


Jan Lunze

Graph-Theoretical Methods in Systems Theory and Control

 Edition MoRa

Edition MoRa

Graphentheoretische Methoden der Regelungstechnik

ISBN 9789463860147

Networked Control of Multi-Agent Systems

Second edition: ISBN 9789403648477

Networked Control of Multi-Agent Systems: Application Studies

ISBN 9789403648484

Feedback Control of Large-Scale Systems

Reprint edition: ISBN 9789463982740

Graph-Theoretical Methods in Systems Theory and Control

ISBN 9789403726854

Jan Lunze

Graph-Theoretical Methods in Systems Theory and Control

Modelling, analysis and design methods
based on directed graphs, bipartite graphs, and random graphs

With 511 figures, 114 examples, 118 exercises
and MATLAB scripts

Author:

Prof. Dr.-Ing. Jan Lunze
Ruhr-Universität Bochum
Lehrstuhl für Automatisierungstechnik und Prozessinformatik
44780 Bochum
Lunze@atp.rub.de

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. For product information, please contact www.mathworks.com

© Jan Lunze, Münster 2024

ISBN: 9789403726854

Cover drawing: Andrea Marschall

publish.bookmundo.de

www.editionmora.de/gmsc

Preface

Engineering systems are composed of interconnected subsystems to satisfy a desired function. Their behaviour can be explained in terms of their structure, which is best represented by a graph showing how the subsystems located in the vertices and the interconnections described by the edges combine to the overall system. That is why graph-theoretical methods provide a deep insight into engineering systems and many phenomena, which at first sight seem to have complicated causes, can be retraced to simple rules when analysing their graph representations.

Engineers think in terms of concepts. The first step to solve an analysis or design problem concentrates on the basic mechanisms that a dynamical system is built on. The focus lies on the qualitative subsystem behaviour and the main component interactions rather than on detailed equations and precise parameter values. For this step graph-theoretical methods are appropriate and successful. They help to analyse the structure, to understand the structure and, finally, to design the structure so as to get the required system performance.

Aims. This book has the goal to bring important results of graph theory and systems theory together and to present them in a consistent language and notation.

- It shows that basic results of systems analysis and control design can be derived from the system structure, which is represented by an appropriate graph. To this aim, it connects graph-theoretical and systems-theoretical notions and methods.
- Algebraic graph theory provides a direct link between the mathematical models used in systems theory and structural representations by graphs. The book presents the basic principles of this theory together with their application to a variety of systems and control problems.
- Although important systems-theoretical properties are essentially based on the system structure, their manifestation in the actual system behaviour depends strongly upon parameter values. To show the border between the validity of structural analysis methods and of quantitative ways of analysis is the third important aim of this book.

Contents. The book is divided into three parts, which distinguish with respect to their graph-theoretical requirements:

- **Part I – Graphs and graph search.** Methods for the decomposition of directed and undirected graphs and for graph search lead to systematic ways for the solution of logical decision problems and to Bayesian networks, which connect graph theory and probability theory for the analysis of uncertain systems.

- **Part II – Algebraic graph theory.** A direct connection between the models of dynamical systems and their graph-theoretical interpretation is established by the algebraic graph theory, which deals with graphs in terms of their matrix representations. The adjacency matrix and the Laplacian matrix of a graph become part of the dynamical models for both continuous-variable and discrete-event systems and are investigated to evaluate the model properties and to decompose models and analysis tasks. The other way round, matrix properties like the rank or the determinant are interpreted in a graph-theoretical way and lead to generic properties of linear systems.
- **Part III – Bipartite and random graphs.** The DM decomposition of bipartite graphs and the small-world property of random graphs are used to manipulate sets of model equations for a sequential way of their solution or to find efficient communication structures for networked systems.

In the table of contents the graph-theoretical chapters are marked grey to indicate the division of the book into these three parts.


Didactical concept. The presentation of graph-theoretical methods and their application to systems analysis and control design are separated by chapters. The style and depth of the introductions to graph theory in Chapters 2, 5 and 10 are adjusted to the needs of the systems-theoretical parts. The application chapters are independent, self-contained, and separately to read by those who are only interested in single topics.

The book focuses on graphs that are not only used to illustrate a problem graphically, but allow to exploit the powerful graph-theoretical tools for finding solutions. Every application chapter begins with an introduction to the systems-theoretical background and its graph-theoretical representation. Important results are marked in the text, summarised as lemmas or theorems, highlighted as framed equations, or transferred to algorithms. The algorithms aim at a profound understanding of its main ideas rather than the lowest possible complexity. Many of them are available as MATLAB code or in other program collections to be used for sample applications of the methods. For industrial purposes, more efficient implementations are commercially available. References at the end of each chapter provide the interested reader with adequate entry points to the literature.

Intentionally, the text includes only a few formal definitions, lemmas and theorems in order not to interrupt the flow of thoughts and to emphasise only the most important notions and results. Numerous examples and exercises from various application domains illustrate the efficiency of the combination of systems theory with graph theory.

The exercises are classified as follows:

- **Exercises** (without an asterisk) help to understand the material just presented. They can be solved in analogy to the examples.
- **Exercises marked by an asterisk** apply the methods to a larger numerical or practical example. Their solutions are given in Appendix 1. The readers are encouraged first to solve the exercises themselves before going to the solution in the appendix.

Exercises with numerical calculations are marked with the MATLAB symbol .

Although the mathematical literature offers numerous textbooks about graph theory, there is no well-standardised terminology. There are not even uniform terms for the basic elements of

a graph like vertices (nodes, points) and edges (lines, links, arcs). This book introduces every notion explicitly, concentrates on one or two terms and mentions synonyms in the notes-and-references sections and the index to make the contents accessible for a broad audience.

Readership. This textbook is valuable to advanced-level students, researchers and experts in industry in a variety of disciplines where graph-theoretic methods are relevant. Competence in linear algebra and in systems and control theory on the level of graduate students is required. Appendices review some basics in matrix theory and probability theory.

Parts of this text has been used regularly in basic control courses at the Ruhr-University Bochum (Germany), for a one-semester graduate course in the summer term 2023 and at several seminars in industry.

Acknowledgements. I am very grateful to my former PhD students for many stimulating discussions about the role of the system structure in the analysis and the design of control systems, in particular to THOMAS STEFFEN, MARTIN KONIECZNY, MICHAEL UNGERMANN, DANIEL VEY and SEBASTIAN PRÖLL. Our most recent research focussed on networked control systems. Some of the results are surveyed in Chapter 12 with contributions and experimental results of ALEXANDER SCHWAB, who has reviewed this chapter. The field of discrete-event systems likewise uses graph-theoretical methods, where I have worked with YANNICK NKE, MELANIE SCHUH and MARKUS ZGORZELSKI.

I have profited from many discussions on structural methods within common projects, at conferences, workshops or summer schools, in particular, with LOUISE TRAVÉ-MASSUYÈS (Toulouse), MOGENS BLANKE (Copenhagen), MICHEL KINNAERT (Brussels) and MARCEL STAROSWIECKI (Lille/Paris) with particular emphasis on fault-tolerant control.

My interest in the graph-theoretical analysis of dynamical systems goes back to my PhD studies on decentralised control. Many discussions and some common publications with Prof. Dr. KURT REINSCHKE supported my ambition to retrace dynamical phenomena back to their structural reasons. Professor KLAUS RÖBENACK (Dresden), who has extensively worked in a similar direction, has made valuable comments on Chapter 7.

Some of the application examples of this book resulted from tutorials in industry thanks to several invitations of Dr. DIETER SCHWARZMANN (R. Bosch GmbH), Dr. THIMO OEHL-SCHLÄGEL (IAV GmbH) and others.

Many thanks are due to Ms. ANDREA MARSCHALL who has drawn and re-drawn most of my figures. She has given approximately 300 graphs a uniform style and helped me to produce the final print data.

Finally, I hope that this book will convince engineers of the beauty of graph theory for applications in the fields of systems and control.

Münster, in January 2024

JAN LUNZE

The webpage www.editionmora.de/gmsc offers additional material including the MATLAB scripts for generating numerous figures of this book and teaching instructions.

Highlights of this textbook

- ★ **Introduction to graph theory:** The main ideas of graph theory and graph search are explained in Chapter 2, algebraic graph theory in Chapter 5, and bipartite and random graphs in Chapter 10.
- ★ **Heuristic search in artificial intelligence:** Since discrete decision problems have generally large search spaces, the A* search method is introduced in Chapter 3, which prefers promising search directions.
- ★ **Graph-theoretical structure of uncertain knowledge:** Bayesian networks represent the conditional stochastic independence of propositions. Reasoning along their edges utilises the structure of the probabilistic information (Chapter 4).
- ★ **Decomposition and aggregation of interconnected systems:** The coupling graph of interconnected systems introduced in Chapter 6 reveals which systems are decomposable into a series-parallel configuration, which simplifies analysis and design steps.
- ★ **Input-output behaviour of interconnected systems:** For signal-flow graphs, an extension of Cramer's rule for the solution of linear equations leads to Mason's formula to get the transfer function between selected input and output signals (Chapter 6).
- ★ **Generic properties of linear systems:** Controllability and observability are generic properties of dynamical systems, which mainly depend upon the structure graph investigated in Chapter 7.
- ★ **Graph-theoretical modelling and analysis of electrical networks:** Graph theory provides the basic steps of a branch current analysis or a node voltage analysis for modelling electrical networks and for the Kron reduction towards the effective resistance between two terminal nodes (Chapter 8).
- ★ **Maximum flows through networks:** For transportation systems, the max-flow min-cut theorem shows which edges represent the bottleneck of the network (Chapter 8).
- ★ **Structural analysis of discrete-event systems:** The automaton graph introduced in Chapter 9 is the basis of powerful tools for the analysis of nondeterministic and stochastic automata, Markov chains and hidden Markov models. In particular, the Viterbi algorithm finds the most probable path through this graph for a given output sequence.

- ★ **Structural analysis of constraint sets:** Bipartite graphs lead to a decomposition of sets of linear equations to select those variables that are uniquely determined by the model and to find a sequential way for computing these unknown variables (Chapter 11).
- ★ **Structural analysis of fault diagnosability:** For fault diagnosis the bipartite structure graph of a system has to be over-determined to reveal the redundancy that makes a system fault detectable (Chapter 11).
- ★ **Structure design of multi-agent systems:** With graph-theoretical methods the structure of leader-follower systems can be tailored to efficient set-point following and the communication structure can be adapted to the movement of mobile agents as shown in Chapter 12.

Contents

| | |
|--|-----|
| Preface | V |
| Highlights | IX |
| List of application examples | XIX |
| 1 Systems theory meets graph theory | 1 |
| 1.1 Graph representations of dynamical systems | 1 |
| 1.2 What graph theory offers | 4 |
| 1.3 The importance of the system structure | 6 |
| 1.4 Limitations of structural analysis methods | 7 |
| Notes and references | 8 |
| 2 Graphs and graph search | 9 |
| 2.1 Basic notions | 9 |
| 2.1.1 Directed graphs | 9 |
| 2.1.2 Undirected graphs | 11 |
| 2.1.3 Weighted graphs | 12 |
| 2.1.4 Specific graphs | 14 |
| 2.2 Reachability analysis | 15 |
| 2.2.1 Paths and cycles | 15 |
| 2.2.2 Reachability sets | 17 |
| 2.2.3 Subgraphs and union of graphs | 18 |
| 2.2.4 Strongly connected components | 19 |
| 2.3 Graph search | 23 |
| 2.3.1 Trémaux algorithm | 23 |
| 2.3.2 Depth-first search | 25 |
| 2.3.3 Breadth-first search | 28 |
| 2.3.4 Dijkstra algorithm | 30 |
| 2.3.5 Properties of graph-search algorithms | 38 |
| 2.4 MATLAB functions for graph objects | 41 |
| 2.4.1 Create graph objects | 41 |
| 2.4.2 Graph properties | 42 |
| 2.4.3 Plotting graphs | 44 |
| 2.4.4 Graph search | 44 |
| Notes and references | 48 |

| | | |
|----------|---|-----|
| 3 | Graph search in logic-based knowledge processing | 49 |
| 3.1 | Decision problems and their graph-theoretical way of solution | 49 |
| 3.2 | Heuristic graph search | 55 |
| 3.2.1 | Extensions of uninformed search methods | 55 |
| 3.2.2 | A* search | 57 |
| 3.2.3 | Example: Path planning for robots | 64 |
| 3.3 | Rule-based problem solving | 69 |
| 3.4 | Logic-based problem solving | 73 |
| 3.4.1 | Propositional logic | 73 |
| 3.4.2 | Logical entailment and inference | 77 |
| 3.4.3 | Resolution calculus | 80 |
| 3.4.4 | Inference graph and search strategies for resolution refutation systems | 83 |
| 3.4.5 | Example: Controller verification by model checking | 88 |
| 3.5 | Summary | 94 |
| | Notes and references | 94 |
| 4 | Graph-theoretical methods for approximate reasoning | 97 |
| 4.1 | Reasoning under uncertainty | 97 |
| 4.2 | Truth maintenance systems | 99 |
| 4.2.1 | Main idea | 99 |
| 4.2.2 | ATMS graph | 99 |
| 4.2.3 | Handling of contradictions | 105 |
| 4.2.4 | Model-based diagnosis with ATMS | 105 |
| 4.3 | Bayesian networks | 111 |
| 4.3.1 | Uncertain knowledge and its probabilistic representation | 111 |
| 4.3.2 | Evidential reasoning | 115 |
| 4.3.3 | Definition and specification of Bayesian networks | 117 |
| 4.3.4 | Causal reasoning with Bayesian networks | 127 |
| 4.3.5 | Diagnostic reasoning | 131 |
| 4.3.6 | Extensions | 137 |
| 4.4 | Summary | 143 |
| | Notes and references | 143 |
| 5 | Algebraic graph theory | 145 |
| 5.1 | Matrix representations of graphs | 145 |
| 5.1.1 | Adjacency matrix | 146 |
| 5.1.2 | Degree matrix | 148 |
| 5.1.3 | Incidence matrix | 149 |
| 5.1.4 | Laplacian matrix | 151 |
| 5.2 | Analysis of graphs | 154 |
| 5.2.1 | Paths | 154 |
| 5.2.2 | Algebraic reachability analysis | 155 |
| 5.2.3 | Strongly connected components | 157 |
| 5.2.4 | Algebraic connectivity | 161 |

| | | |
|----------|--|------------|
| 5.3 | Graph representation of matrices | 163 |
| 5.3.1 | Structure matrices | 163 |
| 5.3.2 | Graph induced by a matrix | 165 |
| 5.3.3 | Graph-theoretical interpretation of the determinant of a matrix | 165 |
| 5.3.4 | Characteristic polynomial | 169 |
| 5.3.5 | Structural rank | 172 |
| 5.3.6 | Decomposition of matrices | 175 |
| 5.4 | MATLAB functions for graph matrices | 178 |
| 5.4.1 | Graph matrices | 178 |
| 5.4.2 | Adjacency matrices of specific graphs | 180 |
| 5.4.3 | Determination of the connected components | 180 |
| 5.4.4 | Structural rank of a matrix | 181 |
| | Notes and references | 182 |
| 6 | Decomposition and aggregation of interconnected dynamical systems | 183 |
| 6.1 | Block diagrams | 183 |
| 6.1.1 | Definition | 183 |
| 6.1.2 | Uni-directional and bidirectional couplings | 185 |
| 6.1.3 | Decomposition and aggregation | 186 |
| 6.2 | Structural analysis of interconnected systems | 188 |
| 6.2.1 | Basic coupling structures | 188 |
| 6.2.2 | Coupling graph | 188 |
| 6.2.3 | Decomposable systems | 190 |
| 6.2.4 | Consequences for modelling and systems analysis | 193 |
| 6.3 | Aggregation of interconnected systems | 199 |
| 6.3.1 | Basic aggregation steps | 199 |
| 6.3.2 | Transformation rules for block diagrams | 201 |
| 6.3.3 | Aggregation of linear state-space models | 203 |
| 6.4 | Well-posedness of interconnected systems | 209 |
| 6.4.1 | Algebraic loops | 209 |
| 6.4.2 | Graph-theoretical analysis of the well-posedness | 212 |
| 6.4.3 | Example: Analysis of the power train of an electrical vehicle | 213 |
| 6.4.4 | Summary and extensions | 219 |
| 6.5 | Signal-flow graphs | 220 |
| 6.5.1 | Main idea | 220 |
| 6.5.2 | Comparison with other structural representations | 224 |
| 6.5.3 | Mason’s formula for signal-flow graphs | 226 |
| | Notes and references | 234 |

| | | |
|----------|--|-----|
| 7 | Generic properties of linear systems | 235 |
| 7.1 | Generic properties and atypical systems | 235 |
| 7.2 | Results on the controllability and observability of linear systems | 238 |
| 7.2.1 | Models | 238 |
| 7.2.2 | Controllability and observability criteria | 239 |
| 7.2.3 | Canonical structure of linear systems | 243 |
| 7.2.4 | Fixed eigenvalues | 244 |
| 7.3 | Structural controllability and structural observability | 246 |
| 7.3.1 | Motivation | 246 |
| 7.3.2 | Structure graph of linear dynamical systems | 247 |
| 7.3.3 | Structure graph of closed-loop systems | 250 |
| 7.3.4 | Reachability analysis of the structure graph | 253 |
| 7.3.5 | Structural controllability of a class of linear systems | 254 |
| 7.3.6 | Structural observability | 261 |
| 7.3.7 | Structurally fixed eigenvalues | 265 |
| 7.3.8 | Structural system decomposition | 269 |
| 7.3.9 | Extensions to nonlinear systems and strong structural controllability | 270 |
| 7.4 | The meaning of structural analysis results | 275 |
| 7.4.1 | Application scenarios of the structural analysis | 275 |
| 7.4.2 | Characterisation of structural properties | 277 |
| 7.5 | Where unobservability helps to satisfy a control goal | 280 |
| 7.5.1 | Motivation | 280 |
| 7.5.2 | Disturbance attenuation by feedback control | 281 |
| 7.5.3 | Unobservable components of state observers | 288 |
| 7.5.4 | Example: Tuned mass dampers in tall buildings | 295 |
| | Notes and references | 302 |
| 8 | Graph-theoretical modelling and analysis of electrical networks and flows | 305 |
| 8.1 | Electrical networks | 305 |
| 8.1.1 | Modelling and analysis problems | 305 |
| 8.1.2 | Electrical circuits | 306 |
| 8.1.3 | Circuit graphs | 308 |
| 8.1.4 | Kirchhoff's and Ohm's laws | 309 |
| 8.2 | Fundamental cycles and the cycle space of graphs | 313 |
| 8.3 | Branch current analysis | 318 |
| 8.3.1 | Graph-theoretical representation of Kirchhoff's laws | 318 |
| 8.3.2 | Modelling and analysis steps | 320 |
| 8.4 | Node voltage analysis | 324 |
| 8.4.1 | Circuits with current sources | 324 |
| 8.4.2 | Graph-theoretical way of modelling | 326 |
| 8.4.3 | Analysis steps | 328 |
| 8.4.4 | Generalisations | 330 |
| 8.5 | Network reduction and effective resistance | 334 |
| 8.5.1 | Problem statement and simple aggregation rules for resistive networks | 334 |
| 8.5.2 | Kron reduction | 338 |

| | | |
|-----------|---|------------|
| 8.6 | General flow networks | 346 |
| 8.6.1 | Flow networks and the maximum-flow problem | 346 |
| 8.6.2 | Properties of flow networks | 350 |
| 8.6.3 | Ford-Fulkerson algorithm to determine the maximum flow | 354 |
| 8.6.4 | Reformulation of the augmenting-path algorithm | 361 |
| 8.7 | MATLAB functions | 366 |
| 8.8 | Summary | 367 |
| | Notes and references | 367 |
| | Appendix 8.A: Proof of Lemma 8.1 | 368 |
| 9 | Structural analysis of discrete-event systems | 371 |
| 9.1 | Graph-theoretical problems in discrete-event systems theory | 371 |
| 9.1.1 | Systems with discrete signals | 371 |
| 9.1.2 | Survey of the graph-theoretical methods for discrete-event systems | 374 |
| 9.2 | Discrete-event system representations | 376 |
| 9.2.1 | Deterministic, nondeterministic and Σ -automata | 376 |
| 9.2.2 | Markov chains | 382 |
| 9.2.3 | Input-output automata | 386 |
| 9.3 | Structural analysis of nondeterministic automata | 388 |
| 9.3.1 | Graph-theoretical interpretation of the system behaviour | 388 |
| 9.3.2 | Reachability analysis | 391 |
| 9.3.3 | Decomposition of the state set | 395 |
| 9.3.4 | Classification of the automaton states | 396 |
| 9.4 | Structural analysis of Σ -automata | 401 |
| 9.4.1 | Languages represented by finite-state automata | 401 |
| 9.4.2 | Regular languages | 405 |
| 9.4.3 | Homomorphic and isomorphic automata | 408 |
| 9.5 | Structural analysis of Markov chains | 412 |
| 9.5.1 | State-based modelling of stochastic discrete-event systems | 412 |
| 9.5.2 | Graph-theoretical interpretation of the Chapman-Kolmogorov equation | 416 |
| 9.5.3 | Periodic Markov chains | 423 |
| 9.5.4 | Stationary probability distribution | 424 |
| 9.6 | Structural analysis of stochastic automata | 431 |
| 9.6.1 | Stochastic Mealy automata and controlled Markov chains | 431 |
| 9.6.2 | Hidden Markov models | 437 |
| 9.6.3 | Viterbi algorithm | 441 |
| | Notes and references | 450 |
| 10 | Bipartite and random graphs | 453 |
| 10.1 | Properties of bipartite graphs | 453 |
| 10.1.1 | Definition | 453 |
| 10.1.2 | Matchings | 458 |
| 10.1.3 | Matching algorithm | 461 |
| 10.2 | Decomposition of bipartite graphs | 466 |

| | | |
|-----------|---|------------|
| 10.2.1 | Bipartite graphs with a complete matching | 466 |
| 10.2.2 | DM decomposition | 469 |
| 10.2.3 | DM decomposition algorithm | 473 |
| 10.3 | Random graphs | 478 |
| 10.3.1 | Edge-generation rule for Erdős-Rényi graphs | 478 |
| 10.3.2 | Matrix representation | 479 |
| 10.3.3 | Combinatorial properties of random graphs | 480 |
| 10.3.4 | Characteristic path length and diameter | 483 |
| 10.3.5 | Phase transitions in random graphs | 485 |
| 10.3.6 | Generalised random graphs | 487 |
| 10.4 | MATLAB functions for bipartite and random graphs | 490 |
| 10.4.1 | DM decomposition | 490 |
| 10.4.2 | Generation of random graphs | 491 |
| 10.4.3 | Determination of the characteristic path length | 491 |
| | Notes and references | 492 |
| 11 | Structural analysis of constraint sets | 493 |
| 11.1 | Bipartite structure graph of systems | 493 |
| 11.1.1 | System representation by constraint sets | 493 |
| 11.1.2 | Structure graph of constraint sets | 496 |
| 11.1.3 | Matchings and causality | 498 |
| 11.2 | Graph-theoretical ways to solve sets of linear equations | 501 |
| 11.2.1 | Consistency of linear constraint sets | 501 |
| 11.2.2 | Structural decomposition of linear constraint sets | 503 |
| 11.2.3 | Constraint ranking in structurally well-determined models | 508 |
| 11.2.4 | Decomposition of structurally well-determined models | 511 |
| 11.2.5 | Extension to nonlinear equations | 514 |
| 11.2.6 | The significance of the structural analysis results | 516 |
| 11.3 | Structural analysis of dynamical systems | 519 |
| 11.3.1 | Differential constraints and their causal interpretation | 519 |
| 11.3.2 | Differential-algebraic constraints and their consistency | 521 |
| 11.3.3 | Relation between structurally over-determined systems and structurally observable system classes | 528 |
| 11.4 | Structural analysis for fault diagnosis | 531 |
| 11.4.1 | Diagnostic problem and its way of solution | 531 |
| 11.4.2 | Structural diagnosability | 534 |
| 11.4.3 | Fault isolation | 538 |
| 11.5 | Fault diagnosis of dynamical systems | 542 |
| 11.5.1 | Residual generation by analytical redundancy relations | 542 |
| 11.5.2 | Residual generation by state observers | 548 |
| 11.5.3 | A comprehensive way towards observer-based residual generators | 549 |
| 11.6 | Summary and extensions | 554 |
| | Notes and references | 555 |

| | |
|--|-----|
| 12 Networked systems | 557 |
| 12.1 Graph-theoretical problems of networked systems | 557 |
| 12.2 Consensus and synchronisation of multi-agent systems | 559 |
| 12.2.1 Synchronisation problem | 559 |
| 12.2.2 Synchronisation condition and its graph-theoretical interpretation | 562 |
| 12.2.3 Relation between the synchronous trajectory and the communication structure | 566 |
| 12.2.4 Convergence properties of the consensus dynamics | 570 |
| 12.3 Design of the communication structure of networked controllers | 577 |
| 12.3.1 Efficient set-point following problem | 577 |
| 12.3.2 Models | 580 |
| 12.3.3 Delay graph | 582 |
| 12.3.4 Communication structure design method | 584 |
| 12.4 Small-world architecture of networked systems | 591 |
| 12.4.1 Self-connecting systems | 591 |
| 12.4.2 Communication graphs with random links | 592 |
| 12.4.3 Properties of the effective communication graph | 595 |
| 12.5 Networked control with self-adapting communication structure | 600 |
| 12.5.1 Switching communication graphs | 600 |
| 12.5.2 Delaunay triangulation | 601 |
| 12.5.3 Retaining the Delaunay property for moving objects | 605 |
| Notes and references | 609 |
| Appendix 12.A: Proof of Theorem 12.1 | 611 |
| Appendix 12.B: Proof of Lemma 12.2 | 613 |
| Appendix 12.C: Proof of Theorem 12.2 | 614 |
| References | 617 |
| Appendix 1: Solutions of the exercises | 631 |
| Appendix 2: Sets and matrices | 719 |
| A2.1 Sets and set properties | 719 |
| A2.2 Matrix properties and matrix operations | 720 |
| A2.3 Specific matrices | 724 |
| A2.4 Solution of linear equations | 728 |
| Appendix 3: Basic notions of probability theory | 729 |
| Appendix 4: Additional MATLAB functions for graphs | 737 |
| Appendix 5: English–German dictionary | 743 |
| Index | 749 |

Application examples

Vehicles

- **Vehicle dynamics and control**

| | |
|---|----------|
| Structural analysis of a braking vehicle (Exercise 7.7 with solution) | 273, 667 |
| Prediction of the behaviour of a braking vehicle (Example 11.7) | 523 |
| Supervision of a single-roller dynamometer (Example 11.11) | 542 |
| Diagnosis of a vehicle (Exercise 11.10 with solution) | 553, 709 |
| Vehicle swarms with cooperative adaptive cruise control (Exercise 12.7 with solution) ... | 590, 713 |

- **Engines**

| | |
|---|-----|
| Probabilistic model of a Diesel engine (Example 4.4) | 115 |
| Bayesian network describing the cooling system of a Diesel engine (Example 4.6) | 121 |
| Probabilistic model of an engine cooling system (Example 4.3) | 113 |
| Diagnosis of an engine cooling system (Example 4.10) | 133 |
| Improvement of the diagnosis of an engine cooling system (Exercise 4.6) | 141 |

- **Power train of electrical vehicles**

| | |
|---|----------|
| Model of the power train of an electrical vehicle (Section 6.4.3) | 213 |
| Well-posedness of the power train of an electrical vehicle (Exercise 6.7) | 220 |
| Algebraic loop a vehicle model (Exercise 11.8 with solution) | 530, 706 |

Transportation systems

- **Travel by trains and lifts**

| | |
|--|-----|
| A travel with the Metro of Barcelona (Exercise 2.5) | 47 |
| Analysis of the lift to Albrechtsburg castle (Example 9.1) | 392 |
| Legal language of a lift (Example 9.5) | 402 |

- **Road networks**

| | |
|---|----|
| Verification of the control of two traffic lights (Example 3.10) | 89 |
| Verification of the traffic lights of two junctions (Exercise 3.11) | 93 |

Process control

| | |
|---|----------|
| Model of a bottling system (Exercise 4.9 with solution) | 142, 649 |
| Supervision of a bioprocess (Exercise 9.13 with solution) | 448, 695 |

• Tank systems

| | |
|---|----------|
| Analysis of a tank system (Exercise 4.4 with solution) | 110, 643 |
| Disturbance decoupling of a tank system (Exercise 7.13 with solution) | 301, 673 |
| Periodic and aperiodic states of a batch process (Example 9.3) | 397 |
| Model of a controlled tank system (Exercise 9.1 with solution) | 399, 685 |

Communication networks

| | |
|---|----------|
| Topology of a wireless communication network (Exercise 8.13 with solution) | 365, 685 |
| Modelling of the access to a computer network as a Bernoulli process (Example 9.10) | 421 |
| Analysis of a communication channel as a birth-death process (Exercise 9.9 with solution) | 430, 692 |
| Decoding of a communicated string (Exercise 9.14 with solution) | 449, 697 |

• Sensor networks

| | |
|--|----------|
| Analysis of a sensor network (Exercise 3.10 with solution) | 93, 637 |
| Diagnosis of a sensor network (Exercise 4.3 with solution) | 110, 642 |

Manufacturing systems

| | |
|--|----------|
| Modelling and analysis of a production cell (Exercise 9.2 with solution) | 400, 687 |
|--|----------|

• Machine tools

| | |
|---|----------|
| Fault diagnosis of a machine tool (Exercise 4.7 with solution) | 141, 646 |
| Equivalence of the models of two machine tools (Example 9.7) | 411 |
| Description of the queueing system of a machine tool as Markov chain (Example 9.8) | 413 |
| Model of a machine tool with two service units (Exercise 9.6) | 429 |
| Control of a machine tool with two service units (Example 9.14) | 433 |
| How many workpieces are refused by a machine tool? (Exercise 9.11) | 448 |

• Mechatronic systems

| | |
|---|----------|
| State observation of an inverted pendulum (Example 7.16) | 292 |
| Structural analysis of a rotary drive (Exercise 7.5 with solution) | 272, 665 |
| Structural properties of a loading bridge (Example 7.10) | 264 |
| Generic properties of loading bridges (Example 7.13) | 275 |
| Signal-flow graph and transfer function of the loading bridge (Exercise 6.9 with solution) | 234, 661 |
| Analysis of a loading bridge with its bipartite structure graph (Exercise 11.7 with solution) | 530, 705 |

Cooperative robots and unmanned aerial vehicles

• Path planning for robots

| | |
|---|---------|
| Path planning for robots (Section 3.2.3) | 64 |
| Configuration space of a robot with two degrees of freedom (Example 3.5) | 65 |
| Path planning for a robot with two degrees of freedom (Example 3.6) | 66 |
| Path planning for a robot with improved heuristics (Exercise 3.3 with solution) | 68, 634 |

• Robot formations

| | |
|--|-----|
| A robot positioning problem (Example 12.6) | 587 |
| Robot positioning with self-organised communication (Example 12.8) | 597 |
| Six degrees of separation in robot positioning (Exercise 12.10) | 600 |
| Communication structure of a robot swarm (Example 12.10) | 608 |

• Multirotor swarms

| | |
|---|----------|
| Coupling analysis of a multirotor (Example 6.3) | 195 |
| Actuator fault diagnosis of a multirotor (Example 11.13) | 551 |
| Formation control of a multirotor swarm (Exercise 12.4 with solution) | 577, 711 |

Smart buildings

| | |
|---|----------|
| Transfer function of a tall building (Example 6.11) | 229 |
| Tuned mass dampers in tall buildings (Section 7.5.4) | 295 |
| Unobservability of the wind oscillations (Example 7.17) | 299 |
| Decomposition of the Taipei 101 model (Exercise 7.12 with solution) | 301, 672 |

Electrical systems and networks

| | |
|--|-----|
| Testing a logic circuit (Exercise 4.2) | 110 |
|--|-----|

• Electrical power networks

| | |
|---|----------|
| Decomposition of an electrical power network (Exercise 6.2 with solution) | 197, 657 |
| Model of a DC microgrid (Exercise 8.3 with solution) | 323, 676 |
| Load flow in an electrical power network (Example 8.8) | 353 |
| Extension of the load flow in an electrical power network (Exercise 8.14) | 366 |

• Control of DC motors

| | |
|---|----------|
| Hierarchical modelling of a DC motor (Example 6.1) | 188 |
| Signal-flow graph of a controlled DC motor (Example 6.7) | 222 |
| Structural analysis of a DC motor (Exercise 11.4 with solution) | 518, 703 |
| Observability of a DC motor (Exercise 7.2 with solution) | 271, 663 |

| | |
|---|----------|
| • Electrical network analysis | |
| Signal-flow graph of a series resonant circuit (Example 6.8) | 223 |
| Application of Mason's formula to a series resonant circuit (Exercise 6.8 with solution) .. | 234, 660 |
| Electrical network analysis (Section 8) | 305 |
| Branch current analysis of an electrical circuit (Exercise 8.2) | 323 |
| Node voltage analysis of an RLC network (Example 8.5) | 331 |
| Determination of the effective resistance by using the Y- Δ transformation (Example 8.6) | 338 |
| Determination of the effective resistance by applying the Kron reduction (Example 8.7) .. | 344 |
| Kron reduction of a series and a star connection (Exercise 8.5 with solution) | 344, 678 |
| Effective resistance of a random electrical network (Exercise 10.6 with solution) | 489, 700 |
| Controllability and observability of an RC circuit (Exercise 7.8 with solution) | 278, 668 |
| Transfer function of a band-pass filter (Exercise 8.4 with solution) | 334, 678 |
| Well-determined and under-determined parts of a DC circuit (Exercise 11.2 with solution) | 518, 701 |
| Structural analysis of an electrical circuit (Exercise 11.3 with solution) | 518, 702 |
| • Synchronisation of oscillator networks | |
| Synchronised oscillator circuits (Example 12.1) | 565 |
| Synchronous trajectory of two harmonic oscillators (Example 12.2) | 568 |
| Oscillator synchronisation with different communication structures (Exercise 12.2) | 576 |
| • Structural analysis of a lamp circuit | |
| Causal reasoning about a lamp circuit (Example 4.9) | 129 |
| Diagnostic reasoning about a lamp circuit (Example 4.11) | 135 |
| Probabilistic model of a lamp circuit (Exercise 4.8 with solution) | 141, 647 |
| Structural model of a lamp circuit (Example 11.1) | 497 |
| Strong components of the causal graph for a lamp circuit (Example 11.5) | 513 |
| Exceptional lamp circuits (Example 11.6) | 517 |
| Causal reasoning with bipartite graphs (Exercise 11.5 with solution) | 519, 704 |
| • Analysis of rear lamps | |
| Logical model of two rear lamps (Example 4.1) | 103 |
| Model-based diagnosis of the two rear lamps (Example 4.2) | 107 |
| Diagnosis of the two rear lamps (Exercise 4.1) | 109 |
| Bayesian network of the two rear lamps (Example 4.7) | 121 |
| Causality of the two rear lamps (Example 11.2) | 499 |
| Structural consistency of the model of the two rear lamps (Example 11.3) | 507 |
| Acyclic causal graph of the two rear lamps (Example 11.4) | 510 |

| | |
|---|-----|
| Diagnosis of rear lamps (Example 11.9) | 536 |
| Residuals for the two rear lamps (Example 11.10) | 539 |
| Fault detection of the two rear lamps (Exercise 11.9) | 541 |

Daily-life examples

| | |
|---|----------|
| Planning a route through Amsterdam (Exercise 3.4) | 68 |
| Can I reach you for a telephone conversation? (Exercise 9.8 with solution) | 430, 692 |
| How <i>Google</i> sorts its search results (Example 9.13) | 428 |
| Verification of the control of a cash dispenser (Exercise 3.12 with solution) | 93, 640 |
| Data traffic at a scientific conference (Exercise 8.10) | 364 |
| What is the reason for traffic jams when driving from Croatia to South Germany? (Exercise 8.11 with solution) | 365, 684 |

Notation

Guidelines for symbols. Scalars are denoted by italics ($k_{ij}, y_i(t)$), vectors by lower case boldface letters ($\mathbf{a}, \mathbf{x}(t)$) and matrices by upper case boldface letters (\mathbf{A}, \mathbf{K}). The notation $\mathbf{A} = (a_{ij})$ indicates that the elements of the matrix \mathbf{A} are denoted by a_{ij} with appropriate indices i and j . To avoid complicated notation, the indices of the matrix are sometimes abbreviated as in $\mathbf{A}_{\mathcal{G}} = (a_{ij})$.

\mathbf{k}^T is the transposed vector, \mathbf{I}_r the r -dimensional unity matrix, \mathbf{O} or $\mathbf{O}_{n \times m}$ the zero matrix of reasonable size or of size $(n \times m)$, respectively, and \otimes the symbol of the Kronecker product (cf. Appendix 2). In structured matrices, sometimes the vanishing blocks are suppressed for the clarity of notation. $\text{diag } k_i$ or $\text{diag } (k_1, k_2, \dots, k_n)$ are abbreviations of a diagonal matrix with the elements k_1, k_2, \dots, k_n in its main diagonal. $\mathbf{0}$ and $\mathbf{1}$ symbolise vectors with all elements equal to zero or one, respectively.

The relations $>, \leq, >, \geq$ apply element by element for vectors and matrices of the same size. Hence, $\mathbf{A} \geq \mathbf{B}$ means $a_{ij} \geq b_{ij}$ for all i, j . A vector \mathbf{x} or a matrix \mathbf{A} is called nonnegative, if all elements are nonnegative. This fact is written as $\mathbf{x} \geq \mathbf{0}, \mathbf{x} \geq 0, \mathbf{A} \geq \mathbf{O}$ or $\mathbf{A} \geq 0$. In the strict versions $\mathbf{A} > \mathbf{O}$ and $\mathbf{A} > 0$, all elements of \mathbf{A} are positive. Likewise, the sign operator $|\cdot|$ applies element by element to matrices and vectors like in the term $|\mathbf{M}|$, which means $|\mathbf{M}| = (|m_{ij}|)$. A system Σ represented by a state-space model with the matrix \mathbf{A} and the vectors \mathbf{b} and \mathbf{c}^T are sometimes abbreviated as $\Sigma = (\mathbf{A}, \mathbf{b}, \mathbf{c}^T)$.

Sets are denoted by calligraphic letters like \mathcal{N} . $|\mathcal{N}|$ denotes the cardinality (number of elements) of the set \mathcal{N} . The difference set $\bar{\mathcal{V}} = \mathcal{V} \setminus \mathcal{S}$ includes all elements of \mathcal{V} that are not elements of \mathcal{S} .

Stochastic variables are denoted by upper-case letters like Z or $Z(k)$.

The symbol \circ represents a boolean operation like in eqn. (9.22) or the concatenation operator as in eqn. (9.26). If it should be emphasised that an equality represents a condition to be satisfied, the equality sign is replaced by $\stackrel{!}{=}$.

Functions of time t are usually denoted by lower-case letters (e. g. $f(t)$) and the corresponding functions in the Laplace domain by the same upper-case letters (e. g. $F(s)$). The Laplace transform is symbolised by $\circ \rightarrow \bullet$ as in the example $f(t) \circ \rightarrow \bullet F(s)$.

|| Important facts are marked on the left-hand side and key formulas are set in frames.

MATLAB®. Short summaries of how to deal with graphs by using MATLAB are given at the end of several chapters and in Appendix 4. Program lines are written in typewriter style. The scripts used to generate figures of this book can be downloaded from the book webpage

www.editionmora.de/gmsc.

Symbols and abbreviations.

| | |
|---|---|
| \mathbf{A} | system matrix in a state-space model |
| $\mathbf{A}_{\mathcal{G}} = (a_{ij}), \widehat{\mathbf{A}}_{\mathcal{G}}$ | adjacency matrix of a graph with the elements a_{ij} ; normalised adjacency matrix |
| \mathbf{A}_{\cup} | adjacency matrix of a bipartite graph |
| \mathbf{b}, \mathbf{c}^T | elements of a state-space model |
| d_i | degree of vertex i ; |
| \mathbf{D} | degree matrix (5.13) |
| \mathcal{E} | set of edges |
| \mathbf{e}_i | canonical basis vector $\mathbf{e}_i = (0, \dots, 0, 1, 0, \dots, 0)^T$ with a single element 1 in the i -th position |
| $\mathcal{G}, \vec{\mathcal{G}}, \mathcal{G}^{\cup}$ | undirected graph; directed graph; bipartite graph |
| k, \mathbf{K} | feedback gain; feedback matrix |
| k | time index of discrete-event systems |
| $\mathbf{L}, \widehat{\mathbf{L}}$ | graphs: Laplacian matrix (5.19); normalised Laplacian matrix systems: coupling matrix |
| M | number of edges |
| N | graph: number of vertices systems: number of subsystems |
| \mathcal{N}_i | set of neighbours of vertex i |
| \mathbf{N} | incidence matrix of a graph |
| p | connection probability |
| \mathbf{p} | state (9.15) of a nondeterministic automaton or state (9.36) of a Markov chain |
| P_i, P_0 | agent, leader agent |
| $P(i-j), P(i \rightarrow j)$ | path between i and j (undirected graphs) or from i towards j (directed graphs) |
| \mathbf{P} | permutation matrix |
| $\mathcal{P}, \mathcal{P} \cup \mathcal{Q}$ | set of propositions; vertex sets of bipartite graphs |
| $\mathbb{R}, \mathbb{R}^+, \mathbb{R}^n$ | set of real numbers; set of positive real numbers; n -dimensional vector space |
| \mathcal{R}, \mathbf{R} | reachability set (Section 2.2.2); reachability matrix (5.36) |
| t | continuous time |
| $u, \mathbf{u}, y, \mathbf{y}$ | input, input vector, output and output vector of a system |

| | |
|--|---|
| \mathcal{V} | set of vertices |
| $\boldsymbol{w}^T, \hat{\boldsymbol{w}}$ | left eigenvector (5.23) and normalised left eigenvector of a Laplacian matrix belonging to the vanishing eigenvalue $\lambda_1 = 0$ |
| $\boldsymbol{x}, \boldsymbol{x}_0$ | state vector; initial state |
| λ_1 | smallest eigenvalue of a Laplacian matrix ($\lambda_1 = 0$) |
| λ_2 | second smallest eigenvalue of a Laplacian matrix |
| $\lambda_{\min}, \lambda_{\max}$ | minimum or maximum eigenvalue of a symmetric matrix |
| $\sigma(t)$ | step function |
| $\Sigma, \overline{\Sigma}$ | system; overall system |
| $2^{\mathcal{Z}}$ | power set of \mathcal{Z} (set of all subsets) |
| \neg | “not” (negation) |
| \wedge | “and” (conjunction) |
| \vee | “or” (disjunction) |
| \Rightarrow | “implies” (implication) |
| \Leftrightarrow | “is equivalent to” (equivalence) |
| \models | entailment (3.34) |
| \vdash | derivation (3.36) |
| \sim, \simeq | isomorphism of graphs (p. 15) |
| ARR | analytical redundancy relation |
| ATMS | assumption-based truth maintenance system |
| EMF | electromotive force |
| i. i. d. | independently identically distributed |
| I/O | input-output (e. g. I/O behaviour) |
| MSO | minimum structurally over-determined (set) |
| PSO | proper structurally over-determined (set) |

1

Systems theory meets graph theory

This chapter summarises the motivation and the limitations of the structural analysis of dynamical systems.

1.1 Graph representations of dynamical systems

A *graph* is a mathematical construct that represents a set of objects together with a relation among the objects. Formally, it is written as a pair of sets like

$$\mathcal{G} = (\mathcal{V}, \mathcal{E})$$

with \mathcal{V} denoting a set of vertices and \mathcal{E} a set of edges connecting pairs of vertices. In engineering fields, graphs are associated with a graphical drawing like the one in the left part of Fig. 1.1.

This abstract definition opens the way towards a broad field of applications, in which the vertices and the edges of a graph have quite different interpretations. In Fig. 1.1, the block diagram of an interconnected system, a matrix, and an electrical circuit lead to the same graph. These three applications of graph theory will be extended throughout this book to arrive at the variety of graphs summarised in Table 1.1 on p. 3.

In its basic form, a graph \mathcal{G} represents a binary relation

$$\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V},$$

where two vertices j and i are connected by an edge ($j-i$) if and only if the pair (j, i) complies with a specified relationship. There are several extensions, for example directed graphs $\vec{\mathcal{G}}$, where the edges have an orientation ($j \rightarrow i$), or weighted graphs, in which each edge ($j \xrightarrow{a_{ij}} i$) is

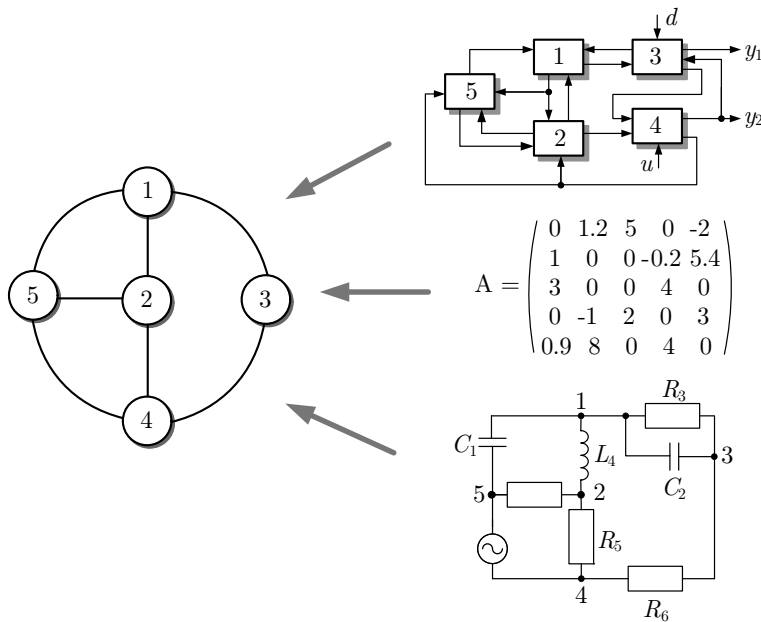


Fig. 1.1: Graph representation of an interconnected system, a matrix, and an electrical circuit

associated with a label a_{ij} . Further extensions allow higher-order relations like in the inference graph, where the edges appear as pairs and connect two predecessor vertices with one successor vertex. The graphs in the table below the double horizontal lines need such extensions.

The question how a graph-theoretical interpretation helps to understand and to solve a systems-theoretical problem has two answers:

- **Graph theory leads to a comprehensible visual representation of a problem.** Engineers are more comfortable with drawings than with texts or long equations as a means of developing and communicating ideas. Graphs provide such an intuitively appealing way to represent relations among signals and components of a dynamical system. For example, it is difficult to understand eqn. (6.53), which is used in Mason's formula to get a transfer function. However, with its graph-theoretical interpretation the required determinant can be found by simply determining sets of disjoint cycles in a graph and multiplying the gains. Block diagrams, electrical circuit diagrams, networks of subsystems likewise enhance the insight into a technological issue and contribute to its solution.
- **Graph-theoretical methods contribute to solve analysis and design problems.** In a deeper application of graph theory, graphs are more than a pictorial representation and lead to effective solution methods. For example, graph search algorithms are essential to solve decision problems or to find paths for a robot to a new configuration. Bayesian networks represent some kind of stochastic independence and lead to a structured way for processing probabilistic information.

Table 1.1. Graphs representing systems and control configurations

| Name | | Interpretation | eqn. |
|--|--|---|---------|
| State-space representation of a problem | $\mathcal{G} = (\mathcal{Z}, \mathcal{E})$ | \mathcal{Z} – set of problem states \mathcal{E} – operator applications | (3.2) |
| Discretised configuration space of a robot | $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ | \mathcal{V} – set of discretisation points \mathcal{E} – robot movements | (3.14) |
| Bayesian network | $\vec{\mathcal{G}} = (\mathcal{P}, \mathcal{E}, \text{Prob})$ | \mathcal{P} – set of propositions \mathcal{E} – set of stochastic dependencies Prob – conditional probabilities | (4.13) |
| Coupling graph | $\vec{\mathcal{G}} = (\mathcal{V}, \mathcal{E})$ | \mathcal{V} – set of subsystems \mathcal{E} – set of direct couplings | (6.1) |
| Algebraic inter-connection graph | $\vec{\mathcal{G}}_s = (\mathcal{V}, \mathcal{E})$ | \mathcal{V} – set of subsystems \mathcal{E} – set of algebraic couplings | (6.29) |
| Signal-flow graph | $\vec{\mathcal{G}} = (\mathcal{V}, \mathcal{E}, \mathcal{T})$ | \mathcal{V} – set of signals \mathcal{E} – set of signal couplings \mathcal{T} – set of transmission labels | (6.43) |
| Structure graph | $\vec{\mathcal{G}}_s = (\mathcal{V}, \mathcal{E})$ | \mathcal{V} – inputs, state variables and outputs \mathcal{E} – set of signal couplings | (7.23) |
| Circuit graph | $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ | \mathcal{V} – set of nodes of an electrical circuit \mathcal{E} – set of circuit branches | (8.1) |
| Flow network | $\vec{\mathcal{G}} = (\mathcal{V}, \mathcal{E}, \mathcal{C})$ | \mathcal{V} – set of vertices of a flow system \mathcal{E} – set of directed edges \mathcal{C} – set of capacity constraints | (8.50) |
| Automaton graph | $\vec{\mathcal{G}} = (\mathcal{Z}, \mathcal{E})$ | \mathcal{Z} – state set \mathcal{E} – set of state transitions | (9.8) |
| Communication graph | $\vec{\mathcal{G}} = (\mathcal{V}, \mathcal{E}, \mathcal{A}_G)$ | \mathcal{V} – set of agents \mathcal{E} – set of communication links \mathcal{A}_G – set of labels | (12.3) |
| Delay graph | $\vec{\mathcal{G}}_\Delta = (\mathcal{V}, \mathcal{E}, \hat{\mathcal{A}}_G, \Delta)$ | \mathcal{V} – set of agents \mathcal{E} – set of communication links $\hat{\mathcal{A}}_G$ – normalised edge labels Δ – set of agent delays | (12.51) |
| Effective communication graph | $\vec{\mathcal{G}} = (\mathcal{V}, \mathcal{E})$ | \mathcal{V} – set of agents \mathcal{E} – set of communication links | (12.69) |
| Triangulation | $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ | \mathcal{V} – set of points \mathcal{E} – set of edges representing triangles | (12.83) |
| Inference graph | $\vec{\mathcal{G}} = (\mathcal{V}, \mathcal{E})$ | \mathcal{V} – set of logical formulas \mathcal{E} – paired edges showing resolution steps | (3.48) |
| ATMS graph | $\vec{\mathcal{G}} = (\mathcal{V}, \mathcal{E})$ | \mathcal{V} – set of logical formulas \mathcal{E} – edge bundles indicating justifications | (4.2) |
| Bipartite structure graph | $\mathcal{G}^\cup = (\mathcal{Z} \cup \mathcal{C}, \mathcal{E})$ | \mathcal{Z} – set of variables \mathcal{C} – set of constraints \mathcal{E} – variable/constraint couplings | (11.2) |
| Causal graph | $\vec{\mathcal{G}} = (\mathcal{V}, \mathcal{E})$ | \mathcal{V} – set of variables and constraints \mathcal{E} – causal data flow | (11.5) |

1.2 What graph theory offers

Graph theory has developed numerous powerful analysis tools, which can be exploited in systems and control theory in an application-oriented way. Once an analysis or design problem has been transferred to a graph problem, a solution is obtained efficiently. This approach is illustrated by the following paragraphs and will be elaborated in detail in this book.

Various representation forms. Graph theory offers, besides the ordinary directed or undirected graphs, a variety of representation forms of binary and higher-order relations among objects, which may be associated with weights or labels. In random graphs, the existence of edges is subject to chance. Each application field can select its most appropriate representation form.

Search. Many graph-theoretical problems are posed by setting requirements on a set of vertices or a set of edges and the question remains how to find sets that satisfy these requirements. For this reason, numerous graph search methods have been elaborated, which can be applied to similar systems-theoretical problems, where likewise trail-and-error methods are the only way of solution. Subsets of vertices or edges are selected and it is tested whether these subsets satisfy the requirements. Hence, many graph-theoretical problems are combinatorial and can be solved only by an efficient enumeration of all possible candidates of a solution. It is important to use the insight into the problem given by a graph to guide the enumeration or search for a solution and, hence, reduce its complexity. The A^* search algorithm developed in the field of artificial intelligence is one of the application-oriented extensions of graph search that prefer promising search directions.

Graph decomposition. An important problem is to decompose a graph into components, particularly into components with strongly connected vertices, and to abstract a graph condensation. The relevant methods can be used whenever a systems-theoretical problem has been posed as a decomposition problem, for example in the analysis of interconnected systems or in the classification of the states of discrete-event systems as transient or recurrent states. If the couplings within a system are sparse, the overall system can be teared into several groups of subsystems with uni-directional couplings and analysis and design methods can be applied sequentially to these groups. For graphs with two sets of vertices, the DM decomposition can be used in the analysis of sets of linear equations or coupled systems. For fault diagnosis, this decomposition yields criteria for the diagnosability of faults based on the available measurements.

Reachability analysis. A standard problem in graph theory is to find a path from an initial vertex towards a goal vertex. A reachability analysis reveals to which vertices such a path exists. It is accomplished either by the application of search methods or in an algebraic manner by determining the powers of the adjacency matrix. These methods lay the basis for solving logical problems by theorem proving or for finding trajectories from an initial state to a goal state for robot manipulation or, more generally, for discrete-event systems. To find paths through a transportation network is likewise the basis for determining a maximum flow. The test of generic properties of linear systems exploit the set of reachable states of a system.

Graph cycles. Cycles are paths in a graph that begin and end in the same vertex. They indicate feedback loops in a system. Methods to find such cycles provide the basis for fundamental mathematical relations like those for the structural rank or the determinant of a matrix. They lead to graph-theoretical modelling ways of electrical circuits, in which Kirchhoff's voltage law is related to cycles in the circuit graph. The well-posedness of interconnected systems likewise depends upon properties of cycles in the algebraic interconnection graph.

Isomorphic graphs. Graph isomorphism means that graphs have, in principle, the same structure and properties. It is transferred to the isomorphism and homomorphism of automata, which represent discrete-event systems.

Efficient graph algorithms. Algorithmic graph theory has developed efficient ways of solution even for problems for which no polynomial-time method exists. Small-scale systems used in the examples and exercises of this book lead to simple graphs that can be analysed manually, but elaborate algorithms are necessary for larger systems for which the complexity increases generally in an exponential way. Such algorithms exist.

This advice is illustrated by the two formulation of a sum:

$$\sum_{j=1}^N a_{ij}x_j = \sum_{j \in \mathcal{N}_i} a_{ij}x_j.$$

On the left-hand side the summation goes over all indices, but for all indices i, j with $a_{ij} = 0$ the addends are evaluated, but do not contribute to the result. On the right-hand side, the summation is restricted to the set \mathcal{N}_i of neighbours of the vertex i in a graph. For the corresponding indices $j \in \mathcal{N}_i$ the weightings a_{ij} do not vanish and the addends are really essential for the sum. In the right formulation the graph has simplified the calculation by restricting the summation to the neighbours of i .

Such a complexity reduction is efficient whenever the graph is sparse, which means that it has much less edges than a complete graph. Generally, the complexity does not depend upon the system size, but on the number of interactions that have to be considered simultaneously. Graphs show which interactions matter. Note that not the edges present in a graph, but the missing edges reduce the complexity.

To classify the complexity of graph algorithms exponential complexity is distinguished from polynomial complexity. This subject has attracted particular attention in graph theory, but it is not elaborated in detail here. The complexity of several algorithms are mentioned, but without a deeper analysis.

Unterhaltungsmathematik? (recreational mathematics?) The application of graph theory is promoted by its intuitive interpretation for simple every-day problems. With their combinatorial character societal problems have been a rich source for graph theory as DÉNES KÖNIG¹ pointed out in 1936 in the preface of his famous book [162] on p. 10 and he referred to it as “*Unterhaltungsmathematik*”. Even LEONHARD EULER² was inspired by a daily-life example with the seven bridges of Königsberg to write the first scientific paper on graph theory [91]

¹ DÉNES KÖNIG (1884 – 1944), Hungarian mathematician

² LEONHARD EULER (1707 – 1783), Swiss mathematician

in 1736. Not the problem statements alone, but also some of the most important ways of solution are intuitively understandable in this way.

“Der große heuristische Wert der Graphentheorie liegt eben darin, daß ihre abstrakten Untersuchungen an konkrete räumliche Verhältnisse erinnern.” ([162], S. 18) (The important heuristic value of graph theory lies in the fact that it combines abstract investigations with actual spatial relations.)

Daily-life examples like the search in a labyrinth, games or the decanting problem may be really considered for recreation.

1.3 The importance of the system structure

The notion of the system structure is used in literature with different meaning and, likewise, this book will refer to this notion in connection with diverse graph-theoretical interpretations. Basically, the structure of a system describes which subsystems are interconnected and which are not. For electrical circuits and communication systems this notion of the structure is equivalent to the notion of topology, which is used in these fields to describe the arrangement of the elements of a system. With a similar meaning, a structure graph connects two signals whenever the corresponding coefficient in the model is non-zero. For discrete-event systems, the structural analysis concerns the question into which states represented by the vertices of a graph a system may be driven by an appropriate input and which of these states may appear repeatedly on a state trajectory.

These diverse definitions have in common that a property is said to be structural if it is essentially determined by the set-up of the system and, therefore, valid not only for a single system, but for a whole class of systems. It is rather independent of parameter values. The structure determines the fixed properties, whereas the parameters describe the variation possibilities. That is why many structural properties can be represented and found by means of graph-theoretical representations of the system.

When starting to analyse or to design a dynamical system, it is advisable for several reasons to concentrate on the system structure first:

- **Appropriateness of a model:** The validity of mathematical models is mainly based on the fact that the model reflects the structural properties of a system.
- **Intuitive understanding of dynamical phenomena:** Structural descriptions provide an intuitive view into the information flow within a system. The main behavioural properties can be much easier recognised by a graphical representation than by numerical calculations.
- **Robustness:** Structural investigations can be made without knowing the precise dimensioning of a system. Their results are rather robust with respect to parameter variations, production tolerances or temporal changes.
- **Control structure:** When controlling a system, the choice of the control configuration is essential. If the structure is appropriate, reasonable parameters are quickly found.

In summary, the engineering intuition says: “Structure counts”.

There are two situations in the relation between the structure and the performance of a system. In most applications, the required behaviour of the system is typical for the structure. Small parameter variations do not destroy the system function. Many properties like the observability of a system are generic in this sense (Chapter 7) or the well-determined part of a model are related to the system structure (Chapter 11).

However, some applications require a system performance that is achievable only for specific parameter values, which restrain the effectivity of the system structure. Then the system has no robustness, because any parameter variation changes the performance drastically. For example, the synchronisation of multi-agent systems (Chapter 12) or the state observation (Chapter 7) are possible only for precisely identical agents or models.

It is a common experience that in engineering projects the most fateful errors are made at the beginning. The book has something to do with this experience. If one chooses a wrong structure, it is hopeless to look for appropriate parameters to satisfy a control goal, whereas with the right structure, appropriate parameters are self-evident.

Abstraction. Any structural representation of a system is the result of an abstraction that simplifies the system by ignoring details. Simultaneously, the focus of the investigations is moved from a single system to a system class. Consequently, any structural analysis has to deal with properties of groups of systems. This fact is explained in detail in Chapter 7, where the (numerical) controllability of a single linear systems is replaced, on the structural level of investigations, by the structural controllability of a system class and the relation between both properties is discussed.

1.4 Limitations of structural analysis methods

A natural question asks which properties cannot be found by graph-theoretical means. To show the limitations of any purely structural analysis is indeed a matter of interest of this textbook.

Generally speaking, graph-theoretical methods exceed their limits whenever a system property depends strongly on the parameter values of the system. In addition to this intuitive response there is even a mathematically precise answer: A property is determined by the system structure if all systems not possessing this property lie on an algebraic variety in the parameter space. As Appendix A2.1 explains in more detail, these systems have to lie on a hypersurface in the space of the parameter vector λ that is described by the zeros of some polynomial $p(\lambda) = 0$. An example of such properties is the observability described by a rank condition, and a counterexample is the stability of dynamical systems, which is characterised by a border for the eigenvalue and, thus, is not generic (cf. Section 7.4).

More generally posed, structural properties have to be valid for a large class of systems with the same structure. Any quantitative performance requirement cannot be analysed by graph-theoretical methods, but need to take into consideration, at least with a certain precision, the parameter values of a system. The reason for this failure of graph-theoretical means is given by the fact that structural properties distinguish, in principle, only between present and absent couplings. Many systems have parallel lines of action, which may erase or amplify each other depending on the parameters used. This distinction cannot be found by any structural model.

ϵ -couplings. Experts of a field know which couplings are important and which are not. With the introduction of ϵ -couplings literature has tried to imitate this experience. If an edge label is less than ϵ , it is ignored. By using the graph-theoretical methods for different thresholds for ϵ one gets structures at different levels of abstraction and recognises, which couplings are important and which are not.

However, usually the sole value of an isolated edge does not show the importance of the edge for the system behaviour. It is, for example, the gain of all labels of a cycle that makes the cycle important for the dynamics of the overall system. Therefore, ϵ -coupling methods are only an attempt to mimic the expert's knowledge about important and unimportant signal couplings

Notes and references

There are a lot of textbooks about graph theory, mostly written by mathematicians, with the personal preference of this author on [72, 92, 120, 329]. The computational complexity of graph problems are addressed in these books and in [26, 107], which explain in detail the classes of undecidable problems and NP hard problems.

The German physicist GUSTAV KIRCHHOFF was probably the first who applied graph theory to an important engineering problem when he considered electrical circuits as graphs [157] in 1847.

There are several forms of graphs and graph-theoretical problems that, similarly to the subject of this textbook, promote the engineering applications:

- Petri nets are bipartite graphs with tokens that show the dynamics of discrete-event systems similarly to automaton graphs [197, 275]. Their analysis tools are mainly based on a reachability analysis.
- Bond graphs provide a direct step from the physical structure of a system to its representation by a set of differential and algebraic equations [32, 108]. The edges of such graphs show the bi-directional energy flow through a physical system, which may consist of a combination of mechanical, electrical or hydraulic components.
- Fault trees are popular means to represent the dependency of symptoms upon faults [274]. More generally, fault propagation graphs are used in failure modes and effects analysis (FMEA) to describe the propagation of a fault from a faulty component through the interconnections to other components to identify the resulting failures [24].
- Graph-based modelling is used in many disciplines as the survey of examples in [347, 348] demonstrates. In particular, graph-theoretical methods provide the basis for modelling tools for multibody systems [295].

Reference [117] developed a theory of abstraction, which points to the way to find graph-theoretical representations for dynamical systems. The ϵ decomposition has been proposed in [298].

2

Graphs and graph search

The chapter summarises important notions and properties of graphs and explains three graph search algorithms for the reachability analysis. Further graph-theoretical notions are introduced in Chapters 5 and 10 on algebraic graph theory or bipartite and random graphs, respectively. Section 8.2 investigates cycles in graphs in more depth. The notion of a cut in graphs is defined in Section 8.6.

2.1 Basic notions

2.1.1 Directed graphs

A directed graph (or digraph or oriented graph)

$$\vec{\mathcal{G}} = (\mathcal{V}, \mathcal{E})$$

is a pair of sets with

- $\mathcal{V} = \{1, 2, \dots, N\}$ – set of vertices (or nodes or points)
- $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ – set of directed edges (or arcs or lines).

In order to indicate that the graph is directed, it is marked by an arrow over its name. The number $i \in \mathcal{V}$ of a vertex is also said to be the index of the vertex. The statement $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is called set-theoretical representation of the graph.

A directed edge from the vertex j towards the vertex i is symbolised by $(j \rightarrow i)$ with the arrow indicating the edge direction. j is called the initial vertex of this edge and i the final

vertex and both are called end vertices (or end points). They are *joined* by an edge. The edges may be enumerated with the k -th edge symbolised by

$$e_k = (j_k \rightarrow i_k) \quad \text{with } j_k, i_k \in \mathcal{V}.$$

Then the edge set has the two representations

$$\begin{aligned} \mathcal{E} &= \{(j_1 \rightarrow i_1), (j_2 \rightarrow i_2), \dots, (j_M \rightarrow i_M)\} \\ &= \{e_1, e_2, \dots, e_M\} \end{aligned}$$

with M denoting the number of edges. Correspondingly, the relations $(j_3 \rightarrow i_3) \in \mathcal{E}$ or $e_3 \in \mathcal{E}$ are used to indicate edges of the graph $\vec{\mathcal{G}}$. The vertices may have a loop ($i \rightarrow i$), which is also said to be a self-loop or a self-edge.

The notions of incidence and adjacency characterise neighbouring edges and vertices, where the former notion is a relation between edges and vertices, whereas the latter notion describes a relation among edges or among vertices. Accordingly, an edge is *incident* with its end vertices, and two edges are *adjacent* if the final vertex of one edge is identical to the initial vertex of the other edge. Two vertices are adjacent (or neighbouring) if they are the end vertices of the same edge.

In the graphs used in this book, between any pair of vertices there exists at most one edge. Hence, the graphs are called *simple* or strict. Although the number N of vertices may be large, the investigations concern graphs with a finite number N , which are called *finite* graphs. The number of edges is denoted by M .

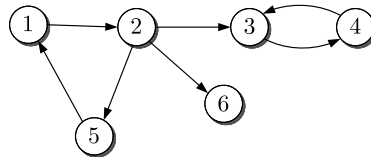


Fig. 2.1: Directed graph

Graphs are drawn as circles representing the vertices and arrows showing the directed edges. The example graph in Fig. 2.1 has the set

$$\mathcal{V} = \{1, 2, 3, 4, 5, 6\}$$

of six vertices and the set of edges

$$\mathcal{E} = \{(1 \rightarrow 2), (2 \rightarrow 3), (2 \rightarrow 5), (2 \rightarrow 6), (3 \rightarrow 4), (4 \rightarrow 3), (5 \rightarrow 1)\}.$$

If the names of the vertices do not matter in an analysis, graphs are sometimes drawn without indicating the vertex names.

The set \mathcal{N}_i of *neighbours* (or predecessors or ancestors) of the vertex i includes all vertices j from which there is a directed edge ($j \rightarrow i$) towards i :

$$\text{Set of neighbours: } \mathcal{N}_i = \{j \mid (j \rightarrow i) \in \mathcal{E}\}, \quad i \in \mathcal{V}. \quad (2.1)$$

If a graph has self-loops, a vertex i may be a neighbour of itself, but the notion of the set of neighbours is generally used only for graphs without self-loops. The number of elements of the set \mathcal{N}_i

$$d_i = |\mathcal{N}_i|, \quad (2.2)$$

which represents the number of neighbours of i , is called the *in-degree* (or in short: degree) of the vertex i . Similarly, the set of successors (or descendants) of the vertex i is

$$\text{Set of successors: } \mathcal{S}_i = \{j \mid (i \rightarrow j) \in \mathcal{E}\}, \quad i \in \mathcal{V} \quad (2.3)$$

and

$$d_{i,\text{out}} = |\mathcal{S}_i| \quad (2.4)$$

is the *out-degree* of the vertex i . Vertices that do not have predecessors are said to be root vertices (or roots), vertices without successors are called leaf vertices (or leaves).

2.1.2 Undirected graphs

For graphs

$$\mathcal{G} = (\mathcal{V}, \mathcal{E})$$

with undirected edges similar notions are used as for directed graphs (Fig. 2.2 (left)). The edges are symbolised by $(i - j) \in \mathcal{E}$ with the understanding that with the edge $(i - j)$ also the edge $(j - i)$ belongs to the edge set \mathcal{E} . The set

$$\mathcal{N}_i = \{j \mid (j - i) \in \mathcal{E}\}, \quad i \in \mathcal{V} \quad (2.5)$$

is said to represent the neighbourhood of the vertex i , whereas the notions of a predecessor or a successor, which have been introduced for directed graphs, are not applicable.

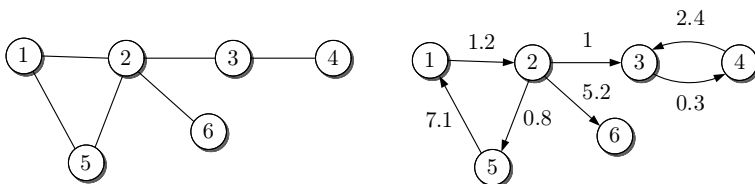


Fig. 2.2: Undirected graph (left) and weighted graph (right)

Graphs with undirected edges can be considered as directed graphs in which every edge $(i - j)$ is replaced by the two edges $(i \rightarrow j)$ and $(j \rightarrow i)$ with opposite directions. On the other hand, directed graphs can be made undirected if all directed edges $(i \rightarrow j) \in \mathcal{E}$ are turned into undirected edges $(i - j)$ and double edges are reduced to a single one to get a simple graph. The resulting undirected graph is called *embedded* in the directed graph or it is said to be the

disoriented graph. As many books on graph theory are concerned with undirected rather than directed graphs, they call the undirected graphs the *ordinary graphs* to emphasise that the edges do not have any direction.

2.1.3 Weighted graphs

In a weighted graph

$$\vec{\mathcal{G}} = (\mathcal{V}, \mathcal{E}, \mathbf{A}_{\mathcal{G}})$$

each edge $(j \xrightarrow{a_{ij}} i) \in \mathcal{E}$ is associated with a real number $a_{ij} > 0$ (Fig. 2.2 (right)) or with the corresponding symbol a_{ij} (cf. Section 5.3), which are the corresponding elements of the $(N \times N)$ -weighting matrix $\mathbf{A}_{\mathcal{G}}$. For $(j \rightarrow i) \notin \mathcal{E}$, the weight is defined to be zero: $a_{ij} = 0$. The definitions (2.2) or (2.4), respectively, of the in-degree and the out-degree of a vertex $i \in \mathcal{V}$ are generalised to be

$$\text{In-degree: } d_i = \sum_{j \in \mathcal{N}_i} a_{ij} \quad (2.6)$$

$$\text{Out-degree: } d_{i,\text{out}} = \sum_{j \in \mathcal{S}_i} a_{ji}. \quad (2.7)$$

Unweighted graphs can be interpreted as weighted graphs in which each edge has the weight $a_{ij} = 1$. For undirected graphs the weights have the property $a_{ij} = a_{ji}$, ($i, j = 1, 2, \dots, N$), which implies that the weighting matrix $\mathbf{A}_{\mathcal{G}}$ is symmetric.

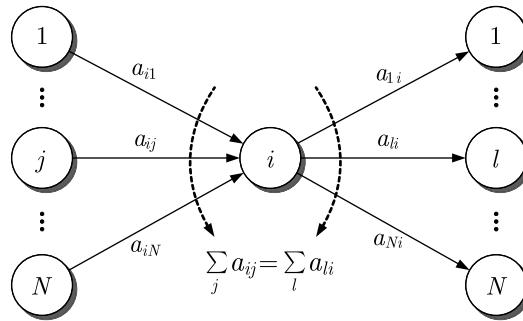


Fig. 2.3: Interpretation of the condition (2.8) for weight-balanced graphs

Balanced graphs. A directed graph $\vec{\mathcal{G}}$ is called *weight-balanced* if for all vertices the in-degree (2.6) equals the out-degree (2.7) or, equivalently, the sum of the weights of all out-going edges is the same as the sum of the weights of all in-coming edges (Fig. 2.3):

$$\text{Weight-balanced graph: } d_i = \sum_{j=1}^N a_{ij} = \sum_{l=1}^N a_{li} = d_{i,\text{out}}, \quad i = 1, 2, \dots, N. \quad (2.8)$$

This requirement can be reformulated as

$$\mathbf{1}^T \mathbf{A}_G = (\mathbf{A}_G \mathbf{1})^T. \quad (2.9)$$

For unweighted graphs, this condition is applied after each edge $(j \rightarrow i) \in \mathcal{E}$ has been associated with the weight $a_{ij} = 1$ and it requires that for all vertices the number of incoming edges is the same as the number of out-going edges.

|| All undirected graphs are weight-balanced,

because for these graphs all incoming edges are also out-going edges.

Example 2.1 *Weight-balanced graphs*

The left graph in Fig. 2.4 is not weight-balanced, because the requirement that all vertices have the same number of in-coming and out-going edges is violated, for example, for the vertex 4, which has two in-coming edges but only one out-going edge. In the weighting matrix

$$\mathbf{A}_1 = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix},$$

the row sum and the column sum differ for the vertex 4 and, hence, violates eqn. (2.9).

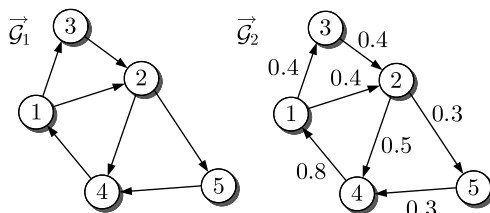


Fig. 2.4: Two example graphs

The graph \vec{G}_2 is weight-balanced, because for all vertices the weight sum of all in-coming edges is the same as the sum of all out-going edges as required in eqn. (2.8). The weighting matrix

$$\mathbf{A}_2 = \begin{pmatrix} 0 & 0 & 0 & 0.8 & 0 \\ 0.4 & 0 & 0.4 & 0 & 0 \\ 0.4 & 0 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 & 0.3 \\ 0 & 0.3 & 0 & 0 & 0 \end{pmatrix},$$

satisfies eqn. (2.9):

$$\mathbf{1}^T \mathbf{A}_2 = (0.8 \ 0.8 \ 0.4 \ 0.8 \ 0.3) = (\mathbf{A}_2 \mathbf{1})^T. \quad \square$$

2.1.4 Specific graphs

The following graphs with particular properties will be considered later (Fig. 2.5):

- **Complete graphs:** A directed graph is called complete if for all pairs (j, i) with $j \neq i$ there exists a directed edge $(j \rightarrow i)$.
- **Path graphs:** A single path like in the top part of Fig. 2.5 is denoted as path graph or arc.
- **Ring graphs:** A graph consisting of a single cycle is called ring graph (or circuit or cycle graph).
- **Regular graphs:** A graph is called regular or k -regular if all vertices have the same degree k . Ring graphs and the complete graphs are examples of regular graphs with $k = 1$ or $k = N - 1$, respectively (Fig. 2.5).

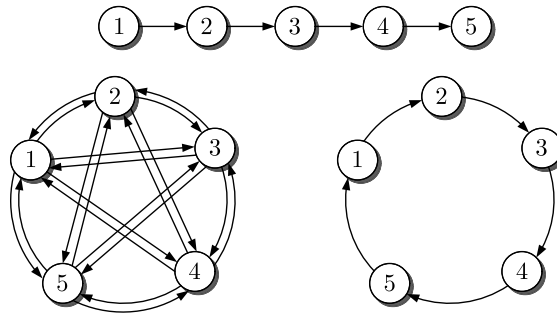


Fig. 2.5: Specific graphs: path graph, complete graph, ring graph

Homomorphic and isomorphic graphs. Graphs should represent relations among a set of objects symbolised by the vertices. One considers two graphs to be identical if they describe the same relation. To formalise what “identical” means, consider the two graphs

$$\vec{\mathcal{G}}_1 = (\mathcal{V}_1, \mathcal{E}_1) \quad \text{and} \quad \vec{\mathcal{G}}_2 = (\mathcal{V}_2, \mathcal{E}_2)$$

and a mapping

$$P : \mathcal{V}_1 \rightarrow \mathcal{V}_2. \quad (2.10)$$

The mapping P is said to be a *homomorphism* if it retains the adjacency of the vertices as follows: If two vertices $i, j \in \mathcal{V}_1$ are adjacent in $\vec{\mathcal{G}}_1$ then the corresponding vertices $P(i), P(j) \in \mathcal{V}_2$ are adjacent in $\vec{\mathcal{G}}_2$. For digraphs this relation is represented by

$$(i \rightarrow j) \in \mathcal{V}_1 \quad \Rightarrow \quad (P(i) \rightarrow P(j)) \in \mathcal{V}_2$$

and a similar expression describes the homomorphism of undirected graphs. Then the graph $\vec{\mathcal{G}}_2$ is said to be the homomorphic image of $\vec{\mathcal{G}}_1$. $\vec{\mathcal{G}}_2$ may be a graph with a smaller number of vertices than $\vec{\mathcal{G}}_1$.

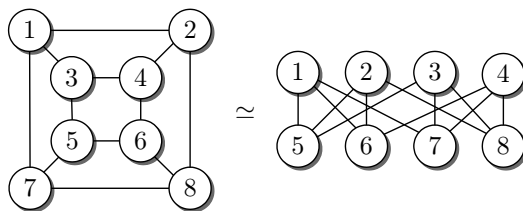


Fig. 2.6: Isomorphic graphs
(What is the mapping P used in eqn. (2.10) for these graphs?)

If the mapping (2.10) is bijective (or a one-to-one correspondence), it is said to be an *isomorphism* and the graphs are called isomorphic (written as $\vec{\mathcal{G}}_1 \simeq \vec{\mathcal{G}}_2$). Isomorphic graphs distinguish only with respect to the enumeration of their vertices. For weighted graphs, the isomorphism also requires that the corresponding edges have the same weight. Figure 2.6 shows an example of isomorphic undirected graphs. Roughly speaking, isomorphic graphs have the same structure and distinguish only in the names of the vertices. Many graph properties are invariant with respect to a renumbering of the vertices. For example, the property of a graph to be strongly connected in the sense introduced in Section 2.2.4 does not depend upon the enumeration of the vertices.

Reversed graph and complement graph. If in a directed graph $\vec{\mathcal{G}}$ the direction of all edges are reversed, the reversed graph (or converse graph or transposed graph) is obtained, which is denoted by $\overleftarrow{\mathcal{G}}$ with the reversed direction of the arrow or by $\vec{\mathcal{G}}^T$. The latter symbol is motivated by the fact that the adjacency matrix introduced in Section 5.1.1 has to be transposed to change the edge directions. The set of successors of $\vec{\mathcal{G}}$ becomes the set of neighbours (predecessors) of $\overleftarrow{\mathcal{G}}$ and vice versa.

The complement graph of $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ (or inverse graph) is the graph $\bar{\mathcal{G}} = (\mathcal{V}, \bar{\mathcal{E}})$ with the same set \mathcal{V} of vertices, in which any pair of vertices is adjacent if and only if these vertices are not adjacent in \mathcal{G} . Hence, the relation $\bar{\mathcal{E}} = (\mathcal{V} \times \mathcal{V}) \setminus \mathcal{E}$ holds.

2.2 Reachability analysis

2.2.1 Paths and cycles

A *path* in a directed graph $\vec{\mathcal{G}}$ is a sequence of adjacent edges, which is written as the sequence of its vertices in the following form:

$$P(i_0 \rightarrow i_e) = (i_0 \rightarrow i_1 \rightarrow \dots \rightarrow i_e). \quad (2.11)$$

i_0 is the initial vertex of the path and i_e the final vertex.

This notion of a path allows that vertices appear more than once in the sequence (2.11). Note that the notion of a path is not uniformly used in literature. Some authors require that a path visits each vertex at most once and call the sequence of adjacent edges with possibly

multiple visits of some vertices a walk or a chain. For the path definition used here, a path that visits any vertex at most once is called a *simple path* (or self-avoiding path).

Path length. The number of edges is defined to be the *length* of a path (2.11) and denoted by

$$|P(i_0 \rightarrow i_e)| = e. \quad (2.12)$$

The length

$$l_{ij} = \min_{P \in \mathcal{P}(j \rightarrow i)} |P|, \quad i, j = 1, 2, \dots, N, \quad i \neq j \quad (2.13)$$

of a shortest path (or geodesic path) from the vertex j towards the vertex i is called the *distance* from j to i . In eqn. (2.13), $\mathcal{P}(j \rightarrow i)$ is the set of all paths $P(j \rightarrow i)$. Obviously, the relation $l_{ij} = l_{ji}$ holds for undirected graphs. If for a pair (i, j) there does not exist any path $P(j \rightarrow i)$, l_{ij} is not defined (or formally set to ∞).

The maximum of the distances between two vertices is called the *diameter* l_{\max} :

$$l_{\max} = \max_{i, j \in \mathcal{V}} l_{ij}. \quad (2.14)$$

It is the length of the longest geodesic path between any pair of vertices in the graph.

Cycles and cycle families. A *closed path* $P(i_0 \rightarrow i_0)$ has an identical start and end point i_0 . It may visit vertices more than once. If a closed path is a simple path, it is called a *cycle* (or circuit). Note that in the notation (2.11) of a closed path

$$P(i_0 \rightarrow i_0) = (i_0 \rightarrow i_1 \rightarrow \dots \rightarrow i_k \rightarrow i_0) \quad (2.15)$$

the identical start and end vertex of the cycle is mentioned twice, but every edge $(i_j \rightarrow i_{j+1})$, ($j = 0, 1, \dots, k$) and $(i_k \rightarrow i_0)$ appears only once.

The length of a cycle is the number of edges involved. A cycle is called even if it includes an even number of edges. A cycle of length 1 is a self-loop according to the definition mentioned above and it is also called a self-cycle. A graph, which does not have any cycle, is called *cycle-free* or acyclic.

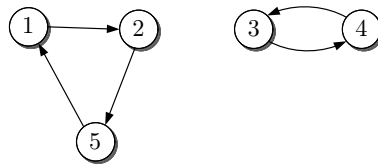


Fig. 2.7: Cycle family of the directed graph of Fig. 2.1

A *cycle family* is a set of cycles that have disjoint vertices and are called disjoint cycles. For example, the digraph in Fig. 2.1 has the cycle family

$$\{(1 \rightarrow 2 \rightarrow 5 \rightarrow 1), (3 \rightarrow 4 \rightarrow 3)\}$$

(Fig. 2.7). The number of vertices appearing in a cycle family is said to be the *width* (or length) of the cycle family. A *spanning cycle family* is a cycle family that includes all vertices of the graph. For a graph with N vertices, the width of a spanning cycle family is N . The graph in Fig. 2.1 does not have a spanning cycle family.

Section 8.2 will introduce the notions of fundamental cycles and the cycle space of graphs and give a method to find both.

2.2.2 Reachability sets

A vertex $j \in \mathcal{V}$ is said to be *reachable* (or accessible) from the vertex $i \in \mathcal{V}$ if there exists a path $P(i \rightarrow j)$. This definition uses the convention that for all vertices $i \in \mathcal{V}$ a trivial path $P(i \rightarrow i)$ of length 0 exists and, hence, each vertex is reachable from itself. The set of all vertices that are reachable from $i \in \mathcal{V}$ is denoted by

$$\mathcal{R}(i) = \{j \in \mathcal{V} \mid \exists P(i \rightarrow j)\}$$

and is said to be the *reachability set* of i . It can be determined recursively for a directed graph $\vec{\mathcal{G}} = (\mathcal{V}, \mathcal{E})$ as follows. Starting with the initial set

$$\mathcal{R}^0 = \{i\}$$

that includes only the vertex i , the algorithm generates a sequence $\mathcal{R}^1, \mathcal{R}^2, \dots, \mathcal{R}^{N-1}$ of sets, in which the set \mathcal{R}^{k+1} is obtained from \mathcal{R}^k by including all new vertices that are found along edges starting in \mathcal{R}^k . The *image* of a subset $\tilde{\mathcal{V}} \subseteq \mathcal{V}$ denotes the set of successors of all vertices in $\tilde{\mathcal{V}}$ and is represented by

$$\text{Im}(\tilde{\mathcal{V}}) = \left\{ j \in \mathcal{V} \mid \exists i \in \tilde{\mathcal{V}} : (i \rightarrow j) \in \mathcal{E} \right\} = \cup_{i \in \tilde{\mathcal{V}}} \mathcal{S}_i$$

(cf. eqn. (2.3)). With this notion, the recursive algorithm is described by

$$\mathcal{R}^{k+1} = \mathcal{R}^k \cup \text{Im}(\mathcal{R}^k), \quad k = 0, 1, \dots, N-2 \quad (2.16)$$

$$\mathcal{R}(i) = \mathcal{R}^{N-1}. \quad (2.17)$$

It generates a sequence of sets with the following monotonicity property:

$$\begin{aligned} \mathcal{R}^0 &\subseteq \mathcal{R}^1 \subseteq \dots \subseteq \mathcal{R}^{N-1} \\ \mathcal{R}^k &= \mathcal{R}^{N-1}, \quad k \geq N-1. \end{aligned}$$

The recursion stops at $k = N-1$, because the longest simple path in a graph with N vertices has a length that does not exceed $N-1$. If, as usual, in an earlier recursion step \tilde{k} the condition $\mathcal{R}^{\tilde{k}+1} = \mathcal{R}^{\tilde{k}}$ is satisfied, the recursion can be stopped with the result

$$\mathcal{R}(i) = \mathcal{R}^{\tilde{k}}.$$

For undirected graphs, this recursion is used with the agreement that with an edge $(j-l)$ also the edge $(l-j)$ belongs to \mathcal{E} .

Reachability trees. For a specific vertex $i \in \mathcal{V}$, paths $P(i \rightarrow j)$ for all $j \in \mathcal{R}(i)$ can be selected such that the union of these paths represents a tree (for the definition of trees cf. Section 2.2.3). The resulting graph is said to be a *reachability tree* and it is denoted by \mathcal{B} . For a given vertex i , the reachability tree \mathcal{B} is not unique.

If there exists a vertex $r \in \mathcal{V}$ such that all other vertices are reachable from r

$$\mathcal{R}(r) = \mathcal{V},$$

then the reachability tree is a spanning tree and r is said to be the root vertex.

2.2.3 Subgraphs and union of graphs

A subgraph $\tilde{\mathcal{G}} = (\tilde{\mathcal{V}}, \tilde{\mathcal{E}})$ of a (directed or undirected) graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a graph formed by subsets

$$\tilde{\mathcal{V}} \subseteq \mathcal{V} \quad \text{and} \quad \tilde{\mathcal{E}} \subseteq \mathcal{E}$$

provided that

$$\tilde{\mathcal{E}} \subseteq \tilde{\mathcal{V}} \times \tilde{\mathcal{V}} \tag{2.18}$$

holds. Equation (2.18) requires that the vertex set $\tilde{\mathcal{V}}$ has to include all vertices that appear as end vertices of the edges in $\tilde{\mathcal{E}}$. However, $\tilde{\mathcal{V}}$ may have additional elements. A spanning subgraph $\tilde{\mathcal{G}}$ of \mathcal{G} is a subgraph that includes all vertices of \mathcal{G} : $\tilde{\mathcal{V}} = \mathcal{V}$. In particular

$$\tilde{\mathcal{G}} = (\mathcal{V}, \emptyset),$$

which does not have any edge, is a spanning subgraph of $\mathcal{G} = (\mathcal{V}, \mathcal{E})$.

Two specific subgraphs will appear in the succeeding chapters:

- **Cycle families** are subgraphs. For the graph $\overrightarrow{\mathcal{G}} = (\mathcal{V}, \mathcal{E})$ depicted in Fig. 2.1, Fig. 2.7 shows a cycle family as the subgraph $\tilde{\mathcal{G}} = (\tilde{\mathcal{V}}, \tilde{\mathcal{E}})$ with

$$\begin{aligned} \tilde{\mathcal{V}} &= \{1, 2, 3, 4, 5\} = \mathcal{V} \\ \tilde{\mathcal{E}} &= \{(1 \rightarrow 2), (2 \rightarrow 5), (5 \rightarrow 1), (3 \rightarrow 4), (4 \rightarrow 3)\} \subset \mathcal{E}. \end{aligned}$$

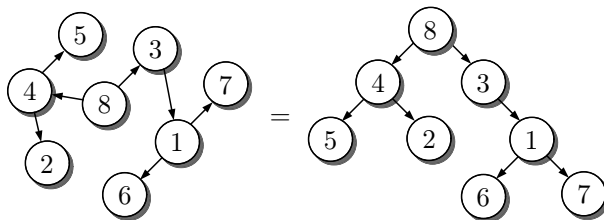


Fig. 2.8: A tree

- **Trees** are subgraphs with the property that there is a vertex r , which is denoted as the root vertex, such that the graph includes for all other vertices $i \neq r$ only a unique path $P(r \rightarrow i)$. If a tree includes all vertices of the original graph, it is called a *spanning tree* (or rooted spanning tree or rooted out-branching). Trees are generally drawn in a rooted manner like in the right part of Fig. 2.8 to show that they introduce a hierarchical structure into the vertex set.

If a subgraph is extracted from a larger graph, the first step is usually to reduce the set of vertices from \mathcal{V} to $\tilde{\mathcal{V}}$. Accordingly, all edges are removed that do not have both end vertices in $\tilde{\mathcal{V}}$. It is taken for granted that in all graph operations used in this book with the removal of some vertices all edges are deleted that are incident to these vertices.

Union of graphs. For two directed graphs

$$\vec{\mathcal{G}}_1 = (\mathcal{V}_1, \mathcal{E}_1) \quad \text{and} \quad \vec{\mathcal{G}}_2 = (\mathcal{V}_2, \mathcal{E}_2)$$

the graph union is defined to be the graph

$$\vec{\mathcal{G}} = (\mathcal{V}_1 \cup \mathcal{V}_2, \mathcal{E}_1 \cup \mathcal{E}_2),$$

which consists of the union of the vertex sets and the union of the edge sets of its components. If the sets defining $\vec{\mathcal{G}}_1$ and $\vec{\mathcal{G}}_2$ are disjoint, the graph union is simply a composition of two independent graphs. If $\vec{\mathcal{G}}_1$ and $\vec{\mathcal{G}}_2$ have the same vertex set, the graph union is the directed graph that includes all edges of its components.

2.2.4 Strongly connected components

An important property of directed graphs refers to the fact that pairs of vertices are mutually reachable.

Definition 2.1 (Strongly connected vertices and graphs)

Two vertices $i, j \in \mathcal{V}$ are said to be strongly connected, if i is reachable from j and j is reachable from i . If any pair of vertices of a graph is strongly connected, the graph is said to be strongly connected (or irreducible).

The property to be strongly connected is an equivalence relation and induces, for some integer q , a partition of the vertex set \mathcal{V} into the equivalence classes \mathcal{V}_i , ($i = 1, 2, \dots, q$)

$$\mathcal{V} = \cup_{i=1}^q \mathcal{V}_i \quad \text{with} \quad \mathcal{V}_i \cap \mathcal{V}_j = \emptyset, \quad i \neq j \quad (2.19)$$

(cf. eqn. (A2.1)) such that any two vertices are strongly connected if and only if they belong to the same set \mathcal{V}_i . If a vertex i is not included in any cycle of the graph, it is an isolated vertex and the only element of the equivalent class $\{i\}$, which is called an acyclic equivalence class.

If all vertices of a graph $\vec{\mathcal{G}}$ are strongly connected, eqn. (2.19) holds with $q = 1$ and the graph $\vec{\mathcal{G}}$ is strongly connected. Otherwise, the vertex set \mathcal{V} can be partitioned into q subsets such that all the corresponding subgraphs

$$\vec{\mathcal{G}}_i = (\mathcal{V}_i, \mathcal{E}_i), \quad i = 1, 2, \dots, q \quad (2.20)$$

are strongly connected. The graphs $\vec{\mathcal{G}}_i$, ($i = 1, 2, \dots, q$) are said to be the *strongly connected components* of $\vec{\mathcal{G}}$ (or strong components). They have one equivalence class \mathcal{V}_i from eqn. (2.19) as their set of vertices and the edge set \mathcal{E}_i that includes all edges of \mathcal{E} that have both end points in \mathcal{V}_i . Any component with more than one vertex possesses at least one cycle.

In summary:

Any directed graph $\vec{\mathcal{G}}$ is either strongly connected or it can be decomposed into several strongly connected subgraphs. The partition of a graph into its strongly connected components is unique.

The resulting subgraphs (2.20) are disjoint, which is symbolised by $\vec{\mathcal{G}}_i \cap \vec{\mathcal{G}}_j = \emptyset$ and implies $\mathcal{V}_i \cap \mathcal{V}_j = \emptyset$ and $\mathcal{E}_i \cap \mathcal{E}_j = \emptyset$ for $i \neq j$. Note that the union of the subgraphs is a subgraph of the original graph, because the edges with end points in different sets \mathcal{V}_i and \mathcal{V}_j are missing.

A directed graph may possess a spanning tree although it is not strongly connected:

$$\vec{\mathcal{G}} \text{ is strongly connected} \implies \vec{\mathcal{G}} \text{ has a spanning tree.}$$

Graph condensation. If one replaces each strongly connected component (2.20) by a vertex of a new graph, an acyclic digraph results like at the bottom of Fig. 2.9. This new graph is said to be a *condensation* $\vec{\mathcal{G}}_C$ of the original graph (or the condensed graph). Any condensation does not have a strongly connected component with more than a single vertex, but it is a directed acyclic graph. In engineering terms, condensed graphs may be referred to as a series-parallel connection of vertices. With an appropriate enumeration of the vertices of $\vec{\mathcal{G}}_C$, edges exist only from vertices with lower index to vertices with higher index.

Example 2.2 *Strongly connected components of a directed graph*

The directed graph in the upper part of Fig. 2.9 is not strongly connected, but can be decomposed into the three strongly connected components $\vec{\mathcal{G}}_i = (\mathcal{V}_i, \mathcal{E}_i)$, ($i = 1, 2, 3$) indicated by the circles with

$$\mathcal{V}_1 = \{1, 2, 5\}, \quad \mathcal{V}_2 = \{3, 4\}, \quad \mathcal{V}_3 = \{6\}.$$

The set \mathcal{V}_3 is a singleton (an acyclic equivalence class), because the vertex 6 is not strongly connected with any other vertex. The union $\vec{\mathcal{G}}_1 \cup \vec{\mathcal{G}}_2 \cup \vec{\mathcal{G}}_3$ is a subgraph of $\vec{\mathcal{G}}$, because the edges $(2 \rightarrow 3)$ and $(2 \rightarrow 6)$ do not belong to any of these subgraphs.

As the graph has three strongly connected components, its condensation $\vec{\mathcal{G}}_C$ is a graph with three vertices, as the bottom of the figure shows. The condensation points to the important fact that the sets of strongly connected vertices may be connected by an edge or a path, but considered as single vertices they are not strongly connected. Condensations are acyclic graphs.

Although the graph in Fig. 2.9 is not strongly connected, it possesses spanning trees with the root vertex 1, 2 or 5. The spanning tree with the root 1 includes all edges with the exception of $(4 \rightarrow 3)$ and $(5 \rightarrow 1)$. \square

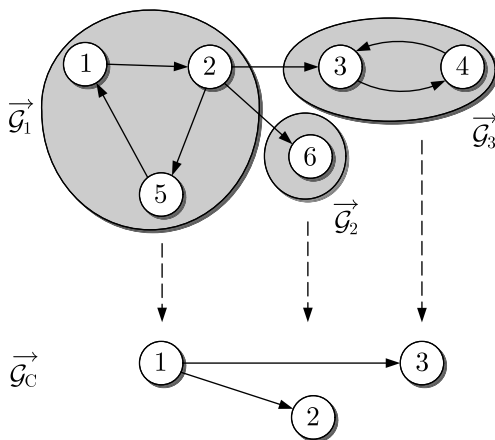


Fig. 2.9: Strongly connected components and condensation of the graph shown in Fig. 2.1

Algorithm to find the graph condensation. Example 2.2 has shown the main steps to be performed for finding the strongly connected components of a graph and the condensed graph. These steps are summarised here in Algorithm 2.1 and it is transferred into matrix operations and a MATLAB implementation in Appendix 4.

Algorithm 2.1 Find strongly connected components and the condensed graph

Given: Digraph $\vec{G} = (\mathcal{V}, \mathcal{E})$

1. Determine the reachability sets $\mathcal{R}(i)$, ($i \in \mathcal{V}$) of the graph \vec{G} and the reachability sets $\mathcal{R}^T(i)$, ($i \in \mathcal{V}$) of the reversed graph \vec{G}^T .
if $\mathcal{R}(i) = \mathcal{V}$ for all $i \in \mathcal{V}$. **then Stop:** The graph is strongly connected. **end**
2. Determine the partition (2.19) of \mathcal{V} into strongly connected sets by investigating the intersections $\mathcal{R}(i) \cap \mathcal{R}^T(i)$. Vertices with the same intersection belong to the same set \mathcal{V}_j in eqn. (2.19).
3. Order the sets \mathcal{V}_i , ($i = 1, 2, \dots, q$) such that edges $(i \rightarrow j) \in \mathcal{E}$ exist only from a set \mathcal{V}_l towards a set \mathcal{V}_m with $l < m$, where \mathcal{V}_l and \mathcal{V}_m are the sets which i and j belong to.
4. Replace the strongly connected sets \mathcal{V}_i by vertices j of the condensed graph and remove redundant edges between the new vertices.

Result: Condensed graph \vec{G}_C .

The algorithm uses the following fact. Whereas the reachability set $\mathcal{R}(i)$ defined in Section 2.2.2 includes all vertices that can be reached on a path starting in i , the reachability set $\mathcal{R}^T(i)$ of the reversed graph comprises all vertices from which a path leads to i . Since the property to be strongly connected holds for pairs of vertices that are connected by paths in both directions, such vertices belong to the intersection $\mathcal{R}(i) \cap \mathcal{R}^T(i)$ used in Step 2. This way for finding strongly connected vertices will be presented in matrix form in eqn. (5.44).

Connectedness of undirected graphs. The notion of strongly connected vertices introduced in Definition 2.1 for directed graphs has to be modified if it should be applied to undirected graphs, because the existence of a path $P(i - j)$ for the vertex pair (i, j) implies the existence of a path $P(j - i)$ for the pair (j, i) . In an undirected graph, a vertex pair is called *connected* if the graph contains a path between these vertices. The graph is said to be connected if every pair of vertices is connected. This notion of a connected graph replaces the notion of a strongly connected digraph. Otherwise the graph is called disconnected. One says that a directed graph is *weakly connected* if its embedded undirected graph is connected. Obviously, a directed graph may be weakly connected although it is not strongly connected like the graph shown in Fig. 2.9.

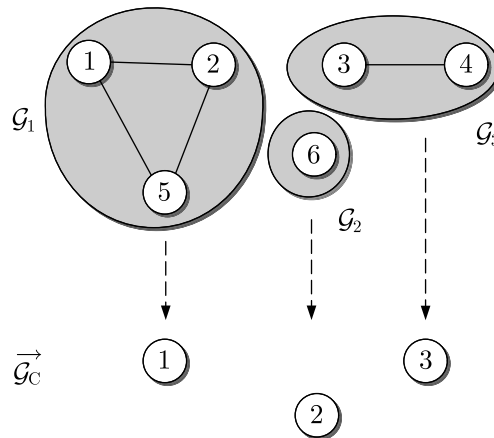


Fig. 2.10: Connected components and condensation of an undirected graph

If an undirected graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ is not connected, a partition of the set of vertices according to eqn. (2.19) can be introduced to get the connected components of \mathcal{G} :

$$\mathcal{G}_i = (\mathcal{V}_i, \mathcal{E}_i), \quad i = 1, 2, \dots, q. \quad (2.21)$$

In the condensation of the graph, every connected component is represented by a vertex and there are no edges at all among the vertices. That is, the connected components represent independent subgraphs as Fig. 2.10 shows.

For undirected graphs the existence of a spanning tree and the connectedness of the graph are equivalent:

$$\mathcal{G} \text{ is connected} \iff \mathcal{G} \text{ has a spanning tree.}$$

Exercise 2.1* Algorithm to check acycle graphs

Find an algorithm with which you can check whether a directed graph $\vec{\mathcal{G}}$ is acyclic. \square

2.3 Graph search

This section is devoted to *algorithmic graph theory*, which aims at elaborating efficient algorithms for checking graph properties. Graph search is an old subject that has already been dealt with by L. EULER in the 18th century and has been put forward mainly in the field of computer science in the last decades.

Since reachability turns out to be a fundamental property of graphs, the algorithms explained below have the task to find all vertices of an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ that are reachable from a fixed vertex $A \in \mathcal{V}$. The result is a *reachability tree* \mathcal{B} (cf. Section 2.2.2). \mathcal{B} should not only include some but all vertices that are reachable from the root vertex A and the algorithms that find such a tree for arbitrary graphs are called *complete*. These algorithms can be extended to get a path between a pair (A, B) of vertices whenever such a path exists, to find a shortest path or to decompose a graph into its connected components.

2.3.1 Trémaux algorithm

All search methods described below are commonly based on the imagination that an algorithm can move along the edges of the graph \mathcal{G} and mark all vertices that it meets. All marked vertices are inserted into the set \mathcal{M} and the edges along which they have been found into the set \mathcal{B} . After the algorithm has finished, \mathcal{M} includes all vertices that are reachable from A , and \mathcal{B} is a reachability tree. Algorithm 2.2, which has been proposed by C. P. TRÉMAUX¹, describes the common basis of the algorithms presented below.

Algorithm 2.2 Trémaux algorithm

Given: Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Root vertex A

Initialisation: Mark A .

Loop: **1.** Choose an edge $(S - X)$ with marked vertex S and unmarked vertex X .
 if No such edge exists. **then** Stop. **end**
 2. Mark X , insert the edge $(S - X)$ into the graph \mathcal{B} and continue with Step 1.
end

Result: All vertices that are reachable from A are marked and \mathcal{B} is a reachability tree.

The Trémaux algorithm starts with an empty graph \mathcal{B} and marks the root vertex A . The crucial point is Step 2, where for the current set \mathcal{M} of marked vertices an edge $(S - X)$ with $S \in \mathcal{M}$ and $X \notin \mathcal{M}$ should be found. This step has the typical description of a search problem, where the required result is specified but no direct way to find this result is given. Therefore, the algorithms described in this section are said to make a *graph search* and they distinguish with respect to the way in which they accomplish Step 2 of the Trémaux algorithm.

¹ CHARLES PIERRE TRÉMAUX (1859 – 1882), French engineer.

To get systematic ways for accomplishing Step 2, the algorithms below choose a marked vertex $S \in \mathcal{M}$ as the *starting point* of the next search step. S can be interpreted as the current location of the algorithms in the graph \mathcal{G} from which they should find an edge $(S - X)$ towards an unmarked vertex X . To make sure that eventually all edges are checked, the algorithms classify the marked vertices as active or passive². A newly marked vertex is put into the list \mathcal{A} to label it as *active* as long as not all its out-going edges have been checked. The fact that \mathcal{A} is empty indicates the end of the algorithms.

In summary, the graph-search algorithms have to maintain the following data:

- S – Starting point of the next search step
- \mathcal{M} – Set of marked vertices, which eventually includes all reachable vertices
- \mathcal{B} – Subgraph of \mathcal{G} , which eventually is a reachability tree
- \mathcal{A} – List of active vertices; if \mathcal{A} is empty, the algorithm ends.

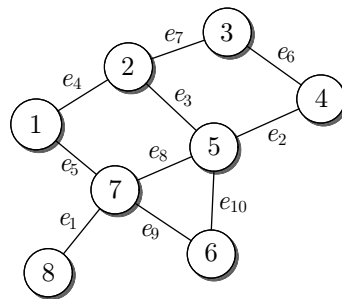


Fig. 2.11: Undirected graph

Notation. To apply graph-search algorithms, the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ has to be stored in a reasonable manner in the computer memory. It can be written as the set of vertex pairs that are connected by an edge, for example

$$\mathcal{G} = ((7 - 8), (4 - 5), (2 - 5), (1 - 2), (1 - 7), (3 - 4), (2 - 3), (5 - 7), (6 - 7), (5 - 6))$$

for the graph in Fig. 2.11³. With an edge $(i - j)$, also the edge $(j - i)$ exists. Alternatively, the *adjacency list* records for each vertex the set of adjacent vertices, e. g.

$$\mathcal{G} = \begin{array}{l} 1 : 2, 7 \\ 2 : 1, 3, 5 \\ 3 : 2, 4 \\ \vdots \\ 8 : 7 \end{array} . \quad (2.22)$$

² In literature, alternative notions are “open” or “unexplored” for active vertices and “closed” or “explored” for passive vertices.

³ The algorithms distinguish between ordered sets like $\mathcal{M} = \{1, 2, 3\}$ and lists like $\mathcal{B} = ((1 - 2), (2 - 3))$. Lists are ordered sets. The symbol \emptyset is also used for empty lists.

Although the order of the vertices and edges is not important for the result whether a vertex is reachable, the reachability tree obtained by an algorithm depends upon this order. The algorithms usually go through the sets or lists from top to bottom and find unmarked vertices in accordance with the edge order in the computer memory.

The assignments in the flowcharts below use the following notation:

- $\mathcal{M} \leftarrow X$ Insert the vertex X into the set \mathcal{M} .
- $\mathcal{B} \leftarrow (S - X)$ Insert the edge $(S - X)$ into the graph \mathcal{B} .
- $S \leftarrow \mathcal{A}$ Take the first element of the list \mathcal{A} as the starting point S of the next search step.
- $S \leftarrow A$ Assign the value A to the variable S .

The last two assignments differ with respect to the fact that A is a scalar, whereas \mathcal{A} is a set.

2.3.2 Depth-first search

For depth-first search, Step 2 of the Trémaux algorithm is carried out as follows:

Depth-first search: For a given vertex S , find the first edge $(S - X)$ with an unmarked vertex X . If such an edge exists, mark X ($\mathcal{M} \leftarrow X$), insert X into the list \mathcal{A} in the first position ($\mathcal{A} \leftarrow X$), insert the edge $(S - X)$ into \mathcal{B} ($\mathcal{B} \leftarrow (S - X)$) and use X as the starting point for the next search step ($S \leftarrow X$). If no such an edge exists, delete the first element of \mathcal{A} and continue the search from the new first element of \mathcal{A} ($S \leftarrow \mathcal{A}$).

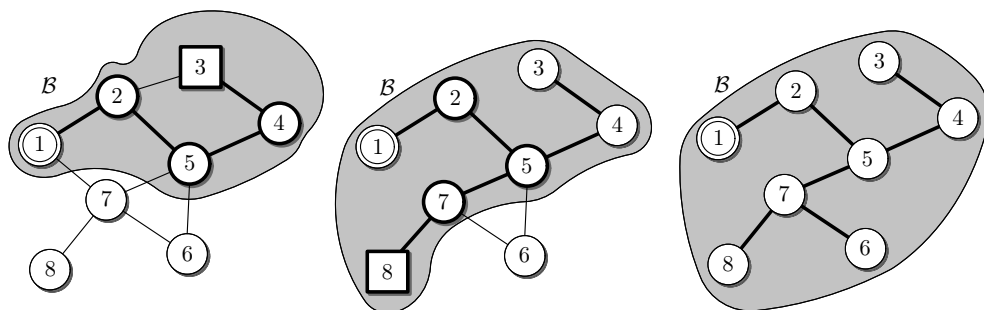


Fig. 2.12: Intermediate results (left and middle) and result (right) of a depth-first search

Obviously, the list \mathcal{A} is organised as a stack (last-in first-out (LIFO) memory). The algorithm takes the deepest vertex as starting point S for the next search step, where the depth $d = |P(A - X)|$ of X is the length of the path $P(A - X)$ in the tree \mathcal{B} . This property is illustrated in Fig. 2.12, which depicts the results of a depth-first search through the graph of Fig. 2.11 with the root vertex $A = 1$. The square vertex indicates the current vertex S , the highlighted

edges the tree \mathcal{B} and all highlighted vertices the list \mathcal{A} . After four steps, \mathcal{B} consists of four edges and from the current vertex $S = 3$ no unmarked vertex can be found (left part of Fig. 2.12). Therefore, the algorithm tracks back the path that it came until it reaches the vertex 5 to find 7 as the next unmarked vertex. This process is called a *backtracking*. It leaves the vertices 3 and 4 as passive vertices, which are no longer highlighted in the middle of the figure. A complete reachability tree is obtained after the algorithm has continued until the list \mathcal{A} of active vertices is empty (Fig. 2.12 (right)).

Algorithm 2.3 *Depth-first search of a reachability tree*

Given: Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Root vertex A

Initialisation: Mark A , insert A into the list \mathcal{A} of active vertices and use A as starting point S of the next search step.

Loop: while $\mathcal{A} \neq \emptyset$

1. **Search step:** Find an edge $(S - X)$ with unmarked vertex X .

2. **Search control:**

if Such an edge exists.

then Mark X , insert X into \mathcal{A} and $(S - X)$ into \mathcal{B} .
Start the next search step from $S \leftarrow X$.

else Delete the first element of \mathcal{A} .

if $\mathcal{A} = \emptyset$

then Output \mathcal{B} ; stop.

else Use the first element of \mathcal{A} as the new starting point S of the search step.

end

end

end

Result: \mathcal{M} includes all reachable vertices and \mathcal{B} is a reachability tree.

The depth-first search is represented as a flowchart in Fig. 2.13 and as Algorithm 2.3. The main parts of the algorithm concern of a search step and the search control. The search step gets the marked vertex S and scans the list of edges from top to bottom to find an edge $(S - X)$ with unmarked end vertex X . The search control should ensure that all necessary search steps are carried out to find all reachable vertices without repeating earlier steps. The characteristic property of depth-first search is the fact that any newly marked vertex is at once used as the starting point S of the next search step (“Step forward”). If the search step is not successful, the algorithm backtracks its way until it eventually reaches the root vertex A . Correspondingly, the step to move the starting point S to any newly found vertex X is tentative and provision is made to return later to an earlier point of search. To ensure that all possibilities have been explored, the search-control strategy uses the list \mathcal{A} of active vertices and quits the search only when this list is empty. The next two algorithms described below distinguish from this depth-first search only with respect to their control strategy.

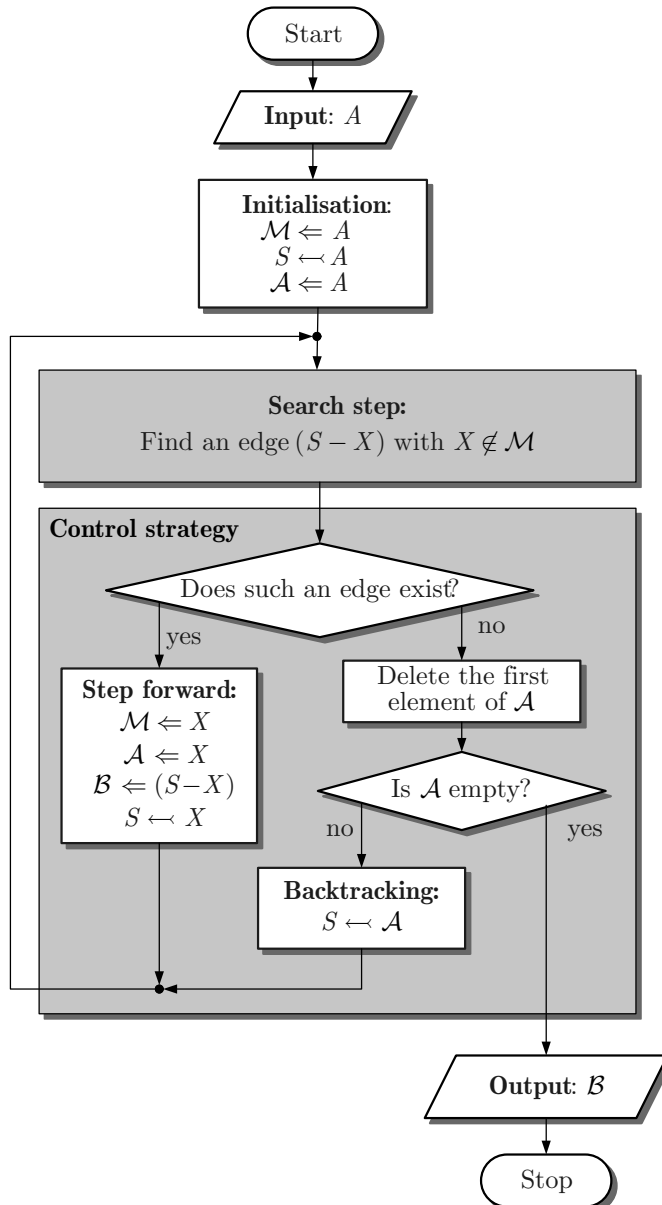


Fig. 2.13: Depth-first search algorithm

|| Depth-first search is complete.

Correspondingly, the depth-first search algorithm finds all vertices that are reachable from the given root vertex A and outputs a reachability tree \mathcal{B} . If the graph is connected, \mathcal{B} is a spanning tree with N vertices and $N - 1$ edges.

2.3.3 Breadth-first search

Breadth-first search uses Step 2 of the Trémaux algorithm in the following way:

|| **Breadth-first search:** For a given vertex S , check all outgoing edges to find unmarked vertices. If an edge to an unmarked vertex X is found, mark X ($\mathcal{M} \leftarrow X$), insert X into the back of the list \mathcal{A} ($\mathcal{A} \leftarrow X$), insert the edge ($S - X$) into \mathcal{B} ($\mathcal{B} \leftarrow (S - X)$) and continue searching. Otherwise delete the first element of \mathcal{A} and continue the search from the new first element of \mathcal{A} ($S \leftarrow \mathcal{A}$).

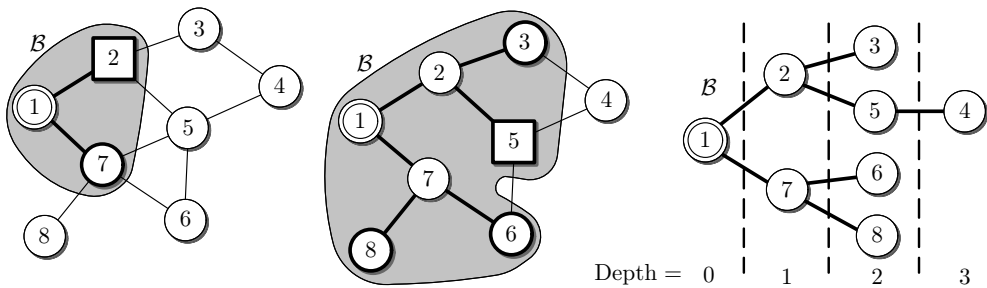


Fig. 2.14: Intermediate results (left and middle) and result (right) of a breadth-first search

The list \mathcal{A} of active vertices is now a queue (first-in first-out (FIFO) memory), which is the only difference of this algorithm with respect to the depth-first search algorithm as Figs. 2.13 and 2.15 reveal.

Figure 2.14 shows that the new control strategy has the effect that from any single vertex S the algorithm looks into all directions before it goes to the next vertex (it searches “breadth first”). The intermediate result in the middle of the figure depicts the situation that the algorithm has investigated all edges with the end vertices 1, 2 and 7 and that these vertices are no longer active. It becomes obvious that all vertices of a certain depth d are investigated before a vertex of depth $d + 1$ is explored. Therefore, the resulting reachability tree includes shortest paths $P(A - X)$ to any reachable vertex X (cf. Fig. 2.14 (right)).

|| Breadth-first search is complete. It marks all vertices in ascending distance from the root vertex A .

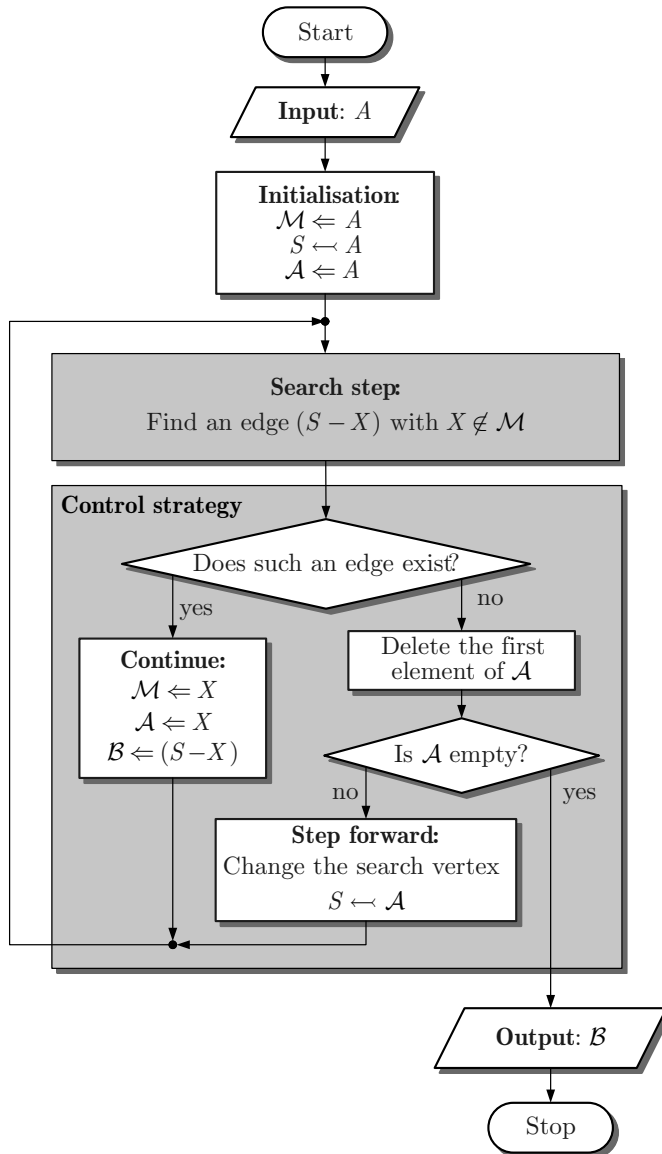


Fig. 2.15: Breadth-first search algorithm

Like depth-first search, this method finds all vertices that are reachable from the specified vertex A . If the graph is connected, the tree \mathcal{B} is a spanning tree with N vertices and $N - 1$ edges.

Complexity. As any edge is investigated only twice when the algorithm is in one of its two end vertices, the time complexity $O(|\mathcal{E}|)$ of breadth-first search is linear with respect to the number of edges. If the distance between any pair of vertices should be found, the algorithm has to be used sequentially with all vertices as root, which gives the overall time complexity $O(|\mathcal{E}| \cdot |\mathcal{V}|)$.

Extensions. Both depth-first search and breadth-first search can be extended to find paths between two given vertices A and B if any newly marked vertex is checked to be the end vertex B . If the answer is in the affirmative, the tree \mathcal{B} includes a path $P(A - B)$. For depth-first search, this path is also represented by the current list \mathcal{A} of active vertices. Breadth-first search gives a shortest path.

2.3.4 Dijkstra algorithm

This section extends the breadth-first search algorithm to find a shortest path (or geodesic path) between two vertices A and B if all edges $(i - j) \in \mathcal{E}$ have a nonnegative length $k(i - j)$, which is indicated as the edge labels in Fig. 2.16. The path length $|P(A - B)|$ is the sum of the edge lengths:

$$|P(A - B)| = \sum_{(i-j) \in P(A-B)} k(i-j). \quad (2.23)$$

The notation $(i - j) \in P(A - B)$ means that the edge $(i - j)$ belongs to the path $P(A - B)$. The algorithm to be presented should find a shortest path

$$P^*(A - B) = \arg \min_{P(A-B) \in \mathcal{P}(A-B)} |P(A - B)|. \quad (2.24)$$

The argument “arg” of the optimisation problem is a path with the minimum length $k_p^*(A - B)$ within the set $\mathcal{P}(A - B)$ of all paths between A and B . Generally, the solution of this problem is not unique, because there may exist several paths of the same shortest length. The reader is advised to determine now a shortest path $P^*(1 - 8)$ in the graph of Fig. 2.16 to understand the search problem (2.24).

It is, of course, possible to solve this problem by determining the set $\mathcal{P}(A - B)$ of all paths between A and B and to find a shortest path by comparing all path lengths. However, this section presents the algorithm proposed by E. W. DIJKSTRA in 1959, which generates only a few paths $P(A - B)$ before it finds a solution $P^*(A - B)$ to the problem (2.24).

The preceding sections have shown that a depth-first search distinguishes from a breadth-first search only in the organisation of the list \mathcal{A} of active vertices, which was a stack or a queue, respectively. The Dijkstra algorithm uses another organisation of \mathcal{A} and sorts the active vertices X according to the length

$$g(X) = |P(A - X)| \quad (2.25)$$

of the path $P(A - X)$ that it has found so far. \mathcal{A} stores the pairs $(X, g(X))$ in ascending order of $g(X)$. As before, the first element of \mathcal{A} is used as the starting point S of the next search step. Hence, the tree \mathcal{B} is extended at the vertex with the shortest path:

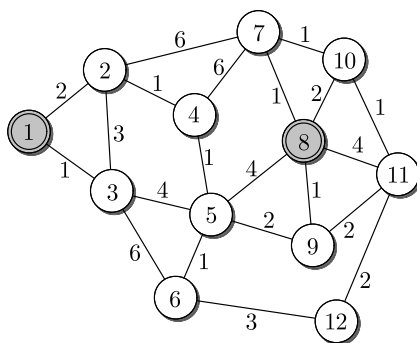


Fig. 2.16: Weighted graph

$$S = \arg \min_{X \in \mathcal{A}} g(X). \quad (2.26)$$

Whereas in the two algorithms presented above it did not matter which reachability tree \mathcal{B} the algorithm generates, the tree \mathcal{B} should now include only those edges that belong to shortest paths between A and the vertices of \mathcal{B} . Therefore, \mathcal{B} is composed no longer of all marked vertices, but only of all passive vertices together with edges among passive vertices, whereas the active vertices form a vertex set outside of \mathcal{B} . \mathcal{K} is a set of edges between \mathcal{B} and \mathcal{A} (dashed edges in Fig. 2.17). In summary, the classification of the vertices has now the following meaning:

- **Unmarked vertices:** Vertices X that have not been investigated by the algorithm so far (the set of unexplored vertices in Fig. 2.17). No path $P(A - X)$ has been found yet.
- **Active vertices:** Vertices X , for which a path $P(A - X)$ is known, but it is not yet clear whether this is a shortest path $P^*(A - X)$ (the frontier of search in the figure).
- **Passive vertices:** Vertices X , for which a shortest path $P^*(A - X)$ is known. This path belongs to \mathcal{B} , which includes all explored vertices.

Accordingly, the Dijkstra algorithm stores the intermediate results in three lists:

- \mathcal{B} – Tree with root A , in which every vertex X is connected with A by a shortest path $P^*(A - X)$ (thick lines in Fig. 2.17). All vertices of \mathcal{B} are passive.
- \mathcal{A} – List of active vertices, which is sorted according to $g(X)$.
- \mathcal{K} – Set of edges between passive and active vertices (dashed lines in the figure).

As in the flowchart in Fig. 2.18, the search step of the breadth-first algorithm is extended in the sense that for a given starting point S not only edges $(S - X)$ towards unmarked vertices $X \notin \mathcal{M}$ are to be found, but also edges towards active vertices $X \in \mathcal{A}$. For all such vertices the distance

$$g(X) = g(S) + k(S - X)$$

is determined to make the following decisions:

- If X was unmarked until now, X is marked, the pair $(X, g(X))$ is inserted into \mathcal{A} and the edge $(S - X)$ is inserted into \mathcal{K} .

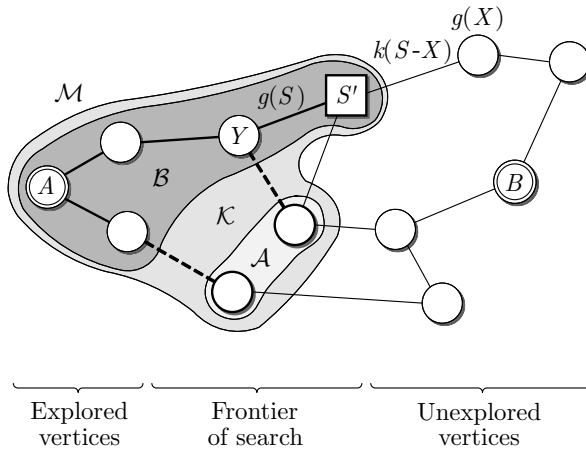


Fig. 2.17: Principle of the Dijkstra algorithm

- (b) If X was already marked, it depends upon the length $g(X)$ of the new path $P(A-X)$ what happens. If the new path $P(A-X)$ is shorter than the old path ($g(X) < g'(X)$), the new edge $(S-X)$ replaces the old edge towards X in \mathcal{K} and X is sorted in \mathcal{A} accordingly. Otherwise the edge $(S-X)$ is discarded. This situation is demonstrated in the steps 3 and 5 of Example 2.3 and considered as the modification of \mathcal{A} and \mathcal{K} in Algorithm 2.4.

Both situations are dealt with by the “continue”-step in Fig. 2.18. To get a concise flowchart they are commonly represented as $\mathcal{M} \leftarrow X$ for marking the new vertex, if necessary, and $\mathcal{A} \leftarrow (X, g(X))$ for inserting or moving the vertex X in the list \mathcal{A} . The new edge $(S-X)$ is included into the set \mathcal{K} .

Like in a breadth-first search, all outgoing edges of the vertex S are checked. Afterwards a new starting point S is taken from \mathcal{A} and the edge $(S-Y)$ removed from \mathcal{K} and put into \mathcal{B} (“Change the search vertex”).

It is important to see that the algorithm does not stop after it has found the destination B , but after this vertex has been chosen as the starting point S of the next search step. The optimum path $P^*(A-B)$ is included in \mathcal{B} . If \mathcal{A} is empty before the algorithm reaches this state, no path between A and B exists.

Theorem 2.1 (Properties of the Dijkstra algorithm)

The Dijkstra algorithm finds a shortest path $P^(A-B)$ in any weighted graph for any pair (A, B) if a path exists.*

The complexity of the algorithm is $O(|\mathcal{V}|^2)$.

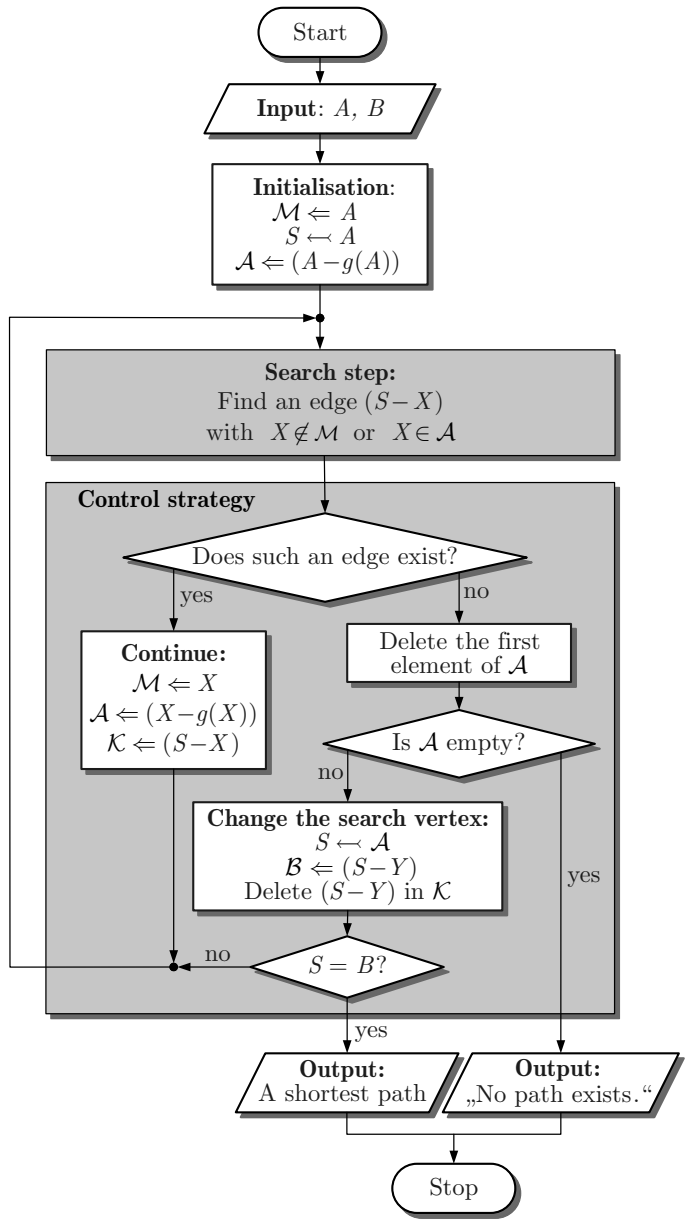


Fig. 2.18: Dijkstra's algorithm

Proof. To show that at the end of Dijkstra's algorithm the graph \mathcal{B} includes a shortest path $P^*(A-B)$, it is proved that at any time the current vertex S satisfies the following relation

$$g(S) = \min_{P(A-S) \in \mathcal{P}(A-S)} |P(A-S)| = k_P^*(A-S) \quad (2.27)$$

with $k_P^*(A-S)$ denoting the length of a shortest path $P^*(A-S)$. That is, the graph \mathcal{B} is extended at any time at a vertex S for which \mathcal{B} includes a shortest path $P^*(A-S)$ and a solution is found if the vertex B is the first element of \mathcal{A} .

The proof is accomplished by induction. In the first search step, all edges starting in A are investigated and according to eqn. (2.26) the vertex with the shortest distance $g(X) = k(A-X)$ is used as the next vertex S :

$$S = \arg \min_{X \in \mathcal{M}} k(A-X).$$

Hence, $g(S) = k_P^*(A-S)$ holds and proves eqn. (2.27) for the first search steps.

The induction hypothesis says that the relation

$$g(Y) = k_P^*(A-Y) \quad \text{for all } Y \in \mathcal{B} \quad (2.28)$$

holds (Fig. 2.19 (left)). It has to be shown that this relation also holds after the next iteration step, which generates the new data S' and \mathcal{B}' (Fig. 2.19 (right)).

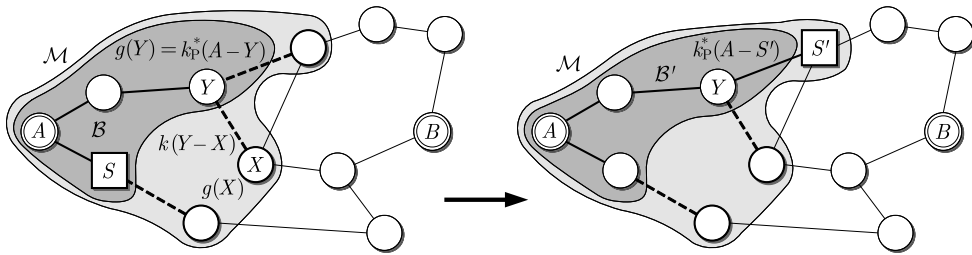


Fig. 2.19: Proof of the correctness of the Dijkstra algorithm

For all vertices X in \mathcal{A} , which lie in the set \mathcal{M} , but not in the tree \mathcal{B} , there is an edge $(Y-X) \in \mathcal{K}$ with $Y \in \mathcal{B}$ for which the relation

$$g(X) = g(Y) + k(Y-X) = k_P^*(A-Y) + k(Y-X)$$

holds (Fig. 2.19 (left)). The next starting point S' of the search step is chosen according to eqn. (2.26):

$$S' = \arg \min_{X \in \mathcal{A}} g(X).$$

The edge $(Y-S')$ from \mathcal{B} towards S' is inserted into \mathcal{B} to get the new tree \mathcal{B}' . For S' one gets

$$\begin{aligned} g(S') &= \min_{X \in \mathcal{A}} g(X) \\ &= \min_{X \in \mathcal{A}} (k_P^*(A-Y) + k(Y-X)) \\ &= k_P^*(A-S'). \end{aligned}$$

The equality sign holds because for all vertices $X \neq S'$ the inequality $g(X) \geq g(S')$ is valid. Hence, when shifting the starting point S of the search towards S' a new shortest path $P^*(A-S')$ is included into \mathcal{B} , for which eqn. (2.27) holds. \square

Algorithm 2.4 *Dijkstra algorithm*

Given: Weighted graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, k)$
End vertices A, B

Initialisation: Mark A , insert $(A, g(A) = 0)$ into the list \mathcal{A} and use $S = A$ as starting point of the search.

Loop: **while** $S \neq B$

1. **Search step:** Find an edge $(S - X)$ with $X \in \mathcal{A}$ or $X \notin \mathcal{M}$.

2. **Search control:**

if Such an edge exists.

then if $X \notin \mathcal{M}$

then Mark X , insert $(X, g(X))$ into \mathcal{A} and $(S - X)$ into \mathcal{K} .

else Modify \mathcal{A} and \mathcal{K} if necessary.

end

else Delete the first element of \mathcal{A} .

if $\mathcal{A} = \emptyset$

then Output “No path exists.”; stop.

else Use the first element of \mathcal{A} as the new starting point S of the search step.

Delete $(S - Y)$ from \mathcal{K} and insert it into \mathcal{B} .

end

end

end

Result: The unique path $P(A - B)$ in \mathcal{B} is a shortest path $P^*(A - B)$.

Example 2.3 *Determination of a shortest path $P^*(A - B)$ by means of Dijkstra’s algorithm*

When searching for a shortest path $P^*(1-8)$ in the graph of Fig. 2.16, Algorithm 2.4 gives the following results.

1. Initialisation:

$$\mathcal{M} = \{1\}$$

$$S = 1$$

$$\mathcal{A} = (1, g(1) = 0)$$

$$\mathcal{K} = \{\}$$

$$\mathcal{B} = ()$$

2. Search steps with $S = 1$ yield the unmarked vertices 2 and 3 and the intermediate results

$$\mathcal{M} = \{1, 2, 3\}$$

$$\mathcal{A} = (1, g(1) = 0; 3, g(3) = 1; 2, g(2) = 2)$$

$$\mathcal{K} = \{(1-2), (1-3)\}.$$

After deleting the first element of \mathcal{A} , $S \leftarrow 3$ is the next search vertex and the corresponding edge $(1-3)$ is deleted from \mathcal{K} and inserted into \mathcal{B} :

$$S = 3$$

$$\mathcal{A} = (3, g(3) = 1; 2, g(2) = 2)$$

$$\mathcal{K} = \{(1-2)\}$$

$$\mathcal{B} = ((1-3)).$$

The edge $(1-3)$ is a shortest path $P^*(1-3)$ between the vertices 1 and 3. Figure 2.20 (left) illustrates the result with the vertices 1, 2 and 3 in the set \mathcal{M} , the edge $(1-3)$ in the tree \mathcal{B} and the vertex 3 as the starting point for the next search step. The vertices 2 and 3 have the labels $g(2) = 2$ and $g(3) = 1$ written next to the vertices in the figure.

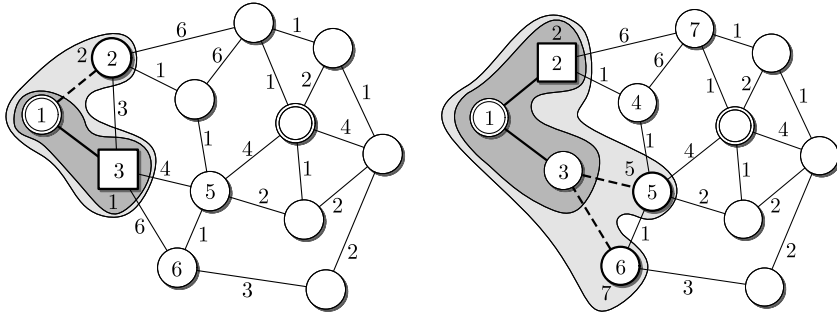


Fig. 2.20: Intermediate results of the Dijkstra algorithm after the first and the second step

3. The next search steps find the unmarked vertices 5 and 6:

$$\mathcal{M} = \{1, 2, 3, 5, 6\}.$$

The edge $(3-2)$ gives the new value $g(2) = 4$ for the vertex 2, which is larger than the old value $g'(2) = 2$ and implies that the edge $(3-2)$ does not lead to any modification:

$$\mathcal{A} = (3, g(3) = 1; 2, g(2) = 2; 5, g(5) = 5; 6, g(6) = 7)$$

$$\mathcal{K} = \{(1-2), (3-5), (3-6)\}.$$

After deleting the first element of \mathcal{A} , the intermediate result is as follows (Fig. 2.20 (right)):

$$\mathcal{M} = \{1, 2, 3, 5, 6\}$$

$$S = 2$$

$$\mathcal{A} = (2, g(2) = 2; 5, g(5) = 5; 6, g(6) = 7)$$

$$\mathcal{K} = \{(3-5), (3-6)\}$$

$$\mathcal{B} = ((1-3), (1-2)).$$

The edge $(2-3)$ has been deleted in the figure, because it does neither belong to \mathcal{B} nor to \mathcal{K} .

4. Searching from $S = 2$ leads to the next results (Fig. 2.21 (left)):

$$\mathcal{M} = \{1, 2, 3, 5, 6, 4, 7\}$$

$$S = 4$$

$$\mathcal{A} = (4, g(4) = 3; 5, g(5) = 5; 6, g(6) = 7; 7, g(7) = 8)$$

$$\mathcal{K} = \{(3-5), (3-6), (2-7)\}$$

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

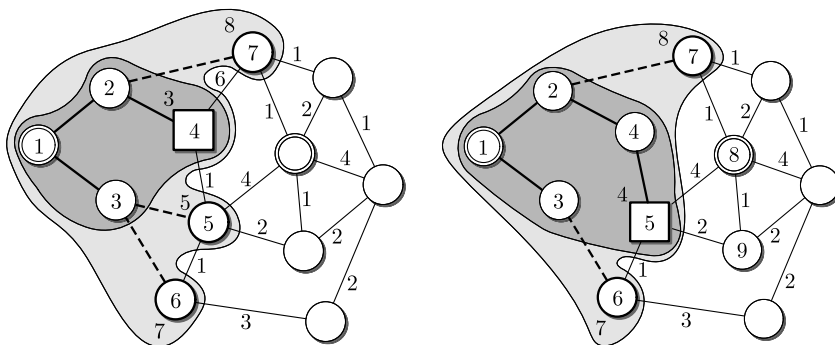


Fig. 2.21: Intermediate results of the Dijkstra algorithm after three and four steps

5. The edge $(4 - 5)$ leads to the path $P(1 - 5) = ((1 - 2), (2 - 4), (4 - 5))$ which is shorter than the known path $P(1 - 5) = ((1 - 3), (3 - 5))$. Hence, the edge $(3 - 5)$ is replaced in \mathcal{K} by the edge $(4 - 5)$ and the vertex 5 appears in the list \mathcal{A} with the new value $g(5) = 4$ (Fig. 2.21 (right)):

$$\begin{aligned} \mathcal{M} &= \{1, 2, 3, 5, 6, 4, 7\} \\ S &= 5 \\ \mathcal{A} &= (5, g(5) = 4; 6, g(6) = 7; 7, g(7) = 8) \\ \mathcal{K} &= \{(3 - 6), (2 - 7)\} \\ \mathcal{B} &= ((1 - 3), (1 - 2), (2 - 4), (4 - 5)). \end{aligned}$$

6. After the next search steps, the destination vertex 8 has been found, but this vertex does not yet appear as the first element of \mathcal{A} . Therefore, the algorithm continues with $S = 6$ (Fig. 2.22 (left)):

$$\begin{aligned} \mathcal{M} &= \{1, 2, 3, 5, 6, 4, 7, 8, 9\} \\ S &= 6 \\ \mathcal{A} &= (6, g(6) = 5; 9, g(9) = 6; 7, g(7) = 8; 8, g(8) = 8) \\ \mathcal{K} &= \{(2 - 7), (5 - 8), (5 - 9)\} \\ \mathcal{B} &= ((1 - 3), (1 - 2), (2 - 4), (4 - 5), (5 - 6)). \end{aligned}$$

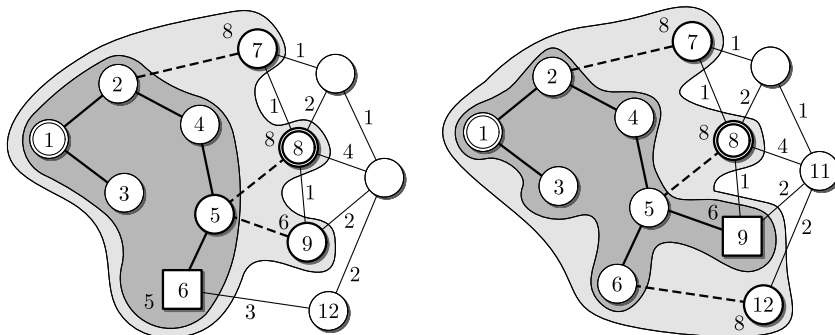


Fig. 2.22: Intermediate results of the Dijkstra algorithm after five and six steps

7. Next, the vertex 12 is found and the starting point moved to $S = 9$ (Fig. 2.22 (right)):

$$\mathcal{M} = \{1, 2, 3, 5, 6, 4, 7, 8, 9, 12\}$$

$$S = 9$$

$$\mathcal{A} = (9, g(9) = 6; 7, g(7) = 8; 8, g(8) = 8; 12, g(12) = 8)$$

$$\mathcal{K} = \{(2-7), (5-8), (6-12)\}$$

$$\mathcal{B} = ((1-3), (1-2), (2-4), (4-5), (5-6), (5-9)).$$

8. Searching from $S = 9$ leads to the newly marked vertex 11 and the new edge $(9-8)$, which modifies the path $P(1-8)$. The new value $g(8) = 7$ moves the destination vertex 8 to the front of \mathcal{A} . Consequently, the Dijkstra algorithm finishes with the following data (Fig. 2.23 (left)):

$$\mathcal{M} = \{1, 2, 3, 5, 6, 4, 7, 8, 9, 12, 11\}$$

$$S = 8$$

$$\mathcal{A} = (8, g(8) = 7; 7, g(7) = 8; 12, g(12) = 8; 11, g(11) = 8)$$

$$\mathcal{K} = \{(2-7), (6-12), (9-11)\}$$

$$\mathcal{B} = ((1-3), (1-2), (2-4), (4-5), (5-6), (5-9), (9-8)).$$

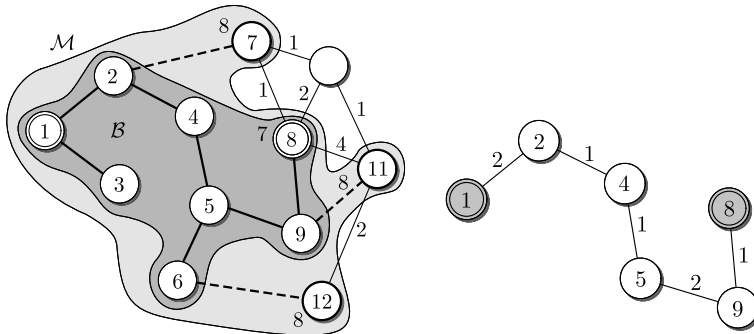


Fig. 2.23: Result of the Dijkstra algorithm after seven steps (left) and shortest path (right)

Result: A shortest path $P^*(1-8)$ is the only path $P(1-8)$ within \mathcal{B} and depicted in Fig. 2.23 (right). Its length is $g(8) = 7$:

$$P^*(1-8) = ((1-2), (2-4), (4-5), (5-9), (9-8))$$

$$k_{\mathcal{P}}^*(1-8) = 7. \quad \square$$

2.3.5 Properties of graph-search algorithms

This section summarises important properties of the search algorithms presented above and points to heuristic extensions, which will be introduced in Chapter 3.

The algorithms have been presented with the two main components

- **Search step:** For a given starting point, find an edge towards an unmarked vertex.

- **Graph-search control:** For a given search result, decide what to do next.

The algorithms accomplish a search for “new” vertices by starting in a given vertex A and moving forward through the graph. From a global point of view, the given graph represents the *search space*, which is finite due to the assumption to consider finite graphs. Search control has the task to keep track of the results and to ensure that an *exhaustive search* is carried out. If a path between any two vertices A and B exists, the algorithms will find it. To this end, search control distinguishes between vertices that have been explored completely and other vertices that may be incident to unmarked vertices. The active vertices mark the frontier of the search with all passive (explored) vertices behind it and all unmarked vertices in front of it (Fig. 2.17). The tree \mathcal{B} represents the intermediate result, which will be called the *search graph* or search tree later on.

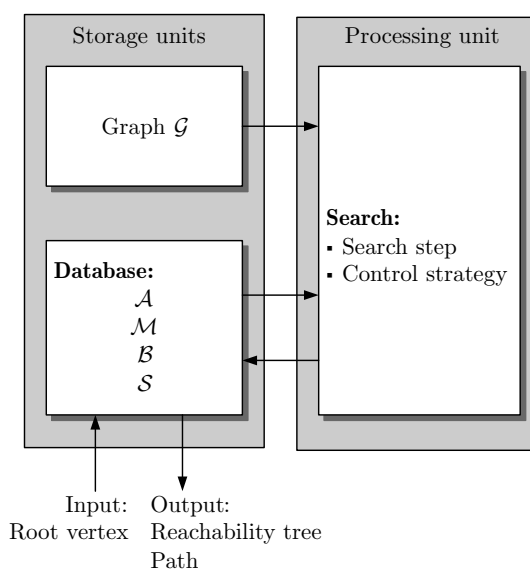


Fig. 2.24: Architecture of graph-search systems

An important characteristic of the Dijkstra algorithm is the fact that the algorithm generally finds a shortest path $P^*(A - B)$ after having generated a search graph, which is small in comparison to the given graph representing the search space. However, all search algorithms presented above, including the Dijkstra algorithm, are *uninformed* (or blind) in the sense that they do not know anything about the structure of the graph and, thus, cannot use specific circumstances to find the required result faster. Further heuristic elements should enable the algorithm to end with a smaller search graph. With this goal, the Algorithm A^* will be introduced in Section 3.2.

Architecture of search systems. The structure of the graph-search algorithms presented above provides the paradigm for the architecture of many search systems like those used in logic-based systems (Chapter 3) or Bayesian networks (Chapter 4). As depicted in Fig. 2.24, such

systems consist of two storage and one processing component. The graph, which defines the search space, is stored in a read-only memory (ROM) that feeds the information to the processing unit. The database includes the elements \mathcal{A} , \mathcal{M} , \mathcal{B} and S that define the current state of the search algorithm. The processing unit accomplishes the steps of searching for the next vertex and of controlling the search process. This architecture will be extended for other search algorithms in the next chapters.

An important point was to see that the search control distinguishes only with respect to the organisation of the list \mathcal{A} of active vertices. This fact will be extended for heuristic search algorithms.

The algorithms have been presented for undirected graphs, but they can be extended easily for directed graphs.

Quality of search methods. Search methods and, in particular, search-control strategies are generally evaluated with respect to two aspects:

- **Completeness:** The search method should find a solution whenever a solution exists. To ensure the completeness of the algorithm, the search-control strategy exploits the list \mathcal{A} of active vertices, which shows unexplored directions of search.
- **Complexity:** Search algorithms differ with respect to their time or space complexity. The more information is used to implement the control strategy, the more computational steps are necessary for it, but if this information is utilised in a clever way, less search steps are necessary until a solution is found. The Dijkstra algorithm has shown how the information of the search graph can be utilised to find a shortest path before the whole graph has been explored. Chapter 3 will demonstrate how search algorithms can be extended by heuristic means to further reduce the search complexity.

The algorithms described above have been considered in their simplest form to explain the main idea of graph search. In an implementation, several measures can be taken to reduce their complexity. For example, it is reasonable to store the number of an edge if the edge leads the algorithm to an unmarked vertex in order to ensure that the next search step continues from this edge rather than from the top of the list.

Exercise 2.2 *Depth-first search in an undirected graph*

Apply Algorithm 2.3 to the graph of Fig. 2.11 with the following modifications:

1. Delete the edge e_3 .
2. Delete the edges e_8 and e_9 .
3. Change the enumeration of the edges and repeat the search.

Track all steps of the algorithm and compare the results with Fig. 2.12. □

Exercise 2.3* *Application of Dijkstra's algorithm*

By using the Dijkstra algorithm, determine a shortest path $P^*(A - B)$ in the graph of Fig. 2.25. Keep track of the assignments of the variables S , \mathcal{M} , \mathcal{A} , \mathcal{K} and \mathcal{B} . The edges are sorted according to the vertices in the set $\mathcal{V} = \{A, B, 1, 2, 3\}$. How does you evaluate the search process? □

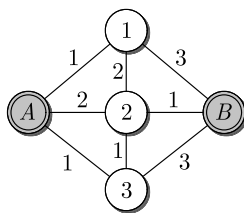


Fig. 2.25: Weighted graph

2.4 MATLAB functions for graph objects

2.4.1 Create graph objects

The MATLAB kernel includes several graph and network algorithms, which work on graph objects. They can be used to apply the methods introduced in this textbook. The vertices are uniformly called `nodes`, directed graphs as `digraph` and undirected graphs as `graph`.

One has to distinguish between `graph` and `digraph` objects, where the first kind is used for undirected graphs. The following summary of the most important functions deals with both kinds of graphs simultaneously where this is possible.

For two M -vectors s and t with the end points of the edges the function calls

```
» G=graph(s,t);
» G=digraph(s,t);
```

create an undirected and a directed graph object. For the directed graph, the vector s includes the initial vertices and t the final vertices of the edges, for example

```
» s=[7 4 2 1 1 3 2 5 6 5];
» t=[8 5 5 2 7 4 3 7 7 6];
» G=graph(s,t);
```

for the graph of Fig. 2.11. For weighted edges, the M -vector w specifies the weights of the edges defined by s and t :

```
» G=graph(s,t,w);
» G=digraph(s,t,w);
```

MATLAB functions for weighted graphs will be introduced in Section 5.4 by using the adjacency matrix for its generation.

For a graph object G , one can use the dot notation to get a table of the edges

```
» Edgelist=G.Edges;
```

or the M -vector of the edge weights:

```
» Edgelist=G.Edges.Weight;
```

The table of end nodes received shows in the `EndNodes` column first the initial vertices and second the final vertices and, if the graph is weighted, in the second column the edge weight.

Several functions are defined to add or to remove edges or nodes to or from a given graph G :

```
% add edges between vertices specified in s and t
» G=addege(G,s,t);
% add the vertex i
» G=addnode(G,i);
% remove the edges with vertices specified in s and t
» G=rmedge(G,s,t);
% remove the vertex i
» G=rmnode(G,i);
```

The following examples illustrate how to use these functions:

```
» G=addege(G,[1 3],[2 5]); % add edges (1-2) and (3-5)
» G=addnode(G,6);
» G=rmedge(G,[1 3],[2 5]); % remove the edges (1-2) and (3-5)
» G=rmnode(G,6);
```

The function `degree` determines a vector d with the degree of each vertex in an undirected graph G

```
» d=degree(G); % undirected graph
```

For directed graphs the functions `indegree` and `outdegree` return the in-degree (2.6) and the out-degree (2.7) of all vertices as column vectors:

```
» d=indegree(G); % directed graph
» dout=outgree(G);
```

2.4.2 Graph properties

For undirected graphs G , the function `neighbors` selects the vertices that are connected by an edge to the vertex i specified in the function call

```
» Ni=neighbors(G,i); % undirected graph
```

N_i is a column vector. For directed graphs G , the functions `predecessors` and `successors` determine, for the vertex i , the sets (2.1) or (2.3) of predecessors or successors, respectively:

```
» Ni=predecessors(G,i); % directed graph
» Si=successors(G,i);
```

The function `nearest` finds all vertices in a graph G , which lie within the distance d from the vertex i (cf. eqn. (2.13):

```
» Nd=nearest(G, i, d);
```

`Nd` is a column vector. For weighted graphs, the distance is determined in accordance with the weights of the corresponding edges.

For connected undirected graphs, a spanning tree can be obtained by the following function. The result obtained with the function call

```
» T=minsantree(G); % undirected graph
```

is a minimum spanning tree in the sense that the sum of the edge weights is minimum. For graphs with equal edge weights, each spanning tree is minimum with $N - 1$ edges for N vertices.

For directed or undirected graphs, the function `conncomp` partitions the set of vertices into the equivalence classes of strongly connected vertices or connected vertices, respectively (cf. Section 2.2.4).

```
» CompNumber=conncomp(G);
```

The row vector `CompNumber` associates with all vertices the number of the strong component that a vertex belongs to. If `CompNumber(i)=j` holds, the vertex i belongs to the component j . If all elements of `CompNumber` are equal to 1, the graph is (strongly) connected. The main idea of an algorithm to find the strongly connected components is described in Appendix 4.

For directed graphs G , the function `condensation` provides a simplified graph each vertex of which represents a connected component of G :

```
» Gcond=condensation(G); % directed graph
```

(cf. p. 20). The function is not applicable to undirected graphs, which fall into independent subgraphs if they are not connected.

The function `subgraph` extracts from a graph G those vertices and the edges among these vertices, which are specified in the vector `Vertices`:

```
» Gsub=subgraph(G, Vertices);
```

(cf. Section 2.2.3).

The function `distances` is used to get a matrix with the minimum path lengths among all pairs of vertices. In the matrix `D` obtained from

```
» D=distances(G);
```

the element d_{ij} is the length of a shortest path $P(j \rightarrow i)$ or $P(j - i)$, respectively. If the edges have nonnegative weights, these weights define the edge length and are used to determine the path length. Otherwise, all edges have length 1. For undirected graphs, the matrix `D` is symmetric. For directed graphs, an element $d_{ij} = \text{'Inf'}$ signifies that the vertex i is not reachable from j .

2.4.3 Plotting graphs

The function call

```
» plot(G);
```

plots the graph G in the usual style in a figure. Several options are available to customise the figure. With

```
» plot(G, 'Layout', 'circle');
```

the vertices are placed on a circle centred at the origin with radius 1. If one inputs the x/y -coordinates of the vertices in the vectors `xcoord` and `ycoord`, the function call

```
» plot(G, 'XData', xcoord, 'YData', ycoord);
```

delivers a figure with the vertices placed in the x/y -plane at the specified points. Several options can be used to control the line width, the line style and the markers for the vertices.

The graph plot object can be assigned a handle h by

```
» h=plot(G);
```

to refer to the figure when using other functions. In particular, the function `highlight` can be used to emphasise subgraphs, specific vertices or specific edges. For example

```
» highlight(h, i, 'NodeColor', 'red')
```

changes the colour of the vertex i to red and

```
» highlight(h, T, 'EdgeColor', 'r', 'LineWidth', 1.5)
```

with T being a subgraph of G shows this subgraph in red with the specified line width for the edges.

2.4.4 Graph search

Basic search algorithms. Depth-first search and breadth-first search introduced in Sections 2.3.2 and 2.3.3 can be applied by the functions

```
» V=dfsearch(G, A);
```

```
» V=bfsearch(G, A);
```

with the variable A specifying the root vertex. The result V is the vector of vertices found. The order in which these vertices have been discovered during the search is retained. Hence, for a graph G and a vertex A both search methods lead to the same set of reachable vertices, but the order of these vertices is generally different.

To visualise the working principle of the search algorithms, the following two search events have to be enabled:

- The event 'discovernode' triggers the marking of a vertex, which is symbolised in Fig. 2.13 by $\mathcal{M} \leftarrow A$ for the root vertex and by $\mathcal{M} \leftarrow X$ for a newly found vertex.
- The event 'edgetonew' indicates that an edge $(S - X)$ to an unmarked vertex X has been found.

The function calls

```
» events={'discovernode','edgetonew'};
» T=dfsearch(G,A,events);
```

lead for the graph G of Fig. 2.11 and for $A=1$ to the following table T:

| Event | Node | Edge | EdgeIndex |
|--------------|------|---------|-----------|
| discovernode | 1 | NaN NaN | NaN |
| edgetonew | NaN | 1 2 | 1 |
| discovernode | 2 | NaN NaN | NaN |
| edgetonew | NaN | 2 3 | 3 |
| discovernode | 3 | NaN NaN | NaN |
| edgetonew | NaN | 3 4 | 5 |
| ⋮ | ⋮ | ⋮ | ⋮ |

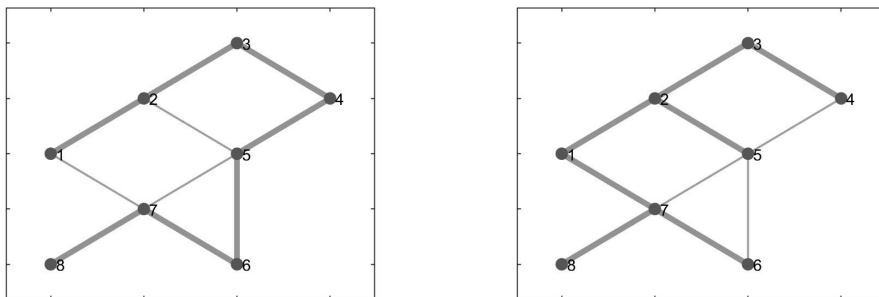


Fig. 2.26: Result of a depth-first search (left) and a breadth-first search (right) in the graph of Fig. 2.11 starting at the vertex $A = 1$

The column Node in the table indicates in which order the vertices of the graph have been found during the search. The edges along which these vertices have been discovered have the edge index shown in the last column. Accordingly, the vector

```
» T.Node(T.Event=='discovernode')
ans =
     1
     2
     3
     ⋮
```

includes all marked vertices like the vector V obtained by the first function call of `dfsearch` above, whereas the vector

```
» T.EdgeIndex(T.Event=='edgetonew')
ans =
     1
     3
     5
     ⋮
```

displays the indices of the edges that have been used by the search algorithm to get reachable vertices. These edges form the tree \mathcal{B} used in Algorithm 2.3 and they are marked in the plot of the graph by the following assignments:

```
» h=plot(G);
» highlight(h, 'Edges', T.EdgeIndex(T.Event=='edgetonew'), ...
    'EdgeColor', 'r', 'LineWidth',1.5, 'MarkerSize',7);
```

(left part of Fig. 2.26).

Both results depicted in Fig. 2.26 should be compared with the right parts of Figs. 2.12 or 2.14, respectively. The figures do not coincide, because the MATLAB implementation of the search algorithms visits the new vertices in the order of their indices beginning with the smallest index. This order is obtained when using the adjacency list (2.22) for the graph representation.

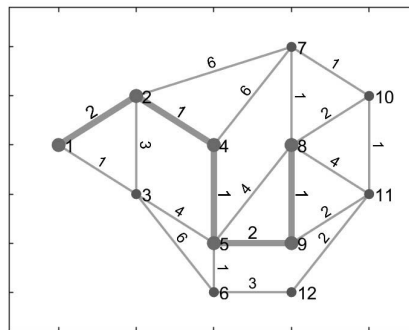


Fig. 2.27: Result of the Dijkstra algorithm for the graph of Fig. 2.16

Dijkstra algorithm. To get shortest paths, in particular by using the Dijkstra algorithm, the function `shortestpath` is available:

```
» [Path,d]=shortestpath(G, A, B, 'Method','positive');
```

The variables `A` and `B` specify the start vertex and the end vertex of the required path. The method named `'positive'` accepts only positive edge weights and applies the Dijkstra algorithm. The result is a shortest path `Path`, which has the length `d`. With

```
» highlight(h,Path,'EdgeColor','r','LineWidth',1.5,...
    'MarkerSize',7);
```

Fig. 2.27 is obtained, after the graph of Fig. 2.16 has been drawn in a figure with handle `h`. If applied with the method `'unweighted'`, the function `shortestpath` ignores the edge weights and performs a breadth-first search.

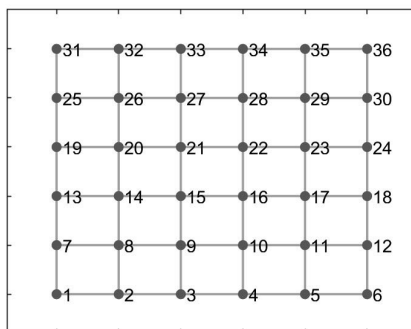


Fig. 2.28: Lattice for graph search

Exercise 2.4* *Graph search methods*



Visualise depth-first search and breadth-first search by drawing the reachability trees that both methods generate for the lattice shown in Fig. 2.28 and compare the results. How do these methods distinguish with respect to the number of vertices that they mark before finding a specific vertex B when starting in a root vertex A . Can you identify a pair (A, B) for which the depth-first search needs a smaller number of search steps than the breadth-first search?

Exercise 2.5 *A travel with the Metro of Barcelona*



Figure 2.29 presents a part of the plan of the underground in the town of Barcelona, Spain. The distances between neighbouring stations are given in kilometers. The edges in the set \mathcal{E} are ordered according to the numbers in parentheses. Determine the shortest path between the stations *Clot* (Vertex 4) and *Passeig de Gràcia* (Vertex 5) or between *Diagonal* (Vertex 3) and *Espanya* (Vertex 8). Extend your MATLAB file to get a program that gives passengers an advice for their travel with the underground between arbitrary stations, which are given as inputs.

Consider your results from a practical viewpoint and answer the question whether your shortest paths are also the quickest ones. Which way would you choose?

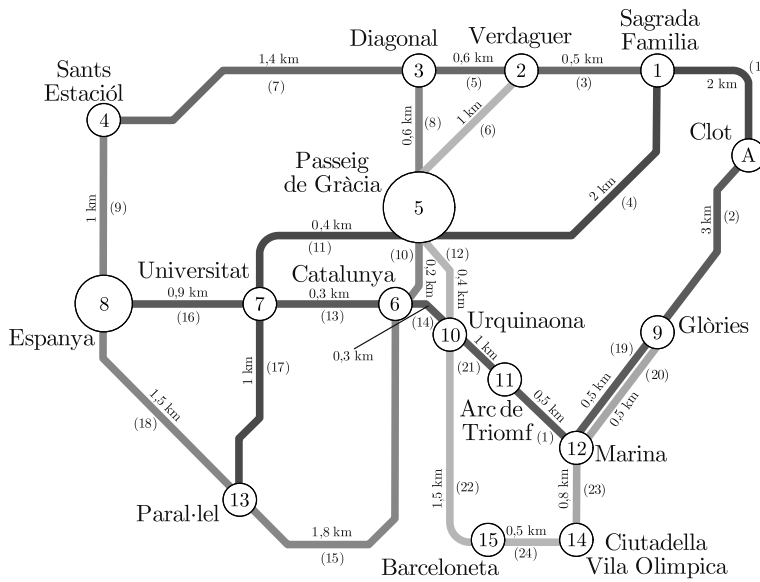


Fig. 2.29: Metro plan of Barcelona

Notes and references

Graph theory has started with the work of L. EULER who investigated the problem that is now called the *Königsberg Bridge Problem* or *Seven Bridges of Königsberg* in a graph-theoretical manner in 1736 [91]. He found that it is impossible to use the seven bridges once and only once while going around the town of Königsberg (now called Kaliningrad in Russia) on a closed walk.

The first book on graph theory has been written by D. KÖNIG in 1936 [162]. Introductions to graph theory, which are easy to understand by engineers, are [30, 92, 120, 286, 329] or the classical book [131] by F. HARARY⁴.

Algorithmic graph theory has been mainly developed in the field of computer science, although the classical book [162] already describes the basic ideas of graph search for a labyrinth problem. Mathematically oriented references about graph search are, for example, [34, 92, 120, 329] most of which give proofs for the main properties of the algorithms like completeness and optimality. References from computer science are [20, 240, 283]. The presentation of the common ideas of graph search algorithms and the algorithms for logic-based knowledge processing presented in Chapter 3 are based on the textbook [196]. The Trémaux algorithm introduced in Section 2.3 has been taken from [329]. According to [162], TRÉMAUX did not publish his algorithm but the first description of this method has been given in [191]. One of the earliest publications on breadth-first search is [226] and for depth-first search reference [314], which is also said to provide the classical algorithm for finding the strongly connected components of a directed graph. Dijkstra's algorithm has been published first in [73]. Algorithm A1.2 has been taken from [237].

⁴ FRANK HARARY (1921 – 2005), American mathematician

3

Graph search in logic-based knowledge processing

The state-space representation of problems and the inference graph visualise decision problems and their ways of solution. Since the search spaces of knowledge processing are generally large, the A algorithm is introduced as a heuristic extension of the search methods for directed graphs, which prefers promising directions.*

3.1 Decision problems and their graph-theoretical way of solution

This chapter deals with discrete decision problems that require to find a sequence of actions to satisfy given requirements for a goal state. Figure 3.1 illustrates the kind of problems to be solved in the “blocks world”, which has been extensively studied in the field of artificial intelligence. Four blocks are piled up as shown on the left-hand side. A robot should move the blocks one by one with the aim to reverse their order. The solution to this decision problem should provide a plan showing the order in which the blocks have to be moved to reach the goal.

The solution to a decision problem has to respect several restrictions. For the blocks-world example, first, only a single block can be moved at a time. Second, only those blocks can be moved that are alone on the ground or that lie on top of a pile. Third, there is room for the deposition of only three blocks on the ground. These restrictions lead, for example, to the fact that the first move can merely change the position of the block A and put it on one of the two empty places.

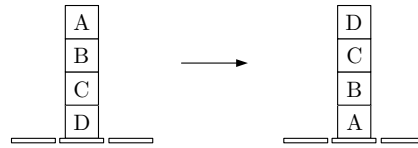


Fig. 3.1: A planning problem

This kind of decision problems require to apply search methods, because the solution steps are known and formalised by the actions, but it is not known in which order these actions have to be activated to reach the goal. This chapter will show how to formulate and solve such decision problems as graph-search problems. As the search space is generally large, Section 3.2 will present heuristic extensions of the original graph-search methods introduced in Chapter 2 to prefer promising directions and to get solutions after fewer search steps. Further heuristics will be discussed for the inference methods investigated in Section 3.4 for logic-based knowledge processing.

In a generalised view, the decision problems considered in this chapter need not to include physical actions like the transport of a block, but the actions may also represent logical deduction steps that are applied to a set of logical formulas and generate new facts about the problem considered. In the formalisation of the problems given below, an *action* is typically represented by preconditions describing the situation in which the action can be applied and by the effect of its execution. The *goal* specifies the situation to be reached.

The main ideas reported in the following sections have been elaborated in the field of artificial intelligence with the notions and concepts used in this chapter. The knowledge about the problem domain is formalised as a set of rules or logical formulas, which are said to constitute a *knowledge base*. This notion leads to the concepts of rule-based systems or logic-based systems in dependence upon the form in which the problem is represented and solved. The solution steps are said to be *inference steps* that derive new knowledge from the knowledge base. Systems that solve such problems are said to be problem solvers or theorem provers.

State-space representation of decision problems. To formalise the decision problems and to make graph-search methods applicable, a state-space representation is introduced. The notion of the state is similarly defined in artificial intelligence for knowledge processing as in control theory for dynamical systems, where a state represents all the information that is necessary to predict the solution steps for a problem or the future behaviour of a system, respectively. In knowledge processing, the *state* includes all the information that is necessary to decide which actions can be performed to bring the problem (hopefully) one step closer to a solution. For example, in the blocks world, the state represents the current configuration of the blocks.

With the notion of the state, the process of solving a decision problem can be interpreted as follows. Starting in the initial state, sequences of actions are applied to generate a sequence of new problem states with the aim to get a goal state. Accordingly, a decision problem is defined by the triple

$$\text{Decision problem: } \mathcal{P} = (z_0, \mathcal{F}, G) \quad (3.1)$$

with

- z_0 – initial state represented by a set of facts that characterise the starting point of the problem,
- \mathcal{F} – set of operators (or actions or successor functions) that generate new states,
- G – a goal predicate (or goal test) giving the conditions to be satisfied by a state, in which a solution to the decision problem has been found.

Note that neither the set of states nor the set of goal states is defined explicitly. The set \mathcal{Z} of possible states appears when applying all operators to all states obtained. During the search for a solution, a part of \mathcal{Z} is explicitly generated. Operators generally have a precondition that the state has to satisfy for the operator to be applicable. If an operator $f \in \mathcal{F}$ is applicable to the state z , it yields the successor state which belongs to \mathcal{Z} :

$$z' = f(z).$$

Likewise, the set $\mathcal{Z}_F \subset \mathcal{Z}$ of goal states is not explicitly defined, but it consists of all states $z \in \mathcal{Z}$ that satisfy the goal predicate:

$$G(z) = T.$$

A predicate is a condition that can only have the truth values T (true) or F (false).

This representation of decision problems leads to the interpretation of the way of solution as a search through a directed graph

$$\frac{\text{State-space representation}}{\text{of a problem:}} \quad \vec{\mathcal{G}} = (\mathcal{Z}, \mathcal{E}) \quad (3.2)$$

with

- \mathcal{Z} – set of problem states,
- \mathcal{E} – set of directed edges representing the application of an operator.

Any edge $(z \xrightarrow{f} z')$ constitutes the application of an operator $f \in \mathcal{F}$ to the state z and it is sometimes labelled by the operator f used (Fig. 3.2). The resulting graph is, in general, not a tree, because a problem state may be reached by more than one sequence of operator executions. It leads to the following reformulation of a decision problem as a graph-search problem:

Graph-theoretical interpretation of a decision problem: Find a path in the search space $\vec{\mathcal{G}} = (\mathcal{Z}, \mathcal{E})$ of the decision problem \mathcal{P} from the initial state z_0 to a goal state z that satisfies the condition $G(z) = T$.

The existence of such a path shows that the decision problem is solvable. The path itself represents the sequence of actions that solves the problem.

Notions, which have been introduced in Chapter 2 for graph-search problems, are directly applicable to this interpretation of decision problems. The directed graph $\vec{\mathcal{G}}$, which appears by

