interface IBeobachter:** Interface dient als Schnittstelle für einen Beobachter, der durch die Methode `aktualisieren` über Änderungen informiert wird.

```

public interface IBeobachter
{
    public void aktualisieren(IBeobachtbar b);
}
  
```

- **Interface IBeobachtbar:** Interface dient als Schnittstelle für das beobachtbare Objekt, das seine Beobachter mit den Methoden `anmelden` und `abmelden` verwaltet. Mit der Methode `getZustand` informieren sich die angemeldeten Beobachter über die Änderungen des beobachtbaren Objekts.

```

public interface IBeobachtbar
{
    public void anmelden(IBeobachter beobachter);
    public void abmelden(IBeobachter beobachter);
    public String getZustand();
}
  
```

- Klasse `Abonnent`: Beobachter durch Implementierung des Interface `IBeobachter`.

```
public class Abonnent implements IBeobachter
{
    private String name;

    public Abonnent(String name)
    {
        this.name = name;
    }

    public void aktualisieren (IBeobachtbar b)
    {
        System.out.println("Eine neue Nachricht fuer " + name + ": " + b.getZustand());
    }
}
```

- Klasse `Newsletter`: beobachtbar durch Implementierung des Interface `IBeobachtbar`.

```
import java.util.Vector;
public class Newsletter implements IBeobachtbar
{
    private Vector<IBeobachter> abonntenen = new Vector<IBeobachter>();

    private String info;

    public void abmelden(IBeobachter beobachter)
    {
        abonntenen.remove (beobachter);
    }
    public void anmelden(IBeobachter beobachter)
    {
        abonntenen.add (beobachter);
    }
    public String getZustand()
    {
        return info;
    }

    public void aendereZustand(String neueInfo)
    {
        info = neueInfo;
        informiere();
    }
    private void informiere()
    {
        for (IBeobachter beobachter : abonntenen)
        {
            beobachter.aktualisieren(this);
        }
    }
}
```

- Klasse `TestBeobachter`: Definition eines Newsletters mit einigen Beobachtern.

```
public class TestBeobachter
{
    public static void main (String[] args)
    {
        Newsletter newsletter = new Newsletter();
        Abonnent a1 = new Abonnent("Erwin");
        Abonnent a2 = new Abonnent("Mike");

        newsletter.anmelden(a1);
        newsletter.anmelden(a2);
        newsletter.aendereZustand("Information 1");

        newsletter.abmelden(a1);
        newsletter.aendereZustand("Information 2");
        newsletter.abmelden(a2);
        newsletter.aendereZustand("Information 3");
    }
}
```

Ausgabe

```
Eine neue Nachricht fuer Erwin: Information 1
Eine neue Nachricht fuer Mike: Information 1
Eine neue Nachricht fuer Mike: Information 2
```

Anwendungen

- Implementierung von Plattformen, Vermittlern u. a.,
- Programmierung grafischer Oberflächen mit Schaltflächen.

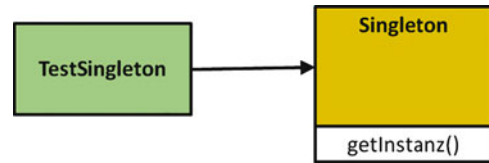
Bewertung

- **Vorteile:** Beobachtbares Objekt muss den Beobachter nicht zur Laufzeit kennen. Beobachter und Beobachtbarer können unabhängig voneinander wiederverwendet werden. Neue Beobachter können jederzeit hinzugefügt werden. Keine Änderung am Beobachtbaren notwendig, um neue Beobachter zu unterstützen.
- **Nachteile:** Information aller Beobachter, auch wenn diese eine Änderung nicht brauchen.

2.7 Erzeugungsmuster

Die Erzeugungsmuster beschreiben, wie Objekte statt mit dem `new`-Operator durch sogenannte Erzeugerklassen erstellt werden. Diese Klassen steuern die Art und Weise der Erzeugung von Objekten. Die Vorteile sind eine gesteigerte Funktionalität und Flexibilität der Softwaresysteme.

Abb. 2.8 Erzeugungsmuster des Singleton



2.7.1 Singleton

Die Aufgabe des Singleton (Singleton Pattern) ist es sicherzustellen, dass nur eine einzige Instanz einer Klasse existiert. Für das Erzeugen der einzigen Instanz soll die genannte Klasse selbst verantwortlich sein.

Problem Einmaliges Erzeugen einer Instanz einer Klasse.

Ziel Klasse nur ein einziges Mal instanziiert.

Lösungsprinzip Ein Singletonobjekt kann nur von der Klasse selbst erzeugt werden. Alle Konstruktoren dieser Klasse werden dadurch mit `private` gekennzeichnet. Die Klasse enthält eine Klassenmethode `getInstance`, die eine Referenz auf das einzig existierende Objekt der Klasse `Singleton` zurückliefert.

Implementierung Wir definieren eine Klasse `Singleton`, von der nur ein einziges Objekt erzeugt werden darf (siehe Abb. 2.8). Der Anwender ruft damit die Klassenmethode `getInstance` der Klasse `Singleton` auf und bekommt als Rückgabewert eine Referenz auf die erstellte Instanz dieser Klasse. Mit dieser Instanz können dann alle vorhandenen Methoden der Klasse `Singleton` aufgerufen werden.

Beispiel Wir demonstrieren das Prinzip an dem folgenden Beispiel.

Wir verwenden hierzu die folgenden Komponenten:

- Klasse `Singleton`: Definition der Klasse, von der nur ein einziges Objekt existieren darf.

```
class Singleton
{
    private static Singleton instance = new Singleton();

    private Singleton()
    {
        System.out.println("Singletonobjekt erzeugt.");
    }
}
```

```
public static Singleton getInstance()
{
    System.out.println("Methode getInstance() aufgerufen");
    return instance;
}
public void methode()
{
    System.out.println("Funktionsmethode aufgerufen.");
}
}
```

Die Methode `getInstance` liefert als Rückgabewert eine Referenz auf die erstellte Instanz dieser Klasse.

- Klasse `TestSingleton`: Zugriff mithilfe der Methode `getInstance`.

```
public class TestSingleton
{
    public static void main (String[] args)
    {
        Singleton sing = Singleton.getInstance();
        sing.methode();
        sing.methode();
    }
}
```

Ausgabe

Singletonobjekt erzeugt.

Methode `getInstance()` aufgerufen.

Funktionsmethode `methode()` aufgerufen.

Funktionsmethode `methode()` aufgerufen.

In der Praxis wird das Singleton häufig auch durch globale Variablen implementiert. Eine Erweiterung des Singleton ist das Multiton zur Erzeugung einer endlichen Zahl von Objekten.

Anwendungen

- Zugriff auf eine zentrale (Hardware-)Ressource,
- Serialisierung von Prozessen bei Vergabe von Nummern an verschiedene Anwender,
- Verhinderung unnötiger Objekterzeugung bei Instanziierungen mit großen Aufwand,
- Verwendung als Submuster in anderen Entwurfsmustern.

Bewertung

- **Vorteile:** einmalige Erzeugung eines Objekts.
- **Nachteile:** Vermischung von Klassendefinition und Objekterzeugung, starke Einschränkung der Funktionsweise.

2.7.2 Fabrik

Das Grundprinzip einer Fabrik ist, die Objekterstellung und den Objektgebrauch zu trennen. Bei einer Fabrik (Factory Method) wird eine Klassenschnittstelle mit Operationen zum Erzeugen eines Objekts definiert. Die Aufgabe der Fabrikmethoden ist die Erzeugung von Objekten an Unterklassen zu übertragen.

Problem Konkrete Klasse kennt nur die Basisklasse des zu erzeugenden Objekts, jedoch nicht die Unterklasse des entsprechenden Objekts.

Ziel Auslagerung der Objekterstellung in eine eigene Klasse.

Lösungsprinzip Die Fabrikmethode basiert auf einem abstrakten Grundgerüst von zwei abstrakten Klassen für das Produkt und den Erzeuger. Die abstrakte Klasse des Erzeugers enthält die zur Erzeugung von Produkten verwendete abstrakte Fabrikmethode, da der jeweilige konkrete Produkttyp noch nicht bekannt ist. Die Erzeugung von Objekten wird durch die konkreten Klassen für das Produkt und den Erzeuger vorgenommen.

Implementierung Die abstrakte Klasse `AbstraktesProdukt` definiert die Schnittstelle der Objekte, die durch die Fabrikmethode erzeugt werden (siehe Abb. 2.9). Alle

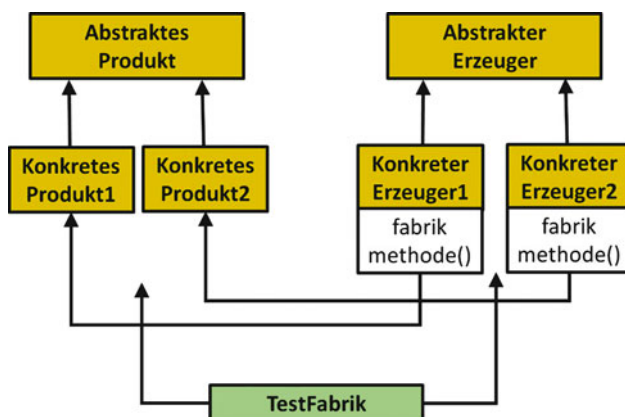


Abb. 2.9 Erzeugungsmuster der Fabrikmethode

konkreten Produkte müssen von der Klasse `AbstraktesProdukt` in Form einer Klasse `KonkretesProdukt1`, `KonkretesProdukt2` usw. abgeleitet sein.

Die abstrakte Klasse `AbstrakterErzeuger` definiert eine abstrakte Fabrikmethode `fabrikmethode`, die eine Referenz auf ein Objekt vom Typ `AbstraktesProdukt` zurückgibt. Diese Methode wird damit von einem Objekt der Klasse `AbstrakterErzeuger` selbst aufgerufen. Die konkreten Klassen `KonkreterErzeuger1`, `KonkreterErzeuger2` von der Klasse `AbstrakterErzeuger` überschreiben die Methode `fabrikmethode` der Klasse `Erzeuger`, um ein neues Objekt zu erzeugen und eine Referenz zurückzugeben.

Der Anwender `TestFabrik` ruft die Methode `fabrikmethode` des gewünschten Erzeugers auf. Das Ergebnis ist ein Objekt der konkreten Klasse `KonkreterErzeuger1`, `KonkreterErzeuger2` die eine Instanz vom Typ `KonkretesProdukt1`, `KonkretesProdukt2` usw. erzeugt und zurückgibt.

Beispiel In diesem Programmbeispiel entsprechen die Klassennamen den genannten Teilnehmern.

Wir verwenden hierzu die folgenden Komponenten:

- Klasse `AbstraktesProdukt`: Abstrakte Klasse definiert eine abstrakte Methode `ausgabe` zur Ausgabe der Produktinformation.

```
abstract class AbstraktesProdukt
{
    abstract public void ausgeben();
}
```

- Klasse `AbstrakterErzeuger`: Abstrakte Klasse definiert mit der Fabrikmethode die Schnittstelle zur Erzeugung von konkreten Produkten.

```
abstract class AbstrakterErzeuger
{
    public abstract AbstraktesProdukt fabrikmethode();
}
```

- Klasse `KonkreterErzeuger1` und `KonkreterErzeuger2`: konkrete Klassen der abstrakten Klasse `AbstrakterErzeuger` zur Erzeugung von konkreten Produkten.

```
class KonkreterErzeuger1 extends AbstrakterErzeuger
{
    public AbstraktesProdukt fabrikmethode()
    {
        return new KonkretesProdukt1();
    }
}
```

```
class KonkreterErzeuger2 extends AbstrakterErzeuger
{
    public AbstraktesProdukt fabrikmethode()
    {
        return new KonkretesProdukt2();
    }
}
```

- Klasse KonkretesProdukt1 und KonkretesProdukt2: konkrete Klassen der abstrakten Klasse AbstraktesProdukt zur Definition von Produkten.

```
class KonkretesProdukt1 extends AbstraktesProdukt
{
    public void ausgeben()
    {
        System.out.println("Produkt 1");
    }
}

class KonkretesProdukt2 extends AbstraktesProdukt
{
    public void ausgeben()
    {
        System.out.println("Produkt 2");
    }
}
```

- Klasse TestFabrik: Erzeugung von zwei konkreten Erzeugern, mit deren Hilfe konkrete Produkte durch die Fabrikmethode erstellt werden.

```
public class TestFabrik
{
    public static void main (String args[])
    {
        AbstrakterErzeuger e1 = new KonkreterErzeuger1();
        AbstraktesProdukt prod1 = e1.fabrikmethode();
        prod1.ausgeben();

        AbstrakterErzeuger e2 = new KonkreterErzeuger2();
        AbstraktesProdukt prod2 = e2.fabrikmethode();
        prod2.ausgeben();
    }
}
```

Ausgabe

Produkt 1

Produkt 2

Anwendungen

- Erzeugung von Objekten einer Klasse, deren konkreter Typ von vornherein nicht bekannt ist,
- Einsatz im Entwurfsmuster Schablone in Form von Einschubmethoden.

Bewertung

- **Vorteile:** keine genauen Kenntnisse des Objekttyps notwendig, gute Erweiterbarkeit der Klassenstruktur, Verhinderung des Einbindens von falschen anwendungsspezifischen Klassen, Entwicklung einer Klassenbibliothek ohne genauere Kenntnisse der konkreten Produktklassen.
- **Nachteile:** Erhöhung des Implementierungsaufwandes durch zusätzliche Unterklassen.

In diesem Buch werden wir einige dieser Entwurfsmuster verwenden, um Algorithmen übersichtlich und strukturiert zu implementieren.

2.8 Übungsaufgaben

Aufgabe 2.1 (Objektorientierte Programmierung) Erstellen Sie eine grafische Benutzeroberfläche zur Eingabe von Messwerten. Die Messwerte sollen anschließend in eine csv-Datei geschrieben werden. Definieren Sie die notwendigen Klassen für die GUI, die Messwerte und die Dateiverarbeitung. Erstellen Sie die GUI in der Form, dass Sie die eingelesenen Daten in einem Diagramm ausgeben können.

Aufgabe 2.2 (Objektorientierte Programmierung) Führen Sie eine objektorientierte Analyse zur Planung einer OP-Software durch. Das Softwaresystem besitzt dabei eine Eingabemaske, in der die Patientendaten, die Befunddaten und die durchzuführende OP eingegeben werden. Die OP-Daten bestehen aus dem behandelnden Arzt, der Anzahl der OP-Schwestern, dem zu operierenden Organ und der Art der OP. Erweitern Sie anschließend diese Planungssoftware um Sensordaten aus OP-Leuchte, OP-Tisch usw., um den optimalen Zeitpunkt der Anwesenheit der behandelnden Ärzte zu berechnen.

Aufgabe 2.3 (Objektorientierte Programmierung) Führen Sie eine objektorientierte Analyse zur Erstellung eines Onlineshops durch. Entwerfen Sie anschließend die notwendigen Klassen und erstellen Sie eine einfache Benutzeroberfläche.

Aufgabe 2.4 (Abstrakte Klassen) Erstellen Sie eine abstrakte Klasse `Konto` mit den Attributen `Kontonummer` und `Kontostand` sowie den folgenden Methoden:

- `getKontostand` gibt den aktuellen Kontostand zurück,
- `getKontonummer` gibt die Kontonummer zurück,
- `einzahlen` erhöht den Kontostand um einen gewissen Betrag,
- `auszahlen` verringert den Kontostand um einen gewissen Betrag.

Erstellen Sie anschließend die konkrete Klasse `Privatkonto` mit einem vorgegebenen Auszahlungslimit, einem Überziehungszinssatz und einer jährlichen Guthabenverzinsung sowie ein `Geschäftskonto` mit einem festgelegten Kreditrahmen.

Aufgabe 2.5 (Adapter) Implementieren Sie in einer Klasse einen Sortieralgorithmus aus dem 1. Band zum Sortieren von Zahlen in der Form:

```
ArrayList<Double> sortiere(ArrayList<Double>)
```

Schreiben Sie die folgenden Adapter der Form:

- a) `ArrayList<Integer> sortiere(ArrayList<Integer>)`
- b) `int[] sort(int [])`
- c) `int[][] sort(int [][])`

Der letzte Adapter soll dabei die Elemente zeilenweise in dem Array sortieren. Wenden Sie das Adaptermuster auf weitere ähnliche Problemstellungen an.

Aufgabe 2.6 (Fassade) Implementieren Sie eine digitale Plattform eines Bestellsystems, sodass ein Kunde der Plattform gewisse Produkte bestellen kann. Jedes Produkt besteht aus einer Menge von Komponenten, die von mehreren Zulieferern oder Servicepartnern produziert und geliefert werden. Entwickeln Sie eine Fassadenklasse um die komplette Bestellprozedur in diesem Prozess zu verbergen. Jede Komponente soll dazu beispielsweise eine Methode `fertige` und `versende` enthalten. Simulieren Sie den gesamten Bestellprozess eines Anwenders durch geeignete Ausgabetexte. Erstellen Sie anschließend eine einfache Benutzeroberfläche.

Aufgabe 2.7 (Dekorierer) Entwerfen Sie eine Klassenstruktur zur Repräsentation der Gehaltsabrechnung von Mitarbeitern in Form von zu dekorierenden Objekten. Konkrete Objekte sind beispielsweise ein `Angestellter` und ein `Manager`. Diese Objekte sollen anschließend um verschiedene dekorierende Objekte ergänzt werden, beispielsweise um einen monatlichen Bonus oder eine einmalige Prämienzahlung. Simulieren Sie den gesamten Prozess eines Anwenders durch geeignete Ausgabetexte.

Aufgabe 2.8 (Schablone) Implementieren Sie für die aus dem 1. Band vorgestellten Algorithmenmuster eines Greedy-Algorithmus bzw. für das Muster des dynamischen Programmierens ein objektorientiertes Entwurfsmuster in Form einer Schablone. Testen Sie die Implementierung an konkreten algorithmischen Verfahren aus dem Bereich der dynamischen Programmierung.

Aufgabe 2.9 (Strategie) Implementieren Sie die im 1. Band vorgestellten Sortieralgorithmen der BubbleSort- und QuickSort-Verfahren in Form eines objektorientierten Strategiemusters. Testen Sie die Implementierung an konkreten Beispielen. Erweitern Sie das Programmpaket um das beschriebene Adaptermuster aus Aufgabe 2.5 für unterschiedliche Formate.

Aufgabe 2.10 (Beobachter) Implementieren Sie für die vorgestellte Beispielanwendung eines Newsletters in Form eines objektorientierten Beobachters eine grafische Benutzeroberfläche. Erweitern Sie anschließend das Programm um weitere Funktionalitäten.

Aufgabe 2.11 (Fabrik) Entwickeln Sie anhand einer Dokumentenverwaltung eine abstrakte Klasse `Dokument` mit konkreten zu erzeugenden Beispielen wie `Rechnung`, `Lieferschein` usw. Die Erzeugerklasse hat anschließend die Aufgabe, konkrete Dokumente zu erzeugen und zu verarbeiten. Entwerfen Sie hierfür geeignete und sinnvolle Methoden.

Literaturhinweise¹

1. Krüger, G. (2014). *Handbuch der Java-Programmierung*. O'Reilly.
2. Ullenboom, C. (2013). *Java ist auch eine Insel*. Galileo Computing.
3. Louis, D., Müller, P. (2013). *Java 7 – Das Handbuch*. Pearson.
4. Goll, J. (2014). *Architektur- und Entwurfsmuster der Softwaretechnik*. Springer Vieweg.
5. Geirhos, M. (2015). *Entwurfsmuster – Das umfassende Handbuch*. Rheinwerk Computing.
6. Siebler, F. (2014). *Design Patterns mit Java*. Hanser.
7. Schiedermeier, R., Köhler, K. (2011). *Das Java-Praktikum*. dpunkt Verlag.
8. Sedgewick, R., Wayne, K. (2011). *Einführung in die Programmierung mit Java*. Pearson.
9. Habelitz, H-P. (2012). *Programmieren lernen mit Java*. Galileo Computing.

¹ Die Bücher [1], [2] und [3] geben einen umfangreichen Gesamtübersicht über die objektorientierte Programmierung in Java. Einen umfassenden Überblick über Entwurfsmuster für die objektorientierte Programmierung in Java findet der Leser in [4], [5] und [6]. Die zentralen Begriffe der Objektorientierung mit einer Gesamtübersicht aller relevanten Entwurfsmuster sind im Anhang dieses Buches zusammengefasst. Gute Übungsbücher mit zahlreichen weiteren praktischen Beispielen und Aufgaben zur objektorientierten Programmierung sind beispielsweise [7], [8] und [9].

Im Band Grundlagen haben wir bereits das Prinzip der strukturellen und objektorientierten Programmierung in Java vorgestellt. In diesem Kapitel stellen wir nun einige zentrale weiterführende Programmierkonzepte wie Fehlerbehandlung, Multithreading, Pakete, Datenbankzugriffe in SQL, XML und Excel sowie Datenstrukturen wie verkettete Listen, Stapel, Warteschlangen, Bäume und Graphen vor.

3.1 Programmierkonzepte

In diesem Abschnitt besprechen wir die folgenden zentralen Programmierkonzepte, die für zahlreiche praktische Aufgabenstellungen von großer Bedeutung sind:

- Exception: strukturierte Behandlung von Fehlern, die während der Programmausführung auftreten;
- Multithreading: mehrere Vorgänge in einem Programm gleichzeitig ausführen;
- Pakete: Gliederung von Klassen in Pakete.

3.1.1 Exception

Unter einer Exception-Behandlung versteht man in Java ein Verfahren zur strukturierten Behandlung von Fehlern, die während der Programmausführung auftreten. Beispielsweise kann diese Art von Laufzeitfehler in folgenden Fällen auftreten: Eine Datei wird nicht gefunden, eine Division durch null wird durchgeführt, ein Arrayzugriff ist außerhalb der definierten Grenzen erfolgt usw.

Mithilfe von Exceptions gibt es in Java eine Methodik, je nach Anforderung mit Laufzeitfehlern flexibel umzugehen. Bei dieser Fehlerbehandlung kann der Fehler nicht sofort,

sondern in Form von Exceptions weiter nach oben im Programm gereicht werden, bis er irgendwo abgefangen und behandelt wird. Damit ermöglichen Exceptions die Trennung von Fehlerauslösern und Fehlerbehandlung. Durch Verzicht auf Rückgabewerte bei der Weiterreichung der Exception wird der Quellcode wesentlich übersichtlicher.

Exception-Behandlung Das allgemeine Schema einer Exception-Behandlung sieht wie folgt aus:

1. **Auftreten eines Fehlers:** Die Exception-Behandlung beginnt mit dem Auftreten eines Fehlers, z. B. Division durch null, Datei nicht vorhanden usw.
2. **Antwort auf den Fehler:** Die Methode, in der der Fehler auftrat, löst eine Exception aus. Die Exception kann entweder an der ausgelösten Stelle behandelt werden oder in einem Objekt mit Fehlerinformationen weitergereicht werden. Falls die Ausnahme weitergegeben wird, kann sie an der Empfängerstelle entweder behandelt oder wieder weitergegeben werden.
3. **Abfangen des Fehlers:** Der Bereich, in dem eine Exception auftreten kann, wird mit dem Schlüsselwort `try` eingeleitet und in geschweifte Klammern gefasst. Die Exception wird in dem zugehörigen `catch`-Block abgearbeitet. Falls die Ausnahme von keinem Programmteil behandelt wird, so wird das Programm mit einer Fehlermeldung abgebrochen.

Exceptions abfangen Das Abfangen von Exceptions besteht aus zwei Blöcken, die mit den Schlüsselwörtern `try` und `catch` eingeleitet werden. Mit `try` wird ein Block definiert, der das Auftreten von Exceptions überwacht. An den `try`-Block angehängt folgen die `catch`-Blöcke, wobei jeder `catch`-Block eine Fehlerbehandlung für einen bestimmten Typ von Exception definiert:

```
try
{
    // Anweisungsüberwachung
}
catch(ExceptionTyp e)
{
    // Fehlerbehandlung
}
```

Beim Auftreten einer Exception wird die Abarbeitung des Programms im `try`-Block sofort unterbrochen. Wenn die `catch`-Anweisung zur Art des aufgetretenen Fehlers passt, wird der Fehler an dieser Stelle aufgefangen und behandelt.

Beispiel 3.1 Wir verwenden die Methode `Integer.parseInt()`, die eine Zeichenkette in eine Zahl umwandelt, zur Demonstration der Exception-Behandlung.


```
public static void main(String[] args)
{
    String s = "10+";
    try
    {
        int zahl = Integer.parseInt(s);
        System.out.printf("Zahl = %d", zahl);
    }
    catch (NumberFormatException e)
    {
        System.out.println("Exception abgefangen.");
    }
}
```

Ausgabe

Exception abgefangen.

Allgemeine Erklärung

Bei einer fehlerhaften Eingabe in die Methode `Integer.parseInt()` wird eine Exception vom Typ `NumberFormatException` ausgelöst.

Auf einen `try`-Block können auch mehrere `catch`-Blöcke folgen, die unterschiedliche Arten von Fehlertypen auf unterschiedliche Arten behandeln.

```
try
{
    // Anweisungsüberwachung
}
catch(ExceptionTyp1 e)
{
    // Fehlerbehandlung 1
}
catch(ExceptionTyp2 e)
{
    // Fehlerbehandlung 2
}
catch(ExceptionTyp3 e)
{
    // Fehlerbehandlung 3
}
```

Beim Auftreten eines Fehlers wird zuerst die erste `catch`-Klausel angelaufen. Falls die erste `catch`-Anweisung nicht zur Art des aufgetretenen Fehlers passt, werden der Reihe nach alle anderen `catch`-Klauseln untersucht. Dabei werden die erste übereinstimmende Klausel ausgewählt und die zugehörige Fehlerbehandlung durchgeführt.

Eine Exception ist ein Objekt, dessen Typ von `Throwable` abgeleitet ist. Jede Ausnahme wird von Exception abgeleitet, welche sich in weitere Unterklassen aufteilt. Diese Hierarchien bedeuten, dass es ausreicht, einen Fehler der Oberklasse aufzufangen.

Beispiel 3.2 Wir betrachten wieder das obige Programm, nun aber mit zwei `catch`-Blöcken:

```
public static void main(String[] args)
{
    String s = "10+";
    try
    {
        int zahl = Integer.parseInt(s);
        System.out.printf("Zahl = %d", zahl);
    }
    catch (NumberFormatException e)
    {
        System.out.println("Exception vom Typ NumberFormat abgefangen.");
    }
    catch (Exception e)
    {
        System.out.println("Allgemeine Exception abgefangen.");
    }
}
```

Ausgabe

Exception vom Typ NumberFormat abgefangen.

Allgemeine Erklärung

Aufgrund der Tatsache, dass immer die erste passende `catch`-Klausel ausgewählt wird, darf nicht die zweite `catch`-Klausel zuerst stehen, da diese auf jeden Fehler passt. Der Compiler gibt in diesem Fall einen Syntaxfehler aus.

Ein leerer `catch`-Block sollte in der Regel nicht verwendet werden, da hierbei der Fehler nur unterdrückt wird.

Um mehrere Ausnahmen auf einmal aufzufangen, werden die Ausnahmen durch einen Schrägstrich in Form von dem logischen Oder getrennt:

```
try
{
    ...
}
catch (E1 | E2 | ... | En exception)
```

Ausgabe von Exceptions Jede Exception liefert die Art des Fehlers, der durch den Typ der Exception angezeigt wird, und Informationen, die im Exception-Objekt abgespeichert sind. Für die vordefinierten Exception-Typen aus der Java-Standardbibliothek gibt es dazu die folgenden Methoden:

- `String getMessage()`: Rückgabe eines Fehlertextes;
- `void printStackTrace()`: Auszug aus dem Laufzeitstack.

Weitergabe einer Exception Für die Exception-Behandlung gibt es genau zwei Möglichkeiten: Ausnahme entweder behandeln (`catch`) oder sie weitergeben (`throw`). Die Weitergabe einer Exception erfolgt mithilfe des Schlüsselwortes `throws` am Ende des Methodenkopfs mit einer Liste aller Ausnahmen, die nicht behandelt werden:

```
public rueckgabetyt methodenname(...) throws ExceptionTyp
```

Wenn eine Ausnahme eintritt, erfolgt die Suche zunächst in der umgebenden `try-catch`-Anweisung bzw. den umgebenden Blöcken. Falls nichts gefunden wurde, wird der Fehler an den Aufrufer der Methode weitergegeben, und die gleiche Suchreihenfolge wird wiederum durchgeführt. Falls auch in der Hauptmethode kein Code zur Fehlerbehandlung gefunden wird, bricht das Programm mit einer Fehlermeldung ab.

Beispiel 3.3 Wir betrachten das folgende Beispiel, das eine Exception auslöst, wenn eine Division durch null erfolgt:

```
public class ExceptionBehandlung
{
    public static double division(double zaehler, double nenner) throws Exception
    {
        if(nenner == 0.0)
            throw new Exception("Division durch null!");
        else
            return zaehler/nenner;
    }

    public static void main(String[] args)
    {
        double zaehler = 2;
        double nenner = 0;
        try
        {
            double bruch = division(zaehler, nenner);
            System.out.println(zaehler + " : " + nenner + " = " + bruch);
        }
        catch (Exception e)
        {
            System.out.println("Exception abgefangen: " + e.getMessage());
        }
    }
}
```

Ausgabe

Exception abgefangen: Division durch null!

Allgemeine Erklärung

1. Die Methode `division()` erkennt den unzulässigen Nenner, erzeugt daraufhin eine Exception und löst diese mit `throw` aus.
2. Dem `Exception`-Konstruktor wird ein Fehlermeldungstext ("Division durch null!") übergeben, der in der Exception abgespeichert und später mit der `Exception`-Methode `getMessage()` abgefragt werden kann.
3. Da die Methode `division()` selbst keinen passenden `catch`-Handler enthält, wird die Exception an die aufrufende Methode `main()` weitergeleitet.
4. In `main()` liegt der Aufruf von `division()` in einem `try`-Block mit einem `catch`-Handler, der Exceptions des Typs `Exception` abfängt. Da die aktuelle Exception ebenfalls von diesem Typ ist, wird sie abgefangen, und der `catch`-Handler wird ausgeführt.

Die Methode könnte auch so implementiert werden, dass sie eine Fehlermeldung ausgibt und `nenner` einfach auf 1 setzt. Diese Form der Fehlerbehandlung ist aber dann fester Teil der Methode, wodurch sie für eine andere Form der Fehlerbehandlung unbrauchbar würde.

Finally-Block Beim sofortigen Beenden der Methoden in der Fehlerbehandlung kann es dazu kommen, dass wichtige Abschlussarbeiten (z. B. Schließen einer Datei), die am Ende der Methodendefinition stehen, nicht mehr ausgeführt werden. Daher gibt es die Möglichkeit, an einen `try`- oder `try-catch`-Block einen `finally`-Block anzuhängen, der auf jeden Fall ausgeführt wird, unabhängig davon, ob im `try`-Block eine Exception auftrat oder nicht.

```
try
{
    // Code, der überwacht wird
}
catch (Exception e)
{
    // Code zur Behandlung des Fehlers
}
finally
{
    // Code, der immer ausgeführt wird
}
```

Der `finally`-Block wird in den folgenden Fällen aufgerufen:

- Das Ende des `try`-Blocks ist erreicht.
- Eine Ausnahme ist aufgetreten, die durch eine `catch`-Klausel behandelt wurde.

- Eine Ausnahme ist aufgetreten, die nicht durch eine `catch`-Klausel behandelt wurde.
- Der `try`-Block wird durch eine Sprunganweisung verlassen.

Beispiel 3.4 Wir betrachten wieder das obige Programm mit einem `finally`-Block:

```
public static void main(String[] args)
{
    String s = "10+";
    try
    {
        int zahl = Integer.parseInt(s);
        System.out.printf("Zahl = %d", zahl);
    }
    catch (NumberFormatException e)
    {
        System.out.println("Fehlertyp: " + e.getMessage());
    }
    finally
    {
        System.out.println("Abschlussarbeiten");
    }
}
```

Ausgabe

Fehlertyp: For input string: "10+"
Abschlussarbeiten

3.1.2 Multithreading

Mit *Multithreading* (dt. Nebenläufigkeit) bezeichnet man die Fähigkeit eines Systems, zwei oder mehr Vorgänge gleichzeitig ausführen zu können. Ein Thread ist ein eigenständiges Programmfragment, das parallel zu anderen Threads innerhalb eines Prozesses abläuft. Alle Threads eines Programms greifen dann auf dieselben Variablen zu. Anwendung von Threads findet man beispielsweise bei der Erstellung von grafischen Anwendungen, bei denen die Bedienbarkeit verbessert wird, indem rechenintensive Anwendungen im Hintergrund ablaufen.

Erzeugen eines neuen Threads Threads werden in Java durch die Klasse `Thread` und das Interface `Runnable` implementiert. In beiden Fällen wird der parallel auszuführende Code durch die Überlagerung der Methode `run` zur Verfügung gestellt. Die Klasse `Thread` stellt die Basismethoden zur Erzeugung, Kontrolle und zum Beenden von Threads zur Verfügung. Für die Erzeugung eines Threads muss eine eigene Klasse aus `Thread` abgeleitet und die Methode `run` überlagert werden:

```
public void run()
```

Mithilfe eines Aufrufes der Methode `start` wird der Thread gestartet und die weitere Ausführung an die Methode `run` übertragen. Die Methode `run` sollte vom Programm niemals direkt aufgerufen werden, da sonst kein neuer Thread erzeugt wird.

Beispiel 3.5 Das folgende Beispiel zeigt einen einfachen Thread, der in einer Endlosschleife einen Zahlenwert hochzählt:

```
class Thread1 extends Thread
{
    // Überlagerung der Methode run
    public void run()
    {
        int i=0;
        while(true)
            System.out.println(i++);
    }
}

public class TestThread
{
    public static void main(String[] args)
    {
        Thread1 t = new Thread1();
        t.start();
    }
}
```

Ausgabe

```
1
2
...
```

Allgemeine Erklärung

In diesem Programm wird ein neues Objekt vom Typ `Thread1` instanziiert. Durch Aufruf von `start` wird ein neuer Thread erzeugt und durch den Aufruf von `run` gestartet. Das Programm läuft so lange, bis es von außerhalb abgebrochen wird.

Durch eine kleine Änderung in Form einer Ausgabe des Namens des Threads können wir nun das Zusammenspiel von zwei Threads beobachten:

```
class Thread1 extends Thread
{
    private String name;
    public Thread1(String name)
    {
        this.name = name;
    }
}
```

```
    public void run()
    {
        int i=0;
        while(true)
            System.out.println(name + " Nummer: " + (i++));
    }
}

public class TestThread
{
    public static void main(String[] args)
    {
        Thread1 t1 = new Thread1("Thread 1");
        Thread1 t2 = new Thread1("Thread 2");
        t1.start();
        t2.start();
    }
}
```

Ausgabe

```
...
Thread 1 Nummer: 97830
Thread 1 Nummer: 97831
Thread 2 Nummer: 92982
Thread 2 Nummer: 92983
...
Thread 2 Nummer: 93004
Thread 1 Nummer: 97832
Thread 2 Nummer: 93005
Thread 2 Nummer: 93006
...
```

Allgemeine Erklärung

Jeder der beiden Threads erzeugt ein eigenes Objekt der Klasse `Thread1`. Damit zählt jedes Objekt die Ausgabezeilen hoch, und wir erhalten als Ergebnis eine Ausgabe der beiden Threads in beliebiger Reihenfolge. Dadurch entstehen die Sprünge bei den Nummern für die einzelnen Threads.

Threads pausieren In vielen Fällen ist es sehr sinnvoll, Threads eine gewisse Zeit pausieren zu lassen. Dafür wird die folgende statische Methode `sleep` verwendet:

```
public static void sleep(long millis)
```

Mit der Methode `sleep` wird der aktuelle Prozess für die in Millisekunden angegebene Zeit unterbrochen. Wichtig hierbei ist, dass der Aufruf von `Thread.sleep` innerhalb

eines try-catch-Blocks erfolgen muss, da die Methode während der Wartezeit eine Ausnahme vom Typ `InterruptedException` auslösen kann. Ohne den try-catch-Block würde diese an den Aufrufer weitergegeben.

Abbrechen eines Threads Der Thread wird beendet, wenn das Ende seiner run-Methode erreicht ist. In manchen Fällen ist es jedoch erforderlich, den Thread von außerhalb abzubrechen, was durch die folgenden Methoden erledigt werden kann:

```
public void interrupt()
public boolean isInterrupted()
public static boolean interrupted()
```

Durch Aufruf von `interrupt` wird ein Flag gesetzt, das eine Unterbrechungsanforderung signalisiert. Durch Aufruf von `isInterrupted` kann der Thread feststellen, ob das Abbruch-Flag gesetzt wurde und der Thread beendet werden soll. Die statische Methode `interrupted` stellt den Status des Abbruch-Flags beim aktuellen Thread fest.

Beispiel 3.6 Wir starten das obige Beispiel nun mit einer `sleep`-Methode, um den Thread abzubrechen.

```
class Thread1 extends Thread
{
    public void run()
    {
        int i=0;
        while(true)
        {
            if (isInterrupted())
                break;
            System.out.println(i++);
        }
    }
}

public class TestThread
{
    public static void main(String[] args)
    {
        Thread1 t = new Thread1();
        t.start();
        try
        {
            Thread.sleep(1000);
        }
        catch (InterruptedException e)
        {
            System.out.println("InterruptedException ausgelöst.");
        }
        t.interrupt();
    }
}
```


Ausgabe

```
1
2
...
170713
```

Allgemeine Erklärung

Die `main`-Methode erzeugt wieder den `Thread` und startet diesen mit der `run`-Methode. In der Methode `run` wird bei jedem Aufruf mit `isInterrupted` geprüft, ob der Abbruch gesetzt wurde. Ist das der Fall, wird keine weitere Zahl ausgegeben, sondern die Schleife wird beendet. In diesem Fall läuft der `Thread` genau eine Sekunde, da mit dem Aufruf von `Thread.sleep(1000)` eine Sekunde pausiert und anschließend das Flag zum Unterbrechen gesetzt wird.

Feststellung des Zustands Mit der Methode `isAlive()` kann festgestellt werden, ob ein `Thread` noch läuft.

```
public final boolean isAlive()
```

Die Methode `isAlive` gibt immer dann `true` zurück, wenn der `Thread` gestartet, aber noch nicht wieder beendet wurde. Beendet wird ein `Thread`, wenn das Ende der `run`-Methode erreicht ist oder wenn die Methode `stop` aufgerufen wurde.

Interface Runnable Mit dem Interface `Runnable` besteht eine alternative Möglichkeit einen `Thread` zu implementieren, insbesondere dann, wenn die Klasse bereits Bestandteil einer Vererbungshierarchie ist. Hierzu sind die folgenden Schritte notwendig:

1. Erzeugung eines neuen Objekts der eigenen Klasse:

```
Thread1 t = new Thread1();
```

2. Anlegen eines neuen Threads durch Übergabe des Objekts an den Konstruktor der Klasse `Thread`:

```
Thread th = new Thread(t);
```

3. Aufruf der Methode `start` des neuen `Thread`-Objekts:

```
th.start();
```

Kürzer kann man die beiden ersten Befehle auch wie folgt schreiben:

```
Thread th = new Thread(new Thread1());
```

Beispiel 3.7 Wir übertragen das obige Programm durch Implementierung der Schnittstelle `Runnable`:

```
class Thread1 implements Runnable
{
    public void run()
    {
        int i=0;
        while(true)
        {
            if (Thread.interrupted())
                break;
            System.out.println(i++);
        }
    }
}

public class TestThread
{
    public static void main(String[] args)
    {
        Thread1 t = new Thread1();
        Thread th = new Thread(t);
        th.start();
        try
        {
            Thread.sleep(1000);
        }
        catch (InterruptedException e)
        {
            System.out.println("InterruptedException ausgelöst.");
        }
        th.interrupt();
    }
}
```

Ausgabe

```
1
2
3
...
```

Allgemeine Erklärung

Bei der Implementierung der Schnittstelle `Runnable` wird anstelle der Methode `isInterrupted` die statische Methode `interrupted` aufgerufen. Durch Übergabe des Objekts von Typ `Thread1` an den Konstruktor der Klasse `Thread` wird dieser mit der `run`-Methode gestartet.

Synchronisation In Java erfolgt die Kommunikation zweier Threads auf der Basis gemeinsamer Variablen, die von beiden Threads erreicht werden können. Führen beide Pro-

Änderungen auf den gemeinsamen Daten durch, so müssen sie synchronisiert werden, denn andernfalls können undefinierte Ergebnisse entstehen.

Beispiel 3.8 Wir betrachten das folgende einfache Programm mit zwei unsynchronisierten Threads:

```
class Thread2 extends Thread
{
    public static int i = 0;
    public void run()
    {
        while(i<100)
            System.out.println(i++);
    }

    public static void main(String[] args)
    {
        Thread t1 = new Thread2();
        Thread t2 = new Thread2();
        t1.start();
        t2.start();
    }
}
```

Ausgabe

```
...
6
7
9
10
8
11
13
...
```

Allgemeine Erklärung

Als Ausgabe dieses Programms erhält man nicht die erwartete Reihenfolge der natürlichen Zahlen. Der Grund ist der folgende: Die beiden Threads greifen unsynchronisiert auf die gemeinsame Klassenvariable *i* zu. Es kann nun vorkommen, dass die Operation zur Ausgabe der Zahl während der Ausführung unterbrochen wird und der andere Thread Rechenzeit bekommt. Erst dann, wenn der unterbrochene Prozess wieder an der Reihe ist, gibt er den vor der Unterbrechung bestimmten Zählerwert aus. In diesem Fall war der zweite Prozess bereits bei einer höheren Zahl angelangt.

Das Beispiel zeigt, dass solche Inkonsistenzen zu fehlerhaften Programmen führen. Daher müssen die beteiligten Prozesse synchronisiert werden. Die Synchronisation nebenläufiger Prozesse erfolgt durch Kapselung des kritischen Bereichs mithilfe einer automatisch verwalteten Sperre, die mit dem Schlüsselwort `synchronized` gekennzeichnet

ist. Diese Sperre wird beim Betreten dieses Bereichs gesetzt und beim Verlassen wieder zurückgenommen. Damit muss ein anderer Prozess warten, bis die Sperre wieder aufgelöst wurde.

```
while (true)
{
    synchronized (getClass())
    {
        System.out.println(i++);
    }
}
```

Beispiel 3.9 Wir betrachten das obige Programm mit zwei synchronisierten Threads:

```
class Thread2 extends Thread
{
    public static int i = 0;
    public void run()
    {
        while(i<100)
        {
            synchronized(getClass())
            {
                System.out.println(i++);
            }
        }
    }

    public static void main(String[] args)
    {
        Thread t1 = new Thread2();
        Thread t2 = new Thread2();
        t1.start();
        t2.start();
    }
}
```

Ausgabe

```
...
6
7
8
9
10
11
12
...
```

Weiterführende Aspekte Neben den angesprochenen Möglichkeiten zum Arbeiten mit Threads gibt es einige weiterführende Themen, die hier nur kurz benannt werden sollen:

- **Verwalten von Threads:** Die Klasse `Thread` besitzt zahlreiche Methoden zum Verwalten von Threads durch Zuweisung von Namen und Prioritäten.
- **Warteliste:** Weitere Synchronisationsmöglichkeiten bestehen durch die Verwendung von `wait` und `notify` über Wartelisten von Objekten.
- **Spezielle Datenstrukturen:** In dem Java-Paket `java.util.concurrent` befinden sich unter anderem verschiedene Klassen mit threadsicheren Datenstrukturen.
- **Verbinden von Threads:** Falls zwei Threads in der Form verbunden werden sollen, dass einer von beiden Daten erzeugt und der andere verarbeitet, ist dies mit den beiden Klassen `PipedInputStream` und `PipedOutputStream` möglich.
- **Teile-und-herrsche:** Das bekannte Algorithmenmuster Teile-und-herrsche kann sehr effizient mit Klassen aus dem Java Paket `java.util.concurrent.RecursiveTask` umgesetzt werden.

3.1.3 Pakete

Pakete sind Strukturelemente, die aus einer Menge von Klassen bestehen, die aus gemeinsamen Verwendungszwecken zusammengefasst sind. In der Programmiersprache Java gehört jede Klasse zu einem Paket wie beispielsweise die Klasse `File` zum Paket `java.io` oder die Klasse `Arrays` zum Paket `java.util.Arrays`.

Für die Verwendung einer Klasse muss das Paket mithilfe einer `import`-Anweisung eingebunden werden:

```
import paket.Klasse;  
import paket.*;
```

Mit dem ersten Befehl wird genau die angegebene Klasse und mit dem zweiten Befehl werden alle Klassen des angegebenen Pakets eingebunden.

Das Konzept der Pakete wurde geschaffen, um große Programmsysteme sauber zu strukturieren. Die Erstellung eigener Pakete ist in Java sehr einfach. Für die Zuordnung einer Klasse in ein Paket, wird als erste Zeile im Programm eine Anweisung aus dem Schlüsselwort `package` und dem Namen des Pakets eingefügt:

```
package paketname;
```

Falls wie bisher keine `package`-Anweisung angegeben wird, wird standardmäßig das `default`-Paket verwendet. Die Klassen dieses Pakets können dann ohne `import`-Anweisung verwendet werden.

Beispiel 3.10 Wir erstellen zwei Pakete, `paket1` mit den beiden Klassen `Klasse1` und `Klasse2` sowie Paket `paket2` mit der Klasse `Klasse3`:

- Paket `paket1` mit `Klasse1`:

```
package paket1;

public class Klasse1
{
    public Klasse1()
    {
        System.out.println("Klasse 1");
    }
}
```

- Paket `paket1` mit `Klasse2`:

```
package paket1;

public class Klasse2
{
    public Klasse2()
    {
        System.out.println("Klasse 2");
    }
}
```

- Paket `paket2` mit `Klasse3`:

```
package paket2;

public class Klasse3
{
    public Klasse3()
    {
        System.out.println("Klasse 3");
    }
}
```

- Testklasse zur Verwendung der definierten Pakete:

```
import paket1.*;
import paket2.*;

public abstract class TestPakete
{
    public static void main(String[] args)
    {
        new Klasse1();
        new Klasse2();
        new Klasse3();
    }
}
```

Ausgabe

Klasse 1

Klasse 2

Klasse 3

Als Ergebnis werden damit zwei Unterverzeichnisse `paket1` und `paket2` mit den jeweiligen zugehörigen Klassen angelegt.

Zu beachten ist, dass eine Klasse eine andere Klasse nur einbinden kann, wenn entweder die beiden Klassen dem gleichen Paket angehören, oder die einzubindende Klasse wird als `public` deklariert.

3.2 Datenbankzugriffe

In diesem Abschnitt stellen wir verschiedene Varianten zum Einlesen von Daten aus unterschiedlichen Quellen vor. Konkret betrachten wir SQL-Datenbanken, XML-Dateien und die für viele praktische Anwendungen hilfreichen Excel-Dateien.

3.2.1 SQL

Datenbankzugriffe sind eine wesentliche Voraussetzung für die Bearbeitung vieler Problemstellungen, deren Grundlagen große Datenmengen in relationalen Datenbanken sind. Relationale Datenbanken sind Tabellen mit Spalten und Zeilen. Eine Zeile bzw. Tupel entspricht einem Element einer Tabelle, eine Spalte bzw. Attribut einem Eintrag einer Tabelle.

SQL ist die bedeutendste Abfragesprache für relationale Datenbanken, bei denen der Benutzer angibt, auf welche Inhalte er zugreifen will. In Java erfolgt der Zugriff auf relationale Datenbanken mit dem JDBC (Java Database Connectivity), einem standardisierten Datenbankinterface. JDBC ist keine eigene Datenbank, sondern eine Schnittstelle zwischen einer speziellen SQL-Datenbank und der Applikation. SQL-Anweisungen werden dann im Programm als Zeichenketten bearbeitet und weiterverarbeitet.

Wir stellen in diesem Abschnitt den grundlegenden Aufbau von JDBC und die zugehörigen datenbankbasierten Java-Anweisungen vor.

Öffnen einer Verbindung Der erste Schritt, um mit JDBC auf eine Datenbank zuzugreifen, ist das Laden eines Datenbanktreibers mit dem Befehl `Class.forName` und dem zugehörigen Klassennamen mithilfe des Pakets `java.sql`:

```
Class.forName("org.hsqldb.jdbcDriver");
```

In diesem Fall handelt es sich um den Treiber `org.hsqldb.jdbcDriver` für die frei verfügbare Java-Datenbank HSQLDB. Wichtig ist, dass der Treiber `hsqldb.jar` von

der Internetseite <http://hsqldb.sourceforge.net/> heruntergeladen und in den Java Build Path hinzugefügt wird.

Im zweiten Schritt wird versucht, eine Verbindung zu einer Datenbank aufzubauen. Dazu gibt es die statische Methode `getConnection` der Klasse `DriverManager` mit drei verschiedenen Varianten:

```
static Connection getConnection(String url)
static Connection getConnection(String url, String user, String pw)
static Connection getConnection(String url, Properties info)
```

In jedem Fall muss ein sogenannter Connection-String in Form einer URL, der aus mehreren durch Doppelpunkte voneinander getrennten Teilen besteht, übergeben werden. Bei der zweiten Variante werden zusätzlich noch der Benutzername und das Passwort an die Datenbank übergeben und bei der dritten Variante können durch ein `Properties`-Objekt weitere treiberspezifische Informationen übergeben werden. Die jeweilige Variante, die verwendet werden muss, ist aus den Treiberdokumentationen zu entnehmen.

Wir benötigen für die obige Datenbank die zweite Darstellung in der folgenden Form:

```
Connection con = DriverManager.getConnection
("jdbc:hsqldb:file:hsqldb-2.3.3; shutdown=true", "sa", "");
```

Der Connection-String besteht immer aus `jdbc`, ein Subprotokoll mit Treibertyp (hier: `hsqldb`) und weiteren treiberspezifischen Details `file:hsqldb-2.3.3; shutdown=true`. Der Name `hsqldb-2.3.3` bezeichnet die zugehörige Datenbank, die entweder neu angelegt wird oder bereits vorhanden ist. Falls die Datenbank nicht geöffnet werden konnte, wird eine Ausnahme des Typs `SQLException` ausgelöst.

Bei erfolgreicher Verbindungsaufnahme liefert `getConnection` ein Objekt, das das Interface `Connection` implementiert. Dieses Verbindungsobjekt dient dazu, Anweisungsobjekte zu erzeugen und globale Einstellungen an der Datenbank zu verändern. Das Connection-Objekt sollte am Ende des Programms durch Aufruf von `close` geschlossen werden.

Erzeugen von Anweisungsobjekten Die Abfragen und Änderungen in einer Datenbank werden mithilfe von Anweisungsobjekten vorgenommen. Diese sogenannten Statement-Objekte werden mit Methoden des Connection-Objekts erzeugt:

```
Statement createStatement();
```

Für das obige Connection-Objekt sieht die Anweisung wie folgt aus:

```
Statement sm = (Statement) con.createStatement();
```

Mit dem Statement-Objekt und dessen beiden Methoden `executeQuery` und `executeUpdate` können nun Daten aus der Datenbank gelesen und geschrieben werden.

Datenbankabfragen Mithilfe des Statement-Objekts kann nun die Methode `executeQuery` verwendet werden, um Daten aus der Datenbank zu lesen:

```
public ResultSet executeQuery(String sql) throws SQLException
```

Dabei muss dieser Methode ein SQL-String als Argument übergeben werden, beispielsweise die Auswahl aller Elemente der Tabelle `TabName` durch den SQL-String `SELECT * FROM TabName`:

```
ResultSet rs = sm.executeQuery("SELECT * FROM TabName");
```

Zurückgegeben werden entweder ein einfacher numerischer Ergebniswert, der den Erfolg der Anweisung anzeigt, oder eine Menge von Datenbanksätzen vom Typ `ResultSet`.

SQL-Befehle Die Datenbanksprache SQL gliedert sich in eine Reihe von unterschiedlichen Abfragetypen:

- **DDL (Data Definition Language):** Erstellen der Tabellen, Beziehungen (mit Schlüsseln) und Indizes.
Typische SQL-Anweisungen: `CREATE/DROP DATABASE`, `CREATE/DROP INDEX`, `CREATE/DROP SYNONYM`, `CREATE/DROP TABLE`, `CREATE/DROP VIEW`.
- **DML (Data Manipulation Language):** Daten hinzufügen und löschen.
Typische SQL-Anweisungen: `INSERT`, `DELETE` und `UPDATE`.
- **DQL (Data Query Language):** Daten auswählen und filtern.
Typischste SQL-Anweisung: `SELECT` mit den Spezialisierungen `ALL`, `DISTINCT`, `ORDER BY`, `GROUP BY`, `HAVING`, Unterabfragen (`IN`, `ANY`, `ALL`, `EXISTS`), Schnittmengen und Joins.

Die wichtigste Anweisung ist hierbei der bereits erwähnte `SELECT`-Befehl:

```
SELECT {Feldname, Feldname,...|*}
FROM Tabelle [, Tabelle, Tabelle....]
[WHERE {Bedingung}]
[ORDER BY Feldname [ASC|DESC]...]
[GROUP BY Feldname [HAVING {Bedingung}]]
```

Eine Abfrage mit `textbox*` liefert alle Spalten zurück. SQL kennt dabei die üblichen Vergleichsoperatoren (`textbox=` [gleich], `textbox<>` [ungleich], `textbox>` [größer], `textbox<` [kleiner], `textbox>=` [größer gleich], `textbox<=` [kleiner gleich]), logische Operatoren (`AND`, `OR`, `NOT`) und Rechenoperatoren (`textbox+`, `-`, `*`, `/`). Anstelle vieler `AND`-Abfragen lässt sich mit zwei SQL-Anweisungen auch der Wertebereich mit `BETWEEN wert1 AND wert2` testen, d.h., ob sich ein Vergleichswert zwischen `wert1`

und `wert2` befindet. Mit dem Befehl `IN` kann man prüfen, ob der Vergleichswert in der angegebenen Werteliste liegt.

Ebenso können die Daten mit einem `ORDER BY` aufsteigend (`ASC`) bzw. absteigend (`DESC`) sortiert werden, indem die Spalte angegeben wird. Mit speziellen Gruppenfunktionen lassen sich auch Durchschnittswerte, Minima oder Maxima über Spalten bestimmen:

```
SELECT COUNT(*) FROM TabName WHERE FeldName = wert
```

Die SQL-Standardfunktionen sind `AVG` (Durchschnittswert), `COUNT` (Anzahl aller Einträge), `MAX` (Maximalwert), `MIN` (Minimalwert) und `SUM` (Summe aller Einträge).

SQL-Datenbankabfragen Ein SQL-String ist eine gültige Anweisung in der Sprache SQL. Für den Zugriff auf eine Datenbank benötigen wir eine gültige `SELECT`-Anweisung.

SQL-Abfragen sind sehr nahe an der Umgangssprache formuliert. Der Satz „Wähle alle Elemente der Spalten `FeldName1` und `FeldName2` mit dem Wert 10 der Spalte `FeldName3` aus der Tabelle `TabName` aus.“ übersetzt sich in SQL wie folgt:

```
SELECT FeldName1, FeldName2 FROM TabName WHERE Feldname3 = 10
```

Das zurückgegebene Objekt ist vom Typ `ResultSet` und besitzt eine Methode `next`, mit der die Resultatmenge schrittweise durchlaufbar ist:

```
boolean next()
```

Durch den Aufruf von `executeQuery` steht der Satzzeiger vor dem ersten Element. Mit dem Aufruf von `next` wird er auf das nächste Element verschoben. Falls er `false` ist, gibt es keine weiteren Elemente in der Ergebnismenge. Ist er dagegen `true`, konnte das nächste Element erfolgreich ausgewählt werden, und beispielsweise mithilfe verschiedener `get`-Methoden (z. B. `getString`, `getDouble`, `getInt`) auf den Ergebnisstring zugegriffen werden:

```
ResultSet rs = sm.executeQuery("SELECT * FROM Tabelle");  
while(rs.next())  
    System.out.printf("%s, %s %n", rs.getString(1), rs.getString(2));
```

Bei einem numerischen Wert als Argument wird dieser als Spaltenindex interpretiert (erste Spalte hat den Index 1). Wird ein String übergeben, so wird er als Name interpretiert und der Wert der Spalte mit diesem Namen zurückgegeben. Der Vorteil ist, dass damit der Aufruf nicht mehr von der Spaltenreihenfolge der Abfrage abhängt.

SQL-Datenbankänderungen Abfragen aus einer Datenbank werden mit der Methode `executeUpdate` auf dem Statement-Objekt durchgeführt:

```
public int executeUpdate(String sql) throws SQLException
```

Der Rückgabewert gibt an, wie viele Datensätze von der Änderung betroffen sind. Die Änderung einer Datenbank erfolgt mit den SQL-Anweisungen `INSERT INTO`, `UPDATE` oder `DELETE FROM`. Das zugehörige Argument ist ein String in Form einer gültigen SQL-Anweisung, beispielsweise:

```
INSERT INTO Tabelle VALUES (...)
```

Könnte diese Anweisung erfolgreich ausgeführt werden, gibt sie 1 zurück, andernfalls würde eine SQL-Exception ausgelöst.

Mit der folgenden Anweisung wird in die Tabelle der Datensatz 50, 'Max', 'Mustermann', 'Musterstrasse 12', 'Musterstadt' geschrieben:

```
sm.executeUpdate("INSERT INTO Tabelle VALUES (50, 'Max', 'Mustermann',  
'Musterstrasse 12', 'Musterstadt')");
```

Beispieldatenbank Wir erzeugen nun eine einfache Beispieldatenbank mit Tabellennamen `Customer` von `HSQldb`. Dazu sind zunächst die folgenden Schritte notwendig:

1. Herunterladen und Auspacken der aktuellen Datei `hsqldb.zip` von der Internetseite <http://sourceforge.net/projects/hsqldb/files/>.
2. Ausführen des Skripts `runManagerSwing.bat` im Ordner `bin`.
3. Setzen der folgenden Parameter im Datenbank-Frontend:
 - Typ: `HSQL Database Engine Standalone`;
 - JDBC-URL: `jdbc:hsqldb:file:pfad`, wobei `pfad` der aktuelle Pfad des Datenbankordners ist.
4. Beenden des Dialogs mit `OK`.
5. Hinzufügen von Tabellen mit Dummydaten durch Aufruf der Operation `Insert Test Data` im Menü `Options`.
6. Nach Beendigung mit `File > Exit` werden eine `log`-Datei, eine `script`-Datei, eine `properties`-Datei und eine `lck`-Datei erzeugt.

Für den Datenbankzugriff aus Java ist nur das Archiv `hsqldb.jar` aus dem `lib`-Verzeichnis von `HSQldb` in den Klassenpfad aufzunehmen. Damit erhalten wir den folgenden Programmcode, um auf die Einträge der Tabelle zuzugreifen:

```

import java.sql.*;

public class SQL_DB1
{
    public static void main( String[] args ) throws Exception
    {
        // --- 1. Laden des Treibers der HSQLDB-Datenbank
        Class.forName("org.hsqldb.jdbcDriver");

        // --- 2. Öffnen einer Verbindung zur Datenbank mit Namen = hsqldb-2.3.3
        Connection con = DriverManager.getConnection(
            "jdbc:hsqldb:file:hsqldb-2.3.3; shutdown=true", "sa", "" );

        // --- 3. Erzeugung eines Anweisungsobjekts
        Statement sm = (Statement) con.createStatement();

        // --- 4. Datenbankabfragen
        ResultSet rs = sm.executeQuery("SELECT * FROM Customer");
        while(rs.next())
            System.out.printf( "%s, %s %s\n", rs.getString(1),
                               rs.getString(2), rs.getString(3));

        // --- 5. Schließen der Datenbank
        rs.close();
        sm.close();
        con.close();
    }
}

```

Ausgabe

```

0, Laura Steel
1, Robert King
2, Robert Sommer
...
48, Bob Schneider
49, Robert Steel

```

Erstellen von Tabellen Für die Erstellung von Tabellen werden die folgenden SQL-Befehle benötigt:

- Erstellung einer neuen Tabelle:

```

CREATE TABLE TabName
(ColName DataType [DEFAULT ConstExpr]
[ColName DataType [DEFAULT ConstExpr]]...)

```

Hierbei sind TabName der Name der Tabelle, ColName der Name der Spalte, IndexName und ConstExpr ein optionaler konstanter Ausdruck, der einen Standardwert für eine Spalte vorgibt.

- Löschen einer Tabelle:

```
DROP TABLE TabName
```

- Änderung der Struktur der Tabelle:

```
ALTER TABLE TabName
ADD (ColName DataType
[ColName DataType]...)
```

- Anlegung bzw. Löschen eines Index:

```
CREATE [UNIQUE] INDEX IndexName
ON TabName
(ColName [ASC|DESC]
[, ColName [ASC|DESC]]...)
```

bzw.

```
DROP INDEX IndexName
```

DataType gibt den Datentyp der Spalte an:

Bezeichnung	Erklärung
CHAR (n)	Zeichenkette der (festen) Länge n
VARCHAR (n)	Zeichenkette variabler Länge mit maximal n Zeichen
SMALLINT	16-Bit-Ganzzahl mit Vorzeichen
INTEGER	32-Bit-Ganzzahl mit Vorzeichen
REAL	Fließkommazahl mit etwa 7 signifikanten Stellen
FLOAT	Fließkommazahl mit etwa 15 signifikanten Stellen
DECIMAL (n,m)	Festkommazahl mit n Stellen, davon m Nachkommastellen
DATE	Datum (eventuell mit Uhrzeit)

Beispiel 3.11 Für das Erstellen einer neuen Tabelle erhalten wir den folgenden Code:

```
public class SQL_DB2
{
    public static void main( String[] args ) throws Exception
    {
        // --- 1. Laden des Treibers der HSQLDB-Datenbank
        Class.forName("org.hsqldb.jdbcDriver");

        // --- 2. Öffnen einer Verbindung zur Datenbank mit Namen = hsqldb-2.3.3
        Connection con = DriverManager.getConnection(
            "jdbc:hsqldb:file:hsqldb-2.3.3; shutdown=true", "sa", "" );
```

```

// --- 3. Erzeugung eines Anweisungsobjekts
Statement sm = (Statement) con.createStatement();

// --- 4. Erzeugung der neuen Tabelle mit Namen Personen
sm.executeUpdate("DROP TABLE Personen"); // löschen falls schon existent
sm.executeUpdate("CREATE TABLE Personen (" + "nummer INTEGER," + "name VARCHAR(100))");

// --- 5. Daten in Tabelle einfügen
String namen[] = {"Anton", "Sophia", "Egon", "Elsbeth"};
for(int i=0; i<namen.length; i++)
{
    String s = "INSERT INTO Personen VALUES(" + i + ", '" + namen[i] + "')";
    sm.executeUpdate(s);
}

// --- 6. Datenbankabfragen
ResultSet rs = sm.executeQuery("SELECT * FROM Personen");
while(rs.next())
    System.out.printf( "%s, %s %n", rs.getString(1), rs.getString(2));

// --- 7. Schließen der Datenbank
rs.close(); sm.close(); con.close();
}
}

```

Ausgabe

```

0, Anton
1, Sophia
2, Egon
3, Elsbeth

```

3.2.2 XML

XML ist eine erweiterbare Auszeichnungssprache (Extensible Markup Language) zur Darstellung hierarchisch strukturierter baumartiger Datenstrukturen in Form von Textdateien. XML-Dateien werden vor allem für den plattformunabhängigen Austausch von Daten zwischen verschiedenen Computersystemen verwendet.

Aufbau XML-Dokument Die erste Zeile in einem XML-Dokument ist der optionale Prolog, der mit der Zeichenfolge `<?xml` beginnt, mit `>` endet und dazwischen die XML-Version (`version="1.0"`) bzw. gegebenenfalls weitere optionale Argumente enthält:

```

<?xml version="1.0"?>
<?xml version="1.0" encoding="iso-8859-15"?>

```

Falls die Encoding-Angabe fehlt, wird als Zeichensatz die Unicode-Kodierung¹ UTF-8 bzw. UTF-16 genommen.

In der zweiten Zeile eines XML-Dokuments steht das sogenannte Wurzelement:

¹ Standard, bei dem für jedes internationale Schriftzeichen ein digitaler Code definiert ist.

```
<wurzel>
...
</wurzel>
```

Das Wurzelement wird von einem öffnenden `<-` und einem schließenden `>-`Tag um einen frei wählbaren Namen `wurzel` umschlossen. Das Wurzelement kann nun selber wieder weitere Elemente oder Textinformationen enthalten:

```
<elementname>
...
</elementname>
```

Die Syntax eines XML-Dokuments kann damit wie folgt definiert werden:

```
xml      : [ prolog ] element
element : <tag> [ ( element | text ) ] </tag>
```

Mit dieser Beschreibungsform können hierarchisch aufgebaute baumartige Datenstrukturen sehr bequem definiert werden. Die einzelnen Elemente können dabei auch eine beliebige Anzahl von Attributen besitzen, welche gewisse Eigenschaften des Elements beschreiben:

```
<elementname attribut = wert>
...
</elementname>
```

Kommentare werden in XML-Dateien zwischen den Syntaxanweisungen `<!--` und `-->` geschrieben.

Beispiel 3.12 Die folgende XML-Datei besteht aus dem Wurzelement `wurzel` mit den zwei Elementen `klasse1` und `klasse2` und den beiden Textzeilen:

```
<?xml version="1.0"?>
<!-- Eine XML-Datei -->
<wurzel>
  <klasse1>
    Text 1
  </klasse1>
  <klasse2>
    Text 2
  </klasse2>
</wurzel>
```

Verarbeitung von XML-Dokumenten In Java gibt es für die Verarbeitung von XML-Dokumenten verschiedene Möglichkeiten, beispielsweise mittels DOM (Document Object Model) durch die Umformung des Dokuments in eine baumartige Objektstruktur, oder mittels SAX (Simple API for XML) bzw. StAX (Streaming API for XML) über einen sogenannten ereignisorientierten Ansatz.

Wir betrachten hier die einfachste Variante in Form des DOM-Ansatzes, bei dem das XML-Dokument als Objektbaum in den Speicher geladen wird. Das Einladen einer XML-Datei `Daten.xml` erfolgt dabei mit den folgenden drei Zeilen:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document document = builder.parse(new File("Daten.xml"));
```

Das Package `javax.xml.parsers` stellt einen sogenannten XML-Parser zum Erzeugen eines `DocumentBuilderFactory` und eines DOM-Baumes mit der Klasse `DocumentBuilder` zur Verfügung. Die dritte Zeile liefert ein Objekt vom Typ `Document` aus dem Paket `org.w3c.dom.`, das den DOM-Baum für das eingelesene XML-Dokument repräsentiert.

Nach dem Einlesen der XML-Datei kann der entstehende DOM-Baum mit den folgenden Methoden weiter verarbeitet werden:

- `NodeList liste = document.getElementsByTagName("elementname");`
Bestimmung einer Liste mit Elementen vom Typ `Element` mit dem angegebenen Namen. Der Zugriff auf das *i*-te Element erfolgt mit dem folgenden Befehl:
`Element element = (Element) liste.item(i)`
- `public String getTextContent()`
Zugriff auf den Textinhalt eines Elements. Für die Entfernung von Leerzeichen und Zeilenumbrüchen kann die Methode `trim` verwendet werden.
- `public String getAttribute(String name)`
Zugriff auf den Wert des Attributs mit dem Bezeichner `name` eines Elements.
- `public String getNodeName()`
Zugriff auf den Bezeichner eines Elements.

Beispiel 3.13 Die folgende XML-Datei definiert zwei Produktkategorien `Mechanik` und `Elektronik` mit verschiedenen Arten von Produkten in Form `Schraube`, `Mutter`, `LED`, `Diode` und `Transistor`. Jedes Produkt wird dabei durch die drei Eigenschaften `Produktname name`, `Artikelnummer nr` und `Preis preis` beschrieben.


```

<?xml version="1.0"?>
<!-- XML-Datei -->
<prod>
  <Produktkategorie>
    <name>Mechanik</name>
    <Produkt>
      <name>Schraube</name>
      <nr>1029</nr>
      <preis>1.99</preis>
    </Produkt>
    <Produkt>
      <name>Mutter</name>
      <nr>1324</nr>
      <preis>0.99</preis>
    </Produkt>
  </Produktkategorie>
  <Produktkategorie>
    <name>Elektronik</name>
    <Produkt>
      <name>LED</name>
      <nr>1232</nr>
      <preis>4.99</preis>
    </Produkt>
    <Produkt>
      <name>Diode</name>
      <nr>1234</nr>
      <preis>0.25</preis>
    </Produkt>
    <Produkt>
      <name>Transistor</name>
      <nr>3232</nr>
      <preis>1.10</preis>
    </Produkt>
  </Produktkategorie>
</prod>

```

Die Verarbeitung der XML-Datei mit der Ausgabe sämtlicher Produkte erzielen wir mit dem folgenden Java-Code:

```

import java.io.File;
import javax.xml.parsers.*;
import org.w3c.dom.*;

public class XML_Verarbeitung
{
  public static void main(String[] args) throws Exception
  {
    // --- 1. Laden des XML-Files mittels DOM
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    DocumentBuilder builder = factory.newDocumentBuilder();
    Document document = builder.parse(new File("Daten.xml"));
  }
}

```

```

// --- 2. Definition einer Liste aus Elementen
NodeList liste = document.getElementsByTagName("Produkt");

// --- 3. Auslesen aller Elemente der definierten Liste
for(int i=0; i<liste.getLength(); i++)
{
    Element element = (Element)liste.item(i);
    String name      = element.getElementsByTagName("name").item(0).getTextContent().trim();
    System.out.println(name);
}
}
}

```

Ausgabe

Schraube
Mutter
LED
Diode
Transistor

3.2.3 Excel

Die Verarbeitung von MS Excel-Daten ist ein sehr häufiger Anwendungsfall in der praktischen Arbeit. Es gibt eine ganze Reihe von unterschiedlichen Java-Paketen zum Lesen und Schreiben von Excel-Dateien. Wir stellen hier die Apache POI der „Apache Software Foundation“ vor.²

Schreiben von Excel-Dateien Für das Schreiben einer Excel-Tabelle sind die folgenden Schritte notwendig:

1. Definition eines leeren Workbook:

```
HSSFWorkbook w = new HSSFWorkbook();
```

2. Erzeugung einer leeren Excel-Tabelle `tab` mit dem Bezeichner `TabName`:

```
HSSFSheet tab = w.createSheet("TabName");
```

3. Erzeugung der `i`-ten Zeile `zeile` der Tabelle `tab`:

```
Row zeile = tab.createRow(i);
```

4. Befüllung der Zelle in der `i`-ten Zeile und `j`-ten Spalte mit dem Inhalt `wert`:

```
zeile.createCell(j).setCellValue(wert);
```

² JAR-Dateien und weitere Informationen: <http://poi.apache.org/download.html>.

5. Schreiben der Excel-Datei mit dem Dateinamen Name.xls:

```
w.write(new FileOutputStream("Name.xls"));
w.close();
```

Beispiel 3.14 Wir erstellen eine Excel-Datei mit dem Namen Datei.xls mit einer einzigen Tabelle mit dem Namen Tabelle 1:

```
import java.io.*;
import org.apache.poi.hssf.usermodel.*;
import org.apache.poi.ss.usermodel.Row;

public class Excel_Schreiben
{
    public static void main(String args[]) throws FileNotFoundException, IOException
    {
        // Definition von Beispieldaten
        int zahlen[][] = {{1,2,3}, {4,5,6}};

        // 1. Definition eines leeren Workbooks
        HSSFWorkbook workbook = new HSSFWorkbook();

        // 2. Erzeugung einer leeren Excel-Tabelle
        HSSFSheet tab = workbook.createSheet("Tabelle 1");

        // 3. Einfügen der Inhalte
        int zeileNr = 0;
        for(int i=0; i<zahlen.length; i++)
        {
            // 3a. Erzeugung der i-ten Zeile
            Row zeile = tab.createRow(zeileNr++);
            for(int j=0; j<zahlen[0].length; j++)
            {
                // 3b. Befüllung der Zelle
                zeile.createCell(j).setCellValue(zahlen[i][j]);
            }
        }

        // 4. Schreiben der Excel-Datei
        workbook.write(new FileOutputStream("Datei.xls"));
        workbook.close();
    }
}
```

Lesen von Excel-Dateien Für das Schreiben einer Excel-Tabelle sind die folgenden Schritte notwendig:

1. Definition des Workbook mit der gewünschten Excel-Datei mit dem Dateinamen Name.xls:

```
HSSFWorkbook w = new HSSFWorkbook(new FileInputStream("Name.xls"));
```

2. Erzeugung der Excel-Tabelle `tab` mit dem Bezeichner `TabName`:

```
HSSFSheet tab = w.createSheet("TabName");
```

3. Erzeugung der Zeile `zeile` mithilfe eines Iterators über die Zeilen der Tabelle `tab`:

```
Iterator<Row> zeileIt = tab.iterator();
Row zeile = zeileIt.next();
```

4. Zugriff auf die `j`-te Zelle:

```
zeile.getCell(j);
```

Beispiel 3.15 Wir lesen die obige Excel-Datei mit dem Namen `Datei.xls` mit einer Tabelle mit dem Namen `Tabelle 1` ein:

```
import java.io.*;
import java.util.Iterator;
import org.apache.poi.hssf.usermodel.*;
import org.apache.poi.ss.usermodel.Row;

public class Excel_Lesen
{
    public static void main(String args[]) throws FileNotFoundException, IOException
    {
        // 1. Definition eines leeren Workbooks
        HSSFWorkbook workbook = new HSSFWorkbook(new FileInputStream("Datei.xls"));

        // 2. Auswahl der Tabelle
        HSSFSheet tab = workbook.getSheet("Tabelle 1");

        // 3. Lesen der Inhalte
        Iterator<Row> zeileIt = tab.iterator();
        while(zeileIt.hasNext())
        {
            Row zeile = zeileIt.next();
            for(int j=0; j<=2; j++)
                System.out.println(zeile.getCell(j));
        }
    }
}
```

Ausgabe

```
1.0
2.0
3.0
4.0
5.0
6.0
```

Apache POI bietet zusätzlich noch Klassen zum Formatieren von Zellen und Tabellen an.³

3.3 Datenstrukturen

Bei der Programmierung von Algorithmen werden oftmals Felder mit unbekannter Elementzahl benötigt. Damit entstehen die folgenden Probleme bei der Verwendung von herkömmlichen statischen Standardfeldern:

- Feld ist zu klein: Auftreten eines Programmfehlers;
- Feld ist zu groß: Verschwendung von Speicherplatz;
- Feld muss vergrößert werden: zeitaufwendiges Umkopieren des Speicherinhaltes;
- Feld besitzt bestimmte Reihenfolge: Einfügen und Löschen benötigt viel Aufwand.

Diese Probleme der statischen Strukturen lassen sich durch die Verwendung dynamischer Strukturen umgehen, die während der Laufzeit „wachsen“ oder „schrumpfen“ können. Die Datenstrukturen (engl. collections) dienen dazu, eine Menge von Daten aufzunehmen und effizient zu verarbeiten. Mithilfe der objektorientierten Programmierung ist es sehr elegant möglich, Datenstrukturen in unabhängigen Klassen zu schreiben, und gezielt in unterschiedlichen Programmen wieder zu verwenden. In diesem Abschnitt stellen wir zunächst die wichtigsten elementaren Datenstrukturen in Java vor. Anschließend zeigen wir, wie man die Datenstrukturen einer Liste, Stapel, Warteschlange, Baum oder Graph in Java implementiert. In vielen Algorithmen verwenden wir spezielle Strukturen, um flexible unterschiedliche Arten von Daten zu verarbeiten.

3.3.1 Listen, Mengen und Hash-Tabellen

Die im 1. Band vorgestellten Datenstrukturen der Klassen `Vector`, `Stack`, `Hashtable` und `BitSet` sind im Laufe der Entwicklung der Sprache Java weiterentwickelt worden. Es gibt im Paket `java.util` über 20 Klassen und Interfaces, die aus den folgenden drei Grundformen `Set`, `List` und `Map` bestehen:

- `List`: Liste von Elementen beliebigen Typs, auf die sowohl wahlfrei, als auch sequenziell zugegriffen werden kann;
- `Set`: Menge von doublettenlosen Elementen, auf die mit typischen Mengenoperationen zugegriffen werden kann;
- `Map`: Abbildung von Elementen eines Typs auf Elemente eines anderen Typs, also eine Menge zusammengehöriger Paare von Objekten.

³ <http://poi.apache.org/spreadsheet/quick-guide.html>.

Der Name einer konkreten Datenstruktur besteht aus dem Teil mit der spezifischen Bezeichnung, gefolgt vom Namen der Grundform.

Typ List

Die Datenstruktur `List` ist eine geordnete Menge von Objekten, auf die entweder sequenziell oder wahlfrei, also über ihren Index zugegriffen werden kann. Das erste Element hat dabei den Index 0 und das letzte den Index `size() - 1`. Bei einer dynamischen Datenstruktur kann an einer beliebigen Stelle der Liste ein Element eingefügt oder gelöscht werden. Zusätzlich stehen Methoden zur Verfügung, um Elemente in der Liste zu suchen.

Das Interface `List` wird von verschiedenen Klassen implementiert:

- `LinkedList`: Liste mit Elementen in Form einer doppelt verketteten linearen Liste (jedes Element hat einen Zeiger auf das nachfolgende als auch auf das vorhergehende Element). Die Operationen zum Einfügen und Löschen sind schneller als die der `ArrayList`. Der wahlfreie Zugriff ist dagegen normalerweise langsamer.
- `ArrayList`: Liste mit Elementen, die bei Bedarf vergrößert werden kann. Der wahlfreie Zugriff ist hier schneller, aber bei großen Elementzahlen kann das Einfügen und Löschen länger dauern als bei einer `LinkedList`.

Einfügen von Elementen Das Anlegen einer neuen leeren Liste erfolgt mithilfe des parameterlosen Konstruktors:

```
public LinkedList()  
public ArrayList()
```

Die Angabe der Datentypen in der Struktur `List` wird durch die Klammern `<...>` gekennzeichnet. Durch Aufruf der Methode `isEmpty` kann geprüft werden, ob eine `List` leer ist. Die Methode `size` liefert die Anzahl der Elemente:

```
public final boolean isEmpty()  
public final int size()
```

Neue Elemente können an einer beliebigen Stelle mit den folgenden Methoden in die Liste eingefügt werden:

```
boolean add(Object obj)  
void add(int index, Object obj)  
boolean addAll(Collection daten)  
boolean addAll(int index, Collection daten)
```

Mit der Methode `add` wird ein Element `obj` an das Ende der Liste eingefügt. Wird zusätzlich noch der Index `index` angegeben, so werden das Element an die angegebene Position eingefügt und alle übrigen Elemente um eine Position nach rechts geschoben. Mit der Methode `addAll` kann eine komplette Datenstruktur `daten` wahlweise an das Ende angehängt oder an einer beliebigen Stelle in der Liste eingefügt werden.

Beispiel 3.16 Das folgende Programm legt eine `LinkedList` an und fügt Wochentage in der richtigen Reihenfolge mit den obigen Methoden ein:

```
import java.util.LinkedList;

public class DS_LinkedList
{
    public static void main(String[] args)
    {
        LinkedList<String> v1 = new LinkedList<String>();
        v1.add("Montag");
        v1.add("Mittwoch");
        v1.add(1, "Dienstag");

        LinkedList<String> v2 = new LinkedList<String>();
        v2.add("Donnerstag");
        v2.add("Freitag");

        v1.addAll(v2);
        for (int i=0; i<v1.size(); i++)
            System.out.printf(" %s ", v1.get(i));
    }
}
```

Ausgabe

Montag Dienstag Mittwoch Donnerstag Freitag

Zugriff auf Elemente Der wahlfreie Zugriff auf Elemente der Liste erfolgt mit einer der folgenden Methoden:

```
public Object getFirst()
public Object getLast()
public Object get(int index)
```

Die Methode `getFirst` liefert das erste Element, `getLast` das letzte, und mit `get` wird auf das Element an Position `index` zugegriffen. Alle drei Methoden verursachen eine Ausnahme, wenn das gesuchte Element nicht vorhanden ist.

Löschen von Elementen Das Löschen von Elementen in der Liste kann mit den folgenden Methoden erfolgen:

```
Object remove(int index)
boolean remove(Object obj)
boolean removeAll(Collection daten)
boolean retainAll(Collection daten)
```

Die Methode `remove` enthält entweder den Index `index` des zu löschenden Objekts oder das Objekt `obj` selbst. Mit `removeAll` werden alle Elemente gelöscht, die in der übergebenen Datenstruktur `daten` enthalten sind. Die Methode `retainAll` löscht hingegen alle Elemente außer denen in der Datenstruktur `daten`.

Iteratoren Iteratoren sind Objekte zum Durchlaufen von Datenstrukturen. Ein Iterator für eine Datenstruktur `list` wird wie folgt angelegt:

```
Iterator it = list.iterator();
```

Mithilfe des Iterators können die folgenden Methoden verwendet werden:

```
boolean hasNext()
Object next()
void remove()
```

Die Methode `hasNext` gibt genau dann `true` zurück, wenn der Iterator mindestens ein weiteres Element enthält. Mit der Methode `next` wird das nächste Element geliefert und mit `remove` gelöscht.

Typ Set

Ein `Set` ist eine Menge und erlaubt im Gegensatz zu `List` keine doppelten Elemente. Versucht man mit `add` ein Element einzufügen, das bereits vorhanden ist, dann wird dieses nicht eingefügt, sondern `add` gibt `false` zurück.

Für die Implementierung verwenden wir die Klasse `HashSet` mit den folgenden zwei Konstruktoren:

```
public HashSet()
public HashSet(int anz)
```

Beim zweiten Konstruktor wird die Anzahl der Elemente der `HashSet` vorgegeben.

Beispiel 3.17 Wir verwenden die Klasse `HashSet` zur Erzeugung eines Lottotipps 6 aus 49:

```
import java.util.HashSet;
import java.util.Iterator;

public class DS_HashSet
{
    public static void main(String[] args)
    {
        HashSet<Integer> set = new HashSet<Integer>(6);

        while (set.size() < 6)
        {
            int num = (int)(Math.random() * 49) + 1;
            set.add(new Integer(num));
        }

        Iterator<Integer> it = set.iterator();
        while (it.hasNext())
            System.out.println(((Integer)it.next()).toString());
    }
}
```

Ausgabe

34, 12, 9, 47, 2, 10

Allgemeine Erklärung

In diesem Programm wird eine `HashSet` so lange mit Zufallszahlen zwischen 1 und 49 bestückt, bis die Anzahl der Elemente 6 ist. Aufgrund der Tatsache, dass in diese Datenstruktur keine Elemente mehrfach eingefügt werden können, enthält ein Tipp jede Zahl nur einmal.

Typ Map

Eine Datenstruktur vom Typ `Map` stellt einen assoziativen Speicher dar, der Schlüssel auf Werte abbildet. Für jeden Schlüssel gibt es maximal einen Eintrag in der Datenstruktur, d. h., ein Schlüssel-Wert-Paar wird nicht eingefügt, wenn dessen Schlüssel bereits existiert. In diesem Fall wird dem vorhandenen Schlüssel ein neuer Wert zugeordnet. Die Schlüssel und auch die Werte können Objekte beliebigen Typs sein.

Wir verwenden die Klasse `HashMap` mit dem Konstruktor zum Anlegen einer neuen `Map`:

```
public HashMap()
```

Mit der Methode `isEmpty` kann geprüft werden, ob die `Map` leer ist. Die Anzahl der Elemente liefert die Methode `size` zurück:

```
int size()
boolean isEmpty()
```

Das Einfügen von Schlüssel-Wert-Paaren erfolgt durch die beiden Methoden:

```
Object put(Object schluessel, Object wert)
void putAll(Map m)
```

Mit der Methode `put` wird ein neues Schlüssel-Wert-Paar eingefügt bzw. dem bereits vorhandenen Schlüssel ein neuer Wert zugeordnet. Die Methode `putAll` erledigt dies für alle Paare die als Argument übergeben werden.

Die folgenden zwei Methoden prüfen, ob die Schlüssel oder Werte in der Map vorhanden sind:

```
boolean containsKey(Object schluessel)
boolean containsValue(Object wert)
```

Mit der Methode `get` kann der Wert zu einem Schlüssel bestimmt werden:

```
Object get(Object key)
```

Die Methode `keySet` liefert die Menge der Schlüssel und `values` liefert die Menge der Werte der Map:

```
public Set keySet()
public Collection value()
```

Die Methode `entrySet` liefert eine Menge von Schlüssel-Wert-Paaren:

```
public Set entrySet()
```

Die Methode `remove` löscht ein Objekt mit dem zugehörigen Schlüssel aus der Map:

```
Object remove(Object schluessel)
```

Beispiel 3.18 Die Datenstruktur `HashMap` eignet sich sehr gut um E-Mail-Adressen zu Personen zu speichern:

```
public class DS_Map
{
    public static void main(String[] args)
    {
        HashMap<String, String> h = new HashMap<String, String>();
        h.put("Sheldon", "s.cooper@bbt.com");
        h.put("Penny", "penny85@bbt.com");
        h.put("Howard", "howy@bbt.com");
    }
}
```

```

        Iterator<String> it = h.keySet().iterator();
        while (it.hasNext())
        {
            String key = (String)it.next();
            System.out.println(key + ": " + (String)h.get(key));
        }
    }
}

```

Ausgabe

Howard: howy@bbt.com

Penny: penny85@bbt.com

Sheldon: s.cooper@bbt.com

In den folgenden Abschnitten zeigen wir, wie zentrale Datenstrukturen in Java selbstständig implementiert werden können, um sie für algorithmische Verfahren anzuwenden bzw. anzupassen.

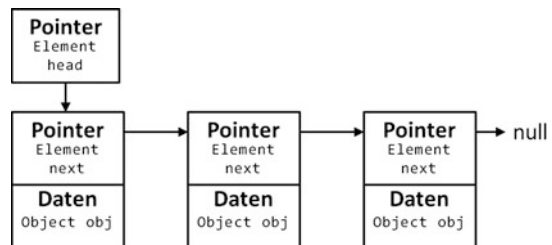
3.3.2 Verkettete Listen

Die wichtigste dynamische Datenstruktur ist die verkettete Liste, dargestellt in Abb. 3.1. Verkettete Listen enthalten nicht mehr Elemente als nötig, können beliebig wachsen und an angegebenen Stellen können Elemente eingefügt oder gelöscht werden. Die Elemente einer verketteten Liste werden im Speicher nicht aufeinanderfolgend, sondern an beliebigen freien Stellen abgelegt. Eine Liste besteht aus einer Menge von Knoten, die untereinander mit einem Art Pointer verbunden sind. Jeder Knoten besitzt damit einen Verweis auf das nachfolgende Element, wobei das letzte Element den Wert `null` hat.

Methoden Eine verkettete Liste besitzt neben einem Konstruktor zum Erzeugen einer neuen leeren Liste und dem ersten Element `head` die folgenden elementaren Operationen:

- `addFirst` hängt ein Element am Anfang der Liste an;
- `removeFirst` löscht das erste Element aus der Liste;
- `getFirst` gibt das erste Element der Liste zurück;

Abb. 3.1 Verkettete Liste mit mehreren Elementen



- `addLast` hängt ein Element an das Ende der Liste an;
- `removeLast` löscht das letzte Element aus der Liste;
- `getLast` gibt das letzte Element der Liste zurück;
- `size` gibt die Länge der Liste zurück;
- `isEmpty` testet, ob die Liste leer ist.

Implementierung der Hilfsklasse `Element` Für die Implementierung einer verketteten Liste in Java muss zunächst die Hilfsklasse `Element` mit den Attributen `Object obj` für das anzulegende Datenelement und dem Pointer `Element next` definiert werden. Zusätzlich werden noch verschiedene `get`- und `set`-Methoden für den Zugriff auf die Attribute implementiert:

```
class Element
{
    private Object obj;
    private Element next;

    public Element()
    {
        obj = null;
        next = null;
    }
    public Element(Object o, Element n)
    {
        obj = o;
        next = n;
    }
    public void setElement(Object o)
    {
        obj = o;
    }
    public Object getElement()
    {
        return obj;
    }
    public void setNext(Element n)
    {
        next = n;
    }
    public Element getNext()
    {
        return next;
    }
}
```

Einfügen eines ersten Elements Zum Einfügen eines neuen Elements verwendet man einen Zeiger `head`, der auf das erste Element der Liste zeigt. Damit kann mit der Methode `addFirst` ein neues Objekt als erstes Element in die Liste eingefügt werden, indem man

Abb. 3.2 Einfügen eines Elements in die verkettete Liste

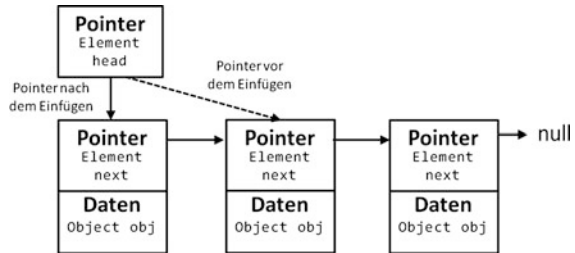
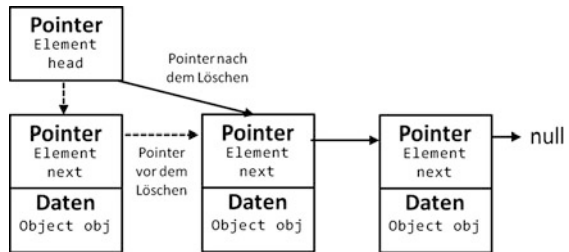


Abb. 3.3 Löschen eines Elements in der verketteten Liste



einen neuen Knoten erzeugt, der auf das erste Element zeigt. Anschließend muss nur der Zeiger head auf das neue Element umgesetzt werden (siehe Abb. 3.2):

```
public void addFirst(Object o)
{
    Element n = new Element(o, head.getNext());
    head.setNext(n);
}
```

Löschen des ersten Elements Zum Löschen des ersten Elements wird die Methode `removeFirst` implementiert, indem der Zeiger head auf den Knoten gesetzt wird, auf den der erste Knoten verweist (siehe Abb. 3.3):

```
public Object removeFirst()
{
    Object o = head.getNext().getElement();
    head.setNext(head.getNext().getNext());
    return o;
}
```

Ausgabe des ersten Elements Die Ausgabe des ersten Elements erfolgt mit der Methode `getFirst`, indem mit dem Zeiger head auf das erste Element zugegriffen wird:

```
public Object getFirst()
{
    return head.getNext().getElement();
}
```

Manipulation des Listenendes Die drei Operationen `addLast`, `getLast` und `removeLast` manipulieren das Listenende, indem ein neues Objekt `obj` als letztes Element an die Liste angehängt, ausgegeben oder gelöscht wird. Da nur der Listenanfang bekannt ist, muss der letzte Knoten durch Verfolgen des `next`-Zeigers gefunden werden. Bei diesem Knoten verweist der `next`-Zeiger auf `null`. Wenn der letzte Knoten erreicht ist, wird der Zeiger auf den neu angelegten Knoten gesetzt. Beim Löschen des letzten Elements wird der Zeiger des vorletzten Elements auf `null` gesetzt:

```
public void addLast(Object o)
{
    Element l = head;
    while (l.getNext() != null)
        l = l.getNext();
    Element n = new Element(o, null);
    l.setNext(n);
}
public Object removeLast()
{
    Element l = head;
    while (l.getNext().getNext() != null)
        l = l.getNext();
    Object o = l.getNext().getElement();
    l.setNext(null);
    return o;
}
public Object getLast()
{
    Element l = head;
    while (l.getNext() != null)
        l = l.getNext();
    return l.getElement();
}
```

Verwendung der Liste Die Grundstruktur der verketteten Liste wird dann in Java wie folgt implementiert:

```
public class Liste
{
    private Element head = null;
    public Liste()
    {
        head = new Element();
    }

    public void addFirst(Object o)
    public Object removeFirst()
    public Object getFirst()

    public void addLast(Object o)
    public Object removeLast()
    public Object getLast()
}
```

```
public int size()
public boolean isEmpty()

public static void main (String args[])
{
    Liste lst = new Liste();
    lst.addFirst("Zwei");
    lst.addFirst("Eins");

    lst.addLast("Drei");
    lst.addLast("Vier");

    while (!lst.isEmpty())
    {
        System.out.println((String) lst.removeFirst());
    }
}
```

Ausgabe

Eins
Zwei
Drei
Vier

Eine mögliche Erweiterung dieser Klasse ist die Implementierung von Methoden, um ein k -tes Element einzufügen, zu löschen oder auszugeben.

Komplexität der Listenoperationen Der Aufwand der sechs Operationen zur Manipulation einer Liste mit n Elementen ist unterschiedlich. Die ersten drei Operationen `addFirst`, `removeFirst` und `getFirst` besitzen nur den konstanten Aufwand $O(1)$. Die restlichen drei Operationen `addLast`, `getLast` und `removeLast` haben den Aufwand $O(n)$, da in diesem Fall durch die gesamte Liste gelaufen werden muss.

Der erhöhte Aufwand zur Manipulation des Listenendes kann durch eine Implementierung einer doppelt verketteten Liste umgangen werden. Bei dieser Art der Liste wird jeder Knoten nicht nur mit seinem Nachfolger, sondern auch mit seinem Vorgänger verwiesen.

3.3.3 Stapel und Warteschlangen

Das Prinzip der Verkettung lässt sich auch zur Implementierung von einem Stapel (Stack) oder einer Warteschlange (Queue) verwenden.

Stapel

Ein Stapel (Keller, Stack) arbeitet nach dem LIFO-Prinzip, d. h., das zuletzt eingefügte Element wird als erstes wieder entnommen. Bei einem Stapel kann damit nur auf das

oberste Element zugegriffen werden. Neue Elemente können somit nur oben auf den Stapel gelegt werden. Stapel haben in der Informatik eine Vielzahl von Anwendungen:

- Analysen von syntaktischen Anweisungen,
- Implementierung von Rekursion,
- Auswertung arithmetischer Ausdrücke,
- Sichern von lokalen Variablen bei Funktionsaufrufen.

Methoden Ein Stapel besitzt neben einem Konstruktor zum Erzeugen eines leeren Stapels die folgenden elementaren Operationen:

- `push` legt ein Element oben auf den Stapel;
- `top` gibt das oberste Element des Stapels zurück;
- `pop` entfernt das oberste Element des Stapels;
- `isEmpty` testet, ob Stapel leer ist.

Stapel werden mit einfach verketteten Listen realisiert. Dabei müssen die Elemente am gleichen Ende eingefügt und entfernt werden. In Java ist ein Stapel bereits durch die Klasse `Stack` mit den zugehörigen Methoden vordefiniert:

```
public Stack()
```

Beispiel 3.19 Füllung eines `Stack` mit den Wochentagen in umgekehrter Reihenfolge:

```
public static void main(String[] args)
{
    Stack<String> s = new Stack<String>();

    s.push("Montag");
    s.push("Dienstag");
    s.push("Mittwoch");

    while (!s.empty())
        System.out.println(s.pop());
}
```

Ausgabe

```
Mittwoch
Dienstag
Montag
```

Auswertung von arithmetischen Ausdrücken Ein Stapel kann zur Auswertung von arithmetischen Ausdrücken verwendet werden. Hierzu werden zwei Stapel für die Operatoren und die Operanden angelegt. Anschließend wird der arithmetische Ausdruck zeichenweise eingelesen. Die Operanden und Operatoren werden hierbei nacheinander auf

einen Stapel gelegt. Bei dem Auftreten einer schließenden Klammer werden die beiden obersten Operanden auf dem Stapel mit dem obersten Operator verknüpft. Das Resultat wird dann wieder auf den Stapel gelegt. Dieser Schritt wird für alle weiteren Operatoren in der Klammer sowie für die folgenden Klammersausdrücke wiederholt. Am Ende steht das Resultat des arithmetischen Ausdrucks an oberster Stelle.

Beispiel 3.20 Wir berechnen den Ausdruck $(11 + (3 + 4) * 2)$ mithilfe der beiden Stapel. Damit erhalten wir die folgende Auswertungsfolge:

Operanden	Operatoren	Ausdruck
		$(11 + (3 + 4) * 2)$
11		$+(3 + 4) * 2)$
11	+	$(3 + 4) * 2)$
11, 3	+	$+4) * 2)$
11, 3	+, +	$4) * 2)$
11, 3, 4	+, +	$*2)$
11, 7	+	$*2)$
11, 7	+, *	$2)$
11, 7, 2	+, *	$)$
11, 14	+	$)$
25		

Das Ergebnis des arithmetischen Ausdrucks ist damit 25.

Warteschlange

Eine Warteschlange (Queue) arbeitet nach dem FIFO-Prinzip, d. h., das zuerst eingefügte Element wird auch als erstes wieder entnommen. Bei einer Warteschlange werden die Elemente am einen Ende eingefügt und am anderen Ende entnommen. Warteschlangen besitzen in der Informatik eine Vielzahl von Anwendungen:

- Abarbeitung von Druckaufträgen,
- Prozessverwaltung im Betriebssystem.

Methoden Eine Warteschlange besitzt neben einem Konstruktor zum Erzeugen eines leeren Stapels die folgenden elementaren Operationen:

- `offer` stellt ein Element hinten in die Warteschlange;
- `peek` gibt das erste Element der Warteschlange zurück;
- `poll` löscht das erste Element der Warteschlange;
- `isEmpty` testet, ob Warteschlange leer ist.

Warteschlangen werden mit einfach verketteten Listen realisiert. Hierbei wird ein effizientes Anhängen am Ende durch einen zusätzlichen Zeiger auf das Schlüsselement

realisiert. Bei einer leeren Warteschlange besitzen die Zeiger auf das erste und letzte Element den Wert `null`. In Java ist eine Warteschlange bereits durch die Klasse `Queue` mit den zugehörigen Methoden vordefiniert und kann beispielsweise mit einer `LinkedList` realisiert werden:

```
public Queue
```

Beispiel 3.21 Füllung einer Warteschlange mit den Jobs von Aufträgen:

```
import java.util.LinkedList;
import java.util.Queue;
public class DS_Queue
{
    public static void main(String[] args)
    {
        Queue<String> q = new LinkedList<String>();
        q.offer("Job 1");
        q.offer("Job 2");
        q.offer("Job 3");
        q.offer("Job 4");

        while(!q.isEmpty())
        {
            System.out.println(q.peek());
            q.poll();
        }
    }
}
```

Ausgabe

```
Job 1
Job 2
Job 3
Job 4
```

3.3.4 Baumstrukturen

Bäume (Trees) können als eine Verallgemeinerung von Listen angesehen werden. Bei einem Baum hat jeder Knoten mehrere Nachfolger. Bäume als Datenstrukturen können sehr gut zum effizienten Abspeichern und Suchen eingesetzt werden.

Definition 3.1 Ein *Baum* besteht aus einer endlichen Anzahl von Knoten, die durch gerichtete Kanten verbunden sind (ohne lose Teile). Die *Wurzel* ist der einzige Knoten der keinen Eingang hat, alle übrigen Knoten haben genau einen Eingang. Die Knoten, die keinen Nachfolger besitzen, heißen *Blätter*. Eine gerichtete Kante beschreibt eine Vater-Sohn-Beziehung, d. h., den Zusammenhang zwischen Vorgänger (Vater) und Nachfolger (Sohn). Knoten, die denselben Vater haben, werden als *Geschwister* bezeichnet.

Bemerkung 3.1

1. Die Ebene eines Knotens ist die Länge des Pfades von diesem Knoten bis zur Wurzel.
2. Die Tiefe eines Baumes ist die maximale Ebene, auf der sich Knoten befinden.
3. Der Verzweigungsgrad eines Knotens ist die Anzahl seiner Kinder.
4. Jeder Knoten in einem Baum ist Wurzel eines Teilbaumes des gegebenen Baumes.
5. Der Teilbaum mit Wurzel k besteht aus dem Knoten k , seinen Kindern, Kindeskindern usw.
6. In einem Baum können Blätter auch als leere Knoten auftreten.

Bäume besitzen in der Informatik eine Vielzahl von Anwendungen:

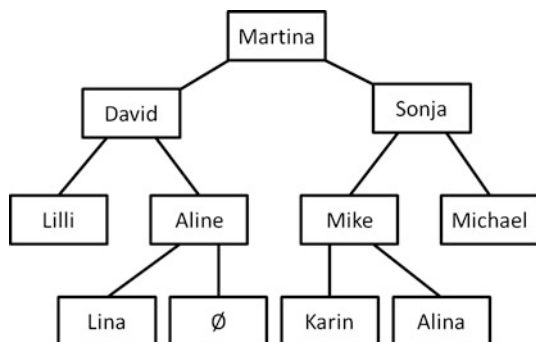
- indizierte Speicherung großer Datenmengen,
- Abspeicherung von Dateisystemen,
- Komprimierung von Daten,
- Ableitungsbäume einer Grammatik,
- Aufrufstruktur von rekursiven Algorithmen,
- Struktur eines mathematischen Ausdrucks,
- hierarchische Unterteilung von Objekten,
- Struktur von Entscheidungssequenzen.

Wir werden im Laufe dieses Buches noch auf einige Anwendungen zu sprechen kommen.

Definition 3.2 Ein Baum heißt *binär*, falls alle nicht leeren Knoten genau zwei Nachfolger besitzen. Ein Binärbaum heißt *voll*, falls alle inneren Knoten den Verzweigungsgrad 2 besitzen. Ein voller Binärbaum heißt *vollständig*, falls alle Blätter auf der gleichen Ebene liegen.

In Abb. 3.4 ist ein voller binärer Baum der Tiefe 3 dargestellt.

Abb. 3.4 Binärer Baum mit Namen



Bemerkung 3.2 Binärbäume können zur effizienten Speicherung von Daten benutzt werden:

Objekt	Zugriff auf Elemente	Einfügen eines Elements
Array	$O(1)$	$O(n)$
Liste	$O(n)$	$O(1)$
Binärbaum	$O(\log n)$	$O(\log n)$

Binäre Bäume besitzen zahlreiche Anwendungen, beispielsweise zur Repräsentation von mathematischen Ausdrücken, für die Implementierung von Sortierverfahren oder zur Darstellung eines Stammbaumes.

Implementierung eines Binärbaumes Wir definieren die Klasse `BinBaum` für Binärbäume, die Methoden für die verschiedenen Operationen auf Binärbäumen bereitstellt. Objekte der Klasse `BinBaum` haben eine Referenz auf den Wurzelknoten des Baumes. Ein Binärbaum besitzt neben den drei Konstruktoren die folgenden elementaren Operationen:

- `getRoot` gibt Wurzel zurück;
- `getLeft` gibt linken Teilbaum zurück;
- `getRight` gibt rechten Teilbaum zurück;
- `isEmpty` testet, ob der Binärbaum leer ist;
- `size` liefert Anzahl der Knoten.

Die Klasse `Knoten` stellt Objekte der Baumknoten dar. Jeder Knoten hat ein Objekt `obj` vom Typ `Object` und besitzt zwei Referenzen: eine auf den Wurzelknoten des linken Teilbaumes und eine auf den Wurzelknoten des rechten Teilbaumes. Die Referenz ist jeweils leer (`null`), wenn es keinen entsprechenden Teilbaum gibt. Ein Binärbaum besitzt neben den Konstruktoren die folgenden elementaren Operationen:

- `getObject` gibt Knotenobjekt zurück;
- `getLeft` gibt linken Knoten zurück;
- `getRight` gibt rechten Knoten zurück;
- `size` liefert Anzahl der Knoten.

```
class Knoten
{
    private Knoten links, rechts;
    private Object obj;
    public Knoten(Knoten links, Object obj, Knoten rechts)
    {
        this.links = links;
        this.obj = obj;
        this.rechts = rechts;
    }
}
```

```

public Object getObject()
{
    return obj;
}
public Knoten getLeft()
{
    return links;
}
public Knoten getRight()
{
    return rechts;
}
public int size()
{
    int sizeLinks = 0;
    int sizeRechts = 0;
    if (links != null)
        sizeLinks = links.size();
    if (rechts != null)
        sizeRechts = rechts.size();

    return 1 + sizeLinks + sizeRechts;
}
}

```

Ein Knoten-Objekt berechnet die Größe des von ihm repräsentierten Teilbaumes, indem es rekursiv die `size`-Methode auf seinem linken und rechten Knoten aufruft und so die Werte `sizeLeft` und `sizeRight` berechnen lässt. Die Größe des Teilbaumes ist dann $1 + \text{sizeLeft} + \text{sizeRight}$.

Die Klasse `BinBaum` implementieren wir dann mithilfe der Klasse `Knoten`:

```

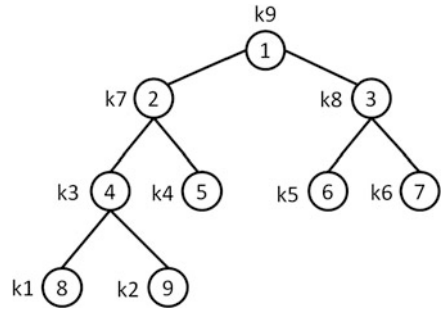
public class BinBaum
{
    private Knoten wurzel;

    public BinBaum()
    {
        wurzel = null;
    }
    public BinBaum(Object obj)
    {
        wurzel = new Knoten(null, obj, null);
    }
    public BinBaum(BinBaum links, Object obj, BinBaum rechts)
    {
        wurzel = new Knoten(links.wurzel, obj, rechts.wurzel);
    }

    public Object getRoot()
    {
        if (wurzel == null)
            throw new NoSuchElementException("Ein leerer Baum hat keine Wurzel.");
        return wurzel.getObject();
    }
}

```

Abb. 3.5 Binärer Baum mit 9
Knoten der Tiefe 3



```

public BinBaum getLeft()
{
    if (wurzel == null)
        throw new NoSuchElementException("Ein leerer Baum hat keinen linken Teilbaum.");
    BinBaum l = new BinBaum();
    l.wurzel = wurzel.getLeft();
    return l;
}
public BinBaum getRight()
{
    if (wurzel == null)
        throw new NoSuchElementException("Ein leerer Baum hat keinen rechten Teilbaum.");
    BinBaum r = new BinBaum();
    r.wurzel = wurzel.getRight();
    return r;
}
public boolean isEmpty()
{
    return wurzel == null;
}
public int size()
{
    if (wurzel == null)
        return 0;
    else
        return wurzel.size();
}
}

```

Der in Abb. 3.5 dargestellte Baum wird nun durch Verwendung der Klasse BinBaum mit den Instanzen k1 bis k8 instanziiert.

```

public static void main(String[] args)
{
    BinBaum k1 = new BinBaum(8);
    BinBaum k2 = new BinBaum(9);
    BinBaum k3 = new BinBaum(k1, 4, k2);
    BinBaum k4 = new BinBaum(5);
    BinBaum k5 = new BinBaum(6);
    BinBaum k6 = new BinBaum(7);
    BinBaum k7 = new BinBaum(k3, 2, k4);
    BinBaum k8 = new BinBaum(k5, 3, k6);
    BinBaum k9 = new BinBaum(k7, 1, k8);
}

```

```
System.out.printf("Anzahl der Knoten im Baum: %d\n", k9.size());
System.out.printf("Anzahl der Knoten im Teilbaum k7: %d\n", k7.size());
System.out.printf("Name des Wurzelknotens vom Teilbaum k7: %d\n", k7.getRoot());
}
```

Ausgabe

Anzahl der Knoten im Baum: 9
Anzahl der Knoten im Teilbaum k7: 5
Name des Wurzelknotens vom Teilbaum k7: 2

3.3.5 Graphen

Graphen sind Modelle für verschiedene Netzwerke wie beispielsweise Verkehrsnetze, Computernetze, Schaltnetze, Molekülstrukturen oder für Planungsablaufpläne. Ein Graph besteht aus genau zwei Objekten, den Knoten (z. B. Städte, Computer) und den Kanten (z. B. Straßen, Leitungen) zwischen zwei Knoten. Die Kanten können dabei gerichtet oder ungerichtet sein.

Definition 3.3 Ein Graph $G = (V, E)$ besteht aus einer Knotenmenge V und einer Kantenmenge E , wobei jeder Kante $e \in E$ zwei (nicht notwendigerweise verschiedene) Knoten aus V zugeordnet sind. Eine *gerichtete* Kante zwischen zwei Endknoten u und v wird in der Form $e = (u, v)$ und eine *ungerichtete* Kante durch $e = \{u, v\}$ beschrieben.

Graphen wie in Abb. 3.6 dargestellt besitzen in der Informatik eine Vielzahl von Anwendungen:

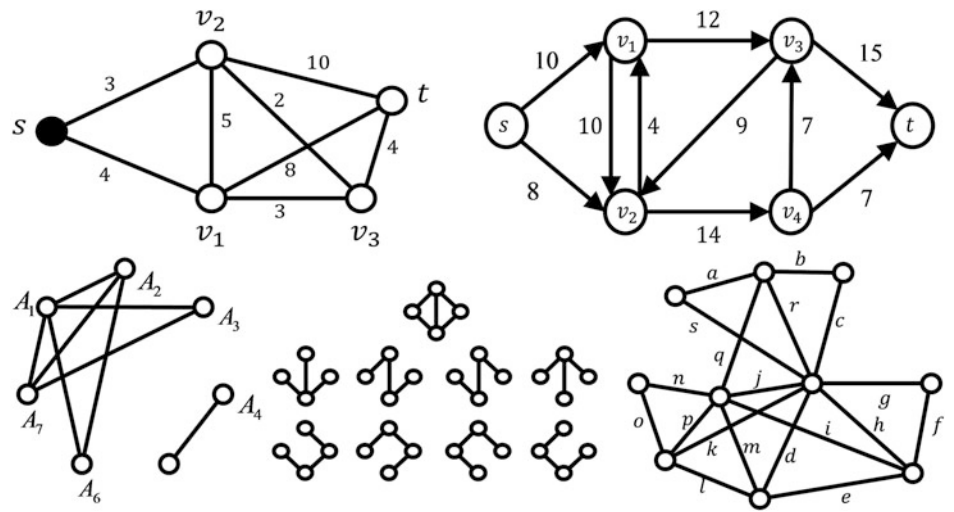
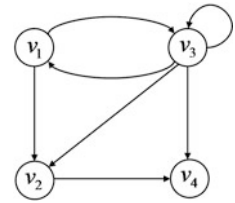


Abb. 3.6 Verschiedene Arten von Graphen

Abb. 3.7 Gerichteter Graph mit 4 Knoten



- Ermittlung kürzester Wege zwischen Städten,
- Planung eines Kommunikationsnetzes oder Versorgungsnetzes.
- Bestimmung des größten Durchflusses von Netzen,
- schnellstmögliche Belieferung von Kunden,
- kostenminimaler Transport von Gütern.

Die Implementierung eines Graphen erfolgt beispielsweise mithilfe der Matrizendarstellung. Sei $G = (V, E)$ ein Graph mit der Knotenmenge $V = \{v_1, v_2, \dots, v_n\}$. Die *Adjazenzmatrix* $A = (a_{ij})$ von G ist eine $n \times n$ Matrix, wobei $a_{i,j}$ die Anzahl der Kanten zwischen v_i und v_j angibt.

Beispiel 3.22 In Abb. 3.7 ist ein Graph mit 4 Knoten dargestellt. Die zugehörige Adjazenzmatrix sieht wie folgt aus:

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

Bei gewichteten Graphen gibt die Adjazenzmatrix $A = (a_{ij})$ von G das Gewicht (Entfernung, Kosten usw.) zwischen den Knoten v_i und v_j an.

Implementierung eines Graphen Die Datenstruktur `Graph` besitzt die Instanzvariablen in Form der Adjazenzmatrix `A`, die Anzahl der Knoten `n` und den booleschen Wert `gerichtet`, der angibt, ob ein Graph gerichtet ist oder nicht. Neben verschiedenen Konstruktoren zur Initialisierung eines Graphen enthält diese Klasse noch die folgenden elementaren Operationen:

- `addKante` fügt eine gewichtete und gerichtete Kante zwischen zwei Knoten ein;
- `loescheKante` löscht eine gewichtete Kante zwischen zwei Knoten;
- `getKnotenanzahl` gibt die Knotenanzahl zurück;
- `getKantenanzahl` gibt die Kantenanzahl zurück;
- `getKnotengrad` gibt den Knotengrad eines Knotens zurück;
- `getNachbar` gibt die Menge der Nachbarknoten zurück;

- getMaxGrad gibt den maximalen Knotengrad zurück;
- getMinGrad gibt den minimalen Knotengrad zurück;
- loescheKnotenmenge löscht eine Menge von Knoten und zugehörigen Kanten.

Die Implementierung der Klasse Graph mit einigen der besprochenen Methoden sieht dann wie folgt aus:

```
public class Graph
{
    private double A[][];           // Adjazenzmatrix
    private int n;                   // Anzahl der Knoten
    private boolean gerichtet;       // Schalter f. gerichtet

    public Graph(int n, boolean gerichtet)
    {
        A = new double[n][n];
        this.n = n;
        this.gerichtet = gerichtet;
    }

    public Graph(double A[][], boolean gerichtet)
    {
        this.A = A;
        this.n = A.length;
        this.gerichtet = gerichtet;
    }

    // Kante hinzufuegen zwischen (v1, v2) (gerichtet) bzw. {v1, v2} ungerichtet
    public void addKante(int v1, int v2, double gewicht)
    {
        A[v1][v2] = gewicht;
        if (!gerichtet)
            A[v2][v1] = gewicht;
    }

    // Berechnung der Knotenanzahl
    public int getKnotenanzahl()
    {
        return n;
    }

    // Berechnung der Kantenanzahl
    public int getKantenanzahl()
    {
        int m = 0;
        for(int i=0; i<n; i++)
            for(int j=0; j<n; j++)
                if (A[i][j] > 0)
                    m = m + 1;

        if (gerichtet)
            return m;
        else
            return m/2;
    }
}
```

Der obige Graph in Abb. 3.7 wird dann wie folgt definiert:

```
double A[] [] = {{0,1,1,0}, {0,0,0,1}, {1,1,1,1}, {0,0,0,0}};  
Graph g = new Graph(A, true)
```

3.4 Übungsaufgaben

Aufgabe 3.1 (Exception) Erstellen Sie eine grafische Oberfläche zum Einlesen von Personendaten (z. B. Name, Vorname, Adresse, Geburtsjahr), und schreiben Sie die eingegebenen Daten in eine Datei. Definieren Sie für alle Eingaben eine Exception-Behandlung, insbesondere für die eingelesenen Zahlen (`NumberFormatException`). Definieren Sie weiterhin eine passende Exception, wenn das Geburtsjahr nicht in einem sinnvollen Intervall liegt.

Aufgabe 3.2 (Exception) Ergänzen Sie für einige ausgewählte Programme aus dem 1. Band eine Exception-Behandlung. Behandeln Sie entweder die Exception sofort, oder geben Sie diese mit dem Befehl `throw` weiter.

Aufgabe 3.3 (Multithreading) Erstellen Sie eine Animation mit einem Objekt, das sich in einem Fenster bewegt. Wenn das Objekt an den Rand kommt, soll dieses dort reflektiert werden. Erweitern Sie anschließend die Anwendung um mehrere verschiedene Objekte, die sich auf der Zeichenebene wie beschrieben bewegen sollen.

Aufgabe 3.4 (Multithreading) Erstellen Sie eine Ampelsteuerung, indem Sie eine GUI mit einer Ampel entwerfen. Mithilfe eines `Threads` soll anschließend die Ampel animiert werden, sodass die Ampel die vier Phasen Rot, Rotorange, Grün und Gelb in vordefinierten zeitlichen Phasen durchläuft. Erweitern Sie die Anwendung anschließend um eine zweite Ampel mit unterschiedlichen Phasenzeiten sowie um einen Start- und Stopp-Button.

Aufgabe 3.5 (Datenbanken) Erstellen Sie eine Klasse mit passenden Methoden zum Erstellen, Bearbeiten und Ausgeben von SQL-Daten. Übertragen Sie anschließend die aus Aufgabe 3.1 beschriebene Personaldatenbank in eine SQL.

Aufgabe 3.6 (Verkettete Listen) Implementieren Sie in der Datenstruktur `Liste` Operationen, um ein k -tes Element einzufügen, zu löschen oder auszugeben.

Aufgabe 3.7 (Verkettete Listen) Implementieren Sie eine doppelt verkettete Liste, bei der jedes Element einen Zeiger auf das nachfolgende als auch auf das vorhergehende Element besitzt.

Aufgabe 3.8 (Warteschlange) Implementieren Sie eine eigene Datenstruktur für einen Stack und eine Warteschlange.

Aufgabe 3.9 (Stack) Erstellen Sie ein Programm zur Auswertung von arithmetischen Ausdrücken mithilfe eines Stacks. Geben Sie die Ausdrücke entweder über eine Konsole oder eine einfache GUI ein.

Aufgabe 3.10 (Baum) Implementieren Sie eine Datenstruktur Baum zur Darstellung von beliebigen Bäumen. Verwenden Sie die Datenstruktur für eine der oben beschriebenen Anwendungen.

Aufgabe 3.11 (Graph) Implementieren Sie in die Datenstruktur Graph alle beschriebenen Methoden, die bisher noch nicht implementiert sind. Testen Sie die Datenstruktur an einigen praktischen Beispielen.

Literaturhinweise⁴

1. Krüger, G. (2014). *Handbuch der Java-Programmierung*. O'Reilly.
2. Ullenboom, C. (2013). *Java ist auch eine Insel*. Galileo Computing.
3. Louis, D., Müller, P. (2013). *Java 7 – Das Handbuch*. Pearson.
4. Gennick, J. (2007). *SQL - kurz & gut*. O'Reilly.
5. Sedgewick, R., Wayne, K. (2011). *Einführung in die Programmierung mit Java*. Pearson.
6. Pomberger, G., Dobler, H. (2008). *Algorithmen und Datenstrukturen*. Pearson.
7. Schiedermeier, R., Köhler, K. (2011). *Das Java-Praktikum*. dpunkt.

⁴ Die Bücher [1], [2] und [3] sind sehr gut geeignet für das Erlernen und Nachschlagen weiterer Programmierkonzepte in Java. Leser, die weitere Informationen zu SQL benötigen, finden in [4] einen sehr kompakten Einstieg. Weitere praktische Anwendungsbeispiele zu der Verwendung von dynamischen Datenstrukturen findet der Leser beispielsweise in [5], [6] und [7].

In Technik und Naturwissenschaft gibt es eine große Anzahl von praktischen Problemen, die mittels spezieller algorithmischer Suchverfahren gelöst werden. Beispiele sind die Suche nach einer günstigen Route zwischen zwei Ortspunkten, die Berechnung der Reihenfolge der Arbeitsschritte bei der Produktionsplanung, die Suche nach einer Lösung in sehr großen Suchbäumen, die Suche nach Gensequenzen in der Bioinformatik oder die näherungsweise Lösung von Optimierungsproblemen. Die Suchverfahren stellen auch die Grundlage vieler Systeme der künstlichen Intelligenz (KI-Systeme) dar, die durch die fortschreitende Digitalisierung in nahezu allen Wissensdisziplinen Einsatzmöglichkeiten besitzen.

Die Suchverfahren lassen sich grob in drei verschiedene Kategorien einteilen: uninformierte, informierte und lokale Suche. Die uninformierte Suche besteht aus dem blinden Durchprobieren aller Möglichkeiten durch verschiedene Suchstrategien in Form der Breiten- und Tiefensuche. Die Breiten- und Tiefensuche hängt dabei nur von der vorgegebenen Struktur der Suchbäume ab. Diese Suchtechnik ist sehr einfach, aber auch blind gegenüber dem konkreten Suchproblem, was in vielen Fällen zu sehr hohen Rechenzeiten führt.

Bei der informierten Suche werden zusätzliche Informationen verwendet, um zu entscheiden, welcher Pfad durch den Suchbaum weiter verfolgt wird. Mithilfe einer Bewertungsfunktion wird dann die vielversprechendste Alternative zuerst untersucht. Oftmals bezeichnet man diese Verfahren auch als heuristische Suche, da man hierzu gewisse heuristische und problemspezifische Bewertungen heranzieht. Beispiele für diese Art der Suche ist die sogenannte gierige Suche oder die A*-Suche. Der A*-Algorithmus garantiert unter gewissen Voraussetzungen auch ein optimales Suchergebnis. In praktischen Anwendungen findet dieses Verfahren dadurch eine breite Anwendungsmöglichkeit.

Zu den sogenannten lokalen Suchalgorithmen zählen diverse heuristische Verfahren aus dem Bereich der evolutionären Algorithmen zur Lösung von diversen Optimierungsproblemen. Zu den zwei bekanntesten Verfahren dieser Klasse zählen Simulated Annealing und genetische Algorithmen. Diese algorithmischen Verfahrenstypen haben eine

herausragende Bedeutung bei der Lösung von vielen technischen oder naturwissenschaftlichen Problemstellungen.

In diesem Kapitel erklären wir die drei verschiedenen Suchtechniken der uninformierten, informierten und lokalen Suche an zahlreichen konkreten Verfahren. Insbesondere im Bereich der künstlichen Intelligenz besitzen diese Verfahrenstypen die vielfältigsten Anwendungsmöglichkeiten.

4.1 Allgemeine Grundlagen

Bei einem Suchproblem wird von einem Startzustand ein Zielzustand in der Menge aller Zustände gesucht. In Abhängigkeit der gegebenen Problemstellung sind nur gewisse Aktionen erlaubt. Eine Kostenfunktion ordnet jeder Aktion dann einen Kostenwert zu. Die Lösung ist dann ein kostenoptimaler Pfad im Suchbaum vom Startzustand zum Zielzustand.

Ein typisches Suchproblem ist beispielsweise eine Routenplanung. Gegeben ist eine Karte mit Städten als Knoten und Straßen zwischen den Städten als gewichtete Kanten mit Entfernungen. Gesucht ist eine möglichst optimale Route von einer Stadt A zu einer Stadt B . Der Zustandsraum sind alle Städte mit der Stadt A als Startzustand und der Stadt B als Zielzustand. Die Aktionen sind der Übergang von der aktuellen Stadt zu einer Nachbarstadt. Die Kostenfunktion gibt dabei die Entfernung zwischen den Städten an. Die Lösung ist eine kürzeste Route vom Startzustand zum Zielzustand.

Begriff des Suchproblems Wir betrachten eine Möglichkeit zur Definition eines allgemeinen Suchproblems:

Definition 4.1 Ein *Suchproblem* ist ein 5-Tupel

$$P = (Z, z_0, z_E, A, c),$$

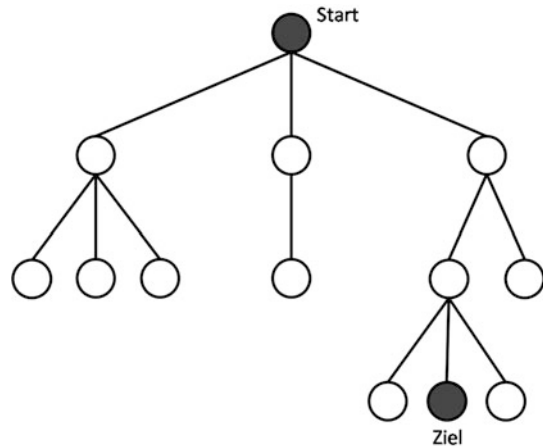
hierbei sind

1. Z eine endliche Menge (*Zustands- bzw. Suchraum*),
2. $z_0 \in Z$ ein Element des Zustandsraums (*Startzustand*),
3. $z_E \in Z$ ein Element des Zustandsraums (*Endzustand*),
4. $A \subseteq Z \times Z$ eine Menge von Tupel (*Aktionen*),
5. $c : A \rightarrow \mathbb{R}^2$ eine Abbildung (*Kostenfunktion*).

Eine *Lösung* eines Suchproblems ist ein Pfad p im Suchbaum vom Startzustand zum Zielzustand.

In Abb. 4.1 ist ein Suchbaum mit einem Pfad zwischen Startzustand und Zielzustand abgebildet. In der Sprache der Graphentheorie können wir alternativ auch die Menge der

Abb. 4.1 Suchbaum vom Startzustand zum Zielzustand



Zustände des Suchproblems als Knoten V bezeichnen. Die Menge der erlaubten Aktionen von einem Zustand $v \in V$ sind dann alle Nachbarknoten $v \in N_G(v)$ im zugrunde liegenden Graphen $G = (V, E)$. Der Startzustand wird mit $s \in V$ und der Zielzustand mit $t \in V$ bezeichnet. Die Lösung ist ein Pfad p vom Startzustand s zum Zielzustand t .

Bemerkung 4.1 Die Anzahl der Zustände in einem Suchbaum kann sehr groß sein. Ein Baum mit konstantem Verzweigungsfaktor b und Tiefe d hat dann die folgende Knotenanzahl:

$$n = \sum_{i=0}^d b^i = \frac{b^{d+1} - 1}{b - 1}.$$

Bei einem Verzweigungsfaktor von $b = 20$ und einer Tiefe von $d = 40$ besitzt der Suchbaum die folgende Anzahl von Zuständen:

$$\sum_{i=0}^{40} 20^i = \frac{20^{41} - 1}{20 - 1} = 1,2 \cdot 10^{52}.$$

Bei einem Computer, der 10^9 Rechenschritte pro Sekunde benötigt, beträgt die Rechenzeit $3,8 \cdot 10^{44}$ Jahre. Das Fazit ist, dass effiziente Suchstrategien notwendig sind, um in absehbarer Zeit große Suchprobleme zu lösen.

Eine typische Anwendung von Suchproblemen mit sehr großem Zustandsraum ist das Schachspiel. Menschen verwenden intelligente Strategien, die den Suchraum deutlich reduzieren. Der Computer ist jedoch nicht in der Lage, durch bloßes „Draufschaun“ auf das Schachbrett viele der möglichen Aktionen sofort auszuschließen. Dieses sogenannte „Draufschaun“ muss dem Rechner durch spezielle Suchalgorithmen beigebracht werden.

Eigenschaften Suchalgorithmen werden nach Vollständigkeit, Optimalität, Zeit- und Speicherkomplexität klassifiziert.

Definition 4.2 Ein Suchalgorithmus heißt

- *vollständig*, wenn er für jedes lösbare Problem eine Lösung findet;
- *korrekt*, wenn ein berechneter Pfad wirklich eine Lösung ist;
- *optimal*, falls er immer den Pfad mit den niedrigsten Kosten findet;
- *deterministisch*, falls jede Aktion zu einem eindeutig bestimmten Nachfolgezustand führt.

Falls ein vollständiger Suchalgorithmus terminiert ist, ohne eine Lösung zu finden, ist das zugehörige Suchproblem nicht lösbar. Vollständige oder optimale Suchalgorithmen sind in vielen Fällen jedoch nicht zweckmäßig, da sie einen enormen Rechenbedarf bei großen Suchproblemen besitzen. In vielen Fällen sind heuristische und nicht deterministische Verfahren, die gute Näherungslösungen finden, deutlich praktikabler.

Klassifizierung von Verfahren Die Methodiken zum Suchen lassen sich grob in die folgenden drei Klassen einordnen:

1. **uninformierte Suche:** blindes Durchprobieren aller Möglichkeiten durch verschiedene Suchstrategien in Form der Breiten- und Tiefensuche.
2. **informierte Suche:** Verwendung einer Heuristikfunktion zur Abschätzung der Kosten einer Lösung, beispielsweise in Form der gierigen Suche oder der A*-Suche;
3. **lokale Suche:** Suche das Element aus dem Suchraum mit kleinstem oder größtem Zielfunktionswert mithilfe evolutionärer Algorithmen wie beispielsweise Simulated Annealing oder genetische Algorithmen.

Der Vorteil der informierten Suche ist, dass diese Suchstrategie in vielen Fällen deutlich schneller ist als die uninformierte Suche. Der Nachteil liegt in der Tatsache, dass es keine Garantie für das Finden einer optimalen Lösung gibt. In praktischen Anwendungen interessieren wir uns häufig für eine schnell gefundene gute Lösung. Aufgrund der hohen Laufzeit von exakten Verfahren werden hierzu Heuristiken verwendet, um eine gute, zulässige Lösung zu bestimmen.

Heuristiken nutzen problemspezifische Informationen, die es ermöglichen schneller zu einer Näherungslösung zu gelangen als durch blindes Suchen. Sie bieten jedoch keine Garantie, dass ein Optimum gefunden wird. Die lokalen Suchverfahren eignen sich mit ihrer nahezu universellen Einsetzbarkeit für eine große Anzahl von Optimierungsproblemen sehr gut.

Ausgewählte Anwendungen Einige der bekanntesten Anwendungen von Suchverfahren sind die folgenden:

- **Routenplanung:** Berechnung des kürzesten Weges zwischen zwei oder mehreren Ortspunkten. Das bekannteste Lösungsverfahren ist die Dijkstra-Suche oder der A*-Algorithmus.

- **Roboternavigation:** Navigation eines Roboters durch eine bekannte oder unbekannte Umgebung. Bei der Navigation werden sehr häufig Suchpfade mit dem A*-Algorithmus bestimmt.
- **Spielprobleme:** Suche nach optimalen Zügen in Spielproblemen. Das Schachspiel ist ein klassisches Spielproblem, bei dem vor allem heuristische Suchverfahren wie der A*-Algorithmus verwendet werden.
- **Chiplayoutplanung:** Berechnung der Anordnung von Bauelementen und Verbindungen auf einem Computerchip um Fläche und Schaltverzögerungen zu minimieren. Bei dieser Art von Problemstellung kommen durch den großen Suchraum heuristische Verfahren wie genetische Algorithmen zur Anwendung.
- **Baugruppenauslegung:** Optimierung von Baugruppen oder Fertigungszeiten im Bereich der Produktentwicklung. Bei diesen Aufgaben kommen evolutionäre Algorithmen wie beispielsweise das universell einsetzbare Simulated-Annealing-Verfahren zur Anwendung.

4.2 Breiten- und Tiefensuche

Die einfachste Art der Suche ist das systematische Durchprobieren aller Möglichkeiten. Wir starten bei einem Startzustand und führen der Reihe nach alle möglichen Aktionen aus. Von den so erreichbaren Zuständen suchen wir weiter, bis der Zielzustand erreicht wird. Diese Art von Suche wird daher als blinde Suche bezeichnet.

Wir unterscheiden hierbei die folgenden zwei elementaren Basisalgorithmen:

- **Breitensuche** erweitert zuerst den flachsten Knoten über ein iteratives Verfahren.
- **Tiefensuche** erweitert zuerst den tiefsten, nicht expandierten Knoten über ein rekursives Verfahren.

Alle diese Suchverfahren sind vollständig und optimal für allgemeine Schrittkosten. Diese Suchstrategien werden in sehr vielen algorithmischen Verfahren verwendet. Wir betrachten in Abb. 4.2 einen Graphen mit einem Startknoten s , der im Folgenden mit der Breiten- und Tiefensuche in der jeweiligen Suchreihenfolge bearbeitet wird.

4.2.1 Breitensuche

Bei der Breitensuche wird der Graph ausgehend vom Startknoten s in die Breite nach einem Element durchsucht. Vom Startknoten s wird nun jede Kante (s, v) betrachtet und getestet, ob der Knoten v schon bearbeitet wurde bzw. das gesuchte Element ist. Ist dies noch nicht der Fall, so wird der entsprechende Knoten in einer Warteschlange gespeichert und im nächsten Schritt bearbeitet.

Abb. 4.2 Graphenstruktur zum Durchsuchen mit Breiten- und Tiefensuche

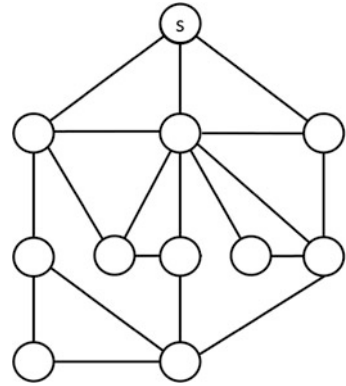
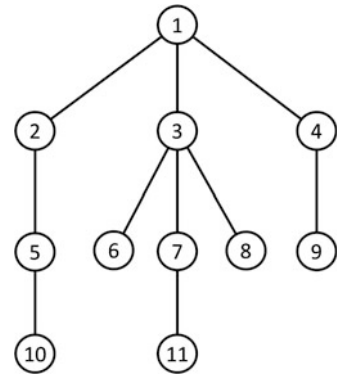


Abb. 4.3 Reihenfolge der Breitensuche durch Nummerierung der Knoten



Beispiel 4.1 In Abb. 4.3 ist das Prinzip der Breitensuche durch Nummerierung der Knotenreihenfolge dargestellt. Vom Startknoten (1) werden zuerst alle drei Nachbarknoten (2, 3, 4) untersucht. Anschließend werden dann alle Nachbarn dieser drei Knoten betrachtet (5, 6, 7, 8, 9). Da der gegebene Suchbaum eine Tiefe von 3 besitzt, werden im letzten Schritt die beiden Knoten der Tiefe 3 besucht (10, 11). Danach sind alle Knoten genau einmal besucht worden. Nach jeder Aktion wird die zugehörige Kostenfunktion berechnet. Die optimale Lösung wird durch das Suchprinzip der Breitensuche immer gefunden.

Prinzip der Breitensuche (BFS)

1. Speicherung des Anfangsknotens s in einer Warteschlange Q
2. Entnahme und Markierung des Knotens am Beginn der Warteschlange
 - (a) Falls das gesuchte Element gefunden wurde $\Rightarrow$ fertig
 - (b) Andernfalls werden alle unmarkierten Nachfolger dieses Knotens, die sich noch nicht in Q befinden, bestimmt und an das Ende der Warteschlange angehängt
3. Falls Q leer ist, ist bereits jeder Knoten untersucht $\Rightarrow$ keine Lösung
4. Wiederholung von Schritt 2

Für die Warteschlange Q verwenden wir die bereits vorgestellten Methoden:

- **OFFER** stellt ein Element hinten in die Warteschlange;
- **PEEK** gibt das 1. Element der Warteschlange zurück;
- **POLL** löscht das 1. Element der Warteschlange.

Algorithmus 1 BREITENSUCHE

Input: Graph $G = (V, E)$, Startknoten s
Output: Nummerierungsreihenfolge $z(v)$, $v \in V$
Komplexität: $O(m + n)$

```

1: for  $v \in V$  do
2:    $z(v) = 0$ 
3:    $Q = \emptyset$ 
4:    $Q.OFFER(s)$ 
5:    $nr = 1$ 
6:   while  $Q \neq \emptyset$  do
7:      $v = Q.PEEK()$ 
8:     if  $z(v) = 0$  then
9:        $z(v) = nr$ 
10:       $nr = nr + 1$ 
11:     $Q.POLL()$ 
12:    for  $(v, w) \in E$  do
13:      if  $z(w) = 0$  and  $w \notin Q$  then
14:         $Q.OFFER(w)$ 

```

Allgemeine Erklärung

In Zeile 1–5 werden der Vektor z der Knotenreihenfolge und die Warteschlange mit dem Startknoten s initialisiert. Die folgende Schleife wird so lange durchlaufen, wie es noch unbesuchte Knoten gibt. In Zeile 7–11 wird das 1. Element v der Warteschlange Q entnommen, und über den Wert von $z(v)$ wird geprüft, ob es schon besucht wurde. Wenn dies der Fall ist, wird die aktuelle Nummer gesetzt. In Zeile 12–14 werden alle unbesuchten Nachbarknoten w von G in die Warteschlange Q hinzugefügt, die hier noch nicht vorhanden sind.

Aufwandsabschätzung

Der Algorithmus fügt jeden Knoten höchstens einmal in die Warteschlange Q ein. Jede Kante wird höchstens einmal verwendet. Die Anzahl der Schritte der Breitensuche ist somit proportional zur Anzahl der Knoten n und Kanten m von G , also aus $O(n + m)$.

Die Implementierung in Java erfolgt als Methode der bereits vorgestellten Klasse `Graph` mit der Datenstruktur `Vector` und den zugehörigen Methoden zum Einfügen `addElement` und Löschen `removeElement` von Elementen:

```

public int[] breitensuche(int s)
{
    // Nummerierung der besuchten Knotenreihenfolge
    int zahl[] = new int[n];

    // Definition der Warteschlange
    Queue<Integer> Q = new LinkedList<Integer>();
    Q.offer(s);

    int v, nr = 1;
    while(Q.size() > 0)
    {
        v = (int) Q.peek();
        if(zahl[v] == 0)    // Knoten noch nicht besucht
        {
            zahl[v] = nr;
            nr      = nr + 1;
        }
        Q.poll(); // Lösche aktuellen Knoten

        // Bestimmung aller Nachbarn
        Vector<Integer> NG = getNachbarn(v);
        for(int i=0; i<NG.size(); i++)
        {
            int w = (int) NG.elementAt(i);
            if((zahl[w] == 0) && (!Q.contains(w))) //Test auf kreisfreiheit
                Q.offer(w);
        }
    }
    return zahl;
}

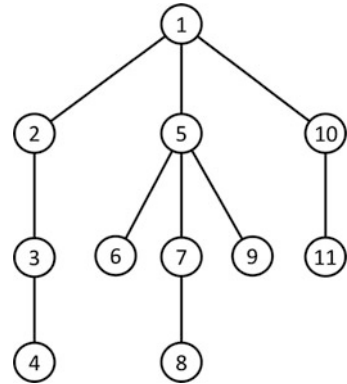
```

4.2.2 Tiefensuche

Bei der Tiefensuche wird der Graph ausgehend vom Startknoten s weiter in der Tiefe untersucht. Vom Startknoten s wird nun die erste Kante (s, v) betrachtet und getestet, ob der gegenüberliegende Knoten v schon entdeckt wurde bzw. das gesuchte Element ist. Ist dies noch nicht der Fall, so wird rekursiv für diesen Knoten die Tiefensuche aufgerufen, wodurch wieder der erste Nachfolger u dieses Knotens v untersucht wird. Die Nachfolger des Anfangsknotens werden in einem Stapel gespeichert.

Diese Art der Suche wird so lange fortgesetzt, bis das gesuchte Element entweder gefunden wurde oder die Suche in einem Knoten angekommen ist, der keine weiteren Nachfolger mehr besitzt. An dieser Stelle kehrt der Algorithmus nun zum zuletzt betrachteten Knoten zurück und untersucht den nächsten Nachfolger von Knoten. Sollte es hier keine weiteren Nachfolger mehr geben, geht der Algorithmus wieder Schritt für Schritt zum jeweiligen Vorgänger zurück und versucht es dort erneut.

Abb. 4.4 Prinzip der Tiefensuche durch Nummerierung der Knotenreihenfolge



Beispiel 4.2 In Abb. 4.4 ist das Prinzip der Tiefensuche durch Nummerierung der Knotenreihenfolge dargestellt. Vom Startknoten (1) werden alle Nachbarknoten und die damit entstehenden Teilbäume (2, 3, 4), (5, 6, 7, 8, 9) und (10, 11) untersucht. Nach jeder Aktion wird die zugehörige Kostenfunktion berechnet. Die optimale Lösung wird durch das Suchprinzip der Tiefensuche immer gefunden.

Prinzip der Tiefensuche (DFS)

1. Speicherung aller Nachfolger des Anfangsknotens s in einem Stapel S
2. Rekursiver Aufruf der Tiefensuche für jeden nicht markierten Knoten in s :
Falls das gesuchte Element gefunden worden ist $\Rightarrow$ fertig

Im folgenden Pseudocode wird der aktuelle Zähler für die Nummerierung am Ende des Arrays z gespeichert. Die Variable z und der Zähler nr werden dabei in dem aufrufenden Hauptprogramm initialisiert.

Algorithmus 2 TIEFENSUCHE

Input: Graph $G = (V, E)$, Startknoten s
Output: Nummerierungsreihenfolge $z(v)$, $v \in V$
Komplexität: $O(m + n)$

```

1: for  $(s, v) \in E$  do
2:   if  $z(v) = 0$  then
3:      $z(v) = nr$ 
4:      $nr = nr + 1$ 
5:     TIEFENSUCHE( $G, v$ )
  
```

Allgemeine Erklärung

Für alle Nachbarknoten v des Startknotens s , die noch nicht besucht sind, wird rekursiv die Prozedur der Tiefensuche mit dem neuen Startknoten v aufgerufen.

Aufwandsabschätzung

Der Algorithmus fügt jeden Knoten höchstens einmal in die Datenstruktur S ein. Jede Kante wird höchstens einmal verwendet. Die Anzahl der Schritte der Tiefensuche ist somit proportional zur Anzahl der Knoten und Kanten von G , also aus $O(n + m)$.

Die Implementierung der Tiefensuche erfolgt wieder in die Klasse `Graph`. Dabei wird der Prozedur das Array `zahl` des Zählers übergeben, an dessen letztem Element die aktuelle Zählvariable `nr` gespeichert wird.

```
public void tiefensuche(int s, int zahl[])
{
    Vector<Integer> NG = getNachbarn(s);
    for(int i=0; i<NG.size(); i++)
    {
        int w = (int) NG.elementAt(i);
        if(zahl[w] == 0)
        {
            zahl[w] = zahl[n];
            zahl[n] = zahl[n] + 1;
            tiefensuche(w, zahl);
        }
    }
}
```

4.3 Dijkstra-Suche

Eines der wichtigsten Optimierungsprobleme ist es, einen kürzesten Weg zwischen zwei bestimmten Knoten eines gerichteten oder ungerichteten Graphen zu finden. In vielen praktischen Anwendungen benötigt man kürzeste Wege oft als Teilproblem eines viel schwierigeren graphentheoretischen Optimierungsproblems.

4.3.1 Einführendes Beispiel

Die Routenplanung ist einer der wichtigsten Anwendungsbereiche für die Bestimmung von kürzesten Wegen in Straßennetzwerken. Wir nehmen an, wir wollen als Autofahrer vom Hochschulcampus aus der Tuttlinger Innenstadt, die leider zum größten Teil aus Einbahnstraßen besteht, den schnellsten Weg nach Hause finden (siehe Abb. 4.5). Dieses Problem lässt sich sehr einfach als gerichteter und gewichteter Graph modellieren: Die Kreuzungen in der Stadt werden als Knoten und die Einbahnstraßen als gerichtete Kanten aufgefasst. Das Gewicht einer Kante ist die Weglänge oder die Fahrzeit. Das Problem des

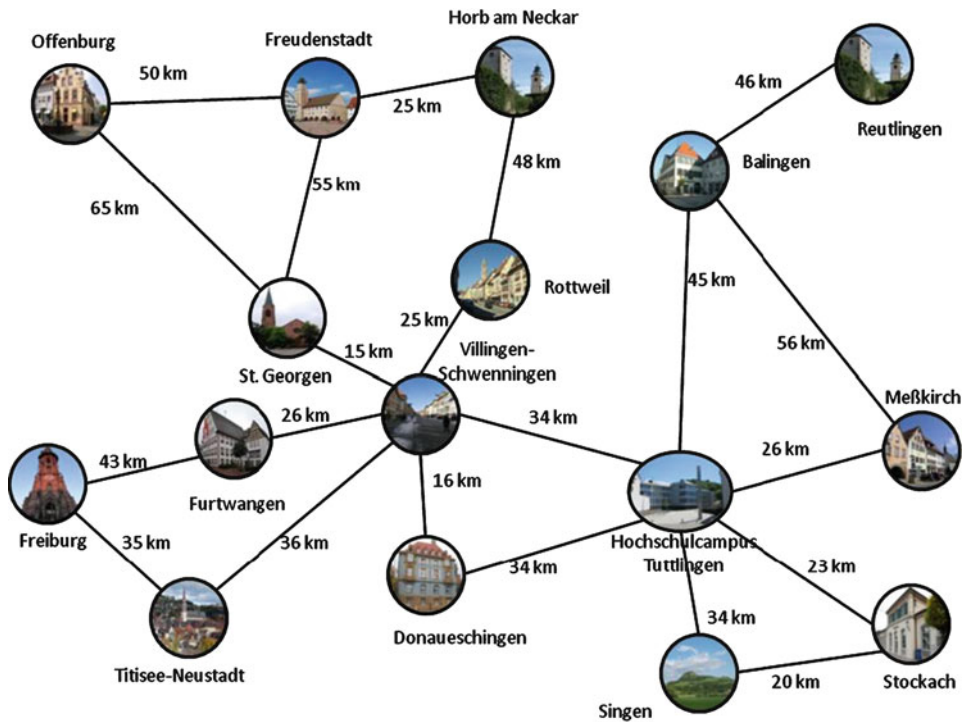


Abb. 4.5 Routenplanungsproblem in der Region um Tuttlingen

kürzesten Weges vom Campus s zum Heimatort t wird dann zu einem Wegeproblem im Graphen: Wie lautet der kürzester Weg vom Knoten s zum Knoten t ?

Naives Ausprobieren würde vielleicht noch im Umkreis von Tuttlingen zum Ziel führen. Doch bei einer Fahrt von Tuttlingen nach Hamburg würde man hier bereits an die Grenzen stoßen. Allein das Straßennetz Deutschlands umfasst ca. 5 Mio. Kreuzungen und ca. 6,2 Mio. Straßen. Das Ausprobieren aller Möglichkeiten liefert nur ein Verfahren, bei dem selbst der schnellste Computer mehrere Jahre rechnen würde. Und so lange wollen wir nicht auf die Wegbeschreibung warten. Jeder, der ein Navigationssystem im Auto besitzt, weiß, dass dieser Routenplaner die Antworten im Bereich von Sekunden liefert. Und dieser kleine Kasten an der Scheibe ist garantiert kein Supercomputer. Wie sehen also diese leistungsfähigen Verfahren zur Bestimmung eines kürzesten Weges aus? Diese Frage werden wir in diesem Abschnitt beantworten.

4.3.2 Problemstellung

Wir betrachten einen gerichteten Graphen $G = (V, E)$ mit der Kantenlänge $c : E \rightarrow \mathbb{R}$. Falls ein ungerichteter Graph gegeben ist, können wir jede ungerichtete Kante durch eine

gerichtete Kante ersetzen. Wenn zwischen zwei Knoten u und v keine Kante verläuft, schreiben wir $c(u, v) = \infty$.

Definition 4.3 Seien $G = (V, E)$ ein Graph und $c : E \rightarrow \mathbb{R}$ eine Gewichtungsfunktion. Die Länge $c(P)$ eines Weges P_{st} in G von s nach t ist definiert durch

$$c(P) = \sum_{i=1}^{k-1} c(v_i, v_{i+1}).$$

Einen Weg P_{st}^* von G bezeichnet man als *kürzesten Weg* von Knoten s nach Knoten t , falls in G kein anderer Weg P_{st} von s nach t mit $c(P_{ij}) < c(P_{ij}^*)$ existiert.

KÜRZESTE WEGE PROBLEM

Gegeben: Ein Graph $G = (V, E)$, Gewichte $c : E \rightarrow \mathbb{R}$ und Knoten $s \in V$.

Gesucht: Ein kürzester Weg von s zu allen Knoten in G .

KÜRZESTER ALLER WEGE PROBLEM

Gegeben: Ein Graph $G = (V, E)$, Gewichte $c : E \rightarrow \mathbb{R}$.

Gesucht: Ein kürzester Weg zwischen allen Knotenpaaren in G .

Wir bezeichnen mit $d(u, v)$ den kürzesten Weg zwischen zwei Knoten u und v in G bezüglich der Gewichtungsfunktion c . Falls in G kein kürzester Weg von u zu einem Knoten v existiert, dann ist G nicht zusammenhängend, und wir setzen $d(u, v) = \infty$. Weiterhin ist $d(v, v) = 0$ für alle $v \in V$.

4.3.3 Grundlegende Lösungsprinzipien

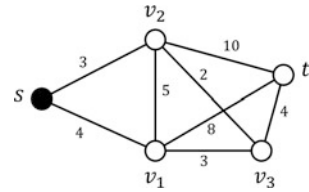
Die Berechnung des kürzesten Weges bei positiven Kantengewichten basiert auf einer einfachen Beobachtung. Die kürzesten Teilstrecken zwischen Knoten in einem Pfad bilden die kürzeste Strecke auf diesem Pfad. Genauer, seien P_{st} ein kürzester Weg von s nach t und v ein beliebiger Knoten aus dem Weg P_{st} . Dann ist der Teilweg P_{sv} von s nach v ein kürzester s - v -Weg und auch der Teilweg P_{vt} ist ein kürzester v - t -Weg. Andernfalls könnte beispielsweise ein kürzerer s - v -Weg P'_{sv} dazu verwendet werden, um einen kürzeren s - t -Weg zu erhalten. Damit ist ein kürzester Weg von s nach w höchstens so lang wie ein beliebiger Weg von s nach w . Für den ausgezeichneten Knoten $s \in V$ gilt somit

$$d(s, w) \leq d(s, v) + c(v, w), \quad \text{für alle } (v, w) \in E.$$

Diese Ungleichung ist der Schlüssel zur Konstruktion eines Algorithmus zur Berechnung der kürzesten Wege in einem Graphen.

Das bekannteste Lösungsverfahren für das *KÜRZESTE WEGE PROBLEM* ist der Algorithmus von Dijkstra. Im Jahre 1959 veröffentlichte der Niederländer Edsger W. Dijkstra (1930–2002) in einem dreiseitigen Artikel dieses bedeutende Verfahren.

Abb. 4.6 Berechnung kürzester Wege mittels Dijkstra-Algorithmus



Der Dijkstra-Algorithmus gehört zum Algorithmenmuster des Greedy-Verfahrens¹. Er berechnet einen kürzesten Weg $d(s, v)$ von dem gegebenen Startknoten s zu allen Knoten $v \in V$. Zur Vereinfachung der folgenden Notation schreiben wir $d(v) = d(s, v)$. Die Kantengewichte des Graphen dürfen dabei nicht negativ sein.

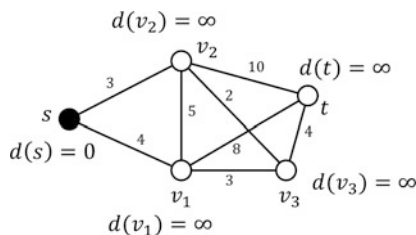
Prinzip des Dijkstra-Algorithmus

1. Initialisierung der Distanz $d(v)$ für alle Knoten v außer Startknoten s mit ∞ und $d(s) = 0$
2. Solange es noch unbesuchte Knoten gibt:
 - (a) Auswahl eines unbesuchten Knotens v mit minimaler Distanz $d(v)$
 - (b) Markierung des Knotens v als besucht
 - (c) Berechnung für alle unbesuchten Nachbarknoten w des Knotens v die Summe des jeweiligen Kantengewichtes $c(v, w)$ und der Distanz $d(v)$ im aktuellen Knoten
 - (d) Falls dieser Wert für einen Knoten kleiner ist als die dort gespeicherte Distanz $d(w)$, erfolgt die Aktualisierung des Gewichtes $d(w)$ und die Speicherung des aktuellen Knotens als Vorgänger

Die Menge der unbesuchten Knoten werden wir in der Menge W speichern. Am Anfang entspricht W der Menge aller Knoten.

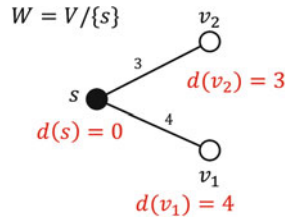
Beispiel 4.3 Wir berechnen den kürzesten Weg des Graphen in Abb. 4.6. Die einzelnen Schritte sehen wie folgt aus:

1. Initialisierung aller Distanzen außer dem Startknoten s mit ∞ und $d(s) = 0$:

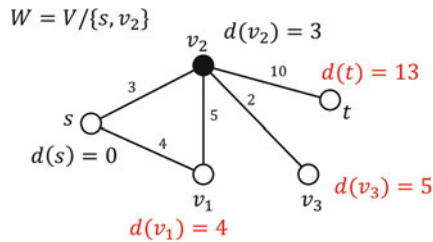


¹ Siehe Band *Grundlagen*, Kapitel Entwurf von Algorithmen.

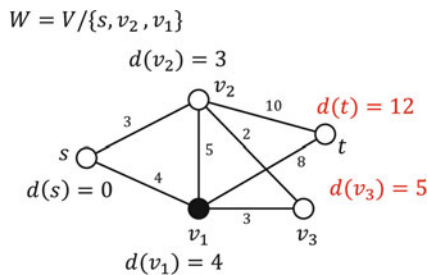
2. Bestimmung der Distanz von Anfangsknoten s zu den Nachbarknoten und Entfernung von s aus der Menge W .



3. Auswählen des Nachbarknotens mit kleinstem Abstand von s , hier der Knoten v_2 , und Entfernung von v_2 aus der Menge W .



4. Auswählen des Nachbarknotens mit kleinstem Abstand von v_2 , hier der Knoten v_1 , und Entfernung von v_1 aus der Menge W .



5. Auswählen des Nachbarknotens mit kleinstem Abstand von v_1 , hier der Knoten v_3 , und Entfernung von v_3 aus der Menge W .

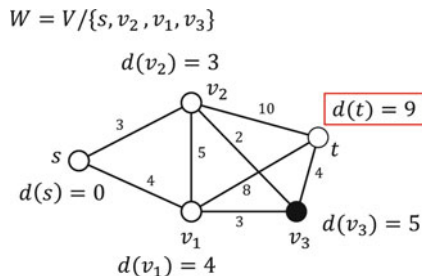
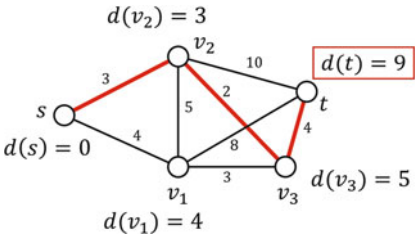


Abb. 4.7 Kürzester Weg von Knoten s zu Knoten t



Das Iterationsende ist erreicht und die kürzesten Entfernungen von s zu allen Nachbarknoten sind berechnet.

Damit erhalten wir die folgenden Iterationen des Dijkstra-Algorithmus:

Iteration	W	$d(v_1)$	$d(v_2)$	$d(v_3)$	$d(t)$
1	$V/\{s\}$	4	3	∞	∞
2	$V/\{s, v_2\}$	4	3	5	13
3	$V/\{s, v_2, v_1\}$	4	3	5	12
4	$V/\{s, v_2, v_1, v_3\}$	4	3	5	9

Über die Pfadrekonstruktion vom Zielknoten t zum Startknoten s erhalten wir $t \rightarrow v_3 \rightarrow v_2 \rightarrow s$, dargestellt in Abb. 4.7.

4.3.4 Algorithmus und Implementierung

Der Dijkstra-Algorithmus gehört zu den Greedy-Algorithmen, da in jedem Schritt der Knoten mit der geringsten Entfernung ausgewählt wird. Anders als andere Greedy-Algorithmen berechnet der Dijkstra-Algorithmus jedoch stets die optimale Lösung. Aus den obigen Überlegungen ergibt sich der folgende Pseudocode:

Algorithmus 3 DIJKSTRA-ALGORITHMUS

Input: Gerichteter Graph $G = (V, E)$, Kantengewichte $c : E \rightarrow \mathbb{R}^+$, Startknoten $s \in V$

Output: Entfernungen $d(v)$, $v \in V$ vom Startknoten s

Komplexität: $O(n^2)$

- 1: **for** $v \in V$ **do**
- 2: $d(v) = \infty$
- 3: $d(s) = 0$
- 4: $W = V$
- 5: **for** $i = 1$ **to** $|V|$ **do**
- 6: Finde einen Knoten $v \in W$ mit $d(v)$ minimal
- 7: $W = W \setminus \{v\}$
- 8: **for** $w \in N_G(v) \cap W$ **do**
- 9: **if** $d(v) + c(v, w) < d(w)$ **then**
- 10: $d(w) = d(v) + c(v, w)$

Allgemeine Erklärung

In Zeile 1–4 werden die Distanz d und die Knotenmenge W initialisiert. In Zeile 5–7 wird v aus der Menge W bestimmt, das bezüglich des Distanzwertes d minimal ist. In Zeile 8–10 wird für alle noch nicht besuchten Nachbarknoten w geprüft, ob eine Verbesserung des kürzesten Weges über den Knoten w möglich ist.

Aufwandsabschätzung

Die Laufzeit dieses Algorithmus ist durch die zwei `for`-Schleifen über die Knotenmenge $O(n^2)$.

Die Implementierung in Java erfolgt als neue Methode in der Klasse `Graph`. In einer dynamischen Datenstruktur W speichern wir die Menge der unbesuchten Knoten. Durch die Methode `getNachbarn(v)` aus der Klasse `Graph` werden dann die jeweiligen Nachbarknoten von v bestimmt. Die Methode `getMinimum(W, d)` bestimmt das minimale Element der Menge W bezüglich der Werte in d . Im Vektor `vV` wird für jeden Knoten der aktuelle Vorgängerknoten im zugehörigen kürzesten Weg abgespeichert, um damit diesen anschließend zu bestimmen.

```
public double[] Dijkstra(int s, int vV[])
{
    double d[] = new double[n];
    Vector<Integer> W = new Vector<Integer>();
    for(int i=0; i<n; i++)
    {
        d[i] = Double.MAX_VALUE;
        W.addElement(i);
    }
    d[s] = 0;

    for(int i=0; i<n; i++)
    {
        // Berechnung des kleinsten Elementes v in W
        int v = (int) W.elementAt(getMinimum(W, d));
        W.removeElementAt(getMinimum(W, d));

        Vector<Integer> NG = getNachbarn(v);
        for(int j=0; j<NG.size(); j++)
        {
            int w = (int) NG.elementAt(j);
            if (d[v] + A[v][w] < d[w])
            {
                d[w] = d[v] + A[v][w];
                vV[w] = v;
            }
        }
    }
    return d;
}
```

Die Rekonstruktion des optimalen Pfades erfolgt mit dem Array `vV` durch die folgende Methode:

```
public Vector<Integer> berechnePfad(int vV[], int s)
{
    // Berechnung des Pfades
    Vector<Integer> v_pfad = new Vector<Integer>();
    int v = n-1;
    while(true)
    {
        if (vV[v] == s)
        {
            v_pfad.add(v);
            v_pfad.add(vV[v]);
            break;
        }
        else
            v_pfad.add(v);

        v = vV[v];
    }
    return v_pfad;
}
```

4.3.5 Anwendungen

Wir betrachten einige Anwendungen der Dijkstra-Verfahrens.

Routingverfahren Der Dijkstra-Algorithmus wird im Internet als Routingalgorithmus im Optimized-Link-State-Routingprotokoll (OLSR-Protokoll) verwendet. Im Bereich der Telekommunikation versteht man unter Routing die Bestimmung der Pfade der Nachrichtenströme in Rechnernetzen. Das OLSR-Protokoll ist eine an die WLAN-Netze angepasste Version des Link-State-Routings, das sich selbstständig aufbaut und konfiguriert. Dieses Netzwerkprotokoll wird von Routern benutzt, um eine Datenbank mit Topologieinformationen aufzubauen.

Das Link-State-Routingprotokoll wird bei vielen Veränderungen in der Routingtabelle verwendet, bei denen die Routingtabelle in gewissen Zeitabständen aktualisiert werden muss. Beim Link-State-Routing werden bei Änderungen im Netzwerk sogenannte LSA (Link-State-Announcements/Advertisements) an alle benachbarten Router geschickt. In der Topologiedatenbank jedes Routers wird auf Basis der empfangenen LSA die gesamte Topologie des Netzwerks generiert. Im Link-State-Algorithmus erfolgt die Umsetzung des Dijkstra-Algorithmus.

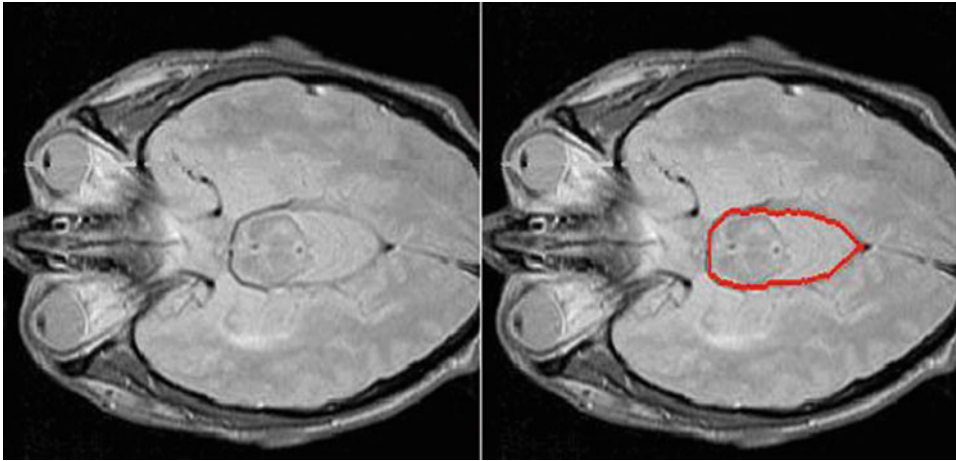


Abb. 4.8 Live-Wire-Verfahren zur Segmentierung medizinischer Bilder mit eingezeichnetem Pfad (Bildmitte)

Segmentierung medizinischer Bilder Die Segmentierung von medizinischen Bildern wird für die rechnergestützte ärztliche Diagnostik und Therapie eingesetzt. Diese Verfahren sind die Grundlage für eine weitergehende Analyse, Vermessung und 3D-Visualisierung von medizinischen Bildobjekten. Beispielsweise muss zunächst ein Tumor segmentiert werden, bevor eine Analyse seiner Eigenschaften (z. B. Form, Volumen) vorgenommen wird. Das Ziel der Segmentierung medizinischer Bilder ist die Abgrenzung von medizinisch relevanten Bildobjekten wie Gewebe, Tumore, Gefäßsysteme von gesunden Strukturen.

Das Live-Wire-Verfahren ist ein bedeutendes Standardverfahren zur Segmentierung einzelner Bildobjekte (siehe Abb. 4.8). Bei diesem Verfahren wird zwischen einem interaktiv markierten Konturpunkt und der aktuellen Mausposition automatisch eine Verbindung entlang der Bildkante berechnet. Bei Bewegungen der Maus wird die berechnete Verbindung der aktuellen Position angepasst.

Diese Art der Visualisierungstechnik hat dem Live-Wire-Verfahren seinen Namen gegeben, da sich durch schnelle Bewegungen der Maus die optimale Kontur zwischen dem Startpunkt und der sich veränderten Mausposition ändert und die damit dargestellte Folge von optimalen Konturen wie ein scheinbar „lebender Draht“ erscheint. Das Ergebnis ist ein Optimierungsproblem zur Berechnung einer kostenoptimalen Verbindung zwischen zwei Konturpunkten, die entlang der Objektkontur verlaufen soll. Den genauen Algorithmus zur Segmentierung stellen wir im Kapitel Bildverarbeitung vor.

Topologische Indizes Ein topologischer Index ist ein numerischer Wert, der aus der Struktur eines chemischen Moleküls abgeleitet wird. Diese spezielle Zahl wird verwendet, um chemische Eigenschaften und biologische Aktivitäten mit der Molekülstruktur zu korrelieren. Für die Ermittlung des Index wird nur die Zusammenhangskomponenten-

te des Moleküls benötigt. Einige topologische Indizes verwenden gewichtete Distanzen zwischen den Atomen eines Moleküls. Die Distanz ist dabei die Anzahl der Bindungen zwischen den Atomen. In Fällen mehrerer möglicher Wege wird der kürzeste Weg dazu gewählt. Mit diesen Indizes können gewisse Eigenschaft-Struktur-Beziehungen aufgestellt werden.

4.4 Gierige Suche

4.4.1 Einführendes Beispiel

Bei der Breiten- und Tiefensuche hängt die Suchstrategie nur von der vorgegebenen Struktur der Suchbäume ab. Bei der informierten Suche werden zusätzliche Informationen verwendet, um zu entscheiden, welcher Pfad durch den Suchbaum weiter verfolgt wird. Mithilfe einer Kostenschätzfunktion wird dann die vielversprechendste Alternative zuerst untersucht.

Wir betrachten wieder das bereits vorgestellte Routenplanungsproblem in der Region um Tuttlingen aus Abb. 4.9. Für die folgenden beiden Algorithmen der gierigen Suche und

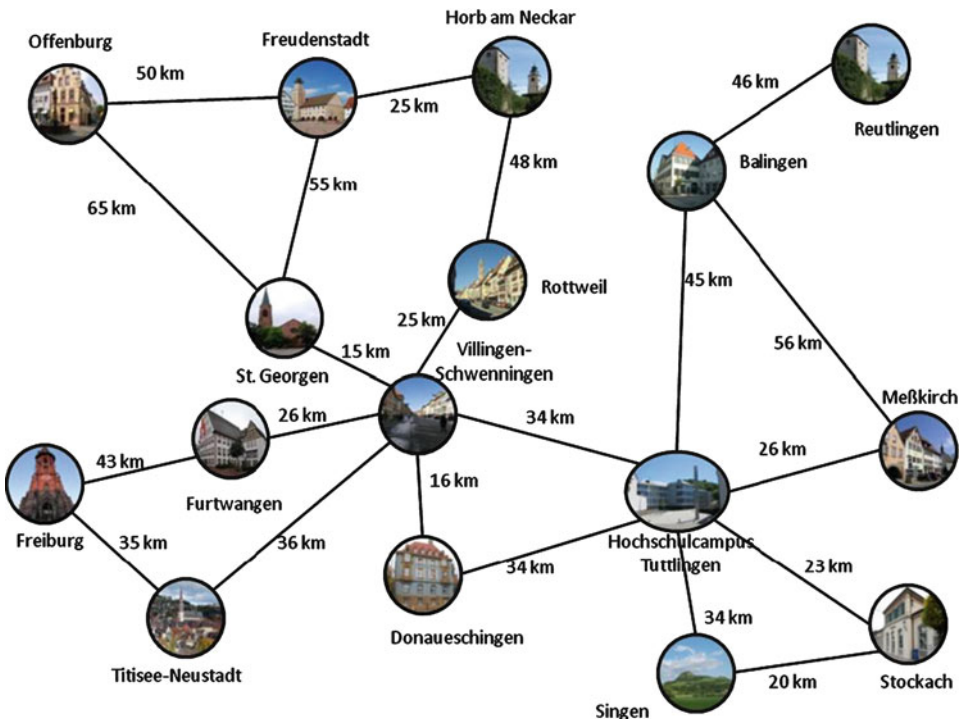


Abb. 4.9 Routenplanungsproblem in der Region um Tuttlingen

des A*-Verfahrens verwenden wir eine Kostenschätzfunktion in Form der Luftlinienentfernung zwischen den einzelnen Städten. Die Luftlinienentfernungen zum Ziel Tuttlingen sind dabei wie folgt gegeben:

Stadt	Entfernung (km)	Stadt	Entfernung (km)
Balingen	37	Reutlingen	68
Donaueschingen	25	Rottweil	27
Freiburg	78	Singen	27
Freudenstadt	65	St. Georgen	43
Furtwangen	48	Stockach	22
Horb am Neckar	56	Titisee-Neustadt	48
Meßkirch	25	Tuttlingen	0
Offenburg	93	Villingen-Schwenningen	25

Die Aufgabe besteht darin, den kürzesten Weg von einem Startknoten nach Tuttlingen über eine Greedy-Suche zu bestimmen. Bei dieser Suchtechnik wird die nächste Stadt so bestimmt, dass sie zum Zeitpunkt der Wahl die kleinstmögliche Luftlinienentfernung zum Ziel besitzt.

4.4.2 Problemstellung

Die gierige Suche gehört zu den Algorithmenmustern der Greedy-Algorithmen. Bei diesem Suchverfahren wird der Folgezustand in jedem Zeitpunkt so bestimmt, dass er zum Zeitpunkt der Wahl den größten Gewinn bezüglich einer gegebenen Bewertungsfunktion liefert. Greedy-Algorithmen sind damit oft sehr effizient, lösen viele Probleme aber nicht optimal.

Die gierige Suche bestimmt einen Suchbaum beginnend bei einem Startknoten s . Die Aufgabe besteht darin, den Zielknoten t im zugrunde liegenden Graphen $G = (V, E)$ zu finden. Gegeben ist hierzu die Kostenschätzfunktion h , die den geschätzten Abstand zu einem Zielknoten t angibt.

GIERIGE SUCHE

Gegeben: Graph $G = (V, E)$, Startknoten s , Zielknoten t , Kostenschätzfunktion $h : V \rightarrow \mathbb{R}$.

Gesucht: Pfad p vom Startknoten s zum Zielknoten t .

4.4.3 Grundlegende Lösungsprinzipien

Wir beginnen die Suche, indem wir unter den vorhandenen Nachbarknoten vom Startknoten s den Knoten v auswählen, der einen minimalen h -Wert besitzt. Anschließend

expandieren wir den Suchbaum mit allen Nachbarknoten von v . Dann bestimmen wir wieder unter allen vorhandenen Nachbarknoten von v einen Knoten w , der einen minimalen h -Wert besitzt. Dieses Prinzip der Suche wird so lange fortgeführt, bis wir den Zielknoten t gefunden haben. Die Art und Weise der Suche ist weder optimal noch vollständig, aber sehr effizient.

Die gierige Suche eignet sich zur Lösung des vereinfachten Problems der Routenplanung mithilfe der Luftlinie. Statt der optimalen Route suchen wir nun von jedem Knoten eine Route mit minimaler Luftlinienentfernung zum Ziel. Damit ist die Kostenschätzfunktion $h(v)$ die Luftlinienentfernung von Stadt v zum Ziel t .

Prinzip der gierigen Suche

1. Bestimmung aller Nachbarknoten $N_G(s)$ von Startknoten s
2. Speicherung der Menge $N_G(s)$ in einer Liste W
3. Wiederholung der folgenden Schritte:
 - (a) Bestimmung des Knotens $v \in W$ mit minimalem h -Wert
 - (b) Falls v der Zielknoten t ist $\Rightarrow$ kürzester Weg gefunden
 - (c) Hinzufügen der Elemente von $N_G(v)$ in die Liste W
 - (d) Entfernung des Knotens v aus W

Zur Speicherung des Weges werden die Knoten rückwärts im Suchbaum verkettet, sodass jeder Knoten einen Verweis auf seinen Vorgänger besitzt. Dann kann der Pfad rückwärts ausgehend vom Zielknoten t rekonstruiert werden.

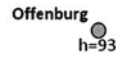
In Abb. 4.10 ist der entstehende Suchbaum der gierigen Suche dargestellt, der den kürzesten Weg vom Startknoten Offenburg nach Tuttlingen bestimmt. Dieser Weg führt von Offenburg nach St. Georgen über Villigen-Schwenningen nach Tuttlingen, mit einer Gesamtlänge von 114 km.

Mit dieser Suchstrategie ist es nicht möglich immer die optimale Lösung zu finden. Wählt man beispielsweise eine Nachbarstadt v_1 von s , die zwar etwas näher an s liegt als eine andere Stadt v_2 , deren Entfernung von v_1 nach t aber deutlich größer ist als die von v_2 nach t , so ist die gefundene Route nicht optimal. Die Heuristik schaut nur „gierig“ zum Ziel, anstatt auch die bis zum aktuellen Knoten schon zurückgelegte Strecke zu berücksichtigen. Mit der im nachfolgenden Abschnitt vorgestellten A*-Suche werden wir dieses Verfahren noch verbessern.

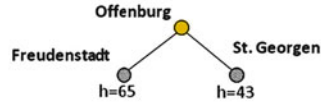
4.4.4 Algorithmus und Implementierung

Wir formulieren das Prinzip der gierigen Suche in Form eines Pseudocodes. Den Pfad p vom Startknoten s zum Zielknoten t können wir alternativ auch als Suchbaum B darstellen. Mit der Methode ADD fügen wir Knoten in den Suchbaum hinzu.

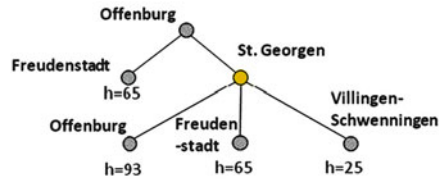
Der Ausgangszustand



Expandierung nach Offenburg



Expandierung nach St. Georgen



Expandierung nach Villingen-Schwenningen

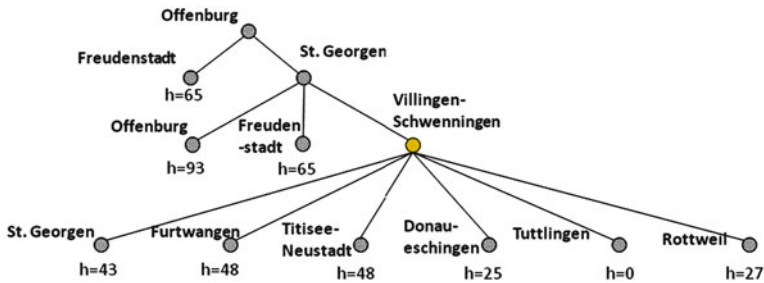


Abb. 4.10 Suchbaum der gierigen Suche eines Routenplanungsproblems

Algorithmus 4 GIERIGE SUCHE

Input: Graph $G = (V, E)$, Startknoten s , Zielknoten t , Kostenschätzfunktion $h : V \rightarrow \mathbb{R}$

Output: Suchbaum B

```

1:  $B.ADD(s)$ 
2:  $B.ADD(N_G(s))$ 
3:  $W = N_G(s)$ 
4: while true do
5:   Bestimme  $v$  mit  $h(v) = \min\{h(u) \mid u \in W\}$ 
6:   if  $v = t$  then
7:     return  $B$ 
8:    $W = W \cup N_G(v)$ 
9:    $W = W \setminus \{v\}$ 
10:   $B.ADD(N_G(v))$ 
  
```

Allgemeine Erklärung

In Zeile 1–3 initialisieren wir den Suchbaum B und die Menge der aktiven Knoten W . Anschließend führen wir in Zeile 5–10 die folgenden Schritte so lange aus, bis wir auf den Zielknoten t stoßen. Wir bestimmen aus der Menge W einen Knoten v , der einen minimalen Wert bezüglich der Kostenschätzfunktion h besitzt. Die Menge der Nachbarknoten von v wird dann zur Menge W hinzugefügt (Zeile 8) und der Knoten v wird aus W entfernt (Zeile 9). Zum Suchbaum W fügen wir zuletzt alle Nachbarknoten von v hinzu (Zeile 10).

Wir erweitern und verbessern die gierige Suche mit dem nachfolgenden A*-Algorithmus, bei dem wir dann auch eine Implementierung in Java angeben.

4.4.5 Anwendungen

Wir stellen eine ganze Reihe von Anwendungen dieser Suchtechnik im nachfolgenden Abschnitt bei der A*-Suche vor.

4.5 A*-Suche

4.5.1 Einführendes Beispiel

Wir betrachten wieder das bekannte Routenplanungsproblem in der Region um Tuttlingen aus Abb. 4.9 mit den Luftlinienentfernungen zum Ziel Tuttlingen:

Stadt	Entfernung (km)	Stadt	Entfernung (km)
Balingen	37	Reutlingen	68
Donaueschingen	25	Rottweil	27
Freiburg	78	Singen	27
Freudenstadt	65	St. Georgen	43
Furtwangen	48	Stockach	22
Horb am Neckar	56	Titisee-Neustadt	48
Meßkirch	25	Tuttlingen	0
Offenburg	93	Villingen-Schwenningen	25

In diesem Abschnitt verbessern wir die Greedy-Suche, indem wir auch die bereits zurückgelegten Entfernungen in der heuristischen Kostenfunktion berücksichtigen.

4.5.2 Problemstellung

Der A*-Algorithmus ist eine Verbesserung der Greedy-Suche, sodass wir unter gewissen Voraussetzungen eine optimale Lösung erhalten. Ähnlich wie der Dijkstra-Algorithmus

dient der A*-Algorithmus zur Bestimmung des kürzesten Weges zwischen zwei Knoten s und t in einem Graphen $G = (V, E)$ mit positiven Kantengewichten. Im A*-Algorithmus wird eine Heuristik zur Beschleunigung der Lösungsfindung verwendet, sodass die vielversprechendsten Knoten zuerst untersucht werden. Es werden die folgenden Kostenwerte für einen Knoten v benötigt:

- $g(v)$, die Kosten vom Startknoten s zum Knoten v ,
- $h(v)$, die geschätzten Kosten von Knoten v zum Zielknoten t ,
- $f(v)$, die Kosten vom Startknoten s zum Zielknoten t über v :

$$f(v) = g(v) + h(v).$$

Die Funktion h stellt die Heuristik des A*-Algorithmus dar. Bei einer Routenplanung wird beispielsweise wieder die Entfernung zwischen zwei Orten in Luftlinie gemessen.

Definition 4.4 Die heuristische Kostenschätzfunktion $h(v)$ heißt *zulässig*, falls diese Funktion die tatsächlichen Kosten vom Zustand v zum Ziel nie überschätzt.

Der Dijkstra-Algorithmus ist eine spezielle Form des A*-Algorithmus, die keine Heuristik verwendet, also $h = 0$ und $f = g$. Der A*-Algorithmus ist effizienter, wenn eine zulässige Heuristik genutzt werden kann. Falls das Ziel bei der Suche noch nicht bekannt ist, beispielsweise bei einer Umkreissuche, ist der Dijkstra-Algorithmus effizienter.

A*-SUCHE

Gegeben: Graph $G = (V, E)$, Startknoten s , Zielknoten t , Kostenschätzfunktion $g : V \rightarrow \mathbb{R}$,
zulässige Kostenschätzfunktion $h : V \rightarrow \mathbb{R}$.

Gesucht: Pfad p vom Startknoten s zum Zielknoten t .

4.5.3 Grundlegende Lösungsprinzipien

Wir zeigen nun eine wesentliche Eigenschaft der A*-Suche, nämlich die Optimalität dieser Suche bei Vorhandensein einer zulässigen Heuristik.

Satz 4.1 *Der A*-Algorithmus ist für eine zulässige Heuristik h optimal.*

Beweis Wir wählen aus der Menge W immer einen Knoten v , sodass kein anderer Knoten $w \in W$ eine bessere heuristische Bewertung bezüglich der Funktion f besitzt, also $f(v) \leq f(w)$ für alle $w \in W$. Aus der Voraussetzung einer zulässigen Heuristik h folgt für einen Lösungsknoten v der Wert $h(v) = 0$, also

$$g(v) = g(v) + h(v) = f(v) \leq f(w) = g(w) + h(w).$$

Für einen beliebigen später gefundenen Lösungsknoten v' mit $h(v') = 0$ ist $f(v') = g(v') + h(v') = g(v')$. Damit erhalten wir die Zulässigkeit der Heuristik in Form der

Ungleichung $f(w) \leq f(v') = g(v')$, d. h., dass die Kosten vom Knoten w zum Knoten v' nicht überschätzt werden. Damit ergibt sich die Ungleichung $g(v) \leq f(w) \leq g(v')$, und somit ist die zuerst gefundene Lösung v optimal. $\square$

Die heuristische Suche bildet zusammen mit einer Bewertungsfunktion $f(v) = g(v) + h(v)$ mit der zulässigen Heuristik h die Grundlage für den A*-Algorithmus. Aufgrund des obigen Satzes ist dieser Algorithmus vollständig und optimal.

Prinzip des A*-Algorithmus

1. Bestimmung aller Nachbarknoten $N_G(s)$ von Startknoten s
2. Speicherung der Menge $N_G(s)$ in einer Liste W
3. Berechnung des f -Wertes für jeden Knoten in W
4. Wiederholung der folgenden Schritte:
 - (a) Bestimmung des Knotens $v \in W$ mit minimalem f -Wert
 - (b) Falls v der Zielknoten t ist $\Rightarrow$ kürzester Weg ist gefunden
 - (c) Hinzufügen der Elemente von $N_G(v)$ in die Liste W
 - (d) Berechnung des f -Wertes für jeden Knoten in $N_G(v)$
 - (e) Entfernung des Knotens v aus W

In Abb. 4.11 ist der entstehende Suchbaum der A*-Suche dargestellt, der den kürzesten Weg vom Startknoten Offenburg nach Tuttlingen mit der A*-Suche bestimmt. Dieser Weg führt von Offenburg nach St. Georgen über Villigen-Schwenningen nach Tuttlingen mit einer Gesamtlänge von 114 km.

4.5.4 Algorithmus und Implementierung

Der Pseudocode des A*-Algorithmus ist bis auf die Zeile 5 identisch mit der Greedy-Suche, die nun aus der Summe von zwei Kostenschätzfunktionen besteht.

Algorithmus 5 A*-ALGORITHMUS

Input: Graph $G = (V, E)$, Startknoten s , Zielknoten t , Kostenschätzfunktion $g : V \rightarrow \mathbb{R}$, zulässige heuristische Kostenschätzfunktion $h : V \rightarrow \mathbb{R}$

Output: Suchbaum B

```

1:  $B.ADD(s)$ 
2:  $B.ADD(N_G(s))$ 
3:  $W = N_G(s)$ 
4: while true do
5:   Bestimme  $v$  mit  $f(v) = \min\{f(u) = g(u) + h(u) \mid u \in W\}$ 
6:   if  $v = t$  then
7:     return  $B$ 
8:    $W = W \cup N_G(v)$ 
9:    $W = W \setminus \{v\}$ 
10:   $B.ADD(N_G(v))$ 

```

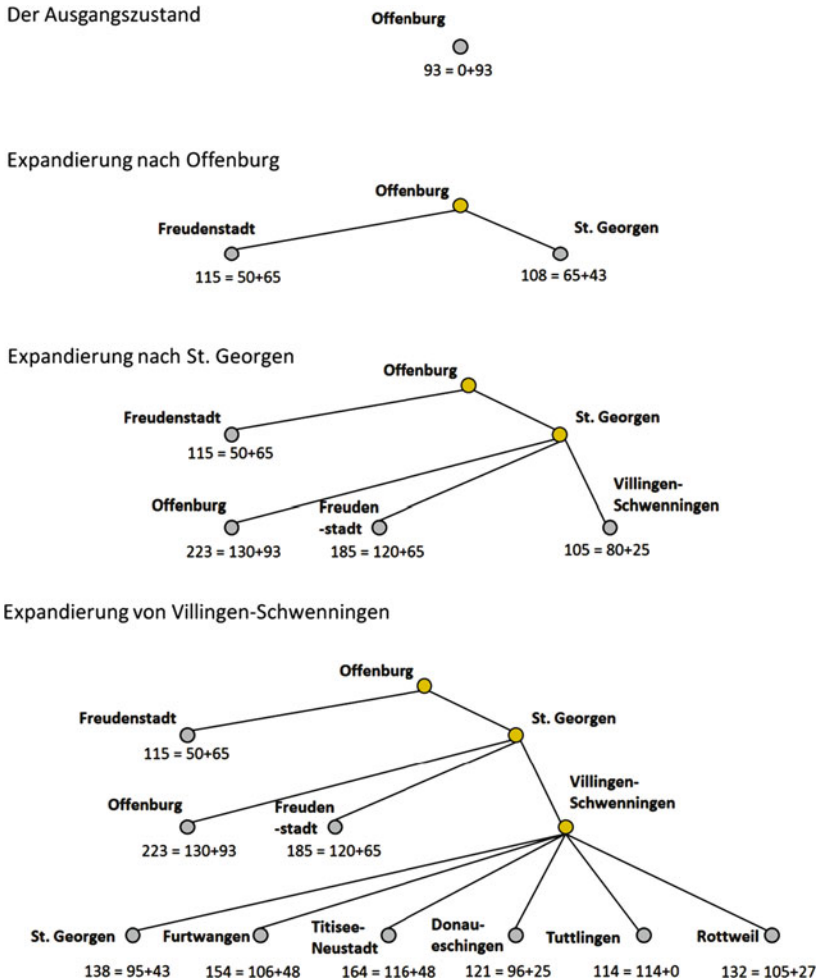


Abb. 4.11 Suchbaum des A*-Algorithmus eines Routenplanungsproblems

Allgemeine Erklärung

In Zeile 1–3 initialisieren wir den Suchbaum B und die Menge der aktiven Knoten W . Anschließend führen wir in Zeile 5–10 die folgenden Schritte so lange aus, bis wir auf den Zielknoten t stoßen. Wir bestimmen aus der Menge W einen Knoten v , der einen minimalen Wert bezüglich der Kostenschätzfunktion f besitzt. Die Menge der Nachbarknoten von v wird dann zur Menge W hinzugefügt (Zeile 8) und der Knoten v wird aus W entfernt (Zeile 9). Zum Suchbaum W fügen wir zuletzt alle Nachbarknoten von v hinzu (Zeile 10).

Aufwandsabschätzung

Die Laufzeit des A*-Algorithmus beträgt in einer einfachen Implementierung $O(n^2)$. Bei einer Implementierung mit speziellen Datenstrukturen kann die Laufzeit auf $O(n \log n + e)$ reduziert werden; e ist die Anzahl der besuchten Kanten.

Für die Implementierung nutzen wir wieder die bereits vorgestellte Klasse `Graph` zur Definition eines Graphen. Die folgende Methode `Astern(int s, int t, double h[])` berechnet den kürzesten Weg vom Startknoten s zum Zielknoten t mit einer gegebenen zulässigen Heuristikfunktion h . Die Methode `getfWert` bestimmt den f -Wert von allen Nachbarknoten von v aus dem bisherigen Wert g mit der Heuristikfunktion h :

```
private Vector<Double> getfWert(int v, double g, double h[])
{
    Vector<Double> f_wert = new Vector<Double>();
    for(int i=0; i<n; i++)
    {
        if (A[v][i] > 0)
        {
            double wert = g + A[v][i] + h[i];
            f_wert.addElement(wert);
        }
    }
    return f_wert;
}
```

Die Implementierung des A*-Algorithmus sieht dann wie folgt aus:

```
public Vector<Integer> Astern(int s, int t, double h[])
{
    // --- 1. Initialisierung
    // Pfad von s nach t mit Wegkosten
    Vector<Integer> d = new Vector<Integer>();
    // Menge W der aktiven Knoten mit Wert f_wert = f_wert_i + A(i,j) + h(j)
    Vector<Integer> W = getNachbarn(s);
    Vector<Double> f_wert = getfWert(s, 0.0, h);
    ausgabeVektor(W, f_wert, true);

    while(true)
    {
        // --- 2. Bestimmung des Knotens in W mit minimalem f-Wert
        int idx = getMinimum(f_wert); // Index des Knoten bzgl. W
        int v = (int) W.elementAt(idx);
        System.out.printf(" Minimum bei Knoten %d mit Wert %f\n", v, f_wert.elementAt(idx));

        // --- 3. Abbruchkriterium falls Zielknoten erreicht
        if (v == t)
            break;
    }
}
```

```

// --- 4. Hinzufügen neuer Nachbarn
Vector<Integer> NG      = getNachbarn(v);
Vector<Double> f_NG_wert = getfWert(v, (double) (f_wert.elementAt(idx)) - h[v], h);
W.addAll(NG);
f_wert.addAll(f_NG_wert);

// --- 5. Löschen des ausgewählten Knotens
W.removeElementAt(idx);
f_wert.removeElementAt(idx);
ausgabeVektor(W, f_wert, true);
}
return d;
}

```

4.5.5 Anwendungen

Wir betrachten einige zentrale Anwendungsbereiche der A*-Suche.

Bahnplanung von Industrierobotern Ein Industrieroboter ist ein Roboter, der aus einer Menge von Gelenken mit einem Greifarm besteht. Unter einem Bahnplanungsproblem von Industrierobotern versteht man die Beschreibung der Bewegungen der einzelnen Gelenke, sodass der Greifarm sich von einem Start- zu einem Zielpunkt, unter Berücksichtigung der darin befindlichen Hindernisse gelenkt wird. Diese kontinuierliche Aufgabenstellung wird in ein zugehöriges diskretes Graphenproblem umgewandelt, das mit der A*-Suche gelöst wird. Die Bahnplanung besteht dann aus den folgenden Schritten:

1. Definition des Konfigurationsraums:

Der Konfigurationsraum des Roboters beschreibt die Bewegungsmöglichkeiten der einzelnen Gelenke des Roboters. Bei einem Roboter mit n Gelenken hat der Konfigurationsraum genau n Freiheitsgrade, beschrieben durch die Gelenkwinkel $\varphi_1, \dots, \varphi_n$.

2. Markierung der Hindernisse im Konfigurationsraum:

Im Konfigurationsraum werden alle Regionen als Hindernisse markiert, bei denen die Gelenke des Roboters mit diesen kollidieren. Hierzu betrachtet man der Reihe nach alle n Gelenke. Wir bestimmen dann für das i -te Gelenk des Roboters, unter allen sicheren Stellungen der Gelenke $1, \dots, i - 1$, die Stellungen, mit denen dieses Gelenk mit einem Hindernis kollidiert. Die Bahnplanung muss dann einen Weg durch den Konfigurationsraum finden, der diese Bereiche nicht durchläuft. In Abb. 4.12 ist der zweidimensionale Konfigurationsraum von einem Roboter mit zwei Gelenken mit den Winkeln φ_1 und φ_2 dargestellt.

3. Diskretisierung des Konfigurationsraums:

Die Diskretisierung überführt die kontinuierliche Bewegung des Roboters in diskrete Bewegungsschritte zwischen Knotenpunkten im zugehörigen Konfigurationsraum. Das Ergebnis ist ein Graph mit den Diskretisierungspunkten als Knoten in einer regelmäßigen Gitterstruktur (siehe Abb. 4.13). Die Knoten werden durch Kanten verbunden, wenn der Roboter sich in einem Schritt zwischen den zwei Konfigurationspunkten bewegen kann.

Abb. 4.12 Zweidimensionaler Konfigurationsraum des Roboters mit eingezeichneten Hindernissen

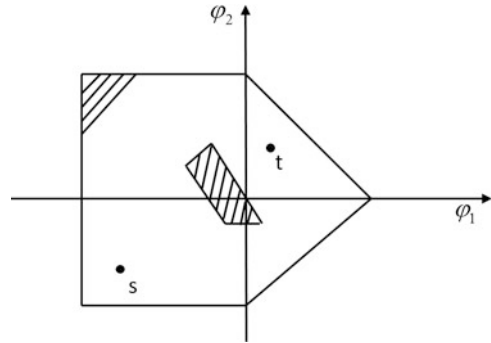
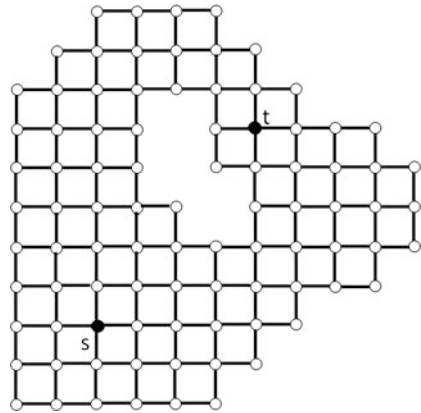


Abb. 4.13 Diskretisierung des Konfigurationsraums



4. Bestimmung des Pfades vom Start- zum Zielpunkt:

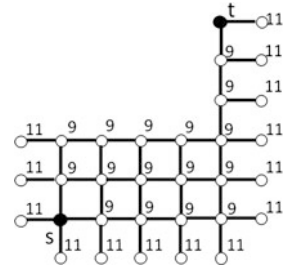
Das Bahnplanungsverfahren besteht aus der Suche des kürzesten Pfades zwischen dem Start- und dem Zielpunkt im zugehörigen Graphen des diskretisierten Konfigurationsraums mit dem A*-Algorithmus. Für die heuristische Schätzfunktion h des Restpfades zwischen zwei Punkten $s = (x_1, \dots, x_n)$ und $t = (y_1, \dots, y_n)$ kann der sogenannte Manhattan-Abstand genommen werden:

$$d(s, t) = \sum_{i=1}^n |x_i - y_i|.$$

In Abb. 4.14 ist der Pfad vom Startpunkt s zum Zielpunkt t , mit den zugehörigen berechneten Abständen mit dem A*-Algorithmus dargestellt.

Routenplanung mobiler Roboter Ein weiteres Anwendungsgebiet der Suche nach kürzesten Wegen ist die Routenplanung von mobilen Robotern. In einer bekannten Umgebung soll ein Roboter von einem Startpunkt s auf kollisionsfreien und kürzesten Wegen zu einem Zielpunkt t fahren. Eine idealisierte Version dieses Problems erhalten wir, indem man die Bewegung des Roboters auf die Ebene beschränkt. Durch Vergrößerung aller Hindernisse um die halbe Roboterbreite kann der Roboter als punktförmiges Gebilde angesehen

Abb. 4.14 Pfad vom Startpunkt s zum Zielpunkt t im diskretisierten Konfigurationsraum

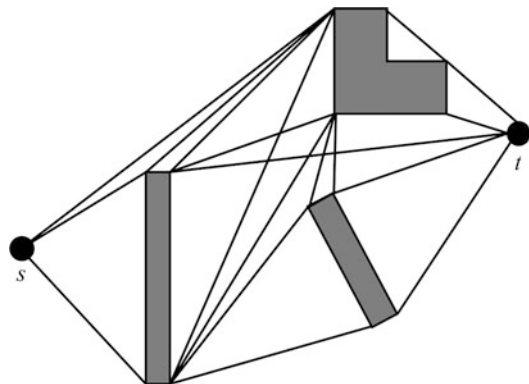


werden. Die Hindernisse der Umgebung werden durch eine Menge von Polygonen beschrieben.

Die Aufgabe besteht nun darin einen kürzesten Pfad in der Ebene von einem Startpunkt s zu einem Zielpunkt t zu finden, der die Polygone nicht schneidet. Wir betrachten dazu das Beispiel aus Abb. 4.15. Dazu wird zunächst aus der Umgebung mit den Polygonen ein Graph aufgebaut. Die Knoten des Graphen sind alle Eckpunkte der Polygone. Zwei Knoten sind dann durch eine Kante verbunden, wenn eine kollisionsfreie gerade Verbindung existiert. Der Pfad von s nach t ist dann eine Folge von Geradensegmenten zwischen den einzelnen Ecken der Hindernisse. Mithilfe des A^* -Algorithmus kann dann der kürzeste Weg in diesem berechneten Graphen bestimmt werden.

8-Puzzle Ein 8-Puzzle oder Schiebepuzzle ist ein bekanntes Spiel aus 8 Quadraten, die mit den Zahlen von 1 bis 8 durchnummeriert sind, und auf einem 3-mal-3-Quadrat angebracht sind. Mithilfe eines freien Feldes können benachbarte Quadrate vertikal oder horizontal verschoben werden. Die Aufgabe ist, durch Verschieben der Quadrate die Zahlen von 1 bis 8 aufsteigend anzuordnen. Die Lösung dieser Aufgabenstellung ist ein klassisches Suchproblem in dem Baum aller möglichen Konfigurationen. Der entstehende Suchbaum hat einen Verzweigungsfaktor von 2 bis 4, je nach Standort des freien Feldes. In Abb. 4.16 ist ein Suchbaum des 8-Puzzles der Tiefe 2 dargestellt, bei dem Kreise der Länge 2 entfernt wurden.

Abb. 4.15 Routenplanung eines mobilen Roboters in einer Umgebung zwischen zwei Knoten s und t



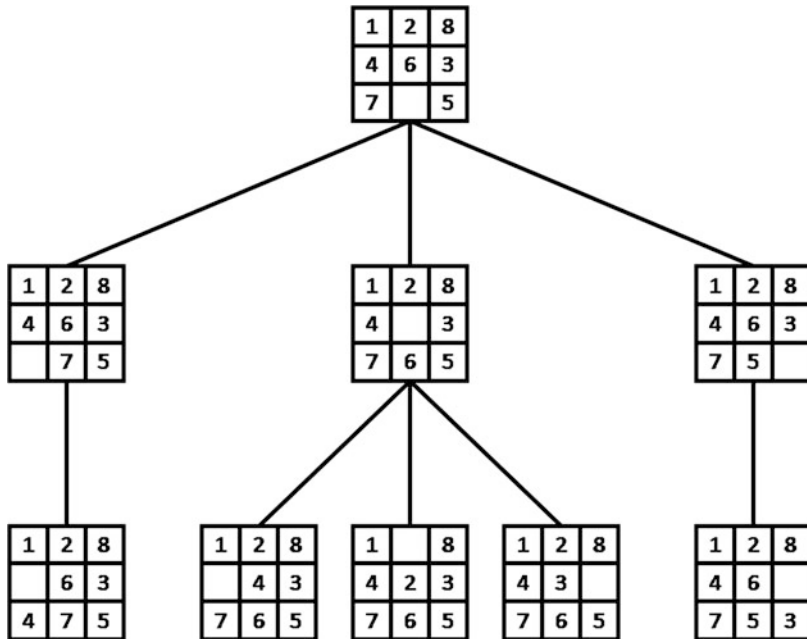


Abb. 4.16 Suchbaum des 8-Puzzles der Tiefe 2 ohne Kreise

Der A*-Algorithmus kann nun wieder verwendet werden, um das beschriebene Problem des 8-Puzzles zu lösen. Dafür gibt es zwei einfache zulässige Heuristiken:

1. Heuristik h_1 zählt die Anzahl der Quadrate, die sich nicht an der richtigen Stelle befinden.
2. Heuristik h_2 verwendet den Manhattan-Abstand, bei dem für alle Quadrate der horizontale und vertikale Abstand zum gleichen Quadrat im Zielzustand addiert werden.

Auf Grundlage einer dieser beiden Heuristiken kann nun die kürzeste Konfigurationsreihenfolge der Quadrate bestimmt werden. Dazu wird das 8-Puzzle in den Zielzustand gebracht, bei dem alle Zahlen 1 bis 8 der Reihenfolge von links oben nach rechts unten angeordnet sind.

4.6 Simulated Annealing

4.6.1 Einführendes Beispiel

Simulated Annealing ist eine Erweiterung der einfachen lokalen Suche. Lokale Suchalgorithmen werden für zahlreiche Optimierungsprobleme in Naturwissenschaft und Technik

verwendet. Gegeben sind ein Suchraum Ω und eine Zielfunktion $f : \Omega \rightarrow \mathbb{R}$. Die Aufgabe ist es, ein Element $x \in \Omega$ mit kleinstem oder größtem Wert $f(x)$ zu bestimmen. Aufgrund der hohen Laufzeit von exakten Verfahren werden hierzu Heuristiken verwendet, um eine gute zulässige Lösung zu bestimmen.

Die lokale Suche beginnt mit einer zulässigen Lösung und verändert diese dann in einer zufälligen Weise durch geringfügige lokale Veränderungen. Sollten diese Veränderungen eine Verbesserung gebracht haben, so wird diese neue Lösung übernommen.

Prinzip der lokalen Suche

1. Zufällige Wahl einer Anfangslösung
2. Zulässige Anfangslösung wird durch geringfügige zufällige Veränderungen modifiziert, beispielsweise:
 - Vertauschen bzw. Verschieben von Elementen innerhalb einer Reihenfolge
 - Umschalten einer Position eines binären Lösungsvektors von 0 auf 1, oder umgekehrt
3. Falls diese Veränderungen eine Verbesserung gebracht haben, wird diese neue Lösung übernommen.

Als Beispiele betrachten wir die bereits im 1. Band vorgestellten klassischen Problemstellungen des Rundreise- und des Rucksackproblems:

RUNDREISEPROBLEM

Gegeben: Matrix $M = (m_{ij})$ aller direkter Entfernungen zwischen einzelnen Städten, wobei $m_{ij} = \infty$, falls keine direkte Straße von i nach j existiert.

Gesucht: kostenminimale Rundreise durch alle Städte in Form einer Permutation $\pi : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ mit

$$\sum_{i=1}^{n-1} m_{\pi(i), \pi(i+1)} + m_{\pi(n), \pi(1)} \rightarrow \text{Minimum.}$$

RUCKSACKPROBLEM

Gegeben: Gegenstände mit den Nutzen $w_j \in \mathbb{N}$, Gewichte $g_j \in \mathbb{N}$ mit $j = 1, \dots, n$, Maximalgewicht G .

Gesucht: Vektor $x = (x_1, \dots, x_n) \in \{0, 1\}^n$ mit

$$\sum_{j=1}^n x_j w_j \rightarrow \text{Maximum} \quad \text{und} \quad \sum_{j=1}^n x_j g_j \leq G.$$

Im Band *Grundlagen* haben wir exakte Algorithmen basierend auf dem Algorithmenmuster dynamischen Programmierens angegeben, um diese beiden Optimierungsprobleme zu lösen. Leider sind für diese beiden Probleme nur exakte Algorithmen mit exponentiellem Aufwand bekannt.

Die Menge aller Lösungen, die durch geringfügige Veränderungen von x entstehen, bezeichnen wir als die Menge der Nachbarlösung $NB(x)$ von x . Beim Rundreiseproblem kann die Menge $NB(x)$ charakterisiert werden durch alle Lösungen, die aus x durch den Austausch von zwei Kanten gegen zwei bislang nicht in der Rundreise enthaltene Kanten entstehen. Beim Rucksackproblem beschreibt die Menge $NB(x)$ alle Lösungen, die sich von x in genau einer oder zwei Komponenten unterscheiden. Es gibt verschiedene Strategien zur Untersuchung der Nachbarschaft. Entweder wird zufällig oder systematisch die Nachbarlösung $NB(x)$ untersucht. Die Wahl einer Nachbarlösung kann dann entweder nach der „First-fit“-Strategie (die erste gefundene verbessernde Nachbarlösung wird verwendet) oder der „Best-fit“-Strategie (beste Nachbarlösung wird verwendet) erfolgen.

Reine Verbesserungsverfahren enden, sobald in einer Iteration keine verbessernde Nachbarlösung existiert bzw. gefunden wird. Die beste erhaltene Lösung stellt für die gewählte Nachbarschaftsdefinition ein lokales Optimum dar, dessen Zielfunktionswert deutlich schlechter als der eines globalen Optimums sein kann. Um ein solches lokales Optimum wieder verlassen zu können, müssen Züge erlaubt werden, die zwischenzeitlich zu Verschlechterungen des Zielfunktionswertes führen. Diese Art von Heuristiken sind die folgenden vorgestellten Verfahren:

- **Simulated Annealing** erweitert die lokale Suche, sodass auch Lösungen mit einer bestimmten Wahrscheinlichkeit (abhängig vom Ausmaß der Verschlechterung) akzeptiert werden, damit der Algorithmus nicht in einem lokalen Optimum hängen bleibt. Im Laufe des Verfahrens wird die Wahrscheinlichkeit, dass schlechtere Lösungen akzeptiert werden, immer kleiner.
- **Genetische Algorithmen** entstehen aus der lokalen Suche, wobei eine große Population an Zuständen verwaltet wird. Neue Zustände entstehen durch Mutation und Kreuzung aus der Kombination zweier Populationspaare.

4.6.2 Problemstellung

Simulated Annealing kann für eine große Anzahl völlig unterschiedlicher Optimierungsprobleme sehr gut angewandt werden.

Definition 4.5 Ein *Optimierungsproblem* ist ein 3-Tupel $(\Omega, f, <)$, hierbei sind

- Ω eine endliche Menge (*Suchraum*),
- $f : \Omega \rightarrow \mathbb{R}$ eine Abbildung (*Bewertungsfunktion*),
- $<$ eine Vergleichsrelation $\{<, >\}$ (*Bewertungsrelation*).

Die Menge aller globalen Optima ist definiert durch

$$X^* = \{x^* \in \Omega \mid f(x^*) < f(x), \forall x \in \Omega\}.$$

Beispiel 4.4

1. Rundreiseproblem ist ein Optimierungsproblem der Form $(\Omega_{\text{TSP}}, f_{\text{TSP}}, <)$ mit dem Suchraum $\Omega_{\text{TSP}} = P(V)$ der Menge aller Teilmengen der Knotenmenge V und der Bewertungsfunktion

$$f_{\text{TSP}} = \sum_{i=1}^{n-1} m_{\pi(i), \pi(i+1)} + m_{\pi(n), \pi(1)}.$$

2. Rucksackproblem ist ein Optimierungsproblem der Form $(\Omega_{\text{RP}}, f_{\text{RP}}, >)$ mit dem Suchraum $\Omega_{\text{RP}} = \{x \in \{0, 1\}^n \mid \sum_{j=1}^n x_j g_j \leq G\}$ und der Menge aller Kombinationsmöglichkeiten der Gegenstände, die in den Rucksack passen, und der Bewertungsfunktion $f_{\text{RP}} = \sum_{j=1}^n x_j w_j$.

SUCHE

Gegeben: Optimierungsproblem $(\Omega, f, <)$.

Gesucht: Element $x \in X^*$ aus der Menge der globalen Optima.

Im Folgenden betrachten wir Optimierungsprobleme zur Minimierung der Zielfunktionswerte. Das ist keine Einschränkung, da wir jedes Maximierungsproblem in ein Minimierungsproblem umwandeln können, indem wir ein negatives Vorzeichen vor die Zielfunktion stellen.

4.6.3 Grundlegende Lösungsprinzipien

Simulated Annealing arbeitet nach dem Prinzip: „Wer den höchsten Gipfel finden will, muss bereit sein, zwischendurch auch einmal abzustiegen“. Unter dem Begriff Simulated Annealing versteht man ein simuliertes langsames Abkühlen bzw. ein Erhitzen und dann langsames Abkühlen.

In der Metallurgie hängen Materialeigenschaften davon ab, ob erhitzte Metalle schnell oder langsam aus dem rotglühenden Zustand abgekühlt werden. Ein Metall ist umso härter, je besser sich eine geordnete Kristallstruktur bildet. Zu Beginn sind die Atome in einem festen Gitter eingebunden. Falls das Metall nun erhitzt wird, lösen sich die Bindungen langsam auf, und die ursprüngliche Struktur wird zerstört. Bei einem langsamen Abkühlen des Metalls suchen sich die Teilchen neue Bindungen. Beim Glühen wird das Metall nochmals erhitzt und dann langsam wieder abgekühlt, sodass sich die Kristallstruktur des Metalls besser ausbilden kann. Die Teilchen verteilen sich dabei oft regelmäßiger als vorher. Bei einem sorgsam eingestellten Prozess wird das Metall flexibler und enthält weniger Unregelmäßigkeiten.

Prinzip des Simulated Annealing

1. Zufällige Auswahl einer Nachbarlösung von einer zulässigen Startlösung x
2. Führt dieser Schritt zu einer Verbesserung des Zielfunktionswertes, so wird die bisherige Lösung x ersetzt
3. Führt dieser Schritt zu einer Verschlechterung des Zielfunktionswertes, so wird x mit einer bestimmten Wahrscheinlichkeit, die abhängig vom Ausmaß der Verschlechterung ist, ersetzt
4. Im Laufe des Verfahrens wird die Wahrscheinlichkeit, dass schlechtere Lösungen akzeptiert werden, immer kleiner

Mit diesem Verfahren soll erreicht werden, dass der Algorithmus nicht in einem lokalen Optimum „hängen bleibt“. Bei der lokalen Suche kann es sein, dass es nicht möglich ist mit den lokalen Nachbarlösungen dieses lokale Optimum zu verlassen, wenn man nicht in den Zwischenschritten schlechtere Lösungen akzeptiert.

Beispiel 4.5 Betrachten wir das Beispiel eines Handelsreisenden der genau 20 Städte besuchen soll. Gesucht ist die Reihenfolge, in der er die Städte besucht, sodass der zurückgelegte Weg möglichst kurz wird.

Wir beginnen mit irgendeiner beliebigen Reihenfolge. Nun können wir diese Tour verbessern, indem wir kleine Änderungen vornehmen. Wir könnten zufällig zwei Kanten gegen zwei bislang nicht in der Rundreise enthaltene Kanten austauschen (siehe Abb. 4.17). Nach Änderungen beider Arten kann die jeweilige Tour besser oder schlechter sein. Wenn sie besser ist, behalten wir die Änderung auf jeden Fall. Ansonsten wird sie nur mit einer bestimmten Wahrscheinlichkeit (abhängig vom Ausmaß der Verschlechterung) behalten.

Die Abb. 4.18 zeigt den Verlauf der Akzeptanzwahrscheinlichkeit und die Verbesserungsschritte der Länge der Rundreise durch die 20 Städte.

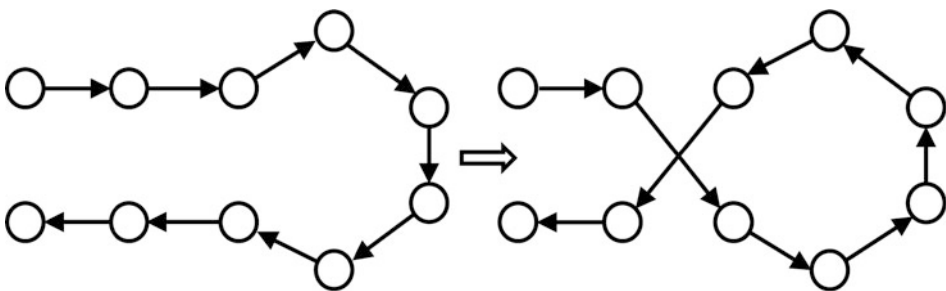


Abb. 4.17 Austausch zweier Kanten einer Rundreise

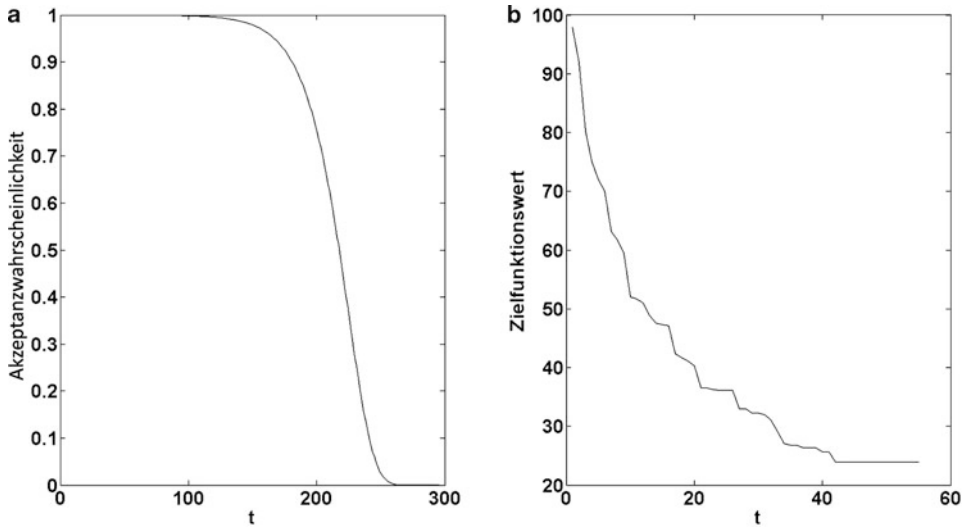


Abb. 4.18 Simulated Annealing für das Rundreiseproblem: Akzeptanzwahrscheinlichkeit (a) und Wert der Zielfunktion (b)

4.6.4 Algorithmus und Implementierung

Der gesamte Prozess des Simulated Annealing lässt sich nun als iterativer Algorithmus in Pseudocode übersetzen. Wir bezeichnen mit $NB(x)$ eine Nachbarlösung von x und mit $\text{random}(0..1)$ eine reelle Zufallszahl zwischen 0 und 1.

Algorithmus 6 SIMULATED ANNEALING

Input: Optimierungsproblem $(\Omega, f, <)$

Output: gute minimale Näherungslösung $x^* \in \Omega$

```

1: Initialisiere  $c$  (Temperatur)
2: Initialisiere  $l$  (Wiederholungsfaktor)
3: Initialisiere  $\epsilon, \alpha$ 
4: Erzeuge Anfangslösung  $x'$ 
5:  $w' = f(x')$ 
6:  $w_{\text{opt}} = w'$ 
7: while  $c > \epsilon$  do
8:   for  $k = 1$  to  $l$  do
9:     Wähle ein  $x \in NB(x')$ 
10:     $w = f(x)$ 
11:     $\Delta = w - w'$ 
12:    if  $w < w_{\text{opt}}$  then
13:       $x^* = x$ 
14:       $w_{\text{opt}} = w$ 
15:    if  $\Delta \leq 0$  OR  $e^{-\Delta/c} \geq \text{random}(0..1)$  then
16:       $x' = x$ 
17:       $w' = w$ 
18:     $c = \alpha c$ 

```

Allgemeine Erklärung

In Zeile 1–6 werden die Simulationsparameter, die Anfangslösung und der Zielfunktionswert initialisiert. Im Wert w_{opt} wird der bisher bestmögliche Zielfunktionswert abgespeichert. In der Zeile 7–18 wird das Simulated Annealing Verfahren durchgeführt, so lange wie der Temperaturparameter größer als die vorgegebene Schranke ε ist. Der Temperaturparameter wird dabei in Zeile 18 jeweils um den Wert $\alpha < 1$ verkleinert. In Zeile 8–17 werden für den aktuellen Temperaturwert c genau l verschiedene Lösungen angeschaut. Dafür werden in Zeile 9 eine beliebige Nachbarlösung x von der aktuellen Lösung x' bestimmt und in Zeile 10 und 11 deren Zielfunktionswert w und der zugehörige Differenzwert Δ bestimmt. In Zeile 12 wird überprüft, ob der neue Zielfunktionswert w kleiner als w_{opt} ist. Falls ja, ist eine neue optimale Lösung gefunden, sodass in Zeile 13–14 der Lösungsvektor und der Zielfunktionswert aktualisiert werden. Führt dieser Schritt zu einer Verschlechterung des Zielfunktionswertes, so wird in Zeile 15–17 die Lösung x akzeptiert, wenn sie entweder besser ist als die letzte Lösung x' , oder wenn die Akzeptierungswahrscheinlichkeit $e^{-\Delta/c}$ größer ist als eine reelle Zufallszahl zwischen 0 und 1.

Bemerkung 4.2

1. Für die Wahl der Konstanten gelten die folgenden Empfehlungen:
 - Konstante α : kleiner 1, etwa $0,8 \leq \alpha \leq 0,99$;
 - Konstante ε : Zahl nahe bei 0;
 - Konstante c : ca. 10-mal so groß wie der größtmögliche Δ -Wert.
2. Am Anfang des Verfahrens, wenn die Temperatur hoch ist, führt das Verfahren eine Zufallsirrfahrt auf den Lösungsraum aus. Für die Akzeptierungswahrscheinlichkeit $P(\Delta, c) = e^{-\Delta/c}$ gilt:

$$\lim_{\Delta \rightarrow 0} P(\Delta, c) = 1 \quad \text{und} \quad \lim_{\Delta \rightarrow \infty} P(\Delta, c) = 0.$$

Je kleiner α gewählt wird, umso schneller reduziert sich die Wahrscheinlichkeit für die Akzeptanz schlechterer Lösungen.

3. Am Ende des Verfahrens, bei einer Temperatur nahe null, wird eine lokale Verbesserungsstrategie angewandt.
4. Für die Laufzeit des Verfahrens ist es notwendig einen guten Kompromiss zwischen Komplexität (gesteuert durch die Temperaturabnahme) und der zu erwartenden Lösungsgüte zu schließen.
5. Eine vereinfachte Variante von Simulated Annealing ist *Threshold Accepting*. Bei diesem Verfahren wird deterministisch entschieden, ob schlechtere Lösungen akzeptiert werden oder nicht. Hierbei wird jede Lösung akzeptiert, die den Zielfunktionswert höchstens um einen vorzugebenden Schwellenwert s verschlechtert, d. h. $\Delta \leq s$. Im Laufe des Verfahrens wird s sukzessive auf 0 reduziert.

Die Implementierung des Simulated-Annealing-Verfahrens erfolgt mit einem objekt-orientierten Entwurfsmuster der Schablone. Hierzu definieren wir zunächst ein Interface `Loesung` zur Repräsentation einer Lösung eines Suchproblems.


```
interface Loesung
{
}
```

Die Implementierung des Simulated-Annealing-Verfahrens erfolgt durch eine abstrakte Klasse `SimulatedAnnealing` mit der Schablonenmethode `optimiere` und den folgenden abstrakten Einschubmethoden:

- `berechneStartLoesung`: Berechnung einer Startlösung des Suchproblems;
- `berechneZielfunktion`: Berechnung der Zielfunktion einer Lösung;
- `aendereLoesung`: Änderung der Lösung in der Nachbarschaft;
- `kopiereLoesung`: Kopierung einer Lösung.

Die Implementierung der abstrakten Klasse mit den beschriebenen Methoden sieht dann wie folgt aus:

```
abstract public class SimulatedAnnealing
{
    private double c      = 10;    // Temperaturfaktor
    private int l         = 400;    // Wiederholungsfaktor
    private double eps     = 0.0001; // Abbruchfaktor c > eps
    private double alpha  = 0.90;   // Absenkungsfaktor von c

    public void setTemperatur(double c)
    {
        this.c = c;
    }
    public void setWiederholung(int l)
    {
        this.l = l;
    }
    public void setAbbruch(double eps)
    {
        this.eps = eps;
    }
    public void setAbsenkung(double alpha)
    {
        this.alpha = alpha;
    }
    public Loesung optimiere()
    {
        System.out.printf("Start der Optimierung: \n");
        Loesung x_neu = berechneStartLoesung();
        double w_neu  = berechneZielfunktion(x_neu);
        Loesung x, x_opt = null;
        double delta=0, w, w_opt = w_neu;
        int idx = 0;
        while (c > eps)
```

```

    {
        for(int k=0; k<1; k++)
        {
            x      = aendereLoesung(x_neu);
            w      = berechneZielfunktion(x);
            delta = w - w_neu;

            if (w < w_opt)
            {
                System.out.printf(" %d: %1.4f\n", idx, Math.abs(w));
                x_opt = kopiereLoesung(x);
                w_opt = w;
            }
            if ((delta <= 0) || (Math.exp(-delta/c) > Math.random()))
            {
                x_neu = kopiereLoesung(x);
                w_neu = w;
                idx   = idx + 1;
            }
        }
        c = alpha * c;
    }
    System.out.printf("Ende der Optimierung nach %d Iterationen\n", idx);
    return x_opt;
}

// Berechnung der Startloesung
abstract protected Loesung berechneStartLoesung();

// Berechnung der Zielfunktion
abstract double berechneZielfunktion(Loesung lsg);

// Aenderung der Loesung in der Nachbarschaft
abstract Loesung aendereLoesung(Loesung lsg);

// Kopieren einer Loesung
abstract Loesung kopiereLoesung(Loesung lsg);
}

```

Wir implementieren nun die konkreten Einschubmethoden am Beispiel des Rundreiseproblems. Dazu stellen wir eine Rundreise mit der Klasse `Rundreise` dar, die das Interface `Loesung` implementiert. Eine Rundreise ist dabei ein Vektor `x` vom Typ eines `Integer`, der die Reihenfolge der besuchten Rundreise angibt.

```

public class Rundreise implements Loesung
{
    private int x[];

    public Rundreise(int n)
    {
        x = new int[n];
    }
    public Rundreise(int x[])
    {
        this.x = x;
    }

    public int[] getRundreise()
    {
        return x;
    }
}

```

Die Klasse Rundreiseproblem implementiert nun die konkreten Einschubmethoden der abstrakten Klasse SimulatedAnnealing:

```

public class Rundreiseproblem extends SimulatedAnnealing
{
    private double abstaende[][]; // Abstandsmatrix
    private int position[][];     // Knotenposition
    private int n;                // Anzahl der Knoten

    public Rundreiseproblem(double A[][], int pos[][])
    {
        abstaende = A;
        position = pos;
        n = A.length;
    }

    public int[][] getPosition()
    {
        return position;
    }

    protected Rundreise berechneStartLoesung()
    {
        int x[] = new int[n+1];
        for (int i=0; i<n; i++)
            x[i] = i;
        x[n] = 0;
        return new Rundreise(x);
    }
}

```

```

protected double berechneZielfunktion(Loesung lsg)
{
    int x[] = ((Rundreise) lsg).getRundreise();
    double w = 0;
    for(int i=0; i<n; i++)
        w = w + abstaende[x[i]][x[i+1]];
    return w;
}

protected Rundreise kopiereLoesung(Loesung lsg)
{
    int x[] = ((Rundreise) lsg).getRundreise();
    return new Rundreise(x);
}

protected Rundreise aendereLoesung(Loesung lsg)
{
    int x[] = ((Rundreise) lsg).getRundreise();
    int x_neu[] = x.clone();

    // v = randperm(n, 2);
    int v1=-1, v2=-1;
    while (v1 == v2)
    {
        v1 = (int) ((n)*Math.random());
        v2 = (int) ((n)*Math.random());
    }

    for(int i=0; i<n+1; i++)
    {
        if (x[i] == v1)
            x_neu[i] = v2;
        else if (x[i] == v2)
            x_neu[i] = v1;
    }

    return new Rundreise(x_neu);
}
}

```

Der Aufruf des Rundreiseproblems erfolgt dann durch die folgenden Anweisungen:

```

Rundreiseproblem algo = new Rundreiseproblem(abstaende);
Rundreise lsg         = (Rundreise) algo.optimiere();
int rundreise[]       = lsg.getRundreise();

```

In der Matrix `abstaende` ist die Adjazenzmatrix des gerichteten und gewichteten Graphen gespeichert. Durch Aufruf der Schablonenmethode `optimiere` wird das Simu-

lated-Annealing-Verfahren gestartet. Die Rundreise wird als Array von `int`-Werten in der Variable `rundreise` gespeichert.

4.6.5 Anwendungen

Die Anwendungsvielfalt von Simulated Annealing und genetischen Algorithmen ist in der Technik, Naturwissenschaft und Wirtschaft enorm.

Optimierung von Baugruppen Lokale Suchalgorithmen werden in vielen industriellen Bereichen zur Optimierung hochdimensionaler Optimierungsprobleme eingesetzt. Betrachten wir die Auslegung eines Systems, dass aus einer großen Anzahl von Elementen besteht. Das System soll in der Form entwickelt werden, dass gewisse Kenngrößen einen optimalen Wert besitzen. Falls das System aus einer Menge von n Elementen besteht, die jeweils in einem gewissen Bereich verändert werden können, entsteht ein n -dimensionales Optimierungsproblem. Dieses Problem ist mit analytischen Verfahren in der Regel nicht mehr effizient zu lösen.

Bei der Anwendung von Simulated Annealing wird zu Beginn für jedes Element zufällig ein Wert aus dem vorgegebenen Bereich gewählt. Damit erhalten wir eine Startkonfiguration mit einem Zielfunktionswert. Anschließend werden zufällig einige wenige Elemente ausgewählt, die die zugehörigen Werte leicht verändern, und damit die Zielfunktion bewertet. Simulated Annealing akzeptiert zu Beginn des Verfahrens mit einer bestimmten Wahrscheinlichkeit auch leicht schlechtere Lösungen, sodass der Algorithmus nicht in einem lokalen Optimum hängen bleibt. Im Laufe des Verfahrens wird die Wahrscheinlichkeit, dass schlechtere Lösungen akzeptiert werden, dann immer kleiner.

Erstellung von Stunden- bzw. Ablaufplänen Die Erstellung von Stunden- bzw. Ablaufplänen ist nicht nur eine klassische Planungsaufgabe in Schulen oder Hochschulen. Optimierte Ablaufpläne zwischen Maschinen und Aufträgen sind der Grundbaustein eines jeden gut optimierten Fertigungsprozesses. Wir betrachten hier als Demonstrationsbeispiel die Erstellung eines Stundenplanes. Hierbei ist eine ganze Menge von notwendigen oder wünschenswerten Bedingungen zu erfüllen. Notwendige Bedingungen sind beispielsweise die folgenden:

- Jeder Lehrer darf nur einer Klasse pro Zeitblock zugeordnet werden.
- Jedem Raum darf nur eine Klasse pro Zeitblock zugeordnet werden.
- Jede Klasse muss pro Woche die vorgegebene Zahl an Stunden besitzen.

Wünschenswerte Bedingungen sind beispielsweise die folgenden:

- Die Stundenpläne jeder Klasse sollten möglichst zusammenhängend sein.
- Die Arbeitsbelastung der Lehrer sollte gleichmäßig über die Woche verteilt sein.

- Fachlich anspruchsvolle Stunden (z. B. MINT-Fächer) sollten vormittags und weniger anspruchsvolle Stunden (z. B. Kunst, Musik, Sport) nachmittags abgehalten werden.

Die wünschenswerten Bedingungen können dann beispielsweise in der Zielfunktion unterschiedlich und individuell gewichtet werden.

Eine mögliche Vorgehensweise sieht wie folgt aus: Wir definieren eine zweidimensionale Tabelle mit einer Dimension für die Klassen und einer Dimension für die Stunden über die ganze Woche:

Klasse	Stunde 1	Stunde 2	Stunde 3
K1	Fach 1, Lehrer L1	Fach 2, Lehrer L2	Fach 3, Lehrer L1
K2	Fach 2, Lehrer L2	Fach 3, Lehrer L1	Fach 4, Lehrer L2

Das Simulated-Annealing-Verfahren generiert nun zufällige Lösungen, die alle zwingenden Bedingungen bezüglich der Klassen, nicht aber bezüglich der Lehrer und Räume erfüllen. Eine Nachbarlösung wird nun erzeugt, sodass eine Stundenbelegung innerhalb einer Klasse getauscht wird, d. h., dass zufällig in einer Zeile zwei Zellen vertauscht werden. Die Optimierung dieses Planungsproblems kann nun in zwei Phasen unterteilt werden:

1. Optimierung der Zielfunktion aus der Anzahl der Verletzungen von zwingenden Bedingungen, mit dem Ziel, dass alle zwingenden Bedingungen erfüllt werden.
2. Optimierung der Zielfunktion aus der Anzahl der Verletzungen von wünschenswerten Bedingungen. Hierbei werden jedoch keine Verletzungen von zwingenden Bedingungen mehr akzeptiert.

Versuchsplanung Damit Unternehmen wettbewerbsfähig sind, müssen ihre Produkte und Fertigungsprozesse ständig verbessert werden. Die Herausforderungen sind dabei die folgenden Punkte:

- Entwicklungszeit neuer Produkte verkürzen,
- Funktionsumfang der Produkte erhöhen,
- Anforderungen der Kunden besser erfüllen,
- Bearbeitungszeiten in der Fertigung verringern,
- Ausschussquote senken,
- Kosten der Produktion senken.

Die statistische Versuchsplanung (DoE) wird bei der Optimierung von Produkten oder Prozessen eingesetzt. Für immer komplexer aufgebaute Produkte kann man nur durch eine systematische Produkt- und Prozessentwicklung die beeinflussenden Parameter untersuchen, sodass Termine, Kosten und Qualität eingehalten werden. Das Ziel der statistischen Versuchsplanung ist, für diese Parameter die optimalen Werte zu bestimmen.

Als Parameter werden beim DoE diejenigen Variablen bezeichnet, die Einfluss auf interessierende Qualitätsmerkmale haben. Mit der statistischen Versuchsplanung wird mit möglichst wenigen Versuchen der Zusammenhang zwischen Einflussfaktoren (unabhängige Variablen) und Zielgrößen (abhängige Variablen) möglichst genau ermittelt. Das Simulated-Annealing-Verfahren eignet sich hervorragend für schnelle und einfache Rechnersimulationen, um geeignete Werte der Einflussfaktoren auf die gewünschte Zielgröße zu untersuchen. Durch Optimierung der Einflussparameter können nahezu optimale Werte für die Konstruktions- und Fertigungsparameter bestimmt werden.

4.7 Genetische Algorithmen

4.7.1 Grundlegende Begriffe

Genetische Algorithmen beruhen auf der Basis der biologischen Evolution durch die Anpassungsstrategien der Natur. Der Konstruktionsplan von Lebewesen ist in deren Genen gespeichert. Bei der Fortpflanzung kommt das auf C. Darwin zurückgehende Gesetz „survival of the fittest“ der Evolutionslehre zur Anwendung. Dieses Prinzip wird bei genetischen Algorithmen in ein algorithmisches Verfahren übersetzt.

Genetische Algorithmen arbeiten mit einer Kombination einer lokalen Verbesserungsstrategie und der genetischem Evolution. Die Mutationen entstehen durch Fehler bei der Reproduktion der DNA, beispielsweise bei Austausch, Einfügung oder Verlust von Basen. Im Menschen findet im Durchschnitt bei jeder 10. Zellteilung eine Veränderung statt. Die Rekombination findet bei der Fortpflanzung statt, bei der das genetische Material der Eltern neu kombiniert wird. Die Selektion bewirkt, dass Individuen, die besser an ihre Umwelt angepasst sind, mehr Nachkommen erzeugen.

Die Begriffe der natürlichen Evolution werden damit wie folgt in evolutionären Algorithmen verwendet:

Begriff	Bedeutung
Individuum	Lösungskandidat für ein Optimierungsproblem
Population	Menge von Individuen einer Generation
Genotyp	Gesamte im Individuum gespeicherte Information
Mutation	Kleine Änderung am Genotyp
Rekombination	Operation, die zwei Individuen miteinander kombiniert
Selektion	Auswahlverfahren der Individuen zur Fortpflanzung
Fitness	Auswahlvorschrift für die Selektion

4.7.2 Problemstellung

Die Definition von Problemen, die mittels genetischer Algorithmen gelöst werden, ist die gleiche wie bei Simulated Annealing.

SUCHE

Gegeben: Optimierungsproblem $(\Omega, f, <)$.

Gesucht: Element $x \in X^*$ aus der Menge der globalen Optima.

Genetische bzw. evolutionäre Algorithmen lehnen sich an der Evolution an und übersetzen die biologischen Prinzipien in vereinfachte algorithmisch ausführbare Verfahren. Genetische Verfahren lassen sich bei Optimierungsproblemen in der Praxis sehr gut anwenden. Beispiele sind die Verbesserung von technischen Konstruktionen, die Optimierung von Prozessen, die Reduzierung von Ausschuss- und Produktionszeiten oder auch die Simulation von naturwissenschaftlichen Phänomenen.

4.7.3 Grundlegende Lösungsprinzipien

Die konkrete Umsetzung von genetischen Algorithmen sieht wie folgt aus: Zuerst wird eine Population mit Lösungskandidaten generiert, die entweder zufällig sind oder bereits gute Lösungskandidaten aus einer Voroptimierung darstellen. Danach erfolgt eine Bewertung der Lösungskandidaten mithilfe einer problemspezifischen Bewertungsfunktion. Aus den stärkeren Kandidaten der Elterngeneration wird nun eine Kindergeneration gleicher Größe gebildet. Die Generierung der neuen Individuen geschieht durch eine Rekombination der Merkmale zweier Elternindividuen mit einer anschließenden Mutation der Kinderindividuen.

Die Rekombination dient der Durchmischung in der Population und die Mutation führt zu kleinen Veränderungen in den einzelnen Individuen. Anschließend beginnt der gesamte Zyklus wieder von vorne. Die Population jeder einzelnen Iteration wird als neue Generation bezeichnet. Am Ende einer Iteration wird überprüft, ob eine Abbruchbedingung bezüglich der gewünschten Güte, einer maximalen Generationszahl oder einer Anzahl der Generationen ohne Verbesserung erfüllt ist.

Für die konkrete Anwendung auf Optimierungsprobleme sind alle möglichen Lösungen Individuen aus einem sehr großen Suchraum. Durch Definition einer geeigneten Bewertungsfunktion und Methode für die Selektion, Kombination und Mutation kann dieser Prozess nach einem einheitlichen Schema ablaufen.

Prinzip der genetischen Optimierung

1. Darstellung aller möglichen Lösungen für ein Problem als Gene von Individuen
2. Erschaffung einer Generation von m Individuen mit zufällig erzeugten Genen
3. Erzeugung der Kindergeneration mit m Individuen:
 - (a) **Selektion:** Auswahl von zwei Individuen aus der Elterngeneration und Bewertung derer Leistungsfähigkeit mit einer Fitnessfunktion. Das Stärkere der beiden Individuen wird der Vater, das Schwächere wird verworfen. Nach den gleichen Regeln wird die Mutter ausgewählt

- (b) **Kombination:** Die Gene von Vater und Mutter werden vermischt und ergeben so die Gene des Kindes
- (c) **Mutation:** Mit einer bestimmten Wahrscheinlichkeit tritt eine spontane Veränderung eines Gens des Kindes auf. Mutation bedeutet beispielsweise, eine Lösung an einer oder mehreren Positionen zu verändern, indem aus einer 1 eine 0 entsteht und umgekehrt

4. Abbruchbedingung durch die Anzahl der Generationen

Selektion bedeutet, aus einer Elterngeneration besonders „fitte“ Individuen, d. h. besonders gute Lösungen, auszuwählen und in die nachkommende Generation (Population) aufzunehmen. Es gibt folgende Unterschiede zu den vorherigen Verfahren:

- Es wird nicht nur eine einzelne Lösung verändert, sondern eine Population von Lösungen.
- Neben lokalen Änderungen der Lösungen (Mutationen), werden auch Kreuzungen zwischen Lösungen erzeugt.
- Nach Erzeugung solcher neuer Lösungen werden alle bewertet, und die neue Population besteht aus denjenigen mit den höchsten Bewertungen.

Bemerkung 4.3

1. Problemspezifische Randbedingungen können bei genetischen Algorithmen wie folgt berücksichtigt werden:
 - Restriktive Methoden: Die Optimierung erfolgt auf dem unbeschränkten Suchraum, aber unzulässige Individuen werden aus der Population aussortiert oder repariert.
 - Tolerante Methoden: Die unzulässigen Individuen werden in der Population zugelassen, bekommen jedoch eine schlechtere Fitness durch Modifikation der Bewertungsfunktion mit einem Strafterm.
2. In vielen praktischen Aufgabenstellungen muss nicht nur eine, sondern auf mehrere Zielfunktionen optimiert werden. Für diese Umsetzung gibt es verschiedene Möglichkeiten zur Zusammenfassung der einzelnen Bewertungsfunktionen:
 - Bildung einer Linearkombination der einzelnen Bewertungsfunktionen $f_1, \dots, f_k$ mit Gewichtungsfaktoren $w_1, \dots, w_k$:

$$f(x) = \sum_{i=1}^k w_i f_i(x).$$

- Minimierung der Entfernung zu einem Zielvektor $y^* = (y_1^*, \dots, y_k^*)$ mit Gewichtungsfaktoren $w_1, \dots, w_k$:

$$f(x) = \sqrt{\sum_{i=1}^k w_i |f_i(x) - y_i^*|^2}.$$

Die optimale Auswahl der Selektions-, Kombinations- und Mutationsmethoden mit deren zugehörigen Parametern hängt vom konkret zu lösenden Problem ab und muss in jedem Anwendungsfall neu eingestellt werden.

Beispiel 4.6 Wir betrachten das Rucksackproblem, bei dem die Gene der Individuen ein boolescher Vektor der Länge n sind. Die erste Population wird zufällig erzeugt und mit den 0-1-Werten belegt, wobei auch unzulässige Lösungen entstehen können. Die Komponenten des genetischen Algorithmus werden dann wie folgt definiert:

1. **Fitnessfunktion:** Die Bewertung eines Individuums erfolgt durch die folgende Funktion

$$F(x) = \begin{cases} \sum_{j=1}^n w_j x_j, & \sum_{j=1}^n g_j x_j \leq G, \\ \sum_{j=1}^n w_j x_j - (\sum_{j=1}^n g_j x_j - G), & \sum_{j=1}^n g_j x_j > G, \end{cases}$$

die Individuum bestraft, die keine zulässige Lösung sind, also nicht in den Rucksack passen.

2. **Selektion:** Auswahl von zwei Individuen aus der Elterngeneration und Bewertung derer Leistungsfähigkeit mit einer Fitnessfunktion. Das Stärkere der beiden Individuen wird der Vater, das Schwächere wird verworfen. Nach den gleichen Regeln wird die Mutter ausgewählt.
3. **Kombination:** Zur Kombination der Gene wählen wir eine Zufallszahl k zwischen 0 und n . Die ersten k Gene kommen von der Mutter, die letzten $n - k$ Gene vom Vater.
4. **Mutation:** Mit Wahrscheinlichkeit $p = 1\%$ mutiert ein einzelnes Gen, kippt also zwischen 0 und 1.

Ein geeignetes Testbeispiel ist das folgende: Wir erzeugen n schwere Gegenstände mit einem zufälligen Wert im Intervall von $[100, 200]$ und einem zufälligen Gewicht im Intervall $[201, 300]$. Dann erzeugen wir weitere n leichte Gegenstände mit einem zufälligen Wert im Intervall $[201, 300]$ und einem zufälligen Gewicht im Bereich $[100, 200]$. Als maximale Last G des Rucksacks verwenden wir genau die Summe der n leichten Gegenstände. Damit besteht die optimale Lösung aus genau den n leichten Gegenständen.

4.7.4 Algorithmus und Implementierung

Das Grundprinzip der genetischen Optimierung lässt sich wie folgt in Pseudocode formulieren. Wir verwenden dazu die folgenden Methoden:

- ERZEUGESTARTLOESUNG(): Erzeugung eines Anfangsindividuums;
- FITNESS: Anwendung der Bewertungsfunktion;
- KOMBINATION: Kombination zweier Individuen;
- MUTATION: Mutation der Gene eines Individuums.

Algorithmus 7 GENETISCHE OPTIMIERUNG**Input:** Optimierungsproblem $(\Omega, f, \succ)$ **Output:** optimale Lösung x

```

1:  $P = \emptyset$ 
2: for  $i = 1$  to  $m$  do
3:    $P = P \cup \text{ERZEUGESTARTLOESUNG}()$ 
4: repeat
5:    $P' = \emptyset$ 
6:   for  $i = 1$  to  $m$  do
7:     Zufällige Wahl zweier Väter  $v_1$  und  $v_2$  aus  $P$ 
8:     Zufällige Wahl zweier Mütter  $m_1$  und  $m_2$  aus  $P$ 
9:     if  $\text{FITNESS}(v_1) > \text{FITNESS}(v_2)$  then
10:       $v = v_1$ 
11:     else
12:       $v = v_2$ 
13:     if  $\text{FITNESS}(m_1) > \text{FITNESS}(m_2)$  then
14:       $m = m_1$ 
15:     else
16:       $m = m_2$ 
17:      $x = \text{KOMBINATION}(v, m)$ 
18:      $x = \text{MUTATION}(x)$ 
19:      $P' = P' \cup \{x\}$ 
20:    $P = P'$ 
21: until Abbruchbedingung erfüllt

```

Allgemeine Erklärung

In Zeile 1–4 wird eine Anfangspopulation der Mächtigkeit m erzeugt. Eine neue Kindergeneration vom Umfang m wird in den Zeilen 6–19 bestimmt. Dazu werden in Zeile 7 und 8 jeweils zwei Väter und zwei Mütter bestimmt. In Zeile 9–16 wird das jeweils stärkere Individuum ausgewählt. Die Zeile 17 führt eine Kombination dieser zwei ausgewählten Individuen durch und die Zeile 18 eine anschließende Mutation.

Die Implementierung des genetischen Algorithmus erfolgt durch eine abstrakte Klasse `GenetischerAlgorithmus` mit der Schablonenmethode `optimiere` und den folgenden abstrakten Einschubmethoden:

- `berechneStartLoesung`: Berechnung einer Startlösung des Suchproblems;
- `berechneFitnessfunktion`: Berechnung der Fitnessfunktion einer Lösung;
- `mutiereLoesung`: Mutation der Gene der Lösung mit Wahrscheinlichkeit p ;
- `kombiniereLoesung`: Kombination zweier Lösungen mit Anteil k und $1 - k$;
- `kopiereLoesung`: Kopieren einer Lösung.

```

abstract public class GenetischerAlgorithmus
{
    private int anzInd    = 300;    // Anzahl der Individuen
    private int anzGen    = 5000;   // Anzahl der Generationen
    private double faktor = 0.5;    // Vermischungsfaktor
    private double p      = 0.05;   // Wahrscheinlichkeit der Genmutation
    private int n;              // Anzahl der Elemente der Loesung

    public void setAnzahlIndividuen(int anzInd)
    {
        this.anzInd = anzInd;
    }
    public void setAnzahlGenerationen(int anzGen)
    {
        this.anzGen = anzGen;
    }
    public void setFaktor(int faktor)
    {
        this.faktor = faktor;
    }
    public void setWhtGenmutation(double p)
    {
        this.p = p;
    }
    protected void setAnzahlLoesungselemente(int n)
    {
        this.n = n;
    }

    // Optimierung eines Minimumproblems
    public Loesung optimiere()

    // Berechnung der Startloesung
    abstract protected Loesung berechneStartLoesung();

    // Berechnung der Fitnessfunktion
    abstract double berechneFitnessfunktion(Loesung lsg);

    // Mutation der Gene der Loesung mit Wahrscheinlichkeit p
    abstract Loesung mutiereLoesung(Loesung lsg, double p);

    // Kombination zweier Loesungen mit Anteil k und 1-k
    abstract Loesung kombiniereLoesung(Loesung lsg1, Loesung lsg2, int k);

    // Kopieren der aktuellen Loesung
    abstract Loesung kopiereLoesung(Loesung lsg);
}

```

Die Schablonenmethode `optimiere` der Klasse `GenetischerAlgorithmus` sieht dann wie folgt aus:

```

public Loesung optimiere()
{
    System.out.printf("Start der Optimierung: \n");

    Loesung x_opt = null;
    double w_opt = Integer.MAX_VALUE;

    // --- 1. Erzeugung einer Generation von Individuen
    Vector<Loesung> generation = new Vector<Loesung>();
    for(int i=0; i<anzInd; i++)
        generation.add(berechneStartLoesung());

    for(int j=0; j<anzGen; j++)
    {
        Vector<Loesung> generation_neu = new Vector<Loesung>();
        for(int i=0; i<anzInd; i++)
        {
            // --- 2. Selektion: Auswahl von zwei Individuen aus der Elterngeneration
            int idx[] = Zufall.randperm(anzInd, 4);
            Loesung x1 = generation.elementAt(idx[0]);
            Loesung x2 = generation.elementAt(idx[1]);
            Loesung x3 = generation.elementAt(idx[2]);
            Loesung x4 = generation.elementAt(idx[3]);

            Loesung mutter, vater;
            if(berechneFitnessfunktion(x1) <= berechneFitnessfunktion(x2))
                vater = x1;
            else
                vater = x2;
            if(berechneFitnessfunktion(x3) <= berechneFitnessfunktion(x4))
                mutter = x3;
            else
                mutter = x4;

            // --- 3. Kombination: Vermischung der Gene der Eltern
            int k = (int) Math.round(faktor * n);
            Loesung x = kombiniereLoesung(vater, mutter, k);

            // --- 4. Mutation: Änderung der Gene mit Wahrscheinlichkeit p
            x = mutiereLoesung(x, p);
            generation_neu.add(x);

            // --- 5. Bewertung der Loesung
            double w = berechneFitnessfunktion(x);
            if (w < w_opt)
            {
                x_opt = kopiereLoesung(x);
                w_opt = w;
            }
        }
        for(int i=0; i<anzInd; i++)
        {
            Loesung lsg = kopiereLoesung(generation_neu.elementAt(i));
            generation.setElementAt(lsg, i);
        }
    }
    return x_opt;
}

```

Wir implementieren nun die konkreten Einschubmethoden am Beispiel des Rucksackproblems. Dazu stellen wir eine Rundreise mit der Klasse `Rucksack` dar, die das Interface `Loesung` implementiert.

```
public class Rucksack implements Loesung
{
    private BitSet x; // Rucksackelemente
    private int n;    // Anzahl der Elemente

    public Rucksack(BitSet x, int n)
    {
        this.x = x;
        this.n = n;
    }

    public BitSet getRucksack()
    {
        return x;
    }
    public int getAnzahl()
    {
        return n;
    }

    // Bestimmung des aktuellen Gewichtes
    public double getGewichtRucksack(double gewicht[])
    {
        double gew = 0;
        for(int i=0; i<gewicht.length; i++)
            if(x.get(i))
                gew = gew + gewicht[i];
        return gew;
    }

    // Bestimmung aller Objekte die nicht im Rucksack sind
    public Vector<Integer> getRucksackFrei()
    {
        Vector<Integer> v = new Vector<Integer>();
        for(int i=0; i<n; i++)
            if(!x.get(i))
                v.add(i);
        return v;
    }
}
```

Die Klasse `Rundreiseproblem` implementiert nun die konkreten Einschubmethoden der abstrakten Klasse `SimulatedAnnealing`:

```
public class Rucksackproblem extends GenetischerAlgorithmus
{
    private double gewinn[]; // Gewinn der Objekte
    private double gewicht[]; // Gewicht der Objekte
    private double G; // Gewichtsgrenze
    private int n; // Anzahl der Knoten

    public Rucksackproblem(double gewinn[], double gewicht[], double G)
    {
        this.gewinn = gewinn;
        this.gewicht = gewicht;
        this.G = G;
        this.n = gewinn.length;
        this.setAnzahlLoesungselemente(n);
    }

    protected Rucksack berechneStartLoesung()
    {
        BitSet x = new BitSet();
        for(int i=0; i<n; i++)
            if(Math.random() < 0.5)
                x.set(i);
        return new Rucksack(x, n);
    }

    protected double berechneFitnessfunktion(Loesung lsg)
    {
        BitSet x = ((Rucksack) lsg).getRucksack();
        double w = 0;
        double g = 0;
        for(int i=0; i<n; i++)
        {
            if (x.get(i))
            {
                w = w + gewinn[i];
                g = g + gewicht[i];
            }
        }
        if (g <= G)
            return -w;
        else
            return -(w - (g - G));
    }

    protected Loesung mutiereLoesung(Loesung lsg, double p)
    {
        BitSet x = ((Rucksack) lsg).getRucksack();
        for(int i=0; i<n; i++)
            if (Math.random() < p)
                x.flip(i);
        return new Rucksack(x, n);
    }
}
```



```

protected Loesung kombiniereLoesung(Loesung lsg1, Loesung lsg2, int k)
{
    BitSet vater = ((Rucksack) lsg1).getRucksack();
    BitSet mutter = ((Rucksack) lsg2).getRucksack();
    for(int i=k; i<n; i++)
    {
        if(vater.get(i) != mutter.get(i))
            vater.flip(i);
    }
    return new Rucksack(vater, n);
}

protected Rucksack kopiereLoesung(Loesung lsg)
{
    BitSet x = ((Rucksack) lsg).getRucksack();
    BitSet x_neu = (BitSet) x.clone();
    return new Rucksack(x_neu, n);
}
}

```

Der Aufruf des Rundreiseproblems erfolgt dann durch die folgenden Anweisungen:

```

Rucksackproblem algo = new Rucksackproblem(gewinn, gewicht, G);
Rucksack lsg          = (Rucksack) algo.optimiere();
BitSet Rucksack       = lsg.getRucksack();

```

4.7.5 Anwendungen

Wir betrachten einige Anwendungen von genetischen Algorithmen.

Packungsprobleme

Packungsprobleme sind typische Problemstellungen, die mit exakten Verfahren nicht mehr effizient lösbar sind. Verschiedene Objekte sollen möglichst platzsparend in einem oder mehreren Behältern untergebracht werden. Ein typische Anwendung ist die Beladung von m Containern der Höhe A_k , Breite B_k , Tiefe C_k und Gewichtsobergrenze g_k mit einer Menge von n quaderförmigen Packstücken der Höhe a_i , Breite b_i , Tiefe c_i und des Gewichtes g_i . Das Ziel ist einen Packungsplan x zu bestimmen, der die n Packstücke auf die m Container aufteilt, sodass gewisse Randbedingungen wie Stabilität der Beladung oder die Berücksichtigung der maximalen Tragfähigkeit eines Behälters beachtet werden.

Eine Möglichkeit der Darstellung der Lösung erfolgt über eine Packliste in Form einer Permutation der zu verstauenden Behälter mit zusätzlichen Informationen zur Drehung bei der Platzierung dieser Objekte. Die Packliste wird dann in dieser Reihenfolge abgearbeitet und jeder Behälter im aktuellen Container verstaut. Wenn ein Behälter nicht mehr

untergebracht werden kann, wird ein neuer leerer Container benutzt. Die Beladungsposition im Container kann durch gewisse Eckpunkte beschrieben werden. Die genetische Optimierung dieser Problemstellung kann dann wie folgt durchgeführt werden:

- **Fitnessfunktion:** Das zentrale Optimierungsziel ist die Maximierung der Packungsdichte:

$$f_1(x) = \frac{\sum_{i=1}^n a_i b_i c_i}{\sum_{i=1}^m A_i B_i C_i}.$$

Zwei weitere Kriterien sind das Gleichgewicht aller Container $f_2(x)$ und die Berücksichtigung der maximal möglichen Belastung pro Abschnitt $f_3(x)$. Eine gemeinsame Bewertungsfunktion kann dann die gewichtete Summe der drei Zielfunktionen darstellen:

$$f_1(x) = w_1 f_1(x) + w_2 f_2(x) + w_3 f_3(x).$$

- **Selektion:** Auswahl von zwei Individuen aus der Elterngeneration und Bewertung derer Leistungsfähigkeit mit einer Fitnessfunktion. Das Stärkere der beiden Individuen wird der Vater, das Schwächere wird verworfen. Nach den gleichen Regeln wird die Mutter ausgewählt.
- **Kombination:** Vom ersten Elternindividuum werden die ersten k Behälter der Packliste übernommen. Die fehlenden Behälter werden gemäß ihrer Reihenfolge in der Packliste des zweiten Elternindikduums aufgefüllt.
- **Mutation:** Rotation einer Kiste in ihrer Ausrichtung, Vertauschung zweier Behälter in der Packliste, Verschiebung ähnlicher Behälter in der Packliste.

Kalibrierung von Maschinen

Wir betrachten einen Kalibrierungsprozess für elektronische Steuergeräte, wie sie beispielsweise in verschiedenen Maschinen für Motoren verwendet werden. In Steuergeräten gibt es Stellgrößen, welche die Maschine in Abhängigkeit von Betriebsbedingungen steuern. Die einzelnen Bestandteile dieses Systems sind die Stellgröße y (z.B. Drehzahl, Luftmasse), die Störgröße z (z. B. Temperatur, Luftdruck, Kraftstofftemperatur) und die Ausgangsgröße x als Zielgröße der Optimierung (z. B. Kraftstoffverbrauch und die Schadstoffemissionen). Die Wahl dieser Größen bestimmt die Leistung der Maschine, den Kraftstoffverbrauch und die Menge der Emissionen. Der Suchraum dieses Optimierungsproblems sind Funktionen, die verschiedene Eingangsgrößen auf die beobachtbaren Ausgangsgrößen x mittels der Funktion $f(y, z)$ abbilden. Im Rahmen der Optimierung sind zusätzlich noch gewisse Randbedingungen zu erfüllen, beispielsweise durch vorgegebene physikalische Grenzen der Stellgrößen oder durch experimentell ermittelte unerlaubte Stellgrößenkombinationen. Die Bedingungen können durch eine Funktion h mit $h(y) < 0$ definiert werden.

Die Aufgabe der Optimierung ist, für unterschiedliche Betriebszustände unter Berücksichtigung der Randbedingungen h eine Einstellung $y \in \mathbb{R}^n$ zu bestimmen, die bezüglich der Zielgröße f optimal ist. Die Optimierung dieses Kalibrierungsprozesses wird durch ein mathematisches Modell des Motorverhaltens wie folgt berechnet:

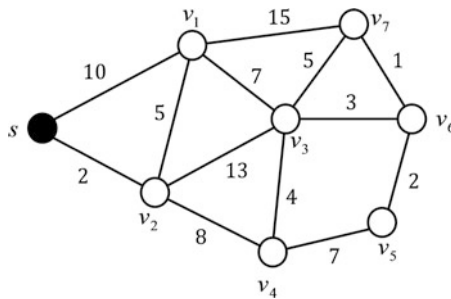
1. Erstellung eines statistischen Versuchsplans mit sinnvollen Stichproben,
2. Durchführung von Messungen der Betriebspunkte am Prüfstand,
3. Aufstellen von mathematischen Modellsystemen aus den Messdaten mit verschiedenen statistischen Regressionsmodellen (z. B. Polynomregression, neuronale Netze) zur Approximation der Zielfunktionen f und der Randbedingungen h ,
4. Optimierung der Modellsysteme bezüglich der Vorhersagegenauigkeit mit Validierungsdaten durch evolutionäre Verfahren,
5. Verifikation der Vorhersageergebnisse auf dem Prüfstand.

Mit dieser Strategie können eine große Anzahl von Messungen eingespart und damit die effektive Prüfstandszeit deutlich reduziert werden.

4.8 Übungsaufgaben

Aufgabe 4.1 (Breiten- und Tiefensuche) Erstellen Sie eine Klasse `Suchproblem` zur Bearbeitung eines allgemeinen Suchproblems mit der Breiten- und Tiefensuche. Stellen Sie dazu den Zustandsraum, den Start- und Zielzustand, die Aktionen sowie die Kostenfunktion als Attribute oder Methoden dar. Testen Sie die Klasse an einem geeigneten Suchproblem, beispielsweise dem 8-Puzzle.

Aufgabe 4.2 (Dijkstra-Suche) Gegeben sei der folgende ungerichtete Graph:



Bestimmen Sie den kürzesten Weg von s zu allen anderen Knoten des Netzwerks mithilfe des Dijkstra-Algorithmus.

Aufgabe 4.3 (Dijkstra-Suche) Eine Firma will die Produktionskosten in der Fertigung für die nächsten 5 Jahre optimieren. Die Betriebs- und Wartungskosten einer Maschine können mit der Zeit stark zunehmen. Daher kann es sinnvoll sein, eine Maschine nach ein paar Jahren durch eine neue desselben Typs zu ersetzen. Die folgende Kostenmatrix C enthält die durch den Kauf der Fertigungsmaschine am Ende des Jahres i und deren Nut-

zung bis zum Ende des Jahres $j = 1, \dots, 5$ entstehenden Kosten:

$$C = \begin{pmatrix} 4 & 15 & 28 & 45 & 62 \\ & 10 & 21 & 31 & 53 \\ & & 15 & 25 & 33 \\ & & & 16 & 30 \\ & & & & 15 \end{pmatrix}.$$

Bestimmen Sie mithilfe der Dijkstra-Suche die Zeitpunkte, zu denen die neue Maschine gegebenenfalls ersetzt werden sollte, um die Kosten der nächsten Jahre zu minimieren.

Aufgabe 4.4 (Gierige Suche) Implementieren Sie die gierige Suche am Beispiel eines Routenplanungsproblems. Vergleichen Sie die Ergebnisse für unterschiedliche Eingabeinstanzen mit der A*-Suche.

Aufgabe 4.5 (A*-Suche) Erstellen Sie ein Programm zur Berechnung eines kostenoptimalen Pfades durch einen zweidimensionalen Konfigurationsraum eines Industrieroboters mit dem A*-Algorithmus. Definieren Sie den diskretisierten Konfigurationsraum des Roboters mit einem geeigneten Eingabeformat.

Aufgabe 4.6 (A*-Suche) Erstellen Sie ein Programm zur Lösung des 8-Puzzle mithilfe des A*-Algorithmus unter Verwendung der heuristischen Schätzfunktion h , welche die Anzahl der Felder angibt, die noch falsch positioniert sind.

Aufgabe 4.7 (A*-Suche) Erstellen Sie ein Programm zur Berechnung der Routenplanung eines mobilen Roboters mit dem A*-Algorithmus (siehe Anwendungsbeispiel). In der Umgebung des Roboters kann sich dabei eine beliebige Anzahl von Hindernissen in Form von Rechtecken beliebiger Größe befinden. Erweitern Sie anschließend das Programm zur Berücksichtigung von beliebigen Hindernissen in Form von Polygonen.

Aufgabe 4.8 (Simulated Annealing) Implementieren Sie ein Simulated-Annealing-Verfahren für das Rucksackproblem mit dem vorgestellten objektorientierten Entwurfsmuster.

Aufgabe 4.9 (Genetische Algorithmen) Implementieren Sie einen genetischen Algorithmus für das Rundreiseproblem mit dem vorgestellten objektorientierten Entwurfsmuster.

Aufgabe 4.10 (Evolutionäres Verfahren) Entwickeln Sie einen Algorithmus für die Gestaltung einer Zeitung. Die Zeitung habe eine gewisse Anzahl von Spalten gleicher Breite und es müssen beliebig große Artikel überschneidungsfrei und ohne größere Lücken platziert werden.

Literaturhinweise²

1. Schöning, U. (2001). *Algorithmik*. Spektrum.
2. Cormen, T.H., Leiserson, C., Rivest, R., Stein, C. (2012). *Algorithmen – Eine Einführung*. Oldenbourg.
3. Saake, G., Sattler, K-U. (2012). *Algorithmen und Datenstrukturen (in Java)*. dpunkt.
4. v. Rimscha, M. (2009). *Algorithmen kompakt und verständlich*. Vieweg (Ebook).
5. Weicker, K. (2015). *Evolutionäre Algorithmen*. Springer Vieweg.
6. Ertel, W. (2012). *Grundkurs Künstliche Intelligenz*. Springer Vieweg.
7. Lunze, J. (2012). *Künstliche Intelligenz für Ingenieure*. Oldenbourg.
8. Russell, S., Norvig, P. (2012). *Künstliche Intelligenz*. Pearson.

² Viele klassische Suchverfahren wie die Breiten- und Tiefensuche, der Dijkstra-Algorithmus oder heuristische Suchverfahren in Form von Simulated Annealing oder genetischer Algorithmen findet man in zahlreichen Algorithmenbüchern wie [1], [2], [3] oder [4]. Einen umfassenden Überblick über evolutionäre Algorithmen mit zahlreichen Anwendungsbeispielen in Technik und Naturwissenschaft bietet das Buch [5]. Hinweise zur Literatur in Bezug auf Graphenalgorithmen findet der Leser in den Literaturhinweisen im Kap. 5.

Die Darstellung des A*-Algorithmus findet man vor allem in Büchern zur künstlichen Intelligenz [6], [7] und [8]. Alle diese drei Bücher enthalten weitere Beispiele, Anwendungen und Übungsaufgaben. Interessante Ingenieur Anwendungen findet man im Buch [6], aus dem auch das in diesem Buch vorgestellte Beispiel zur Bahnplanung eines Industrieroboters stammt.

Graphen sind mathematische Modelle für verschiedene Netze wie beispielsweise Verkehrsnetze, Computernetze, Schaltnetze, Molekülstrukturen oder für Planungsablaufpläne. Ein Graph besteht aus genau zwei Objekten, den Knoten (z. B. Städte, Computer) und den Kanten (z. B. Straßen, Leitungen) zwischen zwei Knoten. Die Kanten können dabei gerichtet oder ungerichtet sein. Der Schwerpunkt in der Graphentheorie liegt in der Untersuchung von Eigenschaften von Graphen. Die Methoden der Graphentheorie werden in der Praxis in vielen verschiedenen Bereichen angewandt. Der Grund ist, dass Netze ein sehr anschauliches Bild komplexer Zusammenhänge liefern und Bestandteil vieler Optimierungsprobleme sind.

Graphen haben neben diesen Anwendungen aus dem Bereich der Informationsverarbeitung auch Bedeutung in der E-Technik (z. B. Kirchhoff'sche Gesetze), in der Mechanik (z. B. Gittergenerierung von FEM-Systemen) oder in der Chemie (z. B. Anzahl der Isomere einer chemischen Verbindung – Molekülstrukturen bei gleicher chemischer Zusammensetzung). Auch im Alltag, bei der Nutzung eines Routenplaners oder des Mobiltelefons nutzen wir diese mathematischen Konzepte und die zugehörigen graphentheoretischen Algorithmen.

Graphen spielen nicht nur als Teil der Eingabe des zu lösenden Problems eine Rolle, sondern sie entstehen auch in Form eines Lösungsraums, in dem wir nach einer optimalen Lösung des betrachteten Problems suchen. Graphen und Netzmodelle sind auch die Grundlage für weitere mathematische Modellstrukturen wie Automaten, Bayes'sche Netze, neuronale Netze usw. Die Abb. 5.1 gibt einen Überblick über diese verschiedenen Arten von Netzstrukturen. Viele dieser Netzmodelle besprechen wir sehr ausführlich mit zahlreichen Anwendungen in den Ingenieur- und Naturwissenschaften im Band *Intelligente Algorithmen* dieser Buchreihe.

In diesem Kapitel betrachten wir die zentralsten Probleme der Graphentheorie und geben einige grundlegende Lösungsalgorithmen dazu an. Wir werden nicht immer den effizientesten Algorithmus vorstellen, da diese im Allgemeinen sehr kompliziert und schwer

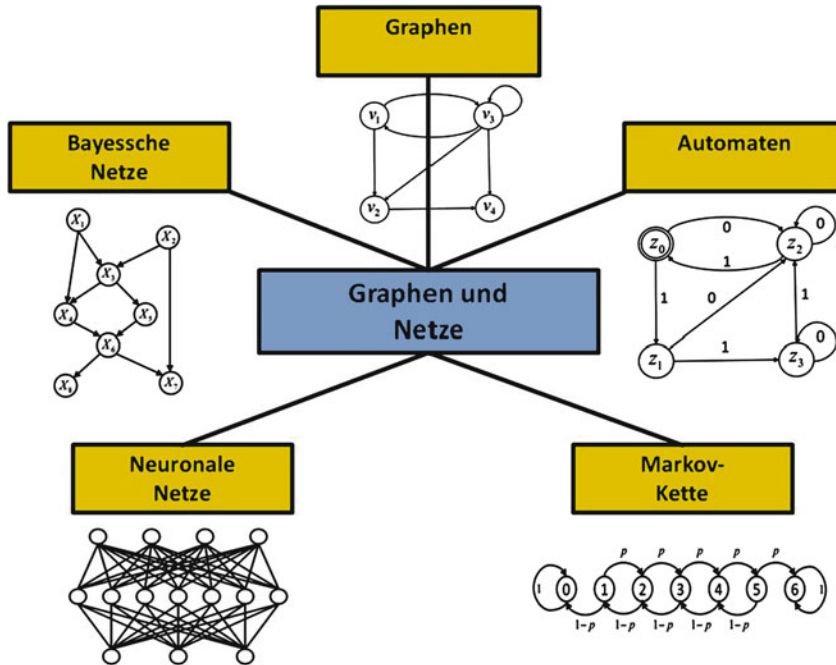


Abb. 5.1 Verschiedene Arten von Graphen und Netzmodellen

zu implementieren sind. Es geht vielmehr um die grundlegenden Ideen und Techniken, die hinter diesen Algorithmen liegen. In vielen praktischen Fällen kann man schon mit einem etwas „schlechteren“ Algorithmus die zugrunde liegende praktische Problemstellung zufriedenstellend lösen.

5.1 Grundlegende Begriffe

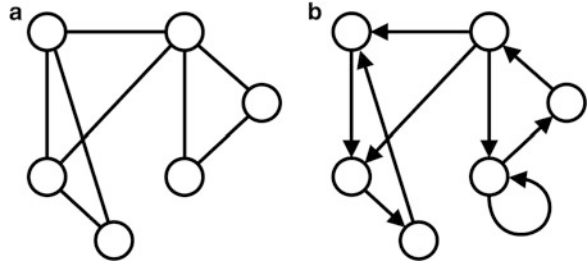
In diesem Abschnitt definieren wir zunächst die wichtigsten Grundbegriffe aus dem Bereich der Graphentheorie.

5.1.1 Graphen und Untergraphen

Definition 5.1 Ein *ungerichteter Graph* $G = (V, E)$ besteht aus einer *Knotenmenge* $V = V(G)$ und einer *Kantenmenge* $E = E(G)$, wobei jeder Kante $e \in E$ zwei (nicht notwendigerweise verschiedene) Knoten aus V zugeordnet sind. Eine ungerichtete Kante mit zwei Endknoten u und v wird in der Form $e = \{u, v\}$ beschrieben.

Wir bezeichnen mit $n = |V|$ die Anzahl der Knoten und mit $m = |E|$ die Anzahl der Kanten von G .

Abb. 5.2 Ungerichteter (a) und gerichteter (b) Graph



Definition 5.2 Ein *gerichteter Graph* $G = (V, E)$ besteht aus einer Knotenmenge $V = V(G)$ und einer Kantenmenge $E = E(G)$, sodass jedem Bogen (gerichtete Kante) $e = (u, v)$, $u, v \in V$ ein geordnetes Paar von Knoten aus V zugeordnet ist. Der Knoten u heißt *Anfangsknoten* und v der *Endknoten* des Bogens $e = (u, v)$.

Definition 5.3 Zwei Knoten $u, v \in V$, die durch eine Kante $e = \{u, v\}$ verbunden sind, heißen *adjacent*. Wenn v ein Endknoten der Kante e ist, so heißt v *inzident* zu e . Eine Kante $e = \{u, v\}$, für welche die Endknoten zusammenfallen, heißt *Schlinge*. Zwei Kanten $e = \{u, v\}$ und $f = \{u, v\}$ zwischen denselben Endknoten heißen *parallel*. Ein Graph, der weder Schlingen noch parallele Kanten besitzt, heißt *schlichter Graph*.

Die Abb. 5.2 zeigt einen ungerichteten und einen gerichteten Graphen mit jeweils 6 Knoten. Der gerichtete Graph besitzt eine Schlinge und keine parallelen Kanten.

Definition 5.4 Die *Nachbarschaft* $N_G(v)$ von Knoten v in G ist die Menge aller Knoten w von G , sodass v und w adjacent sind in G . Die Anzahl der Elemente von $N_G(v)$ heißt der *Grad* $d_G(v)$ von v . Für $U \subset V$ ist $N_G(U)$ die Menge $\bigcup_{v \in U} N_G(v)$ aller Nachbarn der Knotenmenge U .

Definition 5.5 Für einen Graphen $G = (V, E)$ seien die folgenden zwei Knotenkennzahlen definiert:

- *minimaler Grad*:

$$\delta(G) := \min\{d_G(v) \mid v \in V(G)\};$$

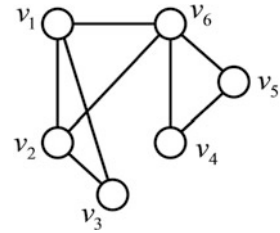
- *maximaler Grad*:

$$\Delta(G) := \max\{d_G(v) \mid v \in V(G)\}.$$

Beispiel 5.1 In Abb. 5.3 ist ein Graph $G = (V, E)$ mit der Knotenmenge $V = \{v_1, v_2, \dots, v_6\}$ und der Kantenmenge

$$E = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_6\}, \{v_2, v_3\}, \{v_2, v_6\}, \{v_4, v_5\}, \{v_4, v_6\}, \{v_5, v_6\}\}$$

Abb. 5.3 Ungerichteter Graph mit 6 Knoten



dargestellt. Für die Nachbarschaft gilt dann:

$$\begin{aligned}
 N_G(v_1) &= \{v_2, v_3, v_6\}, & d_G(v_1) &= 3, \\
 N_G(v_2) &= \{v_1, v_3, v_6\}, & d_G(v_2) &= 3, \\
 N_G(v_3) &= \{v_1, v_2\}, & d_G(v_3) &= 2, \\
 N_G(v_4) &= \{v_5, v_6\}, & d_G(v_4) &= 2, \\
 N_G(v_5) &= \{v_4, v_6\}, & d_G(v_5) &= 2, \\
 N_G(v_6) &= \{v_1, v_2, v_4, v_5\}, & d_G(v_6) &= 4.
 \end{aligned}$$

Damit sind der Minimalgrad $\delta(G) = 2$ und der Maximalgrad $\Delta(G) = 4$.

Bemerkung 5.1 Die Summe aller Knotengrade ist die doppelte Anzahl der Kanten, da jede Kante genau zwei Endknoten besitzt:

$$\sum_{v \in V} d_G(v) = 2|E|.$$

Die Anzahl der Knoten ungeraden Grades ist damit eine gerade Zahl.

Definition 5.6 Ein Graph $H = (W, F)$ ist ein *Untergraph* des Graphen $G = (V, E)$, wenn $W \subseteq V$ und $F \subseteq E$ gilt. Ein *aufspannender Untergraph* $H = (V, F)$ eines Graphen $G = (V, E)$ besitzt dieselbe Knotenmenge wie G .

Die in Abb. 5.4 dargestellten Graphen H_1 und H_2 sind beide Untergraphen des Graphen G aus Abb. 5.3. Der Graph H_2 ist dabei ein aufspannender Untergraph von G .

Abb. 5.4 Zwei Untergraphen H_1 und H_2 des Graphen aus Abb. 5.3

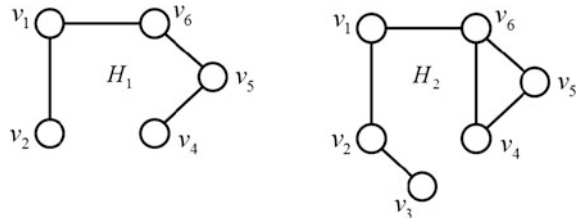
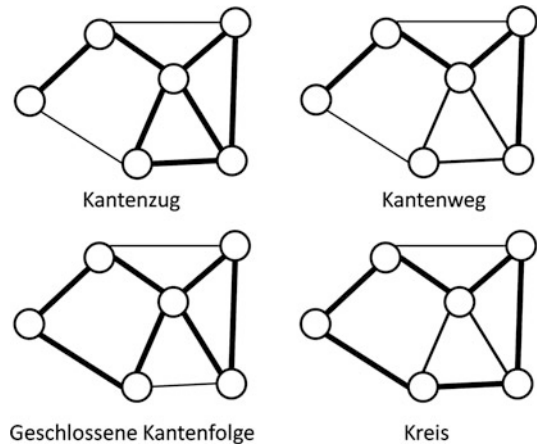


Abb. 5.5 Kantenzug, Kantenweg, geschlossene Kantenfolge und Kreis



5.1.2 Kreise und Wege

Definition 5.7 In einem Graphen $G = (V, E)$ ist eine *Kantenfolge* eine Folge

$$v_1, e_1, v_2, e_2, \dots, v_{k-1}, e_{k-1}, v_k$$

von Knoten $v_i \in V$ und Kanten $e_i \in E$ von G , sodass die Kante e_i für $i = 1, \dots, k-1$ jeweils die Endknoten v_i und v_{i+1} besitzt. Die *Länge der Kantenfolge* ist die Anzahl der Kanten dieser Folge. Eine Kantenfolge in einem Graphen $G = (V, E)$ heißt

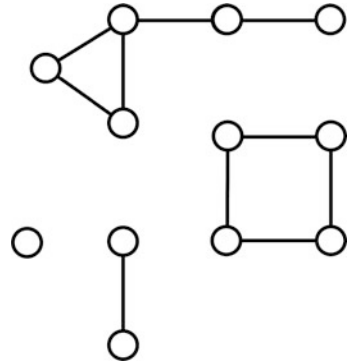
- *Kantenzug*, wenn jede Kante aus E höchstens einmal in dieser Folge auftritt;
- *Kantenweg* (kurz: *Weg*), wenn jeder Knoten aus V höchstens einmal in dieser Folge auftritt;
- *geschlossene Kantenfolge*, falls $v_1 = v_k$ gilt;
- *Kreis*, wenn mit Ausnahme von v_k kein Knoten doppelt in der geschlossenen Kantenfolge vorkommt.

In Abb. 5.5 sind ein Kantenzug, ein Kantenweg, eine geschlossene Kantenfolge und ein Kreis in einem gegebenen Graphen mit 6 Knoten und 9 Kanten dargestellt.

Definition 5.8 Ein Graph $G = (V, E)$ heißt *zusammenhängend*, wenn zwischen je zwei Knoten u und v seiner Knotenmenge ein Weg existiert. Ein maximal zusammenhängender Untergraph eines Graphen G heißt eine *zusammenhängende Komponente* von G .

In Abb. 5.6 ist ein nicht zusammenhängender Graph mit vier maximal zusammenhängenden Komponenten dargestellt, die jeweils 1, 2, 4 und 5 Knoten enthalten.

Abb. 5.6 Nicht zusammenhängender Graph mit vier maximal zusammenhängenden Komponenten



5.1.3 Spezielle Graphen

In vielen Anwendungen kommen spezielle Graphen als mathematisches Modell vor. Die wichtigsten Graphenklassen sind die folgenden:

- *Vollständiger Graph* K_n mit n Knoten besitzt zwischen je zwei Knoten genau eine Kante, insgesamt $m = \frac{n(n-1)}{2}$ Kanten (Abb. 5.7).
- *Weg* P_n der Knotenmenge $V = \{1, 2, \dots, n\}$ besitzt die Kantenmenge $\{\{1, 2\}, \{2, 3\}, \dots, \{n-1, n\}\}$ (Abb. 5.8).
- *Kreis* C_n der Knotenmenge $V = \{1, 2, \dots, n\}$ besitzt die Kantenmenge $\{\{1, 2\}, \{2, 3\}, \dots, \{n-1, n\}, \{n, 1\}\}$ (Abb. 5.9).
- *Baum* T_n mit n Knoten ist ein zusammenhängender Graph, der keinen Kreis besitzt ($m = n - 1$). Eine Menge von Bäumen bezeichnet man als *Wald* (Abb. 5.10). In einem Wald werden die Knoten vom Grad 1 *Blätter* genannt, die übrigen Knoten heißen *innere Knoten*.
- *Bipartiter Graph* $G = (V_1 \cup V_2, E)$ mit $V_1 \cap V_2 = \emptyset$, sodass alle Kanten des Graphen genau einen Endknoten in V_1 und V_2 haben (Abb. 5.11).
- *Regulärer Graph* ist ein Graph, dessen Knoten alle denselben Grad besitzen (beispielsweise Abb. 5.7).

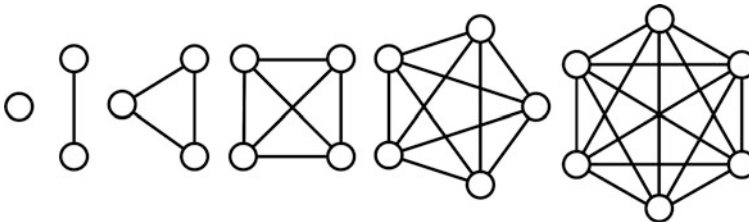


Abb. 5.7 Vollständige Graphen K_1 bis K_6

Abb. 5.8 Weg P_5

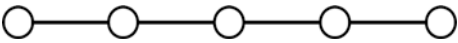


Abb. 5.9 Kreis C_5

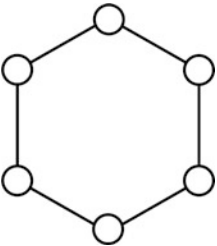


Abb. 5.10 Baum mit 6 Knoten

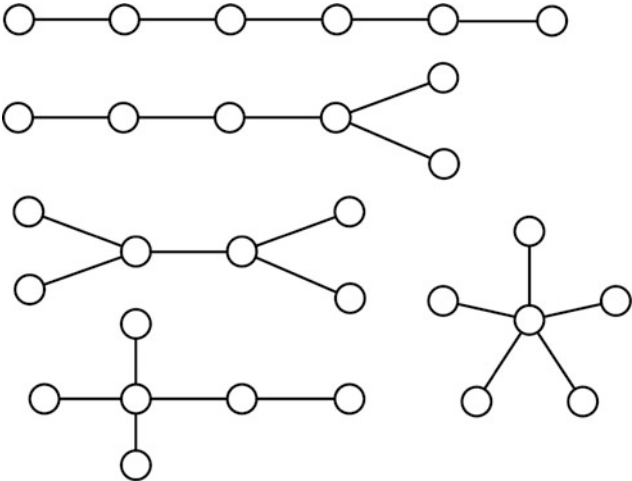


Abb. 5.11 Bipartiter Graph

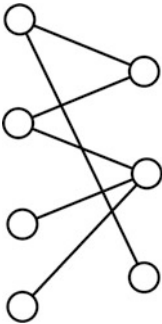
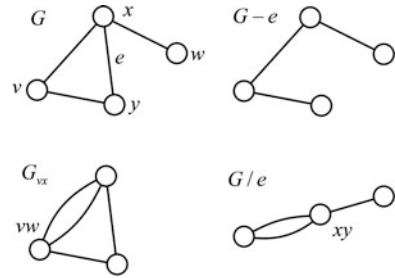


Abb. 5.12 Knoten- und Kantenoperationen



5.1.4 Graphoperationen

Für die Beschreibung von Eigenschaften und Algorithmen zu Graphen $G = (V, E)$ sind oft strukturelle Umformungen notwendig, dargestellt in Abb. 5.12:

- Entfernen einer Kante $e \in E$: $G_{-e} = (V, E \setminus \{e\})$.
- Entfernen eines Knotens $v \in V$: G_{-v} entsteht aus G durch Entfernung von v und aller zu v inzidenten Kanten.
- Fusion von zwei Knoten u und v : G_{uv} entsteht aus G durch Verschmelzung von u und v .
- Kontraktion einer Kante $e = \{u, v\}$: G/e entsteht aus G durch Entfernung von e mit der anschließenden Fusion von u und v .

Alle diese Operationen können auch für eine Menge X von Knoten oder Kanten durchgeführt werden.

5.2 Graphen und Matrizen

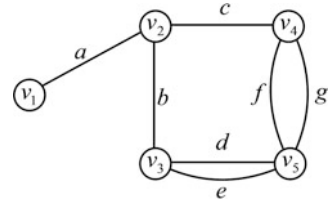
In diesem Abschnitt stellen wir zwei Varianten vor, um Graphen darzustellen, die sogenannte Matrizen- und Listendarstellung. Diese beiden Darstellungsformen werden als Datenstruktur zur Speicherung und Verarbeitung von Graphen im Rechner verwendet.

5.2.1 Matrizendarstellung

Definition 5.9 Sei $G = (V, E)$ ein ungerichteter Graph mit der Knotenmenge $V = \{v_1, v_2, \dots, v_n\}$. Wir definieren die folgenden Darstellungsmöglichkeiten für G :

1. *Adjazenzmatrix*: Die Adjazenzmatrix $A = (a_{ij})$ von G ist eine $n \times n$ -Matrix, wobei $a_{i,j}$ die Anzahl der Kanten zwischen den Knoten v_i und v_j angibt.
2. *Inzidenzmatrix*: Die Inzidenzmatrix $B = (b_{ij})$ von G ist eine $n \times m$ -Matrix mit $b_{i,j} = 1$, falls Knoten v_i benachbart (inzident) zu Kante e_j ist, und null sonst.

Abb. 5.13 Ungerichteter Graph mit 5 Knoten und 7 Kanten



3. *Gradmatrix*: Die Gradmatrix $G = (g_{ij})$ von G ist eine $n \times n$ -Matrix mit dem Knoten-grad $g_{i,i} = d_G(i)$ auf der Hauptdiagonale und null sonst.
4. *Admittanzmatrix*: Die Admittanzmatrix L von G ist eine $n \times n$ -Matrix, definiert durch $L = G - A$.

Beispiel 5.2 Wir betrachten den folgenden Graphen in Abb. 5.13. Dann erhalten wir die folgenden Matrizen:

- Adjazenzmatrix:

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 0 & 2 \\ 0 & 0 & 2 & 2 & 0 \end{pmatrix};$$

- Inzidenzmatrix:

$$B = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix};$$

- Gradmatrix:

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 4 \end{pmatrix};$$

- Admittanzmatrix:

$$L = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 3 & -1 & -1 & 0 \\ 0 & -1 & 3 & 0 & -2 \\ 0 & -1 & 0 & 3 & -2 \\ 0 & 0 & -2 & -2 & 4 \end{pmatrix}.$$

Bemerkung 5.2

1. Für ungerichtete Graphen gelten die folgenden Eigenschaften:

- (a) A , G und L sind symmetrisch.
- (b) Für alle $i = 1, \dots, n$ gilt:

$$\sum_{j=1}^n a_{ij} = d_G(i) \quad \text{und} \quad \sum_{i=1}^n \sum_{j=1}^n a_{ij} = 2m.$$

(c) $B \cdot B^T = A + G.$

(d) $B \cdot B^T + L = 2G.$

2. In gerichteten Graphen ersetzen wir die ungerichtete Kante $\{u, v\}$ durch die gerichtete Kante (u, v) . Der Grad $d_G(i)$ ist dann die Summe aus dem Außengrad $d_G^+(i)$ – Anzahl der Kanten mit Anfangsknoten i – und dem Innengrad $d_G^-(i)$ – Anzahl der Kanten mit Endknoten i jedes Knoten $i \in V$.
3. In gewichteten Graphen verwendet man häufig nur die Adjazenzmatrix, deren Einträge die jeweiligen Kantengewichte darstellen.

5.2.2 Listendarstellung

Die Speicherung eines Graphen mit n Knoten in der Matrizenform benötigt $O(n^2)$ Platz, unabhängig davon, ob der Graph sehr viele oder nur wenige Kanten besitzt. Algorithmen benötigen wegen der Initialisierung der Adjazenzmatrix daher mindestens $O(n^2)$ Rechenschritte. Selbst wenn der Graph bereits in Form einer Adjazenzmatrix als Eingabe gegeben ist und die Adjazenzmatrix nicht erst aufgebaut werden muss, kann man zeigen, dass die Laufzeit der Algorithmen für viele Graphenprobleme mindestens $O(n^2)$ beträgt.

Eine in vielen Fällen geeignetere Datenstruktur für die Darstellung von Graphen sind Adjazenzlisten. Hier wird für jeden Knoten eine linear verkettete Liste aller inzidenten Kanten gespeichert:

1. *Adjazenzliste für ungerichtete Graphen:* Gegeben sind die Außengrade $d_G(1), \dots, d_G(n)$ jedes Knotens und für jeden Knoten $i \in V$ eine Liste seiner Nachbarn $f_i : [1, \dots, d_G(i)] \rightarrow \{1, \dots, n\}$. Der Wert $f_i(j)$ ist der j -te Nachbar von Knoten i .
2. *Adjazenzliste für gerichtete Graphen:* Gegeben sind die Außengrade $d_G^+(1), \dots, d_G^+(n)$ jedes Knotens und für jeden Knoten $i \in V$ eine Liste seiner Nachbarn $f_i : [1, \dots, d_G^+(i)] \rightarrow \{1, \dots, n\}$. Der Wert $f_i(j)$ ist der j -te Nachbar von Knoten i .
3. *Adjazenzliste für gewichtete Graphen:* Gewichtete Graphen werden durch die Funktionen $f_i : [1, \dots, d_G^+(i)] \rightarrow [n] \times \mathbb{N}$ beschrieben, sodass $f_i(j) = (i', w)$, falls es eine Kante (i, i') mit Gewicht w gibt und i' der j -te Nachbar von Knoten i ist.

Beispiel 5.3 Wir betrachten den Graphen in Abb. 5.13 und erhalten die folgende Adjazenzliste, dargestellt durch die Knoten oder Kanten:

v_1	v_2		v_1	a
v_2	v_1, v_3, v_4		v_2	a, b, c
v_3	v_2, v_5, v_5	bzw.	v_3	b, d, e
v_4	v_2, v_5, v_5		v_4	c, f, g
v_5	v_3, v_3, v_4, v_4		v_5	d, e, f, g

Für jeden Knoten wird dann eine Liste von Nachbarknoten bzw. eine Liste von angrenzenden Kanten bestimmt.

5.2.3 Abstandsmatrix

Definition 5.10 Es sei $G = (V, E)$ ein Graph und $u, v \in V$. Der *Abstand* $d(u, v)$ ist die Länge des kürzesten Weges von u nach v . Gibt es keinen solchen Weg, so sei $d(u, v) = \infty$. Die *Abstandsmatrix* $D = (d_{ij})$ ist die Matrix der Abstände der einzelnen Knoten des Graphen. Der *Durchmesser* $d(G)$ von G ist der maximale Abstand zwischen zwei Knoten in G .

Beispiel 5.4 Für den obigen Graphen in Abb. 5.13 erhalten wir die Abstandsmatrix

$$D = \begin{pmatrix} 0 & 1 & 2 & 2 & 3 \\ 1 & 0 & 1 & 1 & 2 \\ 2 & 1 & 0 & 2 & 1 \\ 2 & 1 & 2 & 0 & 1 \\ 3 & 2 & 1 & 1 & 0 \end{pmatrix}.$$

Der Durchmesser $d(G)$ ist damit 3, also der maximale Wert in der Abstandsmatrix.

Bemerkung 5.3

1. Die Abstandsmatrix D ist symmetrisch.
2. Wenn G zusammenhängend ist, gilt $d(G) \leq n - 1$.
3. Für alle $u, v, w \in V$ gilt die Dreiecksungleichung:

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

Der folgende Satz gibt eine Möglichkeit an, die Anzahl der Kantenfolgen gewisser Längen zwischen zwei gegebenen Knoten zu bestimmen.

Satz 5.1 Sei $G = (V, E)$ ein ungerichteter Graph mit der Adjazenzmatrix A . Die Elemente $a_{ij}^{(k)}$ der Potenzmatrix $A^k = (a_{ij}^{(k)})$ zählen die Kantenfolgen der Länge k zwischen Knoten i und j .

Beweis Wir bestimmen das Quadrat der Adjazenzmatrix $C = (c_{ij}) = A^2$:

$$c_{ij} = \sum_{k=1}^n a_{ik} a_{kj}.$$

Das Produkt $a_{ik} a_{kj}$ wird genau dann 1, wenn eine Kante von i nach k und eine Kante von k nach j in G existieren, d.h., c_{ij} zählt die Kantenfolgen der Länge 2 zwischen i und j . Damit zählt $a_{ij}^{(k)}$ die Kantenfolgen der Länge k zwischen i und j . $\square$

Beispiel 5.5 Für den Graphen in Abb. 5.13 mit der Adjazenzmatrix

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

erhalten wir

$$A^2 = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 \\ 0 & 3 & 0 & 0 & 4 \\ 1 & 0 & 5 & 5 & 0 \\ 1 & 0 & 5 & 5 & 0 \\ 0 & 4 & 0 & 0 & 8 \end{pmatrix} \quad \text{und} \quad A^3 = \begin{pmatrix} 0 & 3 & 0 & 0 & 4 \\ 3 & 0 & 11 & 11 & 0 \\ 0 & 11 & 0 & 0 & 20 \\ 0 & 11 & 0 & 0 & 20 \\ 4 & 0 & 20 & 20 & 0 \end{pmatrix}.$$

Damit gibt es beispielsweise $a_{34}^{(2)} = 5$ Kanten der Länge 2 zwischen Knoten v_3 und v_4 bzw. $a_{53}^{(3)} = 20$ Kanten der Länge 3 zwischen Knoten v_5 und v_3 .

Bemerkung 5.4

1. Wenn für einen Graphen G die Beziehungen $a_{ij} = a_{ij}^{(2)} = \dots = a_{ij}^{(k-1)} = 0$ und $a_{ij}^{(k)} \neq 0$ gelten, so ist $d_{ij} = k$.
2. Die Potenzbildung kann nach $n - 1$ Schritten beendet werden, da in einen zusammenhängenden Graphen kein größerer Knotenabstand eintreten kann.

5.2.4 Gerüste

Die Gerüste repräsentieren eine minimale Kantenteilmenge, die den Zusammenhang des Graphen sichert. Diese Eigenschaft ist in der Zuverlässigkeitstheorie und in der Elektrotechnik für das Aufstellen von Gleichungssystemen zur Bestimmung von Strömen und Spannungen wichtig.

Definition 5.11 Ein *Gerüst* eines Graphen $G = (V, E)$ ist ein kreisfreier aufspannender zusammenhängender Untergraph von G .

Satz 5.2 Seien $G = (V, E)$ ein Graph und $\tau(G)$ die Anzahl der Gerüste von G . Dann gilt:

$$\tau(G) = \tau(G_{-e}) + \tau(G_e),$$

wobei

- $\tau(G_{-e})$ der Graph ist, der durch Entfernen der Kante e aus G hervorgeht;
- $\tau(G_e)$ der Graph ist, der durch Kontraktion der Kanten e hervorgeht, d. h., die Kante e wird entfernt und die beiden Endknoten werden fusioniert (ohne Schlinge).

Beweis Die Menge der Gerüste von G zerfällt in zwei disjunkte Teilmengen:

1. Alle Gerüste von G , die Kante $e \in E$ enthalten. Das sind auch Gerüste von G_e .
2. Alle Gerüste von G , die Kante e nicht enthalten. Das sind auch Gerüste von G_{-e} . $\square$

Bemerkung 5.5

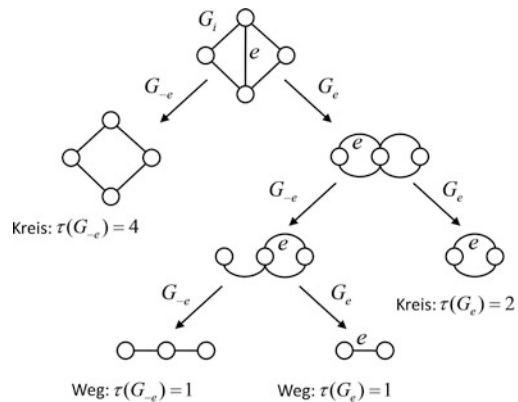
1. Der Baum T_n mit n Knoten: $\tau(T_n) = 1$.
2. Der Kreis C_n mit n Knoten: $\tau(C_n) = n$.

Beispiel 5.6 Die Bestimmung der Anzahl der Gerüste über diese rekursive Formel für einen gegebenen Graphen $G = (V, E)$ ist in Abb. 5.14 dargestellt. Damit ergibt sich die Anzahl der Gerüste aus

$$\tau(G) = \tau(G_{-e}) + \tau(G_e) = 4 + 1 + 1 + 2 = 8.$$

Alle Gerüste eines gegebenen Graphen sind in Abb. 5.15 dargestellt.

Abb. 5.14 Berechnung der Gerüste eines gegebenen Graphen



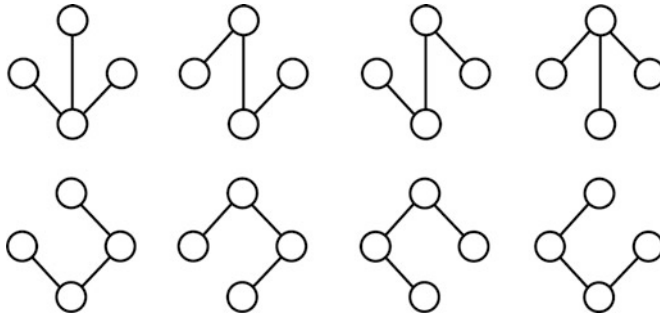


Abb. 5.15 Alle Gerüste des Graphen aus Abb. 5.14

Satz 5.3 (Satz von Kirchhoff) *Es sei G ein Graph mit der Admittanzmatrix L , und L_i sei die Matrix, die aus L durch Streichen von Zeile und Spalte i hervorgeht. Dann ergibt sich die Anzahl der Gerüste aus der Determinante*

$$\tau(G) = \det(L_i), \quad i = 1, \dots, n.$$

Beispiel 5.7 Gegeben sei wieder der Graph $G = (V, E)$ aus Abb. 5.14 mit der Admittanzmatrix

$$L = \begin{pmatrix} 2 & -1 & -1 & 0 \\ -1 & 3 & -1 & -1 \\ -1 & -1 & 3 & -1 \\ 0 & -1 & -1 & 2 \end{pmatrix}.$$

Dann erhalten wir durch Streichen der Zeile und Spalte $i = 1$ die Matrix

$$L_1 = \begin{pmatrix} 3 & -1 & -1 \\ -1 & 3 & -1 \\ -1 & -1 & 2 \end{pmatrix}.$$

Aus der Determinante ergibt sich die Anzahl der Gerüste von $\tau(G) = \det L_1 = 8$. Ebenso wären wir durch Streichen einer beliebigen anderen Zeile und Spalte der Admittanzmatrix auf das gleiche Ergebnis gekommen. Alle Gerüste dieses Graphen sind in Abb. 5.15 dargestellt.

5.3 Minimal aufspannende Bäume

5.3.1 Einführendes Beispiel

Eine Telefongesellschaft will eine bestimmte Teilmenge einer vorhandenen Menge von Kabelverbindungen mieten, wobei eine Kabelverbindung zwei Städte miteinander verbin-

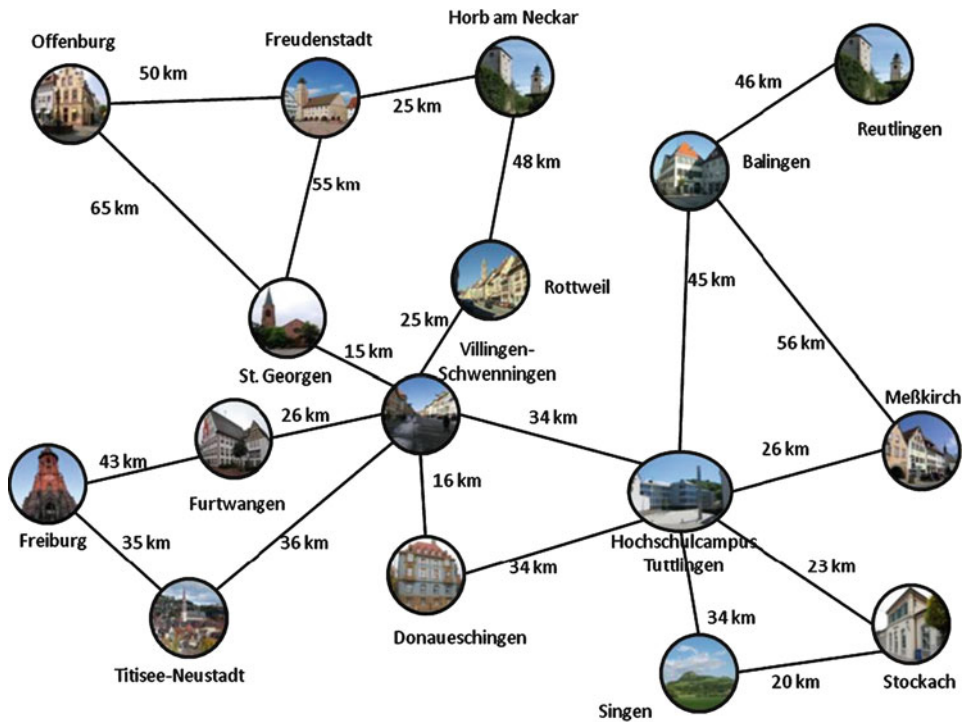


Abb. 5.16 Versorgungsnetz in Form eines minimal aufspannenden Baumes

det. Das Ziel ist, dass die zu mietenden Kabelverbindungen alle Städte auf die billigste Weise miteinander verbinden muss.

Dieses Problem kann man durch einen Graphen einfach darstellen. Die Knoten sind die Städte und die Kanten die Kabelverbindungen. Die zusammenhängenden aufspannenden Teilgraphen des gegebenen Graphen sind seine aufspannenden Bäume. Die Aufgabe liegt also in der Berechnung eines aufspannenden Baumes mit minimalem Gewicht. Dieses Gewicht gibt dann die Mietkosten der Kabelleitungen an.

Die Bestimmung von minimal aufspannenden Bäumen findet in zahlreichen Versorgungsnetzen wie Gas, Wasser oder Elektrizität umfassende Anwendung. Zwischen jeder Station soll ein Netz so gelegt werden, dass jede Station mit jeder anderen verbunden ist. Die Kantengewichte stellen dabei im Allgemeinen die Installationskosten dar. Die Aufgabe besteht immer in der Berechnung eines kostengünstigen Leitungsnetzes. In Abb. 5.16 ist ein Netz im Umkreis von Tuttlingen dargestellt. Die dick schwarz gezeichneten Kanten stellen den minimal aufspannenden Baum dieses Netzes dar.

5.3.2 Problemstellung

Ein minimal aufspannender Baum eines Graphen G ist ein zusammenhängender aufspannender Teilgraph von G mit einer minimalen Anzahl von Kanten. Falls die Kanten des Graphen gewichtet sind, so ist ein aufspannender Baum minimalen Gewichtes gesucht.

Obwohl die Anzahl der aufspannenden Bäume viel größer als diejenige der Wege ist, kann man dieses Problem algorithmisch viel leichter lösen. Genauer gesagt werden wir ein Verfahren zur Bestimmung eines minimal aufspannenden Baumes kennenlernen, das in linearer Zeit bezüglich der Anzahl der Kanten im Graph arbeitet. Auf der anderen Seite kann beispielsweise ein kürzester Weg, der jeden Knoten von G genau einmal besucht, nur in exponentieller Zeit bestimmt werden.

Wir betrachten im Folgenden die zwei Problemklassen:

MINIMAL AUFSPANNENDER BAUM

Gegeben: ein ungerichteter Graph $G = (V, E)$ und Gewichte $c : E \rightarrow \mathbb{R}$.

Gesucht: aufspannender Baum in G mit minimalem Gewicht (falls vorhanden).

In vielen graphentheoretischen Problemen ist die Bestimmung eines minimal aufspannenden Baumes ein Teilproblem, das es zu lösen gilt. Der im Nachfolgenden vorgestellte Algorithmus zur Bestimmung eines minimal aufspannenden Baumes eines Graphen ist somit ein ganz zentraler Algorithmus im Bereich der Graphen- und Netztheorie.

5.3.3 Grundlegende Lösungsprinzipien

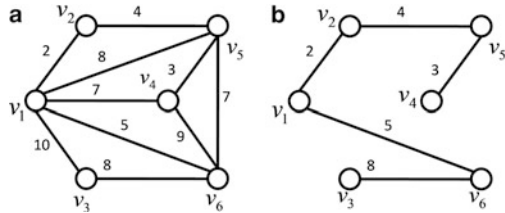
Der Prim- bzw. der Kruskal-Algorithmus sind zwei Algorithmen zur Bestimmung eines minimal aufspannenden Baumes. Der Kruskal-Algorithmus ist ein Greedy-Algorithmus, der die Kanten nach ihren Gewichten der Größe nach sortiert. Anschließend werden in dieser Sortierreihenfolge alle Kanten e in den Baum T eingefügt, sodass $T \cup \{e\}$ kreisfrei ist.

Wir stellen im Folgenden den Prim-Algorithmus näher vor. Der Prim-Algorithmus wählt zunächst einen beliebigen Knoten v aus dem Graphen G . Diesen Knoten bezeichnen wir als erreichbar. Alle anderen Knoten des Graphen G sind damit nicht erreichbar. Der Baum T besteht damit aus der Knotenmenge $V(T) = \{v\}$ und der Kantenmenge $E(T) = \emptyset$. Wir bestimmen dann eine Kante e mit minimalem Gewicht zwischen einem erreichbaren und einem nicht erreichbaren Knoten u und fügen diese Kante zum Baum T hinzu:

$$V(T) = V(T) \cup \{u\} \quad \text{und} \quad E(T) = E(T) \cup \{e\}.$$

Diesen Schritt führen wir so lange aus, bis alle Knoten in T erreichbar sind:

Abb. 5.17 Graph G (a) und zugehöriger minimal aufspannender Baum T (b)



Beispiel 5.8 Wir betrachten den gewichteten Graphen $G = (V, E)$ in Abb. 5.17. Dieser Graph besitzt die Adjazenzmatrix

$$A = \begin{pmatrix} 0 & 2 & 10 & 7 & 8 & 5 \\ 2 & 0 & 0 & 0 & 4 & 0 \\ 10 & 0 & 0 & 0 & 0 & 8 \\ 7 & 0 & 0 & 0 & 3 & 9 \\ 8 & 4 & 0 & 3 & 0 & 7 \\ 5 & 0 & 8 & 9 & 7 & 0 \end{pmatrix}.$$

Wir bezeichnen mit $d(v)$ den Abstand eines Knotens v vom aktuellen Spannbaum T . Der Knoten u ist ein noch nicht in T enthaltener Knoten von G , zu dem eine Kante e mit minimalem Gewicht existiert. Anschließend bestimmen wir eine Kante e mit minimalen Gewicht zwischen einem erreichbaren und einem nicht erreichbaren Knoten:

Iteration	u	$d(v_1)$	$d(v_2)$	$d(v_3)$	$d(v_4)$	$d(v_5)$	$d(v_6)$
1	1	0	2	10	7	8	5
2	2	0	0	10	7	4	5
3	5	0	0	10	3	0	5
4	4	0	0	10	0	0	5
5	6	0	0	8	0	0	0
6	3	0	0	0	0	0	0

Damit erhalten wir den minimal aufspannenden Baum T des Graphen G mit $V(T) = V(G)$ und

$$E(T) = \{\{v_1, v_2\}, \{v_1, v_6\}, \{v_2, v_5\}, \{v_3, v_6\}, \{v_4, v_5\}\}.$$

Prinzip des Prim-Algorithmus

- 1. Wahl eines beliebigen Knotens v als Startgraph für den Baum T
- 2. Solange T noch nicht alle Knoten enthält:
 - (a) Bestimmung einer Kante e mit minimalem Gewicht, die einen noch nicht in T enthaltenen Knoten w mit T verbindet

(b) Einfügen der Kante e und des Knotens u in T :

$$V(T) = V(T) \cup \{u\} \quad \text{und} \quad E(T) = E(T) \cup \{e\}$$

5.3.4 Algorithmus und Implementierung

Wir bezeichnen mit $V(v_i)$ den Vorgängerknoten u des Knotens v_i für die Kante (u, v_i) im Spannbaum T . Weiterhin bestimmt die Funktion $\text{GETMINIMUM}(Q, d)$ das kleinste Element der Menge Q bezüglich der Abstände d aller Knoten von G zum Spannbaum T . Mithilfe der vorgestellten Konstruktionsvorschrift lässt sich der Prim-Algorithmus wie folgt als Pseudocode darstellen:

Algorithmus 8 PRIM-ALGORITHMUS

Input: Graph $G = (V, E)$, Gewichtsfunktion $c : E \rightarrow \mathbb{R}$
Output: aufspannender Baum $T = (V, E')$ mit maximalem Gewicht
Komplexität: $O(n^2)$

```

1:  $Q = V$ 
2: for  $i = 1$  to  $n$  do
3:    $d(v_i) = \infty$ 
4:    $V(v_i) = 0$ 
5:  $d(v_1) = 0$ 
6:  $T = (\{v_1\}, \emptyset)$ 
7: while  $|Q| > 0$  do
8:    $u = \text{GETMINIMUM}(Q, d)$ 
9:   for  $i = 1$  to  $n$  do
10:    if  $(u, v_i) \in E(G)$  and  $v_i \in Q$  and  $c(u, v_i) < d(v_i)$  then
11:       $d(v_i) = c(u, v_i)$ 
12:       $V(v_i) = u$ 
13:    $V(T) = V(T) \cup \{u\}$ 
14:    $E(T) = E(T) \cup \{V(u), u\}$ 
15:    $Q = Q \setminus \{u\}$ 

```

Allgemeine Erklärung

In Zeile 1–6 werden die Menge Q der noch nicht besuchten Knoten, der Vektor $d(v_i)$ der Abstände aller Knoten v_i zu T , die Menge aller Vorgänger $V(v_i)$ sowie der Baum T initialisiert. Die Zeilen 7–15 berechnen den minimalen Spannbaum, sodass jeweils ein noch nicht besuchter Knoten u mit minimalem Abstand zum Baum T bestimmt wird. Anschließend wird in Zeile 9–12 der Abstand $d(v_i)$ für alle $v_i \in Q$ der Nachbarknoten von u aktualisiert. In Zeile 13–15 werden der neue Knoten u und die zugehörige Kante e zum Baum T hinzugefügt.

Aufwandsabschätzung

Die Initialisierungsschritte des Verfahrens benötigen den Aufwand von $O(n)$. Jede Wahl der minimalen Kante erfolgt in $O(n)$. Die Aktualisierung der Werte des Vektors d und V benötigt $O(n)$ Schritte. Aufgrund der Tatsache, dass ein Baum $n - 1$ Kanten enthält, durchläuft der Algorithmus $O(n)$ Iterationen. Damit erhalten wir eine Schranke $O(n^2)$ für die Laufzeit.

Die Implementierung des Prim-Algorithmus in Java erfolgt wieder mithilfe der Datenstruktur Graph. Die Menge Q repräsentieren wir mit der Klasse BitSet.

```
public Graph berechneAufspannendenBaum()
{
    // --- 1. Initialisierung der Variablen
    Graph G    = new Graph(n, gerichtet);
    BitSet Q    = new BitSet(); // Nichtbesuchter Knoten
    double d[] = new double[n]; // Abstand zum Baum T
    int V[]    = new int[n];    // Vorgaengerknoten
    for(int i=0; i<n; i++)
    {
        Q.set(i);
        d[i] = Integer.MAX_VALUE;
    }
    d[0] = 0;

    // --- 2. Berechnung des minimalen Spannbaumes
    while(!Q.isEmpty())
    {
        // --- 2a. Bestimme Knoten mit minimalem Abstand
        int u = getMinimum(Q, d);
        for(int v=0; v<n; v++)
        {
            // --- 2b. Aktualisierung des Abstandes für alle Nachbarknoten in Q
            if (A[u][v] > 0 && Q.get(v) && A[u][v] < d[v])
            {
                d[v] = A[u][v];
                V[v] = u;
            }
        }
        G.addKante(V[u], u, A[V[u]][u]);
        if (!gerichtet)
            G.addKante(u, V[u], A[V[u]][u]);
        Q.clear(u);
    }
    return G;
}
```

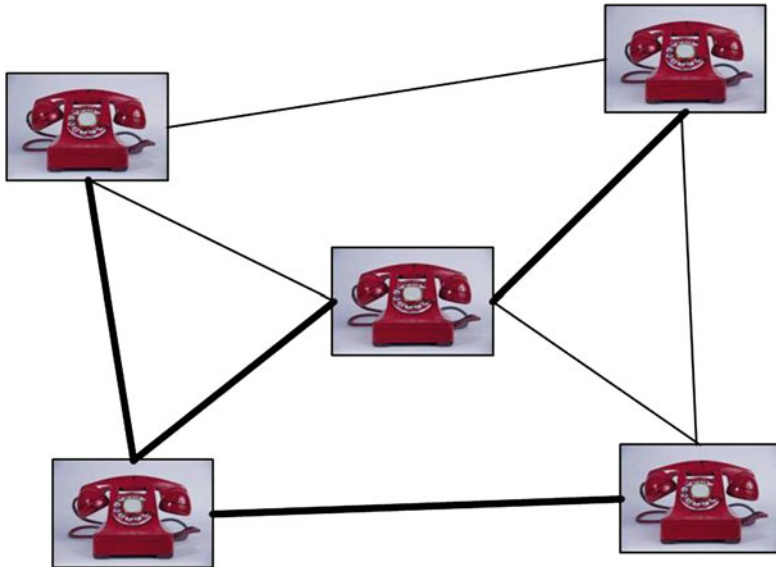


Abb. 5.18 Telefonnetz

5.3.5 Anwendungen

Wir betrachten verschiedene Anwendungen von minimal aufspannenden Bäumen.

Planung von Kommunikationsnetzen Minimal aufspannende Bäume findet man sehr häufig bei der Planung von Kommunikationsnetzen, Computernetzen oder elektrischen Netzen. Bei Telefonnetzen sind die Knoten eines Graphen die hausinternen Telefonanschlüsse einer großen Firma und die Kantenlängen repräsentieren die Kosten für das Legen einer entsprechenden Direktleitung (siehe Abb. 5.18). Die Telefongespräche von einem Anschluss zu einem anderen Anschluss können dabei auch über Zwischenstationen geschaltet werden. Beispielsweise berechnet die amerikanische Telefonfirma AT&T die Gebühren für hausinterne Netze von Firmenkunden nach der Länge eines minimal spannenden Baumes und nicht nach der Länge der tatsächlich verlegten Leitungen.

Planung von Verkehrssystemen Die Planung von Verkehrssystemen ist eine zentrale Anwendung von minimal aufspannenden Bäumen. Beispielsweise wird beim Bau eines Brückensystems minimaler Gesamtlänge zwischen einer Gruppe von Inseln der beschriebene Prim-Algorithmus angewandt. Die Menge der Inseln entspricht der Menge der Knoten des Graphen. Falls zwischen jeder Insel eine Brücke gebaut werden kann, entsteht ein vollständiger Graph. Die Kanten des dazugehörigen minimal aufspannenden Baumes entsprechen den zu bauenden Brücken, um alle Inseln miteinander zu verbinden.

Minimal aufspannende Bäume finden auch bei der Berechnung eines kostengünstigen Anschlusses eines Neubaugebietes an die Kanalisation Verwendung. Ebenso finden Spannbäume bei der Erstellung von Labyrinthen eine Rolle, die nur einen einzigen Lösungsweg besitzen.

5.4 Flüsse in Netzen

5.4.1 Einführendes Beispiel

In diesem Abschnitt beschäftigen wir uns mit den folgenden Fragestellungen: Wie viele Fahrzeuge pro Minute können höchstens durch eine Stadt fahren ohne einen Verkehrskollaps anzurichten? Welche Wassermenge kann durch die Kanalisation höchstens abtransportiert werden? Wie viel elektrische Energie kann über ein Stromnetz von einer Quelle an einen Bedarfsort transportiert werden?

Diese und andere Flussprobleme in Netzen kommen in vielen Bereichen vor. Wir wollen hier maximale Flüsse in Netzen finden, bei denen Kanten Verbindungen repräsentieren, durch die Güter fließen können. Dabei hat jede Kante nur eine beschränkte Kapazität. Zum Beispiel verträgt ein Straßenstück nur einen Durchsatz von 20 Fahrzeugen je Minute oder ein Kanalisationsrohr verkraftet nicht mehr als 30 l/s.

Große Anwendungsbereiche haben Flussprobleme im Bereich der Informationstechnik. In Abb. 5.19 ist ein Datennetz mit einem Sender und einem Empfänger gegeben. Das Netz ist durch eine Menge von Computern beschrieben, die untereinander Datenpakete verschicken können. Durch jeden Kanal fließt nur eine beschränkte Kapazität. Welche Datenmenge kann durch ein Datennetz von einem Sender zu einem Empfänger gelangen?

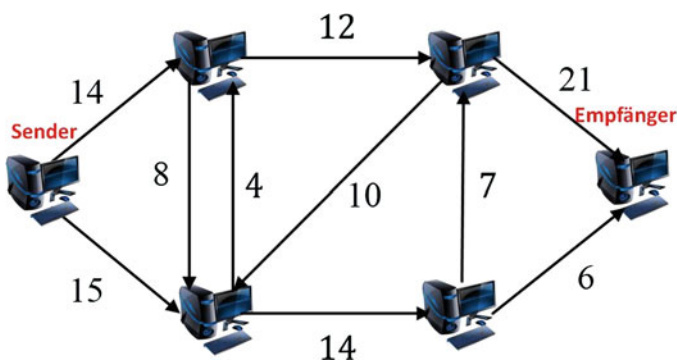


Abb. 5.19 Datennetz mit Sender und Empfänger und beschränkten Kapazitäten

5.4.2 Problemstellung

Gegeben sei ein gerichteter Graph $G = (V, E)$ mit zwei ausgezeichneten Knoten s (*Quelle*) und t (*Senke*). Jeder Kante des Graphen ist eine positive ganze Zahl (*Kapazität*) zugeordnet. Dieses Netz können beispielsweise Straßen oder Leitungen sein, durch die Fahrzeuge oder Flüssigkeit von der Quelle zu der Senke fließen. Die Aufgabe besteht darin, den Kanten Flusswerte zuzuordnen, die die jeweilige Kantenkapazität nicht überschreiten und den Gesamtfluss von der Quelle zur Senke maximieren.

Definition 5.12 Ein *Flussnetzwerk* ist ein gerichteter Graph $G = (V, E)$ mit zwei ausgezeichneten Knoten $s, t \in V$, der *Quelle* s und *Senke* t sowie einer Kapazitätsfunktion $c : V \times V \rightarrow \mathbb{N}$, wobei für $(u, v) \notin E$ gilt $c(u, v) = 0$. Ein *zulässiger* Fluss für ein Netz ist eine Funktion $f : V \times V \rightarrow \mathbb{Z}$ mit den folgenden Eigenschaften:

1. Kapazitätsbedingung: $f(u, v) \leq c(u, v)$;
2. Symmetriebedingung: $f(u, v) = -f(v, u)$;
3. Kirchhoff'sches Gesetz: $\forall u \in V \setminus \{s, t\} : \sum_{v \in V} f(u, v) = 0$.

Der *Betrag* eines Flusses f ist $|f| = \sum_{v \in V} f(s, v)$.

Die Kapazitätsbedingung besagt, dass ein Fluss durch eine Kante nicht größer als seine Kapazität sein darf. Die Symmetriebedingung wird bei dem folgenden Algorithmus benötigt und definiert einen negativen Rückfluss. Das Kirchhoff'sche Gesetz bedeutet, dass die Summe der Einflüsse gleich der Summe der Ausflüsse sein muss. Die Optimierungsaufgabe besteht darin, einen zulässigen Fluss f zu finden, sodass dessen Betrag $|f|$ maximal ist.

MAXIMUM-FLUSS-PROBLEM

Gegeben: ein Flussnetz (G, c, s, t) .

Gesucht: Bestimmung eines s - t -Flusses mit maximalem Wert.

5.4.3 Grundlegende Lösungsprinzipien

Das grundlegende Lösungsprinzip des Flussproblems basiert auf der Bestimmung von Erweiterungspfaden in einem sogenannten Restnetz. Wir werden im Folgenden diese Theorie an einem einfachen Beispiel erläutern.

Beispiel 5.9 Gegeben ist das Flussproblem in Abb. 5.20 mit 6 Knoten, einer Senke s , der Quelle t und der Kapazität c der gerichteten Kanten. In Abb. 5.21 tragen wir einen zulässigen Fluss ein, wobei die erste Zahl der Fluss f und die zweite Zahl die Kapazität c angeben. Die negativen Flüsse sind zwecks besserer Übersicht nicht mit eingetragen. An jedem beliebigen Knoten können wir das Kirchhoff'sche Gesetz verifizieren, d. h., dass

Abb. 5.20 Flussnetz mit Quelle s und Senke t

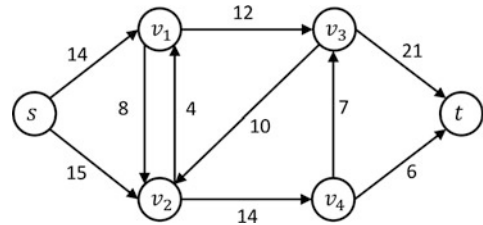


Abb. 5.21 Flussnetz mit einem zulässigen Fluss

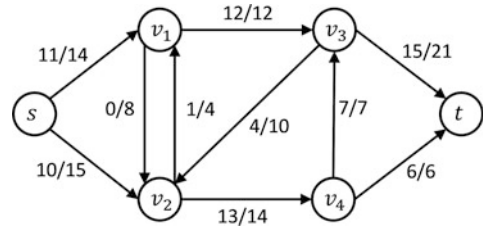


Abb. 5.22 Pfad von der Senke s zur Quelle t

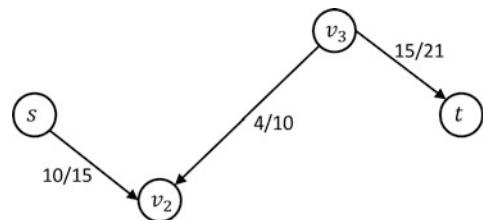
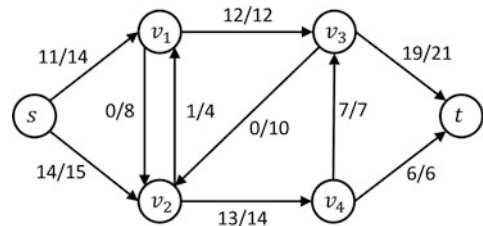


Abb. 5.23 Flussnetz mit maximalem Fluss



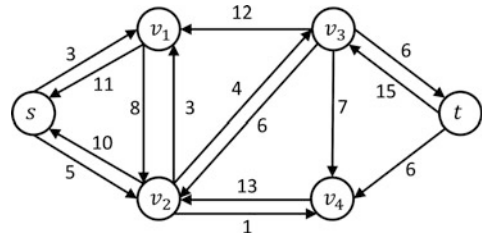
die Summe der einfließenden Ströme gleich der Summe der ausfließenden Ströme ist. Der eingezeichnete Fluss hat den Betrag von $|f| = 10 + 11 = 21$.

Der in Abb. 5.21 eingetragene Fluss ist jedoch nicht optimal, da es einen Pfad zwischen s und t gibt, entlang dem man den Fluss vergrößern kann. Der zugehörige Pfad ist in Abb. 5.22 dargestellt. Der Faktor, um den wir den Fluss vergrößern können, lautet $\min(5, 4, 6) = 4$. Die mittlere Kante dieses Pfades zeigt zwar in die verkehrte Richtung, jedoch kann man die zugehörige Kapazität um 4 verringern, was den Fluss auf dieser Kante auf 0 reduziert.

Damit erhalten wir einen neuen Fluss vom Betrag $|f| = 21 + 4 = 25$, dargestellt in Abb. 5.23.

Dieses Konzept der Verbesserung eines Flusses lässt sich durch den Begriff des Restnetzes charakterisieren.

Abb. 5.24 Restnetz für das Flussproblem



Definition 5.13 Gegeben seien ein Netz G mit Kapazität c und ein zulässiger Fluss f . Das Restnetz G_f mit der Restkapazität $c_f(u, v) = c(u, v) - f(u, v)$ ist gegeben durch $G_f = (V, E_f)$, wobei $E_f = \{(u, v) \mid c_f(u, v) > 0\}$.

Definition 5.14 Ein Erweiterungspfad ist ein einfacher Pfad p von s nach t in einem Netz G_f mit zulässigem Fluss f mit $c_f(u, v) > 0$ für alle Kanten (u, v) auf p . Die Restkapazität von p ist $c_f(p) = \min\{c_f(u, v) \mid (u, v) \text{ liegt auf } p\}$.

In Abb. 5.24 ist das zugehörige Restnetz für den Fluss in Abb. 5.21 mit dem Erweiterungspfad aus Abb. 5.22 eingezeichnet. Bei einem Restnetz G_f mit einem zulässigen Fluss f ist $(f + g)$ ein zulässiger Fluss auf G mit dem Betrag $|(f + g)| = |f| + |g|$. Ein bestehender Fluss f wird dadurch verbessert, dass man im Restnetz G_f einen Erweiterungspfad bestimmt. Damit ist

$$g(u, v) = \begin{cases} c_f(p), & (u, v) \text{ liegt auf } p, \\ -c_f(p), & (v, u) \text{ liegt auf } p, \\ 0, & \text{sonst} \end{cases}$$

ein zulässiger Fluss in G_f , mit dem der Fluss f verbessert werden kann.

Satz 5.4 Ein Fluss f ist ein maximaler Fluss genau dann, wenn das Restnetz G_f keinen Erweiterungspfad enthält.

Dieser Satz liefert den Algorithmus nach Ford und Fulkerson zur Berechnung eines maximalen Flusses. Es wird so lange nach Erweiterungswegen in dem zugrunde liegenden Restnetz gesucht, bis es keine mehr gibt. Entlang der gefundenen Pfade wird der Fluss verbessert.

Die Laufzeit des Ford-Fulkerson-Algorithmus beträgt $O(|f^*| \cdot m)$, da im ungünstigsten Fall die Anzahl der Schleifendurchläufe des Algorithmus vom Betrag $|f^*|$ des maximalen Flusses abhängt. In Abb. 5.25 erkennt man, dass man bei einer ungünstigen Wahl des Erweiterungspfades $s-v-u-t$ und anschließend $s-u-v-t$ erst nach 200 Schleifendurchläufen einen maximalen Fluss erhält. Die Bestimmung eines Erweiterungspfades mittels Breiten- oder Tiefensuche benötigt $O(m)$ Schritte.

Abb. 5.25 Bestimmung eines maximalen Flusses mit dem Ford-Fulkerson-Algorithmus bei ungünstiger Wahl des Erweiterungspfades

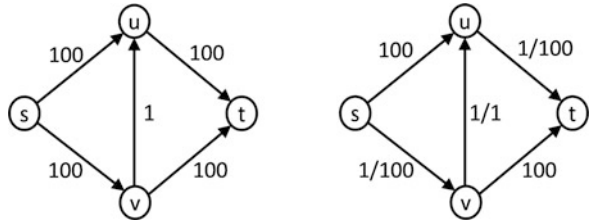
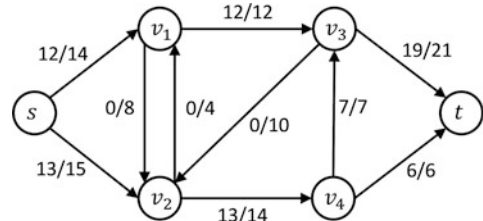


Abb. 5.26 Restnetz für das Flussproblem



Eine Verbesserung des Ford-Fulkerson-Algorithmus erfolgt mit der Edmonds-Karp-Strategie. Bei der Edmonds-Karp-Strategie wählt man als Erweiterungspfad immer den kürzesten möglichen Pfad mit der geringsten Kantenanzahl mithilfe des Dijkstra-Algorithmus aus.

Beispiel 5.10 Mithilfe der Edmonds-Karp-Strategie erhält man für den Graphen in Abb. 5.20 die folgenden Iterationsschritte:

1. Erweiterungspfad $p = (s, v_1, v_3, t)$ mit $c_f(p) = 12$ und $|f| = 12$,
2. Erweiterungspfad $p = (s, v_2, v_4, t)$ mit $c_f(p) = 6$ und $|f| = 18$,
3. Erweiterungspfad $p = (s, v_2, v_4, v_3, t)$ mit $c_f(p) = 7$ und $|f| = 25$.

Damit erhalten wir einen maximalen Fluss f mit dem Betrag von 25 nach 3 Iterationen. Der zugehörige maximale Fluss ist in Abb. 5.26 dargestellt.

Prinzip des Ford-Fulkerson-Algorithmus mit Edmonds-Karp-Strategie

1. Initialisierung des Flusses $f(u, v) = 0$ für alle $(u, v) \in V \times V$
2. Definition des Restnetzes $G_f = G$
3. Solange es noch einen Pfad in G_f von s nach t gibt:
 - (a) Bestimmung des Erweiterungspfades p von s nach t in G_f mit dem Dijkstra-Algorithmus bezüglich dem Kantengewicht von eins
 - (b) Verbesserung des Flusses f durch den Pfad p
 - (c) Bestimmung des neuen Restnetzes G_f

5.4.4 Algorithmus und Implementierung

Der Ford-Fulkerson-Algorithmus sucht nach Erweiterungswegen mithilfe des DIJKSTRA-Verfahrens im Restnetz und erhöht den Fluss entlang der Kanten. Dieses Verfahren läuft so lange, bis es keine Erweiterungswege mehr gibt. Der Pseudocode des Algorithmus lässt sich wie folgt angeben:

Algorithmus 9 MAXIMALER FLUSS

Input: Flussnetz (V, E, c, s, t)

Output: maximaler Fluss f

Komplexität: $O(nm^2)$

```

1:  $f(u, v) = 0, \forall (u, v) \in V \times V$ 
2:  $G_f = G$ 
3:  $f_{\max} = 0$ 
4: while true do
5:   for  $i = 1$  to  $n$  do
6:     for  $j = 1$  to  $n$  do
7:       if  $(i, j) \in E(G_f)$  then
8:          $d(i, j) = 1$ 
9:    $p = \text{DIJKSTRA}(G_f, d, s, t)$ 
10:  if  $t \notin p$  then
11:    break
12:   $c_f(p) = \min(c_f(e) \mid e \in p)$ 
13:   $f_{\max} = f_{\max} + c_f(p)$ 
14:  for  $(u, v) \in p$  do
15:     $f(u, v) = f(u, v) + c_f(p)$ 
16:     $f(v, u) = f(v, u) - c_f(p)$ 

```

Allgemeine Erklärung

In Zeile 1–3 werden der Fluss f , der maximale Betrag $f_{\max}$ des Flusses und das Restnetz G_f initialisiert. Die Zeilen 4–9 berechnen den zugehörigen kürzesten Pfad p vom Knoten s zum Knoten t mithilfe des Dijkstra-Algorithmus. Alle Kanten des Restnetzes G_f bekommen in diesem Fall die Länge 1. Die Zeile 10–11 stellt die Abbruchbedingung für den Fall dar, wenn der Zielknoten t nicht im Pfad p enthalten ist, d. h., es gibt keine neuen Verbesserungspfade in G_f . In Zeile 12–16 wird der Betrag des aktuellen Flusses um die Restkapazität $c_f(p)$, also die minimale Kapazität aller Kanten auf dem Pfad p verbessert.

Aufwandsabschätzung

Der Aufwand des Ford-Fulkerson-Algorithmus mit der Edmonds-Karp-Strategie hängt nur noch von der Größe des Graphen ab. Durch die Bestimmung vom kürzesten zunehmenden Weg für die einzelnen Flussvergrößerungsschritte vergrößert sich die Anzahl

der Kanten auf einem kürzesten Weg von s nach t nach höchstens m Schleifendurchläufen wenigstens um 1. Damit ist die Anzahl der erforderlichen Iterationen beschränkt durch $(n - 1)m$. Ein Erweiterungspfad lässt sich mittels Breitensuche in $O(m)$ Schritten bestimmen. Damit erhalten wir eine Laufzeit von insgesamt $O(nm^2)$ Schritten für die Berechnung eines maximalen Flusses.

Die Implementierung des Ford-Fulkerson-Algorithmus mit der Edmonds-Karp-Strategie erfolgt mithilfe des bekannten Dijkstra-Algorithmus in die Klasse `Graph`:

```
public double berechneMaxFluss(int s, int t)
{
    // --- 1. Initialisierung des Flusses
    double f[][] = new double[n][n];
    double fmax = 0;

    // --- 2. Suche nach Erweiterungspfaden
    while(true)
    {
        // --- 2a. Bestimmung des Graphen für Dijkstra-Algorithmus
        double D[][] = new double[n][n];
        for(int i=0; i<n; i++)
            for(int j=0; j<n; j++)
                if(A[i][j] > 0)
                    D[i][j] = 1;
        Graph G = new Graph(D, true);

        // --- 2b. Berechnung des Erweiterungspfades
        int vV[] = new int[n];
        G.Dijkstra(s, vV);

        // --- 2b. Abbruch: kein Erweiterungspfad mehr von s nach t
        if (vV[t] == 0)
            break;

        // --- 2c. Berechnung des Erweiterungspfades p von s nach t mit Gewicht gew
        Vector p = this.berechnePfad(vV, s);
        double gew = Double.MAX_VALUE;
        System.out.printf(" Erweiterungspfad: ");
        for(int i=0; i<p.size()-1; i++)
        {
            System.out.printf(" %d ", p.elementAt(i));
            gew = Math.min(gew, A[(int) p.elementAt(i+1)][(int) p.elementAt(i)]);
        }

        // --- 2d. Flussverbesserung
        for(int i=0; i<p.size()-1; i++)
        {
            int a = (int) p.elementAt(i+1);
            int b = (int) p.elementAt(i);
            f[a][b] = f[a][b] + gew; // vorwärts
            f[b][a] = f[b][a] - gew; // rückwärts

            A[a][b] = A[a][b] - gew;
        }
        fmax = fmax + gew;
        System.out.printf(" %d mit Kapazität - %1.1f und aktuellem Flusswert - %1.1f \n", s, gew, fmax);
        LinAlgebra.ausgabe(f);
    }
    return fmax;
}
```

5.4.5 Anwendungen

Wir stellen einige Anwendungen von maximalen Flüssen dar.

Arbeiter-Maschinen-Zuordnungsproblem Das Arbeiter-Maschinen-Zuordnungsproblem wird im Bereich der Optimierung von Produktionsanlagen benötigt. Gegeben sind die Maschinen $M_1, \dots, M_n$, die von den Arbeitern $A_1, \dots, A_m$ bedient werden. Ein Arbeiter kann dabei nicht gleichzeitig mehrere Maschinen bedienen. Weiterhin ist nicht jeder Arbeiter in der Lage, jede der n Maschinen zu bedienen. Für jede Maschine M_i , $i = 1, \dots, n$ ist eine nicht leere Teilmenge $S_i \subseteq \{1, \dots, m\}$ derjenigen Arbeiter definiert, die an der Maschine ausgebildet sind (siehe Abb. 5.27).

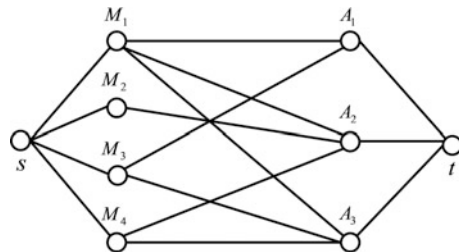
Wir modellieren das Arbeiter-Maschinen-Zuordnungsproblem in Form eines bipartiten gerichteten Graphen $G = (M \cup A, E)$, wobei $M = \{M_1, \dots, M_n\}$ die Menge aller Maschinen und $A = \{A_1, \dots, A_m\}$ die Menge aller Arbeiter darstellen. Der Knoten M_i für die Maschine i und der Knoten A_j für den Arbeiter j sind durch eine Kante (M_i, A_j) verbunden, falls die Maschine i durch den Arbeiter j bedient werden kann, d. h. $j \in S_i$.

Zur Lösung dieser Aufgabenstellung in Form eines maximalen Flussproblems fügen wir noch zwei weitere Knoten s und t hinzu, die über Kanten (s, M_i) für alle $i = 1, \dots, n$ und (A_j, t) für alle $j = 1, \dots, m$ verbunden sind. Die Kanten besitzen dabei alle das Gewicht von 1. Den entstehenden Graphen bezeichnen wir mit G' .

Gesucht ist nun eine Zuordnung, ein sogenanntes Matching $F \subset E$ in Form einer Kantenmenge, sodass keine zwei Kanten aus F einen gemeinsamen Knoten besitzen. Die Anzahl der Kanten in einem Matching mit maximaler Kantenanzahl eines bipartiten Graphen G ist gleich dem Wert eines maximalen Flusses f^* in dem zugehörigen Graphen G' .

Eine Erweiterung dieses Problems besteht darin, dass benötigte Laufzeiten $t_1, \dots, t_n \in \mathbb{R}^+$ für alle Maschinen $M_1, \dots, M_n$ gegeben sind. In diesem Fall benötigt jede Maschine genau einen Arbeiter während ihrer Laufzeit zur Bedienung. Die Aufgabe besteht in der Berechnung der Werte $x_{ij} \in \mathbb{R}_+$, welche die vom Arbeiter A_j an der Maschine M_i verbrachte Zeit mit $j \in S_i$ und die folgenden Bedingungen berücksichtigen:

Abb. 5.27 Modellierung des Arbeiter-Maschinen-Zuordnungsproblems



- Alle zugehörigen Arbeiten auf den Maschinen müssen vollständig erledigt werden:

$$\sum_{j \in S_i} x_{ij} = t_i, \quad \forall i = 1, \dots, n.$$

- Die maximale Zeitdauer an den einzelnen Maschinen soll minimal sein:

$$T(x) = \max_{j \in \{1, \dots, m\}} \sum_{\substack{i=1 \\ j \in S_i}}^n x_{ij}.$$

Damit ordnen wir im obigen Graphen G' den Kanten die Kapazitäten $c : E \rightarrow \mathbb{R}^+$ der Form $c(s, M_i) = t_i$ und $c(e) = T$ mit einem festen Wert $T \geq \sum_{\substack{i=1 \\ j \in S_i}}^n x_{ij}$ für alle anderen Kanten $e \in E(G')$ zu. Damit entsprechen die zulässigen Lösungen x gerade den maximalen s - t -Flüssen mit Wert $\sum_{i=1}^n t_i$ in diesem Netz.

Kommunikationsnetze Maximale Flüsse lassen sich auch zur Beschreibung von Kommunikationsnetzen verwenden. Ein Kommunikationsnetz ist ein Netz von Rechnern, die durch Datenübertragungswege miteinander verbunden sind. Die Aufgabe ist, den Informationsaustausch zwischen verschiedenen Orten zu ermöglichen. Die sogenannte Verletzlichkeit eines Kommunikationsnetzes ist definiert durch die Leitungen, die ausfallen müssen, sodass die Verbindung zwischen zwei Benutzern nicht mehr gegeben ist. Diese minimale Anzahl von Leitungen oder auch Stationen hängt stark von der Struktur des Netzwerkes ab. Ideal ist eine Struktur in Form eines vollständigen Graphen, die jedoch durch die hohe Anzahl von Kanten mit hohen Kosten für die Erstellung und Wartung dieses Netzwerkes verbunden ist.

Maximale Flüsse lassen sich auch zur Charakterisierung der Verletzlichkeit von Kommunikationsnetzen einsetzen. Ein Schnitt in einem Kommunikationsnetz ist eine Menge von Übertragungswegen, deren Ausfall die Kommunikation zwischen Sender s und Empfänger t unterbricht. Ein Schnitt ist formal eine Zerlegung der Knotenmenge $V = A \cup B$ mit $A \cap B = \emptyset$ und $s \in A$ und $t \in B$. Die Kapazität eines Schnitts (A, B) ist definiert durch

$$c(A, B) = \sum_{u \in A, v \in B} c(u, v)$$

und der Fluss über einen Schnitt (A, B) bestimmt sich aus

$$f(A, B) = \sum_{u \in A, v \in B} f(u, v).$$

Nach dem bekannten Min-Cut-Max-Flow-Theorem gilt der folgende Zusammenhang zwischen einem maximalen Fluss f^* und einem minimalen Schnitt in G :

$$f^* = \max_f |f| = \min_{(A,B)} c(A, B).$$

Der Wert des maximalen Flusses entspricht dem Wert eines minimalen Schnitts. Der minimale Schnitt charakterisiert die Verletzlichkeit bzw. Begrenzung eines Kommunikationsnetzes. Wenn diese Kanten ausfallen oder deren Kapazität verringert wird, wirkt sich das unmittelbar auf die Kommunikationsfähigkeit dieses Netzwerkes aus.

Soziale Netzwerke Soziale Netzwerke werden in unserer heutigen Gesellschaft von fast jedem in irgendeiner Art und Weise genutzt. Die Erforschung des Zusammenhangs zwischen der Struktur eines sozialen Netzwerkes und dem Verhalten der Personen bzw. Gruppen ist Gegenstand sozialwissenschaftlicher Untersuchungen, die unter dem Begriff Analytik sozialer Netzwerke zusammengefasst werden. Die zugehörigen Methoden kommen aus dem Bereich der Graphentheorie. Das Ziel ist, anhand von charakteristischen Kenngrößen Aussagen über das Verhalten des Netzwerkes zu machen.

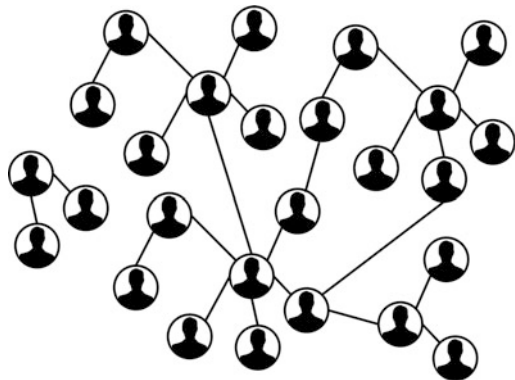
Eine Kenngröße ist beispielsweise das Maß für den Zusammenhang, wie stark zwei Personen oder zwei Gruppen von Personen miteinander vernetzt sind. Der Graph ist das soziale Netz, d. h., zwei Personen sind miteinander durch eine Kante verbunden, wenn sie in irgendeiner Beziehung stehen (siehe Abb. 5.28). Dieser Wert lässt sich über einen maximalen Fluss berechnen. Wir setzen hierzu die Kapazitäten aller Kanten des Netzwerkes auf den Wert 1. Das Maß, wie stark zwei Personen A und B miteinander verbunden sind, ergibt sich dann aus dem Betrag des maximalen Flusses zwischen A und B .

Weitere Maßzahlen für soziale Netzwerke lassen sich wie folgt definieren:

- Dichte: Verhältnis der Anzahl der Kanten zur Anzahl der möglichen Kanten:

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

Abb. 5.28 Soziales Netz von Personen



- Zentralität C_D : Verhältnis der Anzahl der ausgehenden Kanten einer Person A zur Anzahl der Personen im Netz:

$$C_D(A) = \frac{d_G(A)}{n-1}.$$

- Zentralität C_B : Maß für die Bedeutung einer Person A für den Informationsaustausch:

$$C_B(A) = \frac{2}{(n-1)(n-2)} \sum_{s \neq A \neq t, s \neq t} \frac{p_{st}(A)}{p_{st}},$$

wobei p_{st} die Anzahl der kürzesten Pfade von s nach t und $p_{st}(A)$ die Anzahl dieser Pfade des Knotens A enthalten.

- Zentralität C_C : Maß für die Nähe einer Person A zu den anderen:

$$C_C(A) = \frac{\sum_{t \neq A} p_{At}}{n-1},$$

wobei p_{At} die Anzahl der Kanten auf einem kürzesten Weg von A nach t ist.

- Clique: Zerlegung des Netzwerkes in Teilnetze, sodass die zugehörigen Personen alle untereinander verbunden sind.

5.5 Tourenprobleme

5.5.1 Einführendes Beispiel

Das Königsberger Brückenproblem stellte den Startpunkt für die Graphentheorie im Jahr 1736 dar. Die Abb. 5.29 zeigt den Stadtplan der Stadt Königsberg im 18. Jahrhundert. Die beiden Arme des Flusses Pregel umfließen eine Insel. Es gibt insgesamt

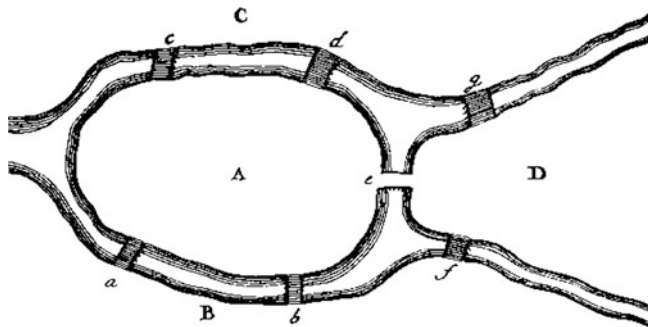
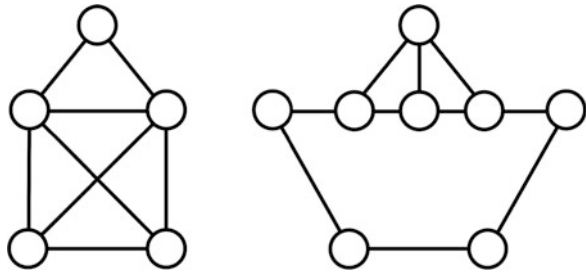


Abb. 5.29 Stadtplan von Königsberg mit den 7 Brücken

Abb. 5.30 Haus vom Nikolaus



7 Brücken über den Fluss. Das Brückenproblem bestand darin zu entscheiden, ob es einen Rundweg durch Königsberg gibt, der jede der Brücken genau einmal überquert.

Für die Lösung des Problems werden die einzelnen Ufer durch Knoten und die Brücken durch Kanten dargestellt. Das Problem reduziert sich nun darauf zu entscheiden, ob es in diesem Graphen einen Rundweg (Kreis) gibt, der jede Kante genau einmal durchläuft. Ein solcher Kreis wird in der Graphentheorie auch als Euler'scher Kreis bezeichnet.

Aus dem resultierenden Graphen sieht man, dass alle 4 Knoten eine ungerade Anzahl von Kanten besitzen. Beispielsweise gehen vom Knoten D 3 Kanten aus, d. h., wenn wir das erste Mal wieder nach D zurückkehren, haben wir 2 der 3 in D einmündenden Brücken überquert. Damit gibt es nach dem Verlassen von D keine Möglichkeit mehr, zu D wieder zurückzukehren, ohne eine Brücke mindestens 2-mal zu durchlaufen. Somit besitzt der Graph keinen Euler'schen Kreis, wenn es Knoten gibt, die eine ungerade Anzahl von Kanten besitzen.

Ein weiteres Beispiel, das sicher jeder noch aus seiner Schulzeit kennt, ist das „Haus des Nikolaus“ (siehe Abb. 5.30). Dabei handelt es sich um einen Graphen aus 5 Knoten und 8 Kanten. Die Frage lautet: Kann man diesen Graphen zeichnen, ohne den Stift abzusetzen und ohne eine Linie doppelt zu ziehen? Die Antwort lautet natürlich „ja“. Es gibt sogar 44 verschiedene Lösungen dafür. Den Weg, den der Stift dabei zurücklegt, nennt man eine Euler-Tour, nach dem bekannten Mathematiker Leonhard Euler.

5.5.2 Problemstellung

Definition 5.15 Ein *Euler-Kreis* in einem Graphen $G = (V, E)$ ist ein geschlossener Kantenzug, der jede Kante aus E genau einmal durchläuft.

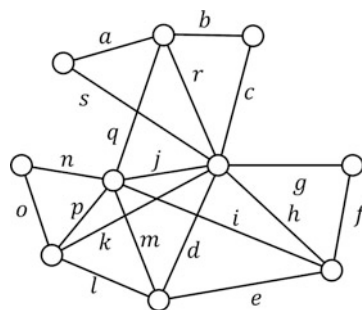
Die Abb. 5.31 zeigt einen Euler-Kreis $abc \dots s$. Mithilfe dieser Definition definieren wir zwei zentrale Problemstellungen:

EULER-KREIS-PROBLEM

Gegeben: ein Graph $G = (V, E)$.

Gesucht: ein Euler-Kreis in G , falls vorhanden.

Abb. 5.31 Euler-Kreis
 $p = (a, b, c, \dots, s)$ in einem
 Graphen



Eine Erweiterung der Euler-Tour führt auf das sogenannte Briefträgerproblem (Chinese Postman Problem). Dabei spielen auch die Längen der Straßen eine Rolle. Gegeben ist ein gewichteter Graph, und gesucht ist ein Pfad durch den Graphen, der jede Kante mindestens einmal besucht. Die Berechnung solcher optimaler Routen ist beispielsweise für die Postzustellung und die Müllabfuhr von zentraler Bedeutung.

BRIEFTRÄGERPROBLEM (CHINESE POSTMAN PROBLEM)

Gegeben: ein gewichteter Graph $G = (V, E)$.

Gesucht: ein geschlossener Pfad minimaler Länge, der jede Kante mindestens einmal besucht.

In dem Fall, dass jeder Knoten des Graphen einen geraden Grad hat, ist jede Euler-Tour eine optimale Briefträgere Tour. Andernfalls müssen einige Kanten mehr als einmal verwendet werden.

5.5.3 Grundlegende Lösungsprinzipien

Wir betrachten nun ein Kriterium, das die Frage beantwortet, wann ein Graph einen Euler-Kreis besitzt.

Satz 5.5 *Ein zusammenhängender Graph besitzt genau dann einen Euler-Kreis, wenn alle Knoten einen geraden Grad besitzen.*

Beweis Wenn der Graph G einen Euler-Kreis besitzt, verbrauchen wir beim Durchlaufen des Kreises jeweils 2 Kanten von allen Knoten. Wenn der Graph G nur Knoten mit einem geraden Grad besitzt, starten wir den Kantenzug an einem beliebigen Knoten $v \in V$ und setzen den Kantenzug so lange fort, bis wir erstmals auf einen besuchten Knoten stoßen. Wenn alle Knoten des entstandenen Kreises aus G entfernt werden, erhält man wieder einen Graphen mit ausschließlich geradem Knotengrad. Das Verfahren lässt sich dann

weiter fortführen, sodass wir eine Menge von kantendisjunkten Kreisen erhalten. Am Ende kann man diese Kreise von einem Punkt, an dem sich zwei oder mehrere Kreise treffen, nacheinander durchlaufen. □

Das „Haus des Nikolaus“ kann nur bei bestimmten Startpunkten in einem Zug gezeichnet werden. Einen Euler-Kreis besitzt es nicht. Durch das Hinzufügen eines neuen Knotens im „Haus vom Nikolaus“, den man mit beiden Knoten vom Grad 3 verbindet, erhält man einen Graphen, indem alle Knoten einen geraden Grad besitzen. Dieser entstehende Graph besitzt einen Euler-Kreis.

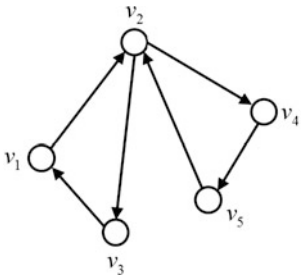
Ein einfacher Algorithmus zur Bestimmung einer Euler-Tour sieht wie folgt aus: Wir starten an einem beliebigen Knoten und verwenden eine beliebige Kante zu einem neuen Knoten. Dort folgen wir wieder einer Kante zu einem neuen Knoten usw. Jede verwendete Kante wird anschließend gelöscht. Die Reihenfolge der besuchten Knoten wird in einem Array für die Euler-Tour p gespeichert. Wenn wir an einem Knoten angekommen sind, der keine weiteren Kanten mehr besitzt, gibt es entweder keine Kanten mehr, und die Euler-Tour ist vollständig bestimmt, oder es gibt noch mindestens einen Knoten mit geradem Knotengrad. Ein solcher Knoten wird damit gesucht und die Euler-Tour an diesem Knoten weitergeführt. Hierbei müssen die neuen Knoten der Euler-Tour im Array hinzugefügt werden.

Beispiel 5.11 Wir betrachten den Graphen G in Abb. 5.32. Damit ergeben sich die folgenden Iterationen:

Iteration	Pfad p
1	(v_1)
2	(v_1, v_2)
3	(v_1, v_2, v_3)
4	(v_1, v_2, v_3, v_1)
5	$(v_1, v_2, v_4, v_3, v_1)$
6	$(v_1, v_2, v_4, v_5, v_3, v_1)$
7	$(v_1, v_2, v_4, v_5, v_2, v_3, v_1)$

In der letzten Iteration ist die vollständige Euler-Tour $p = (v_1, v_2, v_4, v_5, v_2, v_3, v_1)$ bestimmt.

Abb. 5.32 Gerichteter Graph zur Bestimmung des Euler-Kreises



Prinzip des Euler-Tour-Algorithmus

1. Initialisierung der Euler-Tour $p = \emptyset$
2. Auswahl eines beliebigen Startknotens $v \in V$
3. Solange es noch Kanten in G gibt:
 - (a) Wahl eines beliebigen Nachbarknotens w von v .
Falls keiner existiert $\Rightarrow$ Wahl eines anderen Startknotens v
 - (b) Hinzufügen der Kante $e = (v, w)$ in den Euler-Kreis p
 - (c) Löschen der Kante e aus G
 - (d) Setzen von $v = w$

5.5.4 Algorithmus und Implementierung

Der Pseudocode des Algorithmus zur Konstruktion der Euler-Tour sieht, unter einer Verwendung einer geeigneten Datenstruktur für den Pfad p , wie folgt aus:

Algorithmus 10 EULER-KREIS

Input: Graph $G = (V, E)$

Output: Euler-Kreis p

Komplexität: $O(m)$

```

1:  $p = v$ 
2: while  $E \neq \emptyset$  do
3:    $u \in \{v \mid d_G(v) > 0\}$ 
4:    $w = u$ 
5:   repeat
6:     P.ADD( $u$ )
7:      $v \in \{u \mid d_G(u) > 0\}$ 
8:      $E = E \setminus \{u, v\}$ 
9:      $u = v$ 
10:  until  $w = u$ 

```

Allgemeine Erklärung

Solange es noch Kanten im Graphen gibt, suchen wir in Zeile 3 einen beliebigen Knoten mit einer ausgehenden Kante. Anschließend fügen wir so lange in Zeile 6–9 eine Kante zur Euler-Tour hinzu, bis wir wieder am Ausgangsknoten angelangt sind.

Aufwandsabschätzung

Der Aufwand zur Bestimmung einer Kante beträgt $O(n^2)$, da jede Kante genau einmal durchlaufen wird.

Für die Implementierung in Java verwenden wir eine dynamische Datenstruktur vom Typ `Vector`. Im Array `pos` speichern wir für jeden Knoten den Index des letzten besuchten Nachbarknotens ab. In der `for`-Schleife suchen wir jeweils eine ausgehende Kante zu

einem Nachbarknoten. Falls wir auf einen Knoten treffen, der keine ausgehenden Kanten mehr besitzt, ist die aktuelle Kantenfolge beendet. Dann suchen wir einen neuen Startknoten v und fügen die neue Kantenfolge an der Position idx vom Knoten v in die Euler-Tour ein.

```
public Vector<Integer> berechneEulerTour()
{
    // --- 1. Initialisierung der Tour
    Vector<Integer> p = new Vector<Integer>();
    int v = 0;
    p.addElement(v);

    // --- 2. Bestimmung der Eulertour
    int idx = 1;
    int pos[] = new int[n];
    while(this.getKantenanzahl() > 0)
    {
        // --- 2a. Kantenfolge von v aus durchlaufen
        int i;
        for(i=pos[v]; i<n; i++)
        {
            if(A[v][i] > 0) // Kante vorhanden
            {
                p.add(idx, i);           // Kante hinzufuegen
                pos[v] = i;             // letzte Position setzen
                idx++;                  // Index erhoehen
                this.loeschenKante(v, i); // Kante loeschen
                v = i;                  // neuer Startknoten
                break;
            }
        }
        // --- 2b. Neue Kantenfolge mit Starknoten suchen
        if (i==n)
        {
            for(v=0; v<n; v++)
                if(this.getKnotengrad(v) > 0)
                    break;
            idx = p.indexOf(v) + 1;
        }
    }
    return p;
}
```

5.5.5 Anwendungen

Wir betrachten eine praktische Anwendung für Tourenprobleme in Graphen.

Briefträgerproblem Das Briefträgerproblem ist ein Klassiker aus der Graphentheorie. Die Aufgabe eines Postboten ist auf dem kürzesten Weg Briefe in einem Straßennetz auszuliefern. Gegeben ist ein gewichteter zusammenhängender Graph, und gesucht ist ein

Abb. 5.33 Graph zur Bestimmung einer optimalen Route eines Briefträgers

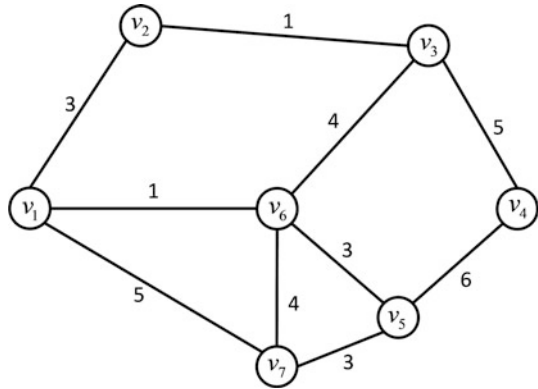
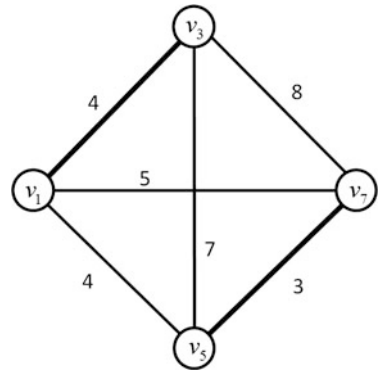


Abb. 5.34 Kürzeste Wege zwischen allen Knoten mit ungeradem Grad mit eingetragenem kostenminimalem Matching

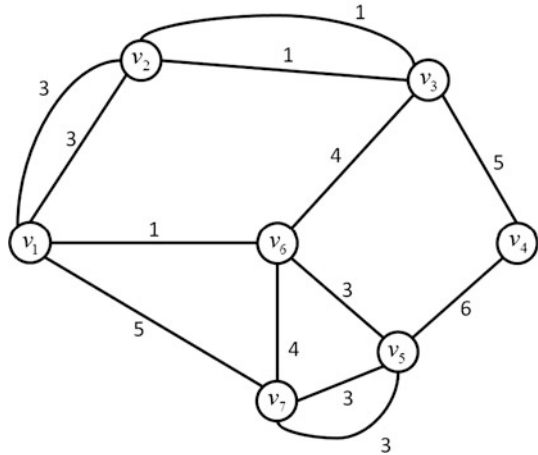


Pfad durch den Graphen, der jede Kante mindestens einmal besucht (siehe Abb. 5.33). Die Berechnung solcher optimalen Routen ist nicht nur für die Postzustellung oder die Müllabfuhr von zentraler Bedeutung.

In dem Fall, dass jeder Knoten des Graphen einen geraden Grad hat, ist jede Euler-Tour eine optimale Briefträgertour. Andernfalls müssen einige Kanten mehr als einmal verwendet werden. In diesem Fall müssen in den Graphen kostenminimale Kanten eingefügt werden, um den Graphen zu einem Euler-Graphen zu machen. Wenn der Graph keinen Euler-Kreis besitzt, hat er eine gerade Anzahl von k Knoten mit ungeradem Knotengrad. Wenn man nun jeweils zwei Knoten ungeraden Knotengrads durch eine zusätzliche Kante verbindet, erhalten wir einen Graphen, bei dem alle Knoten einen geraden Grad besitzen. Jede Euler-Tour in dem so erweiterten Graphen ist dann eine optimale Lösung des Briefträgerproblems.

Für das Hinzufügen dieser $k/2$ kostenminimalen Kanten erstellen wir aus den Knoten mit ungeradem Grad einen vollständigen Graphen, wobei die Kantengewichte den Abstand eines kürzesten Weges zwischen den entsprechenden Knoten im gegebenen Graphen darstellen. In diesem vollständigen Graphen suchen wir dann eine Kantenteilmenge mit genau $k/2$ Kanten, sodass keine zwei Kanten einen gemeinsamen Endknoten besitzen und das Gesamtgewicht minimal ist (siehe Abb. 5.34). Diese Art von Kantenteilmenge

Abb. 5.35 Verdopplung der Kanten mit ungeradem Knotengrad



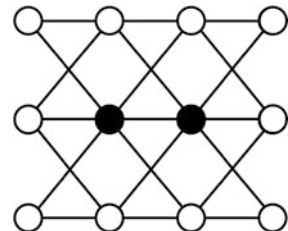
bezeichnet man als kostenminimales Matching. Anschließend ersetzen wir jede Matching-Kante im ursprünglichen Graphen durch die Kanten des entsprechenden kürzesten Weges zwischen den beiden Knoten (siehe Abb. 5.35).

5.6 Spezielle Knotenmengen

5.6.1 Einführendes Beispiel

Wir betrachten im Folgenden den Sitzplan einer Schulklasse. Jeder Knoten entspricht einem Schüler, wobei die sehr guten Schüler durch schwarze Kreise dargestellt sind. Zwei Knoten sind genau dann durch eine Kante verbunden, wenn die entsprechenden Schüler so sitzen, dass sie voneinander abschreiben können. In Abb. 5.36 ist in einem Graphen, der den Sitzplan in der Klasse repräsentiert, eine dominierende Knotenmenge der Mächtigkeit 2 eingezeichnet. In diesem Sitzplan haben sich die sehr guten Schüler so platziert, dass jeder andere Schüler mindestens einen als Nachbarn besitzt. In der Graphentheorie bilden die sehr guten Schüler dann eine dominierende Knotenmenge in dem dazugehörigen Graph.

Abb. 5.36 Knotenüberdeckung durch zwei Knoten



5.6.2 Problemstellung

In einem Graphen lassen sich spezielle Knotenmengen untersuchen, die gewisse Eigenschaften besitzen. Ein Beispiel ist eine unabhängige Knotenmenge U , bei der keine Kante mit irgendeinem anderen Knoten aus U verbunden ist. Diese unabhängigen Mengen besitzen beispielsweise eine Vielfalt von Anwendungsbereichen in der Projektplanung oder bei Zuordnungsproblemen.

Wir betrachten im Folgenden Knotenmengen eines Graphen, die bestimmte Eigenschaften haben. Diese Knotenmengen sind insbesondere für die Definition der Probleme nötig, die wir später algorithmisch lösen wollen.

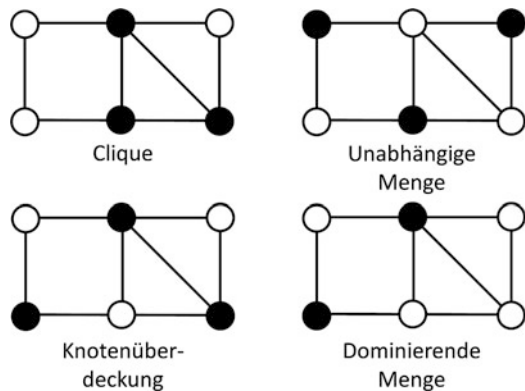
Definition 5.16 Es seien $G = (V, E)$ ein ungerichteter Graph und $U \subseteq V$ eine Teilmenge der Knotenmenge von G . Wir definieren die folgenden Begriffe:

1. U ist eine *Clique* in G , falls $\{u, v\} \in E$ für alle $u, v \in U$ mit $u \neq v$.
2. U ist eine *unabhängige Menge* in G , falls $\{u, v\} \notin E$ für alle $u, v \in U$ mit $u \neq v$.
3. U ist eine *Knotenüberdeckung* in G , falls $\{u, v\} \cap U \neq \emptyset$ für alle Kanten $\{u, v\} \in E$.
4. U ist eine *dominierende Menge* in G , falls es für jeden Knoten $u \in V \setminus U$ einen Knoten $v \in U$ mit $\{u, v\} \in E$ gibt.

Die Größe einer unabhängigen Menge, Clique, Knotenüberdeckung oder einer dominierenden Menge U ist die Anzahl der Knoten in U .

In Abb. 5.37 sind die speziellen Knotenmengen abgebildet. In einer Clique ist der Zusammenhalt der entsprechenden Knoten am größten. Jeder Knoten einer Clique ist mit jedem anderen Knoten der Clique verbunden. Eine Clique eines Graphen G ist ein vollständiger Untergraph von G . Im Gegensatz dazu ist dieser Zusammenhalt in einer unabhängigen Menge nicht vorhanden: Kein Knoten einer unabhängigen Menge U ist mit irgendeinem anderen Knoten aus U verbunden. Eine Knotenüberdeckung eines Graphen

Abb. 5.37 Spezielle Knotenmengen: Clique, unabhängige Menge, Knotenüberdeckung und dominierende Menge



ist eine Knotenmenge, die von jeder Kante des Graphen mindestens einen Endpunkt enthält. Ein Graph wird von einer Teilmenge seiner Knoten dominiert, wenn jeder Knoten des Graphen entweder zu dieser Teilmenge gehört oder einen Nachbarn in ihr hat.

Bei den entsprechenden Problemen ist man an Cliques bzw. an unabhängigen Mengen eines Graphen interessiert, die mindestens eine vorgegebene Größe haben oder die größten Cliques bzw. die größten unabhängigen Mengen. Bei Knotenüberdeckungen bzw. dominierenden Mengen eines Graphen suchen wir nach einer Menge U , die höchstens eine vorgegebene Größe besitzt, oder sogar die kleinsten Knotenüberdeckungen bzw. dominierenden Mengen in diesem Graphen.

Definition 5.17 Wir definieren die folgenden Kennzahlen für einen Graphen $G = (V, E)$:

- Die *Unabhängigkeitszahl* $\alpha(G)$ ist die Größe einer maximalen unabhängigen Knotenmenge von G .
- Die *Cliquenzahl* $\chi(G)$ von G ist die Größe einer größten Clique von G .
- Die *Knotenüberdeckungszahl* $\tau(G)$ ist die Größe einer kleinsten Knotenüberdeckung von G .
- Die *Dominierunugszahl* $\gamma(G)$ ist die Größe einer kleinsten dominierenden Menge von G .

Wir definieren nun die dazugehörigen Entscheidungsprobleme:

UNABHÄNGIGE MENGE (INDEPENDENT SET)

Gegeben: ein ungerichteter Graph $G = (V, E)$ und eine natürliche Zahl $k \leq n$.

Gesucht: Gibt es in G eine unabhängige Menge $U \subset V$ mit $|U| > k$?

CLIQUE (CLIQUE)

Gegeben: ein ungerichteter Graph $G = (V, E)$ und eine natürliche Zahl $k \leq n$.

Gesucht: Gibt es in G eine Clique $U \subset V$ mit $|U| > k$?

KNOTENÜBERDECKUNG (VERTEX COVER)

Gegeben: ein ungerichteter Graph $G = (V, E)$ und eine natürliche Zahl $k \leq n$.

Gesucht: Gibt es in G eine Knotenüberdeckung $U \subset V$ mit $|U| \leq k$?

DOMINIERENDE MENGE (DOMINATING SET)

Gegeben: ein ungerichteter Graph $G = (V, E)$ und eine natürliche Zahl $k \leq n$.

Gesucht: Gibt es in G eine dominierende Menge $U \subset V$ mit $|U| \leq k$?

Alternativ können wir diese Problemstellungen auch als Maximierungsproblem (*UNABHÄNGIGE MENGE*, *CLIQUE*) bzw. Minimierungsproblem (*KNOTENÜBERDECKUNG*, *DOMINIERENDE MENGE*) angeben.

Satz 5.6 Für jeden Graphen $G = (V, E)$ und alle Teilmengen $U \subseteq V$ sind die folgenden drei Aussagen äquivalent:

1. U ist eine unabhängige Menge von G .
2. U ist eine Clique im Komplementgraphen

$$\overline{G} = (V, \{\{u, v\} \mid u, v \in V, u \neq v, \{u, v\} \notin E\}).$$

3. $V \setminus U$ ist eine Knotenüberdeckung von G .

Da diese Problemstellungen alle voneinander abhängen, betrachten wir im Folgenden exemplarisch das Problem der unabhängigen Knotenmenge.

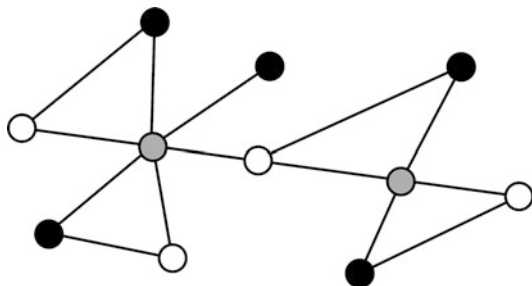
5.6.3 Grundlegende Lösungsprinzipien

In diesem Abschnitt stellen wir einen Algorithmus für die Suche nach einer maximalen unabhängigen Menge.

Definition 5.18 Sei $G = (V, E)$ ein Graph. Eine unabhängige Knotenmenge X von G ist *gesättigt*, wenn keine unabhängige Knotenmenge Y von G existiert, die X echt enthält ($X \subset Y$). Eine *maximale* unabhängige Knotenmenge ist eine unabhängige Knotenmenge maximaler Mächtigkeit von G .

In Abb. 5.38 sind eine gesättigte (grau) und maximale (schwarz) Knotenmenge dargestellt. Die Entwicklung von Algorithmen für das Problem zur Berechnung einer maximalen unabhängigen Menge ist eines der zentralen Probleme in der Graphentheorie. Über viele Jahrzehnte wurden die Algorithmen verbessert, sodass heute der schnellste Algorithmus eine Laufzeit von $O(1,1844^n)$ besitzt. Leider sind diese Algorithmen sehr aufwendig, zum Teil basieren sie auf einer computergenerierten Analyse aller Teilfälle. Wir werden daher einen einfachen rekursiven Algorithmus mit einer Laufzeit von $O(1,38^n)$ vorstellen, um das allgemeine Prinzip zu verstehen. In vielen Anwendungsfällen reichen auch gute und sehr schnelle heuristische Verfahren aus.

Abb. 5.38 Gesättigte (grau) und maximale (schwarz) Knotenmenge



Für den angesprochenen Algorithmus benötigen wir zunächst ein paar Spezialfälle:

Beispiel 5.12

1. Für den vollständigen Graphen K_n ist $\alpha(K_n) = 1$.
2. Für einen Weg P_n ist

$$\alpha(P_n) = \begin{cases} \frac{n}{2}, & \text{falls } n \text{ gerade ist,} \\ \frac{n+1}{2}, & \text{falls } n \text{ ungerade ist.} \end{cases}$$

3. Für einen Kreis C_n ist

$$\alpha(C_n) = \begin{cases} \frac{n}{2}, & \text{falls } n \text{ gerade ist,} \\ \frac{n-1}{2}, & \text{falls } n \text{ ungerade ist.} \end{cases}$$

4. Wenn G aus den Komponenten $G_1, \dots, G_k$ besteht, so gilt

$$\alpha(G) = \sum_{i=1}^k \alpha(G_i).$$

Wir geben nun eine einfache Formel an, um die Unabhängigkeitszahl eines Graphen zu berechnen.

Satz 5.7 *Es seien $G = (V, E)$ ein Graph und $v \in V$. Die Menge aller Nachbarknoten von v in G einschließlich v selbst sei $N_G(v)$. Dann gilt:*

$$\alpha(G) = \max\{\alpha(G_{-v}), \alpha(G_{-N_G(v)}) + 1\},$$

wobei G_{-U} der Graph ist, der aus G entsteht, wenn alle Knoten in der Menge U gelöscht werden.

Beweis Es sei $U \subseteq V$ eine maximale unabhängige Knotenmenge von G mit $\alpha(G) = |U|$. Sei $v \in V$ ein beliebiger Knoten, dann gibt es genau zwei Möglichkeiten:

1. Der Knoten v ist nicht in der unabhängigen Knotenmenge, also ist

$$\alpha(G) = \alpha(G_{-v}).$$

2. Der Knoten v ist in der unabhängigen Knotenmenge, also ist

$$\alpha(G) = \alpha(G_{-N_G(v)}) + 1.$$

□

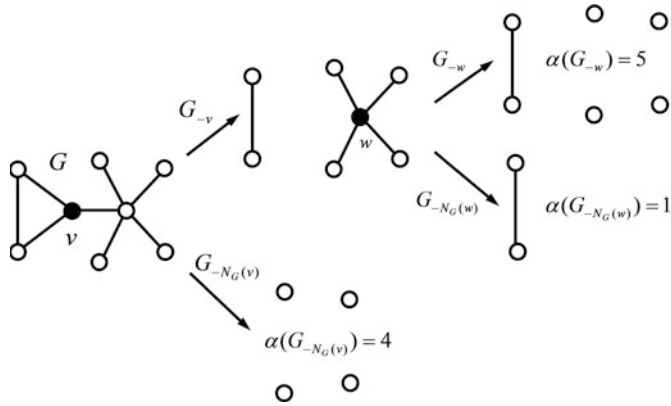


Abb. 5.39 Rekursive Bestimmung der Unabhängigkeitszahl mit Knoten v vom Grad 3

Beispiel 5.13 Wir betrachten den Graphen in Abb. 5.39 zur Berechnung der Unabhängigkeitszahl $\alpha(G)$. Wir wählen den eingezeichneten Knoten v aus und wenden die obige rekursive Formel an. Auf den entstehenden Graphen G_{-v} wird der Knoten w ausgewählt, sodass wir wieder zwei Teilgraphen erhalten. Damit erhalten wir den Wert

$$\begin{aligned}\alpha(G) &= \max\{\alpha(G_{-v}), \alpha(G_{-N_G(v)}) + 1\} \\ &= \max\{\max\{\alpha(G_{-w}), \alpha(G_{-N_G(w)}) + 1\}, 4 + 1\} \\ &= \max\{\max\{5, 1 + 1\}, 5\} = \max\{5, 5\} = 5.\end{aligned}$$

Die obige Formel ist das Kernstück des nachfolgenden Algorithmus zur Bestimmung der Unabhängigkeitszahl $\alpha(G)$.

Prinzip zur Berechnung der Unabhängigkeitszahl

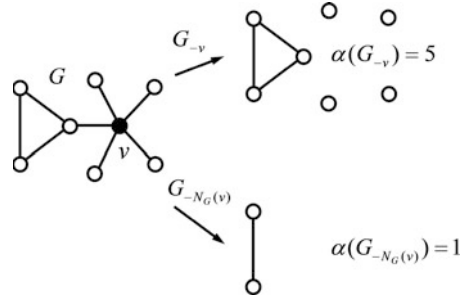
1. Testen ob der maximale Grad $\Delta(G)$ von Graphen G kleiner gleich 2 ist.
Falls ja: Bestimmung aller zusammenhängenden Komponenten $G_1, \dots, G_k$. Für jede Komponente G_i trifft genau einer der folgenden drei Fälle zu:
 - (a) Für einen Graphen K_1 mit nur einem Knoten ist

$$\alpha(K_1) = 1$$

- (b) Für einen Weg P_n ($m = n - 1$) ist

$$\alpha(P_n) = \begin{cases} \frac{n}{2}, & \text{falls } n \text{ gerade ist,} \\ \frac{n+1}{2}, & \text{falls } n \text{ ungerade ist} \end{cases}$$

Abb. 5.40 Rekursive Bestimmung der Unabhängigkeitszahl mit Knoten v vom Grad 5



(c) Für einen Kreis C_n ($m = n$) ist

$$\alpha(C_n) = \begin{cases} \frac{n}{2}, & \text{falls } n \text{ gerade ist,} \\ \frac{n-1}{2}, & \text{falls } n \text{ ungerade ist} \end{cases}$$

Falls G aus den Komponenten $G_1, \dots, G_k$ besteht, so gilt

$$\alpha(G) = \sum_{i=1}^k \alpha(G_i).$$

Falls nein: Anwendung von Schritt zwei

2. Bestimmung eines Knotens v mit maximalem Grad $\Delta(G)$
3. Bestimmung der beiden Graphen G_{-v} und $G_{-N_G(v)}$
4. Rekursive Anwendung der Formel auf die beiden Untergraphen:

$$\alpha(G) = \max\{\alpha(G_{-v}), \alpha(G_{-N_G(v)}) + 1\}$$

Beispiel 5.14 Wir wenden nun diesen Algorithmus auf das obige Beispiel an und erhalten die dargestellten Teilgraphen in Abb. 5.40. Damit erhalten wir den Wert

$$\begin{aligned} \alpha(G) &= \max\{\alpha(G_{-v}), \alpha(G_{-N_G(v)}) + 1\} \\ &= \max\{5, 1 + 1\} = \max\{5, 2\} = 5. \end{aligned}$$

Bei der Auswahl eines Knotens v mit Maximalgrad 5, anstatt wie im vorigen Beispiel mit Grad 3, erhalten wir bereits nach einer Anwendung der rekursiven Formel zwei Teilgraphen mit Maximalgrad 2, die nur isolierte Knoten, Wege oder Kreise beinhalten.

5.6.4 Algorithmus und Implementierung

Für die Angabe des Pseudocodes bezeichnen wir mit **KOMPONENTEN** eine Funktion zur Bestimmung der Menge zusammenhängender Komponenten von G und mit **KNOTEN-MAXGRAD** eine Funktion zur Auswahl eines Knotens mit maximalem Grad.

Algorithmus 11 UNABHÄNGIGKEITSAHL**Input:** Graph $G = (V, E)$ **Output:** Unabhängigkeitszahl $\alpha(G)$

```

1:  $\alpha = 0$ 
2: if  $\Delta(F) \leq 2$  then
3:    $C = \text{KOMPLEMENTEN}(G)$ 
4:   for  $i = 1$  to  $|C|$  do
5:     if  $|V(C_i)| = 1$  then
6:        $\alpha = \alpha + 1$ 
7:     else if  $E(C_i) = V(C_i) - 1$  then
8:        $\alpha = \lfloor \frac{n}{2} \rfloor$ 
9:     else if  $E(C_i) = V(C_i)$  then
10:       $\alpha = \lceil \frac{n}{2} \rceil$ 
11:  $v = \text{KNOTENMAXGRAD}(G)$ 
12:  $\alpha = \max\{\text{UNABHÄNGIGKEITSAHL}(G_{-v}), \text{UNABHÄNGIGKEITSAHL}(G_{-N_G(v)}) + 1\}$ 

```

Allgemeine Erklärung

Die Zeile 2 kennzeichnet die Abbruchbedingung des rekursiven Verfahrens. In Zeile 3 werden alle zugehörigen Komponenten bestimmt. Für alle diese Komponenten wird in Zeile 5–10 die zugehörige Unabhängigkeitszahl berechnet. In Zeile 11 erfolgt die Auswahl eines Knotens mit maximalem Knotengrad. Die Zeile 12 stellt den zweimaligen Aufruf des rekursiven Algorithmus dar.

Aufwandsabschätzung

Wir bezeichnen mit K_n den Rechenaufwand für die Lösung dieses Problems mit n Knoten. Der Algorithmus basiert auf einer Rekursion. Im ersten Fall löschen wir einen Knoten aus dem Graphen. Im zweiten Fall löschen wir mindestens 4 Knoten, da $\Delta(G) \geq 3$ ist. Damit erhalten wir die Rekursionsformel

$$K_n \leq K_{n-1} + K_{n-4} + O(n^2).$$

Die Lösung der Rekursionsformel erfolgt mit dem Ansatz $K_n = \lambda^n$. Damit erhalten wir die charakteristische Gleichung $\lambda^4 - \lambda^3 - 1 = 0$ und die zugehörige Nullstelle von $\lambda = 1,38$. Der Rechenaufwand beträgt damit $O(1,38^n)$.

Die Implementierung des Algorithmus zur Berechnung der Unabhängigkeitszahl erfolgt in der Klasse `Graph` und besteht aus den folgenden drei Teilfunktionen:

1. Bestimmung einer Kantenfolge vom Startknoten v :

```

public Vector<Integer> bestimmePfad(int v)
{
    Vector<Integer> p = new Vector<Integer>();
    p.add(v);
    do // Solange wie ausgehende Kante vorhanden
    {
        for(int i=0; i<n; i++)

```

```

    {
        if (A[v][i] > 0) // Kante (v,i) vorhanden
        {
            p.add(i);
            this.loeschenKante(v, i);
            v = i;
            break;
        }
    }
}
while(this.getKnotengrad(v) > 0);
return p;
}

```

2. Bestimmung aller zusammenhängenden Komponenten des Graphen mit $\Delta(G) \leq 2$:

```

public Vector<Graph> bestimmeKomponenten()
{
    // --- 1. Anlegen der Datenstruktur aus Graphen
    Vector<Graph> C = new Vector<Graph>();
    Graph H = this;
    BitSet b = new BitSet(); // Knotenstatus
    for(int i=0; i<H.getKnotenanzahl(); i++)
        b.set(i);
    for(int v=0; v<b.length(); v++)
    {
        // --- 2. Knoten v ist isoliert und frei
        if (H.getKnotengrad(v) == 0 && b.get(v))
        {
            Graph G = new Graph(1, false);
            C.add(G);
        }
        // --- 3. Knoten v hat eine oder zwei Kanten und ist frei
        else if (H.getKnotengrad(v) > 0 && b.get(v))
        {
            // --- 3a. Bestimmung einer Kantenfolge von Knoten v
            Vector<Integer> p = H.bestimmePfad(v);
            int anz; // Anzahl der Knoten des Pfades
            if (p.firstElement() == p.lastElement())
                anz = p.size() - 1; // Kreis
            else
                anz = p.size(); // Weg
            // --- 3b. Graph definieren
            Graph G = new Graph(anz, false);
            for(int i=0; i<p.size()-1; i++)
            {
                G.addKante(p.elementAt(i)%anz, p.elementAt(i+1)%anz, 1);
                b.clear(p.elementAt(i));
                b.clear(p.elementAt(i+1));
            }
            C.add(G);
        }
    }
    return C;
}

```

3. Berechnung der Unabhängigkeitszahl $\alpha(G)$:

```

public double berechneUnabhängigkeitszahl()
{
    double alpha = 0;
    if(getMaxGrad() <= 2)
    {
        // --- 1. Bestimmung aller zusammenhängenden Komponenten
        Vector<Graph> c = bestimmeKomponenten();
        for(int i=0; i<c.size(); i++)
        {
            Graph G = c.elementAt(i);
            G.ausgabeMatrix();
            int n = G.getKnotenanzahl();
            int m = G.getKantenanzahl();
            if (G.getMaxGrad() == 0) // isolierter Knoten
                alpha = alpha + 1;
            else if (m == n-1) // Weg
                alpha = Math.round(n/2.0);
            else if (m == n) // Kreis
                alpha = n/2;
        }
    }
    else
    {
        // --- 1. Auswahl eines Knotens mit maximalem Grad
        int v = getKnotenMaxGrad();

        // --- 2. Knoten v ist nicht in der max. unabh. Menge
        Graph G1 = new Graph(A, gerichtet);
        Vector<Integer> V1 = new Vector<Integer>();
        V1.addElement(v);
        G1.loescheKnotenmenge(V1); // Lösche v aus G
        double alpha1 = G1.berechneUnabhängigkeitszahl();

        // --- 3. Knoten v ist in der max. unabh. Menge
        Graph G2 = new Graph(A, gerichtet);
        Vector<Integer> V2 = this.getNachbarn(v);
        V2.addElement(v);
        G2.loescheKnotenmenge(V2); // Lösche N_G(v) und v aus G
        double alpha2 = G2.berechneUnabhängigkeitszahl();

        // --- 4. Bestimmung der Unabhängigkeitszahl
        alpha = Math.max(alpha1, alpha2 + 1);
    }
    return alpha;
}

```

5.6.5 Anwendungen

Wir betrachten einige Anwendungen von diesen speziellen Knotenmengen aus der Graphentheorie.

Auftragsplanung Eine Firma soll verschiedene Aufträge $A_1, \dots, A_n$ ausführen, für die jeweils gewisse Maschinen aus der Menge $\{M_1, \dots, M_k\}$ benötigt werden. Zwei Aufträge können genau dann gleichzeitig ausgeführt werden, wenn die Aufträge nicht dieselbe Maschine erfordern. Die Frage lautet nun: Wie viele Aufträge können maximal gleichzeitig ausgeführt werden?

Dieses Problem lösen wir mithilfe eines Graphen $G = (V, E)$. Die Knoten V des Graphen sind die Aufträge $A_1, \dots, A_n$. Zwei Aufträge werden genau dann durch eine Kante verbunden, wenn die beiden Aufträge eine Maschine gemeinsam erfordern. Aus dem Modell folgt, dass zwei Aufträge genau dann gleichzeitig ausführbar sind, wenn die zugeordneten Knoten im Graphen nicht adjazent sind. Wir suchen damit eine maximale unabhängige Knotenmenge von G . Die Anzahl der Aufträge, die maximal gleichzeitig ausgeführt werden können, ist somit $\alpha(G)$.

Kommunikationsnetz Dominierende Mengen spielen in vielen praktischen Anwendungsbereichen eine wichtige Rolle, so etwa im Zusammenhang mit Computer- oder Kommunikationsnetzen. Eine Sendergruppe in einem Kommunikationsnetz ist eine Teilmenge der Knoten, die jeden anderen Knoten in einem Schritt erreicht. Eine Sendergruppe bildet somit eine dominierende Menge in dem Netz.

Eine Sendergruppe kann nun ganz oder teilweise ausfallen, sodass nicht mehr alle Knoten des Netzwerks Sendungen empfangen können. Nützlich sind dann alternative Sendergruppen im Netz, die zu den anderen disjunkt sind. Die domatische Zahl gibt die maximale Anzahl disjunkter Sendergruppen an, die das gegebene Netz in dominierende Mengen partitionieren.

5.7 Färbungen von Knoten

5.7.1 Einführendes Beispiel

Die Vergabe von Frequenzen im Mobilfunk wird durch verschiedene staatliche Behörden geregelt. Die Betreiber können einzelne oder mehrere Frequenzen nutzen. Das gesamte Frequenzband unterteilt sich in eine Menge von Kanälen, die alle jeweils die gleiche Bandbreite besitzen. Damit steht einem Betreiber dann eine Anzahl von Übertragungskanälen zur Verfügung. Beim gleichzeitigen Senden von zwei verschiedenen Standorten auf dem gleichen Kanal entsteht Interferenz, welche die Qualität des Signals beim Empfänger verschlechtert oder es unbrauchbar macht.

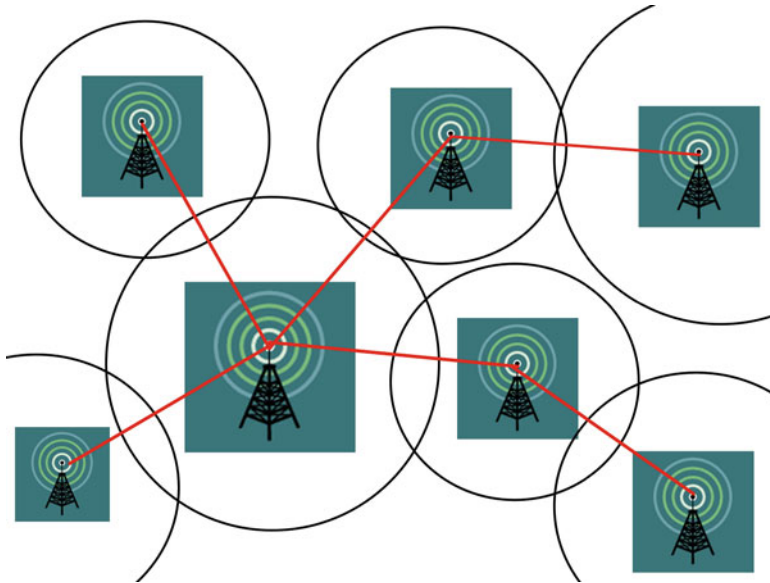


Abb. 5.41 Mobilfunknetz und zugehöriger Interferenzgraph

Falls ein Mobilfunkbetreiber an n Standorten Antennen positioniert hat, muss er jedem Standort eine Frequenz aus den ihm verfügbaren Mengen von k Frequenzen zuweisen, sodass die Interferenz überall unter einer gewissen Schwelle bleibt. Wir verbinden zwei Standorte mit einer Kante, wenn wegen entstehender Interferenz nicht die gleiche Frequenz zugeteilt werden darf (siehe Abb. 5.41). Wir erhalten einen ungerichteten Graphen, den sogenannten Interferenzgraphen für die Frequenzplanung. In diesem Graphen suchen wir eine Zuweisung von Frequenzen an die Knoten, sodass durch eine Kante verbundene Knoten verschiedene Frequenzen erhalten. In der Graphentheorie spricht man hier von einem Färbungsproblem: Der Interferenzgraph soll mit k Farben in der Form gefärbt werden, dass benachbarte Knoten unterschiedliche Farben bekommen.

5.7.2 Problemstellung

Eine Färbung eines ungerichteten Graphen ordnet jedem Knoten im Graphen eine Farbe zu. Ein Spezialfall ist der Vier-Farben-Satz, das jeder planare Graph (Graph, der ohne Kantenüberkreuzung in der Ebene darstellbar ist) mit höchstens 4 Farben färbbar ist. Der Vier-Farben-Satz besagt ebenfalls, dass 4 Farben immer ausreichen, um eine beliebige Landkarte so einzufärben, dass keine zwei angrenzenden Länder die gleiche Farbe bekommen.

Der Satz wurde erstmals 1852 von Francis Guthrie als Vermutung aufgestellt, als er in einer Karte die Grafschaften von England färben wollte. In den folgenden Jahrzehnten wurden zahlreiche Beweise der Vier-Farben-Vermutung veröffentlicht, die sich jedoch alle als fehlerhaft erwiesen. Heinrich Heesch entwickelte in den 1960er- und 1970er-Jahren einen ersten Entwurf eines Computerbeweises.

Definition 5.19 Eine *Färbung* eines Graphen $G = (V, E)$ ist die Zuordnung von Farben für alle Knoten des Graphen. Eine Färbung heißt *zulässig*, wenn keine adjazenten Knoten in G die gleiche Farbe besitzen. Eine zulässige k -Färbung ist eine Abbildung $\varphi : V \rightarrow \{1, \dots, k\}$, wobei $\varphi(v) \neq \varphi(w)$ für alle $v, w \in V$ mit $(v, w) \in E$.

Definition 5.20 Die *chromatische Zahl* $\chi(G)$ ist die minimale Anzahl von Farben, für die der Graph G eine zulässige Färbung besitzt.

Beispiel 5.15

1. Für den vollständigen Graphen K_n ist $\chi(K_n) = n$.
2. Für einen Baum T_n ist $\chi(T_n) = 2$.
3. Für einen Kreis C_n ist

$$\chi(G) = \begin{cases} 2, & \text{falls } n \text{ gerade ist,} \\ 3, & \text{falls } n \text{ ungerade ist.} \end{cases}$$

4. Für einen bipartiten Graphen K_{ab} ist $\chi(K_{ab}) = 2$.

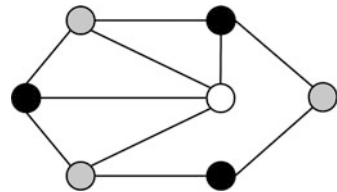
Für die chromatische Zahl gilt $1 \leq \chi(G) \leq n$. Falls der Graph G aus den Komponenten $G_1, \dots, G_c$ besteht, so gilt:

$$\chi(G) = \max_{1 \leq i \leq c} \chi(G_i).$$

Definition 5.21 Eine *Partition* eines Graphen $G = (V, E)$ ist eine Zerlegung der Knotenmenge V in paarweise disjunkte Teilmengen (*Blöcke*), deren Vereinigung wieder die Menge V ist. Eine *unabhängige Partition* ist eine Partition von G , in der die Blöcke unabhängige Mengen von G sind.

In Abb. 5.42 ist ein Graph durch eine Partition mit 3 Blöcken gefärbt. Eine Färbung der Knotenmenge erzeugt eine Partition, wobei die Knoten einer Farbe jeweils einen Block

Abb. 5.42 Färbung eines Graphen durch eine Partition mit 3 Blöcken



der Partition definieren. Ein Graph $G = (V, E)$ ist genau dann k -färbbar, wenn V in k Mengen $U_1, \dots, U_k$ der folgenden Form aufgeteilt werden kann:

1. Die Vereinigung aller Partitionen ist die Knotenmenge V : $\bigcup U_i = V$.
2. Die einzelnen Mengen sind disjunkt: $U_i \cap U_j = \emptyset$, $\forall i, j \in \{1, \dots, k\}$, $i \neq j$.
3. U_i ist eine unabhängige Menge.

Das Problem der Knotenfärbung kann man dann mithilfe von Partitionen wie folgt formulieren:

FÄRBUNG

Gegeben: ein ungerichteter Graph $G = (V, E)$ und eine natürliche Zahl $k \leq n$.

Gesucht: Gibt es eine Partition von V in k unabhängige Mengen?

5.7.3 Grundlegendes Lösungsprinzipien

Im Folgenden geben wir verschiedene Verfahren zur Bestimmung der Graphenfärbung eines Graphen an.

Färbung mittels Backtracking Mit Hilfe des bekannten algorithmischen Entwurfsmusters des Backtrackings geben wir einen Algorithmus an, der zu einem gegebenen Graph eine Färbung mit k Farben bestimmt. Hierbei wird versucht, die Knoten der Reihe nach im Graphen so zu färben, dass eine zulässige Färbung mit k Farben entsteht. In jedem Schritt wird eine Färbung für einen Knoten v_i so bestimmt, dass die Farbe mit der Färbung der vorhergehenden Knoten verträglich ist, also dass keine zwei benachbarten Knoten die gleiche Farbe besitzen. Dazu werden alle Farben von 1 bis k für den Knoten v_i ausprobiert. Falls keine passende Farbe für den Knoten v_i gefunden wurde, so wendet man das Backtracking an und geht einen Schritt zurück, d. h., man ändert die Farbe des Knotens v_{i-1} . Anschließend beginnen die beschriebenen Schritte von vorn.

Die Bestimmung der Farben erfolgt in der Reihenfolge der Knotennummerierung. Auch die Wahl der Farben beginnt immer mit der Farbe 1 und wird anschließend bis zur Zahl k erhöht. Das Verfahren terminiert, wenn entweder der letzte Knoten zulässig gefärbt wurde oder keine zulässige Färbung mit k Farben existiert. Durch Aufruf des Algorithmus mit kleiner werdenden Zahlen k kann die chromatische Zahl leicht bestimmt werden.

Färbung mittels dynamischer Programmierung Einer der ältesten Graphfärbungsalgorithmen ist der Algorithmus von Lawler. Der Algorithmus von Lawler ermittelt mittels dynamischer Programmierung die chromatische Zahl eines Graphen. Die Idee des Algorithmus besteht darin, dass unter allen minimalen zulässigen Färbungen des gegebenen Graphen $G = (V, E)$ wenigstens eine sein muss, die eine maximale unabhängige Menge als Farbklasse enthält. Eine unabhängige Menge heißt maximal, falls sie die Eigenschaft

der Unabhängigkeit durch Hinzunahme eines beliebigen Knotens verlieren würde. Betrachte eine nicht leere Teilmenge $U \subset V$. Hat man nun die chromatische Zahl aller induzierter Teilgraphen $G[U']$ für jede echte Teilmenge $U' \subset U$ bereits bestimmt, so lässt sich die chromatische Zahl des induzierten Teilgraphen $G[U]$ mit der Formel

$$\chi(G[U]) = 1 + \min_W \{\chi(G[U \setminus W])\}$$

bestimmen, wobei über alle maximal unabhängigen Mengen $W \subseteq U$ minimiert wird.

5.7.4 Algorithmus und Implementierung

Wir zeigen nun die Implementierung des Backtrackingalgorithmus. Die Unterroutine `PRUEFEFAERBUNG( $G, v, f, \text{farbe}$ )` prüft hierbei, ob der Knoten v im Graphen G der Färbung f mit dem Farbwert farbe gefärbt werden kann.

Algorithmus 12 FÄRBUNG

Input: Graph $G = (V, E)$, Färbungszahl k , Knoten v , Knotenfärbung $f : V \rightarrow \{1, \dots, k\}$

Output: 1, falls c -Färbung existiert; 0 sonst

```

1: farbe = 0
2: repeat
3:   farbe = farbe + 1
4:   b = false
5:   if PRUEFEFAERBUNG( $G, v, f, \text{farbe}$ ) then
6:      $f(v) = \text{farbe}$ 
7:     if  $i < n$  then
8:        $b = \text{FÄRBUNG}(G, k, v + 1, f)$ 
9:       if  $b = \text{false}$  then
10:         $f(v) = 0$ 
11: until  $b = \text{true}$  or farbe =  $k$ 
```

Allgemeine Erklärung

Der Aufruf des Algorithmus erfolgt mit der Knotenfärbung f als Nullvektor. In Zeile 5 wird geprüft, ob eine Knotenfärbung für v mit farbe existiert. Falls ja, wird diese in Zeile 6 gesetzt. In Zeile 8 erfolgt der rekursive Aufruf der Prozedur mit der nächsthöheren Knotennummer. Der Algorithmus terminiert, wenn entweder der letzte Knoten erfolgreich gefärbt wurde oder keine Knotenfärbung mit k Farben möglich ist.

Die Implementierung in der Klasse Graph wird wie folgt umgesetzt:

```
public boolean bestimmeFaerbung(int v, int f[], int c)
{
    int farbe = 0;
    boolean b;
    do
    {
        farbe = farbe + 1;
        b = false;
        if(pruefeFaerbung(v, f, farbe))
        {
            f[v] = farbe;
            LinAlgebra.ausgabe(f);
            if(v < n - 1)
            {
                b = bestimmeFaerbung(v+1, f, c);
                if(b == false)
                    f[v] = 0;
            }
            else
                b = true;
        }
    }
    while((b == false) && (farbe < c));
    return b;
}

public boolean pruefeFaerbung(int v, int f[], int farbe)
{
    Vector<Integer> NG = getNachbarn(v);
    for(int i=0; i<NG.size(); i++)
        if(f[NG.elementAt(i)] == farbe)
            return false;
    return true;
}
```

5.7.5 Anwendung

Wir betrachten einige Anwendungen des Färbungsproblems.

Frequenzplanung Eine Sendestation eines Funknetzes überdeckt jeweils ein bestimmtes Gebiet, dargestellt in Abb. 5.43. Es kann vorkommen, dass einige Gebiete von mehreren Sendestationen überdeckt werden. Falls zwei solche Stationen die gleiche Frequenz benutzen, so kommt es zu Störungen. Um diese Störungen zu vermeiden, sollten alle

Abb. 5.43 Ausbreitungsbe-
reich eines Funknetzes

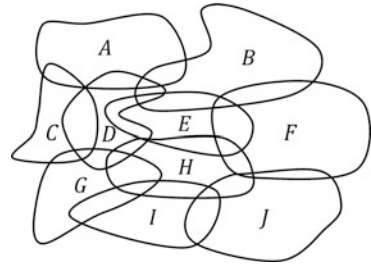
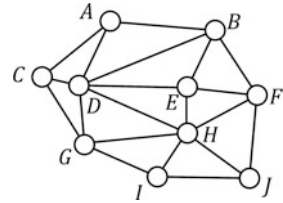


Abb. 5.44 Modellgraph für
Funknetz



Senderstationen, die ein gemeinsames Gebiet erreichen, unterschiedliche Frequenzen benutzen. Frequenzen sind im Allgemeinen sehr teuer, daher will man eine minimale Anzahl von verschiedenen Frequenzen nutzen. Die Frage lautet nun: Wie viele verschiedene Frequenzen sind für einen störungsfreien Betrieb des Funknetzes erforderlich?

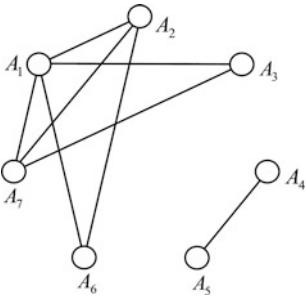
Dieses Problem lösen wir mithilfe von Färbungen in einem Graphen $G = (V, E)$. Die Knoten V des Graphen sind die einzelnen Sendestationen $A, B, C, \dots, J$. Zwei Sender werden genau dann durch eine Kante verbunden, wenn sich ihre Sendebereiche überlappen, dargestellt in Abb. 5.44. Die minimale Anzahl der Frequenzen ist dann die chromatische Zahl $\chi(G)$.

Im praktischen Einsatz sind weitere Anforderungen zu erfüllen wie beispielsweise der Mindestabstand der Frequenzen, die Vorgabe einer Liste von erlaubten Frequenzen oder der Frequenzabstand für Sendestationen, die nicht benachbart sind.

Auftragsplanung Eine Firma soll verschiedene Aufträge $A_1, \dots, A_n$ ausführen, für die jeweils gewisse Maschinen aus der Menge $\{M_1, \dots, M_k\}$ benötigt werden. Zwei Aufträge können genau dann gleichzeitig ausgeführt werden, wenn die Aufträge nicht dieselbe Maschine benötigen. Für jeden Auftrag wird eine gewisse konstante Zeiteinheit eingeplant. Die Frage lautet nun: Wie viel Zeit ist mindestens für die Erledigung aller Aufträge erforderlich?

Dieses Problem lösen wir ebenfalls mithilfe von Färbungen in einem Graphen $G = (V, E)$. Die Knoten V des Graphen sind die Aufträge $A_1, \dots, A_n$. Zwei Aufträge werden genau dann durch eine Kante verbunden, wenn die beiden Aufträge eine Maschine gemeinsam erfordern. Der entstandene Graph besitzt die chromatische Zahl $\chi(G)$. Das heißt, jede unabhängige Partition des Graphen G hat mindestens $\chi(G)$ Blöcke. Die Aufträge können genau dann gleichzeitig ausgeführt werden, wenn sie eine unabhängige Menge

Abb. 5.45 Modellgraph für Auftragsplanung



in G bilden. Die niedrigste Gesamtzeit wird benötigt, wenn die Knotenmenge von G in eine minimale Anzahl unabhängiger Mengen zerlegt wird. Damit ist die chromatische Zahl $\chi(G)$ die gesuchte Zeit für die Erledigung aller Aufträge.

Wir betrachten als Beispiel das folgende Auftragsplanungsproblem:

	M_1	M_2	M_3	M_4	M_5	M_6
A_1			x	x	x	
A_2		x		x		
A_3			x			x
A_4	x					
A_5	x					
A_6		x			x	
A_7			x	x		

Jeder Auftrag benötigt genau 1 h. Mit der obigen Modellierungsvorschrift erhalten wir den folgenden Graphen in Abb. 5.45. Dieser Graph besitzt die chromatische Zahl $\chi(G) = 3$, d. h., 3 h sind erforderlich. Eine Färbung $f : \{A_1, \dots, A_7\} \rightarrow \{1, 2, 3\}$ lautet wie folgt: $f(A_1) = f(A_4) = 1$, $f(A_2) = f(A_3) = f(A_5) = 2$ und $f(A_6) = f(A_7) = 3$.

5.8 Übungsaufgaben

Aufgabe 5.1 (Graphen) Gegeben seien die folgenden Graphen $G_1 = (V_1, E_1)$ mit

$$V_1 = \{v_1, v_2, v_3, v_4, v_5\}, \quad E_1 = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_3\}, \{v_3, v_4\}, \{v_4, v_5\}, \{v_2, v_5\}\}$$

und Graphen $G_2 = (V_2, E_2)$ mit

$$V_1 = \{v_1, v_2, v_3, v_4\}, \quad E = \{(v_1, v_2), (v_4, v_4), (v_1, v_4), (v_4, v_3), (v_1, v_3), (v_3, v_1)\}.$$

Bearbeiten Sie die folgenden Aufgaben:

- a) Zeichnen Sie die zugehörigen Graphen.
- b) Bestimmen Sie die zugehörige Adjazenzmatrix A , Gradmatrix G , Admittanzmatrix L und Abstandsmatrix D .
- c) Beantworten Sie zu den Graphen in Aufgabe 5.1 die folgenden Aufgaben:
 - 1) Ist der Graph schlicht?
 - 2) Bestimmen Sie $N_G(v_1)$.
 - 3) Bestimmen Sie $N_G(\{v_2, v_3\})$.
 - 4) Bestimmen Sie den Minimalgrad $\delta(G)$.
 - 5) Bestimmen Sie den Maximalgrad $\Delta(G)$.
 - 6) Bestimmen Sie einen aufspannenden Untergraphen.
 - 7) Bestimmen Sie einen Kantenzug, Weg und Kreis.

Aufgabe 5.2 (Graphoperationen) Erweitern Sie die Klasse `Graph` in der Matrizendarstellung zur Implementierung der folgenden vier Graphoperationen:

- `entfernenKante`: Entfernen einer Kante $e \in E$: $G_{-e} = (V, E \setminus \{e\})$.
- `entfernenKnoten`: Entfernen eines Knotens $v \in V$: G_{-v} entsteht aus G durch Entfernung von v und aller zu v inzidenten Kanten.
- `FusionKnoten`: Fusion von zwei Knoten u und v : G_{uv} entsteht aus G durch Verschmelzung von u und v .
- `KontraktionKante`: Kontraktion einer Kante $e = \{u, v\}$: G_e entsteht aus G durch Entfernung von e mit der anschließenden Fusion von u und v .

Verwenden Sie diese Methoden zur Implementierung zweier Methoden zum Entfernen von Kantenmengen und Knotenmengen.

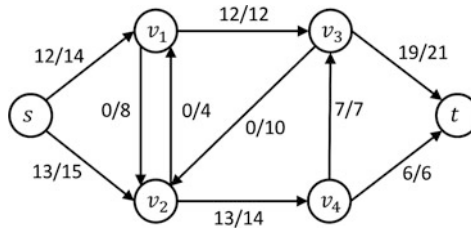
Aufgabe 5.3 (Gerüste) Implementieren Sie den Algorithmus zur Berechnung der Gerüste eines Graphen. Testen Sie das Verfahren an dem folgenden Beispiel: Gegeben sei der Graph $G = (V, E)$ mit $V_1 = \{v_1, v_2, v_3, v_4\}$ und $E = \{\{v_1, v_2\}, \{v_1, v_4\}, \{v_2, v_3\}, \{v_2, v_4\}\}$. Bestimmen Sie die Anzahl der Gerüste von G .

Aufgabe 5.4 (Matrizendarstellung) Erstellen Sie eine Klasse zur Berechnung der Adjazenzmatrix, Inzidenzmatrix, Gradmatrix und Admittanzmatrix eines Graphen.

Aufgabe 5.5 (Listendarstellung) Implementieren Sie die Klasse `Graph` in der Listendarstellung mit allen beschriebenen Methoden analog der Matrizendarstellung.

Aufgabe 5.6 (Elektrisches Netz) Erstellen Sie eine Klasse `ENetz` zur Berechnung der Spannungen und Ströme in einem elektrischen Netz.

Aufgabe 5.7 (Flüsse in Netzwerken) Gegeben sei das folgende Flussnetz $G = (V, E)$ mit der Quelle s , Senke t und der Kapazität c sowie einem Fluss f (Schreibweise f/c):



- Zeigen Sie, dass f ein zulässiger Fluss ist.
- Berechnen Sie den Betrag des Flusses f und geben Sie eine obere Schranke für den maximalen Fluss an.
- Berechnen Sie das dazugehörige Restnetz G_f .
- Verbessern Sie den Fluss f mit dem Erweiterungspfad p von s nach t und geben Sie den Betrag des maximalen Flusses $f_{\max}$ an.

Aufgabe 5.8 (Simulated Annealing) Implementieren Sie in das vorgestellte Schablonenmuster des Simulated Annealing passende konkrete Klassen zur heuristischen Berechnung von größtmöglichen unabhängigen Knotenmengen und zur Bestimmung einer kleinstmöglichen zulässigen Färbung eines gegebenen Graphen.

Literaturhinweise¹

- Tittmann, P. (2011). *Graphentheorie*. Hanser.
- Turau, V. (2009). *Algorithmische Graphentheorie*. Oldenbourg.
- Krumke, S.O., Noltemeier, H. (2012). *Graphentheoretische Konzepte und Algorithmen*. Springer Vieweg.
- Schöning, U. (2001). *Algorithmik*. Spektrum.
- Gurski, F., Rothe, I., Rothe, J., Wanke, E. (2010). *Exakte Algorithmen für schwere Graphenprobleme*. Springer.
- Korte, B., Vygen, J. (2012). *Kombinatorische Optimierung*. Springer Spektrum.
- Lang, H.W. (2012). *Algorithmen in Java*. Oldenbourg.
- Saake, G., Sattler, K-U. (2012). *Algorithmen und Datenstrukturen (in Java)*. dpunkt.
- Gross, J., Yellen, J. (2004). *Handbook of Graph Theory*. CRC Press.
- Gross, J., Yellen, J. (1999). *Graph Theory and its Applications*. CRC Press.

¹ Einen sehr guten Einstieg in das Thema Graphentheorie bietet das kompakte Buch [1]. Weiterführende Lehrbücher zur algorithmischen Graphentheorie sind [2], [3] oder [4]. Einen Überblick über Algorithmen für NP-schwere Graphenprobleme findet der Leser in [5]. Viele Graphenprobleme lassen sich auch in den Bereich der kombinatorischen Optimierung einordnen, hierzu ist das Buch [6] sehr zu empfehlen. Ein Kapitel zu Graphalgorithmen und Java findet man beispielsweise auch in den beiden Lehrbüchern [7] und [8]. Ein ausgezeichnetes umfassendes Handbuch zur Graphentheorie bietet das englischsprachige Buch [9]. Von den beiden Autoren gibt es auch noch das Lehrbuch [10] zu Anwendungen zur Graphentheorie.

Automaten sind ereignisdiskrete Systeme, die sich durch gerichtete Graphen darstellen lassen. Die Knoten entsprechen dabei den Zuständen und die gerichteten Kanten den möglichen Übergängen. Falls nur endlich viele Zustände existieren, spricht man von endlichen Automaten. Ein möglicher Prozess in einem Automaten ist ein Pfad von einem Anfangszustand z_0 zu einem Zustand z_j in diesem Graphen.

Die Automaten haben eine große Bandbreite von Anwendungsbereichen zur Beschreibung des Verhaltens von gesteuerten diskreten Systemen und zur Definition formaler Sprachen. Das Ziel ist die Modellierung und Analyse von technischen Aufgabenstellungen mittels Automaten. Zu nennen sind hierbei der Entwurf von Compilern und digitalen Schaltkreisen, die Entwicklung von Dialogsystemen von Mensch-Maschine-Schnittstellen, die Automatisierung und Steuerung von Prozessen oder die Entwicklung von Software zum Durchsuchen von umfangreichen Texten.

In diesem Kapitel betrachten wir vier verschiedene Arten von Automatenmodellen: Das erste und zweite Modell sind die endlichen Automaten bzw. die Eingabe/Ausgabe-Automaten, die in vielen praktischen Anwendungen eine große Rolle spielen. Das dritte Modell sind die nicht deterministischen Automaten, die zu den sogenannten stochastischen Markov-Modellen erweiterbar sind. Das vierte Modell sind Grammatiken, eine alternative bzw. erweiterte Darstellungsmöglichkeit von endlichen Automaten.

6.1 Deterministische Automaten

6.1.1 Einführende Beispiele

Deterministische Automaten sind die einfachste Darstellungsform von ereignisdiskreten Systemen. Wir betrachten zunächst einfache Anwendungsbeispiele für deterministische Automaten. Zur Beschreibung der Funktionsweise von speziellen Automaten wie Fahr-

kartenautomaten, Getränkeautomaten usw. ist die Modellbildung mittels endlicher Automaten sehr hilfreich.

Kaffeemaschine Bei der Benutzung einer Kaffeemaschine sind die folgenden Aktionen bzw. Ereignisse zu berücksichtigen:

- W: Wasser einfüllen;
- F: Filter einlegen;
- K: Kaffeepulver einfüllen;
- S: Strom einschalten.

Ein korrektes Ereignis oder eine korrekte Eingabefolge in dem Automaten ist damit WFKS, FKWS oder FWKS. Wir sagen dazu, dass der Automat diese Folge akzeptiert. Eine nicht korrekte Eingabefolge wäre beispielsweise WFS.

Der Anfangszustand des Kaffeeautomaten ist damit wie folgt beschrieben: Wasserbehälter leer, kein Filterpapier eingelegt, kein Kaffeepulver vorhanden, Strom abgeschaltet. Der Endzustand des Automaten ist der folgende Zustand: Wasserbehälter voll, Filterpapier eingelegt, Kaffeepulver drin, Strom angeschaltet.

Fahrkartenautomat Aus einem einfachen Fahrkartenautomat erhält man nach Einwurf von exakt EUR 3,00 durch ein oder zwei Euro Münzen und dem Drücken des Ausgabeknopfes einen Fahrschein. Der Fahrkartenautomat lässt sich durch die folgenden drei Ereignisse beschreiben:

- 1: Einwurf von EUR 1,00;
- 2: Einwurf von EUR 2,00;
- A: Drücken des Ausgabeknopfes.

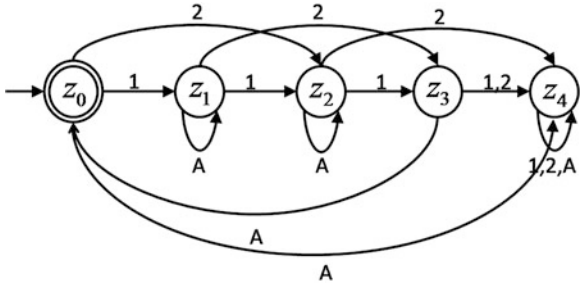
Die Menge der Ereignisse bezeichnet man mit $\Sigma = \{1, 2, A\}$, dem sogenannten Eingabealphabet. Der Automat lässt sich nun durch 5 Zustände

$$Z = \{z_0, z_1, z_2, z_3, z_4\}$$

beschreiben, die den Wert der eingeworfenen Münzen zählen. Der Zustandsübergang beschreibt eine Funktion $\delta : Z \times \Sigma \rightarrow Z$ mit den folgenden Einträgen:

δ	1	2	A
z_0	z_1	z_2	z_4
z_1	z_2	z_3	z_1
z_2	z_3	z_4	z_2
z_3	z_4	z_4	z_0
z_4	z_4	z_4	z_4

Abb. 6.1 Zustandsgraph des Fahrkartenautomaten



In Abb. 6.1 ist der Zustandsgraph des zugehörigen Automaten dargestellt. Zu Beginn ist der Automat im Anfangszustand z_0 , d. h., Geld wurde bisher nicht eingeworfen. Der Automat akzeptiert die Eingabefolge falls genau EUR 3,00 eingeworfen und am Ende der Ausgabeknopf gedrückt wurden. Eine korrekte Eingabefolge ist beispielsweise 111A, 12A, 21A, 11A1A, A1A2A. Mit dieser Eingabe befindet sich dann der Automat in dem Endzustand $F = \{z_0\}$. Bei anderen Wörtern wie 11A, 112A akzeptiert er die Folge nicht und landet im Zustand z_4 .

Zahlendarstellung Die Zahlendarstellung in Java oder anderen Programmiersprachen sieht wie folgt aus:

102, -13,67, 2,51E4, 0,0234E-3.

Unzulässig sind beispielsweise die folgenden Zahlen:

.5, 5, 233, E3, 5.E-3.

Das Eingabealphabet eines endlichen Automaten zur Erkennung einer korrekten Zahlendarstellung lautet dann wie folgt:

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, ., E\}.$$

Ein spezieller Automat zur Darstellung von beliebigen ganzen Zahlen (Datentyp `int` oder `long`) ist in Abb. 6.2 dargestellt. Der Automat zur Darstellung von beliebigen reellen Zahlen (Datentyp `double` oder `float`) ist in Abb. 6.3 modelliert.

Abb. 6.2 Zustandsgraph zur Darstellung ganzer Zahlen

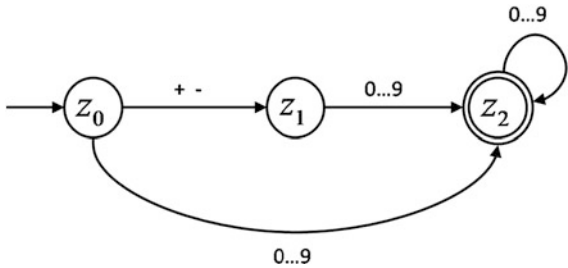
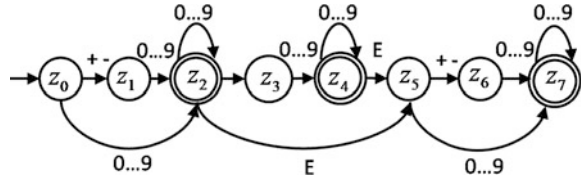


Abb. 6.3 Zustandsgraph zur Darstellung von beliebigen reellen Zahlen



Jeder dieser Automaten kann in ein spezielles Computerprogramm umgesetzt werden. Vor allem im Compilerbau für den Entwurf von Programmiersprachen finden derartige Programme große Anwendung.

6.1.2 Grundlegende Begriffe

Wir definieren zunächst die grundlegenden Begriffe wie Buchstaben, Wörter und Sprache. Anschließend stellen wir das Modell eines Automaten vor und zeigen dessen Darstellungsformen auf.

Definition 6.1 Es sei Σ eine endliche Menge, die man als *Alphabet* bezeichnet. Die Elemente von Σ heißen *Buchstaben*. Ein *Wort* w über Σ ist eine Folge

$$w = w_1 w_2 \cdots w_n$$

von Buchstaben $w_i \in \Sigma$. Die Anzahl $|w|$ der Buchstaben eines Wortes w heißt *Länge* von w . Das Wort der Länge 0 heißt *leeres Wort* und wird bezeichnet mit ε . Es sind

$$\Sigma^n := \{w_1 w_2 \cdots w_n \mid w_i \in \Sigma, i = 1, \dots, n\}$$

die Menge aller Wörter der Länge n und

$$\Sigma^* := \bigcup_{n \geq 0} \Sigma^n$$

die Menge aller Wörter über Σ .

Definition 6.2 Die *Verkettung* der Wörter

$$v = v_1 v_2 \cdots v_n \quad \text{und} \quad w = w_1 w_2 \cdots w_n$$

ergibt das Wort

$$vw = v_1 v_2 \cdots v_n w_1 w_2 \cdots w_n.$$

Für das leere Wort ε gilt $w\varepsilon = \varepsilon w = w$, $w \in \Sigma^*$.

Definition 6.3 Jede Teilmenge $L \subseteq \Sigma^*$ heißt *Sprache* über dem Alphabet Σ . Die *Vereinigung* bzw. die *Verkettung* zweier Sprachen L_1 und L_2 ist definiert durch

$$L_1 \cup L_2 = \{w \mid w \in L_1 \text{ oder } w \in L_2\},$$

$$L_1 L_2 = \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}.$$

Definition eines Automaten Wir geben zunächst die formale Definition eines deterministischen Automaten an.

Definition 6.4 Ein *deterministischer Automat* (kurz: *DEA*) ist ein 4-Tupel

$$\mathcal{A} = (Z, \Sigma, \delta, z_0, F),$$

hierbei ist

1. Z eine endliche Menge (*Zustandsmenge*),
2. Σ eine endliche Menge (*Eingabealphabet*),
3. $\delta : Z \times \Sigma \rightarrow Z$ eine Abbildung (*Übergangsfunktion*),
4. $z_0 \in Z$ ein Element von Z (*Anfangszustand*),
5. $F \subseteq Z$ eine endliche Menge (*Endzustandsmenge*).

Beispiel 6.1

1. Der Automat

$$\mathcal{A} = (\{z_0, z_1, z_2\}, \{0, 1\}, \delta, z_0, \{z_2\})$$

mit

δ	0	1
z_0	z_1	z_0
z_1	z_1	z_2
z_2	z_2	z_2

akzeptiert alle Wörter, die 01 als Teilwort enthalten.

2. Der Automat

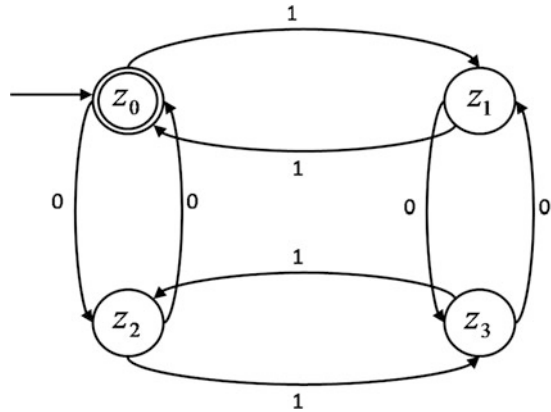
$$\mathcal{A} = (\{z_0, z_1, z_2, z_3\}, \{0, 1\}, \delta, z_0, \{z_0\})$$

mit

δ	0	1
z_0	z_2	z_1
z_1	z_3	z_0
z_2	z_0	z_3
z_3	z_1	z_2

akzeptiert alle Wörter, die eine gerade Anzahl von Nullen und Einsen enthalten. In Abb. 6.4 ist der zugehörige Automatengraph abgebildet.

Abb. 6.4 Automatengraph zur Erkennung einer geraden Anzahl von Nullen und Einsen



Bemerkung 6.1

1. Der Automat bewegt sich von seinem Anfangszustand in Abhängigkeit von der eingelesenen Eingabefolge und stoppt nach deren vollständiger Abarbeitung. Dann kann der Automat entweder in einem Endzustand $z \in F$ ankommen (Eingabefolge wird „akzeptiert“) oder in keinem Endzustand (Eingabefolge wird „nicht akzeptiert“) sein.
2. Die Menge der Endzustände F bedeutet nicht, dass der Automat seine Bewegung in diesem Zustand beendet. Sie dienen dazu Eingabefolgen, nach denen sich der Automat in einem Endzustand befindet, von Eingabefolgen zu unterscheiden, die den Automaten nicht in einen dieser Zustände überführen.
3. Die vorgestellte Automatendefinition ist deterministisch, d. h., es gibt für jeden Zustand $z \in Z$ und Buchstaben $a \in \Sigma$ genau einen Nachfolgezustand z' .
4. Häufig wird in der Modellbildung die Übergangsfunktion partiell definiert, d. h., dass bestimmte Zustandsbuchstabenpaare (z, a) keinen Nachfolgezustand zugeordnet bekommen. Das ist keine Einschränkung, da man einfach einen zusätzlichen Zustand (Fehlerzustand) einfügen kann, in den der Automat immer dann übergeht, wenn die Zustandsübergangsfunktion des ursprünglichen Automaten nicht definiert ist.
5. Ein *autonomer Automat* besitzt keine Eingangsgröße und schaltet einen Zustand von einem zum nächsten diskreten Wert, d. h., die Übergangsfunktion ist eine Abbildung der Form $\delta : Z \rightarrow Z$.
6. Die Zustandsraumdarstellung eines deterministischen Automaten für einen Eingabevektor $v \in \Sigma^*$ lautet

$$z(n+1) = \delta(z(n), v_n), \quad v_n \in \Sigma.$$

Diese Gleichung gibt an, in welchen Zustand $z(n+1)$ der Automat übergeht, wenn er sich vorher im Zustand $z(n)$ befand. Durch Anwendung dieser Gleichung für $n = 0, 1, 2, \dots$ ergibt sich die Zustandsfolge $Z_s = (z(0), z(1), z(2), \dots)$, die im vorgegebenen Anfangszustand $z(0) = z_0$ beginnt.

Die Zustandsübergänge von Automaten erfolgen unmittelbar, wobei die Schaltzeit als vernachlässigbar klein angesehen wird. Die Zustandsübergangsfunktion δ wird manchmal auch als Schaltregel bezeichnet.

Darstellungsformen von Automaten Es gibt verschiedene Möglichkeiten Automaten darzustellen:

- **Automatentabelle:** In vielen Fällen ist die Übergangsfunktion δ nicht analytisch darstellbar. Daher wird sie durch die Menge aller Paare (z, z', a) von Zuständen z zum zugehörigen Nachfolgezustand z' beim Lesen von $a \in \Sigma$ als Tabelle angegeben.
- **Automatengraph:** Automaten werden durch sogenannte Automatengraphen dargestellt. Die Knoten sind die Zustände des Automaten und die gerichteten Kanten sind die durch die Übergangsfunktion δ beschriebenen Nachbarschaftsbeziehungen der Zustände. An die Kante (z, z') schreibt man durch Kommas getrennt alle Elemente $a \in \Sigma$ mit $\delta(z, a) = z'$. Der Anfangszustand wird durch einen Pfeil und die Endzustände werden als Knoten mit doppeltem Rand gekennzeichnet.
- **Matrixdarstellung:** Autonome Automaten können auch durch Matrizen repräsentiert werden. Es seien N die Anzahl der Automatenzustände und $p(k)$ ein N -dimensionaler Vektor, wobei dessen i -te Komponente angibt, ob sich der Automat zum Zeitpunkt k im Zustand i befindet ($p_i(k) = 1$) oder nicht ($p_i(k) = 0$). Die Zustandsübergangsfunktion G wird als (N, N) -Matrix geschrieben, wobei $g_{ij} = 1$ falls $j = G(i)$ gilt, d. h., der Automat geht vom Zustand i in den Zustand j . Alle anderen Elemente sind gleich null. Das Verhalten des Automaten kann folgendermaßen dargestellt werden:

$$p(k+1) = G^T p(k), \quad p(0) = p_0.$$

Der Anfangszustand p_0 enthält bis auf die zu z_0 gehörende Stelle nur Nullen und bei z_0 eine Eins.

Sprache eines Automaten Die Sprache eines Automaten ist der grundlegendste Begriff in der Automatentheorie. Sie dient zur Beschreibung von den durch den Automaten erzeugbaren Wörtern. Für eine kompakte Definition benötigen wir die erweiterte Übergangsfunktion.

Definition 6.5 Die *erweiterte Übergangsfunktion* $\hat{\delta}$ eines DEA $\mathcal{A} = (Z, \Sigma, \delta, z_0, F)$ wird induktiv definiert durch:

1. $\hat{\delta}(z, \varepsilon) := z, \quad \forall z \in Z,$
2. $\hat{\delta}(z, xa) := \delta(\hat{\delta}(z, x), a), \quad \forall z \in Z, \quad \forall x \in \Sigma^*, \quad \forall a \in \Sigma.$

Beispiel 6.2 Wir betrachten den Automaten

$$\mathcal{A} = (\{z_0, z_1, z_2\}, \{0, 1\}, \delta, z_0, \{z_2\})$$

mit

δ	0	1
z_0	z_1	z_0
z_1	z_1	z_2
z_2	z_2	z_2

der alle Wörter $w \in \{0, 1\}^*$ akzeptiert, die 01 als Teilwort enthalten. Damit erhalten wir beispielsweise für das Wort $w = 001$ den Zustand

$$\widehat{\delta}(z, 001) = \delta(\widehat{\delta}(z, 00), 1) = \delta(\delta(\delta(z, 0), 0), 1) = \delta(\delta(z_1, 0), 1) = \delta(z_1, 1) = z_2.$$

Die wichtigste Eigenschaft eines Automaten ist seine Sprache. Darunter versteht man die Menge aller Eingangsfolgen, die der Automat erzeugen kann, während er vom Anfangszustand in einen Endzustand übergeht. Man kann mithilfe des Automaten entweder Folgen erzeugen oder überprüfen, ob eine gegebene Folge zur Sprache des Automaten gehört. Ein breites Anwendungsfeld ist die Informatik zur Überprüfung der Syntax bei der Übersetzung von Programmen. Auch in der Automatisierungstechnik ist die Sprache von großer Bedeutung. Die Aufgabe ist hier, die von einem Steuerkreis erzeugten Ereignisfolgen (Wörter der Sprache) darzustellen und zu analysieren.

Definition 6.6 Es sei $\mathcal{A} = (Z, \Sigma, \delta, z_0, F)$ ein DEA. Dann heißt

$$L(\mathcal{A}) := \{x \in \Sigma^* \mid \widehat{\delta}(z_0, x) \in F\}$$

die *Sprache* von $\mathcal{A}$. Eine Sprache heißt *regulär*, wenn sie von einem DEA akzeptiert wird, d. h., $L \subseteq \Sigma^*$ ist regulär genau dann, wenn es einen DEA $\mathcal{A}$ mit $L = L(\mathcal{A})$ gibt.

Beispiel 6.3 Der Automat

$$\mathcal{A} = (\{z_0, z_1, z_2\}, \{0, 1\}, \delta, z_0, \{z_2\})$$

mit

δ	0	1
z_0	z_1	z_0
z_1	z_2	z_0
z_2	z_2	z_2

beschreibt die Sprache

$$L(\mathcal{A}) = \{00w \mid w \in \{0, 1\}^*\},$$

die Menge aller Wörter aus dem Alphabet $\{0, 1\}$, die mit 00 beginnen.

Die reguläre Sprache hat eine große Bedeutung für die Modellierung von technischen oder naturwissenschaftlichen Prozessen.

6.1.3 Problemstellung

In diesem Abschnitt betrachten wir einige zentrale Problemstellungen zur Analyse der Eigenschaften von Automaten: Akzeptiert ein Automat ein Wort? Sind alle Zustände des Automaten erreichbar? Wann sind zwei Automaten äquivalent? Diese Eigenschaften haben große Bedeutung für die Modellierung und Analyse von technischen Problemstellungen.

Wortproblem Das zentralste Problem in der Automatentheorie ist die Frage, ob ein Wort zur Sprache des Automaten gehört:

WORTPROBLEM

Gegeben: deterministischer Automat $\mathcal{A}$, Wort $w \in \Sigma^*$.

Gesucht: Gehört das Wort zur Sprache $w \in L(\mathcal{A})$?

Das Wortproblem hat eine große Bedeutung bei der Modellierung eines Automaten. Nur diejenigen Ereignisse (Eingabewörter) dürfen akzeptiert werden, die zu der fest definierten Sprache des Automaten gehören.

Erreichbarkeitsanalyse Bei zahlreichen Anwendungen tritt die Frage auf, ob das betrachtete System vom Anfangszustand z_0 aus nach einem oder mehreren Zustandsübergängen jeden beliebigen Zustand $z \in Z$ annehmen kann.

ERREICHBARKEIT

Gegeben: deterministischer Automat $\mathcal{A}$.

Gesucht: Menge aller vom Anfangszustand erreichbaren Zustände.

Nicht erreichbare Zustände treten aus verschiedenen Gründen auf, beispielsweise bei der Definition ungewollter Zustände während der Modellbildung oder durch das Entstehen von verbotenen Zuständen in der Steuereinrichtungen. Die Verhaltensbeschreibung eines Automaten wird deutlich vereinfacht, falls man die Zustände in Abhängigkeit von ihrer gegenseitigen Erreichbarkeit klassifiziert.

Definition 6.7 Zwei Zustände z und z' heißen *stark zusammenhängend*, wenn es im Automatengraph einen Pfad vom Knoten z zum Knoten z' und einen Pfad vom Knoten z' zum Knoten z gibt. Falls alle Zustände untereinander stark zusammenhängen, so heißt der Automat *irreduzibel*.

In einem Automaten mit irreduziblen Zuständen kann jeder dieser Zustände nach einer endlichen Anzahl von Übergängen erneut angenommen werden. Ein Automat heißt *lebendig*, falls er eine beliebig lange Zustandsfolge erzeugen kann. Ein *Livelock* bezeichnet einen lebendigen Automaten, der sich in einer Zustandsmenge befindet, von der er nicht in einen Endzustand kommt.

Äquivalenz und Minimierung Zwei weitere bedeutende Aufgaben bei der Analyse von Automaten sind die folgenden beiden Problemstellungen:

ÄQUIVALENZ

Gegeben: zwei deterministische Automaten $\mathcal{A}_1$ und $\mathcal{A}_2$.

Gesucht: Gilt $L(\mathcal{A}_1) = L(\mathcal{A}_2)$?

MINIMALAUTOMAT

Gegeben: deterministischer Automat $\mathcal{A}$.

Gesucht: deterministischer Automat $\mathcal{A}_M$ mit einer minimalen Anzahl von Zuständen und $L(\mathcal{A}) = L(\mathcal{A}_M)$.

Das Ziel bei der Modellierung einer gegebenen Problemstellung durch einen Automaten ist es, einen Automaten mit der kleinsten Anzahl von Zuständen zu bestimmen. In vielen Fällen wird dies jedoch nicht erreicht. Durch eine anschließende Minimierung des Automaten kann ein optimaler Automat bestimmt werden. Damit lässt sich das zu modellierende Problem wesentlich kompakter beschreiben und effizienter berechnen.

6.1.4 Grundlegende Lösungsprinzipien

Wir zeigen nun die grundlegenden algorithmischen Verfahren zur Lösung der obigen Problemstellungen.

Wortproblem Das Wortproblem zur Entscheidung, ob ein gegebenes Wort w in der Sprache $L(\mathcal{A})$ des Automaten $\mathcal{A} = (Z, \Sigma, \delta, z_0, F)$ liegt, kann sehr schnell und einfach gelöst werden:

1. Definition des Zustands $z = z_0$;
2. Wiederholung, so lange bis das Ende des Wortes erreicht ist:
 - (a) Zugriff auf den i -ten Buchstaben w_i des Wortes w ;
 - (b) Bestimmung des Nachfolgezustands z durch Anwendung der Automatentabelle δ für den Zustand z und den Buchstaben w_i ;
3. falls $z \in F$, wird das Wort w akzeptiert, andernfalls nicht.

Erreichbarkeitsanalyse Für die Analyse der Erreichbarkeit von Automatenzuständen verwenden wir die Methoden der Graphentheorie. Die Erreichbarkeit kann man anhand der Adjazenzmatrix A des Automatengraphen prüfen. Der Zustand i ist vom Zustand j genau dann erreichbar, wenn es eine Zahl k gibt, sodass das (i, j) -te Element von A^k verschieden von null ist, wobei die Zahl k die Länge eines Pfades von i nach j angibt. Es sei N die Anzahl der Zustände des Automaten. Alle von null verschiedenen Elemente in

der zu z_0 gehörenden Spalte der Matrix

$$A = \sum_{k=0}^{N-1} A^k$$

geben an, dass der entsprechende Zustand erreichbar ist.

Alternativ besteht auch die Möglichkeit, den Dijkstra-Algorithmus zur Bestimmung der kürzesten Wege im zugehörigen Automatengraphen anzuwenden. Diejenigen Entfernungen, die unendlich sind, sind dann im Automaten nicht erreichbar.

Äquivalenz und Minimierung Zur Lösung des Minimierungsproblems benötigen wir zunächst ein paar grundlegende Definitionen.

Definition 6.8 Es seien M eine Menge und $\sim$ eine binäre Relation auf M . Die binäre Relation heißt *Äquivalenzrelation* auf M , falls für alle $x, y \in M$ gilt:

1. Reflexivität: $x \sim x$;
2. Symmetrie: $x \sim y \Rightarrow y \sim x$;
3. Transitivität: $x \sim y \wedge y \sim z \rightarrow x \sim z$.

Für $x \in M$ ist die *Äquivalenzklasse* von x in M :

$$[x] = \{y \in M \mid x \sim y\}.$$

Die Menge aller Äquivalenzklassen wird mit $M/\sim$ bezeichnet, und deren Mächtigkeit heißt *Index* von $\sim$.

Beispiel 6.4 Für alle $a, b \in \mathbb{Z}$ und $n \in \mathbb{N}$ ist die folgende Modulooperation eine Äquivalenzrelation:

$$a \equiv b \pmod{n} \Rightarrow \exists k \in \mathbb{Z} : nk = a - b.$$

Dann gilt:

$$\begin{aligned} [0] &:= \{0, n, 2n, \dots\}, \\ [1] &:= \{1, n+1, 2n+1, \dots\}, \\ [2] &:= \{2, n+2, 2n+2, \dots\}. \end{aligned}$$

Definition 6.9 Es sei $\mathcal{A} = (Z, \Sigma, \delta, z_0, F)$ ein DEA. Für zwei Zustände z, z' sei die folgende Äquivalenzrelation definiert:

$$z \sim_A z' \Leftrightarrow \forall w \in \Sigma^* : \widehat{\delta}(z, w) \in F \Leftrightarrow \widehat{\delta}(z', w) \in F.$$

Diese Äquivalenzrelation bedeutet, dass zwei Zustände z und z' äquivalent sind, wenn bei Eingabe eines beliebigen Wortes $w \in \Sigma^*$ der Automat von beiden dieser Zustände in einem Endzustand landet.

Satz 6.1 Eine Sprache L ist genau dann regulär, wenn die Anzahl der erzeugten Äquivalenzklassen endlich ist.

Beispiel 6.5

1. Die Sprache

$$L = \{x \in \{0, 1\}^* \mid x \text{ endet mit } 00\}$$

ist regulär, da sie nur die drei folgenden Äquivalenzklassen besitzt:

$$[\varepsilon] = \{x \mid x \text{ endet nicht mit } 0\},$$

$$[0] = \{x \mid x \text{ endet mit } 0, \text{ aber nicht mit } 00\},$$

$$[00] = \{x \mid x \text{ endet mit } 00\}.$$

2. Die Sprache

$$L = \{a^n b^n \mid n \geq 1\}$$

ist nicht regulär, da sie unendlich viele Äquivalenzklassen $[a^i b]$ besitzt. Denn für $i \neq j$ sind die Wörter $a^i b$ und $a^j b$ nicht äquivalent, da mit $z = b^{i-1}$ das Wort $a^i b z = a^i b^i \in L$ in der Sprache und $a^j b z = a^j b^i \notin L$ nicht in der Sprache liegen.

Mithilfe der obigen Äquivalenzrelation für zwei Zustandspaare definieren wir den sogenannten *Quotientenautomat*, der nur noch die Zustände des ursprünglichen Automaten $\mathcal{A}$ besitzt, die nicht äquivalent sind. Der Quotientenautomat $\mathcal{A}_Q$ besitzt unter allen zu $\mathcal{A}$ äquivalenten Automaten die minimale Anzahl von Zuständen und akzeptiert die identische Sprache $L(\mathcal{A}) = L(\mathcal{A}_Q)$.

Mit Hilfe dieser Äquivalenzrelation kann mit dem nachfolgenden Table-Filling-Algorithmus ein gegebener deterministischer Automat minimiert werden. Die Idee ist die folgende: Falls zwei Zustände zu einer Äquivalenzklasse gehören, so können sie vereinigt werden. Dabei brauchen wir nur alle Wörter zu betrachten mit einer Länge kleiner gleich der Anzahl der Zustände des Automaten.

Prinzip des Table-Filling-Algorithmus

1. Aufstellung einer Tabelle mit allen Zustandspaaren $\{z, z'\}$
2. Markierung aller Paare $\{z, z'\}$ mit $z \in F$ und $z' \notin F$
3. Für jedes unmarkierte Paar $\{z, z'\}$ und jedes $a \in \Sigma$ wird getestet, ob das folgende Paar markiert ist:

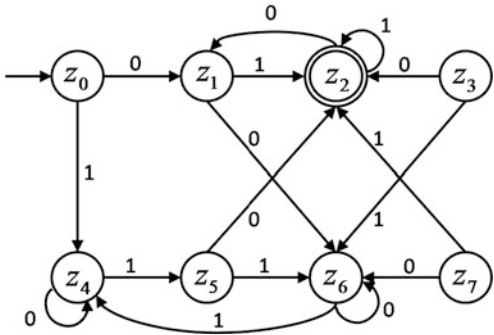
$$\{\delta(z, a), \delta(z', a)\}.$$

Falls ja, wird auch das $\{z, z'\}$ markiert. Dieser Schritt wird so lange wiederholt, bis keine markierten Zustände mehr dazukommen

4. Vereinigung aller unmarkierten Paare zu einem Zustand

Wenn ein Zustandspaar durch den Table-Filling-Algorithmus nicht markiert wird, so sind diese Zustände äquivalent.

Abb. 6.5 Automat zur Minimierung mit dem Table-Filling-Algorithmus



Beispiel 6.6 Gegeben ist der folgende DEA in Abb. 6.5. Mit dem Table-Filling-Algorithmus erhalten wir nun die folgenden Tabelleneinträge.

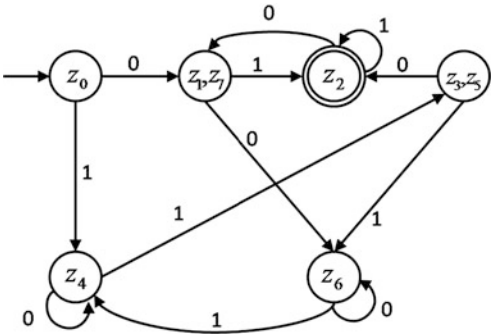
- **1. Iteration:** Wir markieren alle Paare $\{z, z'\}$ mit $z \in F$ und $z' \notin F$ mit dem Wert 1:

	z_0	z_1	z_2	z_3	z_4	z_5	z_6	z_7
z_0	0	0	1	0	0	0	0	0
z_1	0	0	1	0	0	0	0	0
z_2	1	1	0	1	1	1	1	1
z_3	0	0	1	0	0	0	0	0
z_4	0	0	1	0	0	0	0	0
z_5	0	0	1	0	0	0	0	0
z_6	0	0	1	0	0	0	0	0
z_7	0	0	1	0	0	0	0	0

- **2. Iteration:** Für jedes unmarkierte Paar $\{z, z'\}$ in Form des Wertes 0 in der Tabelle und jedes $a \in \Sigma$ testen wir, ob das Paar $\{\delta(z, a), \delta(z', a)\}$ in Form der Nachfolgezustände von z und z' bei Eingabe von a markiert ist. Falls ja, markieren wir auch den Eintrag $\{z, z'\}$ mit dem Wert 1:

	z_0	z_1	z_2	z_3	z_4	z_5	z_6	z_7
z_0	0	1	1	1	0	1	0	1
z_1	1	0	1	1	1	1	1	0
z_2	1	1	0	1	1	1	1	1
z_3	1	1	1	0	1	0	1	1
z_4	0	1	1	1	0	1	0	1
z_5	1	1	1	0	1	0	1	1
z_6	0	1	1	1	1	1	0	1
z_7	1	0	1	1	1	1	1	0

Abb. 6.6 Minimalautomat nach dem Table-Filling-Algorithmus



- **3. Iteration:** Für jedes unmarkierte Paar $\{z, z'\}$ und jedes $a \in \Sigma$ testen wir, ob das Paar $\{\delta(z, a), \delta(z', a)\}$ markiert ist. Falls ja, markieren wir auch das Element $\{z, z'\}$. Bei nochmaliger Wiederholung dieses Schrittes kommen keine markierten Zustände mehr hinzu:

	z_0	z_1	z_2	z_3	z_4	z_5	z_6	z_7
z_0	0	1	1	1	1	1	1	1
z_1	1	0	1	1	1	1	1	0
z_2	1	1	0	1	1	1	1	1
z_3	1	1	1	0	1	0	1	1
z_4	1	1	1	1	0	1	1	1
z_5	1	1	1	0	1	0	1	1
z_6	1	1	1	1	1	1	0	1
z_7	1	0	1	1	1	1	1	0

In der Automatentabelle befinden sich in der 2. und 8. bzw. 4. und 6. Zeile eine Null. Damit sind die Zustände 1 und 7 sowie 3 und 5 äquivalent, d. h., $z_1 \sim z_7$ und $z_3 \sim z_5$. Der minimierte Automat ist in Abb. 6.6 dargestellt.

6.1.5 Algorithmus und Implementierung

Wir formulieren nun die obigen Algorithmen als Pseudocode und geben eine Implementierung in Java an. Dazu definieren wir zunächst eine Klasse `Automat` mit den Instanzvariablen `delta` für die Übergangsfunktion, `z0` für den Anfangszustand und `F` für die Menge aller Endzustände. Weiterhin enthält die Klasse neben einem Konstruktor noch einige `get`-Methoden:

```
public class Automat
{
    private int delta[][];
    private int z0;
    private int F[];
```

```

public Automat(int delta[][], int z0, int F[])
{
    this.delta = delta;
    this.z0    = z0;
    this.F     = F;
}

public int getAnzahlZustaende()
{
    return delta.length;
}
public int getAnzahlAlphabet()
{
    return delta[0].length;
}
public int[][] getUebergangsfunktion()
{
    return delta;
}
public int getAnfangszustand()
{
    return z0;
}
public int[] getEndzustaende()
{
    return F;
}
}

```

In diese Klasse werden wir die nachfolgenden Methoden für das Wortproblem und den Table-Filling-Algorithmus implementieren.

Wortproblem Der Pseudocode des Algorithmus für das Wortproblem folgt unmittelbar durch Anwendung der Übergangsfunktion auf das Eingabewort.

Algorithmus 13 WORTPROBLEM

Input: Automat $\mathcal{A} = (Z, \Sigma, \delta, z_0, F)$, Wort $w \in \Sigma^*$

Output: 1, falls $w \in L(\mathcal{A})$; andernfalls 0

Komplexität: $O(n)$

```

1:  $z = z_0$ 
2: for  $i = 1$  to  $n$  do
3:    $z = \delta(z, w_i)$ 
4: if  $z \in F$  then
5:   return 1
6: else
7:   return 0

```

Allgemeine Erklärung

In Zeile 2–3 wird iterativ der neue Zustand aus dem neuen Buchstaben des Eingabewortes bestimmt. In Zeile 4–7 wird überprüft, ob der resultierende Zustand ein Endzustand ist und damit das Eingabewort akzeptiert.

Aufwandsabschätzung

Der Rechenaufwand des Algorithmus zur Lösung des Wortproblems ist linear in der Anzahl der Buchstaben n des Wortes w , also $O(n)$.

Die Implementierung eines Automaten und die Lösung des zugehörigen Wortproblems stellen wir an dem vorgestellten Beispiel der Zahlendarstellung vor:

1. Definition der Parameter des Automaten:

```
public class Zahlendarstellung
{
    public static void main(String[] args)
    {
        // ----- Eingabe -----
        // --- 1. Anfangszustand
        int z0 = 0;

        // --- 2. Übergangsfunktion: {0..9, +-, ., E}, -1 ... kein Übergang
        int delta[][] = {
            { 2, 1, -1, -1},    //Zustand 0
            { 2, -1, -1, -1},   //Zustand 1
            { 2, -1, 3, 5},     //Zustand 2
            { 4, -1, -1, -1},   //Zustand 3
            { 4, -1, -1, 5},    //Zustand 4
            {-1, 6, -1, -1},    //Zustand 5
            { 7, -1, -1, -1},   //Zustand 6
            { 7, -1, -1, -1},   //Zustand 7
        };

        // --- 3. Endzustand
        int F[] = {2, 4, 7};

        // --- 4. Wort
        String w = "234.98";

        // -----
        // --- 1. Definition eines Automaten
        Automat a = new Automat(delta, z0, F);

        // --- 2. Bearbeitung des Wortproblems
        if (a.wortproblem(string2Int(w)))
            System.out.printf("Automat akzeptiert das Wort.");
        else
            System.out.printf("Automat akzeptiert das Wort nicht.");
    }

    // Gegeben: Codierung der Übergangsfunktion
    public static int[] string2Int(String w)
}
```

Da es für den Automaten irrelevant ist, welche Zahl eingegeben wurde, können wir die Übergangsfunktion deutlich kompakter schreiben. Die Umwandlung eines Eingabewortes w mit den vier verschiedenen Symbolen 0..9, +, . und E in die Codierung von 0, 1, 2, 3 erfolgt mit der Methode `string2Int`:

```
public static int[] string2Int(String w)
{
    int w_idx[] = new int[w.length()];
    for(int i=0; i<w.length(); i++)
    {
        switch(w.charAt(i))
        {
            case '0': case '1': case '2': case '3': case '4': case '5':
            case '6': case '7': case '8': case '9': w_idx[i] = 0; break;

            case '+': case '-': w_idx[i] = 1; break;
            case '.':           w_idx[i] = 2; break;
            case 'E':           w_idx[i] = 3; break;
            default:            w_idx[i] = -1;
        }
    }
    return w_idx;
}
```

2. Implementierung der Funktion `wortproblem` in die Klasse `Automat`:

```
public boolean wortproblem(int w[])
{
    int z = z0;
    for(int i=0; i<w.length; i++)
    {
        z = delta[z][w[i]];
        if (z==-1)
            return false;
    }
    return this.istEndzustand(z);
}

// Überprüfung ob Zustand ein Endzustand ist
public boolean istEndzustand(int z)
{
    for(int i=0; i<F.length; i++)
        if(z == F[i])
            return true;
    return false;
}
```

Minimierung Wir stellen nun zunächst den Pseudocode des Table-Filling-Algorithmus für das Minimierungsproblem vor. Die Methode $\text{ISTMARKIERT}(i, j)$ testet hierbei, ob ein Zustandspaar (i, j) markiert wird.

Algorithmus 14 TABLE-FILLING-ALGORITHMUS

Input: Automaten $\mathcal{A} = (Z, \Sigma, \delta, z_0, F)$
Output: Menge der Zustände Z' des Minimalautomaten
Komplexität: $O(n)$

```

1:  $Z' = Z$ 
2:  $n = |Z|$ 
3: for  $i = 1$  to  $n$  do
4:   for  $j = 1$  to  $n$  do
5:     if  $\text{ISTMARKIERT}(i, j)$  then
6:        $\text{Tabelle}(i, j) = 1$ 
7:     else
8:        $\text{Tabelle}(i, j) = 0$ 
9:   repeat
10:     $b = \text{false}$ 
11:    for  $i = 1$  to  $n$  do
12:      for  $j = 1$  to  $n$  do
13:        if  $\text{Tabelle}(i, j) = 0$  then
14:          for  $a = 1$  to  $|\Sigma|$  do
15:            if  $\text{Tabelle}(\delta(i, a), \delta(j, a)) = 1$  then
16:               $\text{Tabelle}(i, j) = 1$ 
17:               $b = \text{true}$ 
18:    until
19:    for  $i = 2$  to  $n$  do
20:      for  $j = 1$  to  $i - 1$  do
21:        if  $\text{Tabelle}(i, j) = 0$  then
22:           $Z' = Z \setminus \{i, j\} \cup (i, j)$ 

```

Allgemeine Erklärung

In den Zeilen 1–8 wird eine Tabelle mit allen Zustandspaaren $\{z, z'\}$ aufgestellt, bei denen alle Paare $\{z, z'\}$ mit $z \in F$ und $z' \notin F$ durch den Wert 1 markiert sind. In den Zeilen 9–18 wird für jedes unmarkierte Paar $\{z, z'\}$ und jedes $a \in \Sigma$ getestet, ob das folgende Paar $\{\delta(z, a), \delta(z', a)\}$ markiert ist. Wenn dies der Fall ist, wird auch das Paar $\{z, z'\}$ markiert. Dieser Schritt wird so lange wiederholt, bis keine neuen markierten Zustände mehr dazukommen. In den Zeilen 19–22 werden alle unmarkierten Paare zu einem Zustand vereinigt.

Aufwandsabschätzung

Der Rechenaufwand des Algorithmus zur Minimierung eines Automaten besitzt die Rechenzeit von $O(n^2)$, da jedes Zustandspaar maximal einmal markiert wird.

Die Implementierung des Table-Filling-Algorithmus erfolgt ebenfalls wieder in die Klasse `Automat`:

```

public void minimiereAutomaten()
{
    int n = this.getAnzahlZustaende();

    // --- 1. Aufstellen einer Tabelle mit allen Zustandspaaren {z,z'}
    int Tabelle[][] = new int[n][n];

    // --- 2. Markierung aller Paare {z,z'}, wobei z \in F und z' \notin F
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
            if (istMarkiert(i, j))
                Tabelle[i][j] = 1;
    LinAlgebra.ausgabe(Tabelle);
    System.out.println();

    //--- 3. Für jedes unmarkiertes Paar {z,z'} wird getestet, ob
    //         {delta(z, a), delta(z', a)} bereits markiert ist.
    //         Falls ja, wird auch {z,z'} markiert

    boolean adg;
    do
    {
        adg = false;
        for (int i=0; i<n; i++) // fuer alle Tabelleneintraege
        {
            for (int j=0; j<n; j++) // fuer alle Tabelleneintraege
            {
                if (Tabelle[i][j] == 0) // falls unmarkiertes Paar
                {
                    for(int a=0; a<this.getAnzahlAlphabet(); a++) // fuer jeden Buchstaben
                    {
                        if (Tabelle[delta[i][a]] [delta[j][a]] == 1) // falls Paar markiert
                        {
                            Tabelle[i][j] = 1;
                            adg = true;
                        }
                    }
                }
            }
        }
        LinAlgebra.ausgabe(Tabelle);
        System.out.println();
    }
    while(adg);

    // --- 4. Vereinigung aller unmarkierten Paare zu einem Zustand.
    for (int i=1; i<n; i++)
        for (int j=0; j<i-1; j++)
            if (Tabelle[i][j] == 0) // unmarkiertes Paar
                System.out.printf(" Zustand %d und Zustand %d werden verschmolzen.\n", j, i);
}

```

6.1.6 Anwendungen

Die Automaten sind durch ihre Einfachheit für die Modellierung von ereignisdiskreten Systemen von großer Bedeutung. Bei der Verwendung von Automaten für formale Sprachen verwenden wir oftmals die zugehörigen systemtheoretischen Begriffe:

Formale Sprache	Systemtheorie
Buchstabe	Ereignis
Wort	Ereignisfolge
Alphabet	Ereignismenge
Sprache	Verhalten

Bei der Modellierung eines technischen Systems oder eines Prozesses unterscheidet man die folgenden zwei Arten:

- **Zustandsorientierte Modellierung:** Die Zustände des Automaten stellen die Betriebszustände einer Maschine oder eines Prozesses dar und die Ereignisse die zugehörigen Veränderungen, die hierzu ablaufen.
- **Sprachorientierte Modellierung:** Die Zustände haben dabei keine Interpretation und dienen nur als Elemente zur Beschreibung der Übergänge in Form von Ereignissen.

Bei der Modellierung eines Systems durch einen Automaten ist es wichtig, auf die saubere und exakte Definition der Zustände und der jeweiligen Übergänge zu achten.

Prinzip der Modellierung eines Systems durch einen Automaten

Gegeben ist ein technisches System oder ein Prozess und gesucht ist ein deterministischer Automat $\mathcal{A} = (Z, \Sigma, \delta, z_0, F)$.

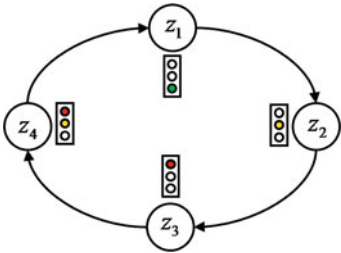
1. Definition der Zustandsmenge Z .
2. Definition der Ereignismenge Σ .
3. Bestimmung der Zustandsübergangsfunktion δ .
4. Festlegung des Anfangszustands z_0
5. Bestimmung der Menge von Endzuständen F .

Die Automaten können je nach Anwendungsfall für die folgenden zwei Interpretationen eingesetzt werden:

- **Generator:** Automat, der von einem Anfangszustand aus durch spontane Zustandsübergänge eine Ereignisfolge erzeugt;
- **Akzeptor:** Automat, der entscheidet, ob eine gegebene Ereignisfolge zur Sprache gehört oder nicht.

Ein deterministischer Automat $\mathcal{A}$ heißt *vollständig steuerbar*, wenn von jedem beliebigen Anfangszustand z_0 in endlicher Zeit durch eine bestimmte Eingangsfolge ein beliebiger Endzustand erreicht werden kann. Die Steuerbarkeit eines Automaten ist gegeben, wenn im zugehörigen Automatengraphen ein Pfad vom Anfangszustand z_0 zu einem Endzustand existiert. Falls im Automatengraphen von einem Knoten mehrere Kanten ausgehen,

Abb. 6.7 Verkehrsampel als deterministischer Automat



kann der Automat in dem jeweiligen Zustand mehrere mögliche Ereignisse erzeugen. Die zugehörige Eingabe hat damit Einfluss darauf, welches Ereignis als nächstes erzeugt wird.

Verkehrsampel Eine Verkehrsampel ist ein sehr einfacher autonomer Automat, der für die Steuerung der Verkehrsströme von großer Bedeutung ist. Die Verkehrsampel schaltet in einem Zyklus zwischen vier Zuständen der Menge $Z = \{z_1, z_2, z_3, z_4\}$ um. Die Interpretation der Zustände sieht wie folgt aus:

Zustand	Interpretation
z_1	Grün
z_2	Gelb
z_3	Rot
z_4	Rot, Gelb

Die Übergangsfunktion δ des autonomen deterministischen Automaten beschreibt das zugehörige Ereignis und sieht wie folgt aus:

z	z'	Ereignis
z_1	z_2	Grün ausschalten, Gelb einschalten
z_2	z_3	Gelb ausschalten, Rot einschalten
z_3	z_4	Gelb einschalten
z_4	z_1	Rot und Gelb ausschalten, Grün einschalten

Der zugehörige Automat der Verkehrsampel ist in Abb. 6.7 dargestellt.

Dialoggestaltung Automaten eignen sich sehr gut zur Dialogverarbeitung in der Mensch-Maschine-Kommunikation. Einfache Dialogsysteme in Form von Automaten wie ELIZA wurden bereits in den 1960er-Jahren geschaffen. Das Ziel besteht darin, eine Art der Kommunikation zwischen Mensch und Maschine herzustellen, um so eine Aufgabe kooperativ zu lösen. Anwendungen findet die Dialoggestaltung beispielsweise bei der Bedienung von Bank- und Fahrkartenautomaten oder bei Fahrerinformationssystemen im Auto. Für

die Dialoggestaltung gibt es eine ganze Reihe von Dialogformen, die gewisse Vor- und Nachteile besitzen:

- Frage-Antwort: System stellt Fragen, auf die der Mensch antwortet;
- Menüauswahl: Auswahl an Möglichkeiten, von denen der Nutzer sich für eine entscheidet;
- Formular: standardisiertes Mittel zur Erfassung, Ansicht und Aufbereitung von Daten;
- Kommandosprache: Eingabe von Befehlen aus einem beschränkten Vokabular;
- natürlich sprachlicher Dialog: Kommunikation in Form der natürlichen Sprache;
- direkte Manipulation: Kommunikation in Form der visuellen Darstellung;
- Multimediadialog: Nutzung verschiedenster Dialogformen.

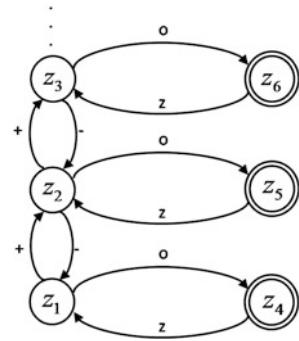
Mithilfe von Zustandsautomaten ist es sehr leicht möglich Dialoge zu entwerfen, darzustellen und als Steuereinrichtung für die Mensch-Maschine-Kommunikation zu verwenden. Die Automaten sind jedoch noch keine Form eines intelligenten Dialogsystems. Diese interaktiven Systeme zeichnen sich im Wesentlichen durch intuitive Ein- und Ausgabetechniken, eine interaktive und benutzerfreundliche Mensch-Maschine-Schnittstelle und eine Form von intelligentem Verhalten aus. Das intelligente Verhalten wird vor allem durch Methoden des maschinellen Lernens und der künstlichen Intelligenz erreicht, sodass das System in der Lage ist, logische Schlüsse aus den Eingaben des Bedieners zu ziehen. Die Mensch-Maschine-Kommunikation muss hierzu eine Form der Wissensverarbeitung besitzen, um aus den vorhandenen Fakten das Systemverhalten geeignet zu steuern.

Interaktive Systeme, die über eine gewisse maschinelle Intelligenz verfügen, heißen Expertensysteme. Ein Expertensystem ist ein komplexes wissensbasiertes Softwarepaket, das Expertenwissen aus einem speziellen Gebiet speichern, verwalten und auswerten kann. Ein Expertensystem enthält dabei die folgenden Komponenten:

- Wissensbasis: Wissensbasis enthält alle Fakten und Regeln aus dem Anwendungsbe-
reich;
- Inferenzkomponente: korrekte Verarbeitung der in der Wissensbasis festgelegten Re-
geln;
- Erklärungskomponente: Komponente zur Nachvollziehbarkeit der Entscheidungen des
Systems;
- Dialogkomponente: Benutzeroberfläche zur Kommunikation zwischen Mensch und
Maschine;
- Wissenserwerbskomponente: Aufbereitung des Wissens in einer Art strukturierter
Form.

Typische Anwendungen von Expertensystemen sind Diagnose-, Vorhersage-, Planungs- oder Steuerungssysteme. Eine Vielzahl von Methoden und Techniken zur Entwicklung von Expertensystemen werden im Band *Intelligente Algorithmen* dieser Buchreihe vorgestellt.

Abb. 6.8 Fahrstuhl als deterministischer Automat



Technisches Steuerungssystem Wir beschreiben ein technisches Steuerungssystem am Beispiel der Bewegung eines Aufzuges in einem Hochhaus mit n Stockwerken. Die Modellierung erfolgt nach der folgenden Idee: Wir beschreiben die Bewegung des Aufzuges durch eine Ereignisfolge. Für jede Etage i des Hochhauses definieren wir zwei Zustände z_i und z_{i+n} . Der erste Zustand z_i beschreibt den Aufzug mit geschlossener Tür und der zweite Zustand z_{i+n} den Aufzug mit offener Tür in der i -ten Etage. Die Menge der Zustände eines Aufzugsautomaten in einem Hochhaus mit n Stockwerken besteht dann aus genau $2n$ Zuständen:

$$Z = \{z_1, z_2, \dots, z_{2n-1}, z_{2n}\}.$$

Eine Ereignisfolge w ist eine Zeichenkette über ein Alphabet $\Sigma = \{+, -, o, z\}$, wobei $+$ für die Bewegung um ein Stockwerk nach oben, $-$ für die Bewegung um ein Stockwerk nach unten, o für Öffnen und z für Schließen der Fahrstuhlür stehen. Der Anfangszustand z_0 kann ein beliebiger Zustand des Automaten sein, beispielsweise der Zustand z_1 , also der Aufzug im Erdgeschoss mit geschlossener Tür. Die Übergangsfunktion $\delta : Z \times \Sigma \rightarrow Z$ des Automaten $\mathcal{A} = (Z, \Sigma, \delta, F, z_0)$ lässt sich wie folgt angeben:

$$\begin{aligned} z_{i+1} &= \delta(z_i, +), \\ z_{i-1} &= \delta(z_i, -), \\ z_{i+n} &= \delta(z_i, o), \\ z_i &= \delta(z_{i-n}, z). \end{aligned}$$

Der Automatengraph des Automaten ist in Abb. 6.8 dargestellt. Die Sprache $L(\mathcal{A})$ beschreibt dann das Verhalten des Automaten. Wenn man beispielsweise annimmt, dass der Aufzug mit geschlossener Tür in irgendeiner Etage steht, dann beginnen alle Wörter mit den Buchstaben $+$, $-$, o . Anschließend können die vier Buchstaben in beliebiger Reihenfolge auftreten, wobei sich jedoch o und z abwechseln müssen.

Autonomer Roboter Wir definieren einen deterministischen Automaten $\mathcal{A} = (Z, \Sigma, \delta, F, z_0)$, der ein autonomes Versorgungsfahrzeug durch eine Fabrikhalle steuert. Die Wege durch die Halle sind dabei in einem Rechteckmuster gegeben, d.h., die Kurven haben immer einen Winkel von 90° . Die Menge der Zustände $Z = \{z_0, z_1, \dots, z_4\}$ wird wie folgt beschrieben:

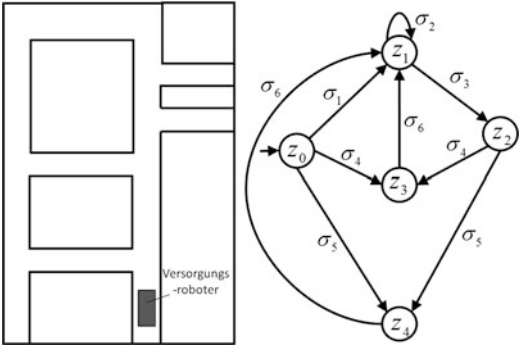
Zustand	Beschreibung
z_0	Start der Fahrt
z_1	Geradeaus fahren
z_2	Abbiegung voraus
z_3	Rechtskurve
z_4	Linkskurve

Das Eingabealphabet $\Sigma = \{\sigma_1, \dots, \sigma_6\}$ besteht aus den folgenden definierten Ereignissen:

Ereignis	Beschreibung
σ_1	Beschleunigung auf Geschwindigkeit v_1
σ_2	Konstante Geschwindigkeit v_1
σ_3	Reduzieren der Geschwindigkeit auf v_2
σ_4	Lenken um Rechtskurve
σ_5	Lenken um Linkskurve
σ_6	Geradeaus lenken, beschleunigen auf v_1

Der Versorgungsroboter startet im Zustand z_0 . Auf einer Geraden bewegt sich der Roboter mit konstanter Geschwindigkeit v_1 . Wenn sich der Versorgungsroboter auf eine Kurve zubewegt, reduziert er die Geschwindigkeit auf v_2 . Am Beginn der Kurve lenkt er entweder nach rechts oder nach links. Nach dem Durchfahren der Kurve lenkt der Roboter geradeaus und beschleunigt wieder auf die Geschwindigkeit v_1 . Der zugehörige Automatengraph in Abb. 6.9 beschreibt die Steuerung des autonomen Roboters durch die Fabrikhalle.

Abb. 6.9 Steuerung eines autonomen Versorgungsroboters in Form eines deterministischen Automaten



6.2 Deterministische E/A-Automaten

In der Automatisierungstechnik spielen technische Systeme eine große Rolle, deren Verhalten durch einen Eingang beeinflusst und durch Messung eines Ausgangs überwacht werden kann. Für derartige Systeme wird in diesem Abschnitt der deterministische Automat mit einem Eingang und Ausgang erweitert.

6.2.1 Einführendes Beispiel

Die Aufgabe ist es, einen Getränkeautomaten durch einen Eingabe/Ausgabe-Automaten zu modellieren. Ein Getränk (G) soll dabei EUR 1,00 kosten und durch Einwerfen von 50-Cent-, 1-Euro-, 2-Euro-Stücken und anschließendes Drücken des Ausgabeknopfes (A) ausgegeben werden. Wir verwenden eine vierelementige Zustandsmenge $Z = \{z_0, z_1, z_2, z_3\}$ mit dem folgenden Eingabealphabet $V = \{50, 1, 2, A\}$ und dem Ausgabealphabet $W = \{0, G, 1G, 50, 1\}$. Für die beschriebene Aufgabenstellung entwerfen wir einen Eingabe/Ausgabe-Automaten, der in Abb. 6.10 dargestellt ist. Hierbei wird die Eingabe und Ausgabe an die betreffende Kante im Automatengraphen als Paar v/w geschrieben.

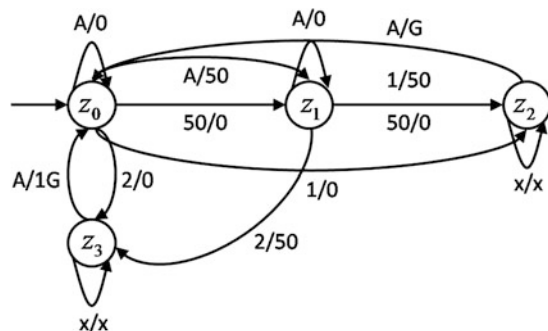
6.2.2 Grundlegende Begriffe

Der E/A-Automat ist ein dynamisches System, das die Eingabefolge $v = (v_1, v_2, \dots, v_m)$ erhält, vom Anfangszustand z_0 aus eine Zustandsfolge durchläuft und dabei eine Ausgabefolge $w = (w_1, w_2, \dots, w_m)$ erzeugt. Der Anwendungsbereich von E/A-Automaten ist wesentlich größer als bei herkömmlichen deterministischen Automaten.

Definition 6.10 Ein *deterministischer E/A-Automat* ist ein 6-Tupel

$$\mathcal{A} = (Z, V, W, G, H, z_0),$$

Abb. 6.10 Modell eines Getränkeautomaten



hierbei sind

1. Z eine endliche Menge (*Zustandsmenge*),
2. V eine endliche Menge (*Eingabealphabet*),
3. W eine endliche Menge (*Ausgabealphabet*),
4. $G : Z \times V \rightarrow Z$ eine Abbildung (*Übergangsfunktion*),
5. $H : Z \times V \rightarrow W$ eine Abbildung (*Ausgabefunktion*),
6. $z_0 \in Z$ ein Element von Z (*Anfangszustand*).

Bemerkung 6.2

1. Die Zustandsraumdarstellung des E/A-Automaten lautet dann:

$$z(n+1) = G(z(n), v_n), \quad z(0) = z_0, \\ w_n = H(z(n), v_n).$$

Die Übergangsfunktion G bestimmt den neuen Zustand und die Ausgabefunktion das neue Ausgabesymbol.

2. Im Unterschied zu normalen Automaten werden bei den E/A-Automaten keine Endzustände definiert. Der Grund dafür ist, dass es bei diesen speziellen Automaten auf die Umwandlung einer Eingabefolge in eine Ausgabefolge ankommt. Bei Verwendung der speziellen Menge $W = \{\text{akzeptiert, nicht akzeptiert}\}$ kann man einen E/A-Automaten auch zur Analyse der Sprache verwenden.
3. Im Automatengraph werden die Eingabe und Ausgabe an die betreffende Kante als Paar v/w geschrieben. Häufig werden die Funktionen G und H partiell definiert, so dass die Automatentabellen nicht alle möglichen Paare enthalten.

Wir betrachten nun zwei grundlegende E/A-Automaten.

Mealy-Automat Automat, bei dem die Ausgabefunktion H von der Eingabe v abhängt:

$$w_n = H(z(n), v_n).$$

Bei diesen Automaten wird die Ausgabe w_n direkt von der zur gleichen Zeit vorkommenden Eingabe v_n beeinflusst und die Ausgabe ist dem Zustandsübergang $z(n) \rightarrow z(n+1)$ zugeordnet.

Moore-Automat Automat, bei dem die Ausgabefunktion H nicht von der Eingangsgröße v abhängt:

$$w_n = H(z(n)).$$

Bei diesen Automaten wird die Ausgabe $w(n)$ dem Zustand $z(n)$ zugeordnet und hängt über den Zustand nur von den vorherigen Eingaben ab. Bei einem Moore-Automaten tritt

in der Tabelle für die Ausgabefunktion H in allen Zeilen, die zu einem bestimmten Zustand z gehören, stets derselbe Wert w für die Ausgabe auf. Im Automatengraphen erkennt man einen Moore-Automaten daran, dass bei allen von einem Zustand ausgehenden Kanten dieselbe Ausgabe steht. Derartige Automaten repräsentieren beispielsweise Systeme, deren Ausgabe den Sensorwerten im aktuellen Zustand entspricht.

6.2.3 Problemstellung

Die zentrale Aufgabe für E/A-Automaten ist es, zu einem gegebenen Automaten und einem Eingabewort das zugehörige Ausgabewort zu bestimmen.

WORTPROBLEM

Gegeben: E/A-Automat $\mathcal{A} = (Z, V, W, G, H, z_0)$, Eingabewort $v \in V$.

Gesucht: Ausgabewort w .

Das Wortproblem hat ähnlich große Bedeutung wie bei einem deterministischen Automaten. Der Unterschied ist hierbei, dass es keine Endzustände gibt und dass das Ergebnis das Ausgabewort w zum Eingabewort v ist.

6.2.4 Grundlegende Lösungsprinzipien

Das Wortproblem zur Bestimmung des Ausgabewortes w zum Eingabewort v des E/A-Automaten $\mathcal{A} = (Z, V, W, G, H, z_0)$ kann wie folgt bestimmt werden:

1. Definition des Zustands $z = z_0$;
2. Wiederholung, so lange bis das Ende des Wortes erreicht ist:
 - (a) Zugriff auf den i -ten Buchstaben v_i des Eingabewortes v ;
 - (b) Bestimmung des Ausgabebuchstabens w_i durch Anwendung der Ausgabefunktion H für den Zustand z und den Buchstaben v_i ;
 - (c) Bestimmung des Nachfolgezustands z durch Anwendung der Übergangsfunktion G für den Zustand z und den Buchstaben v_i ;
3. Ausgabe des Wortes w .

Beispiel 6.7 Im vorgestellten Getränkeautomaten in Abb. 6.10 erhalten wir beispielsweise für die folgenden Eingabewörter die nachfolgenden Zustandsfolgen und Ausgabewörter:

Eingabewort v	Zustandsfolgen	Ausgabewort w
2A	z_0, z_3, z_0	01G
5050G	$z_0, z_1 z_2, z_0$	00G
501G	z_0, z_1, z_2, z_0	050G

6.2.5 Algorithmus und Implementierung

Der Pseudocode des Algorithmus für das Wortproblem für E/A-Automaten lässt sich wie folgt angeben.

Algorithmus 15 WORTPROBLEM

Input: E/A-Automat $\mathcal{A} = (Z, V, W, G, H, z_0)$, Eingabewort $v \in V$

Output: Ausgabewort $w \in W$

Komplexität: $O(n)$

```

1:  $z = z_0$ 
2: for  $i = 1$  to  $n$  do
3:    $w_i = H(z, v_i)$ 
4:    $z = G(z, v_i)$ 

```

Allgemeine Erklärung

Der Algorithmus durchläuft iterativ die Buchstaben des Eingabewortes und erzeugt ein neues Ausgabesymbol w_i mit der Ausgabefunktion H und einen neuen Zustand z mit der Übergangsfunktion G .

Aufwandsabschätzung

Der Rechenaufwand des Algorithmus zur Lösung des Wortproblems ist linear in der Anzahl der Buchstaben n des Wortes w , also $O(n)$.

Für die Darstellung eines E/A-Automaten definieren wir eine Klasse `EA_Automat` mit den Instanzvariablen `z0` für den Anfangszustand, `G` für die Übergangsfunktion und `H` für die Ausgabefunktion. Weiterhin enthält die Klasse neben einem Konstruktor noch einige `get`-Methoden:

```

public class EA_Automat
{
    private int z0;
    private int G[][];
    private String H[][];

    public EA_Automat(int z_0, int G[][], String H[][])
    {
        this.z0 = z_0;
        this.G = G;
        this.H = H;
    }

    public int getAnzahlZustaende()
    {
        return G.length;
    }
}

```

```

public int getAnzahlAlphabet()
{
    return G[0].length;
}
public int[][] getUebergangsfunktion()
{
    return G;
}
public String[][] getAusgabefunktion()
{
    return H;
}
public int getAnfangszustand()
{
    return z0;
}
}

```

Die Implementierung eines Automaten und der Lösung des zugehörigen Wortproblems stellen wir an dem vorgestellten Beispiel des Getränkeautomaten vor:

1. Definition der Parameter des E/A-Automaten:

```

public static void main(String[] args)
{
    // ----- Eingabe -----
    // --- 1. Anfangszustand
    int z0 = 0;

    // --- 2. Übergangsfunktion
    int G[][] = {
        {1, 2, 3, 0},    // Zustand 0
        {2, 2, 3, 1},    // Zustand 1
        {2, 2, 2, 0},    // Zustand 2
        {3, 3, 3, 0},    // Zustand 3
    };

    // --- 3. Ausgabefunktion
    String H[][] = {
        {"0", "0", "0", "0"}, // Zustand 1
        {"0", "50", "50", "0"}, // Zustand 1
        {"50", "1", "2", "G"}, // Zustand 2
        {"50", "1", "2", "1G"}, // Zustand 3
    };

    // --- 4. Wort
    String v[] = {"50", "1", "A"};

    // -----
    EA_Automat a = new EA_Automat(z0, G, H);
    System.out.println("Ausgabe " + a.wortproblem(string2Int(v)));
}

```

Die Umwandlung eines Eingabewortes w aus dem Alphabet $\{50, 1, 2, A\}$ in die Codierung $0, 1, 2, 3$ erfolgt dann mit der Methode `string2Int`:

```
public static int[] string2Int(String w[])
{
    int w_idx[] = new int[w.length];
    for(int i=0; i<w.length; i++)
    {
        switch(w[i])
        {
            case "50": w_idx[i] = 0; break;
            case "1":  w_idx[i] = 1; break;
            case "2":  w_idx[i] = 2; break;
            case "A":  w_idx[i] = 3; break;
        }
    }
    return w_idx;
}
```

2. Implementierung der Methode `wortproblem` in die Klasse `EA_Automat`:

```
public String wortproblem(int v[])
{
    String w = "";
    int z = z0;
    for (int i=0; i<v.length; i++)
    {
        w = w + H[z][v[i]];
        z = G[z][v[i]];
    }
    return w;
}
```

6.2.6 Anwendungen

Wir betrachten Anwendungsbeispiele von E/A-Automaten.

Steuerung einer Produktion Die Steuerung einer Produktion hat die Aufgabe, die einzelnen Fertigungsschritte so anzusteuern, dass ein Produkt nach den definierten Vorgaben gefertigt wird. Die Eingabe v des Steuerungsprogramms enthält Informationen über das herzustellende Produkt und die Ausgabe w stellt die Steuereingriffe dar. Wir skizzieren das Prinzip der Steuerung einer Produktion an dem folgenden E/A-Automaten $\mathcal{A} = (Z, V, W, G, H, z_0)$. Die Menge der Zustände $Z = \{z_0, z_1, \dots, z_5\}$ wird wie folgt beschrieben:

Zustand	Beschreibung
z_0	Ruhezustand
z_1	Vorbereiten
z_2	Schweißen
z_3	Montage
z_4	Beschichtung
z_5	Lackierung

Bei dieser Steuerung hat das Eingabealphabet V die folgende Bedeutung:

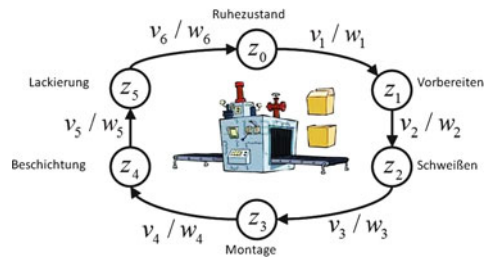
Eingabe	Bedeutung
v_1	Start wurde gedrückt
v_2	Vorbereitung der Maschine ist beendet
v_3	Schweißen des Rohlings ist beendet
v_4	Montage ist nach einer vorgegebenen Anzahl von Schritten beendet
v_5	Beschichtung aller Oberflächen ist abgeschlossen
v_6	Lackierung aller Oberflächen ist beendet

Das Ausgabealphabet W wird wie folgt definiert:

Ausgabe	Bedeutung
w_1	Material auswählen
w_2	Schweißen beginnen
w_3	Montage starten
w_4	Beschichtung anschalten
w_5	Lackierung beginnen
w_6	Programm beginnen

Die Steuerung der Produktion ist in Abb. 6.11 als deterministischer E/A-Automat dargestellt.

Abb. 6.11 Steuerung einer Produktion in Form eines deterministischen E/A-Automaten



Steuerung einer Autowaschanlage Wir modellieren einen Automaten zur Steuerung einer Autowaschanlage zum manuellen Waschen mit einem Hochdruckreiniger mit drei verschiedenen Programmen. In diesem Fall muss zunächst eine Waschmünze eingeworfen werden. Im Anschluss daran kann ein Waschprogramm durch Drücken von einem von drei verfügbaren Knöpfen ausgewählt werden. Die Einstellung des Programms kann zu beliebigen Zeitpunkten geändert werden. Wir definieren für diese Aufgabenstellung den folgenden E/A-Automaten: $\mathcal{A} = (Z, V, W, G, H, z_0)$. Die Menge der Zustände $Z = \{z_0, z_1, \dots, z_4\}$ wird wie folgt beschrieben:

Zustand	Beschreibung
z_0	Keine Aktion
z_1	Waschmünze eingeworfen
z_2	Waschprogramm P_1
z_3	Waschprogramm P_2
z_4	Waschprogramm P_3

Bei dieser Steuerung hat das Eingabealphabet V die folgende Bedeutung:

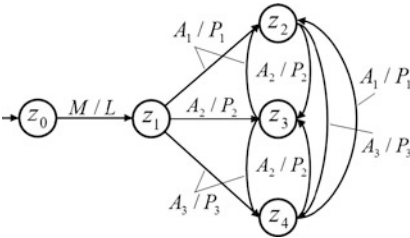
Eingabe	Bedeutung
A_1	Knopf 1 gedrückt
A_2	Knopf 2 gedrückt
A_3	Knopf 3 gedrückt

Das Ausgabealphabet W wird wie folgt definiert:

Ausgabe	Bedeutung
P_1	Waschprogramm P_1 startet
P_2	Waschprogramm P_2 startet
P_3	Waschprogramm P_3 startet

Der Autowaschautomat ist in Abb. 6.12 als deterministischer E/A-Automat dargestellt.

Abb. 6.12 Steuerung einer Waschanlage in Form eines deterministischen E/A-Automaten



6.3 Nicht deterministische Automaten und reguläre Ausdrücke

Bei deterministischen Automaten gibt es für jede Eingabefolge genau eine mögliche Zustandsfolge durch den Automaten. Das Verhalten dieser Art von Automaten ist somit eindeutig aus dem Anfangszustand und der Übergangsfunktion des Automaten vorhersehbar. In vielen technischen oder naturwissenschaftlichen Systemen und Prozessen tritt jedoch Nichtdeterminismus auf. Bei den sogenannten nicht deterministischen Automaten reicht das Wissen aus dem Anfangszustand und der Zustandsübergänge nicht aus, um ihr Verhalten eindeutig vorherzusagen. Eine alternative Variante zu nicht deterministischen Automaten sind die regulären Ausdrücke. Mit dieser relativ einfachen Beschreibungsvariante lassen sich die gleichen Anwendungen wie bei nicht deterministischen Automaten beschreiben, d. h., jeder nicht deterministische Automat kann in einen regulären Ausdruck umgewandelt werden, und umgekehrt.

6.3.1 Einführende Beispiele

In Naturwissenschaft und Technik gibt es eine ganze Reihe unterschiedlicher Anwendungen von nicht deterministischen Systemen. Beispiele dafür sind die folgenden:

- Parallele Prozesse: In einem Prozess mit mehreren Teilprozessen lässt sich nicht eindeutig vorhersagen, welcher Prozess zuerst beendet wird.
- Technisches System: Der Zeitpunkt des Wechsels zwischen den Zuständen eines technischen Systems ist nicht exakt vorhersagbar.
- Brown'sche Bewegung: Die Position der in einer Flüssigkeit oder in einem Gas schwebenden Teilchen ist nicht vorhersagbar.

Der Nichtdeterminismus spielt bei vielen ereignisdiskreten Systemen in der Praxis eine große Rolle. Beispielsweise lassen sich Fertigungsprozesse, die aus einzelnen parallelen Teilprozessen bestehen, meistens nur durch ein nicht deterministisches Modell beschreiben. Bei diskreten Systemen lassen sich eventuell auftretende Störungen des Systems nur in einer veränderten Zustandsfolge modellieren.

6.3.2 Grundlegende Begriffe

Durch die Erweiterung von deterministischen zu nicht deterministischen Automaten ist es möglich, diskrete Systeme darzustellen, deren Verhalten nicht eindeutig vorhergesagt werden kann.

Nicht deterministische Automaten Bei nicht deterministischen Automaten existieren Zustände, bei denen sich der Automat von einem gegebenen Eingabesymbol zu mehreren Nachfolgezuständen bewegen kann.

Definition 6.11 Ein *nicht deterministischer Automat* (kurz: *NEA*) ist ein 5-Tupel

$$\mathcal{A} = (Z, \Sigma, \delta, z_0, F),$$

hierbei sind

1. Z eine endliche Menge (*Zustandsmenge*),
2. Σ eine endliche Menge (*Eingabealphabet*),
3. $\delta : Z \times \Sigma \rightarrow \mathcal{P}(Z)$ ¹ eine Abbildung (*Übergangsfunktion*),
4. $z_0 \in Z$ ein Element von Z (*Anfangszustand*),
5. $F \subseteq Z$ eine endliche Menge (*Endzustandsmenge*).

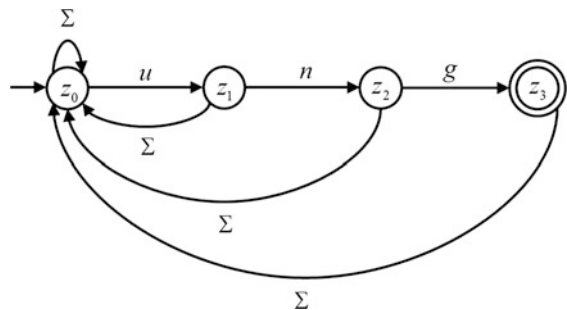
Der nicht deterministische Automat wählt den Nachfolgezustand aus, ohne über weitere Informationen zu verfügen. Im Band *Intelligente Algorithmen* erweitern wir diese Konzepte dann zu sogenannten stochastischen Automaten, bei denen die Wahrscheinlichkeit für die einzelnen möglichen Zustandsübergänge bekannt ist.

Bemerkung 6.3

1. Die Elemente $\delta(z, a)$ stellen eine Menge aller möglichen Folgezustände des Zustands z bei Eingabe von $a \in \Sigma$ dar.
2. Ein nicht deterministischer Automat kann jede Zustandsfolge durchlaufen, bei der aus dem Anfangszustand z_0 ein Nachfolgezustand $z_{n+1} \in \delta(z_n, a)$, $a \in \Sigma$ angenommen wird. Ein konkretes technisches System kann dabei natürlich nur einen Weg durch den Automatengraphen durchlaufen, während das Modell mehrere vorgibt.

Beispiel 6.8 In Abb. 6.13 ist ein Automat dargestellt, der alle Wörter mit Endung „ung“ akzeptiert. Der Anfangszustand ist z_0 und das Eingabealphabet ist $\Sigma = \{a, b, c, \dots, z\}$.

Abb. 6.13 Nicht deterministischer Automat für alle Wörter mit Endung „ung“



¹ $\mathcal{P}(Z)$: Menge aller Teilmengen von Z .

Beispielsweise ist die Menge der möglichen Nachfolgezustände von z_0 nicht ein Element, sondern eine Teilmenge von Z , also die Menge $\{z_0, z_1\}$. Das Wort wird akzeptiert, wenn es nach der Buchstabenfolge „ung“ zu Ende ist, d. h., es gibt eine Zustandsfolge des Automaten, die in dem Endzustand z_3 endet.

Definition 6.12 Die *erweiterte Übergangsfunktion* $\widehat{\delta}$ eines NEA $\mathcal{A} = (Z, \Sigma, \delta, z_0, F)$ wird induktiv definiert durch:

1. $\widehat{\delta}(z, \varepsilon) := \{z\}, \forall z \in Z,$
2. $\widehat{\delta}(z, xa) := \bigcup_{z' \in \widehat{\delta}(z, x)} \delta(z', a), \forall z \in Z, \forall x \in \Sigma^*, \forall a \in \Sigma.$

Der Nichtdeterminismus entsteht in vielen Fällen aus Unkenntnis über das Systemverhalten. Für die Vorhersage des Verhaltens muss man alle Pfade bestimmen, die der nicht deterministische Automat durchlaufen kann. Eine Zeichenkette wird akzeptiert, wenn es im Automatengraphen einen Pfad vom Anfangszustand zu einem Endzustand gibt, der mit diesem Wort beschriftet ist.

Definition 6.13 Es sei $\mathcal{A} = (Z, \Sigma, \delta, z_0, F)$ ein nicht deterministischer Automat. Dann heißt

$$L(\mathcal{A}) := \{x \in \Sigma^* \mid \widehat{\delta}(z_0, x) \cap F \neq \emptyset\}$$

die *Sprache* von $\mathcal{A}$.

Beispiel 6.9 Der Automat

$$\mathcal{A} = (\{z_0, z_1, z_2\}, \{0, 1\}, \delta, z_0, \{z_2\})$$

mit

δ	0	1
z_0	$\{z_0, z_1\}$	$\{z_0\}$
z_1	$\emptyset$	$\{z_2\}$
z_2	$\emptyset$	$\emptyset$

akzeptiert alle Wörter, die auf 01 enden, d. h.,

$$L(\mathcal{A}) = \{w \in \{0, 1\}^* \mid w \text{ endet auf } 01\}.$$

Reguläre Ausdrücke Ein regulärer Ausdruck ist eine Zeichenkette, die einer Beschreibung einer Menge von Zeichenketten auf Grundlage syntaktischer Regeln dient. Wir definieren zunächst die Syntax regulärer Ausdrücke.

Definition 6.14 Ein *regulärer Ausdruck* (RA) über einem Alphabet Σ wird wie folgt induktiv definiert:

1. Jedes Symbol ε , $\emptyset$ und $a \in \Sigma$ sind reguläre Ausdrücke.
2. Falls r_1 und r_2 reguläre Ausdrücke sind, dann sind auch $r_1 + r_2$ und $r_1 r_2$ reguläre Ausdrücke.
3. Falls r ein regulärer Ausdruck ist, dann sind auch r^* und (r) ein regulärer Ausdruck.

Beispiel 6.10 Wir betrachten das Alphabet $\Sigma = \{0, 1\}$. Daraus lassen sich beispielsweise die folgenden regulären Ausdrücke konstruieren:

- $r_1 = 1(0 + 1)^*$
- $r_2 = (0 + 00)1(0 + 1)^*$
- $r_3 = 1(\varepsilon + 0 + 00 + \varepsilon)1(0 + 1)^*$

Aus der Syntax eines regulären Ausdrucks können wir nun durch die Semantik die zugehörige Sprache zuordnen.

Definition 6.15 Einem regulären Ausdruck r über einem Alphabet Σ wird in der folgenden Form eine Sprache $L(r) \subset \Sigma^*$ zugeordnet:

1. Die Sprache der Symbole lautet:

$$L(\varepsilon) = \{\varepsilon\}, \quad L(\emptyset) = \emptyset \quad \text{und} \quad L(a) = \{a\}, \quad a \in \Sigma.$$

2. Wenn r_1 und r_2 reguläre Ausdrücke sind, so ist

$$\begin{aligned} L(r_1 + r_2) &= L(r_1) \cup L(r_2), \\ L(r_1 r_2) &= L(r_1) L(r_2). \end{aligned}$$

3. Für jeden regulären Ausdruck r ist

$$\begin{aligned} L(r^*) &= L(r)^*, \\ L((r)) &= L(r). \end{aligned}$$

Beispiel 6.11

1. $L((01)^*) = L(01)^* = (L(0)L(1))^* = (\{0\}\{1\})^* = \{01\}^* = \{\varepsilon, 01, 0101, \dots\}$
2. $L((0 + 1)^*) = L((0 + 1))^* = L(0 + 1)^* = (L(0) \cup L(1))^* = (\{0\} \cup \{1\})^* = \{0, 1\}^*$
3. $L((0 + (0 + 1)^* 00)) = \{0, 00, 000, 100, 0100, 1000, \dots\}$

Definition 6.16 Zwei reguläre Ausdrücke R und S heißen *äquivalent*, falls die Sprache identisch ist, $L(R) = L(S)$.

6.3.3 Problemstellung

In diesem Abschnitt betrachten wir einige zentrale Problemstellungen zur Analyse von Eigenschaften von nicht deterministischen Automaten und regulären Sprachen.

Wortproblem Das zentralste Problem ist wieder die Frage, ob ein Wort zur Sprache des Automaten gehört:

WORTPROBLEM

Gegeben: nicht deterministischer Automat $\mathcal{A}$, Wort $w \in \Sigma^*$.

Gesucht: Gehört das Wort zur Sprache $w \in L(\mathcal{A})$?

Ein nicht deterministischer Automat darf nur diejenigen Ereignisse, die durch Eingabewörter beschrieben werden und zur definierten Sprache des Automaten gehören, enthalten.

Äquivalenz Im folgenden Abschnitt werden wir zeigen, dass alle Beschreibungsformen aus deterministischen Automaten, nicht deterministischen Automaten und regulären Ausdrücken die gleiche Sprachmenge beschreiben. Damit lassen sich alle Modellformen durch verschiedene Methodiken ineinander umformen.

ÄQUIVALENZ NEA-DEA

Gegeben: nicht deterministischer Automat $\mathcal{N}$.

Gesucht: deterministischer Automat $\mathcal{D}$ mit $L(\mathcal{D}) = L(\mathcal{N})$.

ÄQUIVALENZ NEA-RA

Gegeben: nicht deterministischer Automat $\mathcal{N}$.

Gesucht: regulärer Ausdruck r mit $L(r) = L(\mathcal{N})$.

ÄQUIVALENZ RA-NEA

Gegeben: regulärer Ausdruck r .

Gesucht: nicht deterministischer Automat $\mathcal{N}$ mit $L(\mathcal{N}) = L(r)$.

Die Umwandlung in eine andere Modellform kann in verschiedenen Problemstellungen sehr hilfreich sein, um die Aufgabenstellung einfacher zu bearbeiten.

6.3.4 Grundlegende Lösungsprinzipien

Wir betrachten nun für die obigen Problemstellungen die zugehörigen Lösungsmethodiken.

Wortproblem Für die Überprüfung, ob ein Wort durch einen nicht deterministischen Automaten akzeptiert wird, müssen alle möglichen Zustandsübergänge betrachtet werden. Ein Wort wird akzeptiert, wenn es im zugehörigen Automatengraphen einen zugehörigen Pfad von einem Anfangszustand zu einem Endzustand gibt. Der Algorithmus muss somit alle Zustandsfolgen betrachten, die für ein gegebenes Eingabewort vom Anfangszustand zu einem Endzustand auftreten.

Prinzip des Wortproblems

1. Initialisierung der Menge $Z(0) = z_0$
2. Wiederholung, so lange bis das Ende des Wortes erreicht ist:
 - (a) Zugriff auf den i -ten Buchstaben w_i des Wortes w
 - (b) Bestimmung der Menge der Nachfolgezustände $Z(i)$ aus der Menge $Z(i - 1)$:

$$Z(i) = \{z \in Z \mid z = \delta(z', w_i), z' \in Z(i - 1)\}$$

3. Falls in der Menge $Z(n)$ ein Zustand $z \in F$ existiert, wird das Wort w akzeptiert, andernfalls nicht

Äquivalenz NEA und DEA Zunächst stellen wir die Haupteigenschaft von nicht deterministischen Automaten vor. Jede Sprache L , die durch eine NEA beschreibbar ist, kann auch durch eine DEA akzeptiert werden. Wir zeigen diese Tatsache durch die sogenannte Potenzmengenkonstruktion, die gleichzeitig die Vorschrift zur Aufstellung des zugehörigen deterministischen Automaten liefert.

Wir betrachten einen nicht deterministischen Automaten $\mathcal{N} = (Z, \Sigma, \delta, z_0, F)$. Wir konstruieren einen deterministischen Automaten $\mathcal{D} = (Z', \Sigma, \delta', z_0, F')$ mit der identischen Sprache $L(\mathcal{D}) = L(\mathcal{N})$. Die Menge der Zustände des deterministischen Automaten ist die Potenzmenge $Z' = \mathcal{P}(Z)$, d. h., der Automat $\mathcal{D}$ besitzt damit jede mögliche Teilmenge der Zustände von $\mathcal{N}$. Die zugehörige Übergangsfunktion von $\mathcal{D}$ ist dann definiert durch

$$\delta'(z', a) = \bigcup_{z \in z'} \delta(z, a), \quad z' \in Z',$$

also die Vereinigung aller Zustandsmengen $\delta(z, a)$ für alle Elemente z in der Menge z' . Die Menge der Endzustände von $\mathcal{D}$ ist

$$F' = \{Z' \subseteq Z \mid Z' \cap F \neq \emptyset\}.$$

Für alle Wörter $w = w_1 \dots w_n \in \Sigma^*$ folgt, dass aus $w \in L(\mathcal{N})$ genau dann die Bedingung $\hat{\delta}(z_0, w) \cap F \neq \emptyset$ gilt. In diesem Fall gibt es eine Folge von Teilmengen $Z_1, Z_2, \dots, Z_n$ von Z mit $\delta'(z_0, w_1) = Z_1, \delta'(Z_1, w_2) = Z_2, \dots, \delta'(Z_{n-1}, w_n) = Z_n$ und $Z_n \cap F \neq \emptyset$. Das ist genau dann der Fall, wenn $\hat{\delta}(z_0, a) \in F'$ und damit $x \in L(\mathcal{D})$.

Abb. 6.14 Nichtdeterministischer Automat für alle Wörter mit Endung „ung“

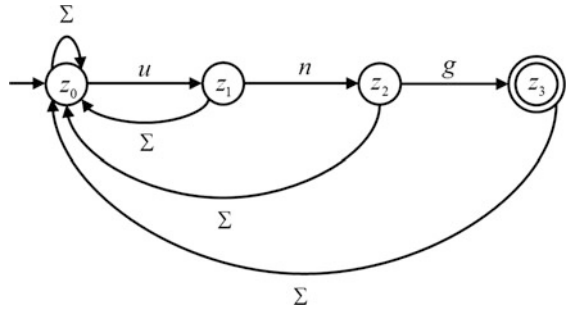
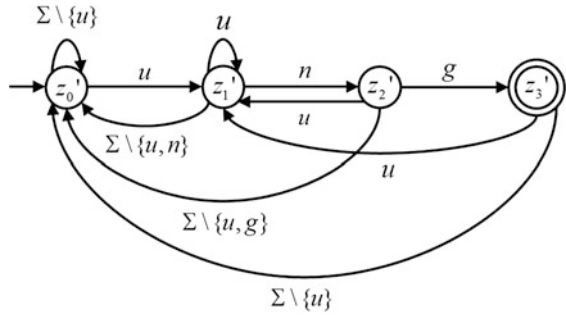


Abb. 6.15 Deterministischer Automat für alle Wörter mit Endung „ung“



Beispiel 6.12 Wir geben zu dem nicht deterministischen Automaten $\mathcal{N} = (Z, \Sigma, \delta, z_0, F)$ in Abb. 6.14 den zugehörigen deterministischen Automaten $\mathcal{D} = (Z', \Sigma, \delta', z_0, F')$ an. Im ersten Schritt des Verfahrens ist $Z' = \{z_0\}$. Für alle Eingabebuchstaben außer u bleibt der Automat in dem Zustand z_0 , also

$$z'_0 := \{z' = \delta(z_0, \Sigma \setminus \{u\})\} = \{z_0\}.$$

Für den Buchstaben u ist die Menge der Nachfolgezustände die Menge

$$z'_1 := \{z' \mid z' = \delta(z_0, u)\} = \{z_0, z_1\}.$$

Damit erhalten wir für die Übergangsfunktion des DEA:

$$\delta'(z'_0, \Sigma \setminus \{u\}) = \{z_0\} = z'_0 \quad \text{und} \quad \delta'(z'_0, u) = \{z_0, z_1\} = z'_1.$$

Auf analoge Weise erhalten wir die beiden nächsten Zustände $z'_2 = \{z_1, z_2\}$ und $z'_3 = \{z_2, z_3\}$. Der zugehörige deterministische Automat ist in Abb. 6.15 dargestellt.

Prinzip der NEA-DEA-Umwandlung

Gegeben ist ein nicht deterministischer Automat $\mathcal{N} = (Z, \Sigma, \delta, z_0, F)$ und gesucht ist ein deterministischer Automat $\mathcal{D} = (Z', \Sigma, \delta', z_0, F')$ mit der identischen Sprache $L(\mathcal{D}) = L(\mathcal{N})$:

1. Initialisierung der Zustandsmenge $Z' = \{z_0\}$.

2. Für alle $\bar{z} \in Z'$:

(a) Bestimmung für alle $a \in \Sigma$ die Menge

$$\bar{z}' = \{z' = \delta(z, a) \mid z \in \bar{z}\}.$$

(b) Definition der Übergangsfunktion:

$$\delta'(\bar{z}, a) = \bar{z}'.$$

(c) Für $\bar{z}' \notin Z'$ folgt

$$Z' = Z' \cup \{\bar{z}'\}.$$

3. Definition der Endzustände

$$F' = \{Z' \subseteq Z \mid Z' \cap F \neq \emptyset\}.$$

Äquivalenz Wir zeigen zunächst, dass zu jedem regulärem Ausdruck ein nicht deterministischer Automat mit ε -Übergängen konstruiert werden kann. Bei einem ε -Übergang kann der Automat seinen Zustand ändern, ohne ein Eingabezeichen zu lesen. Das leere Symbol $\varepsilon \notin \Sigma$ geht dabei in die Übergangsfunktion $\delta : Z \times (\Sigma \cup \varepsilon) \rightarrow \mathcal{P}(Z)$ ein.

Das Verfahren besteht aus einigen Grundregeln, die anschließend auf vorgegebene Weise verknüpft werden, um die drei Operationen $+$ (Vereinigung), $\cdot$ (Verkettung) und $*$ (Potenzbildung) von regulären Ausdrücken zu beschreiben.

Prinzip der Umwandlung eines RA in NEA

Gegeben ist ein regulärer Ausdruck r und gesucht ist ein nicht deterministischer Automat $\mathcal{N}$ mit $L(\mathcal{N}) = L(r)$. Der Automat wird durch die nachfolgenden Regeln gebildet (siehe Abb. 6.16):

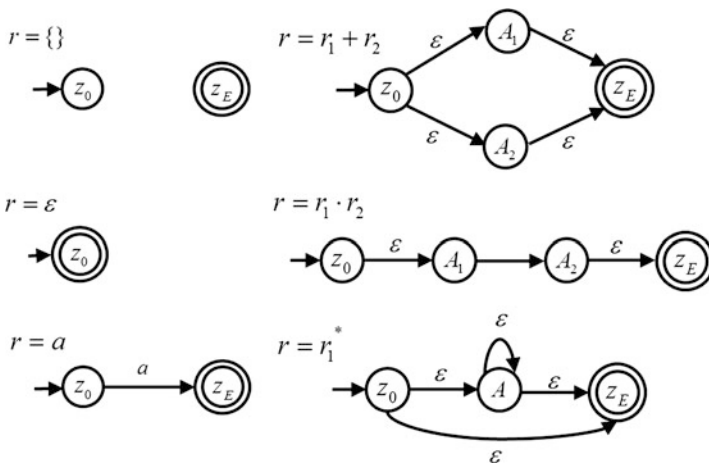


Abb. 6.16 Umwandlung eines RA in einen NEA durch 6 Regeln

1. Für $r = \emptyset$ besitzt der Automat keine Kante vom Anfangszustand z_0 zum Endzustand z_E .
2. Für $r = \varepsilon$ besitzt der Automat nur den Anfangszustand z_0 .
3. Für $r = a \in \Sigma$ besitzt der Automat eine Kante mit dem Symbol a vom Anfangszustand z_0 zum Endzustand z_E .
4. Für $r = r_1 + r_2$ werden die beiden zugehörigen Automaten für A_1 und A_2 parallel geschaltet und durch ε -Übergänge mit dem gemeinsamen Anfangszustand z_0 und dem Endzustand z_E verbunden.
5. Für $r = r_1 \cdot r_2$ werden die beiden zugehörigen Automaten für A_1 und A_2 in Reihe geschaltet und durch ε -Übergänge mit dem gemeinsamen Anfangszustand z_0 und dem Endzustand z_E verbunden.
6. Für $r = r^*$ erhält der Automat A für s eine ε -Kante zwischen seinem Anfangs- und Endzustand sowie einen neuen Anfangszustand z_0 und den Endzustand z_E , der durch einen ε -Übergang mit dem zugehörigen Automaten für s verbunden ist.

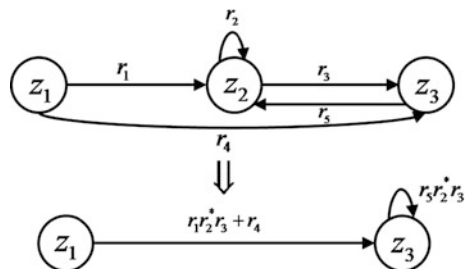
Wir beschreiben nun ein Verfahren, um aus einem NEA einen regulären Ausdruck zu bestimmen. Diese Methode beruht auf einer schrittweisen Elimination aller Automatenzustände, bis der Automatengraph nur noch aus dem Anfangs- und dem Endzustand besteht. Bei diesem Eliminationsverfahren werden die Kanten des Automatengraphen mit den jeweiligen regulären Ausdrücken beschriftet.

Prinzip der Umwandlung eines NEA in RA

Gegeben ist ein nicht deterministischer Automat $\mathcal{N}$ und gesucht ist ein regulärer Ausdruck r mit $L(r) = L(\mathcal{N})$.

1. Umformung des NEA in die folgende Form:
 - (a) Hinzufügung eines Zustands, sodass vom Startzustand keine einlaufenden Kanten vorliegen.
 - (b) Hinzufügung eines Zustands, sodass vom Endzustand keine auslaufenden Kanten vorliegen.
2. Solange noch Zustände vorhanden sind, die weder Anfangs- noch Endzustand sind, erhält man einen RA aus dem NEA durch eine Zustandselimination nach dem Prinzip aus Abb. 6.17.

Abb. 6.17 Prinzip der Zustandselimination bei Umwandlung NEA in RA



Beispiel 6.13 Wir betrachten den nicht deterministischen Automaten in Abb. 6.18. Die Umformung des Automaten in einen regulären Ausdruck erfolgt durch die folgenden Schritte:

1. Hinzufügung zweier Zustände z_4 und z_5 , sodass vom Startzustand z_0 keine einlaufenden und vom Endzustand z_3 keine auslaufenden Kanten vorliegen in Abb. 6.19.
2. Eliminierung von Zustand z_4 und z_5 in Abb. 6.20.
3. Eliminierung von Zustand z_2 und z_3 in Abb. 6.21.

Damit erhalten wir aus dem nicht deterministischen Automaten den regulären Ausdruck

$$r = 00^*1 + 00^*(1 + 10)(01)^*000^*1 + 1(01)^*000^*1.$$

Damit sind die Konzepte des DEA, NEA und des regulären Ausdrucks alle äquivalent. Alle diese Modellkonzepte beschreiben die gleiche Sprachklasse, die sogenannte reguläre Sprache.

Satz 6.2 $RA = DEA = NEA = \varepsilon - NEA$

6.3.5 Algorithmus und Implementierung

Der Umgang mit regulären Ausdrücken ist in Java bereits im Paket `java.util.regex` implementiert. Reguläre Ausdrücke dienen vor allem zur Beschreibung von Mustern, wie

Abb. 6.18 Nicht deterministischer Automat für die Umwandlung in einen regulären Ausdruck

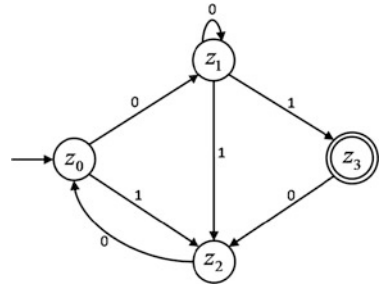
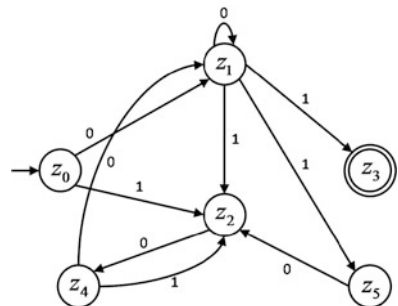


Abb. 6.19 Nicht deterministischer Automat durch Hinzufügen zweier Zustände z_4 und z_5



sie bei der Zeichenkettenverarbeitung verwendet werden. In der Klasse `Pattern` wird mit der Methode `compile` ein regulärer Ausdruck `r` durch eine Zeichenkette definiert:

```
Pattern p = Pattern.compile(r);
```

In regulären Ausdrücken werden dabei häufig verschiedene Arten von Platzhaltern verwendet:

Symbol	Erklärung
.	Beliebiges Zeichen
$X?$	Zeichen X kommt einmal oder nie vor
X^*	Zeichen X kommt keinmal oder beliebig oft vor
X^+	Zeichen X kommt einmal oder beliebig oft vor
$\backslash d$	Beliebige Ziffer
$\backslash D$	Beliebiges Zeichen, das keine Zahl ist
$\backslash s$	Leerzeichen
$\backslash S$	Beliebiges Zeichen, das kein Leerzeichen ist
$\backslash w$	Beliebiges Zeichen
$\backslash W$	Beliebiges Zeichen, das kein Wortzeichen ist

Wichtig dabei ist, dass in Strings das Backslashzeichen (`\`) durch einen doppelten Backslash (`\\`) ersetzt werden muss.

Abb. 6.20 Nicht deterministischer Automat durch Eliminierung von Zustand z_1 und z_2

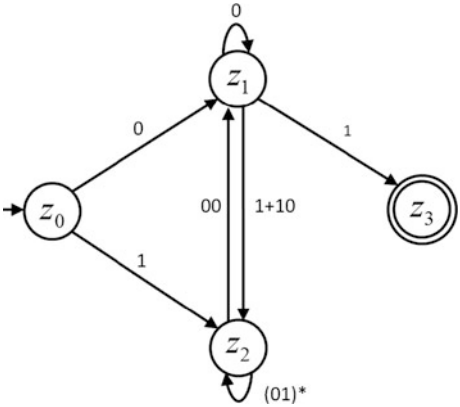
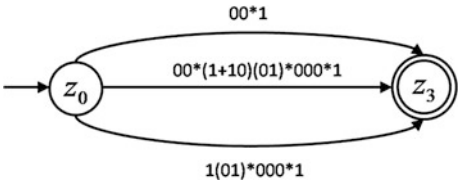


Abb. 6.21 Nicht deterministischer Automat durch Eliminierung von Zustand z_4 und z_5



Mithilfe der Klasse `Matcher` und der gleichnamigen Methode `matcher` kann nun ein sogenanntes `Matcher`-Objekt zum Testen einer Zeichenkette `s` angelegt werden.

```
Matcher m = p.matcher(s);
```

Mit der Methode `matches` kann nun für dieses `Matcher`-Objekt getestet werden, ob die zugehörige Zeichenkette `s` durch den obigen regulären Ausdruck erzeugt wird:

```
boolean b = m.matches();
```

Eine alternative kürzere Variante ist auch die folgende Form:

```
boolean b = Pattern.matches(r, s);
```

Beispiel 6.14 Der folgende Code testet, ob eine beliebige Zeichenkette durch den regulären Ausdruck `(01)*` erzeugbar ist:

```
Pattern p = Pattern.compile("(01)*");  
Matcher m = p.matcher("0101");  
System.out.println(m.matches());
```

Ausgabe

```
true
```

Mit der Methode `split` kann man mithilfe von regulären Ausdrücken eine beliebige Zeichenkette in mehrere Teile aufspalten.

```
String[] = split(CharSequence input);  
String[] = split(CharSequence input, int limit);
```

Mit dem zweiten Parameter wird die Anzahl der Teile definiert, in die die Zeichenkette maximal aufgeteilt werden soll.

Beispiel 6.15 Die folgenden beiden Befehle zerlegen den Eingabestring in Teilwörter:

```
Pattern p1 = Pattern.compile("\\s");  
String result[] = p1.split("Eins Zwei Drei Vier");
```

Ausgabe

```
Eins  
Zwei  
Drei  
Vier
```

Mit der Methode `find` der Klasse `Matcher` kann man feststellen, ob sich eine durch einen regulären Ausdruck beschriebene Teilfolge in einem String befindet. Die Methode `group` liefert den erkannten Substring und die beiden Methoden `start` und `end` die zugehörigen Positionen.

Beispiel 6.16 Die folgenden Programmzeilen suchen alle Zahlen, inklusive der zugehörigen Positionen aus dem Eingabestring heraus:

```
String s = "Eins 01, Zwei 02, Drei 3, Vier 004";
Matcher m = Pattern.compile("\\d+").matcher(s);
while(m.find())
    System.out.printf(" %s: (%d, %d)", m.group(), m.start(), m.end());
```

Ausgabe

```
01: (5, 7)
02: (14, 16)
3: (23, 24)
004: (31, 34)
```

Neben dem Suchen kann auch an der Fundstelle mithilfe der Methode `appendReplacement` der Klasse `Matcher` die Operation Ersetzen vorgenommen werden. Falls in einem String `text` ein Muster `r` erkannt wird, ersetzt die Methode `appendReplacement` es durch eine Alternative, die in den StringBuffer `sb` gespeichert wird.

```
Matcher m = Pattern.compile(r).matcher(text);
appendReplacement(StringBuffer sb, String s);
```

Beispiel 6.17 Der folgende Programmcode wendet die Methode „Suchen und Ersetzen“ an:

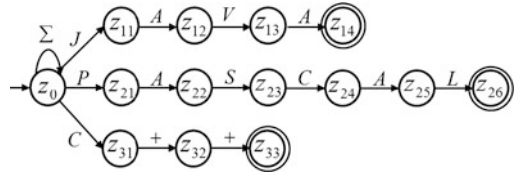
```
String s = "Eins 01, Zwei 02, Drei 3, Vier 004";
Matcher m = Pattern.compile("0+").matcher(s);
StringBuffer sb = new StringBuffer();
while (m.find())
    m.appendReplacement(sb, "");
m.appendTail(sb);
System.out.println(sb);
```

Mit der Methode `appendTail` wird das noch verbleibende Teilstück in den StringBuffer gespeichert.

Ausgabe

```
Eins 1
Zwei 2
Drei 3
Vier 4
```

Abb. 6.22 Textsuche mit einem nicht deterministischen Automaten nach den Wörtern JAVA, PASCAL, C++



6.3.6 Anwendungen

Wir betrachten einige Anwendungen von nicht deterministischen Automaten und regulären Ausdrücken.

Textsuche Die Aufgabe der Textsuche ist es, eine Menge von Wörtern in einem Text zu finden. Mit einem nicht deterministischen Automaten können wir dieses Problem modellieren, sodass dieser immer beim Übergang in einen akzeptierenden Zustand angibt, dass er eines der gesuchten Wörter gefunden hat. Ein nicht deterministischer Automat $\mathcal{N} = (Z, \Sigma, \delta, z_0, F)$, der diese Aufgabe erledigt, besitzt den folgenden Aufbau:

1. Der Startzustand z_0 besitzt eine Schlinge für jedes Eingabezeichen aus dem Alphabet Σ .
2. Für alle zu suchenden Wörter $w_1, \dots, w_k$ werden k Zustände $z_{11}, z_{21}, \dots, z_{k1}$ mit dem Übergangszeichen des ersten Buchstaben des jeweiligen Wortes definiert.
3. Für jedes Wort w_i für $i = 1, \dots, k$ der Länge n_i werden $n_i - 1$ weitere Zustände $z_{i2}, z_{i3}, \dots, z_{in_i}$ mit dem Übergangszeichen des l -ten Buchstaben ($2 \leq l \leq n_i$) zwischen dem Zustand z_{il-1} und z_{il} des jeweiligen Wortes definiert.
4. Die Menge der Endzustände F sind die Zustände $z_{1n_1}, z_{2n_2}, \dots, z_{kn_k}$.

In Abb. 6.22 ist ein Beispiel einer Textsuche nach drei Wörtern dargestellt.

Warteschlange vor einer Fertigungseinrichtung Wir betrachten eine Warteschlange vor einem Fertigungssystem mit drei parallelen Warteschlangen mit jeweils 1, 2 und 3 Plätzen. Der Zustand dieses Wartesystems setzt sich aus insgesamt $2 \cdot 3 \cdot 4 = 24$ möglichen Zuständen zusammen:

$$Z_2 = \left\{ \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 2 \\ 0 \end{pmatrix}, \dots, \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \right\}.$$

Für das Wartesystem gibt es nun zwei verschiedene Ereignisse:

- neues Bauteil kommt zur Warteschlange hinzu,
- Bauteil verlässt die Warteschlange.

Für das Wartesystem mit zwei Warteschlangen erhält man den nicht deterministischen Automaten mit einem Zustandsübergang zwischen zwei Zuständen z_i und z_j , die sich in genau einem Eintrag unterscheiden. Die Schlinge am rechten Knoten beschreibt das Abweisen von Teilen, wenn beide Warteschlangen voll sind.

Natürliche Sprache Viele Konzepte der natürlichen Sprache lassen sich durch reguläre Ausdrücke darstellen. Das Alphabet enthält alle lateinischen Buchstaben und Umlaute. Beispielsweise lässt sich durch den regulären Ausdruck

$$r = (\text{addier} + \text{subtrahier})(e, st, t, en)$$

die folgende reguläre Sprache darstellen:

$$L(r) = \{\text{addiere, addierst, addiert, addieren,} \\ \text{subtrahiere, subtrahierst, subtrahiert, subtrahieren}\}.$$

Entwurf von Programmiersprachen Viele syntaktische Konzepte in Programmiersprachen lassen sich sehr gut durch reguläre Ausdrücke beschreiben. In Java werden beispielsweise die Bezeichner von Variablen eine Folge von Buchstaben, Ziffern und Symbol, wobei das erste Zeichen keine Ziffer sein darf. Ein Bezeichner lässt sich damit durch den folgenden regulären Ausdruck beschreiben:

$$r = (a + b + \dots + z)(a + b + \dots + z + 0 + 1 + \dots + 9)^*.$$

Eine beliebige Zahl mit und ohne Vorzeichen und mit beliebig vielen Dezimalstellen nach dem Komma lässt sich durch diesen regulären Ausdruck beschreiben:

$$r = (\varepsilon + „+“ + „-“)(0 + 1 + \dots + 9)^*, (0 + 1 + \dots + 9)^*.$$

Suche von Textmustern Reguläre Ausdrücke können sehr effizient verwendet werden, um gewisse Textmuster in einer Menge von Wörtern zu suchen. Hierzu werden die Klassen von Textmustern als Suchkommandos durch reguläre Ausdrücke beschrieben. Beispielsweise lässt sich die Suche nach einer Zeichenkette mit Ziffern durch den folgenden regulären Ausdruck definieren:

$$r = (a + b + \dots + z)^*(0 + 1 + \dots + 9)^*(a + b + \dots + z)^*.$$

Eine Adresse wie beispielsweise „Kronenstraße 16, 78532 Tuttlingen“ lässt sich dann wie folgt spezifizieren:

$$r = (a + b + \dots + z)^+(\text{straße} + \text{str.} + \text{weg} + \text{gasse})(\\ (0 + 1 + \dots + 9)^*(,)(0 + 1 + \dots + 9)^5()(a + b + \dots + z)^+.$$

6.4 Grammatiken

Einige ereignisdiskrete Systeme sind nicht durch endliche Automaten zu modellieren. Viele dieser Systeme erzeugen keine regulären Sprachen mehr und lassen sich damit nur durch „höhere“ Automaten darstellen wie beispielsweise Kellerautomaten oder Turing-Maschinen. Diese Automaten können jedoch leider nicht mehr so einfach analysiert werden, wie es bei den Automaten der Fall war. Eine äquivalente Darstellungsform dieser Automatenmodelle sind die sogenannten Grammatiken, eine Art von Regelwerk, nach dem alle möglichen Folgen dieser Sprache konstruierbar sind.

Grammatiken besitzen einen vielfältigen Anwendungsbereich, der von der linguistischen Beschreibung von Sprache, der Dialogverarbeitung in der Mensch-Maschine-Kommunikation bis zur Definition von Programmiersprachen oder den Bau von Compilern reicht. In der Sprachverarbeitung ist die Grammatik eine gut geeignete Darstellungsform, um korrekt formulierte Sätze zu generieren.

6.4.1 Einführendes Beispiel

Wir betrachten eine einfache Grammatik zur Darstellung von arithmetischen Ausdrücken. Sei $\Sigma = \{ (,), +, -, *, /, a \}$ ein Alphabet, bestehend aus den runden Klammern, den vier Grundrechenoperationen und einem Symbol a , das als Platzhalter für beliebige Konstanten oder Variablen dient. Die Grammatik zur Erzeugung beliebiger korrekter arithmetischer Ausdrücke lässt sich dann durch eine Variablenmenge $N = \{ S, T \}$ mit einem Startsymbol S und einer Menge von Ableitungsregeln darstellen:

$$R = \left\{ \begin{array}{l} S \rightarrow a \\ S \rightarrow T \\ T \rightarrow T * T \\ T \rightarrow T / T \\ T \rightarrow (T + T) \\ T \rightarrow (T - T) \\ T \rightarrow a \end{array} \right.$$

Das Startsymbol S kennzeichnet dabei den Beginn der Ableitung eines Wortes. Ein Wort $w \in \Sigma^*$ wird von der Grammatik erzeugt, wenn es vom Startsymbol durch eine Folge von Ableitungen aus der Menge R darstellbar ist. Beispielsweise sind dann

$$(a + a) * a, ((a - a) * a * a + a), (a/a + ((a + a) + a))$$

korrekte Ausdrücke, während

$$(a + a), ((a + a)/a), a + a$$

nicht korrekte Ausdrücke darstellen. Beispielsweise erfolgt die Erzeugung des Ausdrucks $(a + a) * (a - a) * a$ durch die folgenden Ableitungen:

$$\begin{aligned} S &\Rightarrow T \Rightarrow T * T \Rightarrow T * T * T \Rightarrow (T + T) * T * T \Rightarrow (T + T) * (T - T) * T \\ &\Rightarrow (a + a) * (a - a) * a. \end{aligned}$$

Im letzten Schritt wurden alle Ableitungen der Form $T \rightarrow a$ zusammengefasst.

6.4.2 Grundlegende Begriffe

Grammatiken bestehen aus zwei verschiedenen Arten von Symbolen, den sogenannten Terminalsymbolen (Eingabealphabet) und den Nichtterminalsymbolen. Weiterhin müssen sie Regeln der Form $\alpha \rightarrow \beta$ besitzen, in der α der linken und β der rechten Seite entsprechen. Die Anwendung einer Regel bedeutet, dass in dem erzeugten Wort ein Teilwort existiert, das einer linken Regelseite entspricht, die durch die rechte Seite ersetzt wird. Diese Ableitungsschritte werden so lange ausgeführt, bis das Wort nur noch aus Terminalsymbolen besteht. Jedes Wort, das durch solche Ableitungsschritte erzeugt werden kann, gehört zur Sprache der Grammatik.

Definition 6.17 Eine *Grammatik* ist ein 4-Tupel

$$G = (N, \Sigma, R, S),$$

hierbei sind

1. N eine endliche Menge (*Nichtterminalsymbole*),
2. Σ eine endliche Menge (*Terminalsymbole*),
3. $R \subseteq (N \cup \Sigma)^+ \times (N \cup \Sigma)^*$ eine endliche Menge (*Ableitungsregeln*),
4. S ein Element von N (*Startsymbol*).

Aus der Definition einer Grammatik erfolgt unmittelbar die Definition einer Ableitung.

Definition 6.18 Eine *Ableitung* ist eine Folge von Wörtern $(w_0, w_1, \dots, w_n)$ mit $w_0 \in S$, $w_i \in (N \cup \Sigma)^*$ und $w_n \in \Sigma^*$ mit

$$w_0 \Rightarrow w_1 \Rightarrow \dots \Rightarrow w_n.$$

Für zwei Wörter $w, w' \in (N \cup \Sigma)^*$ folgt die Ableitung $w \Rightarrow w'$, falls eine Ableitungsregel $\alpha \rightarrow \beta \in R$ existiert, sodass $w = x\alpha y$ und $w' = x\beta y$ für zwei Wörter $x, y \in (N \cup \Sigma)^*$. Wir schreiben $w \stackrel{*}{\Rightarrow} w'$, falls eine Menge von Wörtern $w_1, \dots, w_n \in (N \cup \Sigma)^*$ existiert, mit $w = w_1 \Rightarrow \dots \Rightarrow w_n = w'$ oder $w = w'$.

Die Sprache einer Grammatik ist die Menge aller Wörter $w \in \Sigma^*$, die aus dem Startsymbol durch eine Ableitung erzeugbar sind.

Definition 6.19 Es sei $G = (N, \Sigma, R, S)$ eine Grammatik. Dann heißt

$$L(G) := \{w \in \Sigma^* \mid S \Rightarrow^* w\}$$

die *Sprache* von G .

Das Ableiten eines Wortes ist im Allgemeinen nicht deterministisch, d. h., für ein gegebenes Wort w_i kann es mehrere verschiedene Wörter w_{i+1} geben mit $w_i \Rightarrow w_{i+1}$. Die grafische Darstellung einer Ableitung entspricht einem Baum mit der Wurzel S , wobei die Blätter dieses Baumes dann die Wörter sind.

Beispiel 6.18 Die Grammatik $G = (\{S, A, B\}, \{0, 1\}, R, S)$ mit den Ableitungsregeln

$$R = \begin{cases} S \rightarrow 1A \\ A \rightarrow 1B \\ B \rightarrow 0B \\ B \rightarrow 0 \end{cases}$$

erzeugt wie der Automat

$$\mathcal{A} = (\{z_0, z_1, z_2, z_3\}, \{0, 1\}, \delta, z_0, \{z_2\})$$

mit der Übergangsfunktion

δ	0	1
z_0	z_3	z_1
z_1	z_3	z_2
z_2	z_2	z_3
z_3	z_3	z_3

alle Wörter des regulären Ausdrucks 110^* .

Für jede Grammatik ist es möglich, einen entsprechenden Automaten zu konstruieren, welcher die von der Grammatik erzeugten Wörter akzeptiert. Der Typ des Automaten hängt dabei von dem Typ der Grammatik in der sogenannten Chomsky-Hierarchie ab. Noam Chomsky teilte die Menge der Grammatiken in vier verschiedene Typen ein. Damit kann man eindeutig an dem Typ der Grammatik ablesen, welchem Automatenmodell sie zuordenbar ist. Für eine Modellbildung eines ereignisdiskreten Systems hat das den Vorteil, dass entweder das zu modellierende Problem durch einen Automaten oder durch eine äquivalente Grammatik beschreibbar ist.

Definition 6.20 Eine Grammatik $G = (N, \Sigma, R, S)$ heißt:

- *Rechtslinear* oder vom Typ 3, wenn alle Regeln in R die Form

$$A \rightarrow aB, A \rightarrow B, A \rightarrow a, A \rightarrow \varepsilon$$

mit $A, B \in N$ und $a \in \Sigma$ besitzen (äquivalent linkslinear).

- *Kontextfrei* oder vom Typ 2, wenn alle Regeln in R die folgende Form besitzen:

$$A \rightarrow \gamma \quad \text{mit} \quad A \in N, \gamma \in (N \cup \Sigma)^*.$$

- *Kontextsensitiv* oder vom Typ 1, wenn alle Regeln in R die Form $\alpha A \gamma \rightarrow \alpha \beta \gamma$ besitzen, mit $\alpha, \beta, \gamma \in (N \cup \Sigma)^*$ und $A \in N, \beta \neq \varepsilon$. Die Regel $S \rightarrow \varepsilon$ ist erlaubt, wenn es keine Regel $\alpha \rightarrow \beta S \gamma$ gibt.
- *Chomsky-Grammatik* vom Typ 0, wenn alle Regeln in R die Form $\alpha \rightarrow \beta$ besitzen, mit $\alpha \in (N \cup \Sigma)^* \setminus \Sigma^*$ und $\beta \in (N \cup \Sigma)^*$

Bei einer kontextfreien Regel vom Typ $A \rightarrow \gamma$ kann die Variable A unabhängig vom Kontext, in dem A steht, durch γ ersetzt werden. Bei einer kontextsensitiven Grammatik ist es hingegen möglich, A durch β zu ersetzen, wenn die Variable A im „Kontext“ zwischen α und γ steht. Die abgeleiteten Wörter $w_i \rightarrow w_{i+1}$ einer kontextsensitiven Grammatik haben die Eigenschaft $|w_i| < |w_{i+1}|$.

Beispiel 6.19

1. Die Grammatik $\mathcal{G} = \{\{S, T\}, \{(\cdot), +, -, *, /, a\}, R, S\}$ mit den Ableitungsregeln

$$R = \left\{ \begin{array}{l} S \rightarrow a \\ S \rightarrow T \\ T \rightarrow T * T \\ T \rightarrow T / T \\ T \rightarrow (T + T) \\ T \rightarrow (T - T) \\ T \rightarrow a \end{array} \right.$$

erzeugt die Sprache aller arithmetischen Ausdrücke und ist vom Typ 2.

2. Die Grammatik $\mathcal{G} = \{\{S\}, \{a, b\}, R, S\}$ mit den Ableitungsregeln

$$R = \left\{ \begin{array}{l} S \rightarrow ab \\ S \rightarrow aSb \end{array} \right.$$

erzeugt die Sprache

$$L(\mathcal{G}) = \{a^n b^n \mid n \geq 1\}$$

und ist vom Typ 2.

3. Die Grammatik $G = (\{S, A, B\}, \{a, b, c\}, R, S)$ mit den Ableitungsregeln

$$R = \begin{cases} S \rightarrow aSBC \\ S \rightarrow aBC \\ CB \rightarrow BC \\ aB \rightarrow ab \\ bB \rightarrow bb \\ bC \rightarrow bc \\ cC \rightarrow cc \end{cases}$$

erzeugt die Sprache

$$L(G) = \{a^n b^n c^n \mid n \geq 1\}$$

und ist vom Typ 1. Beispielsweise erhalten wir das Wort $w = aabbcc$ aus der Ableitung

$$\begin{aligned} S &\Rightarrow aSBC \Rightarrow aaBCBC \Rightarrow aaBBCC \Rightarrow aabBCC \\ &\Rightarrow aabbCC \Rightarrow aabbcC \Rightarrow aabbcc. \end{aligned}$$

Die Chomsky-Normalform ist eine Normalform für kontextfreie Grammatiken, die unter anderem beim CYK-Algorithmus zur Lösung des Wortproblems verwendet wird.

Definition 6.21 Eine Grammatik $G = (N, \Sigma, R, S)$ ist in *Chomsky-Normalform*, wenn jede Ableitungsregel R eine der folgenden Formen annimmt:

$$A \rightarrow BC, \quad A \rightarrow a, \quad S \rightarrow \varepsilon,$$

wobei $A, B, C \in N$, $a \in \Sigma$ und $S \rightarrow \varepsilon$ nur dann zur Grammatik gehören, wenn S nicht auf der rechten Seite einer Produktion steht.

Für jede kontextfreie Sprache gibt es eine Grammatik in Chomsky-Normalform. Man kann die folgenden Korrespondenzen zwischen Sprachklasse und Automaten zeigen:

Hierarchieebene	Sprachklasse	Automat
Typ 3	Reguläre Sprachen	Endliche Automaten
Typ 2	Kontextfreie Sprachen	Kellerautomaten
Typ 1	Kontextsensitive Sprachen	Linear beschränkte Turing-Maschinen
Typ 0	Rekursiv aufzählbare Sprachen	Turing-Maschinen

6.4.3 Problemstellung

Wir betrachten einige praktische Problemstellungen im Zusammenhang mit Grammatiken und Automaten.

Wortproblem Die wichtigste Problemstellung im Bereich von Grammatiken ist die Frage, ob ein Wort zur Sprache der Grammatik gehört.

WORTPROBLEM

Gegeben: Grammatik $G = (N, \Sigma, R, S)$, $w \in \Sigma^*$.

Gesucht: Gehört das Wort zur Sprache $w \in L(G)$?

Das Wortproblem ist nur für Typ-3-, Typ-2- und für Typ-1-Sprachen entscheidbar, d. h., es existiert ein Algorithmus, der in endlicher Zeit entscheiden kann, ob ein Wort zur Sprache der Grammatik gehört. Für reguläre Grammatiken ist das Wortproblem in $O(n)$, für kontextfreie mit dem CYK-Algorithmus in $O(n^3)$ und für kontextsensitive nur in exponentieller Laufzeit berechenbar.

Äquivalenz In einigen praktischen Anwendungen ist man auch daran interessiert, zu einer regulären Grammatik den passenden Automaten zu konstruieren.

ÄQUIVALENZ 1

Gegeben: reguläre Grammatik $G = (N, \Sigma, R, S)$.

Gesucht: nicht deterministischer Automat $\mathcal{N}$ mit $L(\mathcal{N}) = L(G)$.

ÄQUIVALENZ 2

Gegeben: deterministischer Automat $\mathcal{A}$.

Gesucht: reguläre Grammatik $G = (N, \Sigma, R, S)$ mit $L(G) = L(\mathcal{A})$.

Wir werden im Nachfolgenden eine sehr einfache Vorschrift vorstellen, um eine reguläre Grammatik in einen Automaten oder einen Automaten in eine reguläre Grammatik umzuwandeln.

6.4.4 Grundlegende Lösungsprinzipien

Wir stellen nun die Lösungsprinzipien der oben vorgestellten Problemstellungen vor.

Wortproblem Der Cocke-Younger-Kasami-Algorithmus (CYK-Algorithmus) ist ein Algorithmus auf Basis des Prinzips der dynamischen Programmierung, mit dem sich feststellen lässt, ob ein Wort zu einer bestimmten kontextfreien Sprache gehört. Die Eingabe sind eine kontextfreie Grammatik $G = (N, \Sigma, R, S)$ in der sogenannten Chomsky-Normalform und das zu prüfende Wort $w = w_1 w_2 \dots w_n \in \Sigma^*$. Der CYK-Algorithmus konstruiert auf Basis der dynamischen Programmierung² die folgende

² Siehe Band *Grundlagen*, Kapitel Entwurf von Algorithmen.

dreieckige Tabelle:

w_1	w_2	w_3	w_4	w_5
X_{11}	X_{22}	X_{33}	X_{44}	X_{55}
X_{12}	X_{23}	X_{34}	X_{45}	
X_{13}	X_{24}	X_{35}		
X_{14}	X_{25}			
X_{15}				

Auf der horizontalen Achse werden die Positionen des Eingabewortes $w = w_1 w_2 \cdots w_n$ dargestellt. Der Tabelleneintrag X_{ij} stellt die Menge aller Nichtterminalvariablen A dar, sodass $A \xRightarrow{*} w_i w_{i+1} \dots w_j$. Die Aufgabe ist dann die Berechnung des Tabellenwertes X_{1n} , da mit $S \in X_{1n}$ die Ableitung $S \xRightarrow{*} w_1 w_2 \dots w_n$ des gegebenen Eingabewortes erfolgt.

Die Tabelle wird dabei mithilfe des dynamischen Programmierens von oben nach unten zeilenweise gefüllt. Die 1. Zeile betrachtet Wörter der Länge 1, die 2. Zeile Wörter der Länge 2 usw. bis zur n -ten Zeile, die genau dem Eingabewort w entspricht.

Für die Zeile 2 bis n gilt nun die folgende Überlegung: Die Variable A ist in X_{ij} , wenn es die Variablen B und C sowie eine Zahl $i \leq k < j$ gibt, mit $B \in X_{ik}$, $C \in X_{k+1j}$ und eine Ableitungsregel $A \rightarrow BC$ in R existiert.

Prinzip des CYK-Algorithmus

1. Für Zeile 1: Berechnung der ersten Zeile X_{ii} als die Menge der Variablen A , sodass $A \rightarrow w_i$ eine Ableitungsregel von G für alle Buchstaben w_i , $i = 1, \dots, n$ ist
2. Für Zeile 2– n : Berechnung der Elemente X_{ij} durch die Bestimmung der Paare aller bereits berechneten Mengen:

$$(X_{ii}, X_{i+1,j}), (X_{i,i+1}, X_{i+2,j}), \dots, (X_{ij-1}, X_{jj})$$

3. Bestimmung aller Nichtterminalsymbole A aus der Menge X_{ij} der Ableitungsregel $A \rightarrow BC$, sodass die Variablen B und C sowie eine Zahl $i \leq k < j$ existieren, mit $B \in X_{ik}$, $C \in X_{k+1j}$
4. Falls $S \in X_{1,n}$, dann ist $w \in L(G)$

Beispiel 6.20 Wir betrachten die Grammatik $G = \{\{S, A, B, C\}, \{a, b\}, R, S\}$ mit den Ableitungsregeln

$$R = \begin{cases} S \rightarrow AB|BC \\ A \rightarrow BA|a \\ B \rightarrow CC|b \\ C \rightarrow AB|a \end{cases}$$

und dem Eingabewort $w = baaba$. Die Aufgabe ist zu entscheiden, ob das Wort w durch die Grammatik G erzeugt werden kann. Mithilfe des CYK-Algorithmus erhalten wir die folgende Tabelle:

b	a	a	b	a
$\{B\}$	$\{A, C\}$	$\{A, C\}$	$\{B\}$	$\{A, C\}$
$\{S, A\}$	$\{B\}$	$\{S, C\}$	$\{S, A\}$	
—	$\{B\}$	$\{B\}$		
—	$\{S, A, C\}$			
$\{S, A, C\}$				

Die 1. Zeile der Tabelle erhalten wir, indem wir in den Ableitungsregeln der Grammatik nachschauen, welche der Nichtterminalvariablen die gegebenen Buchstaben des Eingabewortes $w = w_1 w_2 \dots w_n = baaba$ erzeugen. In diesem Fall sind das für a die Menge $\{A, C\}$ und für b die Menge $\{B\}$. Damit bekommen wir die Werte $X_{11} = X_{14} = \{B\}$ und $X_{22} = X_{33} = X_{55} = \{A, C\}$.

Die 2. Zeile ergibt sich dann aus den folgenden Rechnungen:

$$\begin{aligned}
 X_{12} &= X_{11} X_{22} = \{BA, BC\} = \{S, A\}, \\
 X_{23} &= X_{22} X_{33} = \{AA, AC, CA, CC\} = \{B\}, \\
 X_{34} &= X_{33} X_{44} = \{AB, CB\} = \{S, C\}, \\
 X_{45} &= X_{44} X_{55} = \{BA, BC\} = \{S, A\}.
 \end{aligned}$$

Analog dazu folgen die Zeilen 3 bis 5 der Tabelle aus den folgenden Termen:

$$\begin{aligned}
 X_{13} &= X_{11} X_{23} \cup X_{12} X_{33}, \\
 X_{24} &= X_{22} X_{34} \cup X_{23} X_{44}, \\
 X_{35} &= X_{33} X_{45} \cup X_{34} X_{55}, \\
 X_{14} &= X_{11} X_{24} \cup X_{12} X_{34} \cup X_{13} X_{44}, \\
 X_{25} &= X_{22} X_{35} \cup X_{23} X_{45} \cup X_{24} X_{55}, \\
 X_{15} &= X_{11} X_{25} \cup X_{12} X_{35} \cup X_{13} X_{45} \cup X_{14} X_{55}.
 \end{aligned}$$

Da das Startsymbol S ein Element der Menge X_{15} ist, kann das Wort w von der Grammatik G erzeugt werden.

Äquivalenz Im ersten Fall betrachten wir einen deterministischen Automaten $A = (Z, \Sigma, \delta, z_0, F)$, den wir in eine Grammatik $G = (N, \Sigma, R, S)$ vom Typ 3 umformen, sodass $L(A) = L(G)$. Die Menge der Nichtterminalsymbole N der Grammatik G

entspricht dabei genau der Menge der Zustände Z und die Startvariable S ist der Anfangszustand z_0 des Automaten $\mathcal{A}$. Die Ableitungsregeln R werden dann wie folgt aus der Übergangsfunktion δ bestimmt:

$$R = \begin{cases} z \rightarrow az', & \text{falls } z' = \delta(z, a), a \in \Sigma, z' \notin F, \\ z \rightarrow a, & \text{falls } z' = \delta(z, a), a \in \Sigma, z' \in F. \end{cases}$$

Der Übergang im Automaten von einem Zustand z zu einem Nachfolgezustand z' wird als Verlängerung der Zeichenkette w um den Buchstaben $a \in \Sigma$ durch diese Ableitungsregel dargestellt. Falls z' zur Menge der Endzustände F des Automaten gehört, wird z nur durch a ersetzt.

Im zweiten Fall betrachten wir eine Grammatik $G = (N, \Sigma, R, S)$ vom Typ 3. Die Aufgabe ist es, einen nicht deterministischen Automaten $A = (Z, \Sigma, \delta, z_0, F)$ aufzustellen, der die gleiche Sprache wie die Grammatik G erzeugt. Dazu definieren wir die Zustandsmenge $Z = N \cup \{X\}$ aus der Menge der Nichtterminalsymbole N mit einem zusätzlichen Zustand $X \notin N$. Der Anfangszustand z_0 entspricht dann wieder dem Startsymbol S und die Menge der Endzustände des Automaten ist $F = \{X\}$. Die Übergangsfunktion δ wird aus der Ableitungsregel R wie folgt gebildet:

$$\delta : \begin{cases} B = \delta(A, a), & \text{falls } A \rightarrow aB \in R, \\ X = \delta(A, a), & \text{falls } A \rightarrow a \in R. \end{cases}$$

Der Automat $\mathcal{A}$ führt dann für jede Regel den entsprechenden Zustandsübergang A zum Nachfolgezustand B bei Eingabe von $a \in \Sigma$ aus. Für einen Übergang von A nach X geht der Automat in seinen Endzustand über.

6.4.5 Algorithmus und Implementierung

Wir zeigen nun einmal exemplarisch eine einfache Implementierung des CYK-Algorithmus für das Testen, ob ein Eingabewort $w \in \Sigma^*$ durch eine Grammatik $G = (N, \Sigma, R, S)$ in Chomsky-Normalform abbildbar ist. Zunächst stellen wir den zugehörigen Pseudocode des Verfahrens vor. Der Algorithmus basiert auf dem vorgestellten Verfahren der dynamischen Programmierung zur Erstellung einer dreieckigen Tabelle X .

Algorithmus 16 CYK-ALGORITHMUS

Input: Grammatik $G = (N, \Sigma, R, S)$ in CN-Form, Wort $w \in \Sigma^*$
Output: 1, falls $w \in L(G)$; andernfalls 0
Komplexität: $O(n^3)$

```

1: for  $i = 1$  to  $n$  do
2:   for  $(\alpha \rightarrow \beta) \in R$  do
3:     if  $\beta = w_i$  then
4:        $X_{ii} = X_{ii} \cup \{\alpha\}$ 
5:   for  $j = 2$  to  $n$  do
6:     for  $i = 1$  to  $n - j + 1$  do
7:       for  $k = 1$  to  $j - 1$  do
8:          $X_{i,j} = X_{i,j} \cup \{\alpha \mid \exists B, C \in N, \alpha \rightarrow BC \in R, B \in X_{i,k}, C \in X_{i+k,j-k}\}$ 
9:   if  $S \in X_{1n}$  then
10:    return 1
11:  else
12:    return 0

```

Allgemeine Erklärung

In Zeile 1–4 wird die erste Zeile der Matrix X aus den Ableitungsregeln der Form $A \rightarrow a$ mit $A \in N$ und $a \in \Sigma$ der gegebenen Grammatik G bestimmt. In Zeile 5–8 erfolgt die zeilenweise Berechnung der weiteren Matrizenelemente von X auf Basis der existierenden Elemente. In Zeile 9–12 erfolgt die Entscheidung, ob w in der Sprache der Grammatik liegt, auf Basis des letzten Elements der Tabelle X .

Aufwandsabschätzung

Die Tabelle des CYK-Algorithmus enthält genau $n(n+1)/2 = O(n^2)$ Tabelleneinträge. Die Berechnung jedes Tabelleneintrages benötigt durch die Bestimmung der einzelnen Terme $O(n)$. Damit beträgt der Rechenaufwand für den CYK-Algorithmus $O(n^3)$.

Für die Implementierung in Java benötigen wir nun einige Hilfsklassen für die Datenehaltung:

- Definition einer Klasse AR1 für die Ableitungsregel der Form $A \rightarrow a$, $A \in N$, $a \in \Sigma$:

```

class AR1 // Ableitungsregel der Form A -> a
{
    private int A; // Nichtterminalsymbol
    private int a; // Terminalsymbol

    public AR1(int A, int a)
    {
        this.A = A;
        this.a = a;
    }
    public int getLinks()
    {
        return A;
    }
    public int getRechts()
    {
        return a;
    }
}

```

- Definition einer Klasse AR2 für die Ableitungsregel der Form $A \rightarrow BC$, $A, B, C \in N$:

```
class AR2 // Ableitungsregel der Form A -> BC
{
    private int A; // Nichtterminalsymbol
    private int B; // Terminalsymbol
    private int C; // Terminalsymbol

    public AR2(int A, int B, int C)
    {
        this.A = A;
        this.B = B;
        this.C = C;
    }
    public int getLinks()
    {
        return A;
    }
    public int[] getRechts()
    {
        int v[] = {B, C};
        return v;
    }
}
```

- Definition einer Klasse NT als Datenstruktur für die Tabelle X :

```
class NT // Menge von Nichtterminalsymbolen
{
    private Vector<Integer> v;

    public NT(Vector<Integer> v)
    {
        this.v = v;
    }
    public Vector<Integer> getElement()
    {
        return v;
    }
}
```

Mit diesen Hilfsklassen können wir zunächst die Eingabe des CYK-Algorithmus definieren. Wir verwenden dazu das obige Beispiel.

```
public abstract class Test_CYK
{
    public static void main(String[] args)
    {
        // ----- Eingabe -----
        // --- 1. Anzahl der Nichtterminalsymbole
        int m = 4; // {S, A, B, C}
```

```

// --- 2. Anzahl der Terminalsymbole
int n = 2; // {a, b}

// --- 3. Ableitungsregeln
AR1 r1[] = new AR1[3]; // Form A -> a
r1[0] = new AR1(1, 0); // A -> a
r1[1] = new AR1(2, 1); // B -> b
r1[2] = new AR1(3, 0); // C -> a

AR2 r2[] = new AR2[5]; // Form A -> BC
r2[0] = new AR2(0, 1, 2); // S -> AB
r2[1] = new AR2(0, 2, 3); // S -> BC
r2[2] = new AR2(1, 2, 1); // A -> BA
r2[3] = new AR2(2, 3, 3); // B -> CC
r2[4] = new AR2(3, 1, 2); // C -> AB

// --- 4. Eingabewort
int w[] = {1, 0, 0, 1, 0};

// -----
CYK algo = new CYK(m, n, r1, r2);
System.out.printf(" Wort w in der Sprache?: %b" , algo.cyk_algo(w));
}
}

```

Jetzt können wir die Klasse CYK für die Implementierung des CYK-Algorithmus erstellen. Die Instanzvariablen sind m für die Anzahl der Nichtterminalsymbole, n für die Anzahl der Terminalsymbole, das Array $r1$ für die Ableitungsregel vom Typ AR1 und das Array $r2$ für die Ableitungsregel vom Typ AR2. Neben dem Konstruktor zur Initialisierung benötigen wir noch die folgenden Methoden:

- Bestimmung der linken Seite aus der Regel $A \rightarrow a$ für Buchstaben w :

```

public Vector<Integer> getTerminal(AR1 r[], int w)
{
    Vector<Integer> v = new Vector<Integer>();
    for(int i=0; i<r.length; i++)
        if(r[i].getRechts() == w)
            v.addElement(r[i].getLinks());
    return v;
}

```

- Produkt zweier Mengen von Nichtterminalsymbolen:

```

public Vector<AR2> vereinige(Vector<Integer> v1, Vector<Integer> v2)
{
    Vector<AR2> v = new Vector<AR2>();
    for(int i=0; i<v1.size(); i++)
        for(int j=0; j<v2.size(); j++)
            v.addElement(new AR2(0, v1.elementAt(i), v2.elementAt(j)));
    return v;
}

```


- Bestimmung der linken Seite aus Regel $A \rightarrow BC$ für eine Menge von Nichtterminalsymbolen:

```
public Vector<Integer> bestimmeAbleitung(Vector<AR2> menge)
{
    boolean b[] = new boolean[m];
    for(int i=0; i<menge.size(); i++) // alle Ableitungen
    {
        int m[] = menge.elementAt(i).getRechts();
        for(int j=0; j<r2.length; j++)
        {
            int r[] = r2[j].getRechts();
            if(m[0] == r[0] && m[1] == r[1])
                b[r2[j].getLinks()] = true;
        }
    }

    Vector<Integer> v = new Vector<Integer>();
    for(int i=0; i<b.length; i++)
        if(b[i])
            v.addElement(i);

    return v;
}
```

Mit diesen Hilfsmethoden ist die Implementierung des CYK-Verfahrens schnell möglich:

```
public boolean cyk_algo(int w[])
{
    int anz = w.length;
    // --- 0. Tabelle für dynamische Programmierung
    NT tabelle[][] = new NT[anz][anz];

    // --- 1. Berechnung der ersten Zeile
    for(int i=0; i<anz; i++)
        tabelle[0][i] = new NT(getTerminal(r1, w[i]));

    // --- 2. Berechnung 2. bis n-te Zeile
    for(int i=1; i<anz; i++) // alle Zeilen
    {
        for(int j=0; j<anz-i; j++) // alle Spalten
        {
            Vector<AR2> v_rechts = new Vector<AR2>();
            int id1 = 1+j; // Index der Tabelle
            int id2 = i+j+1;

            for(int k=0; k<(id2-id1); k++) // alle Vereinigungsmengen
                v_rechts.addAll(vereinige(tabelle[0+k][j].getElement(),
                                           tabelle[i-k-1][j+k+1].getElement()));
            Vector<Integer> v_links = bestimmeAbleitung(v_rechts);
            tabelle[i][j] = new NT(v_links);
        }
    }
}
```

```
// --- 3. Prüfung nach Element S=0
if(tabelle[anz-1][0].getElement().indexOf(0)!=-1)
    return true;
else
    return false;
}
```

6.4.6 Anwendungen

Wir betrachten einige Anwendungen von Grammatiken.

Beschreibung von Programmiersprachen Grammatiken werden zur Beschreibung von Programmiersprachen, Markup-Sprachen wie HTML oder XML verwendet. Anschließend erfolgen mit einem Parser (dt. Zerleger) das Zerlegen der Eingabe und die Umwandlung in ein für die Weiterverarbeitung geeigneteres Format wie beispielsweise in eine Baumstruktur.

Eine einfache Grammatik $G = (\{S\}, \{(\,,)\}, R, S)$ für die Verschachtelung von runden Klammern, wobei die Anzahl der linken und rechten Klammern gleich ist, besitzt die folgenden Produktionen R :

$$S \rightarrow SS \mid (S) \mid \varepsilon.$$

Wir geben eine weitere kontextfreie Grammatik für einige ausgewählte syntaktische Befehle der Programmiersprache Java an. Dazu stellen wir in der Schreibweise $\langle \dots \rangle$ die Nichtterminalsymbole und die Schlüsselwörter `for`, `if` usw. als die Terminalsymbole der Grammatik dar. Die Grammatik ist hierbei gegeben durch die folgenden Ableitungsregeln:

$$\begin{aligned} \langle \text{Anw} \rangle &\rightarrow \langle \text{While-Anw} \rangle \mid \langle \text{If-Anw} \rangle \mid \langle \text{For-Anw} \rangle \\ \langle \text{While-Anw} \rangle &\rightarrow \text{while}(\langle \text{Bedingung} \rangle) \langle \text{Anw} \rangle \\ \langle \text{If-Anw} \rangle &\rightarrow \text{if}(\langle \text{Bedingung} \rangle) \langle \text{Anw} \rangle \\ \langle \text{For-Anw} \rangle &\rightarrow \text{for}(\langle \text{Anw} \rangle; \langle \text{Bedingung} \rangle; \langle \text{Anw} \rangle) \langle \text{Anw} \rangle \\ \langle \text{Bedingung} \rangle &\rightarrow \langle \text{Ausdruck} \rangle \langle \text{Relation} \rangle \langle \text{Ausdruck} \rangle \\ \langle \text{Relation} \rangle &\rightarrow == \mid < \mid <= \mid => \mid > \end{aligned}$$

Beispielsweise sieht dann die Ableitung der folgenden `for`-Schleife

```
for(int i=0; i<10; i++)
    a = a + 10;
```

wie folgt aus:

```

<Anw> ⇒ <For-Anw>
⇒ for(<Anw>; <Bedingung>; <Anw>) <Anw>
⇒ for(<Anw>; <Ausdruck> <Relation> <Ausdruck>; <Anw>) <Anw>
⇒ for(int i=0; i<10; i++) a = a + 10;

```

In Abb. 6.23 ist der zugehörige Ableitungsbaum der for-Schleife dargestellt.

Grammatik für die natürliche Sprache In vielen Sprachanalysetools werden Grammatiken für die natürliche Sprache benötigt. Beispielsweise sieht eine Grammatik dieser Art wie folgt aus:

<Satz>	→	<Subjekt> <Prädikat> <Objekt>
<Subjekt>	→	<Artikel> <Attribut> <Substantiv>
<Artikel>	→	ε der die das
<Attribut>	→	ε <Adjektiv> <Adjektiv> <Attribut>
<Adjektiv>	→	groß klein wenig viel
<Substantiv>	→	Menschen Tiere Mathematik
<Prädikat>	→	sagen reden lieben
<Objekt>	→	<Artikel> <Attribut> <Substantiv>

Die Nichtterminalsymbole sind dabei wieder durch die Schreibweise <> gekennzeichnet. Eine Ableitung sieht dann wie folgt aus:

```

<Satz> ⇒ <Subjekt> <Prädikat> <Objekt>
⇒ <Artikel> <Attribut> <Substantiv> <Prädikat> <Objekt>
⇒ <Artikel> <Attribut> <Substantiv> <Prädikat> <Artikel> <Attribut>
   <Substantiv>
⇒ Die <Substantiv> <Prädikat> <Artikel> <Attribut> <Substantiv>
⇒ Die Menschen <Prädikat> <Artikel> <Attribut> <Substantiv>
⇒ Die Menschen lieben <Artikel> <Attribut> <Substantiv>
⇒ Die Menschen lieben die <Substantiv>
⇒ Die Menschen lieben die Mathematik

```

In Abb. 6.24 ist der zugehörige Ableitungsbaum dieses Satzes dargestellt.

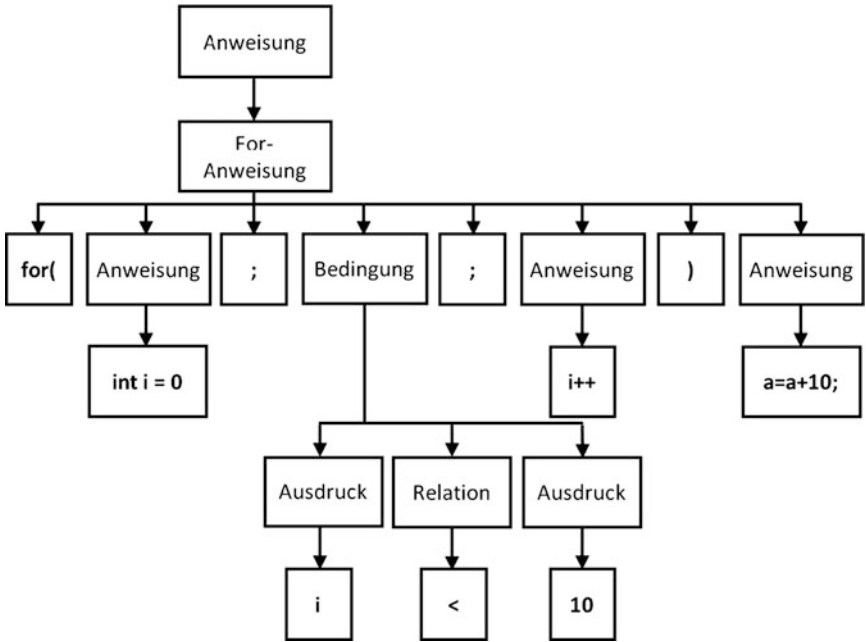


Abb. 6.23 Ableitungsbaum einer for-Schleife

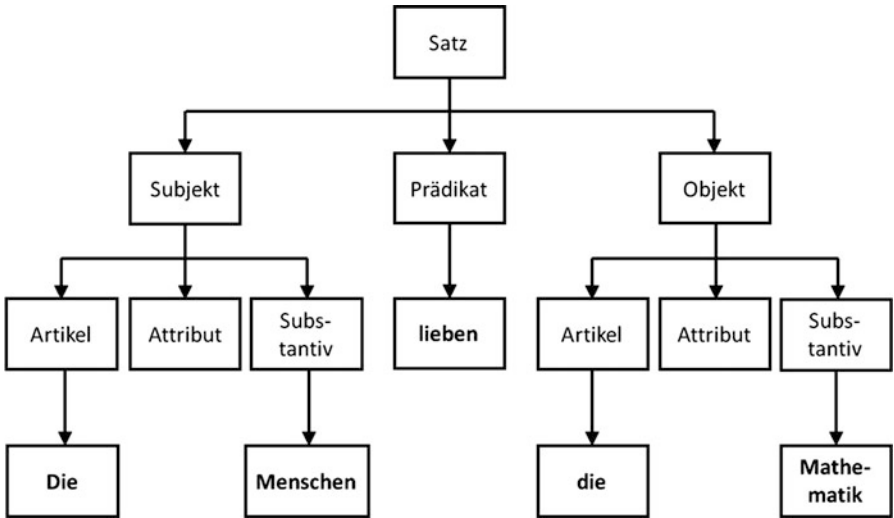


Abb. 6.24 Ableitungsbaum eines Satzes in deutscher Sprache

6.5 Übungsaufgaben

Aufgabe 6.1 (Deterministischer Automat) Ein Bauer muss mit einem Kohlkopf, einem Wolf und einer Ziege einen Fluss überqueren. Sein Boot ist sehr klein, sodass der Bauer entweder Wolf oder Ziege oder Kohlkopf in einer Fahrt mitnehmen kann. Aufgrund dieser Tatsache ergeben sich für ihn zwei Probleme: Wenn die Ziege und der Kohlkopf allein zurückbleiben, so frisst die Ziege den Kohl. Wenn der Wolf und die Ziege allein zurückbleiben, so frisst der Wolf die Ziege. Der Bauer stellt sich die folgende Frage: Ist es möglich den Fluss zu überqueren, ohne dass die Ziege oder der Kohlkopf gefressen werden? Falls es eine Lösung gibt, will der Bauer natürlich die kleinstmögliche Anzahl von Flussüberfahrten bestimmen. Modellieren Sie dieses Problem mit einem endlichen Automaten.

Aufgabe 6.2 (Deterministischer Automat) Ein Tor einer Fabrikhalle hat die Aufgabe, sich selbstständig zu öffnen und zu schließen, wenn ein Fahrzeug ein- und ausfährt. Beschreiben Sie die Bewegung des Tores durch einen deterministischen Automaten.

Aufgabe 6.3 (Deterministischer Automat) Beschreiben Sie das Verhalten eines automatischen Zählers bis zu einer festen Zahl n durch einen deterministischen Automaten.

Aufgabe 6.4 (Deterministischer Automat) Beschreiben Sie die Fahrt eines autonomen Fahrzeugs durch Aktionen wie Beschleunigen, Bremsen, Lenken usw. in Abhängigkeit von der Verkehrssituation (z. B. Straße geradeaus, Rechtskurve, Linkskurve usw.) mithilfe eines Automaten.

Aufgabe 6.5 (E/A-Automat) Ein Automat hat verschiedene Sorten von kleinen Speisen. Der Kunde wirft einen gewissen Geldbetrag ein, dann blinken die Wahlkosten für die vorhandenen Speisen. Nach der Wahl wird das Essen ausgegeben. Durch eine Taste kann der Kunde den Vorgang abbrechen, bevor die Speise ausgegeben wurde. Beschreiben Sie das Verhalten des Automaten durch einen deterministischen E/A-Automaten.

Aufgabe 6.6 (E/A-Automat) Wir betrachten eine mechanische Parkuhr, bei der ein Autofahrer maximal EUR 2,00 einwerfen kann. Danach beginnt die Parkzeit pro 10 min 20 Cent abzulaufen. Bevor die Parkzeit durch das Einwerfen weiterer Münzen verlängert werden kann, muss die Uhr komplett abgelaufen sein. Geben Sie einen E/A-Automaten für die Parkuhr an.

Aufgabe 6.7 (Nicht deterministischer Automat) Implementieren Sie den Algorithmus für das Wortproblem für einen nicht deterministischen Automaten.

Aufgabe 6.8 (Nicht deterministischer Automat) Implementieren Sie den Algorithmus zur Umwandlung eines nicht deterministischen Automaten in einen deterministischen Automaten über die Potenzmengenkonstruktion.

Aufgabe 6.9 (Regulärer Ausdruck) Implementieren Sie den Algorithmus zur Umwandlung eines regulären Ausdrucks in einen nicht deterministischen Automaten.

Aufgabe 6.10 (Grammatik) Implementieren Sie die beiden Vorschriften zur Umwandlung einer regulären Grammatik in einen Automaten, und umgekehrt.

Aufgabe 6.11 (Grammatik) Schreiben Sie einen kleinen Compiler zur Überprüfung der Syntax einer ausgewählten Programmiersprache.

Literaturhinweise³

1. Schöning, U. (2008). *Theoretische Informatik – kurzgefasst*. Spektrum.
2. Schöning, U. (2008). *Ideen der Informatik*. Oldenbourg.
3. Hopcroft, J.E., Motwani, R., Ullman, J.D. (2002). *Einführung in die Automatentheorie. Formale Sprachen und Komplexitätstheorie*. Pearson.
4. Lunze, J. (2012). *Ereignisdiskrete Systeme*. Oldenbourg.
5. Lunze, J. (2012). *Automatisierungstechnik*. Oldenbourg.
6. Schenk, J., Rigoll, G. (2010). *Mensch-Maschine-Kommunikation*. Springer.

³ Automaten und Grammatiken sind ein klassisches Thema aus der theoretischen Informatik. Einen guten Einstieg in das Thema bieten die beiden Lehrbücher [1] und [2]. Weiterführende Themen zur Automatentheorie und formaler Sprache findet der Leser in [3]. Automaten besitzen sehr große Anwendung in den Ingenieurwissenschaften wie beispielsweise in der Automatisierungstechnik oder bei der Modellierung ereignisdiskreter Systeme. Eine gute Darstellung der Automaten aus Ingenieurssicht liefern die beiden Lehrbücher [4] und [5]. Anwendungsmöglichkeiten im Bereich der Mensch-Maschine-Kommunikation in Form von Dialogsystemen findet der Leser in [6].

Unter Bildverarbeitung versteht man im Allgemeinen die Aufbereitung, Analyse, Klassifizierung und Interpretation von visuellen Informationen in Form von Bilddaten. Die zwei wesentlichen Aufgaben der Bildverarbeitung sind die Bildverbesserung und die Bilderkennung. Die Bildverbesserung wird zur Reduktion von unvermeidlichen Fehlern bei der Aufnahme von Bildern eingesetzt, beispielsweise durch verschiedene Arten von Filteroperationen oder spezifische Transformationen. Anschließend ist die Extraktion von Informationen in Form von charakteristischen Merkmalen aus dem Bild effizient möglich. Das Ziel dabei ist, gewisse Objekte im Bild zu erkennen und diese dann zu beschreiben. Das Resultat der Bildverarbeitung ist entweder wieder ein Bild oder eine Menge von Merkmalen des Eingabebildes, die im Rahmen einer Bilderkennung bestimmt wurden. Im Gegensatz dazu hat die Computergrafik die Aufgabe, aus einer abstrakten Beschreibung von Objekten ein Bild zu generieren.

Die Bildverarbeitung wird in der Technik und Naturwissenschaft in vielen unterschiedlichen Bereichen eingesetzt. Im Maschinenbau sind bildverarbeitende Algorithmen von zentraler Bedeutung, um Objekte zu zählen bzw. zu vermessen oder um mobile Roboter zu steuern. Ein typisches Beispiel ist ein Roboter, der durch einen Bildsensor ein Bauteil in seinem Sichtbereich erfassen und eine Aktion wie etwa das Greifen mit einem Aktuator durchführt. Ein weiteres Anwendungsfeld bildverarbeitender Algorithmen ist die Qualitätssicherung in industriellen Produktionsprozessen. In der optischen Qualitätskontrolle können damit Klebeverbindungen von Produktionsrobotern analysiert werden, ob sie die richtige Form besitzen. In der Automobilindustrie sind Bilddaten und die zugehörigen Algorithmen von großer Relevanz im Bereich des maschinellen Sehens für die Entwicklung des autonomen Fahrens.

In der Medizin besitzt die Bildverarbeitung einen großen Stellenwert zur Unterstützung der medizinischen Diagnose und Therapie. Die medizinische Bildverarbeitung ist eines der wichtigsten Teilgebiete der medizinischen Informatik mit einer großen Zahl von Anwendungen. Beispiele dafür findet man im Bereich der Diagnostik, zur Darstellung von

Organen und Organsystemen (z. B. Skelett), zur Erkennung von krankhaften Veränderungen oder in der Chirurgie zur OP-Planung und Kontrolle von Operationen.

Verwandte Gebiete der Bildverarbeitung sind die Mustererkennung und die künstliche Intelligenz. Die Mustererkennung wird neben der digitalen Bildverarbeitung auch in vielen anderen praktischen Problemstellungen verwendet. Zwei Beispiele sind die Verarbeitung verschiedenster Arten von Signalen und die Analyse der natürlichen Sprache bei der Mensch-Maschine-Kommunikation.

In diesem Kapitel stellen wir die grundlegenden Prinzipien, Methodiken und Algorithmen der Bildverarbeitung vor. Zu nennen sind hierbei die Vorverarbeitung, die Registrierung und die Segmentierung von Bilddaten. Die Verfahren zur Analyse und Klassifikation von Bilddaten werden im 3. Band dieser Buchreihe eingeführt. Der Grund dafür ist, dass diese Verfahren neben der Bildverarbeitung noch in vielen anderen Bereichen wie des Data Mining und der künstlichen Intelligenz verwendet werden.

7.1 Allgemeine Grundlagen

Die Hauptaufgabe von Bildverarbeitungssystemen besteht in der computergestützten Analyse, Identifizierung und Visualisierung von Bildobjekten. In der Technik werden vor allem kameragesteuerte Geräte für unterschiedlichste Anwendungen eingesetzt, beispielsweise für autonome mobile Roboter. Ein solches kameragesteuertes Gerät besteht aus einem Sensor zur Erfassung der Umgebung, einem Bildverarbeitungsalgorithmus zur Objekterkennung, einem Steuerungsalgorithmus zur Reaktion des Gerätes und einer Aktorik zur Einstellung der Geschwindigkeit und der Bewegungsrichtung.

In der Medizintechnik besitzen Bilddaten in der Diagnose und Therapie eine herausragende Bedeutung, um in das Innere des menschlichen Körpers zu schauen. Mit der Entdeckung der Röntgenstrahlung durch W. C. Röntgen im Jahre 1895 wurde die Grundlage der Bildgebung gelegt (2-D-Bildverarbeitungsverfahren). In den 1970er-Jahren wurde durch die tomografische Bildgebungstechnik wie der Computer- oder Magnetresonanztomografie ein weiterer bedeutender Fortschritt gemacht (3-D-Bildverarbeitungsverfahren). Mit der Weiterentwicklung dieser Systeme ist es möglich geworden, dynamische Prozesse und Organbewegungen (z. B. Herzbewegung, Lungenbewegung) zu erfassen (4-D-Bildverarbeitungsverfahren). In der medizinischen Bildverarbeitung kommen Analyseverfahren zur Diagnose von Krankheiten auf Basis medizinischer Bilder zur Anwendung.

Grundlegende Phasen In der digitalen Bildverarbeitung werden Methoden und Algorithmen aus den unterschiedlichsten Bereichen der Mathematik und Informatik eingesetzt. Die einzelnen Phasen lassen sich dabei wie in Abb. 7.1 aufteilen:

1. Bilderzeugung

Die Grundlage der Bildverarbeitung besteht aus Rohdaten, die durch geeignete Sensoren unter gewissen Kriterien (z. B. Art der Messgröße, Datenmenge, Spektrum) auf-

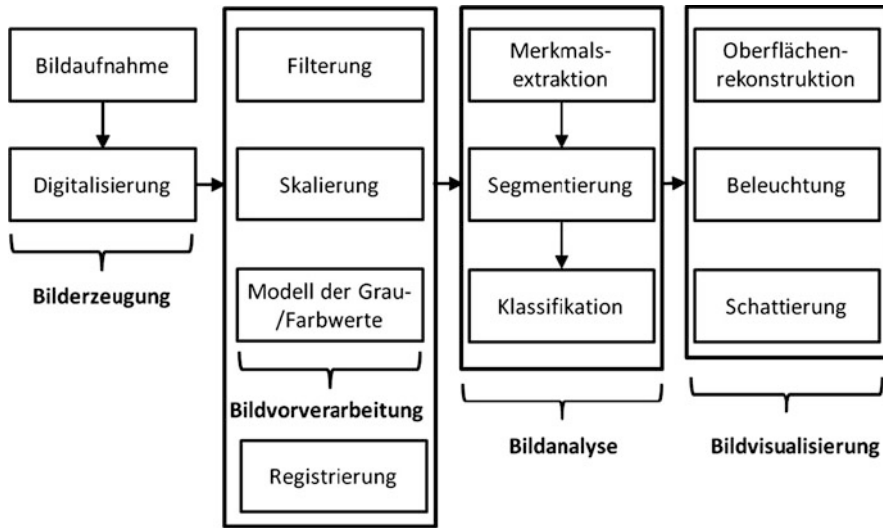


Abb. 7.1 Allgemeine Phasen der Bildverarbeitung

genommen worden sind. Aus den Rohdaten wird das Rohbild digital rekonstruiert, so wie beispielsweise beim CT.¹

2. Bildvorverarbeitung

Die Vorverarbeitung wird im Rahmen der Bildverbesserung eingesetzt, um aus den Rohdaten des Sensors oder dem digitalisierten Datenmaterial Fehler in Form von Störungen durch die Elektronik, Unschärfe durch Bewegung, Verzerrungen durch das Linsensystem oder Abschattungen durch die Beleuchtung zu entfernen. Für die Korrekturen kommen diverse Glättungsfiler zur Verminderung des Bildrauschens, Kantenfilter zum Hervorheben von Kanten, Methoden zur Verbesserung des Bildkontrastes, Skalierung des Grauwertbereichs oder für die Anpassung der Bildgröße zur Anwendung.

3. Bildregistrierung

Bei der Registrierung werden verschiedene Bilder in einem gemeinsamen Koordinatensystem dargestellt, um damit einen direkten Vergleich von Bildstrukturen in den unterschiedlichen Bilddaten zu bekommen. Bei der medizinischen Bildverarbeitung ist die Registrierung von großer Bedeutung zur gemeinsamen Darstellung von Bilddaten eines oder mehrerer Patienten. Auch für die Kombination von verschiedenen Bilddaten mit unterschiedlichem Informationsgrad werden Registrierungsverfahren angewandt. Diese Verfahren besitzen auch im Bereich der Robotik zur Lokalisierung von mobilen Robotern auf Karten durch aufgenommene Scans eine wichtige Rolle.

¹ Siehe Buch *Grundlagen*, Kapitel Technische und naturwissenschaftliche Anwendungen.

4. Bildanalyse

Die Bildanalyse besteht aus der Merkmalsextraktion, der Segmentierung und der Klassifikation. In der Merkmalsextraktion werden die für die konkrete Anwendung zentralen strukturellen Eigenschaften aus der Bildinformation hervorgehoben. Hierzu werden gewisse Merkmale für die Segmentierung des Bildes definiert und mit verschiedensten Verfahrenstypen die zugehörigen Bildpixel bestimmt. Bei der Segmentierung wird das Bild nach gewissen interessierenden Bildobjekten in Bereiche (Segmente) aufgeteilt, die gleiche oder ähnliche Eigenschaften besitzen, und vom Hintergrund getrennt. Die Beschreibung der Segmente erfolgt durch den Flächeninhalt, den Umfang, die Form oder den Schwerpunkt. Die Segmentierung bildet die Grundlage für die Extraktion von charakteristischen Merkmalen von Bildobjekten. Die Erkennung von Bildobjekten wird durch Klassifikationsverfahren (z. B. neuronaler Netze) durchgeführt, um so alle Pixel des Eingabebildes einer vorgegebenen Klasse zuzuordnen.

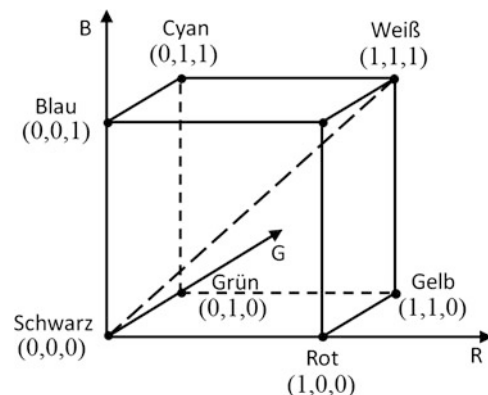
5. Bildvisualisierung

Die Aufgabe der Bildvisualisierung ist die Rekonstruktion der Oberfläche, die Beleuchtung und Schattierung des Bildes. Bei der Visualisierung werden Grauwert- und Farbbilder oder die Resultate der Segmentierung für Bildobjekte dargestellt. Die Visualisierung ist eine Grundlage für die virtuelle Realität, die für die Navigation im virtuellen Körper oder zur Steuerung von virtuellen Werkzeugen verwendet wird.

Bei der Visualisierung von Bildern unterscheidet man zwischen Farbbildern und Graubildern. Farbbilder sind Bilddaten, bei denen jedem Bildpunkt drei Kenngrößen (z. B. RGB) zugeordnet sind. Bei Graubildern gibt es hingegen nur eine Kenngröße. Graubilder werden vor allem in der Radiologie generiert, da man die Grauwerte ordnen kann, wodurch eine Interpretation der gemessenen Signalintensitäten möglich wird.

Farbbilder Für die Farbdefinition gibt es unterschiedliche Farbmodelle, die durch dreidimensionalen Farbräume definiert werden. Man unterscheidet hierbei technisch-physikalische Farbmodelle (z. B. RGB-Modell in Abb. 7.2), die bei der Farbdarstellung in tech-

Abb. 7.2 RGB-Farbmodell mit den drei Grundfarben Rot, Grün und Blau



nischen Systemen eingesetzt werden, und wahrnehmungsorientierte Farbmodelle (z. B. HSV-Farbraum, HSL-Farbraum), in denen Farben anhand physiologischer Kenngrößen wie dem Farbton, der Helligkeit oder der Farbsättigung definiert werden.

Farben sind optische Wahrnehmungen von elektromagnetischer Strahlung. Die Verarbeitung von Farbwerten im RGB-System kann in Java mithilfe der Klasse `Color` aus dem Paket `java.awt.Color` erledigt werden. In diesem System wird eine Farbe durch drei ganze Zahlen zwischen 0 und 255 definiert, die die Intensität zwischen Rot- (r), Grün- (g) und Blauanteilen (b) ausdrücken:

```
Color farbe = new Color(r,g,b);
```

Beispielsweise ist die Farbe Weiß = (255, 255, 255), Schwarz = (0, 0, 0), Rot = (255, 0, 0), Grün = (0, 255, 0) oder Blau = (0, 0, 255).

Graubilder Die Qualität, mit der Bilder auf Anzeigegeräten dargestellt werden, hängt ab von einer Farbeigenschaft, die als *Schwarz-Weiß-Luminanz* bezeichnet wird. Die Umwandlung von Farbbildern in Graubilder ist eine der Standardaufgaben der Bildverarbeitung. Die Luminanz einer Farbe im RGB-Format (r, g, b) wird durch die folgende Gleichung definiert:

$$\text{lum} = 0,299 \cdot r + 0,587 \cdot g + 0,114 \cdot b.$$

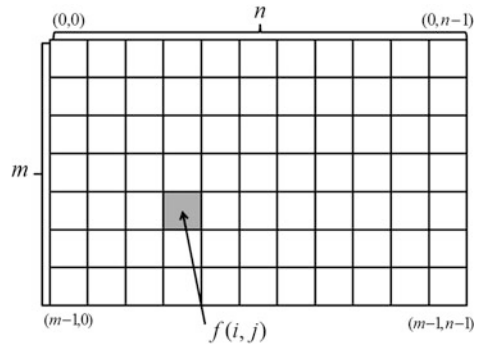
Das Ergebnis ist eine Zahl zwischen 0 und 255. Das RGB-System hat die Eigenschaft, dass bei drei gleichwertigen Farbintensitäten die resultierende Farbe auf der Grauskala liegt, die von Schwarz (0) bis Weiß (255) reicht.

Mit dem Luminanzwert kann beispielsweise entschieden werden, ob zwei Farben in dem Sinne kompatibel sind, d. h., ob ein Text mit der einen Farbe auf einem Hintergrund der anderen Farbe gedruckt werden kann. Als Faustregel gilt, dass die Differenz zwischen den Luminanzwerten der Vorder- und Hintergrundfarbe mindestens 128 betragen sollte. Diese Eigenschaft spielt bei der Entwicklung von Werbematerial, Verkehrszeichen und Webseiten eine sehr wichtige Rolle.

Bildmatrix Die Bildverarbeitung beinhaltet die Analyse und Veränderung der Bildmatrix mit dem Ziel der Verbesserung der Bildeigenschaften. Es sei f die Bildmatrix des Eingabebildes, die aus m Zeilen und n Spalten besteht (siehe Abb. 7.3). Das entsprechende Bildelement $f(i, j)$ repräsentiert den diskreten Grauwert zwischen 1 und 2^k in der Zeile i und Spalte j . Der Ursprung des Koordinatensystems ist dabei links oben. Der mittlere Grauwert f_M ist definiert durch das arithmetische Mittel:

$$f_M = \frac{1}{m \cdot n} \sum_{i=1}^m \sum_{j=1}^n f(i, j).$$

Abb. 7.3 Bildkoordinaten aus
 m Zeilen und n Spalten mit
 Bildmatrix f



Darstellung von Bilddaten Für die Darstellung von Bilddaten implementieren wir in Java eine Klasse `Bild` als Oberklasse. Für die nachfolgenden Anwendungen werden wir von dieser Klasse verschiedene Subklassen ableiten. Diese Klasse `Bild` enthält die Instanzvariablen `BufferedImage bild` für das Bildobjekt, `JFrame frame` für das Java-Frame-Objekt, `int breite`, `hoehe` für die Breite und Höhe des Bildes und `String name` für den Bildnamen. Die Konstruktoren liefern unterschiedliche Möglichkeiten ein Bild zu initialisieren. Die Klasse `Bild` beinhaltet die folgenden Methoden:

- `getHoehe` bestimmt die Höhe des Bildes;
- `getBreite` bestimmt die Breite des Bildes;
- `anzeigen` stellt das Bild dar;
- `get` liefert den Pixelwert an der angegebenen Zeile und Spalte;
- `set` setzt einen Pixelwert an der angegebenen Zeile und Spalte;
- `speichern` zur Abspeicherung eines Bildes.

Die Implementierung dieser Methoden sieht in Java wie folgt aus:

```
public class Bild
{
    private BufferedImage bild;
    private JFrame frame;
    private int breite, hoehe;
    private String name;

    public Bild(int breite, int hoehe)
    {
        bild = new BufferedImage(breite, hoehe, BufferedImage.TYPE_INT_RGB);
    }

    public Bild(String name)
    {
        try
        {
            File file = new File(name);
            bild = ImageIO.read(file);
            this.name = name;
            breite = bild.getWidth(null);
            hoehe = bild.getHeight(null);
        }
    }
}
```

```

        catch(IOException e)
        {
            throw new RuntimeException("Datei " + name + " kann nicht geoeffnet werden.");
        }
    }
    public int getHoehe()
    {
        return hoehe;
    }
    public int getBreite()
    {
        return breite;
    }

    public void anzeigen()
    {
        if (frame == null)
        {
            frame = new JFrame();
            frame.setContentPane(new JLabel(new ImageIcon(bild)));
            frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
            frame.setTitle(name);
            frame.pack();
            frame.setVisible(true);
        }
        frame.repaint();
    }

    public Color get(int spalte, int zeile)
    {
        return new Color(bild.getRGB(spalte, zeile));
    }
    public void set(int spalte, int zeile, Color color)
    {
        bild.setRGB(spalte, zeile, color.getRGB());
    }

    public void speichern(String name)
    {
        String endung = name.substring(name.lastIndexOf('.') + 1);
        try
        {
            ImageIO.write(bild, endung, new File(name));
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}

```

7.2 Bildfilterung

7.2.1 Problemstellung

Mithilfe der Bildvorverarbeitung kann die Qualität der Bilddaten für die weiteren Verarbeitungsschritte deutlich verbessert werden. Die Ziele der Vorverarbeitung sind die Verminderung des Bildrauschens, das Hervorheben von Kanten, die Verbesserung des Bild-

kontrastes, die Skalierung des Grauwertbereichs oder die Anpassung der Bildgröße. Die Operationen in der Bildverarbeitung gliedern sich dabei nach der Anzahl der Bildpunkte, die eine Operation beeinflussen, in die folgenden drei Klassen:

1. **Punktoperatoren:** Operator verändert den Farbwert eines Bildpunktes ohne die nähere Umgebung zu beachten.
2. **Lokale Operatoren:** Operator verändert den Farbwert eines Bildpunktes auf Basis einer begrenzten Umgebung um den betrachteten Bildpunkt.
3. **Globale Operatoren:** Operator verändert den Farbwert eines Bildpunktes auf Basis des gesamten Bildes.

Zusätzlich zu den genannten drei Operatoren kann man noch geometrische Operatoren in Form von Drehung, Skalierung oder Spiegelung betrachten.² Zu den Punktoperatoren gehört beispielsweise das im Abschnitt Segmentierung vorgestellte Schwellwertverfahren. Ein globaler Operator ist die Fourier-Transformation, eine Operation im Frequenzbereich zur Bildvorverarbeitung.

Die in diesem Abschnitt vorgestellten Filter für die Bildglättung (Tiefpassfilter) und Kantenverstärkung (Hochpassfilter) gehören zu den lokalen Operatoren im Ortsbereich. Diese Filter sind dadurch bestimmt, dass bei ihnen für jeden Bildpunkt eine auf seine lokale Nachbarschaft beschränkte Transformation mittels sogenannter Masken durchgeführt wird. Diese Operationen im Ortsbereich haben in der Bildverarbeitung zahlreiche Anwendungen. Die Tiefpassfilter können zur Elimination von gestörten Bildpunkten oder Bildzeilen verwendet werden. Die Hochpassfilter sind sehr gut als Verarbeitungsschritt zur Kantenextraktion geeignet.

FILTERUNG

Gegeben: Bildmatrix $f = f(x, y)$, Filtermaske $g^{(2k+1) \times (2k+1)}(x, y)$ mit $k \in \{1, 2, 3, \dots\}$.
 Gesucht: gefiltertes Bild $\tilde{f} = f * g$.

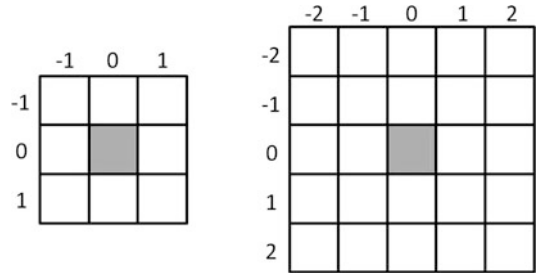
Die Faltungsoperation $*$ wird im nachfolgenden Abschnitt genauer definiert.

7.2.2 Grundlegende Lösungsprinzipien

Gegeben ist eine Bildmatrix $f = f(x, y)$ mit m Zeilen und n Spalten, die mit einer Filtermaske verändert werden soll. Die Masken sind durch die Größe ihrer Bildumgebung $g^{(2k+1) \times (2k+1)}(x, y)$ mit $k \in \{1, 2, 3, \dots\}$ mit dem Bildpunkt (x, y) als Zentrum festgelegt (siehe Abb. 7.4). Die Bildmatrix f wird mit der Maske verknüpft, indem die Maske über die Bildmatrix hinweggeschoben wird. Dazu werden alle innerhalb dieser Rahmen vorkommenden Bildelemente mit den korrespondierenden Maskenelementen multipliziert und die Produkte aufsummiert.

² Siehe Band *Grundlagen*, Kapitel Technische und naturwissenschaftliche Anwendungen.

Abb. 7.4 Masken der Größe 3×3 und 5×5 zur Filterung von Bilddaten



- Die Filterarten lassen sich in zwei Kategorien einordnen:
- 1. Lineare Filter:** lineare Verknüpfungen der Pixel mit der Filtermaske durch Faltung des Originalbildes mit einer $(2k + 1) \times (2k + 1)$ -Maske g (Faltungskern):
$$\tilde{f} = f * g \quad \text{mit} \quad \tilde{f}(x, y) = \sum_{i=-k}^k \sum_{j=-k}^k f(x + i, y + j) \cdot g(i, j).$$
Am Bildrand ist die Bildfunktion f nicht für alle Bildpunkte der Umgebung definiert, sodass entweder das Bild um die Größe des Faltungsoperators verkleinert wird, eine Extrapolation der Randpunkte stattfindet oder die Maske eingeschränkt wird. Beispiele für lineare Filter sind der Mittelwert- oder Gauß-Filter.
 - 2. Nicht lineare Filter:** nicht lineare Verknüpfungen der einzelnen Pixel durch gewisse Transformationsvorschriften. Beispiele dieser Art von nicht linearen Filtern sind der Minimum- und Maximumfilter bei denen entweder helle (Minimum) oder dunkle (Maximum) Punkte entfernt werden.

Glättungs- bzw. Tiefpassfilter

Glättungsfilter können zur Reduzierung des Rauschens und zur Bildglättung eingesetzt werden. Mithilfe dieser Filterart werden lokale Variationen der Bildfunktionswerte reduziert und damit eine gewisse Homogenität des Bildes erzeugt.

Mittelwertfilter Bei dem linearen Mittelwertfilter wird jeder Bildpunkt aus dem gerundeten Mittelwert der durch die Maske überdeckten Nachbarpunkte berechnet (siehe Abb. 7.5). Das Ergebnis ist ein geglättetes Bild mit deutlich geringerem Rauschen und ab-

Abb. 7.5 Maske der Mittelwertfilter 3×3 und 5×5

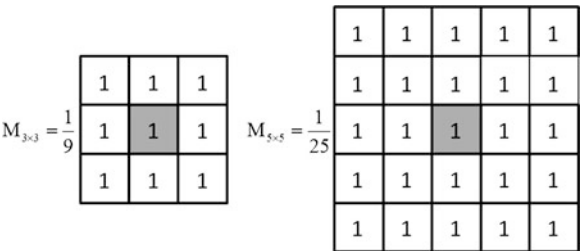




Abb. 7.6 Filterung eines Originalbildes (a) mit einem Mittelwertfilter der Dimension 5×5 ergibt das gefilterte Bild (b)

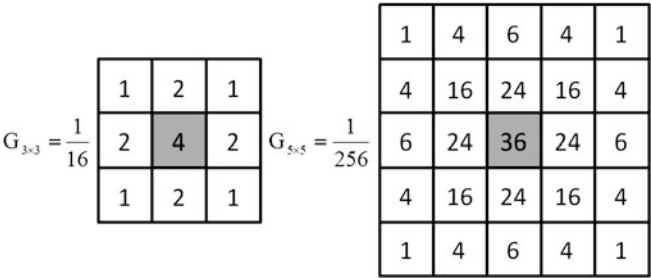


Abb. 7.7 Gauß-Filter der Größe 3×3 und 5×5

geflachten Kanten, wodurch das gefilterte Bild unscharf und weniger strukturiert erscheint (siehe Abb. 7.6).

Gauß-Filter Der Gauß-Filter ist ein linearer Filter zur Rauschreduzierung in der Form einer zweidimensionalen Normalverteilung:

$$N(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right).$$

Bei dem Gauß-Filter wird der Einfluss der Umgebungspunkte auf das Filterergebnis in Abhängigkeit vom Abstand zum zentralen Bildpunkt gewichtet. In Abb. 7.7 ist der zugehörige diskretisierte Gauß-Filter dargestellt (siehe Abb. 7.8).



Abb. 7.8 Filterung eines Originalbildes (a) mit einem Gauß-Filter der Dimension 5×5 ergibt das gefilterte Bild (b)

Medianfilter Der Medianfilter ist ein nicht linearer Glättungsfilter, bei dem sehr kleine Strukturen eliminiert werden. Hierbei wird einem Pixel (x, y) der Median in seiner lokalen Umgebung $U(x, y)$ auftretender Bildfunktionswerte zugeordnet:

$$f_{\text{Median}}(x, y) = \text{Median}_{(x_i, y_i) \in U(x, y)} \{f(x_i, y_i)\}.$$

Zur Bestimmung des Medians werden die in der Umgebung $U(x, y)$ auftretenden Bildfunktionswerte aufsteigend sortiert. Der Median ist dann durch den in der mittleren Position stehenden Bildfunktionswert repräsentiert.

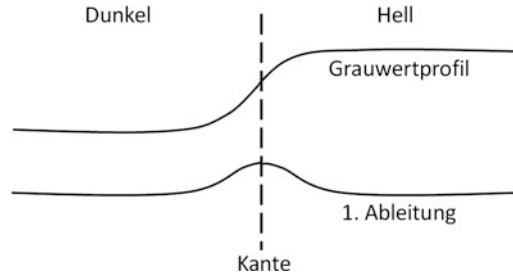
Kanten- bzw. Hochpassfilter

Hochpassfilter ermöglichen die Detektion von Kanten in einem Bild, also die Erkennung von steilen Grauwertveränderungen. Diese Filter werden vor allem zur Hervorhebung und zur Extraktion von Kanten und Konturen im Bild eingesetzt. Eine Kante in einem Bild ist gekennzeichnet durch eine starke lokale Veränderung der Bildfunktion, d. h., der Gradient der Bildfunktion f nimmt hier ein lokales Maximum an (siehe Abb. 7.9):

$$\text{grad } f(x, y) = \nabla f(x, y) = \begin{pmatrix} \frac{\partial f(x, y)}{\partial x} \\ \frac{\partial f(x, y)}{\partial y} \end{pmatrix} = \begin{pmatrix} f_x(x, y) \\ f_y(x, y) \end{pmatrix}.$$

Der Gradient weist stets in Richtung des stärksten Anstieges der Bildfunktion. Der Gradient ist ein Maß der Kantenstärke, und die Gradientenrichtung ist ein Maß für die Kanten-

Abb. 7.9 Verlauf der Werte einer Bildzeile mit Grauwertprofil und die zugehörige 1. Ableitung beim Übergang von einem dunklen zu einem hellen Bereich



richtung, bestimmt durch den Winkel

$$\varphi = \arctan \frac{f_y(x, y)}{f_x(x, y)}.$$

Differenzenoperatoren Die Herleitung des Differenzenoperators erfolgt durch Differenzieren der Bildmatrix nach dem Ort mit der kleinstmöglichen Ortsdifferenz $\Delta x = \Delta y = 1$:

$$f_x(x, y) \approx \frac{\Delta f(x, y)}{\Delta x} = f(x, y) - f(x - 1, y)$$

bzw.

$$f_y(x, y) \approx \frac{\Delta f(x, y)}{\Delta y} = f(x, y) - f(x, y - 1).$$

Die Differenzierung lässt sich mit einer Maske der Größe 3×3 wie folgt schreiben:

$$D_x = \begin{pmatrix} 0 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad \text{bzw.} \quad D_y = \begin{pmatrix} 0 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

Alternativ kann der Gradient durch Betrachtung der beiden Nachbarn in x - und y -Richtung durch den symmetrischen Differenzenoperator approximiert werden:

$$f_x(x, y) \approx \frac{\Delta f(x, y)}{\Delta x} = \frac{f(x + 1, y) - f(x - 1, y)}{2}$$

bzw.

$$f_y(x, y) \approx \frac{\Delta f(x, y)}{\Delta y} = \frac{f(x, y + 1) - f(x, y - 1)}{2}.$$

Die Differenzierung lässt sich mit einer Maske der Größe 3×3 wie folgt schreiben:

$$SD_x = \frac{1}{2} \begin{pmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \quad \text{bzw.} \quad SD_y = \frac{1}{2} \begin{pmatrix} 0 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}.$$

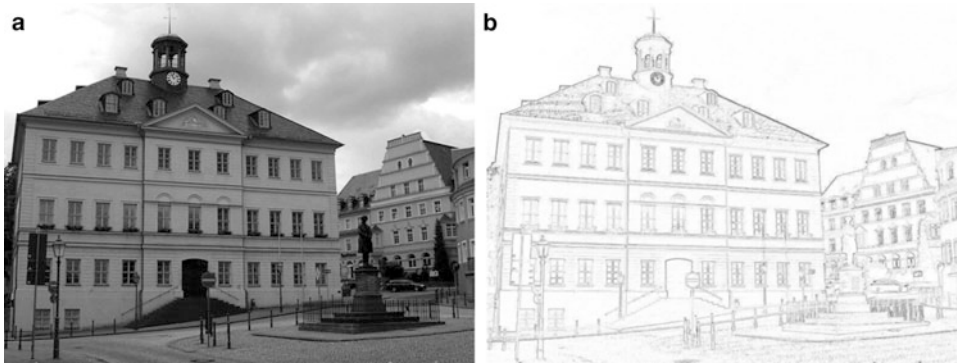


Abb. 7.10 Filterung eines Originalbildes (a) mit einem Differenzenoperator ergibt ein gefiltertes Bild (b)

Durch Einsetzen der diskreten Approximationen der partiellen Ableitungen erhalten wir das Gradientenbild in Abb. 7.10.

$$\text{grad}(f(x, y)) = \sqrt{f_x(x, y)^2 + f_y(x, y)^2}.$$

Der Betrag des Gradienten ist ein Maß für die Stärke der Änderung der Bildfunktion f .

Sobel-Operatoren Die Differenzenoperatoren sind sehr störanfällig, da sie jede Änderung der Grauwerte detektieren. Ein weiterer Nachteil ist, dass die Kanten um ein halbes Pixel verschoben sind. Aus diesem Grunde werden für viele praktische Anwendungen weitere Bildpunkte durch den Sobel-Operator benutzt. Der Sobel-Operator für vertikale (S_x) und horizontale (S_y) Kanten ist definiert durch

$$S_x = \frac{1}{8} \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \quad \text{bzw.} \quad S_y = \frac{1}{8} \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

bzw. für diagonale Kanten durch

$$S_{\setminus} = \frac{1}{8} \begin{pmatrix} 0 & -1 & -2 \\ 1 & 0 & 1 \\ 2 & 1 & 1 \end{pmatrix} \quad \text{bzw.} \quad S_{/} = \frac{1}{8} \begin{pmatrix} -2 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 2 \end{pmatrix}.$$

Neben der Verstärkung kann auch eine Verbreiterung der Kanten erfolgen. Ein Beispiel für die Anwendung des Sobel-Operators ist in Abb. 7.11 dargestellt.

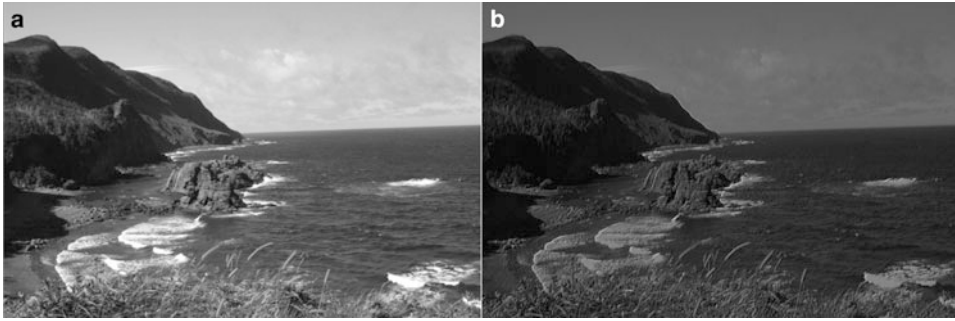


Abb. 7.11 Filterung eines Originalbildes (a) mit einem Sobel-Operator für vertikale (S_x) und horizontale (S_y) Kanten ergibt ein gefiltertes Bild (b)

Abb. 7.12 Laplace-Operator der Größe 3×3

$$L_4 = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad L_8 = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Laplace-Operator Bei der Anwendung des Laplace-Operators besitzt der Betrag der 2. Ableitung einer Bildfunktion f bei einer Kante einen Nulldurchgang. Zur Vermeidung der Richtungsabhängigkeit ist der Laplace-Operator die Summe aus den 2. Ableitungen nach x ,

$$\frac{\Delta(\Delta(f(x, y)))}{\Delta x^2} = f(x + 1, y) - 2f(x, y) + f(x - 1, y),$$

und nach y ,

$$\frac{\Delta(\Delta(f(x, y)))}{\Delta y^2} = f(x, y + 1) - 2f(x, y) + f(x, y - 1).$$

Der Laplace-Operator ist dann definiert durch

$$L = \frac{\Delta(\Delta(f(x, y)))}{\Delta x^2} + \frac{\Delta(\Delta(f(x, y)))}{\Delta y^2}.$$

Mit dem Laplace-Operator werden die Nulldurchgänge eines Laplace-gefilterten Bildes bestimmt. der Laplace-Operator ist ebenfalls ein linearer Operator, dargestellt in Abb. 7.12.

Nachteil des Laplace-Operators ist die große Empfindlichkeit auf Rauschen, die zu Pseudokanten führen kann. Zur Vermeidung dieses Effektes wird der Laplace-Operator häufig mit Rauschreduktionsfiltern verwendet.

Prinzip der Filterung

1. Auswahl der Art des Filters:
 - (a) Glättungsfilter
 - (b) Kantenfilter

2. Auswahl eines speziellen Filtertyps mit der Maske $g^{(2k+1) \times (2k+1)}(x, y)$ der Dimension $(2k + 1) \times (2k + 1)$
3. Bestimmung des gefilterten Bildes $\tilde{f}$ durch Anwendung der Maske g auf jedes Bildpixel (x, y) :

$$\tilde{f}(x, y) = \sum_{i=-k}^k \sum_{j=-k}^k f(x + i, y + j) \cdot g(i, j)$$

7.2.3 Algorithmus und Implementierung

Wir geben nun den Pseudocode zur Berechnung eines gefilterten Bildes $\tilde{f}$ durch Anwendung einer Maske g auf die gegebenen Bilddaten f an.

Algorithmus 17 FILTERUNG

Input: Bilddaten f der Größe $m \times n$, Bildmaske g der Dimension $(2k + 1) \times (2k + 1)$

Output: gefiltertes Bild $\tilde{f}$

```

1: for y = 1 to m do
2:   for x = 1 to n do
3:      $\tilde{f}(y, x) = 0$ 
4:     for i = -k to k do
5:       for j = -k to k do
6:          $\tilde{f}(y, x) = \tilde{f}(y, x) + f(y + i, x + j) \cdot g(i, j)$ 

```

Allgemeine Erklärung

Das Programm besteht aus insgesamt vier `for`-Schleifen, bei denen zwei über die Dimension des Bildes laufen (Zeile 1–2) und die anderen zwei über die Dimension der Maske (Zeile 4–5).

Für die Implementierung dieser verschiedenen Filtertypen eignet sich das bereits vorgestellte objektorientierte Entwurfsmuster der Strategie am besten. Dazu definieren wir zunächst ein Interface `IFilter` zur Darstellung eines allgemeinen Filters:

```

public interface IFilter
{
    public void filterAlgorithmus(BildFilter f);
}

```

Dieses Interface dient als Schnittstelle für alle konkreten Strategien durch Definition der Methode `filterAlgorithmus`, in der dann die jeweilige konkrete Strategie implementiert wird.

Wir definieren nun die Klasse `BildFilter` des Kontextobjekts zur Speicherung der Bilddaten und der jeweiligen Strategie in der Variablen `IFilter filter`. Diese Klasse wird von der bereits vorgestellten Klasse `Bild` abgeleitet, und enthält damit alle deren Attribute und Methoden. In der Methode `setFilter` wird vom Anwender die gewünschte Strategie gesetzt. Die Methode `anwendenFilter` wird vom Anwender aufgerufen, um den jeweiligen konkreten Algorithmus zu starten.

```
public class BildFilter extends Bild
{
    IFilter filter = null;

    public BildFilter(int breite, int hoehe)
    {
        super(breite, hoehe);
    }
    public BildFilter(String name)
    {
        super(name);
    }

    public void setFilter(IFilter filter)
    {
        this.filter = filter;
    }

    public void anwendenFilter()
    {
        filter.filterAlgorithmus(this);
        this.anzeigen();
    }
}
```

Die beiden folgenden Klassen definieren jeweils die konkreten Strategien für den Glättungs- und den Kantenfilter am Beispiel von Graubildern:

- Klasse `Glaettung`: Instanzvariablen `name` für den Filternamen, `groesse` für die Größe der Maske und `filter` für den ausgewählten Bildfilter. Die Auswahl der Filter erfolgt in der Methode `setFilter()`. Der zugehörige Filteralgorithmus wird in der Methode `filterAlgorithmus` implementiert. Die restlichen Methoden sind die Hilfsmethoden `conv2` zur Faltung und `color2Int` zur Umwandlung der Bilddaten in die Grauwerte.

```
public class Glaettung implements IFilter
{
    private String name;
    private int groesse;
    private double filter[][];
```

```

public Glaettung(String name, int groesse)
{
    this.name = name;
    this.groesse = groesse;
    this.setFilter();
}

// Auswahl des Filtertyps
public void setFilter()
{
    filter = new double[groesse][groesse];
    switch (name)
    {
        case "Mittelwert":
            for(int i=0; i<filter.length; i++)
                for(int j=0; j<filter.length; j++)
                    filter[i][j] = berechneMittelwert();
            break;

        case "Gauss":
            double sum = 0;
            for(int i=0; i<filter.length; i++)
            {
                for(int j=0; j<filter.length; j++)
                {
                    filter[i][j] = berechneGauss(i - groesse/2, j - groesse/2);
                    sum = sum + filter[i][j];
                }
            }
            LinAlgebra.mult(filter, sum);
            break;

        default:
            System.out.printf("Filter nicht vorhanden!\n");
    }
}

public double berechneMittelwert()
{
    return 1. / (groesse * groesse);
}

public double berechneGauss(int x, int y)
{
    return 1. / (2 * Math.PI) * Math.exp(-0.5 * (x*x + y*y));
}

public int conv2(int f[][], int x, int y)
{
    int m = (groesse - 1) / 2;
    double farbe = 0;
    for (int i=-m; i <= m; i++)
        for (int j=-m; j <= m; j++)
            farbe = farbe + f[x+i][y+j] * filter[m+i][m+j];
    return (int) farbe;
}

public static int[][] color2Int(BildFilter bild)

```

```

public void filterAlgorithmus(BildFilter bild)
{
    int f[][] = color2Int(bild);
    int breite = bild.getBreite()/2;
    int hoehe = bild.getHoehe();
    int m = (groesse - 1)/2;

    int wert = 0;
    for (int x = breite+1 + m; x < 2*breite-m; x++)
    {
        for (int y = m; y < hoehe-m; y++)
        {
            wert = conv2(f, x-breite, y);
            bild.set(x, y, new Color(wert, wert, wert));
        }
    }
}

```

- Klasse Kanten: Instanzvariablen name für den Filternamen, groesse für die Größe der Maske und filter_x bzw. filter_y für den ausgewählten Kantentfilter in x- und y-Richtung. Die Auswahl der Filter erfolgt wiederum in der Methode setFilter(). Der zugehörige Filteralgorithmus wird in der Methode filterAlgorithmus implementiert. Die restlichen Methoden sind Hilfsmethoden conv2 zur Faltung und color2int zur Umwandlung der Bilddaten in die Grauwerte.

```

public class Kanten implements IFilter
{
    private String name;
    private int groesse = 3;
    private double filter_x[][];
    private double filter_y[][];

    public Kanten(String name)
    {
        this.name = name;
        this.filter_x = this.setFilter(name + "x");
        this.filter_y = this.setFilter(name + "y");
    }

    // Auswahl des Filtertyps
    public double[][] setFilter(String fname)
    {
        switch (fname)
        {
            case "Diffx":
                double filter1[][] = {{0,0,0},{-1,1,0},{0,0,0}};
                return filter1;
            case "Diffy":
                double filter2[][] = {{0,-1,0},{0,1,0},{0,0,0}};
                return filter2;
            case "SymDiffx":
                double filter3[][] = {{0,0,0},{-1.0/2,0,1.0/2},{0,0,0}};
                return filter3;
            case "SymDiffy":
                double filter4[][] = {{0,-1.0/2,0},{0,0,0},{0,1.0/2,0}};
                return filter4;
        }
    }
}

```



```

        case "Sobelx":
            double filter5[][] = {{-1.0/8,0,1.0/8},{-2.0/8,0,2.0/8},{-1.0/8,0,1.0/8}};
            return filter5;
        case "Sobely":
            double filter6[][] = {{-1.0/8,-2.0/8,1.0/8},{0,0,0},{1.0/8,2.0/8,1.0/8}};
            return filter6;
        default:
            System.out.printf("Filter nicht vorhanden!\n");
            return null;
    }
}

public int conv2(int f[], int x, int y, double filter[][])

public static int[][] color2int(BildFilter bild)

public void filterAlgorithmus(BildFilter bild)
{
    int f[][] = color2int(bild);
    int breite = bild.getBreite()/2;
    int hoehe = bild.getHoehe();
    int m = (groesse - 1)/2;

    int wert = 0;
    for (int x = breite+1 + m; x < 2*breite-m; x++)
    {
        for (int y = m; y < hoehe-m; y++)
        {
            double fx = conv2(f, x-breite, y, filter_x);
            double fy = conv2(f, x-breite, y, filter_y);
            wert = (int) Math.sqrt(fx*fx + fy*fy);
            bild.set(x, y, new Color(wert, wert, wert));
        }
    }
}
}

```

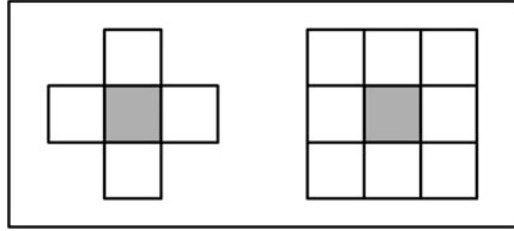
Der Anwender verwendet die vorgestellten Klassen durch Erzeugung einer Instanz der Klasse `BildFilter` und Setzen des Filtertyps mit der Methode `setFilter`.

```

public class TestFilter
{
    public static void main(String args[])
    {
        BildFilter bild = new BildFilter("MR.jpg");
        bild.setFilter(new Glaettung("Mittelwert", 5)); //Mittelwert, Gauss
        bild.setFilter(new Kanten("Sobel")); //Diff, SymDiff, Sobel
        bild.anwendenFilter();
    }
}

```

Abb. 7.13 2-D-Nachbarschaft:
4 Nachbarn und 8 Nachbarn
eines Bildpunktes



7.2.4 Anwendungen

Wir betrachten einige Anwendungen von Filteroperationen im Ortsbereich.

Elimination gestörter Bildpunkte In einigen Bildern tauchen manchmal einzelne Pixelstörungen auf, beispielsweise wenn ein Sensor störenden Einflüssen ausgesetzt ist. In diesem Fall besitzt ein Grauwert eines Pixels einen ganz anderen Wert als die Grauwerte der umliegenden Umgebung.

Zur Erkennung und Eliminierung von gestörten Bildpunkten wird der Grauwert jedes Bildpunktes mit dem Mittelwert seiner 8 Nachbarn verglichen (siehe Abb. 7.13). Falls diese Differenz der beiden Werte einen bestimmten Schwellwert c überschreitet, wird der Bildpunkt als gestört erkannt und durch den Mittelwert m ersetzt:

$$\tilde{f}(x, y) = \begin{cases} m, & |m - f(x, y)| > c, \\ f(x, y), & \text{sonst,} \end{cases}$$

mit dem Mittelwert

$$m = \frac{1}{8} \sum_{i=-1}^1 \sum_{j=-1}^1 f(x+i, y+j) \cdot g(i, j)$$

und der Filtermaske

$$G = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}.$$

Falls der Schwellwert c zu klein gewählt wird, findet eine Glättung mit dem Mittelwertfilter statt. Für ausreichend große Schwellwerte c werden die Grauwerte der Umgebung hingegen nicht verändert, d. h., es findet keine ungewollte Glättung statt.

Extraktion von Kanten Die vorgestellten Kantenfilter in Form von Differenzenoperatoren eignen sich gut zur Extraktion und Weiterverarbeitung von Kanten (siehe Abb. 7.14). Die Kanten eines Bildes f werden zunächst mit einem Differenzenoperator wie dem

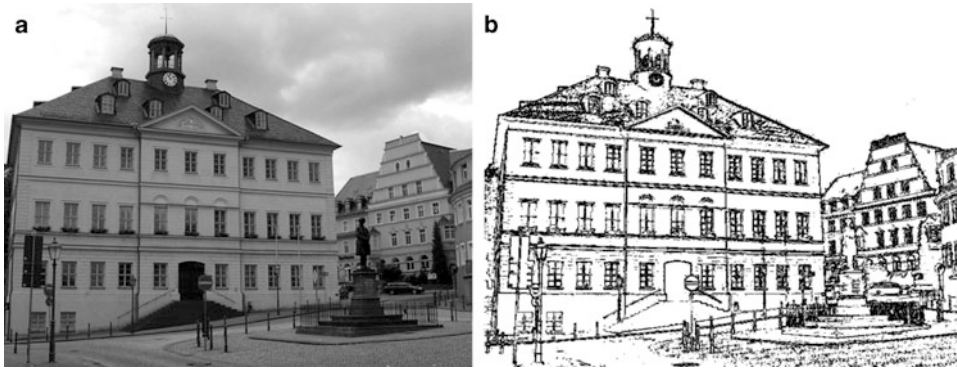


Abb. 7.14 Extraktion der Kanten aus einem Originalbild (a) ergibt ein gefiltertes Bild (b)

Sobel-Operator verstärkt. Anschließend wird der Betrag des Gradienten aus den beiden Richtungskomponenten bestimmt:

$$g = \text{grad}(f(x, y)) = \sqrt{f_x(x, y)^2 + f_y(x, y)^2}.$$

Ein hoher Betrag eines Gradienten kennzeichnet die Bildpunkte, die auf einer Kante liegen. In Form einer einfachen Schwellwertberechnung werden Bildpunkte $\tilde{f}(x, y)$ des gefilterten Bildes mit einem hohen Gradientenbetrag als Kantenpunkte gekennzeichnet:

$$f'(x, y) = \begin{cases} 1, & \text{grad}(f(x, y)) > c, \\ 0, & \text{sonst.} \end{cases}$$

Bei dieser relativ einfachen Methode zur Kantenextraktion kann das Problem auftreten, dass vorhandene Kanten des Bildes entweder nicht erkannt werden oder sich nicht deutlich von anderen Kanten abheben. Der Grund dafür ist, dass in diesem Fall der Wert des Gradienten unter dem Schwellwert liegt. In diesem Fall ist es hilfreich, das entstehende Binärbild von isolierten Einsen zu säubern, die entweder die Linien verdünnen oder unterbrechen. Eine Möglichkeit ist die sogenannte Dilatation, bei der der binäre Wert eines Pixels durch das Maximum der 8-Nachbarschaft mit dem aktuellen Pixelwert bestimmt wird.

7.3 Registrierung

Unter der Registrierung von Bilddaten versteht man die Darstellung von verschiedenen Bildern in einem gemeinsamen Koordinatensystem. Bei der Registrierung werden unterschiedliche Bilder in ein gemeinsames Koordinatensystem zusammengeführt. Damit

müssen die Koordinatensysteme der unterschiedlichen Bilddaten angepasst und in ein gemeinsames Koordinatensystem transformiert werden. In der medizinischen Bildverarbeitung können beispielsweise CT- und MRT-Daten eines Patienten zusammengefasst werden, um so mehr Informationen für die Entscheidungsfindung einer geeigneten Therapie zu besitzen. Bei der Entwicklung des autonomen Fahrens spielen diese Verfahren zur Lokalisierung von Fahrzeugen durch aufgenommene Scanbilder eine bedeutende Rolle.

7.3.1 Problemstellung

Bei der Registrierung betrachten wir zwei Bilddatensätze, von denen einer als Referenzbild R und der andere als Templatebild T bezeichnet werden.

REGISTRIERUNG

Gegeben: zwei Bilder $R : \mathbb{R}^d \rightarrow \mathbb{R}$ und $T : \mathbb{R}^d \rightarrow \mathbb{R}$.

Gesucht: Transformation $t : \mathbb{R}^d \rightarrow \mathbb{R}^d$ mit $d \in \{2, 3\}$, sodass das transformierte Templatebild $T(t(x))$ dem Referenzbild $R(x)$ ähnlich ist.

Die Bildregistrierung kann im Allgemeinen in die folgenden Anwendungsbereiche gegliedert werden:

- **Kameraposition:** Die Bilder zeigen die gleiche Szene bzw. Objekt, jedoch von verschiedenen Kamerapositionen.
- **Zeitpunkte:** Die Bilder zeigen die gleiche Szene bzw. Objekt, jedoch zu unterschiedlichen Zeitpunkten.
- **Bildgebende Verfahren:** Die Bilder zeigen die gleiche Szene bzw. Objekt, jedoch von verschiedenen bildgebenden Verfahren.
- **Modell:** Die Bilder eines Objekts bzw. einer Szene werden mit dem dazugehörigen Modell registriert.

Für die zahlreichen bildgebenden Verfahren gibt es verschiedene Arten von Registrierungsverfahren:

- **Dimension der Bilddaten:** 2-D-2-D (z. B. Vergleich von Bildstrukturen zu unterschiedlichen Zeiten), 2-D-3-D (z. B. CT-Bilder in Röntgenbilder einblenden), 3-D-3-D (z. B. Ausrichtung verschiedener medizinischer Gerätetypen).
- **Transformationsklasse:** Arten der Transformation der Bilddaten mit mathematischen Transformationen.

7.3.2 Grundlegende Lösungsprinzipien

In der Regel besitzen die unterschiedlichen Registrierungsverfahren den folgenden Aufbau:

1. **Auswahl von Merkmalen:** In den zu registrierenden Bildern werden gewisse Merkmale manuell oder automatisch detektiert.
2. **Zuordnung der Merkmalspunkte:** Die Merkmalspunkte in den Bildern werden einander zugeordnet.
3. **Berechnung der Transformation:** Aus einer ausgewählten Transformationsklasse werden die zugehörigen Transformationsparameter berechnet.
4. **Durchführung der Transformation:** Das Objektbild wird mit der im vorherigen Schritt berechneten Umbildung transformiert.

Bei den sogenannten flächenbasierten Verfahren erfolgt die Registrierung direkt mit den Intensitätswerten, ohne dass Merkmale ausgewählt werden. Bei den merkmalsbasierten Verfahren muss hingegen eine gewisse Anzahl von Merkmalen extrahiert werden. Von zentraler Bedeutung bei Registrierungsverfahren sind die Transformationsklassen, die sich in die starren, affinen und perspektivischen Transformationen einteilen lassen. In Abb. 7.15 werden diese Transformationstypen an einem Rechteck angewandt: starre Transformation (Rotationen, Translationen), affine Transformation (Abbildung parallele Linien auf parallele Linien), perspektivische Transformation (Linien auf Linien abgebildet).

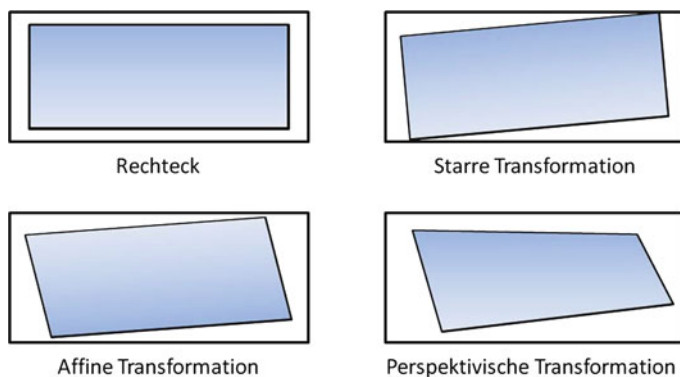


Abb. 7.15 Rechteck wird transformiert: starre Transformation (Rotationen, Translationen), affine Transformation (Abbildung parallele Linien auf parallele Linien), perspektivische Transformation (Linien auf Linien abgebildet)

Wir stellen im Folgenden die verschiedenen Transformationstypen vor. Die beschriebenen Transformationen können ebenfalls durch homogene Koordinaten beschrieben werden. Bei homogenen Koordinaten wird jeder Punkt (x, y, z) eines dreidimensionalen Gebietes durch einen vierdimensionalen Punkt $(x, y, z, 1)$ dargestellt. In diesem Fall werden alle Transformationen im dreidimensionalen Raum mithilfe von 4×4 -Matrizen durchgeführt. Für eine weiterführende Beschreibung dieser Darstellung verweisen wir auf den 1. Band, auf das Kapitel Technische und naturwissenschaftliche Anwendungen.

Starre Transformation Bei einer starren Transformation werden nur Rotationen und Translationen für die Koordinatentransformationen angewandt. Diese Transformationsart $t : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ wird durch den Ortsvektor eines Koordinatenpunktes $\vec{r} = (x, y, z)^T$, durch eine Matrix $R \in \mathbb{R}^{3 \times 3}$ und einen Translationsvektor $\vec{s} = (s_x, s_y, s_z)^T$ beschrieben:

$$t : \vec{r}' = R \cdot \vec{r} + \vec{s}.$$

Diese Transformation sollte durch homogene Matrizen beschrieben werden, sodass auch die Translation durch eine Matrixmultiplikation darstellbar ist.³

Die starre Transformation wird durch 6 Parameter, die Rotationswinkel α_x, α_y und α_z sowie die Komponenten s_x, s_y und s_z des Translationsvektors, eindeutig beschrieben. Bei einer starren Transformation ist die Matrix R eine Rotationsmatrix:

$$R = R_1 \cdot R_2 \cdot R_3$$

mit

- Drehmatrix um x -Achse:

$$R = R_1(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \alpha & -\sin \alpha \\ 0 & \sin \alpha & \cos \alpha \end{pmatrix};$$

- Drehmatrix um y -Achse:

$$R = R_2(\beta) = \begin{pmatrix} \cos \beta & 0 & -\sin \beta \\ 0 & 1 & 0 \\ \sin \beta & 0 & \cos \beta \end{pmatrix};$$

- Drehmatrix um z -Achse:

$$R = R_3(\gamma) = \begin{pmatrix} \cos \gamma & -\sin \gamma & 0 \\ \sin \gamma & \cos \gamma & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

³ Siehe Buch *Grundlagen*, Kapitel Technische und naturwissenschaftliche Anwendungen.

Dann erhalten wir für eine Drehung im $\mathbb{R}^3$ um die drei Koordinatenachsen:

$$R(\alpha, \beta, \gamma) := R_1(\alpha) R_2(\beta) R_3(\gamma) \\ = \begin{pmatrix} \cos \beta \cos \gamma & -\cos \beta \sin \gamma & -\sin \gamma \\ -\sin \alpha \sin \beta \cos \gamma + \cos \alpha \sin \gamma & \sin \alpha \sin \beta \sin \gamma + \cos \alpha \cos \gamma & -\sin \alpha \cos \beta \\ \cos \alpha \sin \beta \cos \gamma + \sin \alpha \sin \gamma & -\cos \alpha \sin \beta \sin \gamma + \sin \alpha \cos \gamma & \cos \alpha \cos \beta \end{pmatrix}.$$

Affine Transformation Bei affinen Transformationen werden parallele Linien auf parallele Linien abgebildet. Bei einer affinen Transformation $t: \mathbb{R}^3 \rightarrow \mathbb{R}^3$ wird der Ortsvektor eines Koordinatenpunktes $\vec{r} = (x, y, z)^T$ durch eine Matrix $R \in \mathbb{R}^{3 \times 3}$ und einen Translationsvektor $\vec{s} = (s_x, s_y, s_z)^T$ wie folgt beschrieben:

$$t: \vec{r}' = R \cdot \vec{r} + \vec{s}.$$

Bei einer affinen Transformation sind die 9 Einträge der Matrix R sowie die 3 Komponenten des Translationsvektors anzugeben.

Perspektivische Transformation Bei perspektivischen Transformationen wird nur garantiert, dass Linien auf Linien abgebildet werden. Bei einer perspektivischen Transformation sind 15 Parameter anzugeben.

Landmarkenbasierte Registrierung

Bei der landmarkenbasierten Registrierung erfolgt die Anpassung an verschiedene Bilddaten durch ausgewählte Punkte, die sogenannten Landmarken. Das Registrierungsproblem ist ein klassisches Optimierungsproblem: Gegeben sind die Paare (A_i, B_i) , $i = 1, \dots, n$ von n zueinander korrespondierenden Landmarken in den beiden Bilddatensätzen A und B . Die Aufgabe besteht in der Bestimmung einer Koordinatentransformation t mit den zugehörigen Transformationsparametern innerhalb einer vorgegebenen Transformationsklasse, sodass

$$\sum_{i=1}^n |t(A_i) - B_i|^2 \rightarrow \text{Minimum}.$$

Damit können die zugehörigen Transformationsparameter in der Funktion t mithilfe der Gauß'schen Methode der kleinsten Fehlerquadrate bestimmt werden.

Wir verwenden eine affine Transformation, bei der parallele Linien auf parallele Linien abgebildet werden, die durch die Transformationsvorschrift $t(x) = Cx + d$ mit einer Matrix C und einem Verschiebungsvektor d beschrieben wird. Es seien $A_i = (x_i^A, y_i^A)$ und $B_i = (x_i^B, y_i^B)$ für $i = 1, \dots, n$ die gegebenen Punkte aus den beiden CT-Bildern A und B . Die Aufgabe besteht in der Transformation des Bilddatensatzes B in den Bilddatensatz A . Gesucht sind somit die Parameter $c_{11}, c_{12}, c_{21}, c_{22}$ und d_1, d_2 aus

der Transformationsvorschrift

$$\begin{pmatrix} x_i^A \\ y_i^A \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} \cdot \begin{pmatrix} x_i^B \\ y_i^B \end{pmatrix} + \begin{pmatrix} d_1 \\ d_2 \end{pmatrix}$$

$$= \begin{pmatrix} x_i^B & y_i^B & 0 & 0 & 1 & 0 \\ 0 & 0 & x_i^B & y_i^B & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} c_{11} \\ c_{12} \\ c_{21} \\ c_{22} \\ d_1 \\ d_2 \end{pmatrix}.$$

Damit erhalten wir in Matrizenschreibweise

$$A = \begin{pmatrix} x_1^B & y_1^B & 0 & 0 & 1 & 0 \\ 0 & 0 & x_1^B & y_1^B & 0 & 1 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ x_n^B & y_n^B & 0 & 0 & 1 & 0 \\ 0 & 0 & x_n^B & y_n^B & 0 & 1 \end{pmatrix}, \quad b = \begin{pmatrix} x_1^A \\ y_1^A \\ \dots \\ x_n^A \\ y_n^A \end{pmatrix} \quad \text{und} \quad c = \begin{pmatrix} c_{11} \\ c_{12} \\ c_{21} \\ c_{22} \\ d_1 \\ d_2 \end{pmatrix}$$

das resultierende Minimumproblem

$$\Phi(c) = |Ac - b|^2 \rightarrow \min.$$

Die Aufgabe ist nun, die unbekannten Koeffizienten $c = (c_1, c_2, \dots, c_6)^T$ so zu bestimmen, dass die Funktion ein Minimum annimmt, um so die Summe aller Abstände zwischen den Approximationspolynomen und den gegebenen Punkten zu minimieren. Wir berechnen dazu die partiellen Ableitungen nach den Koeffizienten c_i , $i = 1, 2, \dots, 6$ und setzen diese null:

$$\frac{\partial \Phi}{\partial c_k} = 2 \sum_{j=0}^m \left(\sum_{i=0}^n c_i x_j^i - y_j \right) x_j^k = 0, \quad k = 0, \dots, n$$

bzw. in Matrizenschreibweise

$$\frac{\partial \Phi}{\partial c} = 2 (A^T A c - A^T b) = 0,$$

wobei die Matrix A^T durch die spaltenweise Summierung mit den Elementen x_j^k entsteht.

Aus der notwendigen Bedingung für ein Minimum folgt das Gauß'sche Normalgleichungssystem

$$A^T A c = A^T b.$$

Prinzip der landmarkenbasierten Registrierung

1. Aufstellung der Matrix A und des Vektors b aus den beiden Bilddatensätzen $A_i = (x_i^A, y_i^A)$ und $B_i = (x_i^B, y_i^B)$ für alle Landmarken $i = 1, \dots, n$
2. Berechnung der Matrix $A^T A$ und des Vektors $A^T b$
3. Lösung des linearen Gleichungssystems $A^T A c = A^T b$
4. Transformation aller Bildpunkte (x^B, y^B) im Datensatz B in die Koordinaten vom Datensatz A :

$$\begin{pmatrix} x^A \\ y^A \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} \cdot \begin{pmatrix} x^B \\ y^B \end{pmatrix} + \begin{pmatrix} d_1 \\ d_2 \end{pmatrix}$$

Neben der landmarkenbasierten Registrierung gibt es weitere Registrierungsverfahren wie beispielsweise die intensitätsbasierte Registrierung.

7.3.3 Algorithmus und Implementierung

Wir geben nun den Pseudocode an, um eine landmarkenbasierte Registrierung zu implementieren. Wir verwenden dazu die obige Transformationsvorschrift

$$\begin{pmatrix} x_i^A \\ y_i^A \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} \cdot \begin{pmatrix} x_i^B \\ y_i^B \end{pmatrix} + \begin{pmatrix} d_1 \\ d_2 \end{pmatrix}$$

zur Bestimmung der unbekannten Parameter $c_{11}, c_{12}, c_{21}, c_{22}$ und d_1, d_2 aus den gegebenen Bildpunkten $A_i = (x_i^A, y_i^A)$ und $B_i = (x_i^B, y_i^B)$ für $i = 1 \dots, n$. Zur Lösung des aufgestellten Gleichungssystems verwenden wir die im 1. Buch vorgestellte LR-Zerlegung.

Algorithmus 18 LANDMARKENBASIERTE REGISTRIERUNG

Input: zwei Bilddaten A, B ; Bildpunkte $A_i = (x_i^A, y_i^A)$ und $B_i = (x_i^B, y_i^B)$ für $i = 1 \dots, n$

Output: Transformationsvektor $c = (c_{11}, c_{12}, c_{21}, c_{22}, d_1, d_2)$

```

1: for  $j = 0$  to  $n$  do
2:    $a_{i*} = (x_i^B, y_i^B, 0, 0, 1, 0)$ 
3:    $a_{i+1*} = (0, 0, x_i^B, y_i^B, 0, 1)$ 
4:    $b_i = x_i^A$ 
5:    $b_{i+1} = y_i^A$ 
6:  $[L, R] = \text{LR}(A^T A)$ 
7:  $x = \text{VORWÄRTSEINSETZEN}(L, A^T b)$ 
8:  $c = \text{RÜCKWÄRTSEINSETZEN}(R, x)$ 

```

Allgemeine Erklärung

In Zeile 1–5 stellen wir die Koeffizientenmatrix A und die rechte Seite b aus den zugehörigen Landmarken auf. In Zeile 6–8 lösen wir das zugehörige Normalgleichungssystem $A^T A c = A^T b$ durch Berechnung des Koeffizientenvektors c mithilfe der bekannten LR-Zerlegung.

Die Implementierung dieser Registrierungsverfahren kann wiederum mit dem objekt-orientierten Entwurfsmuster der Strategie erfolgen. Wir beschränken uns hier auf nur ein Verfahren und implementieren die vorgestellte landmarkenbasierte Registrierung kompakt in nur einer Klasse Landmarken mit den folgenden Methoden:

- `erstelleMatrix`: Aufstellung der Koeffizientenmatrix aus den Landmarken:

```
public static double[][] erstelleMatrix(double Px[], double Py[])
{
    double A[][] = new double[2*Px.length][6];
    for(int i=0; i<Px.length; i++)
    {
        // Zeile fuer Px[i]
        A[2*i][0] = Px[i];
        A[2*i][1] = Py[i];
        A[2*i][4] = 1;
        // Zeile fuer Py[i]
        A[2*i+1][2] = Px[i];
        A[2*i+1][3] = Py[i];
        A[2*i+1][5] = 1;
    }
    return A;
}
```

- `erstelleMatrix`: Aufstellung der rechten Seite aus den Landmarken:

```
public static double[] erstelleVektor(double Px[], double Py[])
{
    double b[] = new double[2*Px.length];
    for(int i=0; i<Px.length; i++)
    {
        b[2*i] = Px[i];
        b[2*i+1] = Py[i];
    }
    return b;
}
```

- `methKleinsteFehlerQuad`: Anwendung der Methode der kleinsten Quadrate:

```
public static double[] methKleinsteFehlerQuad(double A[][], double b[])
{
    // Aufstellung Matrix B = A^T A
    double B[][] = LinAlgebra.mult(LinAlgebra.transpose(A), A);
}
```

```

// Aufstellung Vektor  $c = A^T b$ 
double c[] = LinAlgebra.mult(LinAlgebra.transpose(A), b);

// Loesung Normalgleichung  $Bx = c$ 
return LinGS.gs(B, c);
}

```

- berechneTransformation: Berechnung der linearen Transformation:

```

public static int[] berechneTransformation(double koeff[], int x1, int x2)
{
    int y[] = new int[2];
    y[0] = (int) Math.round(koeff[0] * x1 + koeff[1] * x2 + koeff[4]);
    y[1] = (int) Math.round(koeff[2] * x1 + koeff[3] * x2 + koeff[5]);
    return y;
}

```

- transformiere: Transformation eines Bilddatensatzes mit obiger Transformationsvorschrift:

```

public static void transformiere(Bild bild, double koeff[])
{
    int breite = bild.getBreite();
    int hoehe = bild.getHoehe();
    Bild neu = new Bild(breite, hoehe);
    boolean markiert[][] = new boolean[breite][hoehe];

    // --- 1. Transformation des Bildes
    for(int i=0; i<breite; i++)
    {
        for(int j=0; j<hoehe; j++)
        {
            int x[] = berechneTransformation(koeff, i, j);
            if (x[0] >= 0 && x[1] >= 0 && x[0] < breite && x[1] < hoehe)
            {
                neu.set(x[0], x[1], bild.get(i, j));
                markiert[x[0]][x[1]] = true;
            }
        }
    }

    // --- 2. Nachbearbeitung des Bildes bzgl. fehlender Rasterung
    for(int i=1; i<breite-1; i++)
        for(int j=1; j<hoehe-1; j++)
            if(markiert[i][j]==false)
                neu.set(i,j, neu.get(i+1, j));
    neu.anzeigen();
}

```

- main: Anwendung der landmarkenbasierten Registrierung:

```

public static void main(String args[])
{
    // ----- Eingabe -----
    // --- 1. Bilddaten f. Transformation
    String datei = "MR2.jpg";

    //---2. Landmarken von Bild 1 (MR1.jpg)
    double Lx1[] = {20,157,153,178,217,252};
    double Ly1[] = {121,27,254,60,142,151};

    //---3. Landmarken von Bild 2 (MR2.jpg)
    double Lx2[] = {34,101,218,136,217,252};
    double Ly2[] = {179,24,226,38,90,79};

    // -----
    // --- 1. Aufstellung der Matrix A und Vektor b
    double A[][] = erstelleMatrix(Lx2, Ly2);
    double b[] = erstelleVektor(Lx1, Ly1);

    // --- 2. Berechnung der Koeffizienten der lin. Transformation  $y = Ax+b$ 
    double koeff[] = methKleinsteFehlerQuad(A, b);

    // --- 3. Erzeugung des Bildes
    Bild bild = new Bild(datei);

    // --- 4. Transformation des Bildes
    transformiere(bild, koeff);
}

```

7.3.4 Anwendungen

Wir betrachten einige Anwendungen von Registrierungsverfahren.

Medizinische Bildregistrierung Im Bereich der Medizin werden für die rechnergestützte Navigation und Operation oftmals Marker verwendet. Das Ziel ist es, die gewonnenen CT-Bilddaten an das Koordinatensystem des OP-Saales anzugleichen. In diesem Fall finden die Methodiken der Bildregistrierung hierzu breite Anwendung.

In Abb. 7.16 ist die Registrierung anhand zweier CT-Bilder dargestellt. Beide Bilder wurden in unterschiedlichen Koordinatensystemen und Lagen aufgenommen. In beiden Bildern werden nun an identischen Stellen gewisse Markerpunkte gesetzt (rote Punkte). Anschließend wird mit dem beschriebenen Algorithmus die zugehörige Transformationsvorschrift berechnet, und damit das CT-Bild 2 in das Koordinatensystem von CT-Bild 1 übertragen. Somit ist es nun möglich, beide Bilder miteinander zu vergleichen, um gegebenenfalls Therapieformen aus den bereits behandelten Patienten abzuleiten.

Qualitätskontrolle Die Qualitätskontrolle bei Bauteilen mit ausgeprägten Geometrien wie beispielsweise bei Dichtungen erfolgt in der Regel durch die hohe Komplexität auf

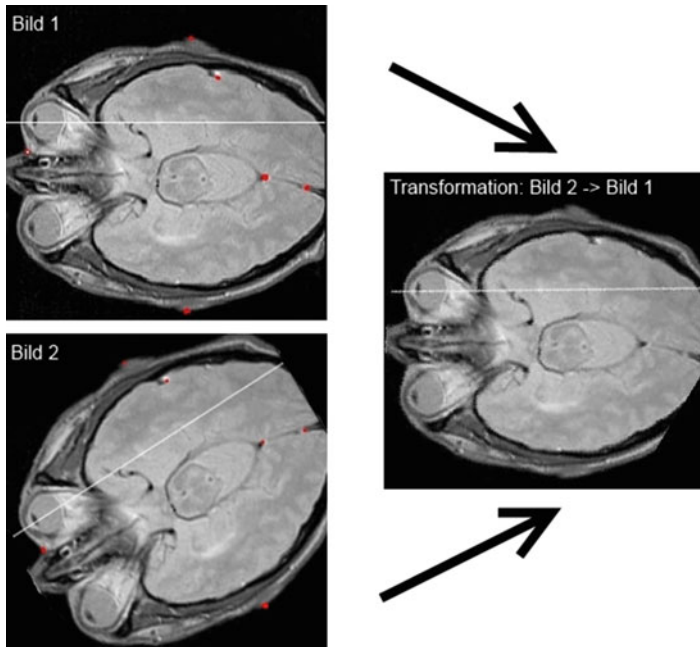


Abb. 7.16 Medizinische Bildregistration von einem CT-Bild 2 auf ein CT-Bild 1

manuelle Weise. Das Ziel im Bereich der Automatisierung ist, durch Bildaufnahmen im Produktionsprozess fehlerbehaftete Teile automatisch zu erkennen. Durch die komplexen Geometrien ist ein herkömmliches Segmentierungsverfahren überfordert, da die Unterscheidung zwischen gewollten und ungewollten Dellen sehr schwer ist.

In diesem Bereich können dazu Registrierungsverfahren eingesetzt werden, um so die Aufnahmen der einzelnen Bauteile mit einem defektlosen Referenzbild zu vergleichen. Dazu werden in der Regel verschiedene Referenzbilder herangezogen, um so unterschiedliche Beleuchtungsvariationen und Oberflächenbeschaffenheiten zu reduzieren. Durch die Registrierung werden Defekte in den produzierten Bauteilen in der automatisierten Qualitätskontrolle sichtbar.

7.4 Segmentierung

7.4.1 Problemstellung

Bei der Segmentierung werden inhaltlich zusammenhängende Regionen in einem Bild-datensatz durch unterschiedliche Arten von Verfahren voneinander abgegrenzt. Bei der Segmentierung unterscheidet man in der Regel die folgenden Arten:

- **Vollständig:** Jedes Pixel ist mindestens einem Segment zugeordnet.
- **Überdeckungsfrei:** Jedes Pixel ist höchstens einem Segment zugeordnet.
- **Vollständig und überdeckungsfrei:** Jedes Pixel ist genau einem Segment zugeordnet.
- **Zusammenhängend:** Jedes Segment ist ein zusammenhängendes Gebiet.

Von den segmentierten Gebieten können anschließend spezielle Kennwertgrößen bestimmt werden, beispielsweise die Fläche oder das Volumen des markierten Bereichs.

SEGMENTIERUNG

Gegeben: Bilddatensatz $f(x, y)$.

Gesucht: Menge der segmentierten Pixel.

Nach der Segmentierung können die bestimmten Segmente mit speziellen Analyseverfahren klassifiziert werden. Hierzu besteht die Möglichkeit verschiedene Arten von Parametern zu berechnen, mit denen die erzeugten Segmente charakterisiert werden. Einige einfache Beispiele segmentbeschreibender Parameter sind die folgenden:

- **Umfang:** Aufsummierung aller Pixel, deren 4 bzw. 8 Nachbarn nicht zum Segment gehören.
- **Flächeninhalt:** Aufsummierung der Pixel innerhalb jedes Segmentes.
- **Kompaktheit:** Quotienten aus dem Quadrat des Umfanges durch den Flächeninhalt.
- **Flächenschwerpunkt:** Aufsummierung aller x - bzw. y -Koordinaten geteilt durch den Flächeninhalt des Segmentes.
- **Löcher:** Anzahl der Löcher innerhalb eines Segmentes.
- **Kontur:** Ausgehend von einem Randpunkt als Startpunkt werden das Segment im Uhrzeigersinn entlang der Kontur durchlaufen und die Richtung (0 – Osten, 1 – Nordosten, 2 – Norden usw.) des jeweiligen nächsten Randpunktes gespeichert.

7.4.2 Grundlegende Lösungsprinzipien

Im Bereich der Segmentierung gibt es eine ganze Reihe unterschiedlicher Verfahren:

- **Pixelorientiert:** Verfahren entscheidet für jedes Pixel separat, ob es zu einem bestimmten Segment gehört oder nicht.
- **Kantenorientiert:** Verfahren sucht in einem Bild nach Kanten bzw. Objektübergängen.
- **Regionenorientiert:** Verfahren sucht im Bild nach zusammenhängenden Objekten.
- **Modellbasiert:** Verfahren basiert auf einem Modell einer bestimmten Form der Objekte.
- **Texturorientiert:** Verfahren betrachtet die innere Struktur der Objekte (z. B. Muster, Streifen).

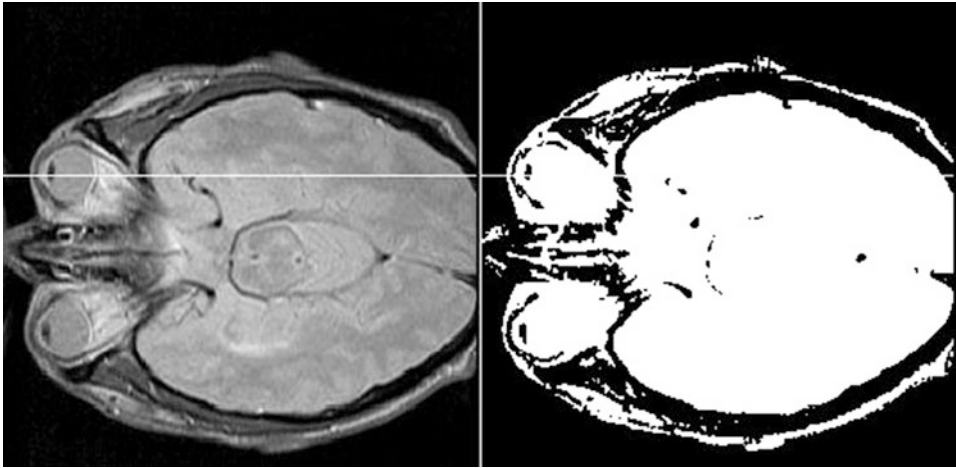


Abb. 7.17 Segmentierung mit dem Schwellwertverfahren, bei dem alle Grauwerte ab einem gewissen Intervall auf 1 gesetzt werden

Viele dieser Verfahrenstypen werden miteinander gekoppelt, um so je nach Anwendungsbereich optimale Ergebnisse zu erreichen. Wir stellen im Folgenden einige der zentralsten Verfahrenstypen vor.

Schwellwertverfahren Das Schwellwertverfahren ist ein pixelorientiertes Verfahren, bei dem zu einem Bildobjekt gehörende Pixel anhand zweier Schwellwerte a und b in dem Bild bzw. der 3-D-Bildfolge separiert und in einem Binärbild B markiert werden:

$$\tilde{f}(x, y) = \begin{cases} 1, & \text{falls } a \leq f(x, y) \leq b, \\ 0, & \text{sonst.} \end{cases}$$

Die Schwellwerte können dabei statisch, also unabhängig von der jeweiligen Aufnahme oder Bildregion, bzw. dynamisch für das vorhandene Bild angepasst sein. Anwendung findet das Schwellwertverfahren bei der Abgrenzung des Bildobjekts vom Bildhintergrund, dargestellt in Abb. 7.17. Ebenfalls haben wir das einfache Schwellwertverfahren bereits genutzt, um Kanten in einem mit einem Differenzenfilter bearbeiteten Bilddatensatz deutlicher hervorzuheben.

Bereichswachstumsverfahren Das Bereichswachstumsverfahren ist ein regionenorientiertes Verfahren, das in der Bildverarbeitung zur Segmentierung einzelner Bereiche eingesetzt wird, wobei die extrahierten Segmente eine zusammenhängende Bildregion bilden und bezüglich eines Bildmerkmals homogen sind. Die Idee des Bereichswachstumsverfahrens ist einfach: Zuerst wird ein Pixel des zu segmentierenden Objekts als Saatpunkt

ausgewählt und in die Menge der aktiven Pixel hinzugefügt. Jedes bereits untersuchte Pixel wird anschließend markiert, sodass im Laufe des Verfahrens kein Pixel zweimal untersucht wird. Falls das aktuelle Pixel ein gewisses Homogenitätskriterium erfüllt (z. B. Farbwertintervall), so wird es in die Menge der segmentierten Pixel aufgenommen. Anschließend wird die Menge der Nachbarpixel des Saatpunktes bestimmt, beispielsweise die 4- oder 8-Nachbarschaft (siehe Abb. 7.18). Alle Pixel der Nachbarschaft, die noch nicht markiert sind, werden in die Menge der aktiven Pixel hinzugenommen. Das ganze Verfahren wird so lange durchgeführt, bis kein Pixel mehr in dieser Menge vorhanden ist. In Abb. 7.19 wird dargestellt, wie das Segmentierungsergebnis von der gewählten Nachbarschaft (4-Nachbarschaft: dunkelgrau; 8-Nachbarschaft: hellgrau) abhängt.

Prinzip der Bereichssegmentierung

Für die Anwendung der Bereichssegmentierung werden die Nachbarschaft $N = N(p)$ eines Pixels p , ein Saatpunkt p und das gewünschte Homogenitätskriterium H definiert.

1. Speicherung des Saatpunktes p in einem Stack S
2. Solange der Stack S nicht leer ist:

Abb. 7.18 2-D-Nachbarschaft:
4 Nachbarn und 8 Nachbarn
eines Bildpunktes

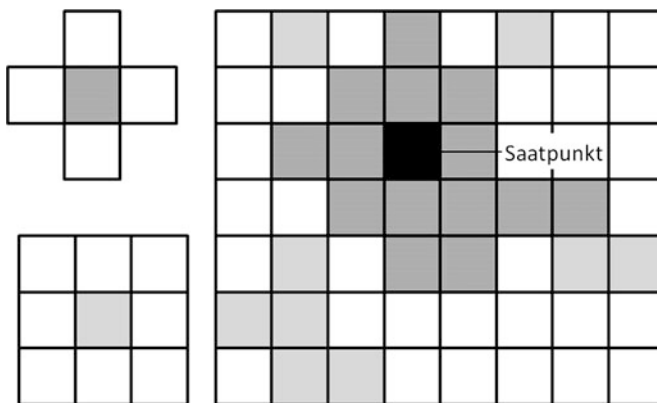
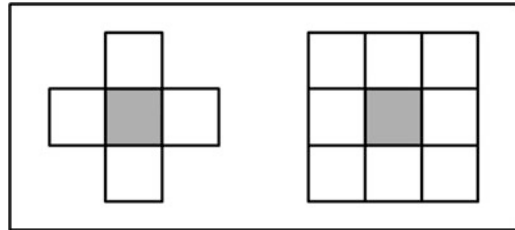


Abb. 7.19 Segmentierung mittels Nachbarschaftsbeziehung für 4-Nachbarschaft (*dunkelgrau*) und 8-Nachbarschaft (*hellgrau*)

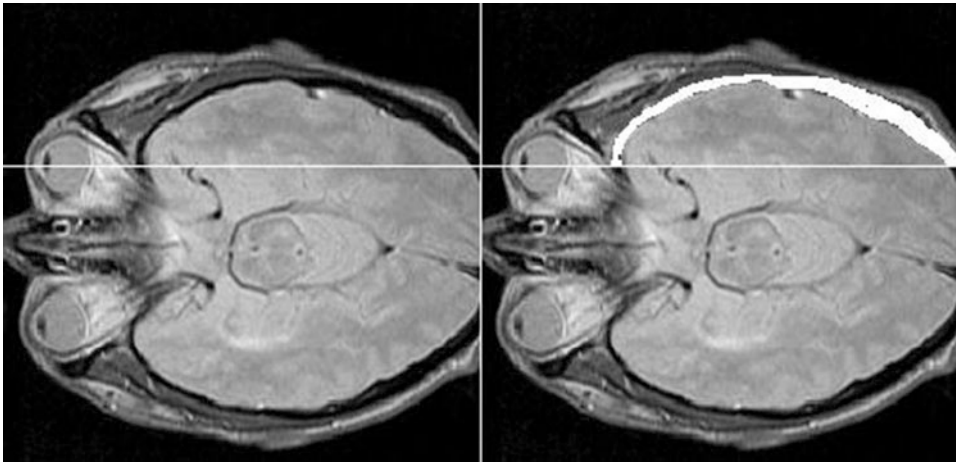


Abb. 7.20 Segmentierung mit dem Bereichswachstumsverfahren von einem Saatpunkt

- (a) Entnahme des oberen Pixels p aus dem Stack S
- (b) Falls das Pixel p noch nicht markiert ist:
 - Markierung des Pixels p
 - Prüfung ob p das Homogenitätskriterium H erfüllt.
Falls ja: Pixel p wird segmentiert
 - Hinzufügung in den Stack S der Menge aller Nachbarn $N(p)$, die noch nicht markiert sind

In Abb. 7.20 ist das Bereichswachstum für einen Saatpunkt angewandt, sodass alle Pixel mit einem ähnlichen Grauwert weiß markiert wurden.

Live-Wire-Verfahren Das Live-Wire-Verfahren ist ein bedeutendes Standardverfahren zur kantenorientierten Segmentierung einzelner Bildobjekte. Bei diesem Verfahren wird zwischen einem ausgewählten Konturpunkt und der aktuellen Mausposition automatisch eine Verbindung entlang der Bildkante berechnet. Bei Bewegungen der Maus wird die berechnete Verbindung der aktuellen Position angepasst.

Diese Art der Visualisierungstechnik hat dem Live-Wire-Verfahren seinen Namen gegeben, da sich durch schnelle Bewegungen der Maus die optimale Kontur zwischen dem Saatpunkt und der sich verändernden Mausposition ändert und die damit dargestellte Folge von optimalen Konturen wie ein scheinbar „lebender Draht“ erscheint.

Bei der Live-Wire-Segmentierung werden graphentheoretische Methoden und Algorithmen zur Bestimmung der gesuchten Konturabschnitte verwendet. Das Ergebnis ist ein Optimierungsproblem zur Berechnung einer kostenoptimalen Verbindung zwischen zwei Konturpunkten, die entlang der Objektkontur verlaufen soll.

KONTURFINDUNG

Gegeben: Bilddatensatz $f(x, y)$, $x \in \{0, \dots, N_x\}$, $y \in \{0, \dots, N_y\}$, Konturpunkte s, t .

Gesucht: kostenoptimale Verbindung zwischen s und t entlang der Objektkontur.

Wir konstruieren einen Graphen $G = (V, E)$ mit einer Kostenfunktion $c : E \rightarrow \mathbb{R}$:

- Knotenmenge des Graphen G : Menge aller Bildpunkte des Bilddatensatzes:

$$V = \{p = (x, y) \mid x \in \{0, \dots, N_x\}, y \in \{0, \dots, N_y\}\}.$$

- Kantenmenge des Graphen G : Zwei Knoten p und q sind durch eine Kante verbunden, wenn die zugehörigen Bildpunkte benachbart sind, d. h., der Knoten q liegt in der Menge $N_G(p)$ der 8-Nachbarschaft von p :

$$E = \{(p, q) \mid p, q \in V, p \in N_G(p)\}.$$

- Kostenfunktion des Graphen G : Die Kostenfunktion c muss so definiert werden, dass Pfade entlang von Objektkanten niedrige Kosten erhalten:

$$c(x, y) = 1 - \frac{|\text{grad} f(x, y)|}{\max_{x,y} |\text{grad} f(x, y)|} \in [0, 1].$$

Punkte der stärksten Kante erhalten dadurch den Kostenwert 0, während Punkte in homogenen Bereichen den Kostenwert 1 bekommen.

Die Aufgabe ist einen Pfad in Form einer Kontur mit minimalen Kosten zwischen den beiden Knoten s und t zu finden. Durch Überführung des Problems der Konturfindung in ein graphentheoretisches Optimierungsproblem kann dieses nun mit den bekannten Standardverfahren der Graphentheorie gelöst werden. Das bekannteste Verfahren zur Bestimmung eines kürzesten Pfades zwischen zwei ausgezeichneten Knoten ist der Algorithmus von Dijkstra.

Prinzip der Live-Wire-Segmentierung

Gegeben sind ein Bilddatensatz $f(x, y)$ und eine Menge aller Konturpunkte K .

1. Aufstellung des gewichteten Graphen $G = (V, E, c)$ aus dem Bilddatensatz $f(x, y)$
2. Für alle benachbarten Konturpunkte s und t aus der Menge K :
 - (a) Bestimmung des kürzesten Weges p zwischen s und t mithilfe des Dijkstra-Algorithmus
 - (b) Markierung des Pfades p zwischen den Bildpixeln s und t im zu segmentierenden Bild

In Abb. 7.21 ist die Live-Wire-Segmentierung für eine Menge von Punkten angewandt, sodass eine optimale Kontur gezeichnet wurde.

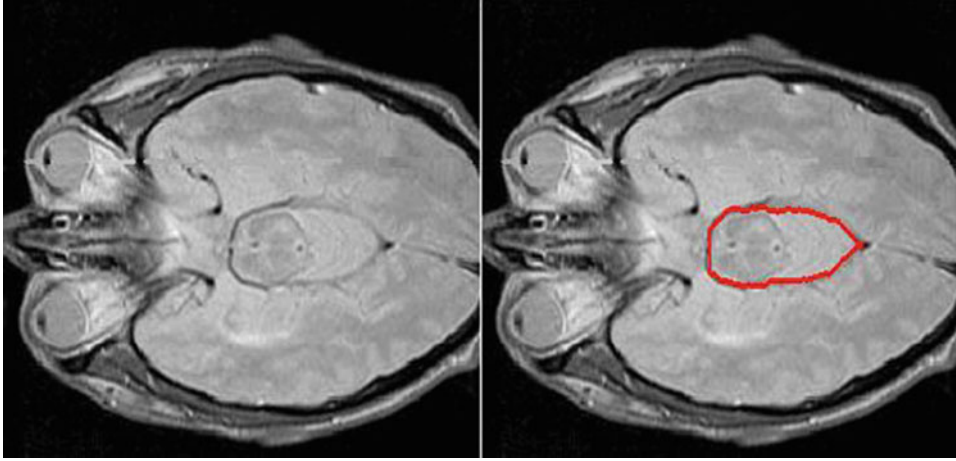


Abb. 7.21 Segmentierung mit dem Live-Wire-Verfahren in Form einer Kontur (rote Linie)

7.4.3 Algorithmus und Implementierung

Wir stellen hier den Pseudocode für das Bereichswachstumsverfahren vor. Hierzu verwenden wir das Homogenitätskriterium H und eine Methode NACHBARN zur Bestimmung der Menge aller Nachbarn eines Punktes p .

Algorithmus 19 BEREICHSWACHSTUMSVERFAHREN

Input: Bilddaten $f(x, y)$, Saatpunkt p

Output: Menge der segmentierten Pixel S_p

```

1: S.push( $p$ )
2: while  $|S| > 0$  do
3:   S.pop( $p$ )
4:   if !markiert( $p$ ) then
5:     markiert( $p$ ) = 1
6:     if  $p \in H$  then
7:        $S_p = S_p \cup \{p\}$ 
8:        $N_p = \text{NACHBARN}(p)$ 
9:       for  $p' \in N_p$  do
10:        if !markiert( $p'$ ) then
11:          S.push( $p'$ )

```

Allgemeine Erklärung

In Zeile 1 wird der leere Stack mit dem Saatpunkt p gefüllt. Die folgende Schleife wird so lange ausgeführt, bis der Stack leer ist. In Zeile 3 wird ein Pixel p entnommen. Falls dieses Pixel p nicht markiert ist (Zeile 4), wird es markiert (Zeile 5). In Zeile 6 wird geprüft, ob das Homogenitätskriterium erfüllt ist. Falls ja, wird das Pixel p segmentiert

(Zeile 7). Anschließend werden in Zeile 8 alle Nachbarpixel von p bestimmt. Falls diese Pixel nicht markiert sind, werden sie in den Stack S hinzugefügt (Zeile 9–11).

Für die Implementierung der verschiedenen Segmentierungsverfahren verwenden wir wieder das objektorientierte Entwurfsmuster der Strategie. Dazu definieren wir zunächst ein Interface `ISegmentierung` zur Darstellung eines allgemeinen Segmentierungsverfahrens:

```
public interface ISegmentierung
{
    public void segmentiere(BildSegmentierung f);
}
```

Dieses Interface dient als Schnittstelle für alle konkreten Segmentierungsverfahren durch Definition der Methode `segmentiere`, in der dann die jeweilige konkrete Strategie implementiert wird.

Wir definieren nun die Klasse `BildSegmentierung` des Kontextobjekts zur Speicherung der Bilddaten und der jeweiligen Strategie in der Variablen `ISegmentierung segment`. Diese Klasse ist ebenfalls wieder von der vorgestellten Klasse `Bild` abgeleitet. In der Methode `setSegmentierung` wird vom Anwender die gewünschte Strategie gesetzt. Die Methode `anwendenSegmentierung` wird vom Anwender aufgerufen, um den jeweiligen konkreten Algorithmus zu starten.

```
public class BildSegmentierung extends Bild
{
    ISegmentierung segment = null;

    public BildSegmentierung(int breite, int hoehe)
    {
        super(breite, hoehe);
    }
    public BildSegmentierung(String name)
    {
        super(name);
    }

    public void setSegmentierung(ISegmentierung segment)
    {
        this.segment = segment;
    }

    public void anwendenSegmentierung()
    {
        segment.segmentiere(this);
        this.anzeigen();
    }
}
```

Wir zeigen die Implementierung des Bereichswachstums mit der gleichnamigen Klasse. Die Klasse `Bereichswachstum` enthält die Instanzvariablen `saat` für den Saatpunkt sowie die Variablen `unten` und `oben` zur Definition des Homogenitätskriteriums in Form eines Farbintervalles mit der Methode `schwelle`.

```
public class Bereichswachstum implements ISegmentierung
{
    private int saat[];
    private int unten;
    private int oben;

    public Bereichswachstum(int p[], int u, int o)
    {
        saat = p;
        unten = u;
        oben = o;
    }

    public boolean schwelle(int farbe)
    {
        if (farbe >= unten && farbe <= oben)
            return true;
        else
            return false;
    }

    public void segmentiere(BildSegmentierung bild)
    {
        int breite = bild.getBreite()/2;
        int hoehe = bild.getHoehe();
        // --- 1. Bild kopieren auf rechten Bereich
        for (int x = breite+1; x < 2*breite; x++)
            for (int y = 0; y < hoehe; y++)
                bild.set(x, y, bild.get(x-breite, y));

        // --- 2. Initialisierung
        boolean markiert[][] = new boolean[breite][hoehe];
        Stack<int[]> aktiv = new Stack<int[]>();
        aktiv.push(saat);

        // --- 3. Wachstumsverfahren
        while(aktiv.size() > 0)
        {
            // --- 3a. Oberstes Pixel entnehmen
            int pixel[] = aktiv.pop();
            int x = pixel[0];
            int y = pixel[1];
            if(!markiert[x][y]) // nicht markiert
            {
                markiert[x][y] = true; // Pixel markieren
                // --- 3b. Prüfung ob Homogenitätskriterium erfüllt
                if(schwelle(bild.get(x, y).getRed()))
                {

```

```

        // --- 3c. Setzung der Farbmarkierung
        bild.set(x+breite, y, new Color(255, 255, 255));
        // --- 3d. Prüfung aller Nachbarn
        Vector<int[]> nachbarn = achtNachbarn(x, y, breite, hoehe);
        for(int i=0; i<nachbarn.size(); i++)
        {
            int p[] = nachbarn.elementAt(i);
            if(!markiert[p[0]][p[1]])
                aktiv.push(p);
        }
    }
}

public Vector<int[]> achtNachbarn(int x, int y, int breite, int hoehe)
{
    Vector<int[]> nachbarn = new Vector<int[]>();
    if(x > 0 && y > 0)
        nachbarn.add(new int[]{x-1, y-1});
    if(y > 0)
        nachbarn.add(new int[]{x, y-1});
    if(x < breite-1 && y > 0)
        nachbarn.add(new int[]{x+1, y-1});
    if(x > 0)
        nachbarn.add(new int[]{x-1, y});
    if(x < breite -1)
        nachbarn.add(new int[]{x+1, y});
    if(x > 0 && y < hoehe-1)
        nachbarn.add(new int[]{x-1, y+1});
    if(y < hoehe-1)
        nachbarn.add(new int[]{x, y+1});
    if(x < breite-1 && y < hoehe-1)
        nachbarn.add(new int[]{x+1, y+1});
    return nachbarn;
}
}

```

Der Anwender verwendet die vorgestellten Klassen und erzeugt eine Instanz der Klasse `BildSegmentierung`. Das Setzen des Segmentierungsverfahrens erfolgt mit der Methode `setSegmentierung`.

```

public class TestSegmentierung
{
    public static void main(String args[])
    {
        BildSegmentierung bild = new BildSegmentierung("MR.jpg");
        bild.setSegmentierung(new Bereichswachstum(new int[]{237, 73}, 0, 40));
        bild.anwendenSegmentierung();
    }
}

```

7.4.4 Anwendungen

Wir stellen einige Anwendungsbereiche von Segmentierungsverfahren vor.

Medizinische Bildverarbeitung In der medizinischen Bildverarbeitung wird die Segmentierung für die rechnergestützte ärztliche Diagnostik und Therapie eingesetzt. Die erzielten Ergebnisse sind dann die Grundlage für eine weitere Analyse, Vermessung oder Visualisierung medizinischer Bildobjekte. Beispielsweise muss zunächst ein Tumor segmentiert werden, bevor eine Analyse seiner Eigenschaften (z. B. Form, Volumen) vorgenommen wird. Das Ziel der Segmentierung medizinischer Bilder ist die Abgrenzung von medizinisch relevanten Bildobjekten wie Gewebe, Tumoren, Gefäßsystemen von gesunden Strukturen.

Optische Qualitätskontrolle Segmentierungsverfahren werden häufig im Bereich der automatischen optischen Qualitätskontrolle von Werkstücken verwendet. Anwendungsbeispiele sind die Überprüfung, ob eine Schweißnaht korrekt ist oder ob sich ein Bohrloch an der richtigen Position im Werkstück befindet. Ebenso können Segmentierungsverfahren genutzt werden, um auf einem Fließband verschiedene Arten von Objekten zu erkennen und diese anschließend in die jeweilige Klasse einzuordnen. Damit können diese Objekte nicht nur auf optische Qualitätsmerkmale untersucht werden, sondern anschließend auch über einen automatisierten Prozess weiterverarbeitet bzw. verpackt werden.

Mensch-Maschine-Kommunikation Die Mensch-Maschine-Kommunikation beschäftigt sich mit der benutzergerechten Gestaltung von interaktiven Mensch-Maschine-Systemen. Durch die fortschreitende Digitalisierung und die Entwicklung von mobilen Geräten gewinnt die Mensch-Maschine-Kommunikation in der Praxis eine immer größere Bedeutung. Zwei zentrale Teilgebiete der Mensch-Maschine-Kommunikation sind die Schrift- und Gesichtserkennung, bei denen Segmentierungsverfahren zum Einsatz kommen. Bei der Schrifterkennung wird im gescannten Bild durch Segmentierung die Schrift vom Hintergrund getrennt. Im Bereich der Gesichtserkennung beschäftigt man sich mit der Lokalisierung eines Gesichts und der Zuordnung des Gesichts zu einer bestimmten Person. Bei einfachen Gesichtserkennungsverfahren erfolgt eine geometrische Vermessung von Augen, Nase und Mund. Ein sehr populäres Verfahren zur Gesichtserkennung ist die sogenannte Viola-Jones-Methode, die auch zu Erkennung von anderen Mustern in digitalen Bildern verwendbar ist. Dieser Algorithmus basiert auf dem maschinellen Lernen mithilfe von Trainingsbeispielen und der AdaBoost-Methode⁴.

⁴ Weitere Information im Band *Intelligente Algorithmen*.

7.5 Spezielle Anwendung

Eine Krebserkrankung ist heute die zweithäufigste Todesursache, an der jeder zweite Mensch im Laufe des Lebens erkrankt. In diesem Anwendungsbeispiel betrachten wir eine Studie vom Max-Planck-Institut für Intelligente Systeme in Stuttgart über die Krebsdiagnose mithilfe der fraktalen Dimension.⁵ In dieser Forschungsarbeit werden Zellen mit einem Reflexionsinterferenzkontrastmikroskop mit einer Auflösung von 1 nm aufgenommen. Dadurch sind die Konturen der Zelle als sogenannte Fraktale zu erkennen, also spezielle geometrische Muster, die wiederholt in immer kleiner werdenden Größenordnungen auftreten.

Die fraktale Dimension kennzeichnet dabei die Zunahme der Strukturdetails mit zunehmender Vergrößerung. Diese Kenngröße wird durch eine automatisierte Bilderkennung der Zellbilder bestimmt und ausgewertet. Das Ergebnis ist, dass 97 % der bösartigen Zellen richtig diagnostiziert werden – signifikant mehr als bei den herkömmlichen Verfahren, die zudem noch teure Markermoleküle verwenden.

Unter einem Tumor (lat. Wucherung, Geschwulst) versteht man im Allgemeinen die Volumenzunahme eines Körpergewebes. Im Speziellen werden dadurch Zellen beschrieben, die sich durch eine Fehlregulation unkontrolliert vermehren. Krebszellen entstehen durch Abweichung vom Gleichgewichtszustand durch unkontrollierte und ungehemmte Zellteilung. Das Ergebnis dieses Prozesses spiegelt sich in dem Aussehen der Zellen wider. Man unterscheidet im Groben zwischen den gutartigen (PatuS-Zellen) und den bösartigen Tumoren (PatuT-Zellen), dargestellt in Abb. 7.22. Bei gutartigen Tumoren wachsen die Zellen langsam, und es bilden sich keine Tochtergeschwülste (Metastasen). Die bösartigen Tumore besitzen ein deutlich schnelleres Wachstum mit einer hohen Zellteilungsrate. Diese Tumore bilden Metastasen und zerstören dabei gesundes Gewebe und Organe. Die Erkennung von Tumorzellen erfolgt mit bildgebenden Verfahren und mithilfe von Biomarkern, die jedoch teilweise eine höhere Anzahl von falschen Diagnosen liefern.

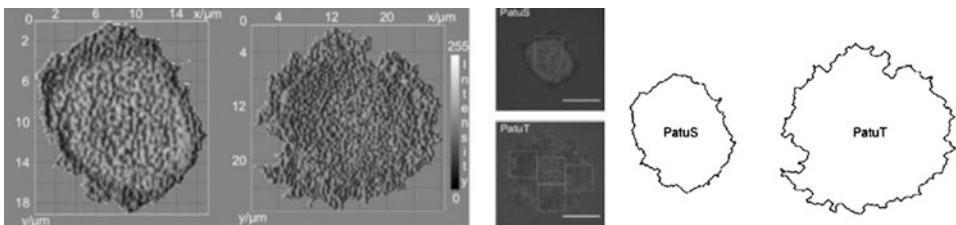


Abb. 7.22 Zellkonturen eines Tumors mit Reflexionsinterferenzkontrastmikroskop (RICM), gutartige PatuS- und bösartige PatuT-Zellen (Quelle: <http://pubs.acs.org/doi/pdf/10.1021/nl4030402>)

⁵ K. Klein, u. a. „Marker-Free Phenotyping of Tumor Cells by Fractal Analysis of Reflection Interference Contrast Microscopy Images“, NanoLetter: S. 5474–5479, 2013.

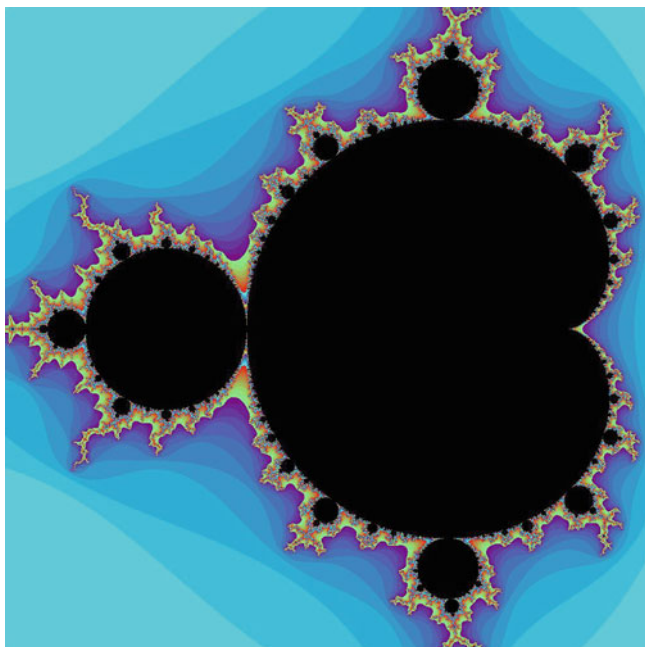


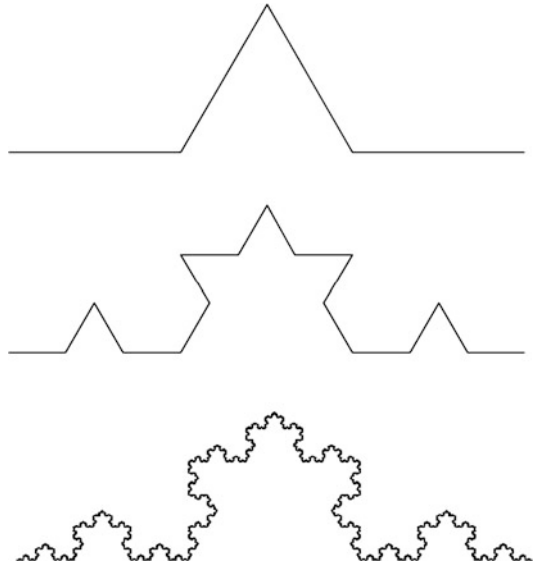
Abb. 7.23 Darstellung der Mandelbrot-Menge

Als Fraktale werden natürliche, künstliche Gebilde oder geometrische Muster bezeichnet. Eines der bekanntesten Fraktale ist die Mandelbrot-Menge (siehe Abb. 7.23), benannt nach Benoit Mandelbrot. Diese Objekte haben die Eigenschaft, einen hohen Grad von Selbstähnlichkeit zu haben, d. h., das dargestellte Objekt besteht aus mehreren verkleinerten Kopien seiner selbst.

In der medizinischen Bildverarbeitung können mithilfe von fraktalen Verfahren fein strukturierte biologische Objekte oder unregelmäßige chaotische Muster, wie sie bei Geweben oder Tumoren auftreten, klassifiziert werden. Fraktale spielen durch ihren Aufbau bei der computergestützten Simulation facettenreicher Strukturen wie beispielsweise realitätsnaher Landschaften eine große Rolle. Fraktale Erscheinungsformen finden auch in den Naturwissenschaften breite Anwendung, beispielsweise bei der Darstellung von Farnen oder anderen Pflanzen, Beschreibung von chemischen Reaktionen oder bei der Modellierung des Kristallwachstums.

Das bekannteste und einfachste Beispiel eines Fraktals ist die Koch-Kurve. Die Koch-Kurve wird nach den folgenden Konstruktionsregeln gezeichnet: Zuerst wird eine einfache Linie gezeichnet. Dann wird diese Linie gedrittelt und das Mittelstück durch ein gleichseitiges Dreieck ohne Grundlinie ersetzt. Das Ergebnis ist eine Koch-Kurve der Ordnung 1 mit 4 gleich langen Teilstrecken. An jeder Teilstrecke wiederholen wir den beschriebenen Vorgang und man erhält Koch-Kurven höherer Ordnung (siehe Abb. 7.24).

Abb. 7.24 Koch-Kurven der Ordnungen 1, 2 und 6



Definition 7.1 Die *fraktale Dimension* einer rekursiv fraktalen Menge berechnet sich aus der Anzahl N ihrer wiederholenden Kopien, die um einen Faktor s verkleinert werden:

$$D = \frac{\log N}{\log \frac{1}{s}}.$$

Für die Koch-Kurve wird die Strecke jeweils durch 3 geteilt ($s = 3$), und es entstehen $N = 4$ Teilstrecken, also $d = 1,26$.

Da der ideale Fall einer Koch-Kurve bei natürlichen Fraktalen nicht vorkommt, verwendet man die sogenannte *fraktale Kästchendimension* bzw. *Boxcounting-Methode*.

Definition 7.2 Ein fraktales Muster wird mit quadratischen Gittern der Maschenweite ε überdeckt. Die *fraktale Kästchendimension* berechnet sich aus der Anzahl $N(\varepsilon)$ der durch das Muster belegten Gitterkästchen in Abhängigkeit der Maschenweite ε :

$$D = \lim_{\varepsilon \rightarrow 0} \frac{\log N(\varepsilon)}{\log \frac{1}{\varepsilon}}.$$

Zum Auswerten wird die Maschenweite ε gegenüber der Anzahl $N(\varepsilon)$ in einem doppelt logarithmischen Koordinatensystem dargestellt. Das Ergebnis ist eine Gerade mit der Steigung m (siehe Abb. 7.25).

Für die Analyse von Krebszellen wird die fraktale Dimension von gutartigen PatuS-Zellen und von bösartigen PatuT-Zellen durchgeführt. Das Ergebnis ist eine signifikant höhere ($p < 0,001$) fraktale Dimension bei PatuT-Zellen ($1,218 \pm 0,005$) als bei PatuS-Zellen ($1,171 \pm 0,004$).

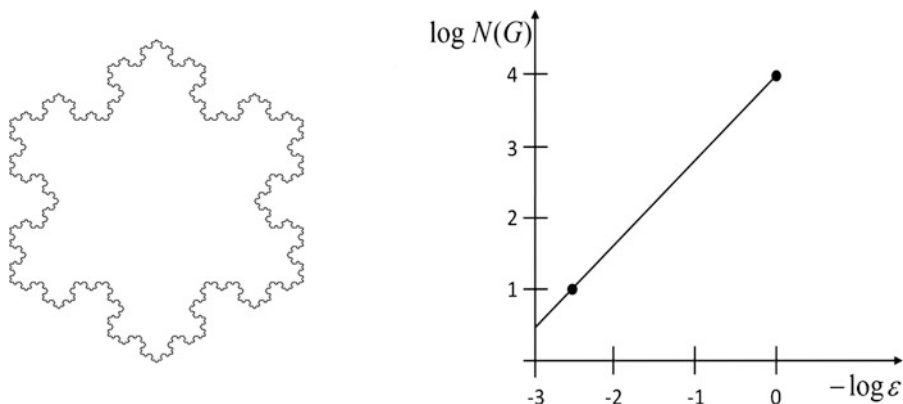


Abb. 7.25 Boxcounting-Methode angewandt auf Koch'sche Schneeflocke

Für eine korrekte klinische Diagnose ist eine niedrige Falsch-negativ-Rate sehr wichtig, da bösartige Tumore auf jeden Fall behandelt werden müssen. Aus diesem Grund wurde eine zweistufige Analyse durchgeführt: Im ersten Schritt werden die Zellen nach dem besten Schwellenparameter (hier 1,19) der fraktalen Dimension der Zellkonturen klassifiziert. Damit werden PatuT-Zellen mit einer Falsch-negativ-Rate von 30 % und einer Richtig-positiv-Rate von 70 % klassifiziert. In einem zweiten Schritt werden die PatuS-Zellen mit dem Schwellenwert der kombinierten fraktalen Dimension (hier 1,33) noch einmal klassifiziert. Dadurch wird die Falsch-negativ-Rate auf 3 % gesenkt und 97 % der PatuT-Zellen werden richtig klassifiziert. Mit den beiden kombinierten Verfahren ist eine bessere Diagnose möglich als bei häufig verwendeten Tumormarkern, die eine Sensitivität von ca. 50–85 % besitzen.

Prinzip der fraktalen Krebsdiagnose

Gegeben ist ein Grauwertbild einer Krebszelle. Gesucht ist die Klassifizierung in PatuS- und PatuT-Zellen.

1. Erzeugung des Gradientenbildes durch Anwendung eines Kantenfilters (z. B. Sobel-Filter)
2. Berechnung des Kantenbildes durch Herausfilterung der größten Intensitätsänderungen
3. Anwendung der Boxcounting-Methode auf das Kantenbild mit einer Maschenweite ε zur Bestimmung der fraktalen Dimension $D(\varepsilon)$. Am Anfang entspricht ε der Anzahl der Pixel des Bildes:
Solange $\varepsilon > 1$:
 - (a) Zählen der Anzahl $N(\varepsilon)$ der durch die Kantenpixel belegten Gitterkästchen
 - (b) Halbierung der Maschenweite ε

4. Bestimmung des Anstiegs m der Regressionsgeraden aus der Punktmenge $P_\varepsilon = (-\log \varepsilon, \log N(\varepsilon))$. Der Anstieg m der Gerade entspricht der fraktalen Dimension D
5. Klassifizierung der fraktalen Dimension D bezüglich eines Schwellwertparameters t_S :
 Falls $D < t_S \Rightarrow$ PatuS-Zelle
 Falls $D \geq t_S \Rightarrow$ PatuT-Zelle

7.6 Übungsaufgaben

Aufgabe 7.1 (Vorverarbeitung) Erweitern Sie die Klasse `Bild` für einen Bilddatensatz mit den folgenden Methoden:

- `berechneLumWert`: Berechnung des Luminanzwertes;
- `umwandelnGraubild`: Umwandlung eines Farbbildes in ein Graubild;
- `pruefeKompatibilitaet`: Prüfung der Hintergrund-Vordergrund-Kompatibilität;
- `skalieren`: Skalierung eines Bildes auf die Größe $w \times h$.

Aufgabe 7.2 (Umwandlung der Größe) Erweitern Sie die Klasse `Bild` um eine Methode zur Umwandlung eines Eingabebildes in N Schritten in ein Zielbild gleicher Größe. Um ein Bild in ein anderes umzuwandeln muss im t -ten Bild jedes Pixel auf ein gewichtetes Mittel des entsprechenden Pixels im Quell- und Zielbild gesetzt werden, wobei die Quelle mit $1 - t/N$ und das Ziel mit t/M gewichtet werden.

Aufgabe 7.3 (Lineare Skalierung) Implementieren Sie in die Klasse `Skalierung` eine lineare Skalierung der folgenden Form, um eine Grauwertverteilung g eines Originalbildes mit den Konstanten c_1 und c_2 zu transformieren:

$$f(g) = \begin{cases} 0, & c_2g + c_1c_2 \leq 0, \\ 255, & c_2g + c_1c_2 > 255, \\ c_2g + c_1c_2, & \text{sonst.} \end{cases}$$

Wegen $f(g) = c_2(g + c_1)$ wird für $c_1 > 0$ das Bild heller und für $c_1 < 0$ das Bild dunkler. Wenn keine Bildinformation verloren gehen soll, bestimmen sich die beiden Konstanten wie folgt:

$$c_1 = -\min \quad \text{und} \quad c_2 = \frac{255}{\max - \min},$$

wobei $\min$ und $\max$ der minimale und maximale Grauwert des Bildes sind. Erstellen Sie anschließend für beide Bilder ein Grauerthistogramm, und beobachten Sie die Resultate für verschiedene Transformationen.

Aufgabe 7.4 (Logarithmische Skalierung) In der linearen Skalierung werden alle Grauwerte gleichmäßig verändert. Für unterschiedliche Anpassungen von Kontrastwerten wird eine logarithmische Skalierung verwendet. Dazu verwendet man die drei Parameter c_1, c_2 und c_3 mit $0 < c_1 < c_2$ und $-255 < c_3 < 255$:

$$\begin{aligned} \min &= \log(c_1), \quad \max = \log(c_2), \quad e = (c_2 - c_1)/255, \\ f(g) &= 255 \cdot \frac{\log(c_1 + g \cdot e) - \min}{\max - \min}. \end{aligned}$$

Eine exponentielle Skalierung erhält man durch Ersetzung des Logarithmus durch die Exponentialfunktion. Ergänzen Sie in der obigen Klasse `Skalierung` zwei Methoden zur logarithmischen und linearen Skalierung von Grauwerten.

Aufgabe 7.5 (Äquidensiten) Eine spezielle Form einer Grauwertskalierung ist die Erzeugung eines Äquidensitenbildes. Diese Skalierungsfunktion f ist für einen Grauwert g über ein stückweises Intervall definiert:

$$f(g) = g_k, \quad l_k < g < l_{k+1}, \quad k = 1, \dots, m,$$

wobei l_k und l_{k+1} aus dem Intervall $\{0, 1, \dots, 255\}$ sind. Hierbei ist es natürlich auch möglich, bestimmte Grauwertbereiche unverändert zu lassen. Erstellen Sie eine Klasse zur Erzeugung beliebiger Äquidensiten.

Aufgabe 7.6 (Elimination gestörter Bildpunkte) Gestörte oder isolierte Bildpunkte treten bei gewissen Anwendungen auf, wenn störende Einflüsse auf den Sensor wirken. Bei einer Entfernung dieser gestörten Bildpunkte durch eine Filterung mit einem Glättungsoperator wird das Bild jedoch unschärfer, bzw. die Grauwerte in der Umgebung der Störung werden ebenso geändert. Eine einfache Variante gestörte Bildpunkte zu eliminieren, ohne andere Grauwerte zu verändern, ist der Vergleich jedes Bildpunktes mit dem Mittelwert m seiner 8 Nachbarn. Falls die Differenz dieser Werte größer als ein bestimmter Schwellwert c ist, wird der Bildpunkt als gestört detektiert und durch den Mittelwert m ersetzt. Erstellen Sie eine Klasse `EliminationPunkte` mit verschiedenen Varianten zur Entfernung gestörter Bildpunkte.

Aufgabe 7.7 (Elimination gestörter Bildzeilen) In einigen Bildern tauchen gestörte Bildzeilen auf, die entweder ganz weiß bzw. schwarz sind oder ein zufälliges Grauwertmuster aufweisen. Die Elimination gestörter Zeilen erfolgt mithilfe der Berechnung der Korrelation von aufeinanderfolgenden Bildzeilen. Bei ungestörten Bildzeilen ist der Übergang der Grauwerte von einer Bildzeile zur nächsten Bildzeile allmählich, d. h., der Kor-

relationskoeffizient ist nahe eins. Implementieren Sie eine Methode zur Elimination gestörter Zeilen durch folgende Vorschrift: Gehen Sie iterativ Bildzeile für Bildzeile des Eingabebildes $f = f(x, y)$ durch, und berechnen Sie die Kovarianz zwischen der x -ten und $(x + 1)$ -ten Zeile:

$$v_{x,x+1} = \frac{1}{n} \sum_{y=0}^n (f(x, y) - m_x)(f(x + 1, y) - m_{x+1}),$$

wobei m_x und m_{x+1} die mittleren Grauwerte der Bildzeilen x und $x + 1$ sind. Durch die Normierung der Kovarianz mit den Streuungen $v_{x,x}$ und $v_{x+1,x+1}$ erhält man den folgenden Korrelationskoeffizienten:

$$r_{x,x+1} = \frac{v_{x,x+1}}{\sqrt{v_{x,x} \cdot v_{x+1,x+1}}}.$$

Der Betrag des Korrelationskoeffizienten $|r_{x,x+1}|$ liegt zwischen 0 (keine Korrelation) und 1 (starke Korrelation). Mithilfe eines geeignet gewählten Schwellwertes c kann nun darüber entschieden werden, ob eine Zeile entfernt wird. Das ist der Fall, wenn der Betrag des Korrelationskoeffizienten $|r_{x,x+1}|$ kleiner als c ist.

Erstellen Sie eine Klasse `EliminationZeilen` zur Entfernung gestörter Bildzeilen.

Aufgabe 7.8 (Extraktion des Randes) Eine einfache Methode zur Extraktion des Randes aus einem Binärbild erfolgt über eine Dilatation und eine anschließende Differenzbildung des entstehenden Bildes mit dem Original. Bei der Dilatation eines Bildes wird der binäre Wert eines Pixels durch das Maximum der 8-Nachbarschaft mit dem aktuellen Pixelwert gesetzt.

Erstellen Sie eine Klasse `Extraktion` zur Extraktion eines Randes mit der Dilatation. Erweitern Sie anschließend diese Klasse um eine zweite Methodik zur Extraktion eines Randes mithilfe des beschriebenen Differenzenfilters in Kombination mit dem Schwellwertverfahren.

Aufgabe 7.9 (Canny-Kantendetektor) Der Canny-Algorithmus ist ein Algorithmus zur Kantendetektion, der aus verschiedenen Faltungsoperationen ein Bild berechnet, das im Idealfall nur noch die Kanten des Ausgangsbildes enthält. Der Algorithmus besteht aus den folgenden Schritten:

1. Umwandlung eines Farbbildes in ein Graubild f .
2. Filterung des Graubildes mit einem Glättungsfilter (z. B. Gauß-Filter).
3. Anwendung des Sobel-Operators in x - und y -Richtung liefert die beiden Bilder $f_x(x, y)$ und $f_y(x, y)$.
4. Bestimmung der Kantenrichtung aus den partiellen Ableitungen

$$\varphi(x, y) = \arctan \frac{f_y(x, y)}{f_x(x, y)},$$

wobei die Kantenrichtungen gerundet auf 0° , 45° , 90° und 135° angegeben werden, da ein Pixel nur 8 Nachbarn besitzt.

5. Berechnung des Bildes der absoluten Kantenstärke aus dem Betrag des Gradienten

$$\text{grad } f(x, y) = \sqrt{f_x(x, y)^2 + f_y(x, y)^2}$$

bzw. als Approximation

$$\text{grad } f(x, y) = |f_x(x, y)| + |f_y(x, y)|.$$

6. Anwendung des Non-maximum-suppression-Schrittes, bei dem für jedes Pixel im Kantenstärkebild in Abhängigkeit der Kantenrichtung die beiden Pixel rechts und links neben der Kante verglichen werden. Ist einer der Werte größer, wird der jeweilige Grauwert auf null gesetzt. Damit bleiben nur die Maxima entlang einer Kante erhalten, sodass eine Kante nicht mehr als ein Pixel breit ist.
7. Durchführung einer sogenannten Hysterese, bei der mit zwei Schwellwerten $c_1 < c_2$ das Bild durchlaufen wird, bis ein Pixel gefunden wird, dessen Stärke größer als c_2 ist. Dieser Kante wird dann beidseitig gefolgt, sodass alle Pixel entlang dieser Kante, deren Wert größer als c_1 ist, als Kantenelement markiert werden.

Implementieren Sie den Canny-Algorithmus und testen Sie ihn an geeigneten Beispielen.

Aufgabe 7.10 (Linienfinder) Ein sogenannter Onlinelinienfinder verarbeitet die eingegebenen Daten online, d. h., dass für die Berechnungen nur die Vorgängerdaten bekannt sein müssen. Die Punkte werden in diesem Fall so lange zu einer Linie addiert, wie die Abweichung des Punktes zu der durch die Vorgänger definierten Linie einen gewissen Schwellwert nicht überschreitet. Wenn dies nicht der Fall ist, wird die aktuelle Linie beendet, und der aktuelle Punkt ist der Beginn einer neuen Linie. Sei $(P_j, \dots, P_k)$ die bisher konstruierte Linie mit den zugehörigen Punkten. Ein neuer Punkt P_{k+1} wird dann zu einer Linie hinzugefügt, falls die folgenden Bedingungen erfüllt sind:

1. Globale Abweichung: Durch den neuen Punkt P_{k+1} weicht die globale Streckensumme nur wenig von der Luftlinie ab:

$$1 \geq \frac{|P_j, P_{k+1}|}{\sum_{i=j}^k |P_i, P_{i+1}|} > 1 - \varepsilon(k),$$

wobei der Parameter $\varepsilon(k)$ mit zunehmendem k größer wird.

2. Lokale Abweichung: Durch den neuen Punkt P_{k+1} weicht die lokale Streckensumme nur wenig von der Luftlinie ab:

$$1 \geq \frac{|a_{k-1}, a_{k+1}|}{|a_{k-1}, a_k| + |a_k, a_{k+1}|} > 1 - \varepsilon.$$

3. Der Abstand zwischen P_k und P_{k+1} sollte einen gewissen Wert nicht überschreiten.

Implementieren Sie den Onlinelinienfinder und testen Sie ihn an geeigneten Beispielen.

Aufgabe 7.11 (Run-Length-Codierung) Im Run-Length-Code einer Zeile eines Binärbildes werden die Positionen und die Längen der 1-Ketten codiert. Der Code (y, l) besagt, dass in der betrachteten Zeile ab der Spaltenposition y eine 1-Kette der Länge l beginnt. Beispielsweise ergibt die folgende Zeile

000011000111100

den Run-Length-Code $(4, 2), (9, 4)$. Implementieren Sie ein Verfahren zur Bestimmung der Run-Length-Codierung.

Aufgabe 7.12 (Filterung) Erweitern Sie die Klasse `Filter` um alle weiteren Filtertypen, die im Abschnitt Vorverarbeitung vorgestellt worden sind. Implementieren Sie hierzu auch die nicht linearen Filtertypen wie den Minimum-, Maximum- und Medianfilter.

Aufgabe 7.13 (Diskrete Fourier-Transformation) Schreiben Sie eine Klasse `Fourier2D` zur Implementierung der 2-D-Fourier-Transformierten zum Filtern von Bilddaten. Definieren Sie anschließend mithilfe von objektorientierten Entwurfsmustern geeignete Klassenstrukturen für die Anwendung von verschiedenen Hoch- und Tiefpassfiltern.

Aufgabe 7.14 (Registrierung) Implementieren Sie den vorgestellten Algorithmus zur landmarkenbasierten Registrierung in Form eines objektorientierten Strategiemusters. Erweitern Sie anschließend dieses Programmpaket um weitere Registrierungsverfahren, beispielsweise um die starre Transformation.

Aufgabe 7.15 (Segmentierung) Erweitern Sie unter Benutzung des Interface `ISegmentierung` und der Klasse `BildSegmentierung` die vorgestellte objektorientierte Implementierung von verschiedenen Segmentierungsstrategien um das Schwellwertverfahren und das Live-Wire-Verfahren.

Aufgabe 7.16 (Segmentierung) Erweitern Sie die Klasse `Bereichswachstum` um verschiedene Arten von Homogenitätskriterien. Implementieren Sie diese Kriterien unter Verwendung eines objektorientierten Schablonenmusters.

Aufgabe 7.17 (Segmentierung) Implementieren Sie in einer Klasse die vorgestellten segmentbeschreibenden Parameter.

Literaturhinweise⁶

1. Nischwitz, A., Fischer, M., Haberäcker, P., Socher, G. (2011). *Computergrafik und Bildverarbeitung*. Springer.
2. Süße, H., Rodner, E. (2014). *Bildverarbeitung und Objekterkennung*. Springer.
3. Burger, W., Burge, M.J. (2015). *Digitale Bildverarbeitung*. Springer Vieweg.
4. Sedgewick, R., Wayne, K. (2011). *Einführung in die Programmierung mit Java*. Pearson.
5. Handels, H. (2009). *Medizinische Bildverarbeitung*. Vieweg&Teubner.
6. Winner, H., Hakuli, S., Lotz, F., Singer, C. (2015). *Handbuch Fahrerassistenzsysteme*. Springer Vieweg.
7. Klawonn, F. (2010). *Grundkurs Computergrafik mit Java*. Vieweg&Teubner.

⁶ Die Bildverarbeitung ist ein sehr umfangreiches Themengebiet mit vielen weiteren algorithmischen Verfahren. Weiterführende Informationen zu diesem Thema findet der Leser in den Lehrbüchern zur Bildverarbeitung [1] und [2]. Das Thema Bildverarbeitung mit Java und ImageJ wird umfassend im Lehrbuch [3] behandelt. Anwendungsorientierte und einführende Beispiele zur Bildverarbeitung findet der Leser auch in [4]. Einen umfassenden Einblick in die medizinische Bildverarbeitung findet man in [5]. Den Überblick über Bildverarbeitungsverfahren im Bereich von Fahrerassistenzsystemen und beim autonomen Fahren bietet das Handbuch [6]. Leser mit Interesse im Bereich der Computergrafik und deren Umsetzung mit Java finden in [7] einen guten Einstieg.

Anhang

A1: API-Dokumentation

Überblick über Pakete

Die zentrale Informationsquelle für Programmierer ist die offizielle API-Dokumentation von Oracle: <http://docs.oracle.com/javase/8/docs/api/>.

In Eclipse kann man mit den Tasten Shift + F2 die eingebettete API-Dokumentation ansehen.

Die aktuelle Java-Klassenbibliothek besitzt mehr als 200 verschiedene Pakete, wovon die wichtigsten die folgenden sind:

Paket	Beschreibung
java.awt	Ausgabe von Grafiken, Erstellung von grafischen Bedienoberflächen
java.awt.event	Behandlung von Ereignissen unter grafischen Oberflächen
java.io	Ein- und Ausgabe für Zugriff auf Dateien
java.lang	Automatisch eingebunden für String-, Thread- und Wrapper-Klassen
java.net	Kommunikation über Netzwerke
java.text	Behandlung von Text, Formatierung von Datumswerten und Zahlen
java.util	Datenstrukturen, Zufallszahlen, Raum und Zeit
javax.swing	Swing-Komponenten für grafische Oberflächen

Ausgewählte Pakete

java.io

Klassen	Beschreibung
BufferedInputStream	Gepufferter Eingabestream
BufferedOutputStream	Gepufferter Ausgabestream
BufferedReader	Gepuffertes Einlesen von Text aus einem Stream
BufferedWriter	Gepuffertes Schreiben von Text in einen Stream
Console	Schreiben und Lesen von Konsole
File	Sequenzieller Zugriff auf Dateien
FileInputStream	Eingabestream aus einer Datei
FileOutputStream	Ausgabestream in eine Datei
FileReader	Lesen von Text aus einer Datei
FileWriter	Schreiben von Text in eine Datei
PrintWriter	Formatierte Ausgabe von Objekten
RandomAccessFile	Wahlfreier Zugriff auf eine Datei
StringReader	Lesen von Zeichen aus String
StringWriter	Schreiben von Zeichen in einen StringBuffer

javax.swing

Klassen	Beschreibung
ButtonGroup	Gruppierung von Auswahlelementen
ImageIcon	Klasse für Icons
JApplet	Klasse für Applets
JButton	Schalter zum Klicken
JComboBox	Kombinationsfeld
JFrame	Klasse für Fenster
JLabel	Beschriftung für Komponenten
JList	Liste von Items
JMenu	Pop-up-Menü
JMenuBar	Menüleiste
JMenuItem	Einzelner Menüeintrag
JRadioButton	Radio-Button
JOptionPane	Dialoge
JPanel	Container für andere Komponenten
JScrollBar	Bildlaufleisten
JScrollPane	Komponente mit automatischen Bildlaufleisten
JTextArea	Komponente mit editierbarem, mehrzeiligem Text
TextField	Komponente mit editierbarer Textzeile
UIManager	Look-And-Feel-Manager

java.util

Klassen	Beschreibung
Arrays	Sortieren, Suchen, Vergleichen, Füllen von Arrays
ArrayDeque	Dynamische Datenstruktur einer Warteschlange
ArrayList	Dynamische Datenstruktur einer Liste
Calendar	Konvertieren von Datumsobjekten
Collections	Sortier- und Suchalgorithmen
Date	Datum und Zeit
DateFormat	Formatieren von Datumsanzeigen
HashMap	Dynamische Datenstruktur einer Hash-Tabelle
HashSet	Dynamische Datenstruktur einer Menge
Hashtable	Dynamische Datenstruktur einer Hashtabelle
Iterator	Durchlaufen von Collections-Objekten
LinkedList	Dynamische Datenstruktur einer Liste
PriorityQueue	Dynamische Datenstruktur einer Prioritätswarteschlange
Random	Erzeugung von Zufallszahlen
Scanner	Einlesen und Parsen von Eingabezeilen
Stack	Dynamische Datenstruktur eines Stacks
StringTokenizer	Zerlegung von Strings in Teilstrings (Tokens)
TreeMap	Dynamische Datenstruktur einer Hash-Tabelle
TreeSet	Dynamische Datenstruktur einer geordneten Menge
Vector	Dynamische Datenstruktur eines Vektors

java.lang

Klassen	Beschreibung
Boolean	Wrapper-Klasse für boolean
Byte	Wrapper-Klasse für byte
Character	Wrapper-Klasse für char
Class	Typen in der Laufzeitumgebung
ClassLoader	Klassenlader
ClassValue	Verbindet einen Wert mit einem Klassentyp
Double	Wrapper-Klasse für double
Enum	Aufzählungen
Float	Wrapper-Klasse für float
InheritableThreadLocal	Verbindet Werte mit einem Thread
Integer	Wrapper-Klasse für int
Long	Wrapper-Klasse für long
Math	Numerische Operationen
Number	Basisklasse für numerische Typen
Object	Basisklasse aller Java-Klassen

Package	Informationen eines Java-Pakets
Process	Kontrolle extern gestarteter Programme
ProcessBuilder	Optionen für externes Programm bestimmen
Runtime	Klasse mit diversen Systemmethoden
RuntimePermission	Rechte mit Laufzeiteigenschaften
SecurityManager	Sicherheitsmanager
Short	Wrapper-Klasse für <code>short</code>
StackTraceElement	Element für den Stack-Trace
StrictMath	Numerische Operationen strikt gerechnet
String	Zeichenketten
StringBuffer	Dynamische Zeichenketten
StringBuilder	Dynamische threadsichere Zeichenketten
System	Diverse Klassenmethoden
Thread	Nebenläufige Programme
ThreadGroup	Gruppiert Threads
ThreadLocal	Verbindung von Werten mit einem Thread
Throwable	Basistyp für Ausnahmen
Void	Spezieller Typ für <code>void</code> -Rückgabe

A2: Java-Schlüsselwörter

<code>abstract</code>	Deklaration abstrakter Klassen und Methoden
<code>assert</code>	Überprüfung von Zusicherungen
<code>boolean</code>	Java-Datentyp für boolesche Werte
<code>break</code>	Herausspringen aus Schleifen oder der <code>switch</code> -Anweisung
<code>byte</code>	Java-Datentyp für 1-Byte-Zahl
<code>case</code>	Auswahl Fälle in der <code>switch</code> -Anweisung
<code>catch</code>	Ausnahmebehandlung bei Exception-Behandlung
<code>char</code>	Java-Datentyp für 2-Byte-Zahl
<code>class</code>	Deklaration einer Klasse
<code>continue</code>	Starten eines neuen Durchgangs in einer Schleife
<code>default</code>	Standardeinsprungsmarke in einer <code>switch</code> -Anweisung
<code>do</code>	Teil einer Schleife
<code>double</code>	Java-Datentyp für 8-Byte-Zahl
<code>else</code>	Teil der bedingten Anweisung
<code>enum</code>	Definition eines Aufzählungstyps
<code>extends</code>	Angabe der Vaterklasse bei der Klassendeklaration
<code>final</code>	Modifikator für Klassen, Methoden, Datenfelder und Variablen
<code>finally</code>	Ausführungsblock bei Exception-Behandlung
<code>float</code>	Java-Datentyp für 4-Byte-Zahl
<code>for</code>	Schleifenanweisung
<code>if</code>	Bedingte Anweisung
<code>implements</code>	Implementierung einer Schnittstelle
<code>import</code>	Einbindung von Klassen aus anderen Paketen
<code>instanceof</code>	Überprüfung der Referenz auf ein Objekt einer bestimmten Klasse
<code>int</code>	Java-Datentyp für 4-Byte-Zahl
<code>interface</code>	Dient zur Deklaration einer Schnittstelle
<code>long</code>	Java-Datentyp für 8-Byte-Zahl
<code>native</code>	Modifikator für Methoden einer anderen Sprache
<code>new</code>	Erzeugung eines neuen Objekts
<code>package</code>	Deklaration eines Pakets
<code>private</code>	Zugriffsmodifikator
<code>protected</code>	Zugriffsmodifikator
<code>public</code>	Zugriffsmodifikator
<code>return</code>	Anweisungen für den Rücksprung aus einer Methode
<code>short</code>	Java-Datentyp für 2-Byte-Zahl
<code>static</code>	Modifikator für Methoden, Datenfelder und Klassen
<code>super</code>	Aufruf des Konstruktors der Vaterklasse ermöglicht
<code>switch</code>	Auswahanweisung
<code>synchronized</code>	Synchronisation eines Threads

<code>this</code>	Referenz auf das eigene Objekt, Aufruf eines klasseneigenen Konstruktors
<code>throw</code>	Auswerfen einer Ausnahme bei Exception
<code>throws</code>	Auflistung der Ausnahmen bei Exception-Behandlung
<code>try</code>	Ausnahmeblock bei Exception-Behandlung
<code>void</code>	Methode besitzt keinen Rückgabewert
<code>volatile</code>	Kennzeichnung von Datenfeldern, die von mehreren Threads verändert werden
<code>while</code>	Schleifenanweisung

A3: Begriffe der objektorientierten Programmierung

Abgeleitete Klasse	Klasse (Unterklasse, Subklasse), die von einer Basisklasse (Oberklasse, Superklasse, Vaterklasse) abgeleitet wird und alle Instanzvariablen und Methoden der Basisklasse erbt
Abhängigkeit	Spezielle Beziehung zwischen zwei Modulen, bei der sich eine Änderung eines Moduls auf das andere Modul auswirken kann
Abstrakte Klasse	Klasse, die mit dem Schlüsselwort <code>abstract</code> versehen ist und von der keine Instanzen gebildet werden können, da sie teilweise nur Methodenköpfe enthalten
Abstraktion	Vorgehensweise, unwichtige Einzelheiten auszublenden und Gemeinsamkeiten zusammenzufassen, um sich so auf das Wesentliche zu konzentrieren
Aggregation	Beschreibung der Zusammensetzung eines Objekts aus anderen Objekten
Assoziation	Beschreibung der Interaktion zweier Objekte
Attribut	Eigenschaft einer Klasse zur Darstellung des Zustands eines Objekts
Basisklasse	Klasse (Superklasse, Oberklasse, übergeordnete Klasse), die in einer Vererbungshierarchie über der aktuellen Klasse steht
Entwurfsmuster	Menge von Klassen, die in einer definierten Art zusammenarbeiten, um gemeinsam die Lösung eines wiederkehrenden Problems zu ermöglichen
Identität	Eigenschaft, durch die sich das Objekt von anderen unterscheidet
Instanz	Spezielles Objekt mit gewissen Attributwerten
Kapselung	Zusammenfassung von Daten und zugehörigen Funktionen in eine gemeinsame Klasse
Klasse	Allgemeine Beschreibung für eine bestimmte Art von Objekten durch eine Struktur (Attribute) und das Verhalten (Methoden) in Form eines Datentyps, von dem Objekte erzeugt werden können
Komposition	Spezialfall einer Aggregation, bei der Abhängigkeiten zwischen den Objekten in der Form bestehen, dass ein beschriebenes Objekt nur durch gewisse Teilobjekte existiert
Konkrete Klasse	Klasse, von der im Gegensatz zu einer abstrakten Klasse spezielle Objekte gebildet werden können
Konstruktor	Spezielle Methode ohne Rückgabotyp, die den Namen der Klasse trägt und zur Initialisierung eines Objekts aufgerufen wird
Methode	Dynamische Eigenschaft zur Beschreibung des Verhaltens durch Implementierung einer Operation
Objekte	Daten- und Programmstrukturen, die Eigenschaften und Verhaltensweisen besitzen

Paket	Gruppierung von Modellelementen wie Klassen, Schnittstellen unter einem Namen
Polymorphie	Verschiedene Objekte, die auf die gleiche Anweisung unterschiedlich reagieren können
Schnittstelle	Vereinbarung von Signaturen von Methoden, die in unterschiedlichen Klassen implementiert ist
Vererbung	Übernahme der Merkmale (Attribute, Methoden) der vorhandenen Klasse durch die neue Klasse

A4: Objektorientierte Entwurfsmuster

Strukturmuster

Adapter	Anpassung der inkompatiblen Schnittstelle der vorhandenen Klasse an die gewünschte Form
Brücke	Trennung der Implementierung und der Schnittstelle für eine unabhängige Entwicklung
Dekorierer	Hinzufügung von zusätzlicher Funktionalität zu einem Objekt während der Laufzeit
Fassade	Definition einer vereinfachten Schnittstelle zum Zugriff auf die Klassen eines Subsystems
Fliegengewicht	Vereinfachung der Anwendung vieler Objekte, die variable Informationen teilen
Kompositum	Zusammensetzung von komplexeren Objekten aus einfacheren Objekten
Proxy	Verbergung eines Objekts hinter einem Stellvertreterobjekt

Verhaltensmuster

Befehl	Kapselung eines Befehls in Form eines Objekts zur Trennung von Erzeugung und Ausführung
Beobachter	Benachrichtigung aller Beobachter bei Zustandsänderung des beobachteten Objekts
Besucher	Hinzufügung neuer Operationen zu einer bestehenden Datenstruktur
Interceptor	Erweiterung der Verarbeitungskette um ein neues Glied zur Hinzufügung von Zusatzfunktionalitäten
Interpreter	Beschreibung von Problemen in einer eigenen Sprache mit anschließender Übersetzung
Iterator	Durchlaufen einer zusammengesetzten Datenstruktur in verschiedenen Durchlaufstrategien
Rolle	Annehmen eines Objekts von unterschiedlichen Rollen in Form von eigenständigen Objekten
Schablone	Festlegung der Struktur eines Algorithmus in einer Basisklasse und deren variablen Teile in diversen Unterklassen
Strategie	Austausch eines Algorithmus in Form einer Kapsel zur Laufzeit
Vermittler	Steuerung des Zusammenspiels vieler Objekte, um die einzelnen Objekte voneinander zu entkoppeln
Zustand	Änderung des Verhaltens eines Objekts, sobald sich ein innerer Zustand ändert

Erzeugungsmuster

Abstrakte Fabrik	Auswahl einer Produktfamilie zur Erzeugung von Objekten zur Laufzeit
Erbauer	Konstruktion von neuen Objekten aus verschiedenen vorhandenen Teilen
Fabrik	Erzeugung einer konkreten Instanz wird in der Methode einer Unterklasse gekapselt
Multiton	Mehrmaliges Erzeugen einer Instanz einer Klasse für begrenzte Anzahl von Ressourcen
Objektpool	Wiederverwendung von aufwendig zu erzeugenden Objekten aus einem Pool
Prototyp	Erzeugung neuer Instanzen auf Grundlage eines Prototyps durch Kopieren oder Anpassen
Singleton	Einmaliges Erzeugen einer Instanz einer Klasse für zentrale Ressourcen

Sachverzeichnis

8-Puzzle, 154

A

A*-Algorithmus, 147

optimal, 148

Ablaufplanung, 166

Abstandsmatrix, 193

abstract, 27

Abstrakt

Klasse, 27

Methode, 27

Adapter, 40

Adjazenzliste, 192

Adjazenzmatrix, 120, 190

Admittanzmatrix, 191

Aggregation, 20

Akzeptor, 260

Algorithmenmuster

Backtracking, 233

dynamische Programmierung, 156, 233

Greedy, 137, 144, 198

Algorithmus

evolutionär, 128

Alphabet, 244

Äquidensiten, 353

Äquivalenzklasse, 251

Äquivalenzproblem, 250, 251, 277, 293, 295

Äquivalenzrelation, 251

Arbeiter-Maschinen-Zuordnungsproblem, 210

Arithmetische Ausdrücke, 288

ArrayList, 102

Auftragsplanung, 230

Automat

Äquivalenz, 250, 251

autonom, 246

deterministisch, 245

E/A-Automat, 265

Erreichbarkeit, 249, 250

Graph, 247

irreduzibel, 249

lebendig, 249

Matrixdarstellung, 247

Mealy, 266

Minimalautomat, 250, 251

Moore, 266

nicht deterministisch, 274

Prozess, 241

Quotientenautomat, 252

Sprache, 247, 248

steuerbar, 260

Tabelle, 247

Wortproblem, 249, 250

Automatengraph, 247

Automatentabelle, 247

Automatisierungstechnik, 9

Fahrzeugautomatisierung, 11

Gebäudeautomatisierung, 11

Informationstechnik, 11

Produktionsautomatisierung, 11

Prozessautomatisierung, 11

Steuerung, 9

Überwachung, 9

B

Bahnplanung, 152

Baugruppenauslegung, 129

Baugruppenoptimierung, 166

Baum, 114

binär, 115

Blätter, 114

Geschwister, 114

Verzweigungsgrad, 115

- voll, 115
- vollständig, 115
- Wurzel, 114
- Beobachter, 58
- Bereichwachstumsverfahren, 339
 - Homogenitätskriterium, 340
- Big Data, 7
- Bilder
 - Farbe, 310
 - Grau, 311
 - RGB, 311
- Bildmatrix, 311
- Bildpunkt
 - Elimination, 326
 - Nachbarschaft, 326
- Bildverarbeitung, 307
 - 2 D, 308
 - 3 D, 308
 - 4 D, 308
 - Bildanalyse, 310
 - Bilderkennung, 307
 - Bilderzeugung, 308
 - Bildregistrierung, 309
 - Bildverbesserung, 307
 - Bildvisualisierung, 310
 - Bildvorverarbeitung, 309
 - Klassifikation, 310
 - Maschinenbau, 307
 - Medizin, 308
 - Merkmalsextraktion, 310
 - Segmentierung, 310
 - Vorverarbeitung, 309
- Bildvorverarbeitung
 - Bildfilterung, 313
 - Geometrische Operatoren, 314
 - Globale Operatoren, 314
- Bildzeilenelimination, 354
- Bipartiter Graph, 188
- BitSet, 101
- Breitensuche, 129
- Briefträgerproblem, 215, 218
- Buchstabe, 244
- C**
 - catch, 72
 - Chiplayoutplanung, 129
 - Chomsky-Hierarchie, 290
 - Class.forName, 87
 - Clique, 222

- Comparable, 35
- Compilerbau, 288
- Computergrafik, 307
- Connection
 - createStatement, 88
- CYK-Algorithmus, 293, 296
- D**
 - Datenstrukturen
 - dynamisch, 101
 - DEA, 245
 - Dekorierer, 47
 - Deterministischer Automat
 - Sprache, 248
 - deterministischer Automat, 245
 - Anfangszustand, 245
 - Eingabealphabet, 245
 - Endzustandsmenge, 245
 - erweiterte Übergangsfunktion, 247
 - Übergangsfunktion, 245
 - Zustandsmenge, 245
 - Dialoggestaltung, 261
 - Digitalisierung, 7, 125
 - Dijkstra-Algorithmus, 134, 137, 139, 148, 207, 251, 342
 - Dominierende Menge, 222
 - DriverManager, 88
 - getConnection, 88
 - Dynamische Programmierung, 293
- E**
 - E/A-Automat, 265
 - Anfangszustand, 266
 - Ausgabealphabet, 266
 - Ausgangsfunktion, 266
 - Eingabealphabet, 266
 - Übergangsfunktion, 266
 - Wortproblem, 267
 - Zustandsmenge, 266
 - Entwurfsmuster, 39
 - Erzeugungsmuster, 40, 61
 - Strukturmuster, 40
 - Verhaltensmuster, 40, 52
 - Ereignisdiskrete Systeme, 241
 - Erreichbarkeitsproblem, 249, 250
 - Erzeugungsmuster, 40, 61
 - Fabrik, 64
 - Singleton, 62
 - Euler-Kreis, 214

- Excel, 98
- Exception, 71
 - finally, 76
 - getMessage, 74
 - printStackTrace, 74
 - throws, 75
- Expertensysteme, 262
- extends, 28

- F**
- Fabrik, 64
- Fahrkartenautomat, 242
- Faltung, 315
 - Kern, 315
- Farbbilder, 311
- Färbung
 - Backtracking, 233
 - chromatische Zahl, 232
 - dynamische Programmierung, 233
- Fassade, 44
- FIFO-Prinzip, 113
- Filter, 315
 - Differenzenoperatoren, 318
 - Gauß, 316
 - Laplace-Operator, 320
 - linear, 315
 - Median, 317
 - Mittelwert, 315
 - nicht linear, 315
 - Sobel-Operatoren, 319
- Filterung
 - Hochpass, 314, 317
 - Maske, 314
 - Tiefpass, 314, 315
- Flussnetz, 203
 - Betrag, 204
 - Edmonds-Karp-Strategie, 207
 - Erweiterungspfad, 206
 - Ford-und-Fulkerson-Algorithmus, 206
 - Kapazität, 204
 - Kapazitätsbedingung, 204
 - Kirchhoff'sches Gesetz, 204
 - Quelle, 204
 - Restkapazität, 206
 - Restnetz, 206
 - Schnitt, 211
 - Senke, 204
 - Symmetriebedingung, 204
- Ford-und-Fulkerson-Algorithmus, 206
- Fraktale, 349
 - Boxcounting-Methode, 350
 - Dimension, 350
- Frequenzplanung, 236
- Funktionszeiger, 37

- G**
- Generator, 260
- Genetische Algorithmen, 157, 168
 - Elterngeneration, 169
 - Fitness, 168
 - Generation, 169
 - Genotyp, 168
 - Individuum, 168
 - Kinderindividuen, 169
 - Kombination, 170
 - Mutation, 168, 170
 - Population, 168
 - Rekombination, 168
 - Selektion, 168, 169
- Gerüst, 195
 - Anzahl, 195
 - Baum, 195
 - Kreis, 195
 - Satz von Kirchhoff, 196
- Gierige Suche, 143
 - Kostenschätzfunktion, 144
- Grad, 185
- Gradient, 317
- Gradmatrix, 191
- Grammatik, 288, 289, 291
 - Ableitung, 289
 - Ableitungsregeln, 289
 - Äquivalenz, 293, 295
 - Chomsky-Normalform, 292
 - kontextfrei, 291
 - Natürliche Sprache, 302
 - Nichtterminalsymbole, 289
 - rechtslinear, 290
 - sensitiv, 291
 - Sprache, 290
 - Startsymbol, 289
 - Terminalsymbole, 289
 - Typ 0, 291
 - Typ 1, 291
 - Typ 2, 291
 - Typ 3, 290
 - Wortproblem, 293
- Graph, 119, 183

Abstand, 193
 aufspannender Untergraph, 186
 Baum, 188
 bipartit, 188
 chromatische Zahl, 232
 Färbung, 232
 gerichtet, 119, 185
 gerichtete Kante, 185
 Gerüst, 195
 inzident, 185
 Kantenfolge, 187, 194
 Kantenmenge, 119, 184
 Knotenmenge, 119, 184
 Komplement, 223
 Komponente, 187
 Kreis, 188
 Listendarstellung, 192
 Matrizendarstellung, 190
 maximaler Grad, 185
 minimaler Grad, 185
 Partition, 232
 regulär, 188
 schlicht, 185
 ungerichtet, 119, 184
 ungerichtete Kante, 184
 Untergraph, 186
 vollständig, 188
 Wald, 188
 Weg, 188
 zusammenhängend, 187
 Graphen, 241
 Cliquenzahl, 222
 Dominierungszahl, 222
 Knotenüberdeckungszahl, 222
 Unabhängigkeitszahl, 222
 Graphentheorie, 3
 Graphfärbung, 230
 Graubilder, 311

H

HashSet, 104
 Hashtable, 101
 Haus von Nikolaus, 214
 heuristische Kostenschätzfunktion
 zulässig, 148
 Hochpassfilter, 314, 317
 Homogene Koordinaten, 330
 HSQldb, 88

I

implements, 32
 import, 85
 Industrie 4.0, 2
 Industrieroboter, 152
 Interface, 31
 Definition, 32
 Funktionszeiger, 37
 Implementierung, 32
 Konstanten, 36
 statischer Import, 36
 interface, 32
 Inzidenzmatrix, 190
 Iterator, 104
 iterator, 104

J

JDBC, 87

K

Kante
 Canny-Kantendetektor, 354
 Entfernen, 190
 Extraktion, 326
 Kontraktion, 190
 parallel, 185
 Schlinge, 185
 Kantenfolge, 187, 194
 geschlossene Kantenfolge, 187
 Kantenweg, 187
 Kantenzug, 187
 Kreis, 187
 Kellerautomat, 288, 292
 Klasse
 ableiten, 28
 abstrakt, 27
 innere, 23
 konkret, 28
 lokal, 25
 Mitgliedsklasse, 23
 statische innere, 24
 Knoten
 adjacent, 185
 Außengrad, 192
 Fusion, 190
 Grad, 185
 Innengrad, 192
 Nachbarschaft, 185
 Knotenmengen, 220

- Clique, [221](#)
- dominierende, [221](#)
- Überdeckung, [221](#)
- unabhängige, [221](#)
- Knotenüberdeckung, [222](#)
- Koch-Kurve, [349](#)
- Kombinatorische Optimierung, [4](#)
- Kommunikationsnetze, [202](#), [211](#), [230](#)
- Komposition, [20](#)
- Königsberger Brückenproblem, [213](#)
- Konturfindung, [341](#)
- Krankenhausinformationssysteme, [6](#)
- Krebsdiagnose, [348](#)
- Kreis, [187](#)
- Kruskal-Algorithmus, [198](#)
- Künstliche Intelligenz, [8](#), [125](#)
- Kürzeste Wege Problem, [136](#)

L

- LIFO-Prinzip, [111](#)
- Lineare Optimierung, [3](#)
- Linguistik, [288](#)
- Linienfinder, [355](#)
- LinkedList, [102](#)
- Link-State-Algorithmus, [141](#)
- Liskov'sches Substitutionsprinzip, [28](#)
- List, [102](#)
 - ArrayList, [102](#)
 - LinkedList, [102](#)
- Listendarstellung, [192](#)
 - gerichteten Graphen, [192](#)
 - gewichteten Graphen, [192](#)
 - ungerichteten Graphen, [192](#)
- Livelock, [249](#)
- Live-Wire-Verfahren, [341](#)
- Logistik, [2](#)

M

- Map, [102](#), [105](#)
 - HashMap, [105](#)
- Maschinenkalibrierung, [178](#)
- Matcher, [284](#)
 - appendReplacement, [285](#)
 - appendTail, [285](#)
 - endend, [284](#)
 - findfind, [284](#)
 - groupgroup, [284](#)
 - matcher, [284](#)
 - startstart, [284](#)

- Matching, [210](#), [220](#)

Matrix

- Abstand, [193](#)
- Adjazenz, [190](#)
- Admittanz, [191](#)
- Grad, [191](#)
- Inzidenz, [190](#)
- Potenz, [194](#)
- Matrizendarstellung, [190](#)
- Mealy-Automat, [266](#)
- Medizinische Bildverarbeitung, [7](#)
- Medizinische Dokumentation, [6](#)
- Medizinische Expertensysteme, [8](#)
- Medizinische Informatik, [5](#)
- Medizinische Informationssysteme, [6](#)
- Mensch-Maschine-Kommunikation, [262](#), [288](#), [308](#)
- Methode der kleinsten Fehlerquadrate, [331](#)
- Minimal aufspannende Bäume, [196](#)
- Minimalautomat, [250](#), [251](#)
- Mitgliedsklasse, [23](#)
- Mobilfunkfrequenzen, [230](#)
- Moore-Automat, [266](#)
- Multithreading, [77](#)

N

- Nachbarschaft, [185](#)
- Natürliche Sprache, [287](#)
- NEA, [274](#)
- Netzplantechnik, [3](#)
- Nicht deterministischer Automat, [274](#)
 - Anfangszustand, [274](#)
 - Eingabealphabet, [274](#)
 - Endzustandsmenge, [274](#)
 - Sprache, [275](#)
 - Übergangsfunktion, [274](#)
 - Wortproblem, [277](#)
 - Zustandsmenge, [274](#)
- Nichtdeterminismus, [273](#)

O

- Objekt, [20](#)
 - Aggregation, [20](#)
 - Analyse, [20](#)
 - Attribute, [20](#)
 - Interaktion, [20](#)
 - Komposition, [20](#)
 - Konzepte, [22](#)
 - kooperierend, [20](#)

- Verhalten, 20
- Objektorientierte Analyse, 20
- Objektorientierte Konzepte, 22
- Objektorientierte Programmierung
 - Entwurfsmuster, 39
 - Erweiterbarkeit, 22
 - Flexibilität, 22
 - Redundanzfreiheit, 22
 - Veränderbarkeit, 22
 - Verständlichkeit, 22
 - Wartbarkeit, 22
 - Wiederverwendbarkeit, 22
- OLSR-Protokoll, 141
- Operations Research, 1
- Operatoren
 - geometrisch, 314
 - global, 314
- Optimierung
 - kombinatorisch, 4
 - linear, 3
 - nicht linear, 3
- Optimierungsproblem, 157
 - Bewertungsfunktion, 157
 - Bewertungsrelation, 157
 - Suchraum, 157
- P**
 - package, 85
 - Packungsprobleme, 177
 - Pakete, 85
 - Partition, 232
 - Block, 232
 - unabhängige, 232
 - Pattern, 283
 - split, 284
 - Polymorphie, 27
 - Überladen, 27
 - Überschreiben, 27
 - Potenzmatrix, 194
 - Potenzmengenkonstruktion, 278
 - Prim-Algorithmus, 198
 - Produktion, 2
 - Produktionssteuerung, 270
 - Programmiersprache
 - Beschreibung, 301
 - Syntax, 287
 - Programmierung
 - objektorientiert, 17
 - strukturiert, 16

- Q**
 - Queue, 114
 - Quotientenautomat, 252
- R**
 - Randextraktion, 354
 - Refactoring, 23
 - Registrierung, 327
 - flächenbasiert, 329
 - Landmarkenbasiert, 331
 - merkmalsbasiert, 329
 - Regulärer Ausdruck, 276
 - Natürliche Sprache, 287
 - Programmiersprache, 287
 - Semantik, 276
 - Syntax, 275
 - Wortproblem, 278
- ResultSet
 - next, 90
- Roboternavigation, 129
- Routenplanung, 126, 128, 134, 153
- Routingverfahren, 141
- Rucksackproblem, 156
- Rundreiseproblem, 156
- S**
 - Schablone, 52
 - Schaltregel, 247
 - Schaltzeit, 247
 - Schnittstelle, 31
 - Schwarz-Weiß-Luminanz, 311
 - Schwellwertverfahren, 339
 - Segmentierung, 142, 337
 - Bereichwachstum, 339
 - beschreibende Parameter, 338
 - Live-Wire-Verfahren, 341
 - Schwellwert, 339
 - überdeckungsfrei, 338
 - vollständig, 338
 - zusammenhängend, 338
 - Set, 102, 104
 - HashSet, 104
 - Simplex-Algorithmus, 3
 - Simulated Annealing, 157, 159
 - Singleton, 62
 - Skalierung
 - linear, 352
 - logarithmisch, 353
 - Soziale Netze, 212

- Clique, 213
- Dichte, 212
- Flussnetz, 212
- Zentralität, 213
- Spielproblem, 129
- Sprache, 245, 247, 248, 290
 - kontextfrei, 292
 - kontextsensitiv, 292
 - regulär, 248, 252, 275, 282, 292
 - rekursiv aufzählbar, 292
 - Verkettung, 245
- Sprachorientierte Modellierung, 260
- SQL, 87
 - Befehle, 89
- Stack, 101, 112
- Stapel, 111
- Statement
 - executeQuery, 89
 - executeUpdate, 91
- Strategie, 55
- Strukturmuster, 40
 - Adapter, 40
 - Dekorierer, 47
 - Fassade, 44
- Stundenplanung, 166
- Substitution, 28
- Suchalgorithmus, 128
 - deterministisch, 128
 - korrekt, 128
 - lokal, 128
 - optimal, 128
 - vollständig, 128
- Suche, 125, 158, 169
 - A*, 147, 148
 - Breitensuche, 129
 - Dijkstra, 134
 - gierig, 143
 - informiert, 128
 - lokal, 156
 - Tiefensuche, 129, 132
 - uninformiert, 128
- Suchproblem, 126
 - Aktion, 126
 - Endzustand, 126
 - Kostenfunktion, 126
 - Lösung, 126
 - Pfad, 126
 - Startzustand, 126
 - Suchbaum, 126

- Suchraum, 126
- Zustandsraum, 126

T

- Table-Filling-Algorithmus, 252, 258
- Textmuster, 287
- Textsuche, 286
- Thread, 77
 - interrupt, 80
 - interrupted, 80
 - isAlive, 81
 - isInterrupted, 80
 - run, 78
- Runnable, 77, 81
- sleep, 79
- start, 78
- Synchronisation, 83
 - synchronized, 84
- Threshold Accepting, 161
- Throwable, 73
- Tiefensuche, 132
- Tiefpassfilter, 314, 315
- Topologische Indizes, 143
- Tourenprobleme, 213
- Transformation
 - affin, 331
 - perspektivisch, 331
 - starr, 330
- try, 72
- Turing-Maschine, 288, 292

U

- Übergangsfunktion, 245
 - partiell, 246
- Überladen, 27
- Überschreiben, 27
- Unabhängige Knotenmenge, 221
 - gesättigt, 223
 - maximal, 223
- Unabhängige Menge, 222

V

- Vector, 101
- Vererbung, 28
- Verhaltensmuster, 40, 52
 - Beobachter, 58
 - Schablone, 52
 - Strategie, 55
- Verkehrssampel, 261

Verkehrsnetze, [202](#)

Verkettete Listen, [107](#)

Versuchsplanung, [167](#)

W

Warteschlange, [113](#), [131](#), [286](#)

Warteschlangentheorie, [4](#)

Weg

 kürzester, [136](#)

 Länge, [136](#)

Wort, [244](#)

 Länge, [244](#)

 leer, [244](#)

 Verkettung, [244](#)

Wortproblem, [249](#), [250](#), [267](#), [277](#), [293](#)

X

XML, [94](#)

Z

Zahlendarstellung, [243](#)

Zustand, [241](#)

 stark zusammenhängend, [249](#)

Zustandsorientierte Modellierung, [260](#)