

Algorithms on Trees and Graphs

Springer-Verlag Berlin Heidelberg GmbH

Gabriel Valiente

Algorithms on Trees and Graphs

With 157 Figures



Springer

Prof. Dr. Gabriel Valiente
Technical University of Catalonia
Department of Software
Jordi Girona, 1- 3 (Module C6)
08034 Barcelona
Spain

Library of Congress Cataloging-in-Publication Data applied for
Die Deutsche Bibliothek-CIP-Einheitsaufnahme
Valiente, Gabriel: Algorithms on trees and graphs / Gabriel Valiente

ISBN 978-3-642-07809-5 ISBN 978-3-662-04921-1 (eBook)
DOI 10.1007/978-3-662-04921-1

ACM Computing Classification (1998) F.2, G.2.2, G.4, D.1

ISBN 978-3-642-07809-5

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilm or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag Berlin Heidelberg GmbH. Violations are liable for prosecution under the German Copyright Law.

© Springer-Verlag Berlin Heidelberg 2002
Originally published by Springer-Verlag Berlin Heidelberg New York in 2002
Softcover reprint of the hardcover 1st edition 2002

The use of designations, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Cover Design: KünkelLopka, Heidelberg
Typesetting: Computer to film by author's data
Printed on acid-free paper SPIN 10855180 45/3142PS 5 4 3 2 1 0

To Daina, Kristiāns, and Dāvis

Preface

Graph algorithms, a long-established subject in mathematics and computer science curricula, are also of much interest to disciplines such as computational molecular biology and computational chemistry. This book goes beyond the *classical* graph problems of shortest paths, spanning trees, flows in networks, and matchings in bipartite graphs, and addresses further algorithmic problems of practical application of trees and graphs. Much of the material presented in the book is only available in the specialized research literature.

The book is structured around the fundamental problem of isomorphism. Tree isomorphism is covered in much detail, together with the related problems of subtree isomorphism, maximum common subtree isomorphism, and tree comparison. Graph isomorphism is also covered in much detail, together with the related problems of subgraph isomorphism, maximal common subgraph isomorphism, and graph edit distance. Building blocks for solving some of these isomorphism problems are algorithms for finding maximal and maximum cliques.

Most intractable graph problems of practical application are not even approximable to within a constant bound, and several of the isomorphism problems addressed in this book are no exception. The book can thus be seen as a companion to recent texts on approximation algorithms [15, 344], but also as a complement to previous texts on combinatorial and graph algorithms [69, 78, 107, 127, 130, 203, 232, 236, 229, 255, 271, 304, 321, 336, 367].

The book is conceived on the ground of first, introducing simple algorithms for these problems in order to develop some intuition before moving on to more complicated algorithms from the research literature, and second, stimulating graduate research on tree and graph

algorithms by providing, together with the underlying theory, a solid basis for experimentation and further development.

Algorithms are presented on an intuitive basis, followed by a detailed exposition in a literate programming style. Correctness proofs are also given, together with a worst-case analysis of the algorithms. Further, full C++ implementation of all the algorithms using the LEDA library of efficient data structures and algorithms is given throughout the book. These implementations include result checking of implementation correctness using correctness certificates.

The choice of LEDA, which is becoming a defacto standard for graduate courses on graph algorithms throughout the world, is not casual, because it allows the student, lecturer, researcher, and practitioner to complement algorithmic graph theory with actual implementation and experimentation, building upon a thorough library of efficient implementations of modern data structures and fundamental algorithms.

An interactive demonstration including animations of all the algorithms using LEDA is given in an appendix and is available on a web site, at <http://www.lsi.upc.es/~valiente/algorithm/>, together with the source code of all the algorithms. The interactive demonstration also includes visual checkers of implementation correctness.

Structure

The book is divided into four parts. Part I is of an introductory nature and consists of two chapters. Chap. 1 includes a review of basic graph-theoretical notions and results used throughout the book, a brief primer of literate programming, and an exposition of the implementation correctness approach by result checking using correctness certificates.

Chap. 2 is devoted exclusively to the fundamental algorithmic techniques used in the book: backtracking, branch-and-bound, divide-and-conquer, and dynamic programming. These techniques are illustrated by means of a running example: algorithms for the tree edit distance problem.

Part II also consists of two chapters. Chap. 3 addresses the most common methods for traversing general, rooted trees: depth-first prefix leftmost (preorder), depth-first prefix rightmost, depth-first postfix leftmost (postorder), depth-first postfix rightmost, breadth-first leftmost (top-down), breadth-first rightmost, and bottom-up traversal. Tree drawing is also discussed as an application of tree traversal methods.

Chap. 4 addresses several isomorphism problems on ordered and unordered trees: tree isomorphism, subtree isomorphism, and maximum common subtree isomorphism. Computational molecular biology is also discussed as an application of the different isomorphism problems on trees.

Part III consists of three chapters. Chap. 5 addresses the most common methods for traversing graphs: depth-first and breadth-first traversal, which respectively generalize depth-first prefix leftmost (preorder) and breadth-first leftmost (top-down) tree traversal. Leftmost depth-first traversal of an undirected graph, a particular case of depth-first traversal, is also discussed. Isomorphism of ordered graphs is also discussed as an application of graph traversal methods.

Chap. 6 addresses the related problems of finding cliques, independent sets, and vertex covers in trees and graphs. Multiple alignment of protein sequences in computational molecular biology is also discussed as an application of clique algorithms.

Chap. 7 addresses several isomorphism problems on graphs: graph isomorphism, graph automorphism, subgraph isomorphism, and maximal common subgraph isomorphism. Chemical structure search is also discussed as an application of the different graph isomorphism problems.

Part IV consists of three appendices, followed by bibliographic references and an index. Appendix A gives an overview of LEDA, including a simple C++ representation of trees as LEDA graphs, and a C++ implementation of radix sort using LEDA. The interactive demonstration of graph algorithms presented throughout the book is put together in Appendix B. Finally, Appendix C contains a complete index to all program modules described in the book.

Audience

This book is suitable for use in upper undergraduate and graduate level courses on algorithmic graph theory. It can also be used as a supplementary text in basic undergraduate and graduate-level courses on algorithms and data structures, and in computational molecular biology and computational chemistry courses as well. Some basic knowledge of discrete mathematics, data structures, algorithms, and programming at the undergraduate level is assumed.

Acknowledgments

This book is based on lectures taught at the Technical University of Catalonia, Barcelona between 1996 and 2002, and the University of Latvia, Riga between 2000 and 2002.

Numerous colleagues at the Technical University of Catalonia have influenced the approach to data structures and algorithms on trees and graphs expressed in this book. In particular, the author would like to thank José L. Balcázar, Rafel Casas, Jordi Cortadella, Josep Díaz, Conrado Martínez, Xavier Messeguer, Roberto Nieuwenhuis, Fernando Orejas, Jordi Petit, Salvador Roura, and Maria Serna, to name just a few. It has been a pleasure to share teaching and research experiences with them over the last several years.

The author would also like to thank Ricardo Baeza-Yates, Francesc Rosselló, and Steven Skiena, for their standing support and encouragement, and Hans-Jörg Kreowski, for supporting basic and applied research on graph algorithms within the field of graph transformation.

It has been a pleasure for the author to work out editorial matters together with Alfred Hofmann, Ingeborg Mayer, and Peter Straßer of Springer-Verlag.

Special thanks go to the Technical University of Catalonia for funding the sabbatical year during which this book was written, and to the Institute of Mathematics and Computer Science, University of Latvia, in particular to Jānis Bārzdiņš and Rūsinš Freivalds, for hosting the sabbatical visit.

Contents

Part I. Introduction

1. Introduction	3
1.1 Trees and Graphs	3
1.2 Literate Programming	23
1.3 Implementation Correctness	28
1.4 Representation of Trees and Graphs	34
Summary	49
Bibliographic Notes	49
Review Problems	50
Exercises	51
2. Algorithmic Techniques	55
2.1 The Tree Edit Distance Problem	55
2.2 Backtracking	70
2.3 Branch-and-Bound	80
2.4 Divide-and-Conquer	85
2.5 Dynamic Programming	94
Summary	107
Bibliographic Notes	107
Review Problems	108
Exercises	110

Part II. Algorithms on Trees

3. Tree Traversal	113
3.1 Preorder Traversal of a Tree	113
3.2 Postorder Traversal of a Tree	119

3.3	Top-Down Traversal of a Tree	124
3.4	Bottom-Up Traversal of a Tree	129
3.5	Applications	135
	Summary	145
	Bibliographic Notes	145
	Review Problems	146
	Exercises	147
4.	Tree Isomorphism	151
4.1	Tree Isomorphism	151
4.2	Subtree Isomorphism	170
4.3	Maximum Common Subtree Isomorphism	206
4.4	Applications	239
	Summary	242
	Bibliographic Notes	244
	Review Problems	246
	Exercises	249

Part III. Algorithms on Graphs

5.	Graph Traversal	255
5.1	Depth-First Traversal of a Graph	255
5.2	Breadth-First Traversal of a Graph	276
5.3	Applications	288
	Summary	294
	Bibliographic Notes	294
	Review Problems	295
	Exercises	297
6.	Clique, Independent Set, and Vertex Cover	299
6.1	Cliques, Maximal Cliques, and Maximum Cliques	299
6.2	Maximal and Maximum Independent Sets	322
6.3	Minimal and Minimum Vertex Covers	332
6.4	Applications	341
	Summary	346
	Bibliographic Notes	346

Review Problems	347
Exercises	348
7. Graph Isomorphism	351
7.1 Graph Isomorphism	351
7.2 Graph Automorphism	363
7.3 Subgraph Isomorphism	367
7.4 Maximal Common Subgraph Isomorphism.....	377
7.5 Applications	389
Summary	392
Bibliographic Notes	392
Review Problems	394
Exercises	395

Part IV. Appendices

A. An Overview of LEDA	401
A.1 Introduction	401
A.2 Data Structures	402
A.3 Fundamental Graph Algorithms	424
A.4 A Simple Representation of Trees	426
A.5 A Simple Implementation of Radix Sort	434
Bibliographic Notes	445
B. Interactive Demonstration of Graph Algorithms	447
C. Program Modules	453
Bibliography	459
Index	489

Part I

Introduction

1. Introduction

Many algorithms published in journals are presented in a rather cryptic manner. The main concepts are given, but there is little in way of explanation by means of examples, and even less may be said about implementation.

—Alfs T. Berztiss [40]

Trees and graphs count among the most useful mathematical abstractions and, at the same time, the most common combinatorial structures in computer science. Both the graph-theoretical notions underlying algorithms on trees and graphs and appropriate data structures for their representation are the subject of this chapter. Further, a brief primer of the *literate programming* style adopted in this book for the presentation of algorithms and data structures is included in this chapter, together with a brief exposition of the implementation correctness approach by result checking using correctness certificates. The reader who is not familiar with the LEDA library of efficient C++ data structures and algorithms is referred to Appendix A.

1.1 Trees and Graphs

The notion of graph which is most useful in computer science is that of directed graph, or just graph. A graph is a combinatorial structure consisting of a finite nonempty set of objects, called *vertices*, together with a finite (possibly empty) set of ordered pairs of vertices, called *directed edges* or *arcs*.

Definition 1.1. *A graph $G = (V, E)$ consists of a finite nonempty set V of vertices and a finite set $E \subseteq V \times V$ of arcs. The order of a graph*

$G = (V, E)$, denoted by n , is the number of vertices, $n = |V|$ and the size, denoted by m , is the number of arcs, $m = |E|$. An arc $e = (v, w)$ is said to be **incident** with vertices v and w , where v is the **source** and w the **target** of arc e , and vertices v and w are said to be **adjacent**. Edges (u, v) and (v, w) are said to be **adjacent**, as are edges (u, v) and (w, v) , and also edges (v, u) and (v, w) .

Graphs are often drawn as a set of points in the plane and a set of arrows, each of which joins two (not necessarily different) points. In a drawing of a graph $G = (V, E)$, each vertex $v \in V$ is drawn as a point or a small circle, and each arc $(v, w) \in E$ is drawn as an arrow from the point or circle of vertex v to the point or circle corresponding to vertex w .

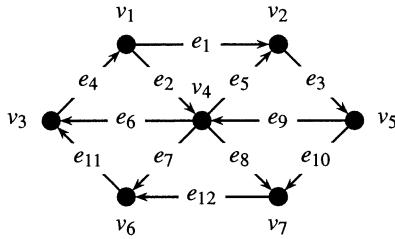


Fig. 1.1. A graph of order 7 and size 12.

Example 1.2. The graph $G = (V, E)$ of Fig. 1.1 has order 7 and size 12. The vertex set is $V = \{v_1, \dots, v_7\}$ and the arc set is $E = \{e_1, \dots, e_{12}\}$, where $e_1 = (v_1, v_2)$, $e_2 = (v_1, v_4)$, $e_3 = (v_2, v_5)$, $e_4 = (v_3, v_1)$, $e_5 = (v_4, v_2)$, $e_6 = (v_4, v_3)$, $e_7 = (v_4, v_6)$, $e_8 = (v_4, v_7)$, $e_9 = (v_5, v_4)$, $e_{10} = (v_5, v_7)$, $e_{11} = (v_6, v_3)$, $e_{12} = (v_7, v_6)$.

A vertex has two degrees in a graph, one given by the number of arcs coming into the vertex and the other given by the number of arcs in the graph going out of the vertex.

Definition 1.3. The **indegree** of a vertex v in a graph $G = (V, E)$ is the number of arcs in G whose target is v , that is, $\text{indeg}(v) = |\{(u, v) \mid (u, v) \in E\}|$. The **outdegree** of a vertex v in a graph $G = (V, E)$ is the number of arcs in G whose source is v , that is, $\text{outdeg}(v) = |\{(v, w) \mid$

$(v, w) \in E\}].$ The **degree** of a vertex v in a graph $G = (V, E)$ is the sum of the indegree and the outdegree of the vertex, that is, $\deg(v) = \text{indeg}(v) + \text{outdeg}(v).$

Example 1.4. The degrees of the vertices in the graph of Fig. 1.1 are the following:

vertex	v_1	v_2	v_3	v_4	v_5	v_6	v_7	sum
indegree	1	2	2	2	1	2	2	12
outdegree	2	1	1	4	2	1	1	12

As can be seen in Example 1.4, the sum of the indegrees and the sum of the outdegrees of the vertices of the graph in Fig. 1.1 are both equal to 12, the number of arcs of the graph. As a matter of fact, there is a basic relationship between the size of a graph and the degrees of its vertices, which will prove to be very useful in analyzing the computational complexity of algorithms on graphs.

Theorem 1.5. Let $G = (V, E)$ be a graph with n vertices and m arcs, and let $V = \{v_1, \dots, v_n\}$. Then,

$$\sum_{i=1}^n \text{indeg}(v_i) = \sum_{i=1}^n \text{outdeg}(v_i) = m.$$

Proof. Let $G = (V, E)$ be a graph. Every arc $(v_i, v_j) \in E$ contributes one to $\text{indeg}(v_j)$ and one to $\text{outdeg}(v_i)$. \square

Walks, trails, and paths in a graph are alternating sequences of vertices and arcs in the graph such that each arc in the sequence is preceded by its source vertex and followed by its target vertex. Trails are walks having no repeated arcs, and paths are trails having no repeated vertices.

Definition 1.6. A **walk** from vertex v_i to vertex v_j in a graph is an alternating sequence

$$[v_i, e_{i+1}, v_{i+1}, e_{i+2}, \dots, v_{j-1}, e_j, v_j]$$

of vertices and arcs in the graph, such that $e_k = (v_{k-1}, v_k)$ for $k = i+1, \dots, j$. A **trail** is a walk with no repeated arcs, and a **path** is a trail with no repeated vertices (except, possibly, the initial and final vertices). The **length** of a walk, trail or path is the number of arcs in the sequence.

Since an arc in a graph is uniquely determined by its source and target vertices, a walk, trail or path can be abbreviated by just enumerating either the vertices $[v_i, v_{i+1}, \dots, v_{j-1}, v_j]$ or the arcs $[e_{i+1}, e_{i+2}, \dots, e_j]$ in the alternating sequence $[v_i, e_{i+1}, v_{i+1}, e_{i+2}, \dots, v_{j-1}, e_j, v_j]$ of vertices and arcs.

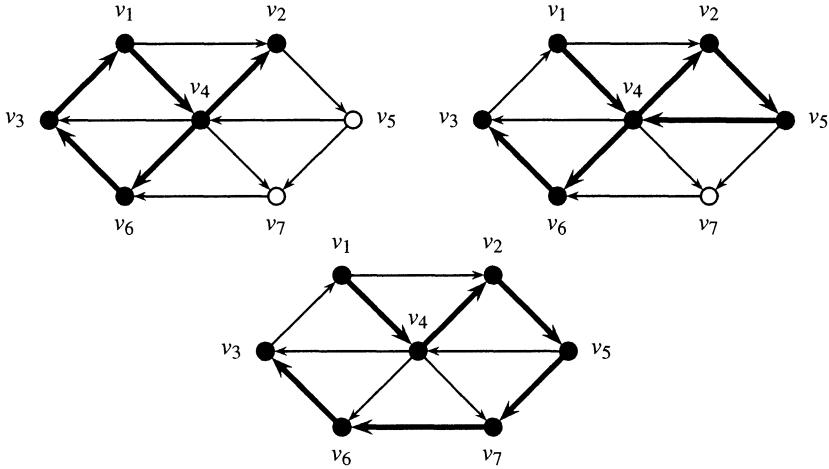


Fig. 1.2. A walk $[v_1, v_4, v_6, v_3, v_1, v_4, v_2]$, a trail $[v_1, v_4, v_2, v_5, v_4, v_6, v_3]$, and a path $[v_1, v_4, v_2, v_5, v_7, v_6, v_3]$ in the graph of Fig. 1.1.

Example 1.7. The vertex sequence $[v_1, v_4, v_6, v_3, v_1, v_4, v_2]$ in Fig. 1.2 is a walk of length 6 which is not a trail, $[v_1, v_4, v_2, v_5, v_4, v_6, v_3]$ is a trail of length 6 which is not a path, and $[v_1, v_4, v_2, v_5, v_7, v_6, v_3]$ is a path of length 6.

Walks are closed if their initial and final vertices coincide.

Definition 1.8. A walk, trail, or path $[v_i, e_{i+1}, v_{i+1}, e_{i+2}, \dots, v_{j-1}, e_j, v_j]$ is said to be **closed** if $v_i = v_j$. A **cycle** is a closed path of length at least one.

Example 1.9. The vertex sequence $[v_1, v_4, v_6, v_3, v_1]$ in Fig. 1.3 is a cycle of length 4.

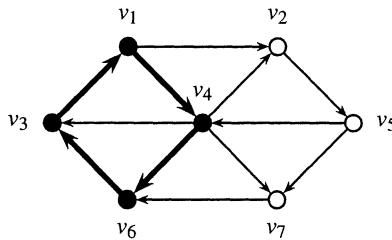


Fig. 1.3. A cycle $[v_1, v_4, v_6, v_3, v_1]$ in the graph of Fig. 1.1.

The combinatorial structure of a graph encompasses two notions of substructure. A subgraph of a graph is just a graph whose vertex and arc sets are contained in the vertex and arc sets of the given graph, respectively. The subgraph of a graph induced by a subset of its vertices has as arcs the set of arcs in the given graph whose source and target belong to the subset of vertices.

Definition 1.10. Let $G = (V, E)$ be a graph, and let $W \subseteq V$. A graph (W, S) is a **subgraph** of G if $S \subseteq E$. The subgraph of G **induced** by W is the graph $(W, E \cap W \times W)$.

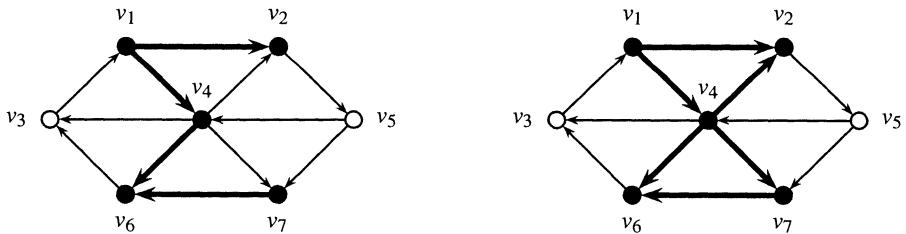


Fig. 1.4. A subgraph and an induced subgraph of the graph of Fig. 1.1.

Example 1.11. The subgraph with vertex set $\{v_1, v_2, v_4, v_6, v_7\}$ and arc set $\{(v_1, v_2), (v_1, v_4), (v_4, v_6), (v_7, v_6)\}$ shown in Fig. 1.4, is not an induced subgraph. The subgraph induced by $\{v_1, v_2, v_4, v_6, v_7\}$ has arc set $\{(v_1, v_2), (v_1, v_4), (v_4, v_2), (v_4, v_6), (v_4, v_7), (v_7, v_6)\}$.

Undirected Graphs

The notion of graph which is most often found in mathematics is that of undirected graph. Unlike the directed edges or arcs of a graph, edges of an undirected graph have no direction associated to them and therefore no distinction is made between source and target vertices of an edge. In a mathematical sense, an undirected graph consists of a set of vertices and a finite set of undirected edges, where each edge has a set of one or two vertices associated to it. In the computer science view of undirected graphs, though, an undirected graph is the particular case of a directed graph in which for every arc (v, w) of the graph, the reversed arc (w, v) also belongs to the graph. Undirected graphs are also called *bidirected*.

Definition 1.12. A graph $G = (V, E)$ is *undirected* if $(v, w) \in E$ implies $(w, v) \in E$, for all $v, w \in V$.

Undirected graphs are often drawn as a set of points in the plane and a set of line segments, each of which joins two (not necessarily different) points. In a drawing of an undirected graph $G = (V, E)$, each vertex $v \in V$ is drawn as a point or a small circle and each pair of counter-parallel arcs $(v, w), (w, v) \in E$ is drawn as a line segment between the points or circles corresponding to vertices v and w .

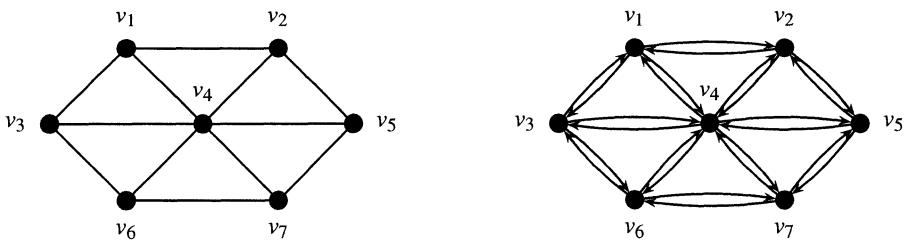


Fig. 1.5. The undirected graph underlying the graph of Fig. 1.1. The standard presentation is given in the drawing on the left, while the understanding of undirected edges as pairs of counter-parallel arcs is emphasized in the drawing on the right.

Example 1.13. The undirected graph underlying the graph of Fig. 1.1 is shown in Fig. 1.5.

The terminology of directed graphs carries over to undirected graphs, although a few differences deserve mention. First of all, since an edge in an undirected graph is understood as a pair of counter-parallel arcs in the corresponding bidirected graph, the number of arcs coming into a given vertex and the number of arcs going out of the vertex coincide and therefore no distinction is made between the indegree and the outdegree of a vertex. That is, the *degree* of a vertex in an undirected graph is equal to the indegree and the outdegree of the vertex in the corresponding bidirected graph. For the same reason, the size of an undirected graph is half the size of the corresponding bidirected graph.

Definition 1.14. *The degree of a vertex v in an undirected graph $G = (V, E)$ is the number of edges in G that are incident with v . The degree sequence of G is the sequence of n nonnegative integers obtained by arranging the vertex degrees in nondecreasing order.*

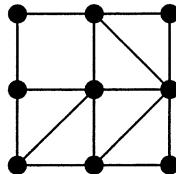


Fig. 1.6. Undirected graph with degree sequence $[2, 2, 2, 3, 3, 4, 4, 5, 5]$.

Example 1.15. There are three vertices of degree 2 in the graph of Fig. 1.6, two vertices of degree 3, two vertices of degree 4, and two vertices of degree 5. The degree sequence of the graph is $[2, 2, 2, 3, 3, 4, 4, 5, 5]$.

A walk, trail, or path in an undirected graph is a walk, trail, or path, respectively, in the corresponding bidirected graph. Another important graph-theoretical notion is that of connected and disconnected graphs. An undirected graph is *connected* if every pair of its vertices is joined by some walk, and an undirected graph that is not connected is said to be *disconnected*. On the other hand, a graph is connected if for all vertices v and w in the graph, there is a walk from v to w , and it is

strongly connected if there are walks from v to w and from w back to v , for all vertices v and w in the graph.

Definition 1.16. An undirected graph $G = (V, E)$ is **connected** if for every pair of vertices $v, w \in V$, there is a walk between v and w . A graph is **connected** if the underlying undirected graph is connected. A graph G is **strongly connected** if for every pair of vertices $v, w \in V$, there are walks from v to w and from w to v .

Connectivity is an equivalence relation on the vertex set of a graph, which induces a partition of the graph into subgraphs induced by connected vertices, called *connected components* or just *components*.

Definition 1.17. A **component** of an undirected graph G is a connected subgraph of G which is not properly contained in any other connected subgraph of G . A **strong component** of a graph G is a strongly connected subgraph of G which is not properly contained in any other strongly connected subgraph of G .

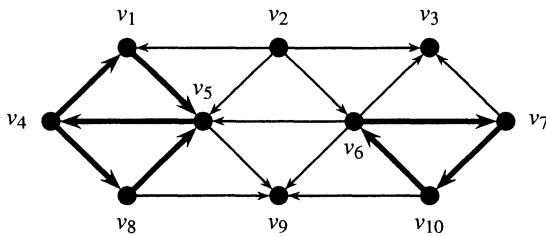


Fig. 1.7. A graph with five strong components whose underlying undirected graph is connected.

Example 1.18. The graph of Fig. 1.7 has five strong components, induced respectively by the vertex sets $\{v_1, v_4, v_5, v_8\}$, $\{v_2\}$, $\{v_3\}$, $\{v_6, v_7, v_{10}\}$, and $\{v_9\}$. The underlying undirected graph is, however, connected.

Some common families of undirected graphs are trees, complete graphs, path graphs, cycle graphs, wheel graphs, bipartite graphs, and regular graphs. A *tree* is a connected graph having no cycles.

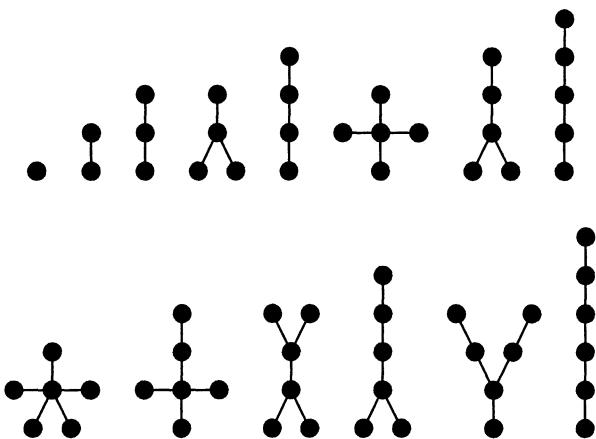


Fig. 1.8. The first 14 undirected trees.

Definition 1.19. An undirected graph $G = (V, E)$ is said to be an **undirected tree** if it is connected and has no cycles.

Example 1.20. The undirected trees with one, two, three, four, five, and six vertices are shown in Fig. 1.8.

In a *complete* graph, every pair of distinct vertices is joined by an edge.

Definition 1.21. An undirected graph $G = (V, E)$ is said to be a **complete graph** if for all $v, w \in V$ with $v \neq w$, $(v, w) \in E$. The complete graph on n vertices is denoted by K_n .

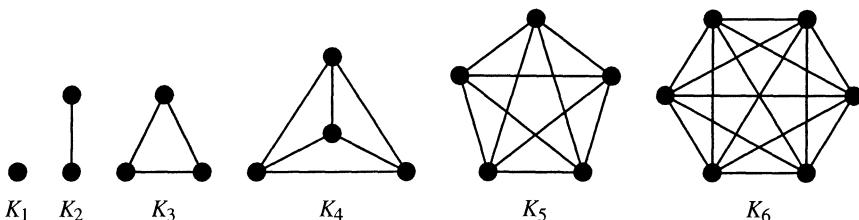


Fig. 1.9. The first six complete graphs.

Example 1.22. The complete graphs with one, two, three, four, five, and six vertices are shown in Fig. 1.9.

A *path graph* can be drawn such that all vertices lie on a straight line.

Definition 1.23. A connected undirected graph $G = (V, E)$ with $n = m + 1$ is said to be a **path graph** if it can be drawn such that all vertices lie on a straight line. The path graph on n vertices is denoted by P_n .

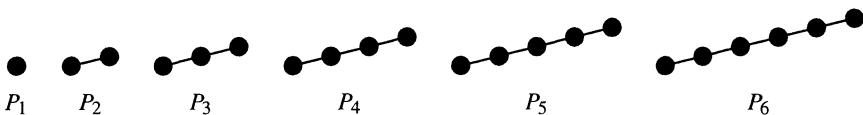


Fig. 1.10. The first six path graphs.

Example 1.24. The path graphs with one, two, three, four, five, and six vertices are shown in Fig. 1.10.

A *cycle graph* can be drawn such that all vertices lie on a circle.

Definition 1.25. A connected undirected graph $G = (V, E)$ with $n = m$ is said to be a **cycle graph** if it can be drawn such that all vertices lie on a circle. The cycle graph on n vertices is denoted by C_n .

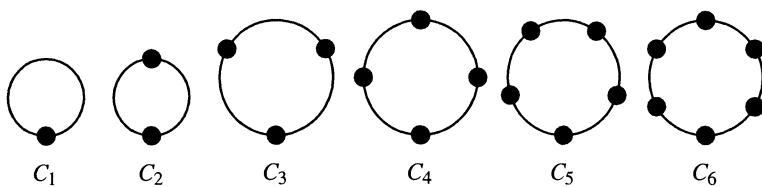


Fig. 1.11. The first six cycle graphs.

Example 1.26. The cycle graphs with one, two, three, four, five, and six vertices are shown in Fig. 1.11.

A *wheel graph* has a distinguished (inner) vertex which is connected to all other (outer) vertices, and it can be drawn such that all outer vertices lie on a circle centered at the inner vertex.

Definition 1.27. A connected undirected graph $G = (V, W)$ is said to be a **wheel graph** if it can be drawn such that all but a single vertex lie on a circle centered at the single vertex, which is connected to all other vertices. The wheel graph with n outer vertices is denoted by W_{n+1} .

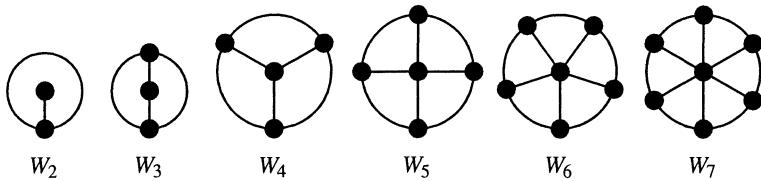


Fig. 1.12. The first six wheel graphs.

Example 1.28. The wheel graphs with one, two, three, four, five, and six outer vertices are shown in Fig. 1.12.

Another common family of undirected graphs are the *bipartite* graphs. The vertex set of a bipartite graph can be partitioned into two subsets in such a way that every edge of the graph joins a vertex of one subset with a vertex of the other subset. In a *complete bipartite* graph, every vertex of one subset is joined by some edge with every vertex of the other subset.

Definition 1.29. An undirected graph $G = (V, W)$ is said to be a **bipartite graph** if V can be partitioned into two subsets U and W such that for all $(v, w) \in E$, either $v \in U$ and $w \in W$, or $v \in W$ and $w \in U$, and it is said to be a **complete bipartite graph** if, furthermore, $(u, w), (w, u) \in E$ for all $u \in U$ and $w \in W$. The complete bipartite graph on $p + q$ vertices, where the subset U has p vertices and the subset W has q vertices, is denoted by $K_{p,q}$.

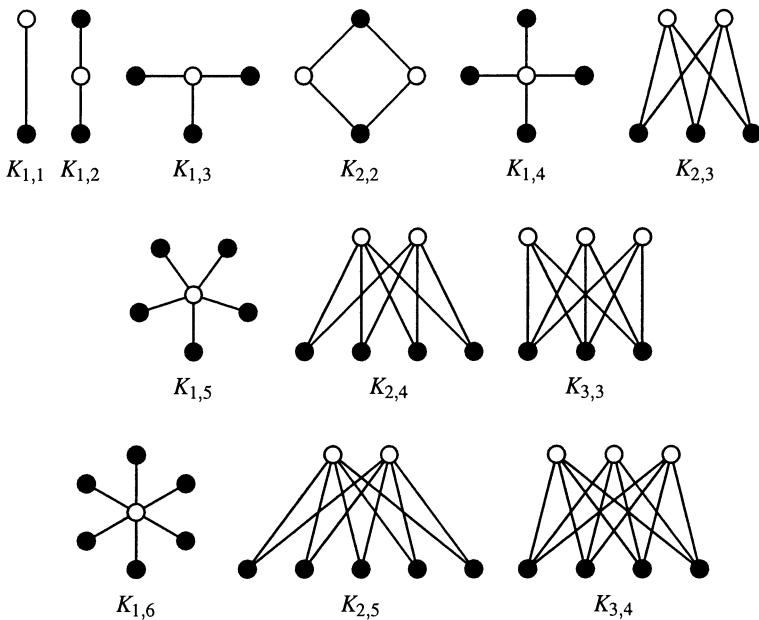


Fig. 1.13. The first 12 complete bipartite graphs.

Example 1.30. The complete bipartite graphs with two, three, four, five, six, and seven vertices are shown in Fig. 1.13. The bipartite sets are distinguished by drawing vertices either in white or in black. Then, every edge joins a white vertex with a black vertex.

A natural generalization of bipartite graphs are the *k-partite* graphs. The vertex set of a *k*-partite graph can be partitioned into *k* subsets in such a way that every edge of the graph joins a vertex of one subset with a vertex of another subset.

Definition 1.31. An undirected graph $G = (V, W)$ is said to be a *k-partite graph* if V can be partitioned into $k \geq 2$ subsets V_1, V_2, \dots, V_k such that for all $(v, w) \in E$ with $v \in V_i$ and $w \in V_j$, it holds that $i \neq j$.

Complete graphs and cycle graphs are trivial examples of *regular* graphs. In a regular graph, all vertices have the same degree.

Definition 1.32. An undirected graph $G = (V, W)$ is said to be a *regular graph* if $\deg(v) = \deg(w)$ for all $v, w \in V$.



Fig. 1.14. Regular graphs \$G_1\$ and \$G_2\$ of the same order.

Example 1.33. The regular graphs \$G_1\$ and \$G_2\$ shown in Fig. 1.14 share the same degree sequence, [2, 2, 2, 2, 2, 2].

Labeled Graphs

Most applications of graphs in computer science involve *labeled* graphs, where vertices and edges have additional attributes such as color or weight. For a labeled graph \$G = (V, E)\$, the label of a vertex \$v \in V\$ is denoted by \$G[v]\$ and the label of an edge \$(v, w) \in E\$ is denoted by \$G[v, w]\$.

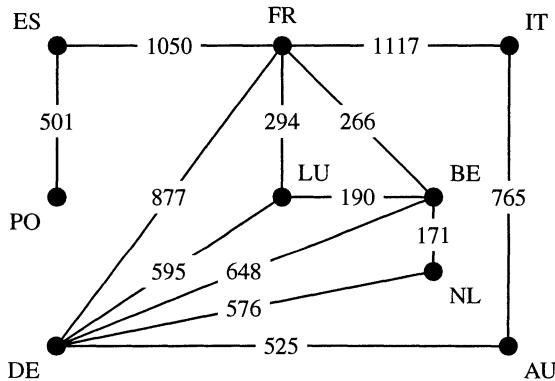


Fig. 1.15. A labeled undirected graph of geographical adjacencies between some European states.

Example 1.34. An undirected graph of the geographical adjacencies between some of the first European states to adopt the European currency is shown in Fig. 1.15. Vertices represent European states, and are labeled with a standard abbreviation for the name of the state. There is an edge between two vertices if the corresponding states share a border. Edges are labeled with the distance in kilometers between the capital cities of the states they join.

Ordered Graphs

An ordered graph is a graph in which the relative order of the adjacent vertices is fixed for each vertex. Ordered graphs arise when a graph is drawn or *embedded* on a certain surface, for instance, in the Euclidean plane.

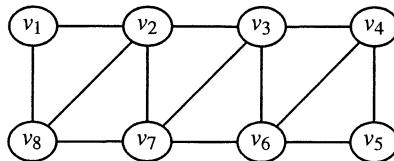


Fig. 1.16. An ordered undirected graph. The relative order of the vertices adjacent to a given vertex reflects the counterclockwise ordering of the outgoing edges of the vertex in the drawing.

Example 1.35. The relative order of the vertices adjacent to each of the vertices of the ordered undirected graph shown in Fig. 1.16 reflects the counterclockwise ordering of the outgoing edges of the vertex in the drawing. For instance, the ordered sequence of vertices $[v_1, v_8, v_7, v_3]$ adjacent to vertex v_2 reflects the counterclockwise ordering $\{v_2, v_1\}, \{v_2, v_8\}, \{v_2, v_7\}, \{v_2, v_3\}$ of the edges incident with vertex v_2 in the drawing of the graph.

Trees

The families of undirected graphs introduced above have a directed graph counterpart. In particular, while the notion of tree most often found in discrete mathematics is that of an undirected tree, the notion of tree which is most useful in computer science is that of a rooted directed tree, or just a tree. A tree is the particular case of a graph in which there is a distinguished vertex, called the root of the tree, such that there is a unique walk from the root to any vertex of the tree. The vertices of a tree are called *nodes*.

Definition 1.36. A connected graph $G = (V, E)$ is said to be a *tree* if the underlying undirected graph has no cycles and there is a distinguished node $r \in V$, called the *root* of the tree and denoted by $\text{root}[T]$

such that for all nodes $v \in V$, there is a path in G from the root r to node v .

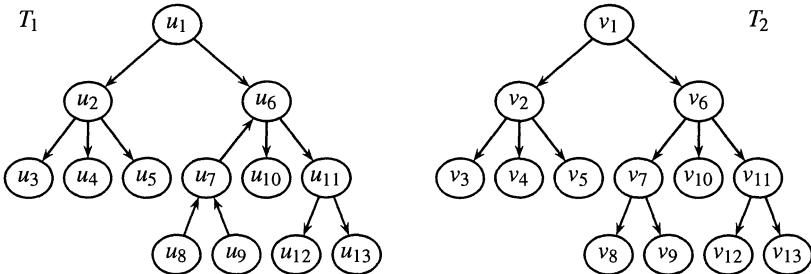


Fig. 1.17. Graph T_1 is not a rooted tree, although it is connected and has no cycles. Graph T_2 is a tree rooted at node v_1 .

Example 1.37. Two connected graphs are shown in Fig. 1.17. Graph T_1 is not a tree rooted at node u_1 , although the underlying undirected graph has no cycles. As a matter of fact, there is no path in T_1 from node u_1 to any of the nodes u_7 , u_8 , and u_9 . Graph T_2 , on the other hand, is indeed a tree rooted at node v_1 .

The existence of a unique path in a tree from the root to any other node imposes a hierarchical structure in the tree. The root lies at the top, and nodes can be partitioned in hierarchical levels according to their distance from the root of the tree.

Definition 1.38. Let $T = (V, E)$ be a tree. The **depth** of node $v \in V$, denoted by $\text{depth}[v]$, is the length of the unique path from the root node $\text{root}[T]$ to node v , for all nodes $v \in V$. The depth of T is the maximum among the depths of all nodes $v \in V$.

The hierarchical structure embodied in a tree leads to further distinctions among the nodes of the tree: *parent* nodes are joined by arcs to their *children* nodes, nodes sharing the same parent are called *siblings*, and nodes having no children are called *leaves*.

Definition 1.39. Let $T = (V, E)$ be a tree. Node $v \in V$ is said to be the **parent** of node $w \in V$, denoted by $\text{parent}[w]$, if $(v, w) \in E$ and, in

such a case, node w is said to be a **child** of node v . The **children** of node $v \in V$ are the set of nodes $W \subseteq V$ such that $(v, w) \in E$ for all $w \in W$. A node having no children is said to be a **leaf node**. Nonroot nodes $v, w \in V$ are said to be **sibling nodes** if $\text{parent}[v] = \text{parent}[w]$. The number of children of node $v \in V$ is denoted by $\text{children}[v]$.

Example 1.40. In tree T_2 of Fig. 1.17, node v_2 is the parent of node v_5 , and v_8, v_9 are sibling nodes, because they are both children of node v_7 . The root of T_2 is node v_1 , and the leaves are $v_3, v_4, v_5, v_8, v_9, v_{10}, v_{12}, v_{13}$. Nodes v_2, v_6, v_7, v_{11} are neither root nor leaf nodes.

The reader may prefer to consider the hierarchical structure of a tree the other way around, though. The leaves constitute a first level of height zero, their parents form a second level of height one, and so on. In other words, nodes can be partitioned into levels according to their height.

Definition 1.41. Let $T = (V, E)$ be a tree. The **height** of node $v \in V$, denoted by $\text{height}[v]$, is the length of a longest path from node v to any node in the subtree of T rooted at node v , for all nodes $v \in V$. The height of T is the maximum among the heights of all nodes $v \in V$.

It follows from Definition 1.38 that in a tree $T = (V, E)$, $\text{depth}[v] \geq 0$ for all nodes $v \in V$, and that $\text{depth}[v] = 0$ if and only if $v = \text{root}[T]$. Further, it follows from Definition 1.41 that $\text{height}[v] \geq 0$ for all nodes $v \in V$, and that $\text{height}[v] = 0$ if and only if v is a leaf node.

Example 1.42. In the tree shown twice in Fig. 1.18, node v_1 is the root and has depth zero and height three, which is also the height of the tree. Node v_2 has depth and height one, nodes v_3, v_4, v_5 have depth two and height zero, node v_6 has depth one and height two, nodes v_7, v_{11} have depth two and height one, and nodes $v_8, v_9, v_{10}, v_{12}, v_{13}$ have depth three, which is also the depth of the tree, and height zero, which is the height of all the leaves in the tree.

Trees are, indeed, the least connected graphs. The number of arcs in a tree is one less than the number of nodes in the tree.

Theorem 1.43. Let $T = (V, E)$ be a tree with n nodes and m arcs. Then, $m = n - 1$.

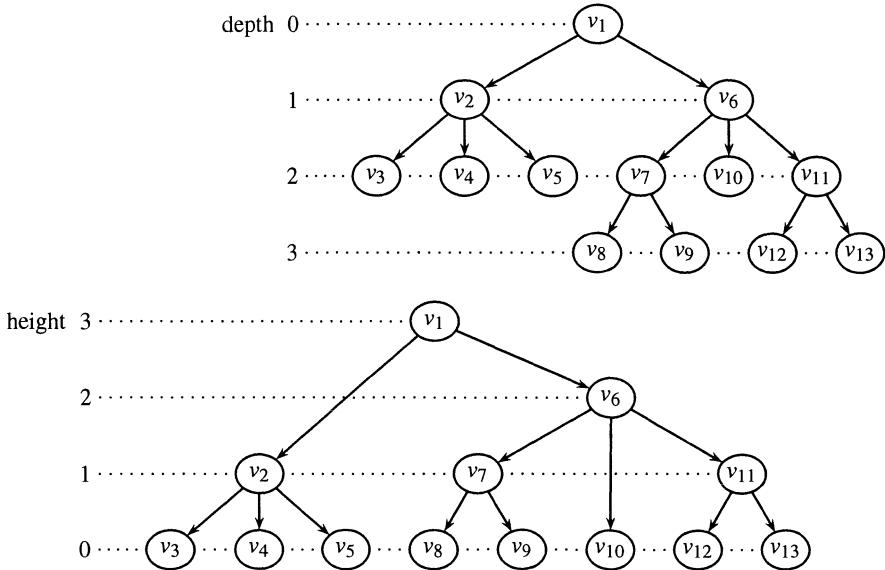


Fig. 1.18. The tree of Fig. 1.17 is shown twice, emphasizing the depth (top) and the height (bottom) of all nodes in the tree.

Proof. By induction on the number of nodes n in the tree. For $n = 1$, a tree with one node has no arcs, that is, $m = 0$. Assume, then, that every tree with $n \geq 1$ nodes has $n - 1$ arcs.

Let $T = (V, E)$ be a tree with $n + 1$ nodes, and let $v \in V$ be any leaf node of T . The subgraph of T induced by $V \setminus \{v\}$ is a tree with n nodes and has, by the induction hypothesis, $n - 1$ arcs. Now, since there is only arc $(w, v) \in E$ (namely, with w the parent in T of node v), it follows that the subgraph of T induced by $V \setminus \{v\}$ has one arc less than T . Therefore, T has $n + 1$ nodes and n arcs. \square

As with graphs, there is also a basic relationship between the number of nodes in a tree and the number of children of the nodes of the tree, which will prove to be very useful in analyzing the computational complexity of algorithms on trees.

Lemma 1.44. *Let $T = (V, E)$ be a tree on n nodes, and let $V = \{v_1, \dots, v_n\}$. Then,*

$$\sum_{i=1}^n \text{children}(v_i) = n - 1.$$

Proof. Let $T = (V, E)$ be a tree. Since every nonroot node $w \in V$ is the target of a different arc $(v, w) \in E$, it holds by Theorem 1.5 and Lemma 1.44 that $\sum_{i=1}^n \text{children}(v_i) = \sum_{i=1}^n \text{outdeg}(v_i) - 1 = m - 1$ and then, by Theorem 1.43, $\sum_{i=1}^n \text{children}(v_i) = n - 1$. \square

Now, given that trees are a particular case of graphs, it is natural to consider trees as subgraphs of a given graph and, as a matter of fact, those trees that span all the vertices of a graph arise in several graph algorithms. A *spanning tree* of a graph is a subgraph that contains all the vertices of the graph and is a tree.

Definition 1.45. Let $G = (V, E)$ be a graph. A subgraph (W, S) of G is a *spanning tree* of the graph G if $W = V$ and (W, S) is a tree.

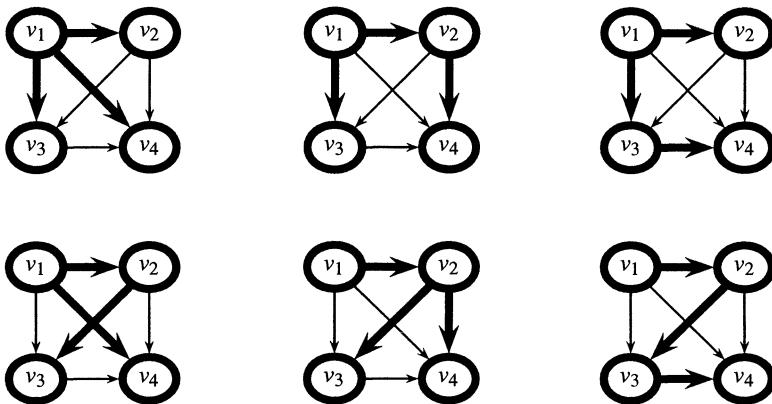


Fig. 1.19. Spanning trees of a graph.

Example 1.46. The graph of Fig. 1.19 has $n = 4$ vertices and $m = 6$ arcs, and thus $\binom{6}{3} = 20$ subgraphs with four vertices and three arcs. However, only 6 of these subgraphs are trees. These spanning trees of the graph are shown highlighted.

Not every graph has a spanning tree, though, but every graph has a *spanning forest*, that is, an ordered set of pairwise-disjoint subgraphs that are trees and which, together, span all the vertices of the graph.

Definition 1.47. Let $G = (V, E)$ be a graph. A sequence $[(W_1, S_1), \dots, (W_k, S_k)]$ of $k \geq 1$ subgraphs of G is a **spanning forest** of the graph G if $W_1 \cup \dots \cup W_k = V$; $W_i \cap W_j = \emptyset$ for all $1 \leq i, j \leq k$ with $i \neq j$; and (W_i, S_i) is a tree, for all $1 \leq i \leq k$.

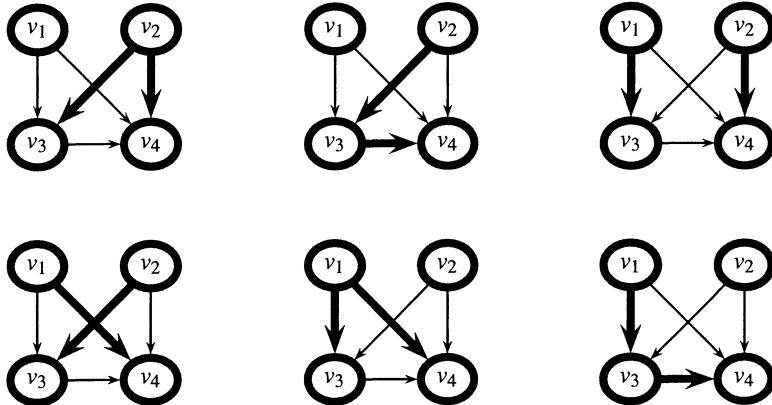


Fig. 1.20. Spanning forests of a graph.

Example 1.48. The graph of Fig. 1.20 has no spanning tree, but there are 12 forests that span all the vertices of the graph. These spanning forests of the graph are shown highlighted, up to a permutation of the sequence of trees in a forest. Consider, for instance, trees $T_1 = (\{v_1\}, \emptyset)$ and $T_2 = (\{v_2, v_3, v_4\}, \{(v_2, v_3), (v_2, v_4)\})$. Spanning forests $[T_1, T_2]$ and $[T_2, T_1]$ are both highlighted together, at the left of the top row.

Ordered Trees

An ordered tree is a tree in which the relative order of the children is fixed for each node. As a particular case of ordered graphs, ordered trees arise when a tree is drawn or *embedded* in the Euclidean plane.

Example 1.49. In the ordered tree shown in Fig. 1.21, the children of node v_1 are $[v_2, v_6]$; the children of node v_2 are $[v_3, v_4, v_5]$; the children of node v_6 are $[v_7, v_{10}, v_{11}]$; the children of node v_7 are $[v_8, v_9]$;

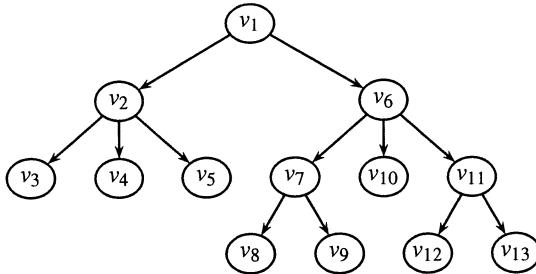


Fig. 1.21. An ordered tree. The relative order of the children of a given node reflects the counterclockwise ordering of the outgoing arcs of the node in the drawing.

and the children of node v_{11} are $[v_{12}, v_{13}]$; The remaining nodes have no children. The relative order of the children of each of the nodes reflects the counterclockwise ordering of the outgoing arcs of the node in the drawing. For instance, the ordered sequence $[v_7, v_{10}, v_{11}]$ of the children of node v_6 reflects the counterclockwise ordering $(v_6, v_7), (v_6, v_{10}), (v_6, v_{11})$ of the arcs going out of node v_6 in the drawing of the ordered tree.

The relative order of children nodes leads to further distinctions among the nodes of an ordered tree: children nodes are ordered from the *first* up to the *last*, nonfirst children nodes have a *previous sibling*, and nonlast children nodes have a *next sibling*. For sibling nodes v and w , let $v < w$ denote that v precedes w in the ordered tree.

Definition 1.50. Let $T = (V, E)$ be an ordered tree, and let $(v, w) \in E$. Node $w \in V$ is said to be the **first child** of node $v \in V$, denoted by $\text{first}[v]$, if there is no node $z \in V$ such that $(v, z) \in E$ and $z < w$. Node $w \in V$ is said to be the **last child** of node $v \in V$, denoted by $\text{last}[v]$, if there is no node $z \in V$ such that $(v, z) \in E$ and $w < z$. Node $z \in V$ is said to be the **next sibling** of node $w \in V$, denoted by $\text{next}[w]$, if $(v, z) \in E$, $w < z$, and there is no node $x \in V$ such that $(v, x) \in E$ and $w < x < z$. Node $w \in V$ is said to be the **previous sibling** of node $z \in V$, denoted by $\text{previous}[z]$, if $\text{next}[w] = z$.

Example 1.51. In the ordered tree shown in Fig. 1.21, $\text{first}[v_6] = v_7$, $\text{last}[v_6] = v_{11}$, and $\text{next}[v_7] = v_{10} = \text{previous}[v_{11}]$.

1.2 Literate Programming

Literate programming is a methodology of program development based on the idea of embedding an implementation into a document that describes the algorithm, instead of including documentation in the form of comments within the program code, and has the goal of producing programs that are easy to understand and maintain. All the algorithms presented in this book are implemented hand in hand with their documentation using literate programming tools and the whole book, typeset using L^AT_EX, together with running C++ programs, are produced from a single document.

The structure of a literate program is determined by the programmer, instead of the programming language compiler. Program code can be broken into small, understandable program modules, also called *chunks*, and documented in the order which is best suited for human comprehension.

A literate program is structured as a collection of sections, each of which has a documentation part and a code part. The documentation part describes the purpose of the section, often including a verbal description of an algorithm and any textual information that may help in understanding the problem or the algorithmic solution, and is intended to be processed by a typesetting system. The code part is a fragment of the code for the whole program.

Some of the main advantages of literate programming are summarized next.

- There is only one source document, from which both the documentation and the program code are extracted. The documented code is thus always consistent with the code that is executed.
- A program is thought of as a graph or web of structural relationships among simple parts, instead of a tree reflecting the hierarchical structure of the program. Both top-down and bottom-up programming are thus supported at the same time.
- Each part of a program can have the appropriate size, without distorting the readability of other parts of the program. Further, each part can be broken down into smaller parts if needed, in order to improve program readability.

- Literate programs take less time to debug, because the programmer is encouraged to impose herself a discipline, and they are also easier to maintain, because design decisions are documented during program development.
- Literate programming provides for cognitive reinforcement, because documentation is typeset in a pretty-printed format, where programming language constructs are displayed in an appropriate style.
- Documentation can make full use of the typographical capabilities that come with a typesetting system, including those for setting mathematical formulae and equations, tables, and diagrams.
- Reading aids such as a table of contents, an index, and cross-references among program modules can be generated, thus turning a literate program into a printed hypertext document, with pointers to related concepts and sections.

Perhaps the main disadvantage of literate programming lies in the need to master three different languages: the literate programming language, the typesetting language, and the programming language. However, the training necessary to learn how to program in a literate style is rather small, and it can be argued that the burden posed by literate programming pays off in practice.

Now, documentation capabilities also make literate programming well suited for the exposition of algorithms. All algorithms in this book are thus described in a literate programming style, using the noweb literate programming tool [264, 265]. Reading a literate program is much easier than writing programs in a literate style, but anyway, a brief primer on support tools will help to better understand what literate programming is all about.

A literate program is an ASCII document, called a *web*, containing code interleaved with documentation or, better, documentation interleaved with code. Literate programs are thus written in a combination of two languages: a typesetting language and a programming language, with a third, literate programming language binding documentation and code together. In particular, the literate programming language noweb uses either \TeX , \LaTeX , HTML, or troff as the typesetting language and can handle almost any programming language,

including: awk, C, C++, FORTRAN, Haskell, Java, Maple, Mathematica, Miranda, Modula-2, Pascal, perl, Prolog, Scheme, Standard ML, and *T_EX* itself. See the bibliographic notes below for references to further literate programming languages and their support tools.

Literate programs can be processed along two different lines. One line of processing is called *weaving* the web, and produces a document that describes the program. The other line of processing is called *tangling* the web, and produces an executable program.

Literate Programming using Noweb

The noweb literate programming language is a rather simple markup language, and the basic support tools for literate programming using noweb are the noweave program, used for weaving the web, and the notangle program, used for tangling the web. Their use in processing a literate program written in the C++ programming language is illustrated in Fig. 1.22.

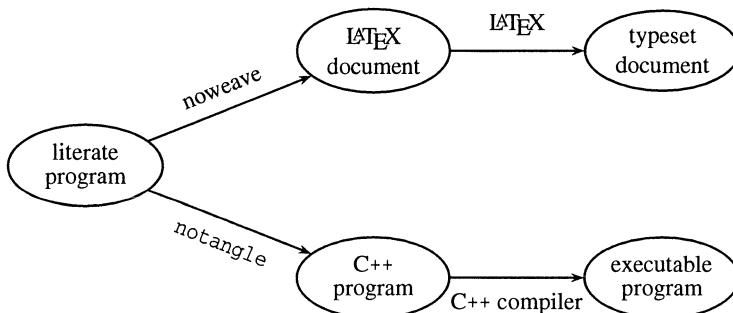


Fig. 1.22. Processing a C++ literate program using noweb. The woven document can be typeset using L^AT_EX, and the tangled program can be compiled using a C++ compiler.

A noweb file is a sequence of interleaved documentation and code sections, which may appear in any order. A documentation section begins with an at sign @_ in the first column of a line, followed by a space, or an at sign @ in the first column of a line by itself, and is terminated by either the beginning of another section, or the end of the file. A code section, on the other hand, begins with

<<section name>>=

in the first column of a line by itself, where the whole string enclosed in double angular brackets is the name of the section, and is also terminated by either the beginning of another section, or the end of the file.

Code sections contain program code and references to other code sections. Multiple code sections with the same name are cross-referenced by page number by the weaving process, and their definitions are concatenated into a single code section by the tangling process.

A code section definition is much like a macro definition without parameters. A tangled program is extracted from a literate program by expanding one particular code section, named `<<*>>` by default, whose definition contains references to other code sections, which are themselves expanded, and so on.

A documentation section, as well as the name of a section, may include a quoted code fragment enclosed in double square brackets, and special typographical treatment is given to these double square brackets by the weaving process.

Consider, for instance, the following excerpt from a literate program for printing a table of the first thousand prime numbers, adapted from [188], the first article on literate programming.

```
@ The remaining task is to fill table [[p]] with the
correct numbers. Let us do this by generating its entries
one at a time: Assuming that we have computed all primes
that are [[j]] or less, we will advance [[j]] to the next
suitable value, and continue doing this until the table
is completely full.
```

```
The program includes a provision to initialize
the variables in certain data structures that will be
introduced later.
```

```
<<fill table [[p]] with the first [[m]] prime numbers>>
<<initialize the data structures>>
while ( k < m ) {
    <<increase [[j]] until it is the next prime number>>
    p[++k] = j;
    <<double-check that [[p[k]]] is indeed prime>>
}
```

The corresponding woven document, typeset using \LaTeX , is shown in Fig. 1.23. Each reference to a code section in the typeset document includes a unique identifier, consisting of the page number in which the section was defined together with an alphabetic subpage reference.

The remaining task is to fill table p with the correct numbers. Let us do this by generating its entries one at a time: Assuming that we have computed all primes that are j or less, we will advance j to the next suitable value, and continue doing this until the table is completely full.

The program includes a provision to initialize the variables in certain data structures that will be introduced later.

27a ⟨fill table p with the first m prime numbers 27a⟩≡
 ⟨initialize the data structures 27c⟩
 while ($k < m$) {
 ⟨increase j until it is the next prime number 27b⟩
 $p[+k] = j;$
 ⟨double-check that $p[k]$ is indeed prime 27d⟩
 }
 }

Fig. 1.23. Typeset documentation for the excerpt from a literate program for printing a table of the first thousand prime numbers.

The typeset document also reflects a series of typographical conventions for C++ programs:

- C++ keywords are set in **boldface**, variables are set in *italics*, comments are set in roman, and strings are set in `typewriter` font.
- C++ arithmetic, relational, logical, bitwise, and member operators are set using appropriate mathematical symbols, most of them borrowed from the CWEB literate programming language [213].

-	-	+	+	*	*	/	÷	%	%	--	-	++	++
>	>	≥	≥	<	<	≤	≤	=	≡	!=	≠	&&	∧
	∨	!	¬	^	⊕	~	~	>>	»»	<<	««	->	→

- Double angular brackets in section names are set using corresponding mathematical symbols, and the equality sign in the definition of a code section is set using the equivalence symbol \equiv , for the first occurrence, and using the plus sign followed by the equivalence symbol $+≡$ for all subsequent occurrences of that code section in the literate program.

1.3 Implementation Correctness

The main goal of literate programming is producing programs that are easy to understand and maintain, and literate programming can be argued to help producing more robust and reliable programs. Exposing actual program code, though, also opens up the possibility to include result checking within the program code itself.

A program correctness checker is an algorithm that, given a program and a problem instance on which the program is executed, certifies whether the output of the program on that particular problem instance is correct.

Consider, for instance, implementation correctness of a sorting algorithm. The LEDA member function *sort()* of the *list* class sorts a list in ascending order by quick sort, using the default linear ordering on the elements of the list, and the member function *sort(cmp)* also sorts a list of elements of type *E* in ascending order by quick sort, but using instead the linear ordering defined by the compare function *cmp : E × E → int*. LEDA makes extensive use of result checking, especially for complex combinatorial and geometric algorithms, but does not include a checker of implementation correctness for sorting algorithms.

Let *L* be a list of elements, and let *L'* be the output of the LEDA sorting algorithm upon *L*. Checking implementation correctness of sorting involves verifying that the output list *L'* is indeed in sorted order, but also that *L'* is a permutation of the input list *L*.

In the following procedure, saving a copy *C* of the input list *L*, sorting *L*, and checking that *L* is a sorted permutation of *C* are all wrapped together. The whole procedure is parametrized by the type *E* of the elements of the list.

```
28 <implementation correctness of sorting 28>≡
  template<class E>
  void certified_sort(
    list<E>& L)
  {
    list<E> C = L; // save a copy C of list L
    L.sort();
    <double-check that L is a permutation of C 29b>
    <double-check that L is sorted 29a>
  }
```

The latter double check is straightforward. The following code verifies that list L is indeed sorted, that is, that $x \leq \text{succ}(x)$ for all but the last element x of L .

29a ⟨double-check that L is sorted 29a⟩≡
 $\{ \text{list_item } it;$
 $\forall \text{all_items}(it,L) \{$
 $\quad \text{if} (it \neq L.\text{last}() \wedge L[it] > L[L.\text{succ}(it)]) \{$
 $\quad \quad \text{error_handler}(1,\text{"Wrong implementation of sort"});$
 $\quad \}$ }

The former double check is a little more involved. A list is a permutation of another list if and only if both lists represent the same multiset. An alternative formulation is the following: a list L is a permutation of another list C if and only if the length of L and the length of C are the same and, for all elements x of L , the number of occurrences of x in L and in C are also the same.

The following code verifies that list L is indeed a permutation of list C , implementing the latter formulation. The number of occurrences of each element x in list L is stored in a dictionary array NL of integers indexed by the elements of L , and the number of occurrences of each element x in list C is stored in another dictionary array NC of integers indexed by the elements of C . Then, an exception is raised if either the length of L and the length of C differ, or there is an element x of L with a different number of occurrences in L and in C .

29b ⟨double-check that L is a permutation of C 29b⟩≡
 $\{ \text{if} (L.\text{length}() \neq C.\text{length}())$
 $\quad \text{error_handler}(1,\text{"Wrong implementation of sort"});$
 $\quad d.\text{array}\langle E,\text{int} \rangle NL(0); \quad // \text{initialize all entries to zero}$
 $\quad d.\text{array}\langle E,\text{int} \rangle NC(0); \quad // \text{idem}$
 $\quad E x;$
 $\quad \forall all(x,L) NL[x]++;
\quad // \text{increment by one the number of occurrences}$
 $\quad \forall all(x,C) NC[x]++;
\quad // \text{ibidem}$
 $\quad \forall all(x,L) \{$
 $\quad \quad \text{if} (NL[x] \neq NC[x]) \{ \quad // \text{different number of occurrences}$
 $\quad \quad \quad \text{error_handler}(1,\text{"Wrong implementation of sort"});$
 $\quad \}$ }

Remark 1.52. Exception handling is a standard mechanism to report error conditions. Depending on the particular application, though, some alternative approach may be more appropriate, and some corrective action may even be undertaken upon detecting an error.

Every decidable problem admits a correctness checker, although polynomial-time result checking algorithms for intractable problems are neither known nor believed to exist. Nevertheless, result checking of some intractable problems becomes tractable with the help of additional information in the form of correctness certificates, also called certification trails.

A correctness certificate is just additional information generated by a program when executed on a problem instance, and has the purpose of making result checking easier. A program correctness checker can now be redefined to be an algorithm that, given a program, a problem instance on which the program is executed, and a correctness certificate of the execution of the program on that problem instance, certifies, with the help of the correctness certificate, whether the output of the program on that particular problem instance is correct.

Correctness certificates are useful not only for result checking of algorithms for intractable problems, though. An example of a correctness certificate for a program implementing a sorting algorithm on a list of elements is a permutation that, when applied to the sorted list, produces the original list of elements.

Let L be a list of n elements. In the following procedure, the elements of L are tagged with an integer between 1 and n , which gives their position in L . The list of tagged elements is represented by a list T of ordered pairs $\langle x, i \rangle$, where x is the i th element in list L .

```
30 <implementation correctness of sorting 28>+≡
  template<class E>
  void faster_certified_sort(
    list<E>& L)
  {
    list<two_tuple<E,int>> T;
    E x;
    int i = 1;
    forall(x,L) {
      two_tuple<E,int> p(x,i++);
      T.append(p);
    }
    T.sort(cmp);
    <double-check that T is a permutation of L 31b>
    <double-check that T is sorted 32>

    L.clear();
    two_tuple<E,int> p;
    forall(p,T) {
```

```
L.append(p.first());
}
```

When sorting the list of tagged elements, the ordered pairs of element and integer position are compared according to their first component.

31a ⟨implementation correctness of sorting 28⟩ $\doteq\equiv$

```
template<class E>
int cmp(
    const two_tuple<E,int>& p,
    const two_tuple<E,int>& q)
{
    return compare(p.first(),q.first());
}
```

Now, the double check that the projection over the first component of the resulting list of tagged elements is indeed a permutation of the original list, can be performed with the help of the correctness certificate as follows.

Let n be the length of list L , and let A be an array of n elements, with $A[i]$ the i th element in L , for $1 \leq i \leq n$. Let also B be a Boolean array of n entries, with $B[i] = \text{true}$ if there is exactly one ordered pair of the form $\langle x, i \rangle$ in T , and $B[i] = \text{false}$ otherwise.

Then, assuming that T has n ordered pairs, the projection of T over the first component is a permutation of list L if and only if the following conditions are satisfied:

- $A[i] = x$, for all ordered pairs $\langle x, i \rangle$ in T .
- $B[i] = \text{true}$, for $1 \leq i \leq n$.

The following code verifies that the projection of T over the first component is a permutation of L , implementing a test of the previous conditions.

31b ⟨double-check that T is a permutation of L 31b⟩ \equiv

```
{ int n = L.length();
  if( T.length() != n )
    error_handler(1,"Wrong implementation of sort");

  array<E> A(1,n);
  int i = 1;
  E x;
  forall(x,L)
    A[i++] = x;
```

```

two_tuple<E,int> p;
forall(p,T) {
  if ( p.first() ≠ A[p.second()] ) {
    error_handler(1,"Wrong implementation of sort");
  }
}

array<bool> B(1,n);
B.init(false);
forall(p,T) {
  i = p.second();
  if ( B[i] )
    error_handler(1,"Wrong implementation of sort"); // repeated
  B[i] = true;
}
for ( int i = 1; i ≤ n; i++ ) {
  if ( ¬B[i] ) {
    error_handler(1,"Wrong implementation of sort"); // absent
  }
}

```

The double check that the resulting list of tagged elements is sorted in ascending order of first component is, again, straightforward. The following code verifies that list T is indeed sorted, that is, that $x \leq y$ for all consecutive ordered pairs $\langle x, i \rangle$ and $\langle y, j \rangle$ of T .

32 ⟨double-check that T is sorted 32⟩ ≡

```

{ list.item it;
forall_items(it,T) {
  if ( it ≠ T.last() ∧ T[it].first() > T[T.succ(it)].first() ) {
    error_handler(1,"Wrong implementation of sort");
  }
}

```

Remark 1.53. The former result checking algorithm for sorting runs in $O(n \log n)$ time using $O(n)$ additional space, where n is the length of the list to be sorted. As a matter of fact, the double check that the sorted list is indeed in sorted order takes $O(n)$ time, and the double check that the sorted list is a permutation of the original list makes $O(n)$ updates to a dictionary array, thus taking $O(n \log n)$ time.

The latter result checking algorithm for sorting, however, runs in $O(n)$ time using $O(n)$ additional space. It is often the case that correctness certificates allow for result checking algorithms which take asymptotically less time than the algorithm for solving the actual problem, and even than *any* algorithm for solving the problem. In the particular case of sorting, comparison-based sorting takes $\Omega(n \log n)$ time, and the problem of determining multiset equality can be shown to be equivalent to sorting.

Result checking does not actually establish implementation correctness, although it allows one to get some reassurance of correctness of the implementation. As long as result checking algorithms are themselves correct, result checking establishes implementation correctness for *every* problem instance upon which the program is executed. Therefore, program result checking is also an important software testing and debugging method.

For instance, the previous correctness checker for sorting can be tested in a systematic way by a simple procedure, on different problem instances. The following code implements a test of the correctness checker on sorting a random permutation of a list of n integers between 1 and n , for each n a power of 2 between 2^8 and 2^{16} .

```
33 <test implementation correctness of sorting 33>≡
  #include <LEDA/array.h>
  #include <LEDA/list.h>
  #include <LEDA/d_array.h>
  #include <LEDA/tuple.h>

  <implementation correctness of sorting 28>

  void main()
  {
    int min = 0x100; // pow(2,8)
    int max = 0x10000; // pow(2,16)
    list<int> L;
    for ( int n = min; n ≤ max; n = 2*n ) {
      cout ≪ "sorting " ≪ n ≪ " elements" ≪ endl;
      L.clear();
      for ( int i = 1; i ≤ n; i++ )
        L.append(i);
      L.permute();
      faster_certified_sort(L);
    }
    cout ≪ "passed" ≪ endl;
  }
```

All the algorithms presented in this book include a double-check of implementation correctness, and the interactive demonstration of graph algorithms also includes visual checkers of implementation correctness for all of the algorithms.

1.4 Representation of Trees and Graphs

There are several different ways in which graphs and, in particular, trees can be represented in a computer, and the choice of a data structure often depends on the efficiency with which the operations that access and update the data need to be supported. As a matter of fact, there is often a tradeoff between the space complexity of a data structure and the time complexity of the operations.

Representation of Graphs

A graph representation consists of a collection of abstract operations, a concrete representation of graphs by appropriate data structures, and an implementation of the abstract operations using the concrete data structures.

The choice of abstract operations to be included in a graph representation depends on the particular application area. For instance, the LEDA representation of graphs supports about 120 different operations, roughly half of which address the needs of computational geometry algorithms. The following collection of 28 operations, though, covers the needs of most of the algorithms on graphs presented in this book. See Appendix A for more details.

<i>number_of_nodes()</i>	<i>number_of.edges()</i>	<i>opposite(v,e)</i>	<i>adjacent(v,w)</i>
<i>indeg(v)</i>	<i>outdeg(v)</i>	<i>source(e)</i>	<i>target(e)</i>
<i>first_node()</i>	<i>last_node()</i>	<i>pred_node(v)</i>	<i>succ_node(v)</i>
<i>first_edge()</i>	<i>last_edge()</i>	<i>pred_edge(e)</i>	<i>succ_edge(e)</i>
<i>first_in_edge(v)</i>	<i>last_in_edge(v)</i>	<i>in_pred(e)</i>	<i>in_succ(e)</i>
<i>first_adj_edge(v)</i>	<i>last_adj_edge(v)</i>	<i>adj_pred(e)</i>	<i>adj_succ(e)</i>
<i>new_node()</i>	<i>new_edge(v,w)</i>	<i>del_node(v)</i>	<i>del_edge(e)</i>

- $G.\text{number_of_nodes}()$ and $G.\text{number_of_edges}()$ give respectively the order n and the size m of graph G .
- $G.\text{indeg}(v)$ and $G.\text{outdeg}(v)$ give respectively the number of arcs coming into and going out of vertex v in G .
- $G.\text{adjacent}(v,w)$ is true if there is an arc in G going out of vertex v and coming into vertex w , and false otherwise.
- $G.\text{source}(e)$ and $G.\text{target}(e)$ give respectively the source and the target vertex of edge e in G . Also, $G.\text{opposite}(v,e)$ gives $G.\text{target}(e)$ if vertex v is the source of arc e in G , and $G.\text{source}(e)$ otherwise.

- $G.\text{first_node}()$ and $G.\text{last_node}()$ give respectively the first and the last vertex in the representation of G . Further, $G.\text{pred_node}(v)$ and $G.\text{succ_node}(v)$ give respectively the predecessor and the successor of vertex v in the representation of G . These operations support iteration over the vertices of the graph.
- $G.\text{first_edge}()$ and $G.\text{last_edge}()$ give respectively the first and the last arc in the representation of G . Further, $G.\text{pred_edge}(e)$ and $G.\text{succ_edge}(e)$ give respectively the predecessor and the successor of arc e in the representation of G . These operations support iteration over the arcs of the graph.
- $G.\text{first_in_edge}(v)$ and $G.\text{last_in_edge}(v)$ give respectively the first and the last arc in the representation of G coming into vertex v . Further, $G.\text{in_pred}(e)$ and $G.\text{in_succ}(e)$ give respectively the previous and the next arc after e in the representation of G coming into vertex $G.\text{target}(e)$. These operations support iteration over the vertices of the graph adjacent to a given vertex.
- $G.\text{first_adj_edge}(v)$ and $G.\text{last_adj_edge}(v)$ give respectively the first and the last arc in the representation of G going out of vertex v . Further, $G.\text{adj_pred}(e)$ and $G.\text{adj_succ}(e)$ give respectively the previous and the next arc after e in the representation of G going out of vertex $G.\text{source}(e)$. These operations also support iteration over the vertices of the graph adjacent to a given vertex.
- $G.\text{new_node}()$ inserts a new vertex in G , and $G.\text{new_edge}(v,w)$ inserts a new arc in G going out of vertex v and coming into vertex w . Further, $G.\text{del_node}(v)$ deletes vertex v from G , together with all those arcs going out of or coming into vertex v , and $G.\text{del_edge}(e)$ deletes arc e from G . These operations support dynamic graphs.

The previous operations support iteration over the vertices and arcs of a graph. For instance, in the following procedure, variable v is assigned each of the vertices of graph G in turn,

35a \langle iteration over all vertices v of graph G 35a $\rangle \equiv$
 $\quad \{ v = G.\text{first_node};$
 $\quad \quad \text{while } (v \neq \text{nil}) \{$
 $\quad \quad \quad // \text{process vertex } v$
 $\quad \quad \quad v = G.\text{succ_node}(v);$
 $\quad \quad \}$

and in the following procedure, all those vertices of graph G which are adjacent to vertex v are assigned in turn to variable w .

35b ⟨iteration over all vertices w adjacent in G to vertex v 35b⟩ \equiv

```

{ edge e = G.first_adj_edge(v);
  while ( e ≠ nil ) {
    w = G.opposite(v,e);
    // process vertex w
    e = G.adj_succ(e);
  }
}
```

Notice that LEDA provides a series of *macros* as a more convenient mechanism for iteration over the vertices and arcs of a graph. Some of these macros are listed next where, for instance, *forall_nodes*(v, G) corresponds to the iteration over all vertices v of the graph, and *forall_adj_nodes*(w, v) implements the iteration over all vertices w adjacent in the graph to vertex v . See Appendix A for more details.

<i>forall_nodes</i> (v, G)	<i>forall_rev_nodes</i> (v, G)
<i>forall_edges</i> (e, G)	<i>forall_rev_edges</i> (e, G)
<i>forall_in_edges</i> (e, v)	<i>forall_adj_edges</i> (e, v)
<i>forall_adj_nodes</i> (w, v)	

The data structures most often used for representing graphs are adjacency matrices and adjacency lists. The adjacency matrix representation of a graph is a Boolean matrix, with an entry for each ordered pair of vertices in the graph, where the entry corresponding to vertices v and w has the value *true* if there is an arc in the graph going out of vertex v and coming into vertex w , and it has the value *false* otherwise.

Definition 1.54. Let $G = (V, E)$ be a graph with n vertices. The **adjacency matrix representation** of G is an $n \times n$ Boolean matrix with an entry for each ordered pair of vertices of G , where the entry corresponding to vertices v and w is equal to *true* if $(v, w) \in E$ and is equal to *false* otherwise, for all vertices $v, w \in V$.

Example 1.55. The adjacency matrix representation of the graph of Fig. 1.1 is shown in Fig. 1.24. Entries equal to *true* are denoted by 1, and *false* entries are denoted by 0.

Notice that with most data structures used for representing graphs, there is an order on the vertices fixed by the representation. In the case of the adjacency matrix representation A of a graph $G = (V, E)$, vertex v precedes vertex w if and only if the row of A corresponding

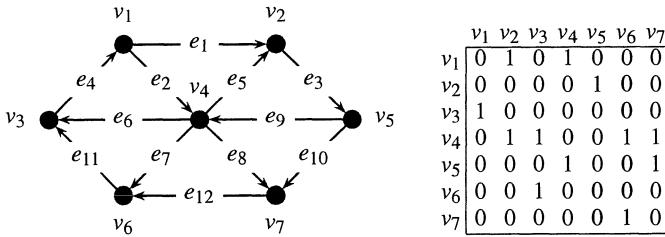


Fig. 1.24. Adjacency matrix representation of the graph of Fig. 1.1.

to v lies above the row of A corresponding to w or, in an equivalent formulation, the column of A corresponding to v lies to the left of the column of A corresponding to w , for all vertices $v, w \in V$.

Now, since the arcs are implicit in the adjacency matrix representation of a graph, those operations having an arc as argument or giving an arc as result cannot be implemented using an adjacency matrix representation. These operations are: $G.\text{adj_pred}(e)$, $G.\text{adj_succ}(e)$, $G.\text{del_edge}(e)$, $G.\text{first_adj_edge}(v)$, $G.\text{first_edge}()$, $G.\text{first_in_edge}(v)$, $G.\text{in_pred}(e)$, $G.\text{in_succ}(e)$, $G.\text{last_adj_edge}(v)$, $G.\text{last_edge}()$, $G.\text{last_in_edge}(v)$, $G.\text{opposite}(v,e)$, $G.\text{pred_edge}(e)$, $G.\text{source}(e)$, $G.\text{target}(e)$, and $G.\text{succ_edge}(e)$.

Furthermore, $G.\text{new_node}()$ and $G.\text{del_node}(v)$ cannot be implemented unless the adjacency matrix can be dynamically resized, and $G.\text{new_edge}(v,w)$ can be implemented by setting to *true* the entry of A at the row corresponding to vertex v and the column corresponding to vertex w , and takes $O(1)$ time, but the operation cannot give the new arc as a result.

Let $G = (V, E)$ be a graph with n vertices, and let A be the adjacency matrix representation of G . The remaining operations can be implemented as follows:

- $G.\text{number_of_nodes}()$ is equal to the number of rows or columns in A , and takes $O(1)$ time.
- $G.\text{number_of_edges}()$ is implemented by counting the number of *true* entries in A , and takes $O(n^2)$ time.
- $G.\text{adjacent}(v,w)$ is equal to the value of A at the row corresponding to vertex v and the column corresponding to vertex w , and takes $O(1)$ time.

- $G.\text{indeg}(v)$ is implemented by counting the number of *true* entries in the column of A corresponding to vertex v , and takes $O(\text{indeg}(v))$ time.
- $G.\text{outdeg}(v)$ is implemented by counting the number of *true* entries in the row of A corresponding to vertex v , and takes $O(\text{outdeg}(v))$ time.
- $G.\text{first_node}()$ is equal to the first vertex of A in the order fixed by the representation, and takes $O(1)$ time.
- $G.\text{last_node}()$ is equal to the last vertex of A in the order fixed by the representation, and takes $O(1)$ time.
- $G.\text{pred_node}(v)$ is equal to the previous vertex of A before vertex v in the order fixed by the representation, and takes $O(1)$ time.
- $G.\text{succ_node}(v)$ is equal to the next vertex of A after vertex v in the order fixed by the representation, and takes $O(1)$ time.

The adjacency matrix representation of a graph $G = (V, E)$ with n vertices takes $\Theta(n^2)$ space. Therefore, no graph algorithm can be implemented using an adjacency matrix representation to run in $O(n)$ time. The main advantage of the adjacency matrix representation is the support of adjacency test in $O(1)$ time.

The adjacency list representation of a graph, on the other hand, is a vector of linked lists, one for each vertex in the graph, where the list corresponding to vertex v contains the target vertices of the arcs coming out of vertex v .

Definition 1.56. Let $G = (V, E)$ be a graph with n vertices. The **adjacency list representation** of G is an array of n linked lists. The list corresponding to vertex v contains all vertices $w \in V$ with $(v, w) \in E$, for all vertices $v \in V$.

Notice that adjacency lists are not necessarily arranged in any particular order although there is, as a matter of fact, an order on the vertices and arcs fixed by the adjacency list representation of the graph. Given the adjacency list representation of a graph $G = (V, E)$, vertex precedence is given by the order of the corresponding entries in the array of linked lists, and arc (v, w) precedes arc (v, z) if and only if vertex w precedes vertex z in the linked list corresponding to vertex v , for all arcs $(v, w), (v, z) \in E$.

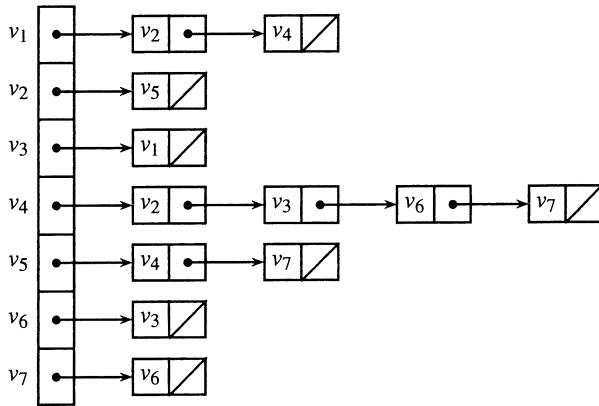


Fig. 1.25. Adjacency list representation of the graph of Fig. 1.1.

Example 1.57. The adjacency list representation of the graph from Fig. 1.1 is shown in Fig. 1.25.

As in the case of adjacency matrices, arcs are implicit in the adjacency list representation of a graph, and those operations having an arc as argument or giving an arc as result cannot be implemented using an adjacency list representation. These operations are: *G.opposite(v,e)*, *G.pred_edge(e)*, *G.succ_edge(e)*, *G.in_pred(e)*, *G.in_succ(e)*, *G.adj_pred(e)*, *G.adj_succ(e)*, *G.source(e)*, *G.target(e)*, and *G.del_edge(e)*.

Furthermore, *G.new_node()* and *G.del_node(v)* cannot be implemented unless the array of linked lists can be dynamically resized, and *G.new_edge(v,w)* can be implemented by appending vertex *w* to the linked list corresponding to vertex *v*, and takes $O(1)$ time, but the operation cannot give the new arc as result.

Let $G = (V, E)$ be a graph with n vertices and m arcs. Using the adjacency list representation, operations *G.indeg(v)*, *G.first_in_edge(v)*, and *G.last_in_edge(v)* require scanning the whole adjacency list representation and thus take $O(n + m)$ time. The remaining operations can be implemented as follows:

- *G.number_of_nodes()* is equal to the number of entries in the array of linked lists, and takes $O(1)$ time.

- $G.\text{number_of_edges}()$ is implemented by summing up the length of each linked list, and takes $O(n)$ time.
- $G.\text{adjacent}(v,w)$ is implemented by scanning the linked list corresponding to vertex v , and takes $O(\text{outdeg}(v))$ time.
- $G.\text{outdeg}(v)$ is equal to the length of the linked list corresponding to vertex v , and takes $O(1)$ time.
- $G.\text{first_node}()$ and $G.\text{last_node}()$ are respectively the first and the last entry in the array of linked lists, and take $O(1)$ time.
- $G.\text{pred_node}(v)$ and $G.\text{succ_node}(v)$ are the entries in the array of linked lists respectively right before and right after the entry corresponding to vertex v , and take $O(1)$ time.
- $G.\text{first_edge}()$ is the first vertex in the first linked list, and takes $O(1)$ time.
- $G.\text{last_edge}()$ is the last vertex in the last linked list, and takes time linear in the maximum outdegree of the vertices, or $O(1)$ time if the lists are doubly linked.
- $G.\text{first_adj_edge}(v)$ is the first vertex in the linked list corresponding to vertex v , and takes $O(1)$ time.
- $G.\text{last_adj_edge}(v)$ is the last vertex in the linked list corresponding to vertex v , and takes $O(\text{outdeg}(v))$ time, or $O(1)$ time if the lists are doubly linked.

The adjacency list representation of a graph $G = (V, E)$ with n vertices and m arcs takes $\Theta(n + m)$ space. Beside the low space requirement, the main advantage of the adjacency list representation is the support of iteration operations in $O(1)$ time. Their main disadvantage, though, lies in the fact that an adjacency test $G.\text{adjacent}(v,w)$ is not supported in $O(1)$ but in $O(\text{outdeg}(v))$ time.

Notice that the adjacency matrix representation of a graph can be easily obtained from the LEDA adjacency list representation. The following procedure makes A the adjacency matrix of graph G , and runs in $O(n^2 + m) = O(n^2)$ time.

40 ⟨subroutines 40⟩≡
void adjacency_matrix(
 const graph& G ,
 node_matrix<bool>& A)
{
 $A.\text{init}(G,\text{false});$
 edge e ;

```

forall_edges(e,G) {
    A(G.source(e),G.target(e)) = true;
}

```

Then, implementing the adjacency test using the adjacency matrix representation of a graph is straightforward. The following procedure *adjacent*(v, w, A) returns *true* if $(v, w) \in E$ and false otherwise and, given the adjacency matrix representation A of the graph, runs in $O(\text{outdeg}(v))$ time.

41a $\langle \text{subroutines 40} \rangle + \equiv$

```

bool adjacent(
    node v,
    node w,
    const node_matrix<bool>& A)
{
    return A(v,w);
}

```

An alternative implementation of the adjacency test consists of scanning adjacent vertices in the LEDA adjacency list representation. Let $G = (V, E)$ be a graph, and let $v, w \in V$. The following procedure *adjacent*(v, w) returns *true* if $(v, w) \in E$, and false otherwise, and runs in $O(\text{outdeg}(v))$ time.

41b

$\langle \text{subroutines 40} \rangle + \equiv$

```

bool adjacent(
    node v,
    node w)
{
    node u;
    forall_adj_nodes(u,v)
        if ( u == w ) return true;
    return false;
}

```

The representation of graphs adopted in LEDA is based on the adjacency list representation, although it has been extended in order to support dynamic graphs (in which vertices and arcs can be inserted, deleted, and rearranged at any time) and also to support computational geometry algorithms.

Essentially, the LEDA representation of a graph consists of a doubly linked list of vertices and a doubly linked list of arcs. Associated with each vertex are two doubly linked lists of incoming and outgoing

arcs. Associated with each arc are its source and target vertices, and two doubly linked lists of arcs incident with the same source vertex and with the same target vertex, respectively.

vertex	incoming arcs		outgoing arcs	
arc	source	target	same source	same target
$v_1,$		$[e_4]$		$[e_1, e_2]$
$v_2,$		$[e_1, e_5]$		$[e_3]$
$v_3,$		$[e_6, e_{11}]$		$[e_4]$
$v_4,$		$[e_2, e_9]$		$[e_5, e_6, e_7, e_8]$
$v_5,$		$[e_3]$		$[e_9, e_{10}]$
$v_6,$		$[e_7, e_{12}]$		$[e_{11}]$
$v_7]$		$[e_8, e_{10}]$		$[e_{12}]$
$e_1,$	v_1	v_2	$[e_2]$	$[e_5]$
$e_2,$	v_1	v_4	$[e_1]$	$[e_9]$
$e_3,$	v_2	v_5	$[]$	$[]$
$e_4,$	v_3	v_1	$[]$	$[]$
$e_5,$	v_4	v_2	$[e_6, e_7, e_8]$	$[e_1]$
$e_6,$	v_4	v_3	$[e_7, e_8, e_5]$	$[e_{11}]$
$e_7,$	v_4	v_6	$[e_8, e_5, e_6]$	$[e_{12}]$
$e_8,$	v_4	v_7	$[e_5, e_6, e_7]$	$[e_{10}]$
$e_9,$	v_5	v_4	$[e_{10}]$	$[e_2]$
$e_{10},$	v_5	v_7	$[e_9]$	$[e_8]$
$e_{11},$	v_6	v_3	$[]$	$[e_6]$
$e_{12}]$	v_7	v_6	$[]$	$[e_7]$

Fig. 1.26. Simplified presentation of the LEDA adjacency list representation for the graph of Fig. 1.1. Doubly linked lists are shown as sequences.

Example 1.58. The LEDA adjacency list representation of the graph of Fig. 1.1 is shown in a simplified form in Fig. 1.26, where doubly linked lists of vertices and arcs are shown as sequences.

Notice that arcs are explicit in the LEDA graph representation. Therefore, all of the previous operations can be implemented using the extended adjacency list representation and take $O(1)$ time, with the exception of $G.del_node(v)$, which takes $O(\deg(v))$ time, and $G.adjacent(v,w)$, which is not provided by LEDA. The operations are implemented as follows:

- $G.number_of_nodes()$ and $G.number_of_edges()$ are respectively the length of the doubly linked list of vertices and the length of the doubly linked list of arcs.

- $G.opposite(v,e)$ is $G.target(e)$ if vertex v is equal to $G.source(e)$, and is $G.source(e)$ otherwise.
- $G.indeg(v)$ and $G.outdeg(v)$ are respectively the length of the list of arcs coming into and the length of the list of arcs going out of vertex v .
- $G.source(e)$ and $G.target(e)$ are respectively the source and the target vertex associated with arc e .
- $G.first_node()$ and $G.last_node()$ are respectively the first and the last vertex in the doubly linked list of vertices.
- $G.pred_node(v)$ and $G.succ_node(v)$ are respectively the predecessor and the successor of vertex v in the doubly linked list of vertices.
- $G.first_edge()$ and $G.last_edge()$ are respectively the first and the last arcs in the doubly linked list of arcs.
- $G.pred_edge(e)$ and $G.succ_edge(e)$ are respectively the predecessor and the successor of arc e in the doubly linked list of arcs.
- $G.first_in_edge(v)$ and $G.last_in_edge(v)$ are respectively the first and the last arc in the doubly linked list of arcs incident with the same target vertex v .
- $G.in_pred(e)$ and $G.in_succ(e)$ are respectively the predecessor and the successor of arc e in the doubly linked list of arcs incident with the same target vertex $G.target(e)$.
- $G.first_adj_edge(v)$ and $G.last_adj_edge(v)$ are respectively the first and the last arc in the doubly linked list of arcs incident with the same source vertex v .
- $G.adj_pred(e)$ and $G.adj_succ(e)$ are respectively the predecessor and the successor of arc e in the doubly linked list of arcs incident with the same source vertex $G.source(e)$.
- $G.new_node()$ is implemented by appending a new vertex v to the doubly linked list of vertices, and returning vertex v .
- $G.new_edge(v,w)$ is implemented by appending a new arc e to the doubly linked list of arcs, setting to v the source vertex associated with arc e , setting to w the target vertex associated with arc e , setting the doubly linked list of arcs incident with the same source vertex $G.source(e)$ to the doubly linked list of arcs going out of vertex v , setting the doubly linked list of arcs incident with the same target vertex $G.target(e)$ to the doubly linked list of arcs coming into ver-

- tex w , and then appending e to the doubly linked lists of arcs coming into vertex w and going out of vertex v , and returning arc e .
- $G.del_node(v)$ is implemented by performing $G.del_edge(e)$ for each arc e in the doubly linked lists of arcs coming into and going out of vertex v , and then deleting vertex v from the doubly linked list of vertices.
 - $G.del_edge(e)$ is implemented by deleting arc e from the doubly linked list of arcs, and also from the doubly linked list of arcs coming into vertex $G.target(e)$, from the doubly linked list of arcs going out of vertex $G.source(e)$, from the doubly linked list of arcs incident with the same source vertex $G.source(e')$ of each arc e' in the doubly linked list of arcs incident with the same source vertex $G.source(e)$, and from the doubly linked list of arcs incident with the same target vertex $G.target(e')$ of each arc e' in the doubly linked list of arcs incident with the same target vertex $G.target(e)$.

Notice that these operations can be implemented in $O(1)$ time because most of the doubly linked lists are shared and are associated with vertices and arcs by means of pointers. For instance, for the LEDA graph representation shown in Fig. 1.26, all of the arcs e_5, e_6, e_7, e_8 go out of vertex v_4 , and their doubly linked lists of arcs incident with the same source vertex are stored only once, as $[e_5, e_6, e_7, e_8]$, and associated with each of these arcs are pointers to the predecessor and the successor arc in the doubly linked list.

The space complexity of the LEDA graph representation is $O(n + m)$. More precisely, the space requirement of a LEDA graph is $1788 + 44m + 52n$ bytes.

Representation of Trees

As in the case of graphs, a tree representation consists of a collection of abstract operations, a concrete representation of trees by appropriate data structures, and an implementation of the abstract operations using the concrete data structures.

The following collection of 12 operations covers the needs of most of the algorithms on trees presented in this book. Again, see Appendix A for more details.

<i>T.number_of_nodes()</i>	<i>T.root()</i>	<i>T.number_of_children(v)</i>
<i>T.parent(v)</i>	<i>T.is_root(v)</i>	<i>T.is_leaf(v)</i>
<i>T.first_child(v)</i>	<i>T.next_sibling(v)</i>	<i>T.is_last_child(v)</i>
<i>T.last_child(v)</i>	<i>T.previous_sibling(v)</i>	<i>T.is_first_child(v)</i>

- $T.\text{number_of_nodes}()$ gives the number of nodes of tree $T = (V, E)$.
- $T.\text{root}()$ gives $\text{root}[T]$. Further, $T.\text{is_root}(v)$ is true if node $v = \text{root}[T]$, and false otherwise.
- $T.\text{number_of_children}(v)$ gives $\text{children}[v]$, that is, the number of nodes $w \in V$ such that $(v, w) \in E$.
- $T.\text{parent}(v)$ gives $\text{parent}[v]$, that is, the parent in T of node v .
- $T.\text{is_leaf}(v)$ is true if node v is a leaf of T , and false otherwise.
- $T.\text{first_child}(v)$ and $T.\text{last_child}(v)$ give respectively the first and the last of the children of node v , according to the order of the children of v fixed by the representation of T .
- $T.\text{previous_ sibling}(v)$ and $T.\text{next_ sibling}(v)$ give respectively the previous child of $T.\text{parent}(v)$ before vertex v and the next child of $T.\text{parent}(v)$ after vertex v , according to the order of the children of $T.\text{parent}(v)$ fixed by the representation of T .
- $T.\text{is_first_child}(v)$ and $T.\text{is_last_child}(v)$ are true if node v is respectively the first and the last of the children of node $T.\text{parent}(v)$, according to the order of the children of $T.\text{parent}(v)$ fixed by the representation of T , and false otherwise.

The previous operations support iteration over the children of a node in a tree, much in the same way that the graph operations support iteration over the vertices and arcs of a graph. In the following procedure, the children of node v in tree T are assigned in turn to variable w .

45a \langle iteration over all children w of node v in T 45a $\rangle \equiv$
 $\{ w = T.\text{first_child}(v);$
 $\quad \text{while } (w \neq \text{nil}) \{$
 $\quad \quad // \text{process node } w$
 $\quad \quad w = T.\text{next_ sibling}(w);$
 $\quad \}$

The following procedure is an alternative form of iteration over the children w of a node v in a tree T .

45b \langle alternative iteration over all children w of node v in T 45b $\rangle \equiv$
 $\{ \text{if } (\neg T.\text{is_leaf}(v)) \{$

```
w = T.first_child(v);
// process node w
while ( w ≠ T.last_child(v) ) {
    w = T.next_sibling(w);
    // process node w
} } }
```

The macro *forall_children(w,v)*, which is based on the LEDA macros for iteration over the vertices and arcs of a graph, implements the iteration over all children w of node v in the tree. See Appendix A for more details.

The data structures most often used for representing trees are the array-of-parents representation and the first-child, next-sibling representation. The array-of-parents representation of a tree is an array of nodes, with an entry for each node in the tree, where the entry corresponding to node v has the value $\text{parent}[v]$ if v is not the root of the tree, and it has the value *nil* otherwise.

Definition 1.59. Let $T = (V, E)$ be a tree with n nodes. The **array-of-parents** representation of T is an array P of n nodes, indexed by the nodes of the tree, where $P[v] = \text{parent}[v]$ if $v \neq \text{root}[T]$, and $P[v] = \text{nil}$ otherwise, for all nodes $v \in V$.

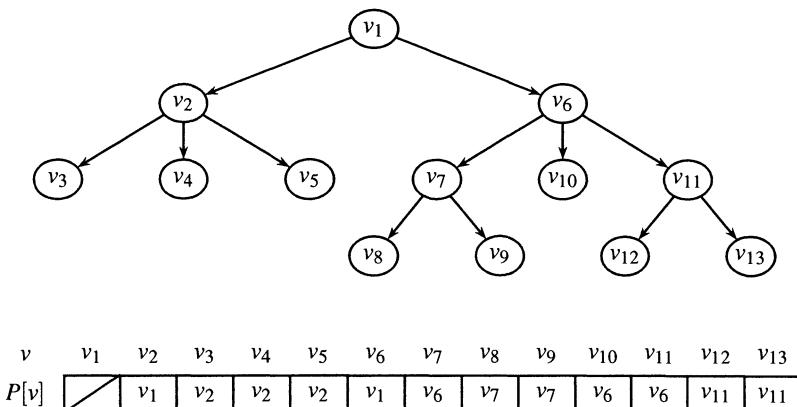


Fig. 1.27. Array-of-parents representation of a tree.

Example 1.60. The array-of-parents representation P of the tree $T = (V, E)$ of Fig. 1.21 is shown in Fig. 1.27. The parent of node v is given by $P[v]$, for all nodes $v \in V$.

The array of parent nodes is not necessarily arranged in any particular order although there is, as a matter of fact, an order on the nodes fixed by the representation. Given the array-of-parents representation of a tree, node precedence is given by the order of the nodes as array indices, and precedence between sibling nodes is also given by the order of array indices.

Let $T = (V, E)$ be a tree with n nodes, and let P be the array-of-parents representation of T . Operations $T.\text{first_child}(v)$, $T.\text{last_child}(v)$, $T.\text{is_first_child}(v)$, $T.\text{is_last_child}(v)$, $T.\text{previous_sibling}(v)$, $T.\text{is_leaf}(v)$, $T.\text{next_sibling}(v)$, $T.\text{root}()$, and $T.\text{number_of_children}(v)$ require scanning the whole array and thus take $O(n)$ time. The remaining operations can be implemented to take $O(1)$ time as follows:

- $T.\text{number_of_nodes}()$ is the size of array P .
- $T.\text{parent}(v)$ is equal to $P[v]$.
- $T.\text{is_root}(v)$ is true if $P[v] = \text{nil}$, and false otherwise.

The first-child, next-sibling representation, on the other hand, consists of two arrays of nodes, each of them with an entry for each node in the tree, where the entries corresponding to node v have respectively the value $\text{first}[v]$, or nil if v is a leaf node, and $\text{next}[v]$, or nil if v is a last sibling.

Definition 1.61. Let $T = (V, E)$ be a tree with n nodes. The **first-child next-sibling representation** of T is a pair (F, N) of arrays of n nodes, indexed by the nodes of the tree, where $F[v] = \text{first}[v]$ if v is not a leaf node, $F[v] = \text{nil}$ otherwise, $N[v] = \text{next}[v]$ if v is not a last child node, and $N[v] = \text{nil}$ otherwise, for all nodes $v \in V$.

Example 1.62. The array-of-parents representation of the tree $T = (V, E)$ of Fig. 1.21 is shown in Fig. 1.28. The first child and the next sibling of node v are given by $F[v]$ and $N[v]$, respectively, for all nodes $v \in V$.

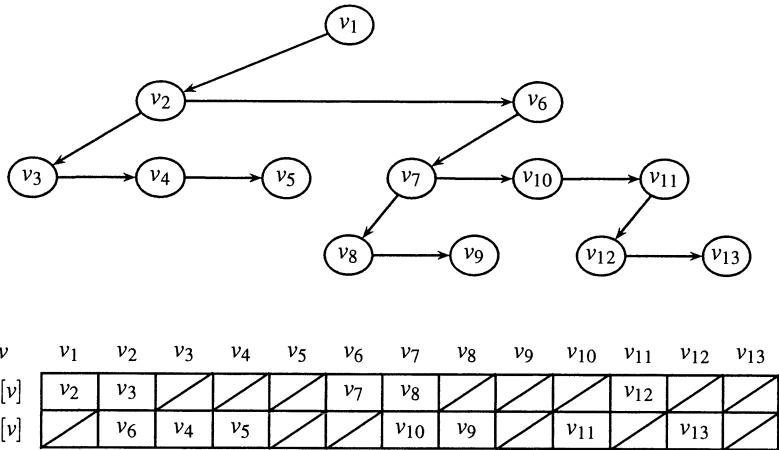


Fig. 1.28. First-child, next-sibling representation of a tree. Vertical arrows denote first-child links, and horizontal arrows denote next-sibling links.

As with the array-of-parents representation, the arrays of first children and next siblings are not necessarily arranged in any particular order, as long as they are arranged in the same order. Anyway, there is an order on the nodes fixed by the representation. Given the first-child, next-sibling representation of a tree, node precedence is given by the order of the nodes as array indices, and precedence between sibling nodes is also given by the order of array indices.

Let $T = (V, E)$ be a tree with n nodes, and let (F, N) be the first-child, next-sibling representation of T . Operations $T.parent(v)$, $T.root()$, and $T.is_root(v)$ require scanning both of arrays F and N , and $T.previous_sibling(v)$ and $T.is_first_child(v)$ require scanning array N , and thus take $O(n)$ time. Further, $T.last_child(v)$ and $T.number_of_children(v)$ require following up to $children[v]$ next-sibling links, where the latter also requires following a first-child link, and therefore take $O(children[v])$ time. The remaining operations can be implemented to take $O(1)$ time as follows:

- $T.number_of_nodes()$ is the size of array F , or the size of array N .
- $T.first_child(v)$ is given by $F[v]$. Further, $T.is_leaf(v)$ is true if $F[v] = nil$, and false otherwise.
- $T.next_sibling(v)$ is given by $N[v]$. Further, $T.is_last_child(v)$ is true if $N[v] = nil$, and false otherwise.

The representation of trees adopted in this book is based instead on the LEDA graph representation. Operations $T.\text{number_of_nodes}()$, $T.\text{parent}(v)$, $T.\text{is_root}(v)$, $T.\text{first_child}(v)$, $T.\text{last_child}(v)$, $T.\text{previous_ sibling}(v)$, $T.\text{next_sibling}(v)$, $T.\text{is_first_child}(v)$, $T.\text{is_last_child}(v)$, $T.\text{is_leaf}(v)$, and $T.\text{number_of_children}(v)$ take $O(1)$ time, and $T.\text{root}()$ takes time linear in the depth or height of the tree. The representation uses $O(n)$ space. See Appendix A for more details.

Summary

Basic graph-theoretical notions underlying algorithms on trees and graphs, together with appropriate data structures for their representation, are reviewed in this chapter. References to more compact, implicit representations are given in the bibliographic notes below. The literate programming style adopted in this book for the presentation of algorithms and data structures, as well as the implementation correctness approach by result checking using correctness certificates, are also discussed in some detail.

Bibliographic Notes

A more detailed exposition of the graph-theoretical notions reviewed in this chapter can be found in [23, 37, 49, 50, 63, 68, 139, 152, 175, 253, 269, 330]. Ordered or embedded graphs are the subject of topological graph theory. See, for instance, [249, 138, 365], and also [236, Chap. 8]. General references to algorithms and data structures include [2, 31, 83, 193, 194, 195, 290, 363], and further references to combinatorial and graph algorithms include [69, 78, 107, 127, 130, 203, 232, 236, 229, 255, 271, 304, 321, 336, 367].

Literate programming was introduced in [188]. Some of the most influential articles about literate programming are collected in [191]. Large examples of literate programs include [116, 146, 189, 190, 192, 196, 260]. The *programming pearls* column of *Commun. ACM* has turned into a *literate programming* column. See [34, 35, 32, 89, 337, 338, 339, 340, 341, 342], and see also [42, 55, 56, 80, 82, 216, 217, 329]. The most widely used literate programming tools are

CWEB [144, 212, 213, 323] and noweb [264, 265]. LEDA includes extensive support for literate programming [236, Chap. 14], based on noweb. See also [65, 75, 76, 77, 115, 180, 294, 295]. The reader interested in further details of L^AT_EX is referred to [131, 132, 133, 205].

The importance of result checking was made evident by the famous division bug in the Intel Pentium microprocessor [47]. See also [235] and [236, Sect. 2.14]. Correctness certificates are known for several algorithms and data structures. See, for instance, [11, 96, 112, 120, 183, 320]. The correctness certificate for sorting algorithms is based on [52]. Correctness certificates are also used to achieve fault tolerance by detecting transient hardware and software faults. See, for instance, [44, 45, 46, 311, 358]. Alternative software testing approaches include the acceptance test method [266], where some tests are performed on the solution to a problem instance in order to determine if an error has occurred, and the n -version programming method [16], where $n \geq 2$ separate algorithms are implemented independently and executed on the same problem instance.

Adjacency matrices are most often used in combinatorics and graph theory. Adjacency lists are the preferred graph representation in computer science, though, because a large number of graph algorithms can be implemented to run in time linear in the number of vertices and arcs in the graph using an adjacency-list representation, while no graph algorithm can be implemented to run in linear time using an adjacency-matrix representation. The use of adjacency lists was advocated in [166, 317] and traversing incident arcs or edges, instead of adjacent vertices, was proposed in [102]. More compact, implicit representations of static graphs are studied in [10, 122, 154, 155, 177, 256, 314, 326]. See also [22, 110]. Implicit representations of static trees include Prüfer sequences [261], which are used for counting and generating trees. See also the bibliographic notes for Chap. 4.

Review Problems

1.1 Determine the size of K_n and $K_{p,q}$.

1.2 Determine the values of n for which C_n is bipartite, and also the values of n for which K_n is bipartite.

1.3 Give all the spanning trees of the graph in Fig. 1.29, and also the number of spanning trees in the underlying undirected graph.

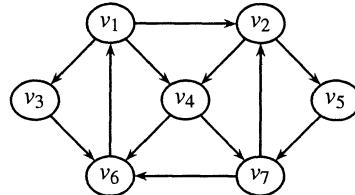


Fig. 1.29. Graph for problem 1.3.

1.4 Explain the main advantages and disadvantages of literate programming, from your particular point of view.

1.5 Extend the adjacency matrix graph representation by replacing those operations having an arc as argument or giving an arc as result, by corresponding operations having as argument or giving as result the source and target vertices of the arc: $G.\text{adj_pred}(v,w)$, $G.\text{adj_succ}(v,w)$, $G.\text{first_adj_edge}(v)$, $G.\text{last_adj_edge}(v)$, $G.\text{first_in_edge}(w)$, $G.\text{last_in_edge}(w)$, $G.\text{del_edge}(v,w)$, $G.\text{first_edge}()$, $G.\text{last_edge}()$, $G.\text{source}(v,w)$, $G.\text{target}(v,w)$, $G.\text{in_pred}(v,w)$, $G.\text{in_succ}(v,w)$, $G.\text{pred_edge}(v,w)$, and $G.\text{succ_edge}(v,w)$.

1.6 Extend the first-child, next-sibling tree representation, in order to support the collection of 12 basic operations in $O(1)$ time.

Exercises

1.1 The standard representation of an undirected graph in the format adopted for the DIMACS Implementation Challenges [171, 172] consists of a problem definition line of the form ‘ p edge $n m$ ’, where n and m are respectively the number of vertices and the number of edges, followed by m edge descriptor lines of the form ‘ $e i j$ ’, each of them giving an edge as a pair of vertex numbers in the range 1 to n .

Comment lines of the form ‘c ...’ are also allowed. Implement procedures to read a graph in DIMACS format in LEDA and to write a LEDA graph in DIMACS format.

1.2 The external representation of a graph in the Stanford GraphBase (SGB) format [192] consists essentially of a first line of the form ‘* GraphBase graph (util_types ..., nV, mA)’, where n and m are respectively the number of vertices and the number of arcs; a second line containing an identification string; a ‘* Vertices’ line; n vertex descriptor lines of the form ‘*label*, $A_i, 0, 0$ ’, where i is the number of the first arc in the range 0 to $m - 1$ going out of the vertex and *label* is a string label; an ‘* Arcs’ line; m arc descriptor lines of the form ‘ $V_j, A_i, label, 0$ ’, where j is the number of the target vertex in the range 0 to $n - 1$, i is the number of the next arc in the range 0 to $m - 1$ going out of the same source vertex, and *label* is an integer label; and a last ‘* Checksum ...’ line. Further, in the description of a vertex with no outgoing arc, or an arc with no successor going out of the same source vertex, ‘ A_i ’ becomes ‘0’. Implement procedures to read a SGB graph in LEDA and to write a LEDA graph in SGB format.

1.3 Write a literate program for computing the greatest common divisor of two integers. Include all those mathematical formulae, equations, tables, and diagrams you consider appropriate to better explain the greatest common divisor problem, the algorithm, and your implementation of the algorithm.

1.4 Take some program that you have written before, and rewrite it in a literate programming style. Explain any difficulties you may have found in remembering all of the relevant details about your program.

1.5 Give a certificate of implementation correctness for the greatest common divisor program of Exercise 1.3.

1.6 Give a visual checker of implementation correctness for the greatest common divisor program of Exercises 1.3 and 1.5, and implement an animation of the algorithm.

1.7 Give another certificate of implementation correctness for the LEDA member function *sort()* of the *list* class, based on some alternative method of determining multiset equality.

- 1.8** Implement the extended adjacency matrix graph representation given in Problem 1.5, wrapped in a C++ class, using either the LEDA *node_matrix* class, or the LEDA *array* class together with the internal numbering of the vertices.
- 1.9** Give an implementation of operation $T.\text{previous_sibling}(v)$ using the array-of-parents tree representation.
- 1.10** Implement the extended first-child, next-sibling tree representation of Problem 1.6, wrapped in a C++ class, using either the LEDA *node_array* class, or the LEDA *array2* class together with the internal numbering of the nodes.

2. Algorithmic Techniques

For decades, computer scientists have focused attention on problems that admit efficient algorithms. Algorithms whose running time grows no faster than some polynomial of low degree in the size of the input data still dominate textbooks on algorithms.

—Jürg Nievergelt [247]

Combinatorial problems on trees and graphs can be studied in the general framework of assigning values to nodes or vertices in such a way that some specified constraints are satisfied. Some of the basic algorithmic techniques for solving these problems—backtracking, branch-and-bound, divide-and-conquer, and dynamic programming—are presented in this chapter, and they are illustrated by the problem of computing the edit distance between two trees.

2.1 The Tree Edit Distance Problem

The problem of transforming or *editing* trees is a primary means of tree comparison, with practical applications in combinatorial pattern matching, pattern recognition, chemical structure search, computational molecular biology, and other areas of engineering and life sciences. In the most general sense, trees can be transformed by application of elementary edit operations, which have a certain cost or weight associated with them, and the edit distance between two trees is the cost of a least-cost sequence of elementary edit operations, or the length of a shortest sequence of elementary edit operations, that allows one to transform one of the trees into the other.

Rooted ordered trees are considered in this chapter, with the following elementary edit operations: deletion of a leaf node from a tree, insertion of a new leaf node into a tree, and substitution of a node in a tree for a node in another tree.

Definition 2.1. Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be ordered trees. An **elementary edit operation** on T_1 and T_2 is either the **deletion** from T_1 of a leaf node $v \in V_1$, denoted by $v \mapsto \lambda$ or (v, λ) ; the **substitution** of a node $w \in V_2$ for a node $v \in V_1$, denoted by $v \mapsto w$ or (v, w) ; or the **insertion** into T_2 of a node $w \notin V_2$ as a new leaf, denoted by $\lambda \mapsto w$ or (λ, w) . Deletion of a nonroot node $v \in V_1$ implies the deletion of the arc $(\text{parent}[v], v) \in E_1$, and insertion of a nonroot node $w \notin V_2$ as a child of a leaf node $\text{parent}[w] \in V_2$ implies insertion of an arc $(\text{parent}[w], w) \notin E_2$.

Deletion and insertion operations are thus made on leaves only. The deletion of a nonleaf node requires first the deletion of the whole subtree rooted at the node, and the same applies to the insertion of nonleaves.

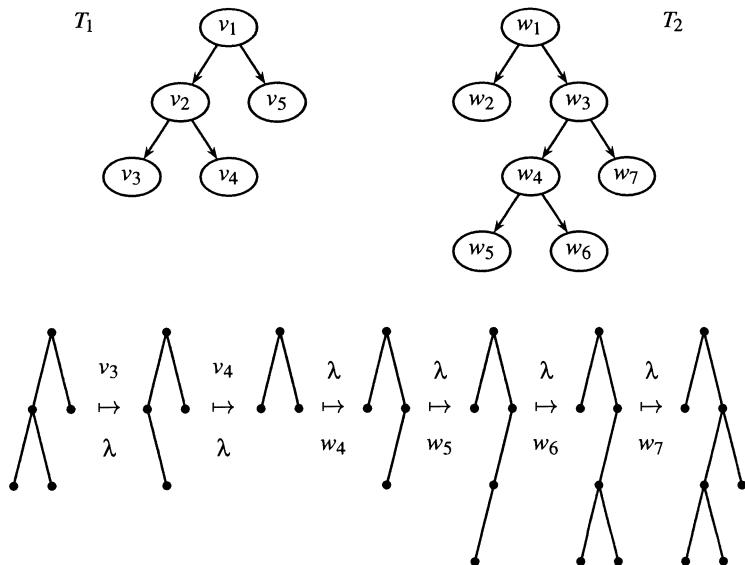


Fig. 2.1. A transformation between two ordered trees. Nodes are numbered according to the order in which they are visited during a preorder traversal.

Now, a tree can be transformed into another tree by application of a sequence of elementary edit operations. Recall that a relation $R \subseteq A \times B$ is a set of ordered pairs $\{(a, b) \mid a \in A, b \in B\}$. An *ordered relation* $R \subseteq A \times B$ is an ordered set, or sequence, of ordered pairs $[(a, b) \mid a \in A, b \in B]$.

Definition 2.2. Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be ordered trees. A *transformation* of T_1 into T_2 is an ordered relation $E \subseteq (V_1 \cup \{\lambda\}) \times (V_2 \cup \{\lambda\})$ such that

- $\{v \in V_1 \mid (v, w) \in E, w \in V_2 \cup \{\lambda\}\} = V_1$
- $\{w \in V_2 \mid (v, w) \in E, v \in V_1 \cup \{\lambda\}\} = V_2$
- $(v_1, w), (v_2, w) \in E$ implies $v_1 = v_2$, for all nodes $v_1, v_2 \in V_1 \cup \{\lambda\}$ and $w \in V_2$
- $(v, w_1), (v, w_2) \in E$ implies $w_1 = w_2$, for all nodes $v \in V_1$ and $w_1, w_2 \in V_2 \cup \{\lambda\}$.

Remark 2.3. Notice that in a transformation $E \subseteq (V_1 \cup \{\lambda\}) \times (V_2 \cup \{\lambda\})$ of an ordered tree $T_1 = (V_1, E_1)$ into an ordered tree $T_2 = (V_2, E_2)$, the relation $E \cap V_1 \times V_2$ is a bijection. In fact, the condition $(v_1, w), (v_2, w) \in E$ implies $v_1 = v_2$ establishes injectivity, and $(v, w_1), (v, w_2) \in E$ implies $w_1 = w_2$ establishes the surjectivity of $E \cap V_1 \times V_2$.

Example 2.4. A transformation of an ordered tree T_1 into another ordered tree T_2 is illustrated in Fig. 2.1. The transformation consists of deleting leaves v_3, v_4 from T_1 , substituting nodes v_1, v_2, v_5 of T_1 by nodes w_1, w_2, w_3 of T_2 , respectively, and inserting leaves w_4, w_5, w_6, w_7 into T_2 . Substitution of corresponding nodes is left implicit in the figure. The transformation is denoted by $[v_1 \mapsto w_1, v_2 \mapsto w_2, v_3 \mapsto \lambda, v_4 \mapsto \lambda, v_5 \mapsto w_3, \lambda \mapsto w_4, \lambda \mapsto w_5, \lambda \mapsto w_6, \lambda \mapsto w_7]$ and also $[(v_1, w_1), (v_2, w_2), (v_3, \lambda), (v_4, \lambda), (v_5, w_3), (\lambda, w_4), (\lambda, w_5), (\lambda, w_6), (\lambda, w_7)]$.

Not every sequence of elementary edit operations corresponds to a valid transformation between two ordered trees, though. On the one hand, deletions and insertions must appear in a bottom-up order—for instance, according to a postorder traversal of the trees—in order to ensure that deletions and insertions are indeed made on leaves. In

the postorder traversal of an ordered tree, which will be introduced in Sect. 3.2, a postorder traversal of the subtree rooted in turn at each of the children of the root is performed first, and the root of the tree is visited last, where the subtree rooted at the first child is traversed first, followed by the subtree rooted at the next sibling, etc.

On the other hand, for a tree transformation to be valid both parent and sibling order must be preserved by the transformation, in order to ensure that the result of the transformation is indeed an ordered tree. That is, in a valid transformation of an ordered tree T_1 into another ordered tree T_2 , the parent of a nonroot node of T_1 which is substituted by a nonroot node of T_2 must be substituted by the parent of the node of T_2 . Further, whenever sibling nodes of T_1 are substituted by sibling nodes of T_2 , the substitution must preserve their relative order.

The latter requirement for a transformation between two ordered trees to be valid is formalized by the notion of a *mapping*, also known as a *trace*.

Definition 2.5. Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be ordered trees, let $W_1 \subseteq V_1$, and let $W_2 \subseteq V_2$. A *mapping* M of T_1 to T_2 is a bijection $M \subseteq W_1 \times W_2$ such that

- $(\text{root}[T_1], \text{root}[T_2]) \in M$ if $M \neq \emptyset$
- $(v, w) \in M$ only if $(\text{parent}[v], \text{parent}[w]) \in M$, for all nonroot nodes $v \in W_1$ and $w \in W_2$
- v_1 is a left sibling of v_2 if and only if w_1 is a left sibling of w_2 , for all nodes $v_1, v_2 \in W_1$ and $w_1, w_2 \in W_2$ with $(v_1, w_1), (v_2, w_2) \in M$.

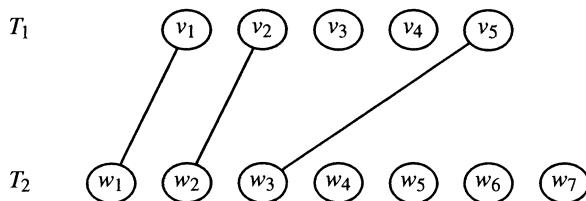


Fig. 2.2. The mapping underlying the tree transformation of Fig. 2.1. Nodes are numbered according to the order in which they are visited during a preorder traversal.

Example 2.6. The mapping M corresponding to the transformation of $T_1 = (V_1, E_1)$ to $T_2 = (V_2, E_2)$ given in Example 2.4 is illustrated in Fig. 2.2. As a matter of fact, $M \subseteq \{v_1, v_2, v_5\} \times \{w_1, w_2, w_3\} \subseteq V_1 \times V_2$ is indeed a bijection such that node v_1 , the parent of node v_2 , is mapped to node w_1 , the parent of node w_2 , because node v_2 is mapped to node w_2 and, furthermore, node v_2 , a left sibling of node v_5 , is mapped to node w_2 , a left sibling of node w_3 , and node v_5 is mapped to node w_3 .

Lemma 2.7. *Let M be a mapping of an ordered tree $T_1 = (V_1, E_1)$ to another ordered tree $T_2 = (V_2, E_2)$. Then, $\text{depth}[v] = \text{depth}[w]$ for all $(v, w) \in M$.*

Proof. Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be ordered trees, and let $M \subseteq W_1 \times W_2$ be a mapping of T_1 to T_2 , where $W_1 \subseteq V_1$, and $W_2 \subseteq V_2$. Suppose $\text{depth}[v] \neq \text{depth}[w]$ for some nodes $v \in V_1$ and $w \in V_2$ with $(v, w) \in M$ and assume, without loss of generality, that $\text{depth}[v] < \text{depth}[w]$.

The proof is by induction on the depth of node v . If $\text{depth}[v] = 0$ then $v = \text{root}[T_1]$, and therefore $w \neq \text{root}[T_2]$, because $\text{depth}[w] > 0 = \text{depth}[v]$ and, since M is a bijection, $(\text{root}[T_1], \text{root}[T_2]) \notin M \neq \emptyset$, contradicting the assumption that M is a mapping. Then, $\text{depth}[v] = \text{depth}[w]$.

Assume now that $\text{depth}[x] = \text{depth}[y]$ for all $(x, y) \in M$ with $\text{depth}[x] \leq n$, where $n < \text{depth}[T_1]$. If $\text{depth}[v] = n + 1$, then $(x, y) \in M$ for some children $x \in V_1$ and $y \in V_2$ with $\text{parent}[x] = v$ and $\text{parent}[y] = w$, because M is a mapping, and $\text{depth}[x] = \text{depth}[y] = n$ by the induction hypothesis. Then, $\text{depth}[v] = \text{depth}[x] + 1 = \text{depth}[y] + 1 = \text{depth}[w]$. Therefore, $\text{depth}[v] = \text{depth}[w]$ for all $(v, w) \in M$. \square

A transformation is thus valid if node deletions and insertions are made on leaves only, and node substitutions constitute a mapping.

Definition 2.8. *Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be ordered trees. A transformation $E \subseteq (V_1 \cup \{\lambda\}) \times (V_2 \cup \{\lambda\})$ of T_1 into T_2 is said to be a valid transformation if*

- (v_j, λ) occurs before (v_i, λ) in E , for all $(v_i, \lambda), (v_j, \lambda) \in E \cap V_1 \times \{\lambda\}$ such that node v_j is a descendent of node v_i in T_1

- (λ, w_i) occurs before (λ, w_j) in E , for all $(\lambda, w_i), (\lambda, w_j) \in E \cap \{\lambda\} \times V_2$ such that node w_j is a descendent of node w_i in T_2
- $E \cap V_1 \times V_2$ is a mapping of T_1 to T_2 .

Now, the elementary edit operations of deletion, substitution, and insertion are, as a matter of fact, sufficient for the transformation of any tree into any other tree.

Lemma 2.9. *Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be ordered trees. Then, there is a sequence of elementary edit operations transforming T_1 into T_2 .*

Proof. Let $[v_i, \dots, v_j]$ be the set of nodes V_1 in the order in which they are first visited during a postorder traversal of T_1 , and let $[w_k, \dots, w_\ell]$ be the set of nodes V_2 in the order in which they are first visited during a postorder traversal of T_2 . Then, the sequence of elementary edit operations $[v_i \mapsto \lambda, \dots, v_j \mapsto \lambda, \lambda \mapsto w_k, \dots, \lambda \mapsto w_\ell]$ allows one to transform T_1 into T_2 . \square

It follows from the proof of Lemma 2.9 that the elementary edit operation of substitution is not necessary for the transformation of a tree into another tree. However, substitutions are convenient because they allow the determination of the shortest or, in general, least-cost transformations between trees.

Definition 2.10. *Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be ordered trees. The cost of an elementary edit operation on T_1 and T_2 is given by a function $\gamma: V_1 \cup V_2 \cup \{\lambda\} \times V_1 \cup V_2 \cup \{\lambda\} \rightarrow \mathbb{R}$ such that*

- $\gamma(v, w) \geq 0$
- $\gamma(v, w) = 0$ if and only if $v = w$
- $\gamma(v, w) = \gamma(w, v)$
- $\gamma(v, w) \leq \gamma(v, z) + \gamma(z, w)$

for all $v, w, z \in V_1 \cup V_2 \cup \{\lambda\}$.

The first two conditions in Definition 2.10 are known as *nonnegativity* and *definiteness*, the third condition is the *symmetry* of the cost function, and the fourth condition is known as *triangularity* or the *triangular inequality*. Together, these conditions on the cost of elementary edit operations turn ordered trees into a metric space.

Definition 2.11. Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be ordered trees. The **cost** of a transformation $E \subseteq (V_1 \cup \{\lambda\}) \times (V_2 \cup \{\lambda\})$ of T_1 into T_2 is given by $\gamma(E) = \sum_{(v,w) \in E} \gamma(v, w)$.

Now, the edit distance between two ordered trees is the cost of a least-cost valid transformation between the trees.

Definition 2.12. The **edit distance** between ordered trees T_1 and T_2 is $\delta(T_1, T_2) = \min\{\gamma(E) \mid E \text{ is a valid transformation of } T_1 \text{ to } T_2\}$.

Example 2.13. Assume that the cost of elementary edit operations is such that $\gamma(v, w) = 1$ if either $v = \lambda$ or $w = \lambda$, and $\gamma(v, w) = 0$ otherwise. That is, node deletions and insertions have cost equal to one, and node substitutions have zero cost. Then, the cost of the transformation of T_1 into T_2 given in Example 2.4 is equal to 6. A least-cost valid transformation of T_1 into T_2 is given by $[(v_1, w_1), (\lambda, w_2), (v_2, w_3), (v_3, w_4), (\lambda, w_5), (\lambda, w_6), (v_4, w_7), (v_5, \lambda)]$, and the edit distance between T_1 and T_2 is thus equal to 4.

An alternative formulation of the tree edit problem consists of building a kind of grid graph on the nodes of the ordered trees, called the *edit graph* of the trees, and then reducing the problem of finding a valid transformation of one tree into the other to that of finding a path in the edit graph from one corner down to the opposite corner.

Recall that in the preorder traversal of an ordered tree, which will be discussed in more detail in Sect. 3.1, the root of the tree is visited first, followed by a preorder traversal of the subtree rooted in turn at each of the children of the root, where the subtree rooted at the first child is traversed first, followed by the subtree rooted at the next sibling, etc.

Definition 2.14. Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be ordered trees. The **edit graph** of T_1 and T_2 has a vertex of the form vw for each pair of nodes $v \in \{v_0\} \cup V_1$ and $w \in \{w_0\} \cup V_2$, where $v_0 \notin V_1$ and $w_0 \notin V_2$ are dummy nodes. Further,

- $(v_i w_j, v_{i+1} w_j) \in E$ if and only if $\text{depth}[v_{i+1}] \geq \text{depth}[w_{j+1}]$
- $(v_i w_j, v_{i+1} w_{j+1}) \in E$ if and only if $\text{depth}[v_{i+1}] = \text{depth}[w_{j+1}]$
- $(v_i w_j, v_i w_{j+1}) \in E$ if and only if $\text{depth}[v_{i+1}] \leq \text{depth}[w_{j+1}]$

for $0 \leq i < n_1$ and $0 \leq j < n_2$, where nodes are numbered according to the order in which they are visited during a preorder traversal of the trees. Moreover, $(v_i w_{n_2}, v_{i+1} w_{n_2}) \in E$ for $0 \leq i < n_1$, and $(v_{n_1} w_j, v_{n_1} w_{j+1}) \in E$ for $0 \leq j < n_2$.

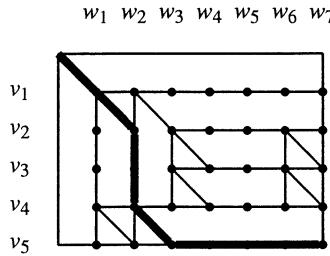


Fig. 2.3. Edit graph for the trees of Fig. 2.1. The path in the graph corresponding to the valid transformation of T_1 to T_2 given in Example 2.4 is shown highlighted.

In the edit graph of ordered trees T_1 and T_2 , a vertical arc of the form $(v_i w_j, v_{i+1} w_j)$ represents the deletion of node v_{i+1} from T_1 , where the condition $\text{depth}[v_{i+1}] \geq \text{depth}[w_{j+1}]$ ensures that node w_{j+1} does not belong to the subtree of T_2 rooted at node w_j , that is, node w_{j+1} does not have to be inserted into T_2 before node v_{i+1} can be deleted from T_1 . Further, a *diagonal* arc of the form $(v_i w_j, v_{i+1} w_{j+1})$ represents the substitution of node w_{j+1} of T_2 for node v_{i+1} of T_1 , where the condition $\text{depth}[v_{i+1}] = \text{depth}[w_{j+1}]$ follows from Lemma 2.7. A *horizontal* arc, on the other hand, of the form $(v_i w_j, v_i w_{j+1})$ represents the insertion of node w_{j+1} into T_2 , where the condition $\text{depth}[v_{i+1}] \leq \text{depth}[w_{j+1}]$ ensures that node v_{i+1} does not belong to the subtree of T_1 rooted at node v_i , that is, node v_{i+1} does not have to be deleted from T_1 before node w_{j+1} can be inserted into T_2 .

Missing horizontal and diagonal arcs thus ensure that once a path in the edit graph traverses a vertical arc, corresponding to the deletion of a certain node, that path can only be extended by traversing further vertical arcs, corresponding to the deletion of all the nodes in the subtree rooted at that node. Conversely, missing vertical and diagonal arcs ensure that once a path in the edit graph traverses a horizontal arc, corresponding to the insertion of a certain node, that path can only be

extended by traversing further horizontal arcs, corresponding to the insertion of all the nodes in the subtree rooted at that node.

Example 2.15. The edit graph for the trees of Fig. 2.1 is shown in Fig. 2.3. In the drawing of a tree edit graph, nodes are placed in a rectangular grid in the order in which they are visited during a preorder traversal of the trees, deletions correspond to vertical edges, substitutions correspond to diagonal edges, and insertions correspond to horizontal edges. Further, edges are indeed arcs, directed from left to right and from top to bottom. The highlighted path in Fig. 2.3 corresponds to the transformation of T_1 to T_2 given in Example 2.4, consisting of the substitution of w_1 for v_1 , the substitution of w_2 for v_2 , the deletion of v_3 and v_4 from T_1 , the substitution of w_3 for v_5 , and the insertion of w_4 , w_5 , w_6 , and w_7 into T_2 .

The correspondence of valid transformations between ordered trees T_1 and T_2 , and paths in the edit graph of T_1 and T_2 from the top-left to the bottom-right corner, is given by the fact that node substitutions (diagonal arcs) along a path in the edit graph of T_1 and T_2 from the top-left to the bottom-right corner constitute a mapping of T_1 to T_2 , which can thus be extended to a valid transformation of T_1 into T_2 .

Lemma 2.16. *Let P be a path in the edit graph of ordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ from the top-left to the bottom-right corner, and let $M = \{(v_{i+1}, w_{j+1}) \in V_1 \times V_2 \mid (v_i w_j, v_{i+1} w_{j+1}) \in P\}$. Then, M is a mapping of T_1 to T_2 . Conversely, let $M \subseteq V_1 \times V_2$ be a mapping of T_1 to T_2 . Then, there is a path P in the edit graph of T_1 and T_2 from the top-left to the bottom-right corner such that $\{(v_{i+1}, w_{j+1}) \in V_1 \times V_2 \mid (v_i w_j, v_{i+1} w_{j+1}) \in P\} = M$.*

Proof. Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be ordered trees, let P be a path in the edit graph of T_1 and T_2 from the top-left to the bottom-right corner, and let $M = \{(v_{i+1}, w_{j+1}) \in V_1 \times V_2 \mid (v_i w_j, v_{i+1} w_{j+1}) \in P\}$. If $(v_1, w_1) \notin M$, then $(v_0 v_1, w_0 w_1) \notin P$ and then, P has no diagonal arcs and thus $M = \emptyset$, that is, M is a trivial mapping of T_1 to T_2 .

Otherwise, $(v_1, w_1) \in M$ and, since nodes of T_1 and T_2 are numbered according to the order in which they are visited during a preorder traversal of the trees, $\text{root}[T_1] = v_1$ and $\text{root}[T_2] = w_1$. Then, $(\text{root}[T_1], \text{root}[T_2]) \in M$.

Now, let $v_{i+1} \in V_1$ and $w_{j+1} \in V_2$ be nodes with $(v_{i+1}, w_{j+1}) \in M$, where $i, j \neq 0$, let $v_{k+1} = \text{parent}[v_{i+1}]$, let $w_{\ell+1} = \text{parent}[w_{j+1}]$, and suppose $(v_{k+1}, w_{\ell+1}) \notin M$. Since $(v_k w_\ell, v_{k+1} w_{\ell+1}) \notin P$, there is no path contained in P from node $v_k w_\ell$ to node $v_i w_j$, because $k < i, \ell < j$, $\text{depth}[v_k] < \text{depth}[v_i]$, and $\text{depth}[w_\ell] < \text{depth}[w_j]$, thus contradicting the hypothesis that $(v_{i+1}, w_{j+1}) \in M$, that is, $(v_i w_j, v_{i+1} w_{j+1}) \in P$. Then, $(\text{parent}[v_{i+1}], \text{parent}[w_{j+1}]) \in M$.

Finally, let $v_{i+1}, v_{k+1} \in V_1$ and $w_{j+1}, w_{\ell+1} \in V_2$, $i, j, k, \ell \neq 0$, be nodes with $(v_{i+1}, w_{j+1}), (v_{k+1}, w_{\ell+1}) \in M$. Then, v_{i+1} is a left sibling of v_{k+1} if and only if w_{j+1} is a left sibling of $w_{\ell+1}$, because nodes in the edit graph are numbered according to a preorder traversal of the trees, and arcs in the edit graph go from lower-numbered to equal or higher-numbered nodes. Therefore, M is a mapping of T_1 to T_2 .

Conversely, let $M \subseteq V_1 \times V_2$ be a mapping of T_1 to T_2 , and let X be the set of arcs in the edit graph of T_1 and T_2 induced by M , that is, $X = \{(v_i w_j, v_{i+1} w_{j+1}) \in (\{v_0\} \cup V_1) \times (\{w_0\} \cup V_2) \mid (v_{i+1}, w_{j+1}) \in M\}$. Since M is an injection, there are no parallel diagonal arcs in X of the form $(v_i w_j, v_{i+1} w_{j+1})$ and $(v_k w_\ell, v_{k+1} w_{\ell+1})$ with $i \neq k$. Further, since M is a mapping, there are no parallel diagonal arcs in X of the form $(v_i w_j, v_{i+1} w_{j+1})$ and $(v_i w_\ell, v_{i+1} w_{\ell+1})$ with $j \neq \ell$. Therefore, there is a path $P \supseteq X$ in the edit graph of T_1 and T_2 from the top-left to the bottom-right corner such that $\{(v_{i+1}, w_{j+1}) \in V_1 \times V_2 \mid (v_i w_j, v_{i+1} w_{j+1}) \in P\} = M$. \square

Remark 2.17. Notice that a valid transformation of T_1 into T_2 can be obtained from a path in the edit graph of T_1 and T_2 from the top-left to the bottom-right corner, by just reordering the edit operations between each pair of successive node substitutions according to the preorder traversal of the trees. That is, in such a way that deletion and insertion of children nodes do not precede deletion and insertion of their parent or their left sibling nodes.

Now, computing the edit distance between two ordered trees can be reduced to the problem of finding a shortest path in the edit graph of the trees, with arcs in the edit graph labeled or weighted by the cost of the respective tree edit operation.

Lemma 2.18. *Let G be the edit graph of ordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$, with arcs of the form $(v_i w_j, v_k w_\ell) \in E$ weighted by*

$\gamma(v_k, \lambda)$ if $k = i + 1$ and $\ell = j$, weighted by $\gamma(v_k, w_\ell)$ if $k = i + 1$ and $\ell = j + 1$, and weighted by $\gamma(\lambda, w_\ell)$ if $k = i$ and $\ell = j + 1$. Let also P be a shortest path in G from the top-left to the bottom-right corner, and let $E = \{(v_{i+1}, \lambda) \in V_1 \times \{\lambda\} \mid (v_i w_j, v_{i+1} w_j) \in P\} \cup \{(v_{i+1}, w_{j+1}) \in V_1 \times V_2 \mid (v_i w_j, v_{i+1} w_{j+1}) \in P\} \cup \{(\lambda, w_{j+1}) \in \{\lambda\} \times V_2 \mid (v_i w_j, v_i w_{j+1}) \in P\}$. Then, $\delta(T_1, T_2) = \sum_{(v,w) \in E} \gamma(v, w)$.

Proof. Let G be the weighted edit graph of ordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$, let P be a shortest path in G from the top-left to the bottom-right corner, and let $E = \{(v_{i+1}, \lambda) \in V_1 \times \{\lambda\} \mid (v_i w_j, v_{i+1} w_j) \in P\} \cup \{(v_{i+1}, w_{j+1}) \in V_1 \times V_2 \mid (v_i w_j, v_{i+1} w_{j+1}) \in P\} \cup \{(\lambda, w_{j+1}) \in \{\lambda\} \times V_2 \mid (v_i w_j, v_i w_{j+1}) \in P\}$. Then, $E \cap V_1 \times V_2$ is a mapping of T_1 to T_2 , by Lemma 2.16. Further, E can be reordered into a valid transformation of T_1 into T_2 , by Remark 2.17 and then $\delta(T_1, T_2) = \sum_{(v,w) \in E} \gamma(v, w)$ by Definition 2.12. \square

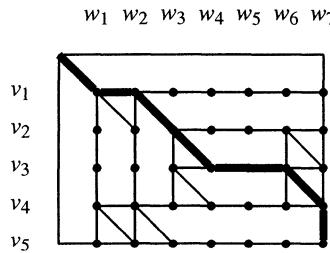


Fig. 2.4. Shortest path in the edit graph of two ordered trees.

Example 2.19. A shortest path in the edit graph for the trees from Fig. 2.1, corresponding to the least-cost transformation of T_1 to T_2 given in Example 2.13, is shown highlighted in Fig. 2.4.

The following procedure builds the edit graph G of ordered trees T_1 and T_2 . The edit graph is parametrized by a string label for each vertex and another string label for each arc. Vertices are labeled by a string consisting of the preorder number of the nodes of T_1 and T_2 they correspond to, separated by a colon, and arcs are labeled by a string indicating the tree edit operation they correspond to: “del” for

deletion of a node from T_2 , “sub” for substitution of a node of T_2 for a node of T_1 , and “ins” for insertion of a node into T_2 .

66

$\langle \text{techniques } 66 \rangle \equiv$

```

void tree_edit_graph(
    tree& T1,
    tree& T2,
    GRAPH<string,string>& G)
{
    G.clear();
    rearrange_tree_in_preorder(T1);
    rearrange_tree_in_preorder(T2);
    int n1 = T1.number_of_nodes();
    int n2 = T2.number_of_nodes();

    node array<int> num1(T1), num2(T2), depth1(T1), depth2(T2);
    preorder_tree_depth(T1,num1,depth1);
    preorder_tree_depth(T2,num2,depth2);
    node v,w;
    array<int> d1(1,n1);
    forall_nodes(v,T1)
        d1[num1[v]] = depth1[v];
    array<int> d2(1,n2);
    forall_nodes(w,T2)
        d2[num2[w]] = depth2[w];

    array2<node> A(0,n1,0,n2);
    for (int i = 0; i <= n1; i++) {
        for (int j = 0; j <= n2; j++) {
            A(i,j) = v = G.new_node(string(“%i:%i”,i,j));
        }
    }

    for (int i = 0; i < n1; i++) G.new_edge(A(i,n2),A(i+1,n2),“del”);
    for (int j = 0; j < n2; j++) G.new_edge(A(n1,j),A(n1,j+1),“ins”);
    for (int i = 0; i < n1; i++) {
        for (int j = 0; j < n2; j++) {
            if (d1[i+1] ≥ d2[j+1]) G.new_edge(A(i,j),A(i+1,j),“del”);
            if (d1[i+1] ≡ d2[j+1]) G.new_edge(A(i,j),A(i+1,j+1),“sub”);
            if (d1[i+1] ≤ d2[j+1]) G.new_edge(A(i,j),A(i,j+1),“ins”);
        }
    }
}
```

The actual computation of a shortest path P in the edit graph G of ordered trees T_1 and T_2 is done by the following procedure. The shortest path P is recovered from the predecessor arcs $pred$ computed by the LEDA acyclic shortest path procedure, by starting off with the last vertex of G (the bottom-right corner of the edit graph) and following up the thread of predecessor arcs until reaching the first vertex of G (the top-left corner of the edit graph), which has no predecessor arc. The shortest path P is represented by a list of arcs (of G).

The cost of elementary tree edit operations is set to $\gamma(v, w) = 1$, for all edit operations of the form $v \mapsto w$ with $v \in V_1 \cup \{\lambda\}$, $w \in V_2 \cup \{\lambda\}$, and $v \neq w$, by defining the weight or cost of the respective arcs in the edit graph of the trees.

67a

```
(techniques 66) +≡
void tree_edit(
    tree& T1,
    tree& T2,
    GRAPH<string,string>& G,
    list<edge>& P)
{
    tree_edit_graph(T1,T2,G);
    node s = G.first_node(); // top-left corner
    node t = G.last_node(); // bottom-right corner

    edge_array<double> cost(G,1); // cost of elementary edit operations
    node_array<edge> pred(G);
    node_array<double> dist(G);
    ACYCLIC_SHORTEST_PATH(G,s,cost,dist,pred);

    P.clear();
    node v = t;
    edge e;
    while (pred[v] ≠ nil) {
        e = pred[v];
        P.push(e);
        v = G.source(e);
    }
}
```

A few implementation details still need to be filled in. The preorder number of the nodes is further used to rearrange the representation of the trees, in such a way that the order on the nodes fixed by the representation of each of the trees coincides with the order in which they are visited during a preorder traversal of the tree, as required by the formulation of the tree edit problem.

67b

```
(techniques 66) +≡
void rearrange_tree_in_preorder(
    tree& T)
{
    node_array<int> order(T);
    node_array<int> depth(T);
    preorder_tree_depth(T,order,depth);
    T.sort_nodes(order);
}
```

Lemma 2.20. *The tree edit algorithm based on shortest paths in the edit graph for finding a least-cost transformation of an ordered tree*

T_1 to an ordered tree T_2 with respectively n_1 and n_2 nodes, runs in $O(n_1 n_2)$ time using $O(n_1 n_2)$ additional space.

Proof. Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be ordered trees with respectively n_1 and n_2 nodes. The edit graph of T_1 and T_2 has $(n_1 + 1)(n_2 + 1)$ vertices and at most $3n_1 n_2$ arcs, and building the edit graph thus takes $O(n_1 n_2)$ time and uses $O(n_1 n_2)$ additional space. Further, the acyclic shortest path procedure also takes $O(n_1 n_2)$ time. Therefore, the tree edit algorithm runs in $O(n_1 n_2)$ time using $O(n_1 n_2)$ additional space. \square

2.1.1 Interactive Demonstration of Tree Edit

The tree edit algorithm based on shortest paths in the edit graph is integrated next in the interactive demonstration of graph algorithms. A simple checker for a tree edit that provides some visual reassurance consists of highlighting the shortest path P in the edit graph G corresponding to a least-cost transformation between ordered trees T_1 and T_2 , where each arc in the shortest path corresponds to an elementary edit operation on the trees. The target vertices of those arcs of P that correspond to the deletion of a node from T_1 are shown in red, the target vertices of arcs in G that correspond to the substitution of a node of T_2 for a node of T_1 are shown in green, and the target vertices of arcs in G that correspond to the insertion of a node into T_2 are shown in blue.

```
68 {demo techniques 68}≡
void gw_tree_edit(
    GraphWin& gw1)
{
    graph& G1 = gw1.get_graph();
    tree T1(G1);

    graph G2;
    GraphWin gw2(G2,500,500,"Tree Edit");
    gw2.display();
    gw2.message("Enter second tree. Press done when finished");
    gw2.edit();
    gw2.del_message();

    tree T2(G2);

    GRAPH<string,string> G;
    list<edge> P;
```

```

tree_edit(T1,T2,G,P);
⟨set layout of edit graph 69⟩

gw1.save_all_attributes();
gw1.set_node_label_type(data_label);
gw2.set_node_label_type(data_label);
node_array<int> order1(T1);
preorder_tree_traversal(T1,order1);
node_array<int> order2(T2);
preorder_tree_traversal(T2,order2);
node v,w;
forall_nodes(v,T1)
  gw1.set_label(v,string("%i ",order1[v]));
forall_nodes(w,T2)
  gw2.set_label(w,string("%i ",order2[w]));

GraphWin gw(G,500,500,"Tree Edit");
gw.set_node_label_type(data_label);
gw.display(window::center,window::center);
gw.adjust_coords_to_win(xcoord,ycoord);
gw.set_layout(xcoord,ycoord);

edge e;
forall(e,P) {
  v = G.target(e);
  if (G[e] ≡ "del") gw.set_color(v,red); // deletion
  if (G[e] ≡ "sub") gw.set_color(v,green); // substitution
  if (G[e] ≡ "ins") gw.set_color(v,blue); // insertion
  gw.set_width(e,3);
}

gw.wait();
gw1.restore_all_attributes();
}

```

Visual reassurance is further enhanced by laying out the edit graph along a rectangular grid, as is usual for the layout of grid graphs.

69

```

⟨set layout of edit graph 69⟩≡
node_array<double> xcoord(G);
node_array<double> ycoord(G);
{ int n1 = T1.number_of_nodes();
  int n2 = T2.number_of_nodes();
  int N = (n1 + 1) * (n2 + 1);
  xcoord.init(G,N,0);
  ycoord.init(G,N,0);
  double d = 1.0 ÷ ((n1 + 1) + (n2 + 1) + 1);
  int i,j;
  node v;
  forall_nodes(v,G) {
    i = index(v) ÷ (n2 + 1);
    j = index(v) - i * (n2 + 1);
  }
}

```

```

    xcoord[v] = (j + 1) * d;
    ycoord[v] = - (i + 1) * d;
}
}

```

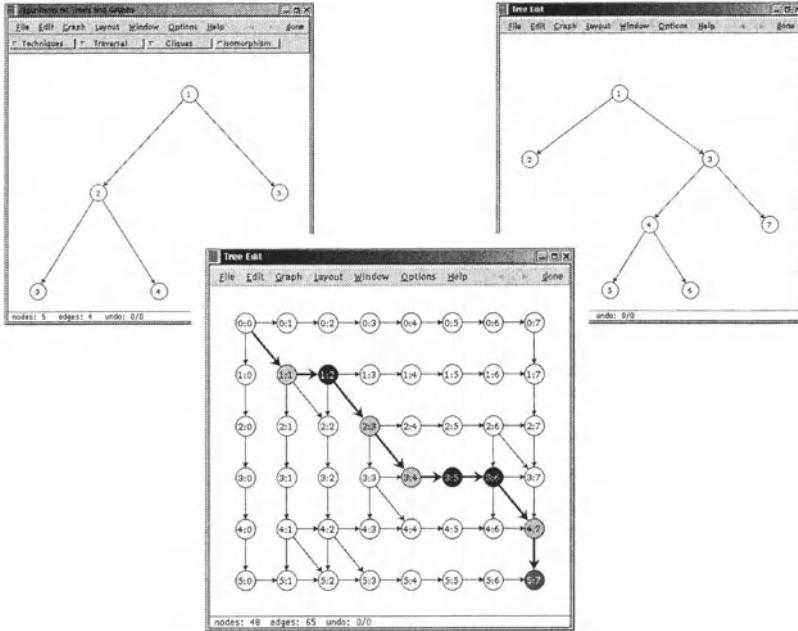


Fig. 2.5. Execution of the tree edit procedure based on shortest paths in the edit graph of ordered trees.

Example 2.21. Execution of the interactive demonstration of graph algorithms to produce a shortest path in the edit graph for the trees of Fig. 2.1 is illustrated in Fig. 2.5. The shortest path shown corresponds to the least-cost transformation of T_1 to T_2 given in Example 2.13, also shown in Example 2.19.

2.2 Backtracking

The solution to combinatorial problems on trees and graphs often requires an exhaustive search of the set of all possible solutions. As a

matter of fact, most combinatorial problems on trees and graphs can be stated in the general framework of assigning values of a finite domain to each of a series of variables, where the value for each variable must be taken from a corresponding finite domain. An ordered set of values satisfying some specified constraints represents a solution to the problem, and an exhaustive search for a solution must consider the elements of the Cartesian product of variable domains as potential solutions.

Consider, for instance, the tree edit problem. Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be ordered trees. A potential solution to the problem of finding a transformation of T_1 into T_2 is given by a bijection $M \subseteq W_1 \times W_2$, where $W_1 \subseteq V_1$ and $W_2 \subseteq V_2$, which can also be seen as an assignment of a node $w \in V_2 \cup \{\lambda\}$ to each node $v \in V_1$. The dummy node λ represents the deletion of a node from T_1 , and the insertion of a node into T_2 remains implicit, that is, all nodes $w \in V_2 \setminus W_2$ are inserted into T_2 .

M	v_1	v_2	v_3	v_4	v_5
$\{\}$	λ	λ	λ	λ	λ
$\{(v_1, w_1)\}$	w_1	λ	λ	λ	λ
$\{(v_1, w_1), (v_5, w_2)\}$	w_1	λ	λ	λ	w_2
$\{(v_1, w_1), (v_5, w_3)\}$	w_1	λ	λ	λ	w_3
$\{(v_1, w_1), (v_2, w_2)\}$	w_1	w_2	λ	λ	λ
$\{(v_1, w_1), (v_2, w_2), (v_5, w_3)\}$	w_1	w_2	λ	λ	w_3
$\{(v_1, w_1), (v_2, w_3)\}$	w_1	w_3	λ	λ	λ
$\{(v_1, w_1), (v_2, w_3), (v_4, w_4)\}$	w_1	w_3	λ	w_4	λ
$\{(v_1, w_1), (v_2, w_3), (v_5, w_7)\}$	w_1	w_3	λ	λ	w_7
$\{(v_1, w_1), (v_2, w_3), (v_3, w_4)\}$	w_1	w_3	w_4	λ	λ
$\{(v_1, w_1), (v_2, w_3), (v_3, w_4), (v_4, w_7)\}$	w_1	w_3	w_4	w_7	λ
$\{(v_1, w_1), (v_2, w_3), (v_3, w_7)\}$	w_1	w_3	w_7	λ	λ

Fig. 2.6. Valid transformations between the ordered trees of Fig. 2.1.

However, not every assignment of a node $w \in V_2 \cup \{\lambda\}$ to each node $v \in V_1$ corresponds to a valid transformation of T_1 into T_2 . The bijection $M \subseteq W_1 \times W_2 \subseteq V_1 \times V_2$ must be a mapping of T_1 to T_2 , according to Definition 2.8. Notice that in this case a valid transformation of T_1 into T_2 can be obtained by first deleting from T_1 all those nodes of T_1 which were assigned to the dummy node λ , in the order given by a preorder traversal of T_1 ; substituting all nodes of T_1 which were as-

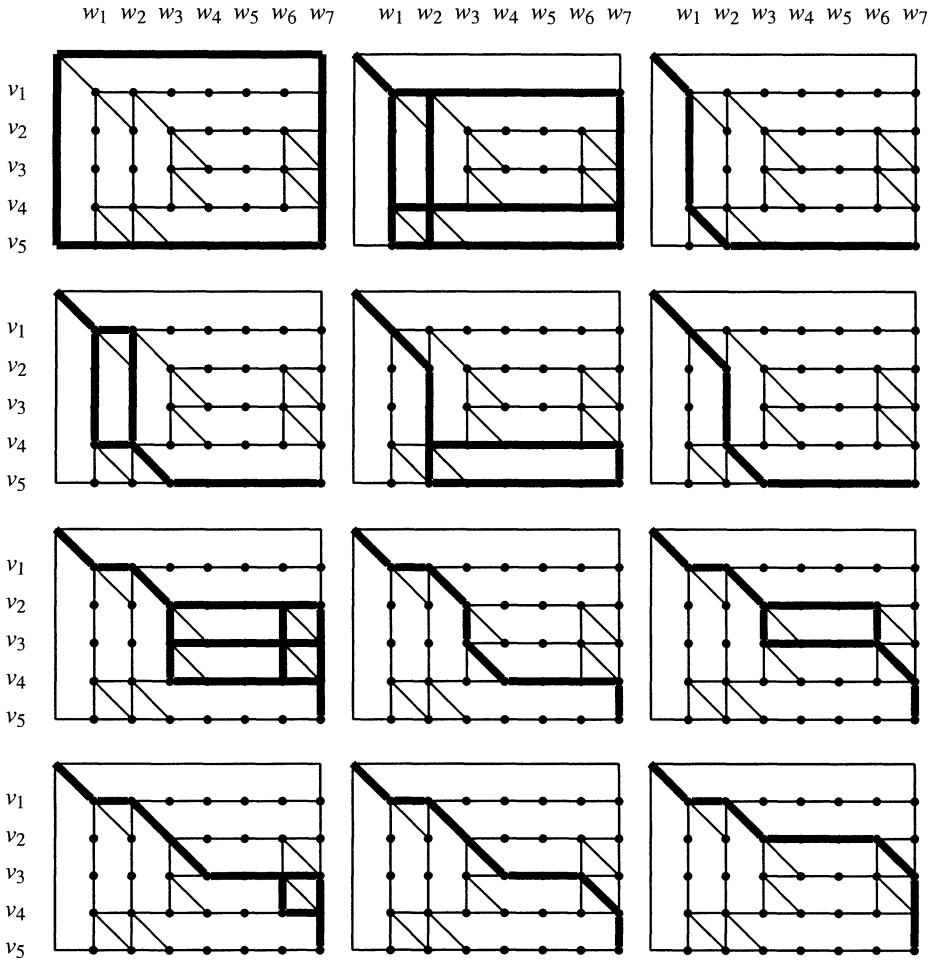


Fig. 2.7. Transformations between the ordered trees of Fig. 2.1. All paths in the edit graphs that correspond to a valid transformation of T_1 to T_2 are shown highlighted.

signed to a nondummy node of T_2 by the node they were assigned to; and inserting into T_2 all those nodes of T_2 which were not assigned to any node of T_1 , in the order given by a preorder traversal of T_2 .

Example 2.22. For the transformation of the ordered tree $T_1 = (V_1, E_1)$ to the ordered tree $T_2 = (V_2, E_2)$ of Fig. 2.1, there are $8^5 = 32,768$ possible assignments of the $8 = 7 + 1$ nodes of $V_2 \cup \{\lambda\}$ to the five nodes of V_1 . However, only 9,276 of them correspond to a bijection

of V_1 to V_2 . Further, 98 of these bijections map nodes of the same depth, 20 of them also preserve the parent-child relation, and only 12 of them, shown in Fig. 2.6, also preserve sibling order and thus represent a valid transformation of T_1 to T_2 . These valid transformations are further illustrated in Fig. 2.7.

Backtracking is a general technique for organizing the exhaustive search for a solution to a combinatorial problem. Roughly, the technique consists of repeatedly extending a partial solution to the problem—represented as an assignment of values of a finite domain, satisfying certain constraints, to an ordered finite set of variables—to a complete solution to the problem, by extending the representation of the partial solution one variable at a time, and shrinking the representation of the partial solution (backtracking) whenever a partial solution cannot be further extended.

The backtracking technique can be applied to those problems that exhibit the *domino principle*, meaning that if a constraint is not satisfied by a given partial solution, the constraint will not be satisfied by any extension of the partial solution at all. The domino principle allows one to stop extending a partial solution as soon as it can be established that the extension will not lead to a solution to the problem, because the partial solution already violates some constraint.

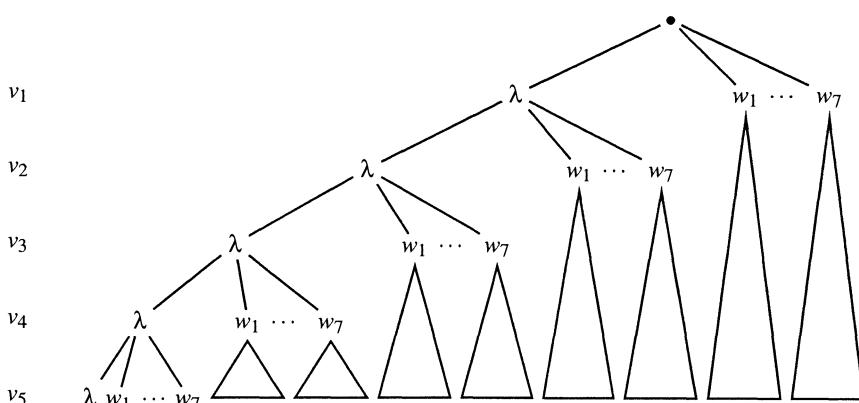


Fig. 2.8. Schematic view of the backtracking tree for the tree edit problem. Each of the values λ, w_1, \dots, w_7 can, in principle, be assigned to variables v_1, \dots, v_5 .

The procedure of extending a partial solution towards a complete solution to the problem, and shrinking a partial solution that cannot be further extended, can be better understood in terms of a *backtracking tree* of possible assignments of values to the variables that represent the problem. The root of the backtracking tree is a dummy node, the nodes at level one represent the possible values which the first variable can be assigned to, the nodes at level two represent the possible values which the second variable can be assigned to, given the value which the first variable was assigned to, the nodes at level three represent the possible values which the third variable can be assigned to, given the values which the first and second variables were assigned to, and so on. The backtracking tree for an instance of the tree edit problem is illustrated in Fig. 2.8.

The domino principle allows *pruning* of the backtracking tree at those nodes representing partial solutions which violate some constraint of the problem.

Finding a solution to a tree edit problem by backtracking means finding a node in the backtracking tree for which all the constraints of the problem are satisfied, and finding all solutions to the tree edit problem means finding all such nodes in the backtracking tree. These solution nodes will be leaves in the backtracking tree for a tree edit problem, because of the problem formulation, but this is not always the case.

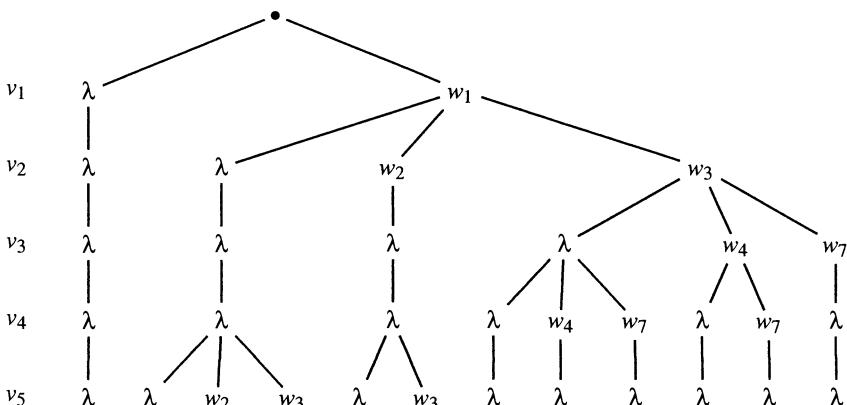


Fig. 2.9. Backtracking tree for the tree edit problem.

Example 2.23. The backtracking tree for the transformation between the ordered trees of Fig. 2.1 is shown in Fig. 2.9. A path from the root to a leaf in the backtracking tree corresponds to a valid transformation of T_1 to T_2 . Pruned subtrees are not shown, for clarity.

Notice that the backtracking tree for a combinatorial problem is not explicit but remains implicit in the formulation of the problem. Further, the size of the backtracking tree is often exponential in the number of variables and possible values. Therefore, it is crucial for a backtracking procedure to build only that portion of the backtracking tree that is needed at each stage, during the search for one or more solutions to the problem.

Now, the following backtracking algorithm for enumerating node assignments that correspond to valid tree transformations, extends an (initially empty) mapping M in all possible ways, in order to enumerate all assignments $M \subseteq V_1 \times V_2 \cup \{\lambda\}$ of nodes of an ordered tree $T_2 = (V_2, E_2)$, including a dummy node λ , to the nodes of an ordered tree $T_1 = (V_1, E_1)$. The node assignments found are collected in a list L of arrays of nodes (of tree T_2) indexed by the nodes of tree T_1 .

Further, the preorder number num and depth $depth$ of the nodes in the trees will be used for testing the constraints at each stage, and the candidate nodes which each node can be mapped to are represented by an array C of lists of nodes (of T_2) indexed by the nodes of T_1 .

75

\langle techniques 66 $\rangle + \equiv$

```
void backtracking_tree_edit(
    tree& T1,
    tree& T2,
    node_array<node>& M,
    list<node_array<node> >& L)
{
    L.clear();
    rearrange_tree_in_preorder(T1);
    rearrange_tree_in_preorder(T2);
    node_array<int> num1(T1), num2(T2), depth1(T1), depth2(T2);
    (set up candidate nodes 76b)
    node v = T1.first_node();
    extend_backtracking_tree_edit(T1,T2,num1,num2,depth1,depth2,C,v,M,L);
}
```

The actual extension of a partial solution is done by the following recursive procedure, which extends a node assignment $M \subseteq V_1 \times V_2 \cup \{\lambda\}$ by mapping node $v \in V_1$ to each node $w \in C[v] \subseteq V_2 \cup \{\lambda\}$ in

turn, that is, to each remaining candidate node, making a recursive call upon the successor (in preorder) of node $v \in V_1$ whenever $M \cup \{(v, w)\}$ is a node assignment corresponding to a valid transformation of the subtree of T_1 induced by the nodes up to, and including, node v into the subtree of T_2 induced by those nodes of T_2 which were already mapped to nodes of T_1 .

All such node assignments M which are defined for all nodes of T_1 , that is, all node assignments that correspond to a valid transformation of T_1 to T_2 , are collected in a list L of node assignments. Notice that the actual backtracking is hidden in the mechanism used by the programming language to implement the recursion.

76a

```
<techniques 66>+≡
void extend_backtracking_tree_edit(
    const tree& T1,
    const tree& T2,
    const node_array<int>& num1,
    const node_array<int>& num2,
    const node_array<int>& depth1,
    const node_array<int>& depth2,
    node_array<list<node> >& C,
    node v,
    node_array<node>& M,
    list<node_array<node> >& L)
{
    node w,x,y;
    forall(w,C[v]) {
        M[v] = w;
        <refine candidate nodes 77a>
        if ( v ≡ T1.last_node() ) {
            double_check_tree_edit(T1,T2,M);
            L.append(M);
        } else {
            extend_backtracking_tree_edit
                (T1,T2,num1,num2,depth1,depth2,N,T1.succ_node(v),M,L);
        } } }
```

The candidate nodes which a node of T_1 can be mapped to are just those nodes of T_2 of the same depth as the node of T_1 , together with the dummy node λ representing the deletion of the node from T_1 . That is, $C[v] = \{w \in V_2 \mid \text{depth}[v] = \text{depth}[w]\} \cup \{\lambda\}$. The dummy node λ is represented by the *nil* value.

76b

```
<set up candidate nodes 76b>≡
node_array<list<node> > C(T1);
{ preorder_tree_depth(T1,num1,depth1);
  preorder_tree_depth(T2,num2,depth2);
```

```

node v,w;
forall_nodes(v,T1) {
  C[v].append(nil); // dummy node, representing deletion of node v
  forall_nodes(w,T2) {
    if (depth1[v] ≡ depth2[w] ) {
      C[v].append(w);
    } } } }
```

After having assigned a node $v \in V_1$ to a node $w \in V_2 \cup \{\lambda\}$, the candidate nodes which the remaining nodes of T_1 can still be mapped to are found by removing from a copy N of C , all those candidate nodes which violate some constraint.

77a ⟨refine candidate nodes 77a⟩≡
 $\text{node_array} < \text{list} < \text{node} > N(T1);$
 $N = C;$
 $\text{list_item } it;$
⟨ensure that tree edit mapping is a bijection 77b⟩
⟨ensure that tree edit mapping preserves parent-child relation 77c⟩
⟨ensure that tree edit mapping preserves sibling order 78a⟩

If node $v \in V_1$ was just assigned to a nondummy node $w \in V_2$, assigning another node of T_1 to node w would violate the constraint that the assignment be a bijection. All such occurrences of node $w \in V_2$ as a candidate which the remaining nodes of T_1 could be assigned to, are removed from N by the following procedure.

77b ⟨ensure that tree edit mapping is a bijection 77b⟩≡
if ($w \neq \text{nil}$) {
 forall_nodes(x,T1) {
 forall_items(it,N[x]) {
 $y = N[x].contents(it);$
if ($y \equiv w$) {
 $N[x].del(it);$
 } } } }

The following procedure removes from N those candidate nodes $y \in V_2$ for each child x of node $v \in V_1$, which are not children of node $w \in V_2$.

77c ⟨ensure that tree edit mapping preserves parent-child relation 77c⟩≡
forall_children(x,v) {
 forall_items(it,N[x]) {
 $y = N[x].contents(it);$
if ($y \neq \text{nil} \wedge T2.parent(y) \neq w$) {
 $N[x].del(it);$
 } } }

Finally, the following procedure removes from N those candidate nodes which violate a sibling order constraint, thus ensuring that no right sibling x of node $v \in V_1$ is mapped later to a left sibling y of node $w \in V_2$. Notice that for sibling nodes $v, x \in V_1$, node x is a right sibling of node v if and only if $\text{preorder}[v] < \text{preorder}[x]$. Further, for sibling nodes $w, y \in V_2$, node y is a left sibling of node w if and only if $\text{preorder}[y] < \text{preorder}[w]$.

78a $\langle\text{ensure that tree edit mapping preserves sibling order 78a}\rangle \equiv$

```

if (  $\neg T1.\text{is\_root}(v) \wedge w \neq \text{nil}$  ) {
    forall_children( $x, T1.\text{parent}(v)$ ) {
        if (  $\text{num1}[v] < \text{num1}[x]$  ) {
            forall_items( $it, N[x]$ ) {
                 $y = N[x].\text{contents}(it);$ 
                if (  $y \neq \text{nil} \wedge \text{num2}[w] > \text{num2}[y]$  ) {
                     $N[x].\text{del}(it);$ 
                }
            }
        }
    }
}

```

The following double-check of the tree edit, although being redundant, gives some reassurance of the correctness of the implementation. It verifies that M is a mapping of T_1 to T_2 , according to Definition 2.5.

78b $\langle\text{double-check tree edit 78b}\rangle \equiv$

```

void double_check_tree_edit(
    const tree&  $T1$ ,
    const tree&  $T2$ ,
    const node_array<node>&  $M$ )
{
    node_array<int>  $\text{num1}(T1);$ 
    node_array<int>  $\text{depth1}(T1);$ 
     $\text{preorder\_tree\_depth}(T1, \text{num1}, \text{depth1});$ 
    node_array<int>  $\text{num2}(T2);$ 
    node_array<int>  $\text{depth2}(T2);$ 
     $\text{preorder\_tree\_depth}(T2, \text{num2}, \text{depth2});$ 

    node  $v = T1.\text{root}();$ 
    node  $w = T2.\text{root}();$ 
    if (  $M[v] \neq \text{nil} \wedge M[v] \neq w$  )
        error_handler(1, "Wrong implementation of tree edit");

    forall_nodes( $v, T1$ ) {
        if (  $M[v] \neq \text{nil}$  ) {
            if (  $\text{depth1}[v] \neq \text{depth2}[M[v]]$  )
                error_handler(1, "Wrong implementation of tree edit");
            if (  $\neg T1.\text{is\_root}(v) \wedge T2.\text{parent}(M[v]) \neq M[T1.\text{parent}(v)]$  )
                error_handler(1, "Wrong implementation of tree edit");
            forall_nodes( $w, T1$ )
                if (  $M[w] \neq \text{nil} \wedge \text{num1}[v] < \text{num1}[w] \wedge \text{num2}[M[v]] \geq \text{num2}[M[w]]$  )
                    error_handler(1, "Wrong implementation of tree edit");
        }
    }
}

```

A few implementation details still need to be filled in, though. The preorder number and depth of all nodes in the trees is computed by the following procedure, during an iterative preorder traversal of the trees. See Sect. 3.1 for a detailed discussion of the procedure.

79a $\langle \text{techniques 66} \rangle + \equiv$

```

void preorder_tree_depth(
    const tree& T,
    node_array<int>& order,
    node_array<int>& depth)
{
    stack<node> S;
    node v,w;
    S.push(T.root());
    int num = 1;
    do {
        v = S.pop();
        order[v] = num++; // visit node v
        if ( T.is_root(v) ) {
            depth[v] = 0;
        } else {
            depth[v] = depth[T.parent(v)] + 1;
        }
        w = T.last_child(v);
        while ( w ≠ nil ) {
            S.push(w);
            w = T.previous_sibling(w);
        }
    } while ( ¬S.empty() );
}

```

2.2.1 Interactive Demonstration of Backtracking

The backtracking algorithm for the tree edit is integrated next in the interactive demonstration of graph algorithms. A simple checker for the tree edit that provides some visual reassurance consists of highlighting the nodes of the ordered trees T_1 and T_2 that belong in the mapping corresponding to the valid transformation of T_1 into T_2 . Non-highlighted nodes of T_1 are deleted from T_1 , and nonhighlighted nodes of T_2 are inserted into T_2 by the transformation.

79b $\langle \text{demo techniques 68} \rangle + \equiv$

```

void gw_backtracking_tree_edit(
    GraphWin& gw1)

```

```

{
graph& GI = gw1.get_graph();
tree TI(GI);

graph G2;
GraphWin gw2(G2,500,500,"Backtracking Tree Edit");
gw2.display();
gw2.message("Enter second tree. Press done when finished");
gw2.edit();
gw2.del_message();

tree T2(G2);

node_array<node> M(TI);
list<node_array<node>> L;
backtracking_tree_edit(TI,T2,M,L);

panel P;
make_proof_panel(P,
  string("There are %i valid tree transformations",L.length()),true);
if ( gw1.open_panel(P) ) { // proof button pressed
  forall(M,L) {
    gw1.save_all_attributes();
    gw2.save_all_attributes();
    node v;
    forall_nodes(v,TI) {
      if ( M[v] ≠ nil ) {
        gw1.set_color(v,blue);
        gw2.set_color(M[v],blue);
      }
    }
    panel Q;
    make_yes_no_panel(Q,"Continue",true);
    if ( gw1.open_panel(Q) ) { // no button pressed
      gw1.restore_all_attributes();
      break;
    }
    gw1.restore_all_attributes();
    gw2.restore_all_attributes();
  }
}
}

```

2.3 Branch-and-Bound

The backtracking technique can be used for finding either one solution or all solutions to a combinatorial problem. When a cost can be associated with each partial solution, a least-cost solution can be found in a more efficient way by a simple variation of backtracking, called *branch-and-bound*. The technique consists of remembering the

lowest-cost solution found at each stage of the backtracking search for a solution, and to use the cost of the lowest-cost solution found so far as a lower bound on the cost of a least-cost solution to the problem, in order to discard partial solutions as soon as it can be established that they will not improve on the lowest-cost solution found so far.

The branch-and-bound technique can be applied to those problems that exhibit an extension of the *domino principle*, meaning not only that if a constraint is not satisfied by a given partial solution, the constraint will not be satisfied by any extension of the partial solution at all—as in the case of backtracking—but also that the cost of any extension of a partial solution will be greater than or equal to the cost of the partial solution itself. Such an extended domino principle allows one to stop extending a partial solution as soon as it can be established that the extension will not lead to a solution to the problem, because the partial solution already violates some constraint, and also as soon as it can be established that the extension will not lead to a least-cost solution to the problem, because the cost of the partial solution already reaches or exceeds the cost of the lowest-cost solution found so far.

As in the case of backtracking, the procedure of extending a partial solution towards a least-cost solution to the problem, and shrinking a partial solution that cannot be further extended to a solution of lesser cost, can be better understood in terms of a *branch-and-bound tree* of possible assignments of values to the variables that represent the problem. The root of the branch-and-bound tree is a dummy node of zero cost, the nodes at level one represent the possible values which the first variable can be assigned to, the nodes at level two represent the possible values which the second variable can be assigned to, given the value which the first variable was assigned to, the nodes at level three represent the possible values which the third variable can be assigned to, given the values which the first and second variables were assigned to, and so on. Further, $\text{cost}[v] \geq \text{cost}[w]$ for all leaves $v \in V_1$ and all successor nodes w of node v in a preorder traversal of the branch-and-bound tree. That is, subtrees in the backtracking tree rooted at nodes of cost greater than the cost of a previous leaf node, are pruned off the branch-and-bound tree.

Consider, for instance, the tree edit problem. Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be ordered trees, and recall that a solution to the problem of finding a valid transformation of T_1 into T_2 is given by an assignment $M \subseteq V_1 \times V_2 \cup \{\lambda\}$ of nodes of T_2 to the nodes of T_1 , where the dummy node λ represents the deletion of a node from T_1 , and all nodes in $\{w \in V_2 \mid \exists v \in V_1, (v, w) \in M\}$ are inserted into T_2 .

Also let the cost of a (partial) solution $M \subseteq V_1 \times V_2 \cup \{\lambda\}$ be given by the number of nodes deleted from T_1 , that is, $\text{cost}[M] = |\{(v, w) \in M \mid w = \lambda\}|$. Then, finding a least-cost solution to a tree edit problem by branch-and-bound means finding a node of least cost in the branch-and-bound tree for which all the constraints of the problem are satisfied. Again, because of the problem formulation, a least-cost solution node will be a leaf in the branch-and-bound tree for a tree edit problem.

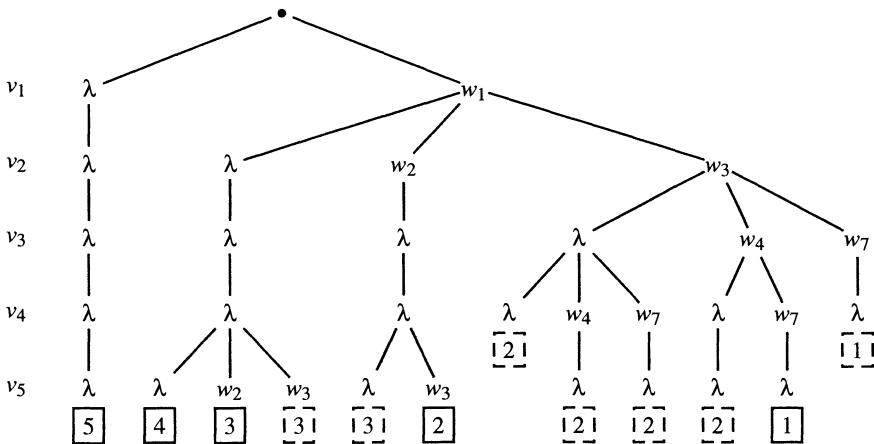


Fig. 2.10. Branch-and-bound tree for the tree edit problem.

Example 2.24. The branch-and-bound tree for the transformation between the ordered trees of Fig. 2.1 is shown in Fig. 2.10. Leaves of cost less than the cost of all predecessor leaves in preorder are distinguished by showing their cost in a framed box and the remaining leaves, of cost equal to the cost of their previous leaf in preorder, have their cost shown in a dashed box. The least-cost solution found is $\{(v_1, w_1), (v_2, w_3), (v_3, w_4), (v_4, w_7), (v_5, \lambda)\}$ and has cost 1.

Now, the following branch-and-bound algorithm for finding a node assignment that corresponds to a valid tree transformation of least cost, extends an (initially empty) mapping M in all possible ways, in order to find a least-cost assignment $M \subseteq V_1 \times V_2 \cup \{\lambda\}$ of nodes of an ordered tree $T_2 = (V_2, E_2)$, including a dummy node λ , to the nodes of an ordered tree $T_1 = (V_1, E_1)$. Both the node assignment M tried at each stage and the least-cost node assignment A found, are represented by an array of nodes (of tree T_2) indexed by the nodes of tree T_1 .

Further, as in the case of backtracking, the preorder number num and depth $depth$ of the nodes in the trees will be used for testing the constraints at each stage, and the candidate nodes which each node can be mapped to are represented by an array C of lists of nodes (of T_2) indexed by the nodes of T_1 .

83

```
(techniques 66) +≡
void branch_and_bound_tree_edit(
    tree& T1,
    tree& T2,
    node_array<node>& M)
{
    rearrange_tree_in_preorder(T1);
    rearrange_tree_in_preorder(T2);
    node_array<int> num1(T1), num2(T2), depth1(T1), depth2(T2);
    {set up candidate nodes 76b}
    node v = T1.first_node();
    int cost = 0;
    node_array<node> A;
    int lowcost = T1.number_of_nodes();
    extend_branch_and_bound_tree_edit
        (T1, T2, num1, num2, depth1, depth2, C, v, M, cost, A, lowcost);
    M = A; // return least-cost solution found
}
```

The actual extension of a partial solution is done by the following recursive procedure, which extends a node assignment $M \subseteq V_1 \times V_2 \cup \{\lambda\}$ by mapping node $v \in V_1$ to each node $w \in C[v] \subseteq V_2 \cup \{\lambda\}$ in turn, that is, to each remaining candidate node, making a recursive call upon the successor (in preorder) of node $v \in V_1$ whenever $M \cup \{(v, w)\}$ is a node assignment corresponding to a valid transformation of the subtree of T_1 induced by the nodes up to, and including, node v into the subtree of T_2 induced by those nodes of T_2 which were already mapped to nodes of T_1 and, furthermore, the cost $cost$ of $M \cup \{(v, w)\}$

is less than the cost $lowcost$ of the lowest-cost node assignment A found so far.

84a

```
<techniques 66>+≡
void extend_branch_and_bound_tree_edit(
    const tree& T1,
    const tree& T2,
    const node_array<int>& num1,
    const node_array<int>& num2,
    const node_array<int>& depth1,
    const node_array<int>& depth2,
    node_array<list<node>>& C,
    node v,
    node_array<node>& M,
    int cost,
    node_array<node>& A,
    int& lowcost)
{
    node w,x,y;
    forall(w,C[v]) {
        M[v] = w;
        <refine candidate nodes 77a>
        if (w ≡ nil) cost++; // deletion of node v
        if (cost < lowcost) { // lower-cost (partial) solution found
            if (v ≡ T1.last_node()) {
                double_check_tree_edit(T1,T2,M);
                A = M; // remember lowest-cost solution so far
                lowcost = cost;
            } else {
                extend_branch_and_bound_tree_edit
                    (T1,T2,num1,num2,depth1,depth2,N,T1.succ_node(v),M,cost,A,lowcost);
            }
            if (w ≡ nil) cost--; // deletion of node v
        }
    }
}
```

2.3.1 Interactive Demonstration of Branch-and-Bound

The branch-and-bound algorithm for the tree edit is integrated next in the interactive demonstration of graph algorithms. Again, a simple checker for the tree edit that provides some visual reassurance consists of highlighting the nodes of the ordered trees T_1 and T_2 that belong in the mapping corresponding to the least-cost transformation of T_1 into T_2 , where nonhighlighted nodes of T_1 are deleted from T_1 , and nonhighlighted nodes of T_2 are inserted into T_2 by the transformation. The number of node deletions and insertions made in the least-cost transformation found is also reported.

```

84b <demo techniques 68>+≡
    void gw_branch_and_bound_tree_edit(
        GraphWin& gw1)
    {
        graph& G1 = gw1.get_graph();
        tree T1(G1);

        graph G2;
        GraphWin gw2(G2,500,500,"Branch-and-Bound Tree Edit");
        gw2.display();
        gw2.message("Enter second tree. Press done when finished");
        gw2.edit();
        gw2.del_message();

        tree T2(G2);

        node_array<node> M(T1);
        branch_and_bound_tree_edit(T1,T2,M);

        gw1.save_all_attributes();
        int del = 0;
        int ins = T2.number_of_nodes();
        node v;
        forall_nodes(v,T1) {
            if ( M[v] ≡ nil ) {
                del++;
            } else {
                ins--;
                gw1.set_color(v,blue);
                gw2.set_color(M[v],blue);
            }
        }
        gw1.message(string("Least-cost transformation makes %i node deletions
            and %i node insertions",del,ins));
        gw1.wait();
        gw1.del_message();
        gw1.restore_all_attributes();
    }
}

```

2.4 Divide-and-Conquer

The backtracking technique can be used for finding either one solution or all solutions to a combinatorial problem, and the branch-and-bound technique can be used for finding, in a more efficient way, a least-cost solution. The *divide-and-conquer* technique can also be used to find a solution and, in particular, a least-cost solution to a combinatorial problem.

Divide-and-conquer is the general technique of solving a problem by dividing it into smaller, easier to solve subproblems, recursively solving these smaller subproblems, and then combining their solutions into a solution to the original problem. A subproblem whose size is small enough is just solved in a straightforward way. Further, the subproblem size must be a fraction of the problem size for the divide-and-conquer algorithm to be most efficient, as follows from the solution to the recurrence relations that describe the time and space complexity of divide-and-conquer algorithms. Dividing a problem into subproblems of about the same size as the original problem leads to less efficient algorithms, sometimes called *subtract-and-surrender* algorithms.

The divide-and-conquer technique can be applied to those problems that exhibit the *independence principle*, meaning that a problem instance can be divided into a series of smaller problem instances which are independent of each other.

Consider, for instance, the tree edit problem. Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be ordered trees, and recall that a solution to the problem of finding a valid transformation of T_1 into T_2 of least cost is given by an assignment $M \subseteq V_1 \times V_2 \cup \{\lambda\}$ of nodes of T_2 to the nodes of T_1 of least cost, where the dummy node λ represents the deletion of a node from T_1 , all nodes in $\{w \in V_2 \mid \exists v \in V_1, (v, w) \in M\}$ are inserted into T_2 , and the cost of a solution $M \subseteq V_1 \times V_2 \cup \{\lambda\}$ is given by the number of nodes deleted from T_1 , that is, $\text{cost}[M] = |\{(v, w) \in M \mid w = \lambda\}|$.

As a matter of fact, it is more convenient to state the least-cost tree transformation problem as a greatest-benefit problem, where the *benefit* of a least-cost solution $M \subseteq V_1 \times V_2 \cup \{\lambda\}$ is given by the number of nodes of T_1 substituted by nodes of T_2 , that is, $\text{benefit}[M] = |\{(v, w) \in M \mid w \neq \lambda\}|$. In the rest of this chapter least-cost and greatest-benefit tree transformations will not be distinguished.

Then, finding a least-cost solution to a tree edit problem by divide-and-conquer means dividing T_1 and T_2 into subtrees, recursively finding a least-cost transformation of each of the subtrees of T_1 to the corresponding subtree of T_2 , and then combining these transformations of subtrees into a least-cost transformation of T_1 to T_2 . The transformation of a trivial subtree of T_1 of the form $(\{v\}, \emptyset)$, that is, a subtree consisting of a single node, to a trivial subtree of T_2 of the form

$(\{w\}, \emptyset)$ is just $M = \{(v, w)\}$, that is, the substitution of node $w \in V_2$ for node $v \in V_1$.

The division of T_1 and T_2 into subtrees can be made on the basis of a node $v \in V_1$ and a node $w \in V_2$ of the same depth, as follows. Let A_1 be the subtree of T_1 containing all nodes from the root down to the predecessor of node v in preorder, let B_1 be the subtree of T_1 rooted at node v , let A_2 be the subtree of T_2 containing all nodes from the root down to the predecessor of node w in preorder, and let B_2 be the subtree of T_2 rooted at node w . Then, in a least-cost transformation of T_1 to T_2 , either node v is deleted from T_1 (and A_1 is transformed to T_1), node w is inserted into T_2 (and T_1 is transformed to A_2), or node v is substituted by node w (and A_1 is transformed to A_2 , and B_1 to B_2). Notice that the case of node v being substituted by a node of T_2 different from node w is implied by the insertion of node w into T_2 , and the case of a node of T_1 different from node v being substituted by node w is implied by the deletion of node v from T_1 .

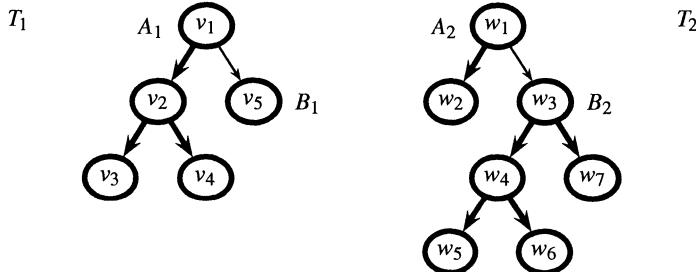


Fig. 2.11. A division of the ordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ of Fig. 2.1 upon nodes $v_5 \in V_1$ and $w_3 \in V_2$. Nodes are numbered according to the order in which they are visited during a preorder traversal of the trees.

Example 2.25. A division of ordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ of Fig. 2.1 is illustrated in Fig. 2.11. A least-cost transformation of T_1 to T_2 can be obtained by dividing tree T_1 upon node $v_5 \in V_1$ in subtrees A_1 and B_1 , dividing tree T_2 upon node $w_3 \in V_2$ in subtrees A_2 and B_2 , and then taking the transformation of least cost among a least-cost transformation of A_1 to T_2 , a least-cost transformation of T_1 to A_2 , and a least-cost transformation of A_1 to A_2 together with

a least-cost transformation of B_1 to B_2 . The least-cost transformation $M = \{(v_1, w_1), (v_2, w_3), (v_3, w_4), (v_4, w_7)\}$ of A_1 to T_2 gives a least-cost transformation $M \cup \{(v_5, \lambda)\}$ of T_1 to T_2 , of cost 1, corresponding to the deletion of node v_5 from T_1 (and the insertion of nodes w_2, w_5, w_6 into T_2).

Now, given the recursive nature of the divide-and-conquer algorithm for the tree edit problem, it is convenient to delimit the subtrees involved in each recursive call to the algorithm by giving the first and last nodes in preorder of each subtree, together with the whole trees, instead of making explicit copies of the subtrees. In this sense, a recursive call to the divide-and-conquer algorithm will involve the subtree of T_1 with nodes from v_1 to v_2 , and the subtree of T_2 with nodes from w_1 to w_2 . Several cases need to be distinguished.

- The particular case of $v_1 = v_2$ and $w_1 = w_2$ will correspond to the transformation $M = \{(v_1, w_1)\}$ of the trivial subtree $(\{v_1\}, \emptyset)$ of T_1 to the trivial subtree $(\{w_2\}, \emptyset)$ of T_2 , with benefit 1.
- The particular case of $v_1 \neq v_2$ and $w_1 = w_2$ will correspond to the transformation of the subtree A_1 of T_1 (the subtree of T_1 with nodes from v_1 down to the predecessor in preorder of node v_2) to tree T_2 , that is, the deletion of the subtree of T_1 rooted at node v_2 .
- The particular case of $v_1 = v_2$ and $w_1 \neq w_2$ will correspond to the transformation of tree T_1 to the subtree A_2 of T_2 (the subtree of T_2 with nodes from w_1 down to the predecessor in preorder of node w_2), that is, the insertion of the subtree of T_2 rooted at node w_2 .
- The general case of $v_1 \neq v_2$ and $w_1 \neq w_2$ will correspond to either the transformation of the subtree A_1 of T_1 to tree T_2 , the transformation of tree T_1 to the subtree A_2 of T_2 , or the transformation of the subtree A_1 of T_1 to the subtree A_2 of T_2 together with the transformation of the subtree B_1 of T_1 (the subtree of T_1 rooted at node v_2) to the subtree B_2 of T_2 (the subtree of T_2 rooted at node w_2).

Now, the following choice of nodes $v \in V_1$ and $w \in V_2$ upon which to make the division of the subtree of T_1 with nodes from v_1 down to v_2 , to the subtree of T_2 with nodes from w_1 down to w_2 , will be considered in the rest of this section. Node v will be the last child of node v_1 that belongs to the subtree of T_1 with nodes from v_1 down

to v_2 . In a similar way, node w will be the last child of node w_1 that belongs to the subtree of T_2 with nodes from w_1 down to w_2 .

The divide-and-conquer procedure for the tree edit problem can be better understood in terms of a *divide-and-conquer tree* of all recursive calls made for the solution to a tree edit problem instance. The root of the divide-and-conquer tree stands for the initial call to the recursive procedure (upon the whole tree T_1 and the whole tree T_2), and the remaining nodes represent recursive calls (upon subtrees A_1, B_1 of T_1 and subtrees A_2, B_2 of T_2) and have either none, one, or four children depending on the relative size of the subtrees. That is, the node corresponding to a recursive call upon the subtree of T_1 with nodes from v_1 down to v_2 and the subtree of T_2 with nodes from w_1 down to w_2 , will have no children if $A_1 = \emptyset$ and $B_1 = \emptyset$ (that is, if $v_1 = w_1$ and $w_1 = w_2$), will have one child if either $A_1 \neq \emptyset$ and $B_1 = \emptyset$ or $A_1 = \emptyset$ and $B_1 \neq \emptyset$, and will have four children if both $A_1 \neq \emptyset$ and $B_1 \neq \emptyset$.

Example 2.26. The divide-and-conquer tree for the transformation between ordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ of Fig. 2.1 is shown in Fig. 2.12. A node labeled $v_i v_j w_k w_\ell$ corresponds to a recursive call to the divide-and-conquer procedure upon the subtree of T_1 with nodes from v_i down to v_j and the subtree of T_2 with nodes from w_k down to w_ℓ .

The following divide-and-conquer algorithm for the tree edit problem makes an initial call to the recursive divide-and-conquer procedure upon the subtree of T_1 from the first node down to the last node in preorder and the subtree of T_2 from the first node down to the last node in preorder, that is, upon the whole tree T_1 and the whole tree T_2 , and returns the cost of a least-cost transformation of T_1 to T_2 , together with the corresponding mapping M .

The representation of the trees is rearranged according to their preorder traversal, as required by the formulation of the tree edit problem, and the preorder number of all nodes in the trees is stored in integer arrays *order1* and *order2* indexed, respectively, by the nodes of T_1 and T_2 .

```
89 <techniques 66> +≡
int divide_and_conquer_tree_edit(
    tree& T1,
    tree& T2,
```

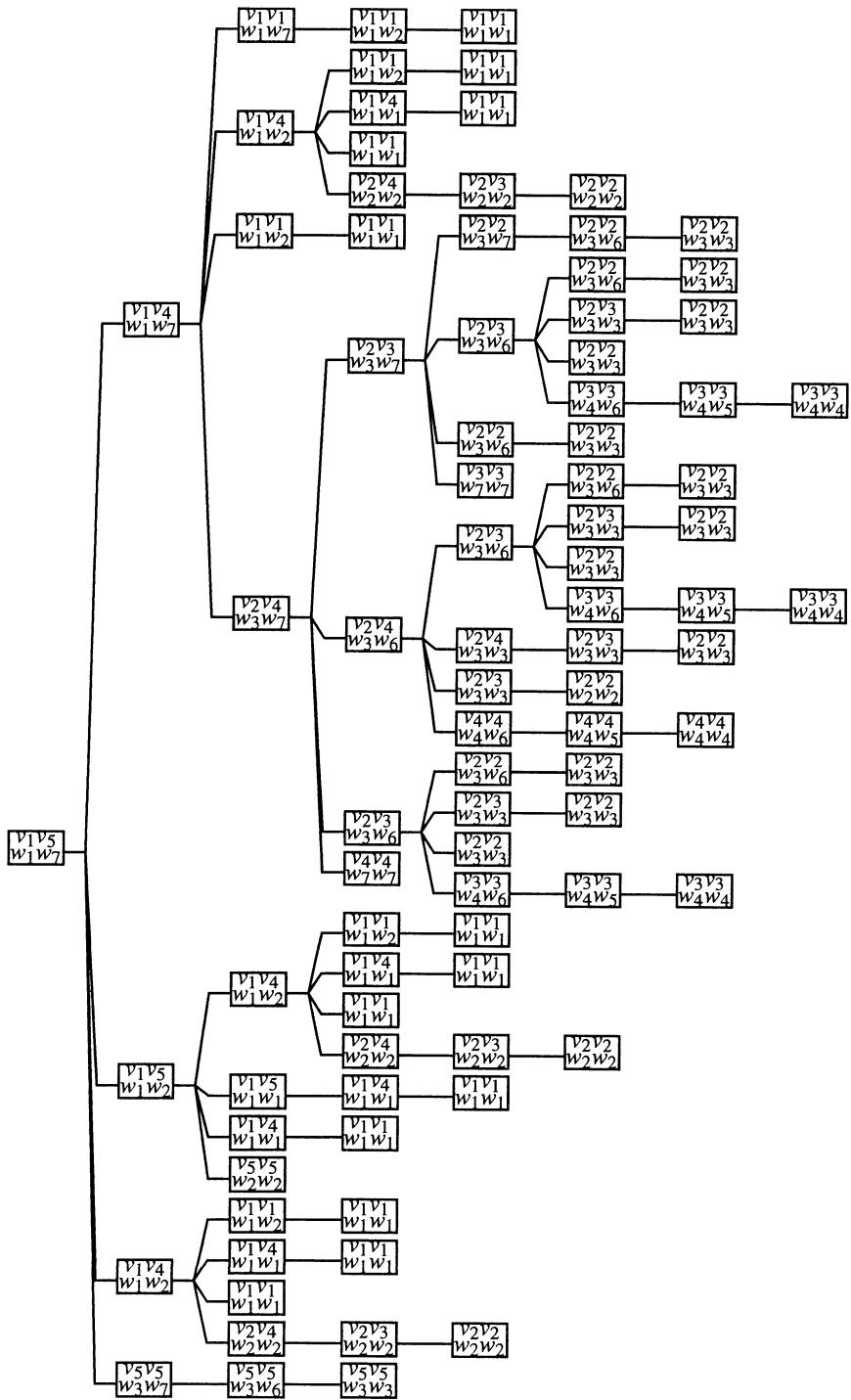


Fig. 2.12. Divide-and-conquer tree for the tree edit problem.

```

node_array<node>& M)
{
    rearrange_tree_in_preorder(T1);
    rearrange_tree_in_preorder(T2);
    node v,w;
    node_array<int> order1(T1);
    node_array<int> order2(T2);
    int i = 1;
    forall_nodes(v,T1)
        order1[v] = i++;
    int j = 1;
    forall_nodes(w,T2)
        order2[w] = j++;

    return divide_and_conquer_tree_edit(T1,order1,
        T1.first_node(),T1.last_node(),T2,order2,T2.first_node(),T2.last_node(),M);
}

```

The actual computation of a least-cost transformation M of T_1 to T_2 is done by the following recursive procedure which, in the general case, takes the predecessor k_1 in preorder of the last child of node v_1 , takes the predecessor k_2 in preorder of the last child of node w_1 , divides the subtree of T_1 with nodes from v_1 down to v_2 , into a subtree A_1 with nodes from v_1 down to k_1 and a subtree B_1 with nodes from the successor of node k_1 in preorder down to v_2 , and also divides the subtree of T_2 with nodes from w_1 down to w_2 , into a subtree A_2 with nodes from w_1 down to k_2 and a subtree B_2 with nodes from the successor of node k_2 in preorder down to w_2 .

91

```

⟨techniques 66⟩+≡
int divide_and_conquer_tree_edit(
    const tree& T1,
    const node_array<int>& order1,
    node v1,
    node v2,
    const tree& T2,
    const node_array<int>& order2,
    node w1,
    node w2,
    node_array<node>& M)
{
    int dist,del,ins,pre,pos;
    node k1,k2,l1,l2;

    if( v1 ≡ v2 ) {
        if( w1 ≡ w2 ) { // substitution of w1 for v1
            M[v1] = w1;
            dist = 1;
        }
        else {
            M[v1] = w1;
            dist = 1;
        }
    }
    else {
        if( v1 < v2 ) {
            if( w1 < w2 ) {
                M[v1] = w1;
                dist = 1;
            }
            else {
                M[v1] = w1;
                dist = 1;
            }
        }
        else {
            if( w1 < w2 ) {
                M[v1] = w1;
                dist = 1;
            }
            else {
                M[v1] = w1;
                dist = 1;
            }
        }
    }
}
```

```

} else { // insertion of k2

    k2 = predecessor_of_last_child(T2,order2,w1,w2);
    dist = divide_and_conquer_tree_edit(T1,order1,v1,v2,T2,order2,w1,k2,M);

}

} else {
    if ( w1 ≡ w2 ) { // deletion of k1

        k1 = predecessor_of_last_child(T1,order1,v1,v2);
        dist = divide_and_conquer_tree_edit(T1,order1,v1,k1,T2,order2,w1,w2,M);

    } else { // substitution of k2 for k1

        k1 = predecessor_of_last_child(T1,order1,v1,v2);
        node_array<node> M1(T1); M1 = M;
        del = divide_and_conquer_tree_edit(T1,order1,v1,k1,T2,order2,w1,w2,M1);

        k2 = predecessor_of_last_child(T2,order2,w1,w2);
        node_array<node> M2(T1); M2 = M;
        ins = divide_and_conquer_tree_edit(T1,order1,v1,v2,T2,order2,w1,k2,M2);

        node_array<node> M3(T1); M3 = M;
        pre = divide_and_conquer_tree_edit(T1,order1,v1,k1,T2,order2,w1,k2,M3);
        pos = divide_and_conquer_tree_edit(T1,order1,
            T1.succ_node(k1),v2,T2,order2,T2.succ_node(k2),w2,M3);

        dist = leda_max(leda_max(del,ins),pre+pos);
        if ( dist ≡ del ) M = M1;
        else if ( dist ≡ ins ) M = M2;
        else M = M3;
    }
}

return dist;
}

```

A few implementation details still need to be filled in. In the subtree of an ordered tree $T = (V, E)$ with nodes from $v_1 \in V$ down to $v_2 \in V$, where node v_1 is either equal to node v_2 or it is a predecessor of v_2 in preorder, the predecessor $k_1 \in V$ in preorder of the last child of node v_1 is, as a matter of fact, the predecessor in preorder of the last child of node v_1 that lies between nodes v_1 and v_2 , for otherwise node k_1 would not belong to the subtree of T_1 with nodes from v_1 down to v_2 .

92 ⟨techniques 66⟩ +≡
 $\text{node predecessor_of_last_child}$
 $\text{const tree\& } T,$
 $\text{const node_array<int>\& } order,$

```

node v1,
node v2)
{
    node k = T.last_child(v1);
    while ( order[k] > order[v2] )
        k = T.previous_sibling(k);
    return T.pred_node(k);
}

```

2.4.1 Interactive Demonstration of Divide-and-Conquer

The divide-and-conquer algorithm for the tree edit is integrated next in the interactive demonstration of graph algorithms. Again, a simple checker for the tree edit that provides some visual reassurance consists of highlighting the nodes of the ordered trees T_1 and T_2 that belong in the mapping corresponding to the least-cost transformation of T_1 into T_2 , where nonhighlighted nodes of T_1 are deleted from T_1 , and nonhighlighted nodes of T_2 are inserted into T_2 by the transformation. The number of node deletions and insertions made in the least-cost transformation found is also reported.

93

```

⟨demo techniques 68⟩+≡
void gw_divide_and_conquer_tree_edit(
    GraphWin& gw1)
{
    graph& G1 = gw1.get_graph();
    tree T1(G1);

    graph G2;
    GraphWin gw2(G2,500,500,"Divide-and-Conquer Tree Edit");
    gw2.display();
    gw2.message("Enter second tree. Press done when finished");
    gw2.edit();
    gw2.del_message();

    tree T2(G2);

    node_array<node> M(T1);
    int dist = divide_and_conquer_tree_edit(T1,T2,M);

    gw1.save_all_attributes();
    int del = 0;
    int ins = T2.number_of_nodes();
    node v;
    forall_nodes(v,T1) {
        if ( M[v] ≡ nil ) {
            del++;
        }
    }
}
```

```

} else {
    ins--;
    gw1.set_color(v,blue);
    gw2.set_color(M[v],blue);
}
gw1.message(string("Least-cost transformation makes %i node deletions
    and %i node insertions",del,ins));
gw1.wait();
gw1.del_message();
gw1.restore_all_attributes();
}

```

2.5 Dynamic Programming

The divide-and-conquer technique can be used for finding a solution to combinatorial problems that exhibit the *independence principle*, where a problem instance can be divided into a series of smaller problem instances which are independent of each other. When applied to finding a least-cost solution, the independence principle is often called the *optimality principle*, meaning that a least-cost solution to a problem instance can be decomposed into a series of least-cost solutions to smaller problem instances which are independent of each other.

It is often the case, however, that these independent smaller problem instances are overlapping, and a divide-and-conquer algorithm will spend unnecessary effort in recomputing a least-cost solution to overlapping problem instances. For instance, in the divide-and-conquer tree for the tree edit problem instance given in Example 2.12, there are 88 independent smaller problem instances solved for finding a least-cost transformation of tree T_1 to tree T_2 , of which 32 are solved exactly once, and the remaining 56 independent smaller problem instances are overlapping. In particular, a least-cost transformation of the subtree of T_1 with nodes from v_2 down to v_3 to the subtree of T_2 with nodes from w_3 down to w_6 is solved three times, during the search of a least-cost transformation of the subtree of T_1 with nodes from v_2 down to v_4 to the subtree of T_2 with nodes from w_3 down to w_7 .

The dynamic programming technique improves on the divide-and-conquer technique for optimization problems that exhibit the optimality principle by storing solutions to subproblems, thus avoiding

the recomputation of solutions to overlapping subproblems—smaller problems which are common to larger independent problems—during the search for an optimal solution to the problem. Since solutions to common smaller problems are shared among instances of larger problems, the dynamic programming tree corresponding to the decomposition of an optimization problem in smaller problems is, in fact, a directed acyclic graph and is smaller than the corresponding divide-and-conquer tree, as illustrated in Fig. 2.13.

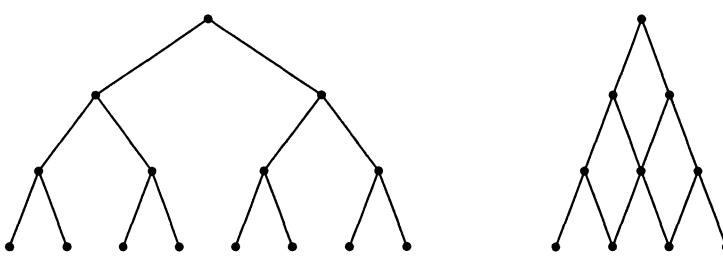


Fig. 2.13. Divide-and-conquer tree (left) and dynamic programming directed acyclic graph (right) for the decomposition of an optimization problem into smaller problems.

The very name of *dynamic programming* does not refer to a programming methodology but to a tabular method, by which optimal solutions to smaller problems are computed only once and stored in a dictionary for later lookup, whenever the smaller problem is encountered again during the search for an optimal solution to the original problem.

Essentially, there are two ways in which such a tabular method can be implemented. In top-down dynamic programming, also known as *memoization*, a recursive, divide-and-conquer algorithm for an optimization problem is enhanced by first looking up whether a smaller problem was already solved (and the solution to the smaller problem is already available in a dictionary), and solving it by divide-and-conquer otherwise. In bottom-up dynamic programming, by contrast, computation proceeds in a bottom-up fashion and optimal solutions to smaller problems are thus readily available, when solving a larger problem.

Consider, for instance, the tree edit problem. Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be ordered trees, and recall that a solution to the problem

of finding a valid transformation of T_1 into T_2 of least cost is given by an assignment $M \subseteq V_1 \times V_2 \cup \{\lambda\}$ of nodes of T_2 to the nodes of T_1 of least cost, where the dummy node λ represents the deletion of a node from T_1 , all nodes in $\{w \in V_2 \mid \exists v \in V_1, (v, w) \in M\}$ are inserted into T_2 , and the cost of a solution $M \subseteq V_1 \times V_2 \cup \{\lambda\}$ is given by the number of nodes deleted from T_1 , that is, $\text{cost}[M] = |\{(v, w) \in M \mid w = \lambda\}|$.

Again, it is more convenient to state the least-cost tree transformation problem as a greatest-benefit problem, where the *benefit* of a least-cost solution $M \subseteq V_1 \times V_2 \cup \{\lambda\}$ is given by the number of nodes of T_1 substituted by nodes of T_2 , that is, $\text{benefit}[M] = |\{(v, w) \in M \mid w \neq \lambda\}|$.

In a top-down dynamic programming algorithm based on the divide-and-conquer algorithm given in Sect. 2.4 for finding a least-cost transformation between ordered trees T_1 and T_2 , a least-cost transformation of the subtree of T_1 with nodes from v_i down to v_j to the subtree of T_2 with nodes from w_j down to w_ℓ can be memoized with the help of a dictionary, indexed by two nodes of T_1 and two nodes of T_2 , of two-tuples consisting of an integer edit distance or cost, and an array M of nodes (of tree T_1) indexed by the nodes of tree T_2 .

T_1	T_2		M						
v_i	v_j	w_k	w_ℓ	dist	v_1	v_2	v_3	v_4	v_5
v_1	v_1	w_1	w_1	1	w_1	λ	λ	λ	λ
v_1	v_1	w_1	w_2	1	w_1	λ	λ	λ	λ
v_1	v_1	w_1	w_7	1	w_1	λ	λ	λ	λ
v_1	v_4	w_1	w_1	1	w_1	λ	λ	λ	λ
v_2	v_2	w_2	w_2	1	w_1	w_2	λ	λ	λ
v_2	v_3	w_2	w_2	1	w_1	w_2	λ	λ	λ
v_2	v_4	w_2	w_2	1	w_1	w_2	λ	λ	λ
v_1	v_4	w_1	w_2	2	w_1	w_2	λ	λ	λ
v_2	v_2	w_3	w_3	1	w_1	w_3	λ	λ	λ
v_2	v_2	w_3	w_6	1	w_1	w_3	λ	λ	λ
v_2	v_2	w_3	w_7	1	w_1	w_3	λ	λ	λ
v_2	v_3	w_3	w_3	1	w_1	w_3	λ	λ	λ
v_3	v_3	w_4	w_4	1	w_1	w_3	w_4	λ	λ
v_3	v_3	w_4	w_5	1	w_1	w_3	w_4	λ	λ
v_3	v_3	w_4	w_6	1	w_1	w_3	w_4	λ	λ
v_2	v_3	w_3	w_6	2	w_1	w_3	w_4	λ	λ
v_3	v_3	w_7	w_7	1	w_1	w_3	w_7	λ	λ

T_1	T_2		M						
v_i	v_j	w_k	w_ℓ	dist	v_1	v_2	v_3	v_4	v_5
v_2	v_3	w_3	w_7	2	w_1	w_3	w_4	λ	λ
v_2	v_4	w_3	w_3	1	w_1	w_3	λ	λ	λ
v_4	v_4	w_4	w_4	1	w_1	w_3	λ	w_4	λ
v_4	v_4	w_4	w_5	1	w_1	w_3	λ	w_4	λ
v_4	v_4	w_4	w_6	1	w_1	w_3	λ	w_4	λ
v_2	v_4	w_3	w_6	2	w_1	w_3	w_4	λ	λ
v_4	v_4	w_7	w_7	1	w_1	w_3	w_4	w_7	λ
v_2	v_4	w_3	w_7	3	w_1	w_3	w_4	w_7	λ
v_1	v_4	w_1	w_7	4	w_1	w_3	w_4	w_7	λ
v_1	v_5	w_1	w_1	1	w_1	λ	λ	λ	λ
v_5	v_5	w_2	w_2	1	w_1	λ	λ	λ	w_2
v_1	v_5	w_1	w_2	2	w_1	w_2	λ	λ	λ
v_5	v_5	w_3	w_3	1	w_1	w_2	λ	λ	w_3
v_5	v_5	w_3	w_6	1	w_1	w_2	λ	λ	w_3
v_5	v_5	w_3	w_7	1	w_1	w_2	λ	λ	w_3
v_1	v_5	w_1	w_7	4	w_1	w_3	w_4	w_7	λ

Fig. 2.14. Memoization of optimal solutions to smaller tree edit distance problems in top-down dynamic programming.

Example 2.27. The execution of the top-down dynamic programming procedure for finding a least-cost transformation between ordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ of Fig. 2.1 is illustrated in Fig. 2.14. The benefit $dist$ and the node assignment $M \subseteq V_1 \times V_2 \cup \{\lambda\}$ corresponding to a least-cost transformation of the subtree of T_1 with nodes from v_i down to v_j to the subtree of T_2 with nodes from w_k down to w_ℓ is shown for each smaller problem solved, in the order in which solutions are completed.

The following top-down dynamic programming algorithm for the tree edit problem makes an initial call to the recursive top-down dynamic programming procedure upon the subtree of T_1 from the first node down to the last node in preorder and the subtree of T_2 from the first node down to the last node in preorder, that is, upon the whole tree T_1 and the whole tree T_2 , and returns the cost of a least-cost transformation of T_1 to T_2 , together with the corresponding mapping M . Again, the representation of the trees is rearranged according to their preorder traversal, as required by the formulation of the tree edit problem, and the preorder number of all nodes in the trees is stored in integer arrays $order1$ and $order2$ indexed, respectively, by the nodes of T_1 and T_2 .

97

```
(techniques 66) +≡
int top_down_dynamic_programming_tree_edit(
    tree& T1,
    tree& T2,
    node_array<node>& M)
{
    rearrange_tree_in_preorder(T1);
    rearrange_tree_in_preorder(T2);
    node v,w;
    node_array<int> order1(T1);
    node_array<int> order2(T2);
    int i = 1;
    forall_nodes(v,T1)
        order1[v] = i++;
    int j = 1;
    forall_nodes(w,T2)
        order2[w] = j++;
    h_array<four_tuple<node,node,node,node>,
    two_tuple<int,node_array<node> >> S;
    return top_down_dyn_prog_tree_edit(T1,order1,T1.first_node(),T1.last_node(),
        T2,order2,T2.first_node(),T2.last_node(),S,M);
}
```

The actual computation of a least-cost transformation M of T_1 to T_2 is done by the following recursive procedure which, in the general case, takes the predecessor k_1 in preorder of the last child of node v_1 , takes the predecessor k_2 in preorder of the last child of node w_1 , divides the subtree of T_1 with nodes from v_1 down to v_2 , into a subtree A_1 with nodes from v_1 down to k_1 and a subtree B_1 with nodes from the successor of node k_1 in preorder down to v_2 , and also divides the subtree of T_2 with nodes from w_1 down to w_2 , into a subtree A_2 with nodes from w_1 down to k_2 and a subtree B_2 with nodes from the successor of node k_2 in preorder down to w_2 .

The procedure is identical to the divide-and-conquer procedure given in Sect. 2.4, except that in a recursive call upon the subtree of T_1 with nodes from v_i down to v_j and the subtree of T_2 with nodes from w_k down to w_ℓ , the four-tuple $\langle v_i, v_j, w_k, w_\ell \rangle$ is first looked up in dictionary S , which is an additional parameter of the recursive procedure, and the corresponding tree edit problem is decomposed only if no such four-tuple is defined in the dictionary. Further, each solution to a smaller problem found is inserted in S . The dictionary is implemented by a hashing table.

98

```
<techniques 66>+≡
int top_down_dyn_prog_tree_edit(
    const tree& T1,
    const node_array<int>& order1,
    node v1,
    node v2,
    const tree& T2,
    const node_array<int>& order2,
    node w1,
    node w2,
    h_array<four_tuple<node,node,node,node>,
        two_tuple<int,node_array<node> >>& S,
    node_array<node>& M)
{
    int dist;
    four_tuple<node,node,node,node> N(v1,v2,w1,w2);
    if (S.defined(N)) { // problem v1,v2,w1,w2 was already solved
        dist = S[N].first();
        M = S[N].second();
    } else { // solve problem v1,v2,w1,w2
        int del,ins,pre,pos;
        node k1,k2,l1,l2;
```

```

if (  $v_1 \equiv v_2$  ) {
  if (  $w_1 \equiv w_2$  ) // substitution of  $w_1$  for  $v_1$ 

     $M[v_1] = w_1;$ 
     $dist = 1;$ 

  } else { // insertion of  $k_2$ 

     $k_2 = predecessor\_of\_last\_child(T_2, order_2, w_1, w_2);$ 
     $dist = top\_down\_dyn\_prog\_tree\_edit(T_1, order_1, v_1, v_2,$ 
     $T_2, order_2, w_1, k_2, S, M);$ 
  }
} else {
  if (  $w_1 \equiv w_2$  ) // deletion of  $k_1$ 

     $k_1 = predecessor\_of\_last\_child(T_1, order_1, v_1, v_2);$ 
     $dist = top\_down\_dyn\_prog\_tree\_edit(T_1, order_1, v_1, k_1,$ 
     $T_2, order_2, w_1, w_2, S, M);$ 

  } else { // substitution of  $k_2$  for  $k_1$ 

     $k_1 = predecessor\_of\_last\_child(T_1, order_1, v_1, v_2);$ 
     $node\_array<node> M1(T_1); M1 = M;$ 
     $del = top\_down\_dyn\_prog\_tree\_edit(T_1, order_1, v_1, k_1,$ 
     $T_2, order_2, w_1, w_2, S, M1);$ 

     $k_2 = predecessor\_of\_last\_child(T_2, order_2, w_1, w_2);$ 
     $node\_array<node> M2(T_1); M2 = M;$ 
     $ins = top\_down\_dyn\_prog\_tree\_edit(T_1, order_1, v_1, v_2,$ 
     $T_2, order_2, w_1, k_2, S, M2);$ 

     $node\_array<node> M3(T_1); M3 = M;$ 
     $pre = top\_down\_dyn\_prog\_tree\_edit(T_1, order_1, v_1, k_1,$ 
     $T_2, order_2, w_1, k_2, S, M3);$ 
     $pos = top\_down\_dyn\_prog\_tree\_edit(T_1, order_1, T_1.succ\_node(k_1), v_2,$ 
     $T_2, order_2, T_2.succ\_node(k_2), w_2, S, M3);$ 

     $dist = leda\_max(leda\_max(del, ins), pre + pos);$ 
    if (  $dist \equiv del$  )  $M = M1;$ 
    else if (  $dist \equiv ins$  )  $M = M2;$ 
    else  $M = M3;$ 
  }
}

two_tuple<int, node_array<node>> R(dist, M);
 $S[N] = R;$  // save solution to problem  $v_1, v_2, w_1, w_2$  for later lookup
}

return dist;
}

```

Consider now a bottom-up dynamic programming algorithm for the tree edit problem. A bottom-up order of computation ensures that

optimal solutions to smaller problems are computed before an optimal solution to a larger problem is computed. For the tree edit problem, a natural way to achieve such a bottom-up order of computation consists in finding a least-cost transformation of the subgraph of T_1 with nodes from v_i down to v_j to the subgraph of T_2 with nodes from w_k down to w_ℓ , where $1 \leq i \leq j \leq n_1$ and $1 \leq k \leq \ell \leq n_2$, for decreasing values of i and k starting off with $i = j = n_1$ and $k = \ell = n_2$. However, most of these least-cost transformations are not really needed, and their computation can be avoided as follows.

Recall that the division of T_1 and T_2 into subtrees is made on the basis of a node $v \in V_1$ and a node $w \in V_2$ of the same depth. Therefore, only solutions to the smaller problems of finding a least-cost transformation of the subgraph of T_1 with nodes from v_i down to v_j to the subgraph of T_2 with nodes from w_k down to w_ℓ , with $\text{depth}[v_i] = \text{depth}[w_k]$, need to be computed. Notice that, unlike the top-down dynamic programming algorithm for the tree edit problem, smaller problems are now subgraphs, not necessarily subtrees, of T_1 and T_2 .

Then, a least-cost transformation of the subtree of T_1 rooted at node $v \in V_1$ to the subtree of T_2 rooted at node $w \in V_2$, with $\text{depth}[v] = \text{depth}[w]$, can be found by taking the transformation of least cost among

$$\begin{array}{cccc} \langle v_i, w_k \rangle & \langle v_i, w_k w_{k+1} \rangle & \cdots & \langle v_i, w_k \dots w_\ell \rangle \\ \langle v_i v_{i+1}, w_k \rangle & \langle v_i v_{i+1}, w_k w_{k+1} \rangle & \cdots & \langle v_i v_{i+1}, w_k \dots w_\ell \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle v_i \dots v_j, w_k \rangle & \langle v_i \dots v_j, w_k w_{k+1} \rangle & \cdots & \langle v_i \dots v_j, w_k \dots w_\ell \rangle \end{array}$$

where v_i, \dots, v_j are the children of node v , w_k, \dots, w_ℓ are the children of node w , and $\langle v_i \dots v_j, w_k \dots w_\ell \rangle$ denotes a least-cost transformation of the subgraph of T_1 consisting of the subtrees rooted at nodes v_i, \dots, v_j , into the subgraph of T_2 consisting of the subtrees rooted at nodes w_k, \dots, w_ℓ . The bottom-up order of computation of a least-cost transformation of the subtree of T_1 rooted at node v into the subtree of T_2 rooted at node w , corresponds to computing the entries of the previous matrix of smaller problems in row order. Further, a least-cost transformation of the subtree of T_1 rooted at node v_1 to the subtree of T_2 rooted at node w_1 gives a least-cost transformation of T_1 into T_2 .

Again, it is more convenient to state the least-cost tree transformation problem as a greatest-benefit problem, where the *benefit* of a least-cost transformation is given by the number of node substitutions in the transformation.

v	w	T_1		T_2		$dist$	M				
		v_i	v_j	w_k	w_ℓ		v_1	v_2	v_3	v_4	v_5
v_2	w_2	v_2	v_4	w_2	w_2	1	λ	w_2	λ	λ	λ
v_3	w_4	v_3	v_3	w_4	w_6	1	λ	λ	w_4	λ	λ
v_3	w_7	v_3	v_3	w_4	w_7	1	λ	λ	w_4	λ	λ
v_4	w_4	v_3	v_4	w_4	w_6	1	λ	λ	w_4	λ	λ
v_4	w_7	v_3	v_4	w_4	w_7	2	λ	λ	w_4	w_7	λ
v_2	w_3	v_2	v_4	w_2	w_7	3	λ	w_3	w_4	w_7	λ
v_5	w_2	v_2	v_5	w_2	w_2	1	λ	w_2	λ	λ	λ
v_5	w_3	v_2	v_5	w_2	w_7	3	λ	w_3	w_4	w_7	λ
v_1	w_1	v_1	v_5	w_1	w_7	4	w_1	w_3	w_4	w_7	λ

v_2	w_2	w_3
0	0	0
0	1	3
0	1	3

v_3	w_4	w_7
0	0	0
0	1	1
0	1	2

Fig. 2.15. Finding optimal solutions to smaller tree edit distance problems in bottom-up dynamic programming.

Example 2.28. The execution of the bottom-up dynamic programming procedure for finding a least-cost transformation between ordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ of Fig. 2.1 is illustrated in Fig. 2.15. The benefit $dist$ and the node assignment $M \subseteq V_1 \times V_2 \cup \{\lambda\}$ corresponding to a least-cost transformation of the subgraph of T_1 with nodes from v_i down to v_j to the subgraph of T_2 with nodes from w_k down to w_ℓ is shown for each smaller problem solved, in the order in which solutions are completed.

Further, the matrices of smaller problems are also shown where, for instance, a least-cost transformation of the subgraph of T_1 consisting of the subtrees rooted at nodes v_2, v_5 , into the subgraph of T_2 consisting of the subtrees rooted at nodes w_2, w_3 has benefit 3, corresponding to the substitution of nodes w_3, w_4, w_7 for nodes v_2, v_3, v_4 , respectively, and a least-cost transformation of the subtree of T_1 rooted at node v_1 into the subtree of T_2 rooted at node w_1 thus has benefit $3 + 1 = 4$, corresponding to the substitution of nodes w_1, w_3, w_4, w_7 for nodes v_1, v_2, v_3, v_4 , respectively. Notice that matrices of smaller problems have an additional left column and an additional top row, in order to allow for edit operations on the initial subtree of T_1 and the initial subtree of T_2 .

The following bottom-up dynamic programming algorithm for the tree edit problem makes an initial call to the recursive bottom-up dynamic programming procedure upon the subtree of T_1 rooted at node root[T_1] and the subtree of T_2 rooted at node root[T_2], in order to extract a least-cost mapping M of T_1 to T_2 from the list of tree edit operations L computed by the recursive procedure.

102a ⟨techniques 66⟩ +≡

```

int bottom_up_dynamic_programming_tree_edit(
    const tree& T1,
    const tree& T2,
    node_array<node>& M)
{
    ⟨set up children arrays 104b⟩
    list<two_tuple<node,node> > L;
    bottom_up_dyn_prog_tree_edit(T1,child1,T1.root(),T2,child2,T2.root(),L);
    two_tuple<node,node> EDIT;
    forall(EDIT,L) {
        node v = EDIT.first();
        node w = EDIT.second();
        if (v ≠ nil) { // deletion or substitution
            M[v] = w;
        }
    }
}

```

The actual computation of a least-cost transformation of T_1 to T_2 is done by the following recursive procedure, which finds the least-cost transformation of the subtree of T_1 rooted at node v , with children nodes v_i, \dots, v_j , into the subtree of T_2 rooted at node w , with children nodes w_k, \dots, w_ℓ , by taking the transformation of least cost among the least-cost transformations of the subgraph of T_1 consisting of the subtrees rooted in turn at nodes $\{v_i\}, \{v_i, v_{i+1}\}, \dots, \{v_i, v_{i+1}, \dots, v_j\}$, into the subgraph of T_2 consisting of the subtrees rooted in turn at nodes $\{w_k\}, \{w_k, w_{k+1}\}, \dots, \{w_k, w_{k+1}, \dots, w_\ell\}$.

The least-cost transformation is represented by a list L of tree edit operations, and the matrix of smaller problems is represented by two matrices, an integer matrix D of benefits and a matrix E of lists of edit operations. Both matrices have an additional left column and an additional top row, in order to allow for edit operations on the initial subtrees of T_1 and T_2 .

102b ⟨techniques 66⟩ +≡

```

int bottom_up_dyn_prog_tree_edit(
    const tree& T1,
    const node_array<array<node> >& child1,
    node r1,

```

```

const tree& T2,
const node_array<array<node>>& child2,
node r2,
list<two_tuple<node,node>>& L)
{
int m = T1.number_of_children(r1);
int n = T2.number_of_children(r2);
array2<int> D(0,m,0,n);
array2<list<two_tuple<node,node>>> E(0,m,0,n);
D(0,0) = 0;
for (int i = 1; i ≤ m; i++) {
    D(i,0) = D(i-1,0);
    two_tuple<node,node> DEL(child1[r1][i],nil);
    E(i,0) = E(i-1,0);
    E(i,0).append(DEL);
}
for (int j = 1; j ≤ n; j++) {
    D(0,j) = D(0,j-1);
    two_tuple<node,node> INS(nil,child2[r2][j]);
    E(0,j) = E(0,j-1);
    E(0,j).append(INS);
}
for (int i = 1; i ≤ m; i++) {
    for (int j = 1; j ≤ n; j++) {
        node rr1 = child1[r1][i];
        node rr2 = child2[r2][j];
        int del = D(i-1,j);
        int ins = D(i,j-1);
        D(i,j) = led_max(del,ins);
        list<two_tuple<node,node>> LL;
        int subst = bottom_up_dyn_prog_tree_edit(T1,child1,rr1,T2,child2,rr2,LL);
        if (del ≥ D(i-1,j-1) + subst) { // delete
            D(i,j) = del;
            two_tuple<node,node> DEL(rr1,nil);
            E(i,j) = E(i-1,j);
            E(i,j).append(DEL);
        } else {
            if (ins ≥ D(i-1,j-1) + subst) { // insert
                D(i,j) = ins;
                two_tuple<node,node> INS(nil,rr2);
                E(i,j) = E(i,j-1);
                E(i,j).append(INS);
            } else { // substitute
                D(i,j) = D(i-1,j-1) + subst;
                E(i,j) = E(i-1,j-1);
                E(i,j).conc(LL);
            }
        }
    }
    two_tuple<node,node> SUBST(r1,r2);
    L = E(m,n);
    L.append(SUBST);
}
return D(m,n)+1;
}

```

A few implementation details still need to be filled in, though. In the top-down dynamic programming algorithm for the tree edit, maintaining a dictionary (implemented by a hashing table) with four-tuples of nodes as keys, requires the definition of a hash function mapping four-tuples of nodes to integers. The default LEDA hash function is re-defined by the following procedure to a simple hash function on four-tuples of nodes, consisting of summing up the indices of the nodes in the key, where the index of a node in a tree is given by the internal numbering of the nodes in the LEDA graph representation of the tree.

104a $\langle \text{techniques } 66 \rangle + \equiv$

```

int Hash(
    const four_tuple<node,node,node,node>& N)
{
    int sum = index(N.first());
    sum += index(N.second());
    sum += index(N.third());
    sum += index(N.fourth());
    return sum;
}

```

Further, the bottom-up dynamic programming algorithm for the tree edit needs efficient access to the k th child of a node in an ordered tree, an operation which is not offered by the representation of trees as LEDA graphs given in Sect. 1.4. Therefore, the children of a node v in a tree are stored in an array of nodes (of the tree) indexed by children number, and the children of all nodes in the tree are stored in an array of these arrays of nodes, indexed by node. The i th child of node v in tree T_1 is stored in $\text{child1}[v][i]$, and the j th child of node w in tree T_2 is stored in $\text{child2}[w][j]$.

104b $\langle \text{set up children arrays } 104b \rangle \equiv$

```

node_array<array<node>> child1(T1);
{ node v,w;
  forall_nodes(v,T1) {
    if (  $\neg T1.\text{is\_leaf}(v)$  ) {
      array<node> CHILD(1,T1.number_of_children(v));
      int k = 1;
      forall_children(w,v)
        CHILD[k] = w;
        child1[v] = CHILD;
    } }
node_array<array<node>> child2(T2);
{ node v,w;
  forall_nodes(v,T2) {
    if (  $\neg T2.\text{is\_leaf}(v)$  ) {

```

```

array<node> CHILD(1,T2.number_of_children(v));
int k = 1;
forall_children(w,v)
    CHILD[k++] = w;
    child2[v] = CHILD;
} } }

```

Lemma 2.29. *The bottom-up dynamic programming algorithm for finding a least-cost transformation of an ordered tree T_1 to an ordered tree T_2 with respectively n_1 and n_2 nodes, runs in $O(n_1 n_2)$ time using $O(n_1 n_2)$ additional space.*

Proof. Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be ordered trees with respectively n_1 and n_2 nodes. The algorithm makes one recursive call to the bottom-up dynamic programming procedure upon each pair of nodes $v \in V_1$ and $w \in V_2$ with $\text{depth}[v] = \text{depth}[w]$. Let also $m = \min\{\text{depth}[T_1], \text{depth}[T_2]\}$, and let $a_i = |\{v \in V_1 \mid \text{depth}[v] = i\}|$ and $b_i = |\{w \in V_2 \mid \text{depth}[w] = i\}|$ for $0 \leq i \leq m$. The number of recursive calls is thus $\sum_{i=0}^m a_i b_i \leq \sum_{i=0}^m a_i \sum_{i=0}^m b_i \leq n_1 n_2$, because $a_i \geq 0$ for $0 \leq i \leq m$. Therefore, the algorithm makes $O(n_1 n_2)$ recursive calls. Further, $O(n_1 n_2)$ additional space is used. \square

2.5.1 Interactive Demonstration of Dynamic Programming

The top-down dynamic programming algorithm for the tree edit is integrated next in the interactive demonstration of graph algorithms. Again, a simple checker for the tree edit that provides some visual reassurance consists of highlighting the nodes of the ordered trees T_1 and T_2 that belong in the mapping corresponding to the least-cost transformation of T_1 into T_2 , where nonhighlighted nodes of T_1 are deleted from T_1 , and nonhighlighted nodes of T_2 are inserted into T_2 by the transformation. The number of node deletions and insertions made in the least-cost transformation found is also reported.

```

<demo techniques 68>+≡
void gw_top_down_dynamic_programming_tree_edit(
    GraphWin& gwI)
{
    graph& G1 = gwI.get_graph();
    tree T1(G1);

```

```

graph G2;
GraphWin gw2(G2,500,500,"Top-Down Dynamic Programming Tree Edit");
gw2.display();
gw2.message("Enter second tree. Press done when finished");
gw2.edit();
gw2.del_message();

tree T2(G2);

node_array<node> M(T1);
int dist = top_down_dynamic_programming_tree_edit(T1,T2,M);

gw1.save_all_attributes();
int del = 0;
int ins = T2.number_of_nodes();
node v;
forall_nodes(v,T1) {
    if ( M[v] == nil ) {
        del++;
    } else {
        ins--;
        gw1.set_color(v,blue);
        gw2.set_color(M[v],blue);
    }
}
gw1.message(string("Least-cost transformation makes %i node deletions
and %i node insertions",del,ins));
gw1.wait();
gw1.del_message();
gw1.restore_all_attributes();
}
}

```

The bottom-up dynamic programming algorithm for the tree edit is also integrated next in the interactive demonstration of graph algorithms.

```

⟨demo techniques 68⟩ +≡
void gw_bottom_up_dynamic_programming_tree_edit(
    GraphWin& gw1)
{
    graph& G1 = gw1.get_graph();
    tree T1(G1);

    graph G2;
    GraphWin gw2(G2,500,500,"Bottom-Up Dynamic Programming Tree Edit");
    gw2.display();
    gw2.message("Enter second tree. Press done when finished");
    gw2.edit();
    gw2.del_message();

    tree T2(G2);

    node_array<node> M(T1);

```

```

int dist = bottom_up_dynamic_programming_tree_edit(T1,T2,M);

gw1.save_all_attributes();
int del = 0;
int ins = T2.number_of_nodes();
node v;
forall_nodes(v,T1) {
    if ( M[v] == nil ) {
        del++;
    } else {
        ins--;
        gw1.set_color(v,blue);
        gw2.set_color(M[v],blue);
    }
}
gw1.message(string("Least-cost transformation makes %i node deletions
    and %i node insertions",del,ins));
gw1.wait();
gw1.del_message();
gw1.restore_all_attributes();
}
}

```

Summary

Some of the fundamental algorithmic techniques used for solving combinatorial problems on trees and graphs are reviewed in this chapter. The techniques of backtracking, branch-and-bound, divide-and-conquer, and dynamic programming are discussed in detail. Further, these techniques are illustrated by the problem of computing the edit distance and finding a least-cost transformation between two ordered trees.

Bibliographic Notes

The backtracking technique was introduced in [43, 129, 353]. The average-case analysis of backtracking algorithms is discussed in [197, 263]. A thorough treatment of backtracking and branch-and-bound for exhaustive search can be found in [271, Chap. 4]. The branch-and-bound technique is further reviewed in [207].

The divide-and-conquer algorithm for finding a least-cost transformation between two ordered trees is based on [250]. A thorough exposition of techniques for solving general recurrences can be

found in [83, Chap. 4] and also in [136, 263, 291]. Techniques for the solution of divide-and-conquer recurrences are further discussed in [33, 178, 276, 346, 347, 357].

A dynamic programming formulation of several combinatorial problems is given in [156]. A thorough treatment of dynamic programming can be found in [28, 30, 98]. See also [83, Chap. 15]. The dynamic programming algorithm for finding a least-cost transformation between two ordered trees is based on [293, 377].

The edit distance between ordered trees constrained by insertion and deletion of leaves only was introduced in [250, 293, 377]. Further algorithms for computing the edit distance between ordered trees include [218, 301, 313, 316, 368, 380, 382], where algorithms for computing the edit distance between unordered trees include [316, 381, 383], and [315, 384] between free unordered trees, that is, acyclic graphs.

Edit graphs were used by several efficient algorithms for comparing strings and sequences [237, 244, 375], and were first used for tree comparison in [70]. The tree edit distance problem is closely related with isomorphism problems on trees and with the tree inclusion problem [6, 182], in which the elementary edit operation of insertion is forbidden. See also the bibliographic notes for Chap. 4.

Review Problems

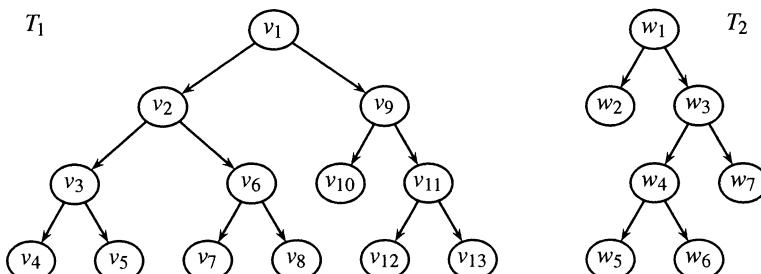


Fig. 2.16. Ordered trees for problem 2.1. Nodes are numbered according to the order in which they are visited during a preorder traversal.

- 2.1** Give all valid transformations and a least-cost transformation between the trees of Fig. 2.16, under the assumption that $\gamma(v, w) = 1$ for all edit operations of the form $v \mapsto w$ with $v \in V_1 \cup \{\lambda\}$, $w \in V_2 \cup \{\lambda\}$, and $v \neq w$.
- 2.2** Give a formulation of the general tree edit distance problem, without the constraint that deletions and insertions be made on leaves only. In a general transformation between two ordered trees, the parent of a deleted node becomes the parent of the children (if any) of the node, and the relative order among sibling nodes is preserved (left siblings of the deleted node become left siblings of the children of the deleted node which, in turn, become left siblings of the right siblings of the deleted node). The same applies to the insertion of nonleaves.
- 2.3** Extend the formulation of the tree edit distance problem to deal with labeled trees. Recall that in a tree $T = (V, E)$, the information attached to a node $v \in V$ is denoted by $T[v]$ and the label of an arc $(v, w) \in E$ is denoted by $T[v, w]$.
- 2.4** Give a formulation of the tree inclusion problem, without the constraint that deletions be made on leaves only but with the additional constraint that insertions are forbidden.
- 2.5** In the divide-and-conquer formulation of the tree edit, the division of a tree into the subtree with nodes up to a certain node in preorder and the subtree with the remaining nodes, has the advantage that subtrees are defined by just giving their first and last node in preorder, but the size of the second subtree can be much smaller than half the size of the tree. Give an alternative divide-and-conquer formulation of the tree edit, dividing the trees into subtrees of about the same size.
- 2.6** Give an upper bound on the number of nodes in the backtracking tree, the branch-and-bound tree, and the divide-and-conquer tree for a tree edit problem. Give also an upper bound on the number of vertices in the dynamic programming directed acyclic graph for a tree edit problem.

Exercises

- 2.1** Adapt the tree edit algorithm based on shortest paths in the edit graph of the trees, in order to give the node assignment corresponding to the valid tree transformation found.
- 2.2** Give a backtracking algorithm for the general tree edit distance problem formulated in problem 2.2.
- 2.3** Extend the backtracking algorithm for the tree edit in order to enumerate not only node assignments that correspond to valid tree transformations, but the valid transformations themselves.
- 2.4** Extend the backtracking and branch-and-bound algorithms for the tree edit to labeled trees, according to the formulation of problem 2.3.
- 2.5** Give a branch-and-bound algorithm for the tree inclusion problem formulated in problem 2.4.
- 2.6** Give a correctness certificate for the tree edit problem.
- 2.7** Give a divide-and-conquer algorithm for the tree edit implementing the alternative formulation of problem 2.5, by which trees are divided into subtrees of about the same size.
- 2.8** Perform experiments on random ordered trees to compare the efficiency of the different algorithms for the tree edit problem. Perform additional experiments on complete binary trees, for the certification of the upper bound given in problem 2.6.
- 2.9** Give a divide-and-conquer algorithm for the tree inclusion problem formulated in problem 2.4. Use memoization to extend it to a top-down dynamic programming algorithm for the tree inclusion problem.
- 2.10** Give a bottom-up dynamic programming algorithm for the tree inclusion problem formulated in problem 2.4. Give also the time and space complexity of the algorithm.

Part II

Algorithms on Trees

3. Tree Traversal

It's like switching your point of view. Things sometimes look complicated from one angle, but simple from another.

—Douglas R. Hofstadter [164]

Most algorithms on trees require a systematic method of visiting the nodes of a tree. The most common methods of exploring a tree are *preorder*, *postorder*, *top-down*, and *bottom-up* traversal.

- In a *preorder* traversal of a rooted tree, parents are visited before children, and siblings are visited in left-to-right order.
- In a *postorder* traversal of a rooted tree, children are visited before parents, and siblings are visited in left-to-right order.
- In a *top-down* traversal of a rooted tree, also known as *level-order* traversal, nodes are visited in order of nondecreasing depth.
- In a *bottom-up* traversal of a tree, not necessarily rooted, nodes are visited in order of nondecreasing height.

These systematic methods of visiting the nodes of a tree are the subject of this chapter.

3.1 Preorder Traversal of a Tree

A traversal of a tree $T = (V, E)$ on n nodes is just a bijection order : $V \rightarrow \{1, \dots, n\}$. In an operational view, a traversal of a tree consists of visiting first the node v with $\text{order}[v] = 1$, then the node w with $\text{order}[w] = 2$, and so on, until visiting the node z with $\text{order}[z] = n$.

In a preorder traversal of a tree, the root of the tree is visited first, followed by a preorder traversal of the subtree rooted in turn at each

of the children of the root. The order in which the children nodes are considered is significant for an ordered tree. The subtree rooted at the first child is traversed first, followed by the subtree rooted at the next sibling, etc. Such an order among the children of a node is also fixed by the representation adopted for both unordered and ordered trees.

Recall from Sect. 1.1 that in a tree $T = (V, E)$, $\text{first}[v]$ denotes the first child of node v , $\text{next}[v]$ denotes the next sibling of node v , $\text{last}[v]$ denotes the last child of node v , and $\text{size}[v]$ denotes the number of nodes in the subtree of T rooted at node v , for all nodes $v \in V$.

Definition 3.1. Let $T = (V, E)$ be a tree on n nodes rooted at node r . A bijection $\text{order} : V \rightarrow \{1, \dots, n\}$ is a **preorder traversal** of T if $\text{order}[r] = 1$ and

- $\text{order}[\text{first}[v]] = \text{order}[v] + 1$ (if v is not a leaf)
- $\text{order}[\text{next}[v]] = \text{order}[v] + \text{size}[v]$ (if v is not a last child)

for all nodes $v \in V$.

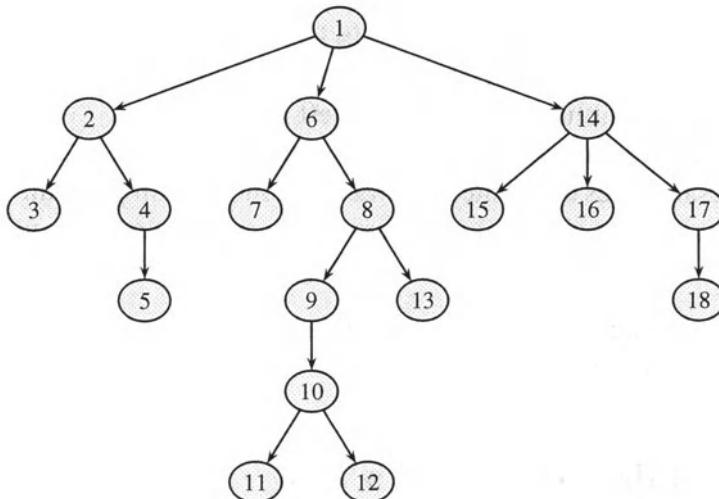


Fig. 3.1. Preorder traversal of a rooted tree. Nodes are numbered according to the order in which they are visited during the traversal.

Example 3.2. The preorder traversal of a rooted tree is illustrated in Fig. 3.1. The root has order 1, the first child of a nonleaf node with

order k has order $k + 1$, and the next sibling of a node with order k has order k plus the number of nodes in the subtree rooted at the node. For instance, the first child of the node with order 6 has order $7 = 6 + 1$, and the next sibling of the same node has order $14 = 6 + 8$, because the subtree rooted at the node with order 6 has 8 nodes.

Now, the preorder traversal of a tree can be constructed from the preorder traversals of the subtrees rooted at the children of the root of the tree. This decomposition property allows application of the divide-and-conquer technique, yielding a simple recursive algorithm.

The following algorithm performs a preorder traversal of a rooted tree. The order in which nodes are visited is stored in the array of nodes $\text{order} : V \rightarrow \text{int}$, which is shared among all recursive calls. Thus, nothing has to be done in order to combine the traversals of the subtrees into a traversal of the whole tree.

115a

```
<preorder tree traversal 115a>≡
void rec_preorder_tree_traversal(
    const tree& T,
    node_array<int>& order)
{
    int num = 1;
    rec_preorder_subtree_traversal(T.T.root(), order, num);
    double_check_preorder_tree_traversal(T, order);
}
```

115b

```
<preorder tree traversal 115a>+≡
void rec_preorder_subtree_traversal(
    const tree& T,
    const node v,
    node_array<int>& order,
    int& num)
{
    order[v] = num++; // visit node v
    node w;
    forall_children(w, v)
        rec_preorder_subtree_traversal(T, w, order, num);
}
```

The following double-check of preorder tree traversal, although being redundant, gives some reassurance of correctness of the implementation. It verifies that order is a preorder traversal of tree T , according to Definition 3.1. The size of the subtree rooted at each node of the tree is first computed during a preorder traversal of the tree.

115c $\langle \text{double-check preorder tree traversal 115c} \rangle \equiv$

```

void subtree_size(
    const tree& T,
    const node v,
    node_array<int>& size)
{
    size[v] = 1;
    if (  $\neg T.\text{is\_leaf}(v)$  ) {
        node w;
        forall_children(w,v) {
            subtree_size(T,w,size);
            size[v] += size[w];
        } } }
}

```

116 $\langle \text{double-check preorder tree traversal 115c} \rangle + \equiv$

```

void double_check_preorder_tree_traversal(
    const tree& T,
    node_array<int>& order)
{
    node_array<int> size(T);
    subtree_size(T,T.root(),size);
    if ( order[T.root()]  $\neq$  1 )
        error_handler(1,
            "Wrong implementation of preorder tree traversal");
    node v;
    forall_nodes(v,T) {
        if (  $\neg T.\text{is\_leaf}(v)$   $\wedge$  order[T.first_child(v)]  $\neq$  order[v] + 1 )
            error_handler(1,
                "Wrong implementation of preorder tree traversal");
        if (  $\neg T.\text{is\_last\_child}(v)$   $\wedge$ 
            order[T.next_sibling(v)]  $\neq$  order[v] + size[v] )
            error_handler(1,
                "Wrong implementation of preorder tree traversal");
    } }
}

```

Lemma 3.3. *The recursive algorithm for the preorder traversal of a rooted tree runs in $O(n)$ time using $O(n)$ additional space, where n is the number of nodes in the tree.*

Proof. Let $T = (V, E)$ be a tree on n nodes. The algorithm makes $O(n)$ recursive calls, one for each node of the tree. Further, $O(n)$ additional space is used.

The double-check of preorder tree traversal also runs in $O(n)$ time using $O(n)$ additional space. As a matter of fact, $O(n)$ recursive calls are made for computing subtree sizes, and the double-check itself is done in $O(n)$ time using $O(n)$ additional space. \square

The stack implicit in the recursive algorithm for preorder tree traversal can be made explicit, yielding a simple iterative algorithm in which a stack of nodes holds those nodes, the subtrees rooted at which are waiting to be traversed. Initially, the stack contains only the root of the tree. Every time a node is popped and visited, the children of the popped node are pushed, one after the other, starting with the last child, following with the previous sibling of the last child, and so on, until having pushed the first child of the popped node. The procedure terminates when the stack has been emptied and no children nodes remain to be pushed.

117

`<preorder tree traversal 115a> +≡`

```
void preorder_tree_traversal(
    const tree& T,
    node_array<int>& order)
{
    stack<node> S;
    node v,w;
    S.push(T.root());
    int num = 1;
    do {
        v = S.pop();
        order[v] = num++; // visit node v
        w = T.last_child(v);
        while ( w ≠ nil ) {
            S.push(w);
            w = T.previous_sibling(w);
        }
    } while ( !S.empty() );
    double_check_preorder_tree_traversal(T,order);
}
```

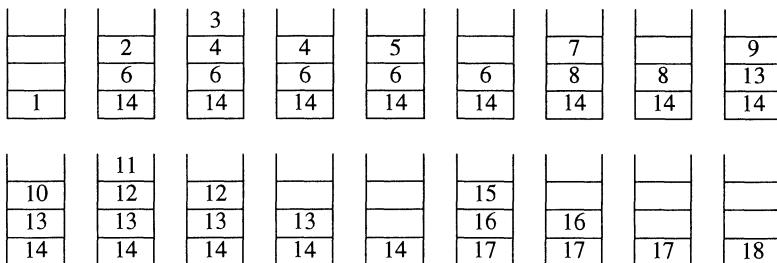


Fig. 3.2. Evolution of the stack of nodes during execution of the preorder traversal procedure, upon the tree of Fig. 3.1.

Example 3.4. Consider the execution of the preorder traversal procedure with the help of a stack of nodes, illustrated in Fig. 3.2. Starting off with a stack containing only the root of the tree, the following evolution of the stack of nodes right before a node is popped shows that nodes are, as a matter of fact, popped in preorder traversal order: 1, 2, ..., 18.

Lemma 3.5. *The iterative algorithm for preorder traversal of a rooted tree runs in $O(n)$ time using $O(n)$ additional space, where n is the number of nodes in the tree.*

Proof. Let $T = (V, E)$ be a tree on n nodes. Since each node of the tree is pushed and popped exactly once, the loop is executed n times and the algorithm runs in $O(n)$ time. Further, since each node is pushed only once, the stack cannot ever contain more than n nodes and the algorithm uses $O(n)$ additional space.

The double-check of preorder tree traversal also runs in $O(n)$ time using $O(n)$ additional space. \square

3.1.1 Interactive Demonstration of Preorder Traversal

The preorder traversal algorithm, in both the recursive and the iterative implementation, is integrated next in the interactive demonstration of graph algorithms. A simple checker for tree traversal that provides some visual reassurance consists of highlighting the nodes of the tree as the preorder traversal proceeds.

118a

```
<demo preorder tree traversal 118a>≡
void gw_rec_preorder_tree_traversal(
    GraphWin& gw)
{
    graph& G = gw.get_graph();
    tree T(G);
    node_array<int> order(T);
    rec_preorder_tree_traversal(T,order);
    <show tree traversal 119>
}
```

118b

```
<demo preorder tree traversal 118a>+≡
void gw_preorder_tree_traversal(
    GraphWin& gw)
{
    graph& G = gw.get_graph();
```

```

tree T(G);
node_array<int> order(T);
preorder_tree_traversal(T,order);
⟨show tree traversal 119⟩
}

```

A tree traversal can be demonstrated by highlighting each of the nodes of the tree in turn, according to the order defined by the traversal. A pause of half a second between nodes should suffice to get the picture of how the traversal proceeds.

119

```

⟨show tree traversal 119⟩≡
{ gw.save_all_attributes();

  node v;
  forall_nodes(v,T)
    gw.set_label(v,string("%i",order[v]));

  int n = T.number_of_nodes();
  array<node> disorder(1,n);
  forall_nodes(v,T)
    disorder[order[v]] = v;

  for ( int i = 1; i ≤ n; i++ ) {
    gw.set_color(disorder[i],blue);
    leda_wait(0.5);
  }

  gw.wait();
  gw.restore_all_attributes();
}

```

3.2 Postorder Traversal of a Tree

In a postorder traversal of a tree, a postorder traversal of the subtree rooted in turn at each of the children of the root is performed first, and the root of the tree is visited last. The order in which the children nodes are considered is significant. The subtree rooted at the first child is traversed first, followed by the subtree rooted at the next sibling, etc.

Definition 3.6. Let $T = (V, E)$ be a tree on n nodes rooted at node r . A bijection $\text{order} : V \rightarrow \{1, \dots, n\}$ is a **postorder traversal** of T if $\text{order}[r] = n$ and

- $\text{order}[\text{last}[v]] = \text{order}[v] - 1$ *(if v is not a leaf)*
- $\text{order}[\text{next}[v]] = \text{order}[v] + \text{size}[\text{next}[v]]$ *(if v is not a last child)*

for all nodes $v \in V$.

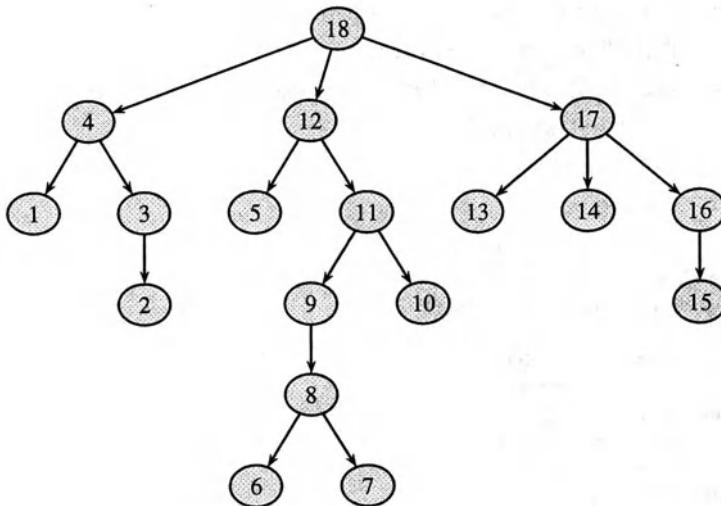


Fig. 3.3. Postorder traversal of the rooted tree of Fig. 3.1. Nodes are numbered according to the order in which they are visited during the traversal.

Example 3.7. The postorder traversal of a rooted tree is illustrated in Fig. 3.3. The root has order n , the last child of a nonleaf node with order k has order $k - 1$, and the next sibling of a node with order k has order k plus the number of nodes in the subtree rooted at the next sibling node. For instance, the last child of the node with order 12 has order $11 = 12 - 1$, and the next sibling of the same node has order $17 = 12 + 5$, because the subtree rooted at the node with order 17 has five nodes.

As with preorder traversal, the postorder traversal of a tree can be constructed from the postorder traversals of the subtrees rooted at the children of the root of the tree. This decomposition property allows application of the divide-and-conquer technique, yielding again a simple recursive algorithm.

The following algorithm performs a postorder traversal of a rooted tree. The order in which nodes are visited is stored in the array of nodes $\text{order} : V \rightarrow \text{int}$, which is shared among all recursive calls. Thus, nothing has to be done in order to combine the traversals of the subtrees into a traversal of the whole tree.

121a

```
(postorder tree traversal 121a)≡
void rec_postorder_tree_traversal(
    const tree& T,
    node_array<int>& order)
{
    int num = 1;
    rec_postorder_subtree_traversal(T,T.root(),order,num);
    double_check_postorder_tree_traversal(T,order);
}
```

121b

```
(postorder tree traversal 121a)+≡
void rec_postorder_subtree_traversal(
    const tree& T,
    const node v,
    node_array<int>& order,
    int& num)
{
    node w;
    forall_children(w,v)
        rec_postorder_subtree_traversal(T,w,order,num);
    order[v] = num++; // visit node v
}
```

The following double-check of postorder tree traversal, although being redundant, gives some reassurance of correctness of the implementation. It verifies that order is a postorder traversal of tree T , according to Definition 3.6.

121c

```
(double-check postorder tree traversal 121c)≡
void double_check_postorder_tree_traversal(
    const tree& T,
    node_array<int>& order)
{
    node_array<int> size(T);
    subtree_size(T,T.root(),size);
    if (order[T.root()] ≠ T.number_of_nodes())
        error_handler(1,
            "Wrong implementation of postorder tree traversal");
    node v;
    forall_nodes(v,T) {
        if (¬T.is_leaf(v) ∧ order[T.last_child(v)] ≠ order[v] - 1)
            error_handler(1,
                "Wrong implementation of postorder tree traversal");
```

```

if (  $\neg T.\text{is\_last\_child}(v) \wedge$ 
     $\text{order}[T.\text{next\_ sibling}(v)] \neq$ 
     $\text{order}[v] + \text{size}[T.\text{next\_ sibling}(v)]$  )
error_handler(1,
    "Wrong implementation of postorder tree traversal");
}
}

```

Lemma 3.8. *The recursive algorithm for postorder traversal of a rooted tree runs in $O(n)$ time using $O(n)$ additional space, where n is the number of nodes in the tree.*

Proof. Let $T = (V, E)$ be a tree on n nodes. The algorithm makes $O(n)$ recursive calls, one for each node of the tree. Further, $O(n)$ additional space is used.

The double-check of postorder tree traversal also runs in $O(n)$ time using $O(n)$ additional space. As a matter of fact, $O(n)$ recursive calls are made for computing subtree sizes, and the double-check itself is done in $O(n)$ time using $O(n)$ additional space. \square

The stack implicit in the recursive algorithm for postorder tree traversal can also be made explicit, yielding a simple iterative algorithm in which a stack of nodes holds those nodes, the subtrees rooted at which are waiting to be traversed. Initially, the stack contains only the root of the tree. Every time a node is popped and visited, the children of the popped node are pushed, one after the other, starting with the first child, following with the next sibling of the first child, and so on, until having pushed the last child of the popped node.

The procedure terminates when the stack has been emptied and no children nodes remain to be pushed. However, nodes are popped in reverse postorder traversal order and then, the postorder number assigned to node v when visiting it is $n - \text{order}[v] + 1$, instead of just $\text{order}[v]$.

```

⟨postorder tree traversal 121a⟩ +≡
void postorder_tree_traversal(
    const tree&  $T$ ,
    node_array<int>&  $\text{order}$ )
{
    stack<node>  $S$ ;
    node  $v, w$ ;
     $S.\text{push}(T.\text{root}())$ ;
    int  $\text{num} = 1$ ;
}

```

```

do {
     $v = S.pop();$ 
     $order[v] = T.number\_of\_nodes() - num++ + 1;$  // visit node  $v$ 
    forall_children( $w, v$ )
         $S.push(w);$ 
} while ( $\neg S.empty()$ );
double_check_postorder_tree_traversal( $T, order$ );
}

```

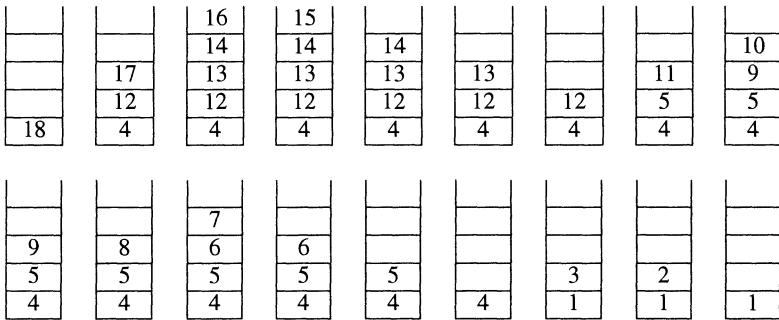


Fig. 3.4. Evolution of the stack of nodes during execution of the postorder traversal procedure, upon the tree of Fig. 3.3.

Example 3.9. Consider the execution of the postorder traversal procedure with the help of a stack of nodes, illustrated in Fig. 3.4. Starting off with a stack containing only the root of the tree, the following evolution of the stack of nodes right before a node is popped shows that nodes are, as a matter of fact, popped in reverse postorder traversal order: $18, 17, \dots, 1$.

Lemma 3.10. *The iterative algorithm for postorder traversal of a rooted tree runs in $O(n)$ time using $O(n)$ additional space, where n is the number of nodes in the tree.*

Proof. Let $T = (V, E)$ be a tree on n nodes. Since each node of the tree is pushed and popped exactly once, the loop is executed n times and the algorithm runs in $O(n)$ time. Further, since each node is pushed only once, the stack cannot ever contain more than n nodes and the algorithm uses $O(n)$ additional space.

The double-check of postorder tree traversal also runs in $O(n)$ time using $O(n)$ additional space. \square

3.2.1 Interactive Demonstration of Postorder Traversal

The postorder traversal algorithm, in both the recursive and the iterative implementation, is integrated next in the interactive demonstration of graph algorithms. A simple checker for tree traversal that provides some visual reassurance consists of highlighting the nodes of the tree as the postorder traversal proceeds.

124a

```
<demo postorder tree traversal 124a>≡
void gw_rec_postorder_tree_traversal(
    GraphWin& gw)
{
    graph& G = gw.get_graph();
    tree T(G);
    node_array<int> order(T);
    rec_postorder_tree_traversal(T,order);
    <show tree traversal 119>
}
```

124b

```
<demo postorder tree traversal 124a>+≡
void gw_postorder_tree_traversal(
    GraphWin& gw)
{
    graph& G = gw.get_graph();
    tree T(G);
    node_array<int> order(T);
    postorder_tree_traversal(T,order);
    <show tree traversal 119>
}
```

3.3 Top-Down Traversal of a Tree

In a top-down traversal of a tree, nodes are visited in order of nondecreasing depth, and nodes at the same depth are visited in left-to-right order.

Recall from Sect. 1.1 that in a tree $T = (V, E)$, $\text{depth}[v]$ denotes the depth of node v , that is, the length of the unique path from the root of T to node v , for all nodes $v \in V$. Let also $\text{rank}[v]$ denote the order in which node v is visited during a preorder traversal of T , for all nodes $v \in V$.

Definition 3.11. Let $T = (V, E)$ be a tree on n nodes rooted at node r . A bijection $\text{order} : V \rightarrow \{1, \dots, n\}$ is a **top-down traversal** of T if the following conditions

- if $\text{order}[v] < \text{order}[w]$ then $\text{depth}[v] \leq \text{depth}[w]$
- if $\text{depth}[v] = \text{depth}[w]$ and $\text{rank}[v] < \text{rank}[w]$ then $\text{order}[v] < \text{order}[w]$

are satisfied, for all nodes $v, w \in V$.

It follows that in a top-down traversal of a tree, the root is the first node to be visited.

Lemma 3.12. Let $T = (V, E)$ be a tree on n nodes rooted at node r , and let $\text{order} : V \rightarrow \{1, \dots, n\}$ be a top-down traversal of T . Then, $\text{order}[r] = 1$.

Proof. Let $T = (V, E)$ be a tree on n nodes rooted at node r , let $\text{order} : V \rightarrow \{1, \dots, n\}$ be a top-down traversal of T , and suppose that $\text{order}[r] > 1$. Then, there is a node $v \in V$, $v \neq r$ with $\text{order}[v] = 1$, because $\text{order} : V \rightarrow \{1, \dots, n\}$ is a bijection and then, $\text{depth}[v] \leq \text{depth}[r]$ by Definition 3.11. But since $\text{depth}[r] = 0$, it must also be true that $\text{depth}[v] = 0$, that is, $v = r$, yielding a contradiction. Therefore, $\text{order}[r] = 1$. \square

Example 3.13. Fig. 3.5 illustrates the top-down traversal of a rooted tree. The root has order 1, nodes are visited in order of nondecreasing depth, and nodes at the same depth are visited by increasing rank of preorder traversal. For instance, the node with order 8 at depth 3 and preorder rank 8 is visited before the node with order 13 at depth 4, and also before the node with order 11 at depth 3, which has preorder rank 17.

Unlike preorder and postorder traversal, the top-down traversal of a tree *cannot* be constructed from the traversals of the subtrees rooted at the children of the root of the tree. A top-down traversal of a tree can be easily realized with the help of a queue of nodes, which holds those nodes, the subtrees rooted at which are waiting to be traversed.

Initially, the queue contains only the root of the tree. Every time a node is dequeued and visited, the children of the dequeued node are

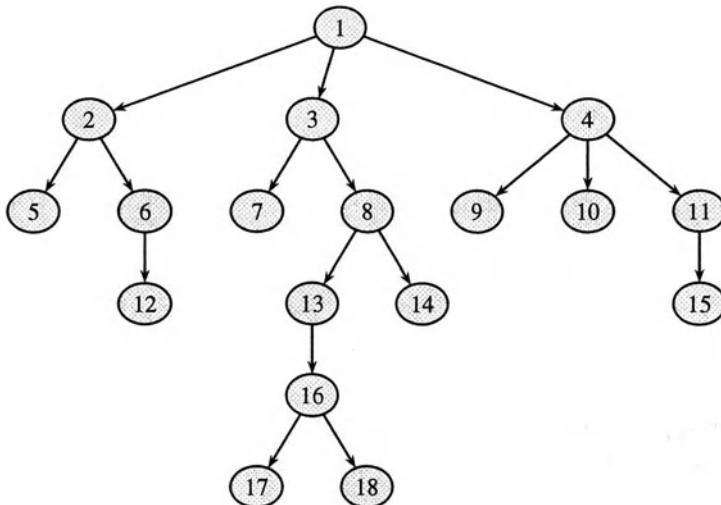


Fig. 3.5. Top-down traversal of the rooted tree of Fig. 3.1. Nodes are numbered according to the order in which they are visited during the traversal.

enqueued, one after the other, starting with the first child, following with the next sibling of the first child, and so on, until having enqueue the last child of the dequeued node. The procedure terminates when the queue has been emptied and no children nodes remain to be enqueued.

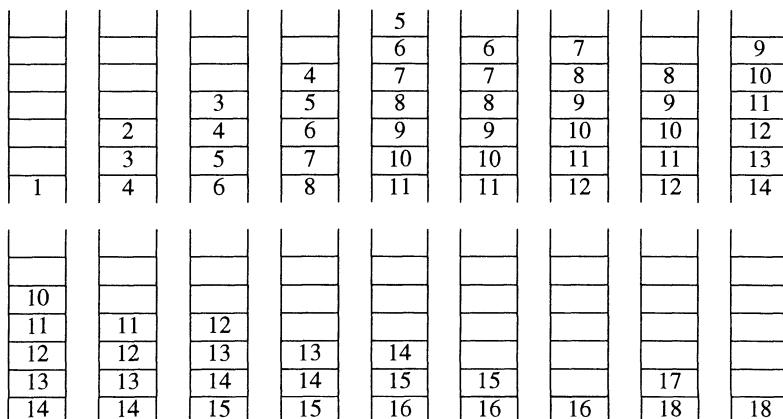


Fig. 3.6. Evolution of the queue of nodes during execution of the top-down traversal procedure, upon the tree of Fig. 3.5.

Example 3.14. Consider the execution of the top-down traversal procedure with the help of a queue of nodes, illustrated in Fig. 3.6. Starting off with a queue containing only the root of the tree, the evolution of the queue of nodes right before a node is dequeued shows that nodes are, as a matter of fact, dequeued in top-down traversal order: 1, 2, ..., 18.

The following algorithm performs a top-down traversal of a rooted tree $T = (V, E)$, using a queue of nodes Q to implement the previous procedure. The order in which nodes are visited during the top-down traversal is stored in the array of nodes $\text{order} : V \rightarrow \text{int}$.

127a $\langle\text{top-down tree traversal 127a}\rangle \equiv$

```
void top_down_tree_traversal(
    const tree& T,
    node_array<int>& order)
{
    queue<node> Q;
    node v,w;
    Q.append(T.root());
    int num = 1;
    do {
        v = Q.pop();
        order[v] = num++; // visit node v
        forall_children(w,v)
            Q.append(w);
    } while ( !Q.empty() );
    double_check_top_down_tree_traversal(T,order);
}
```

The following double-check of top-down tree traversal, although being redundant, gives some reassurance of correctness of the implementation. It verifies that order is a top-down traversal of tree T , according to Definition 3.11.

127b $\langle\text{double-check top-down tree traversal 127b}\rangle \equiv$

```
void double_check_top_down_tree_traversal(
    const tree& T,
    node_array<int>& order)
{
    // compute depth and preorder rank of all nodes 128

    node v,w;
    forall_nodes(v,T) {
        forall_nodes(w,T) {
            if ( order[v] < order[w] & depth[v] > depth[w] )
                error_handler(1,
                    "Wrong implementation of top-down tree traversal");
        }
    }
}
```

```

if ( depth[v]  $\equiv$  depth[w]  $\wedge$  rank[v]  $<$  rank[w]  $\wedge$ 
    order[v]  $\geqslant$  order[w] )
    error_handler(1,
        "Wrong implementation of top-down tree traversal");
} } }

```

The depth and preorder rank of all nodes in the tree are computed next, during an iterative preorder traversal of the tree.

128

`(compute depth and preorder rank of all nodes 128)≡`

```

node_array<int> depth(T);
node_array<int> rank(T);
{ stack<node> S;
  node v,w;
  S.push(T.root());
  int num = 1;
  do {
    v = S.pop();
    rank[v] = num++;
    if ( T.is_root(v) ) {
      depth[v] = 0;
    } else {
      depth[v] = depth[T.parent(v)] + 1;
    }
    w = T.last_child(v);
    while ( w  $\neq$  nil ) {
      S.push(w);
      w = T.previous_sibling(w);
    }
  } while (  $\neg$ S.empty() );
  double_check_preorder_tree_traversal(T,rank);
}

```

Remark 3.15. Notice that the top-down tree traversal algorithm and the iterative preorder tree traversal algorithm differ only on the data structure used for holding those nodes, the subtrees rooted at which are waiting to be traversed (a stack for the preorder traversal, and a queue for the top-down traversal) and in the order in which the children of a node are inserted into the data structure (children nodes are enqueued in left-to-right order, but pushed into the stack in right-to-left order).

Lemma 3.16. *The algorithm for top-down traversal of a rooted tree runs in $O(n)$ time using $O(n)$ additional space, where n is the number*

of nodes in the tree. The double-check of top-down tree traversal runs in $O(n^2)$ time using $O(n)$ additional space.

Proof. Let $T = (V, E)$ be a tree on n nodes. Since each node of T is enqueued and dequeued exactly once, the loop is executed n times. Processing each dequeued node takes time linear in the number of children of the node, and then, processing all dequeued nodes takes $O(n)$ time. Therefore, the algorithm runs in $O(n)$ time. Further, since each node is enqueued only once, the queue cannot ever contain more than n nodes and the algorithm uses $O(n)$ additional space.

The double-check of top-down tree traversal runs in $O(n^2)$ time using $O(n)$ additional space. As a matter of fact, the depth and rank of all nodes are computed in $O(n)$ time using $O(n)$ additional space during a preorder traversal of the tree, and the double-check itself is done in $O(n^2)$ time in two nested loops over all nodes of the tree. \square

3.3.1 Interactive Demonstration of Top-Down Traversal

The top-down traversal algorithm is integrated next in the interactive demonstration of graph algorithms. Again, a simple checker for tree traversal that provides some visual reassurance consists of highlighting the nodes of the tree as the top-down traversal proceeds.

129

```
(demo top-down tree traversal 129)≡
void gw_top_down_tree_traversal(
    GraphWin& gw)
{
    graph& G = gw.get_graph();
    tree T(G);
    node_array<int> order(T);
    top_down_tree_traversal(T,order);
    {show tree traversal 119}
}
```

3.4 Bottom-Up Traversal of a Tree

In a bottom-up traversal of a tree, nodes are visited in order of non-decreasing height. Nodes at the same height are visited in order of nondecreasing depth, and nodes of the same height and depth are visited in left-to-right order.

Recall from Sect. 1.1 that in a tree $T = (V, E)$, $\text{height}[v]$ denotes the height of node v , that is, the length of a longest path from node v to any node in the subtree of T rooted at node v , for all nodes $v \in V$. Let also $\text{rank}[v]$ denote the order in which node v is visited during a preorder traversal of T , for all nodes $v \in V$.

Definition 3.17. *Let $T = (V, E)$ be a tree on n nodes rooted at node r . A bijection $\text{order} : V \rightarrow \{1, \dots, n\}$ is a **bottom-up traversal** of T if the following conditions*

- if $\text{order}[v] < \text{order}[w]$ then $\text{height}[v] \leq \text{height}[w]$
- if $\text{height}[v] = \text{height}[w]$ and $\text{depth}[v] < \text{depth}[w]$ then $\text{order}[v] < \text{order}[w]$
- if $\text{height}[v] = \text{height}[w]$, $\text{depth}[v] = \text{depth}[w]$ and $\text{rank}[v] < \text{rank}[w]$ then $\text{order}[v] < \text{order}[w]$

are satisfied, for all nodes $v, w \in V$.

The first condition in Definition 3.17 guarantees that in a bottom-up traversal of a tree, nodes are visited in order of nondecreasing height, while the second condition ensures that nodes of the same height are visited in order of nondecreasing depth, and the third condition ensures that nodes of the same height and depth are visited in left-to-right order.

It follows that in a bottom-up traversal of a tree, the root is the last node to be visited.

Lemma 3.18. *Let $T = (V, E)$ be a tree on n nodes rooted at node r , and let $\text{order} : V \rightarrow \{1, \dots, n\}$ be a bottom-up traversal of T . Then, $\text{order}[r] = n$.*

Proof. Let $T = (V, E)$ be a tree on n nodes rooted at node r , let $\text{order} : V \rightarrow \{1, \dots, n\}$ be a bottom-up traversal of T , and suppose that $\text{order}[r] < n$. Then, there is a node $v \in V$, $v \neq r$, with $\text{order}[v] = n$, because $\text{order} : V \rightarrow \{1, \dots, n\}$ is a bijection and then, $\text{height}[r] \leq \text{height}[v]$ by Definition 3.17. But since $\text{height}[r] = \max_{w \in V} \text{height}[w]$, it must also be true that $\text{height}[v] = \text{height}[r]$, that is, $v = r$, yielding a contradiction. Therefore, $\text{order}[r] = r$. \square

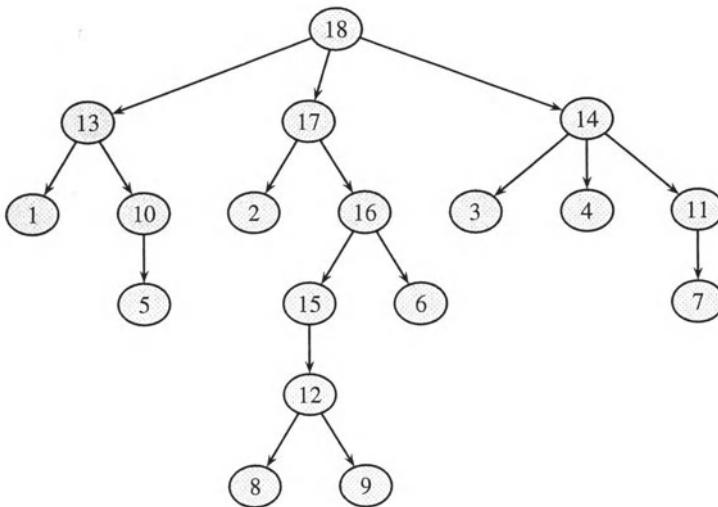


Fig. 3.7. Bottom-up traversal of the tree of Fig. 3.1. Nodes are numbered according to the order in which they are visited during the traversal.

Example 3.19. In Fig. 3.7, the bottom-up traversal of a tree is illustrated. The root has order n , nodes are visited in order of nondecreasing height, nodes at the same height are visited in order of nondecreasing depth, and nodes of the same height and depth are visited by increasing rank of preorder traversal. For instance, the node with order 13, which has height 2, depth 1, and preorder rank 2, is visited before the node with order 16, which has height 3, and also before the node with order 15, which has height 2 and depth 3, and before the node with order 14, which has height 2, depth 1, and preorder rank 14.

Unlike preorder and postorder traversal, but as with top-down traversal, the bottom-up traversal of a tree *cannot* be constructed from the traversals of the subtrees rooted at the children of the root of the tree. A bottom-up traversal of a tree can be easily realized with the help of a queue of nodes, though, which holds those nodes which are waiting to be traversed and the subtrees rooted at the children of which have already been traversed.

Initially, the queue contains all leaves of the tree in top-down traversal order. Every time a node is dequeued and visited, the number of nonvisited children of the parent of the node is decreased by one and, as soon as it reaches zero, the parent of the node is enqueued. The

procedure terminates when the queue has been emptied and no nodes remain to be enqueued.

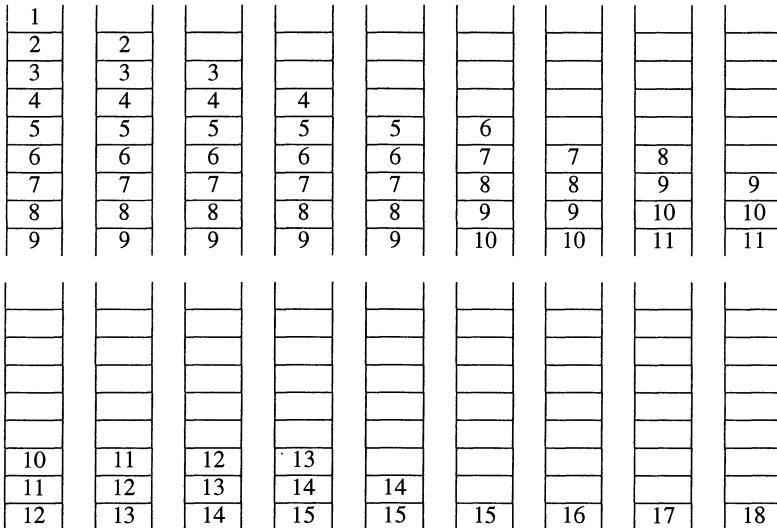


Fig. 3.8. Evolution of the queue of nodes during execution of the bottom-up traversal procedure, upon the tree of Fig. 3.7.

Example 3.20. Consider the execution of the bottom-up traversal procedure with the help of a queue of nodes, illustrated in Fig. 3.8. Starting off with a queue containing all leaves of the tree in top-down traversal order, the evolution of the queue of nodes right before a node is dequeued shows that nodes are, as a matter of fact, dequeued in bottom-up traversal order: 1, 2, ..., 18.

The following algorithm performs a bottom-up traversal of a rooted tree $T = (V, E)$, using queues of nodes Q and R to implement the previous procedure. The order in which nodes are visited during the bottom-up traversal is stored in the array of nodes $\text{order} : V \rightarrow \text{int}$.

```
<bottom-up tree traversal 132>≡
void bottom_up_tree_traversal(
    const tree& T,
    node_array<int>& order)
{
    queue<node> Q;
```

```

node_array<int> children(T,0);
edge e;
forall_edges(e,T)
  children[T.source(e)]++;
node v,w;

```

⟨enqueue all leaves in top-down traversal order 133a⟩

```

int num = 1;
do {
  v = Q.pop();
  order[v] = num++; // visit node v
  if ( !T.is_root(v) ) {
    children[T.parent(v)]--;
    if ( children[T.parent(v)] == 0 )
      Q.append(T.parent(v));
  }
} while ( !Q.empty() );
double_check_bottom_up_tree_traversal(T,order);
}

```

During a top-down traversal of the tree using a second queue R of nodes, all leaves are enqueued into queue Q in the required top-to-bottom, left-to-right order.

133a

⟨enqueue all leaves in top-down traversal order 133a⟩≡

```

queue<node> R;
R.append(T.root());
do {
  v = R.pop();
  forall_children(w,v) {
    if ( T.is_leaf(w) ) {
      Q.append(w);
    } else {
      R.append(w);
    }
  }
} while ( !R.empty() );

```

The following double-check of bottom-up tree traversal, although being redundant, gives some reassurance of correctness of the implementation. It verifies that $order$ is a bottom-up traversal of tree T , according to Definition 3.17.

133b

⟨double-check bottom-up tree traversal 133b⟩≡

```

void double_check_bottom_up_tree_traversal(
  const tree& T,
  node_array<int>& order)
{
  ⟨compute depth and preorder rank of all nodes 128⟩
  ⟨compute height of all nodes 134⟩
}

```

```

node v,w;
forall_nodes(v,T) {
  forall_nodes(w,T) {
    if (order[v] < order[w] ∧ height[v] > height[w])
      error_handler(1,
        "Wrong implementation of bottom-up tree traversal");
    if (height[v] ≡ height[w] ∧ depth[v] < depth[w] ∧
        order[v] ≥ order[w])
      error_handler(1,
        "Wrong implementation of bottom-up tree traversal");
    if (height[v] ≡ height[w] ∧ depth[v] ≡ depth[w] ∧
        rank[v] < rank[w] ∧ order[v] > order[w])
      error_handler(1,
        "Wrong implementation of bottom-up tree traversal");
  } } }

```

The height of all nodes in the tree is computed next, during an iterative postorder traversal of the tree. The height of a leaf is equal to zero, and the height of a nonleaf node is equal to one plus the largest height among the children of the node.

134

```

〈compute height of all nodes 134〉≡
  node_array<int> height(T);
  { list<node> L;

    stack<node> S;
    node v,w;
    S.push(T.root());
    do {
      v = S.pop();
      L.push(v); // visit node v
      forall_children(w,v)
        S.push(w);
    } while ( ¬S.empty() );

    forall(v,L) {
      if (T.is_leaf(v)) {
        height[v] = 0;
      } else {
        forall_children(w,v)
          height[v] = led_max(height[w],height[v]);
        height[v]++;
      } } }

```

Lemma 3.21. *The algorithm for bottom-up traversal of a rooted tree runs in $O(n)$ time using $O(n)$ additional space, where n is the number of nodes in the tree. The double-check of bottom-up tree traversal runs in $O(n^2)$ time using $O(n)$ additional space.*

Proof. Let $T = (V, E)$ be a tree on n nodes. All leaves of T are enqueued in $O(n)$ time using $O(n)$ additional space during a top-down traversal. Further, since each node of T is enqueueued and dequeued exactly once, the loop is executed n times and given that processing each dequeued node takes $O(1)$ time, the algorithm runs in $O(n)$ time. Further, since each node is enqueueued only once, the queue cannot ever contain more than n nodes and the algorithm uses $O(n)$ additional space.

The double-check of bottom-up tree traversal runs in $O(n^2)$ time using $O(n)$ additional space. As a matter of fact, the depth and rank of all nodes are computed in $O(n)$ time using $O(n)$ additional space during a preorder traversal of the tree, the height of all nodes is also computed in $O(n)$ time using $O(n)$ additional space during a postorder traversal of the tree, and the double-check itself is done in $O(n^2)$ time in two nested loops over all nodes of the tree. \square

3.4.1 Interactive Demonstration of Bottom-Up Traversal

The bottom-up traversal algorithm is integrated next in the interactive demonstration of graph algorithms. Once more, a simple checker for tree traversal that provides some visual reassurance consists of highlighting the nodes of the tree as the bottom-up traversal proceeds.

135

```
(demo bottom-up tree traversal 135)≡
void gw_bottom_up_tree_traversal(
    GraphWin& gw)
{
    graph& G = gw.get_graph();
    tree T(G);
    node_array<int> order(T);
    bottom_up_tree_traversal(T,order);
    {show tree traversal 119}
}
```

3.5 Applications

The different methods for exploring a tree can be used, for instance, to compute the depth and the height of the tree, as well as for producing simple tree layouts. Further applications of tree traversal will be addressed in Chap. 4.

The depth of the nodes of a rooted tree can be assigned in $O(n)$ time by a preorder traversal of the tree. The depth of the tree is equal to the depth of a deepest node in the tree.

136a ⟨preorder tree traversal 115a⟩ +≡

```
void preorder_tree_depth(
    const tree& T,
    node_array<int>& depth,
    node& deepest)
{
    deepest = T.first_node();

    stack<node> S;
    node v,w;
    S.push(T.root());

    do {
        v = S.pop();

        if ( T.is_root(v) ) {
            depth[v] = 0;
        } else {
            depth[v] = depth[T.parent(v)] + 1;
            if ( depth[v] > depth[deepest] ) deepest = v;
        }

        w = T.last_child(v);
        while ( w != nil ) {
            S.push(w);
            w = T.previous_sibling(w);
        }
    } while ( !S.empty() );
}
```

A simple checker for tree depth that provides some visual reassurance consists of highlighting the nodes and edges along the path from the root down to a deepest node in the tree; for instance, down to the first deepest node found during a preorder traversal of the tree.

136b

⟨demo preorder tree traversal 118a⟩ +≡

```
void gw_preorder_tree_depth(
    GraphWin& gw)
{
    graph& G = gw.get_graph();
    tree T(G);
    node_array<int> depth(T);
    node deepest;
    preorder_tree_depth(T,depth,deepest);

    gw.save_all_attributes();
    node v;
```

```

forall_nodes(v,T)
  gw.set_label(v,string("%i",depth[v]));

panel P;
make_proof_panel(P,
  string("The depth of the tree is %i",depth[deepest]),true);
if ( gw.open_panel(P) ) { // proof button pressed
  gw.set_color(deepest,blue);
  int num = depth[deepest];
  edge e;
  while ( !T.is_root(deepest) ) {
    e = T.first_in_edge(deepest);
    v = T.source(e);
    gw.set_color(v,blue);
    gw.set_color(e,blue);
    gw.set_label(e,string("%i",num));
    gw.set_width(e,2);
    deepest = v;
  }
  gw.wait();
}
gw.restore_all_attributes();
}

```

The height of the nodes of a tree can be assigned in $O(n)$ time by a bottom-up traversal of the tree, in which the order among leaves is not significant. The height of a leaf is equal to zero, and the height of a nonleaf node is equal to one plus the largest height among the children of the node. For a rooted tree, the height of the tree is equal to the height of the root.

(bottom-up tree traversal 132) + ≡

```

void bottom_up_tree_height(
  const tree& T,
  node_array<int>& height)
{
  queue<node> Q;
  node_array<int> children(T,0);
  edge e;
  forall_edges(e,T)
    children[T.source(e)]++;
  node v,w;
  forall_nodes(v,T)
    if ( children[v] == 0 )
      Q.append(v);
  do {
    v = Q.pop();

    if ( T.is_leaf(v) ) {
      height[v] = 0;
    } else {

```

```

forall_children(w,v)
  height[v] = leda_max(height[w],height[v]);
  height[v]++;
}

if ( ¬T.is_root(v) ) {
  w = T.parent(v);
  children[w]--;
  if ( children[w] ≡ 0 )
    Q.append(w);
}
} while ( ¬Q.empty() );
}

```

A simple checker for tree height that provides some visual reassurance consists of highlighting the nodes and edges along the path from the root over nodes of height one less than the height of the previous node in the path, down to a leaf; for instance, over the first such node found during a preorder traversal of the tree.

138

```

⟨demo bottom-up tree traversal 135⟩+≡
void gw_bottom_up_tree_height(
  GraphWin& gw)
{
  graph& G = gw.get_graph();
  tree T(G);
  node_array<int> height(T);
  bottom_up_tree_height(T,height);

  gw.save_all_attributes();
  node v;
  forall_nodes(v,T)
    gw.set_label(v,string("%i",height[v]));
  node r = T.root();

  panel P;
  make_proof_panel(P,string("The height of the tree is %i",height[r]),true);
  if ( gw.open_panel(P) ) { // proof button pressed
    gw.set_color(r,blue);

    node v = r;
    node w;
    edge e,aux;
    int num = 1;
    while ( ¬T.is_leaf(v) ) {
      forall_out_edges(aux,v) {
        w = T.opposite(v,aux);
        if ( height[w] ≡ height[v] - 1 ) {
          v = w;
          e = aux;
          break;
        }
      }
    }
  }
}

```

```

    } }
    gw.set_color(v,blue);
    gw.set_color(e,blue);
    gw.set_label(e,string("%i",num++));
    gw.set_width(e,2);
}
gw.wait();
}
gw.restore_all_attributes();
}

```

As a further application of tree traversal, consider the problem of producing a layout of a rooted tree. A straight-line *drawing* or *layout* of a graph is a mapping of nodes of the graph to distinct points in the plane, and it induces a mapping of arcs of the graph to straight-line segments, usually drawn as arrows, joining the points corresponding to their source and target nodes.

A good drawing of a graph should provide some intuition to understand the model of a problem represented by the graph. When it comes to rooted trees, one of the most important sources of intuition is given by the hierarchical structure of the tree. Therefore, rooted trees are usually drawn using a *layered* layout, in which nodes are arranged into vertical layers according to node depth and, since rooted trees in computer science are most often drawn downwards with the root at the top, each node is drawn at a vertical coordinate proportional to opposite the depth of the node. The tree drawings shown in Fig. 3.1, 3.3, 3.5, and 3.7 are all straight-line, layered drawings.

The intuition conveyed by a tree drawing can be backed up by a series of aesthetic criteria. As a matter of fact, the following aesthetic criteria are often adopted for drawing rooted trees:

1. Since rooted trees impose a distance on the nodes, no node should be closer to the root than any of its ancestors.
2. Nodes at the same depth should lie along a straight line, and the straight lines corresponding to the depths should be parallel.
3. The relative order of nodes on any level should be the same as in the top-down traversal of the tree.
4. For a binary tree, a left child should be positioned to the left of its parent node and a right child to the right.
5. A parent node should be centered over its children nodes.

6. A subtree of a given tree should be drawn the same way regardless of where it occurs in the tree.

The first three aesthetic rules guarantee that the drawing is planar, that is, that no two distinct, nonadjacent arcs cross or intersect. The sixth aesthetic rule guarantees that isomorphic subtrees are *congruent*, thus helping to visually identify repeated patterns in the drawing of a tree.

Further criteria usually adopted for drawing trees and graphs arise from physical constraints. For instance, as long as trees are drawn on media having a drawing surface of bounded width, a layered drawing of a rooted tree should be as narrow as possible.

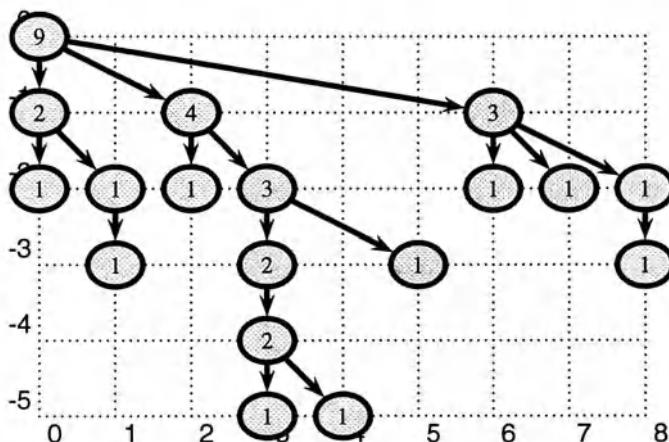


Fig. 3.9. Computing initial node coordinates in a simple layered layout for the rooted tree of Fig. 3.1. Nodes are labeled by their breadth.

Now, a simple layered layout of a rooted tree can be obtained in $O(n)$ time using $O(n)$ additional space by a series of traversals of the tree, as follows. Let the *breadth* of a rooted tree be the largest number of nodes at any depth in the tree, and let the breadth of a node in a rooted tree be the breadth of the subtree rooted at the node.

1. The depth of all nodes in a tree can be computed during either a preorder or a top-down traversal, and the breadth of all nodes can

be computed during either a postorder or a bottom-up traversal of the tree.

2. Given the depth and breadth of all nodes in the tree, the horizontal coordinate of a node can be initially set proportional to the horizontal coordinate plus the breadth of the previous sibling (or, if the node is a first child, to the horizontal coordinate of the parent), and vertical coordinates can be set proportional to opposite the depth of the nodes, during either a preorder or a top-down traversal of the tree.
3. Parent nodes can be centered over their children nodes during either a postorder or a bottom-up traversal of the tree.

The initial horizontal coordinates computed in the second step of the previous procedure are alright for leaves, as can be seen in Fig. 3.9. Horizontal coordinates are improved for nonleaves in the third step. The resulting layered tree layout is illustrated in Fig. 3.10.

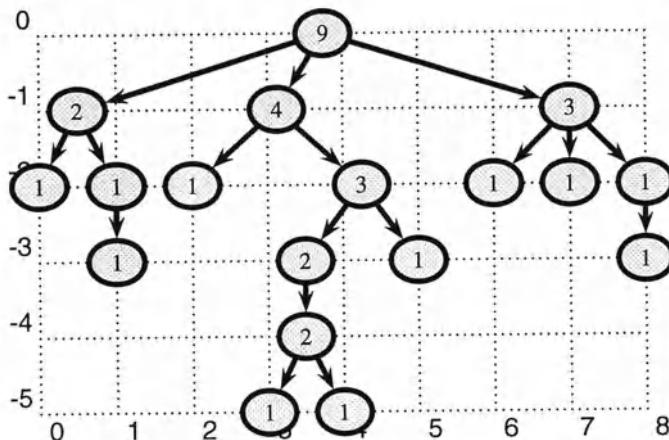


Fig. 3.10. Improving initial node coordinates by centering parent nodes over their children nodes, in the simple layered layout of Fig. 3.9. Nodes are labeled by their breadth.

Despite the simplicity of the procedure, all six aesthetic criteria for drawing rooted trees are satisfied. See the bibliographic notes below for more complex algorithms producing tidier tree drawings.

The following algorithm computes a layered layout of a rooted tree $T = (V, E)$, performing a preorder traversal, a bottom-up traversal,

a top-down traversal, and a bottom-up traversal of T to implement the previous procedure. The layout is stored in the form of horizontal coordinates $x[v]$ and vertical coordinates $y[v]$, for all nodes $v \in V$.

142a \langle tree traversal 142a $\rangle \equiv$

```
void layered_tree_layout(  
    const tree& T,  
    node_array<double>& x,  
    node_array<double>& y)  
{  
    <compute depth of all nodes 142b $\bigr>$   
    <compute breadth of all nodes 142c $\bigr>$   
    <set initial node coordinates 143 $\bigr>$   
    <center parent nodes over their children nodes 144a $\bigr>$   
}
```

The depth of all nodes in the tree is computed during an iterative preorder traversal of the tree.

142b \langle compute depth of all nodes 142b $\rangle \equiv$

```
node_array<int> depth(T);  
{ stack<node> S;  
  node v,w;  
  S.push(T.root());  
  do {  
    v = S.pop();  
  
    if ( T.is_root(v) ) {  
      depth[v] = 0;  
    } else {  
      depth[v] = depth[T.parent(v)] + 1;  
    }  
  
    w = T.last_child(v);  
    while ( w  $\neq$  nil ) {  
      S.push(w);  
      w = T.previous_sibling(w);  
    }  
  } while (  $\neg$ S.empty() );  
}
```

The breadth of all nodes in the tree is easily computed during a bottom-up traversal of the tree, in which the top-down order among leaves is not significant. The breadth of a leaf node is equal to one, and the breadth of a nonleaf node is equal to the sum of the breadth of the children of the node.

142c \langle compute breadth of all nodes 142c $\rangle \equiv$

```
node_array<int> breadth(T);
```

```

{ queue<node> Q;
  node_array<int> children(T,0);
  edge e;
  forall_edges(e,T)
    children[T.source(e)]++;
  node v;
  forall_nodes(v,T)
    if( children[v] == 0 )
      Q.append(v);
  do {
    v = Q.pop();
    if( T.is_leaf(v) )
      breadth[v] = 1;
    if( !T.is_root(v) ) {
      breadth[T.parent(v)] += breadth[v];
      children[T.parent(v)]--;
      if( children[T.parent(v)] == 0 )
        Q.append(T.parent(v));
    }
  } while( !Q.empty() );
}

```

Initial coordinates for all nodes in the tree are computed during a top-down traversal of the tree. The horizontal coordinate of the root node is set to zero. The horizontal coordinate $x(v)$ of a nonroot node v is set to either $x(v) = x(\text{parent}[v])$, if v is a first child, or $x(v) = x(u) + \text{breadth}[u]$, where node u is the previous sibling of node v . The vertical coordinate $x(v)$ of a node v is set to $y(v) = -\text{depth}[v]$.

143

```

⟨set initial node coordinates 143⟩≡
{ queue<node> Q;
  node v,w;
  Q.append(T.root());
  do {
    v = Q.pop();
    if( T.is_root(v) ) {
      x[v] = 0;
    } else {
      if( T.is_first_child(v) ) {
        x[v] = x[T.parent(v)];
      } else {
        x[v] = x[T.previous_sibling(v)]
          + breadth[T.previous_sibling(v)];
      }
    }
    y[v] = - depth[v];
    forall_children(w,v)
      Q.append(w);
  } while( !Q.empty() );
}

```

Finally, parent nodes are centered over their first and last children nodes during a bottom-up traversal of the tree, in which the top-down order among leaves is not significant.

144a ⟨center parent nodes over their children nodes 144a⟩≡

```

{ queue<node> Q;
  node_array<int> children(T,0);
  edge e;
  forall_edges(e,T)
    children[T.source(e)]++;
  node v;
  forall_nodes(v,T)
    if ( children[v] ≡ 0 )
      Q.append(v);
  do {
    v = Q.pop();
    if ( ¬T.is_leaf(v) )
      x[v] = ( x[T.first_child(v)] + x[T.last_child(v)] ) ÷ 2;
    if ( ¬T.is_root(v) ) {
      children[T.parent(v)]--;
      if ( children[T.parent(v)] ≡ 0 )
        Q.append(T.parent(v));
    }
  } while ( ¬Q.empty() );
}

```

The simple layered layout algorithm for rooted trees is integrated next in the interactive demonstration of graph algorithms. The layered layout itself constitutes a checker that provides some visual reassurance.

144b ⟨demo tree traversal 144b⟩≡

```

void gw_layered_tree_layout(
  GraphWin& gw)
{
  graph& G = gw.get_graph();
  tree T(G);

  node_array<double> xcoord(T);
  node_array<double> ycoord(T);
  layered_tree_layout(T,xcoord,ycoord);
  gw.adjust_coords_to_win(xcoord,ycoord);
  gw.set_layout(xcoord,ycoord);
  gw.display();
}

```

Summary

The most common methods of exploring a general, rooted tree were addressed in this chapter. Simple algorithms are given in detail (or proposed as exercises) for the different methods of tree traversal: depth-first prefix leftmost (preorder), depth-first prefix rightmost, depth-first postfix leftmost (postorder), depth-first postfix rightmost, breadth-first leftmost (top-down), breadth-first rightmost and, finally, bottom-up traversal. References to more space-efficient traversal algorithms on *threaded trees* are given in the bibliographic notes below, together with references to further methods for the traversal of ordered binary trees. Application of the different tree traversal methods to tree drawing is also discussed in detail.

Bibliographic Notes

All the tree traversal algorithms discussed in this chapter require, in the worst case, $O(n)$ additional space. More space-efficient algorithms are known [62, 160, 274, 303, 305] which visit all vertices of a binary tree on n nodes in $O(n)$ time using $O(1)$ additional space. Space efficiency is achieved by representing trees as *threaded trees*, and temporarily modifying the threads during the traversal.

There is still a further common method of exploring a tree. In a *symmetric order* traversal of a binary tree, also known as *inorder* traversal, the left subtree is traversed before visiting the root of the tree, followed by a traversal of the right subtree. A detailed account of the symmetric order traversal of a binary tree and its significance in solving searching and sorting problems can be found, for instance, in [193, 195, 290, 363].

A comprehensive treatment of tree and graph drawing can be found in [92]. A rather simple algorithm for drawing binary trees was given in [187] which consists of setting the vertical coordinate of a node proportional to opposite the depth of the node and the horizontal coordinate proportional to the symmetric traversal order of the node, and which despite its simplicity, still satisfies the first four aesthetic criteria. Several algorithms have since been proposed for producing tidier

drawings of rooted trees, all of them also running in $O(n)$ time using $O(n)$ additional space. The first five aesthetic criteria for drawing rooted trees are due to [364], and algorithms for drawing binary trees satisfying these criteria were presented in [343, 364]. The sixth of the aesthetic criteria was introduced in [272], together with a divide-and-conquer algorithm for drawing binary trees, which was improved in [57] in order to produce different drawings of nonisomorphic subtrees, and extended in [354] for drawing general, ordered rooted trees. The problem of producing a drawing of a binary tree satisfying all six aesthetic criteria and having minimum width can be solved in polynomial time by means of linear programming, but becomes NP-hard if a grid drawing, that is, a drawing in which all nodes have integral coordinates, is required [312].

Review Problems

3.1 Give the order in which nodes are visited during a preorder traversal and during a top-down traversal of the tree in Fig. 3.11.

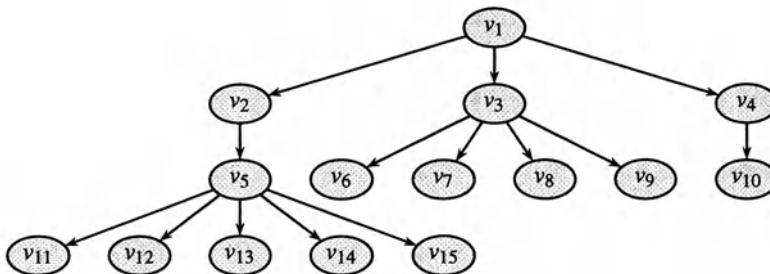


Fig. 3.11. Rooted tree for problems 3.1–3.4.

3.2 Give the order in which nodes are visited during a postorder traversal and during a bottom-up traversal of the tree in Fig. 3.11.

3.3 Show the evolution of the stack of nodes during execution of the iterative preorder traversal algorithm upon the tree in Fig. 3.11.

3.4 Show the evolution of the queue of nodes during execution of the top-down traversal algorithm upon the tree in Fig. 3.11.

3.5 Give a full binary tree on $2k + 1$ nodes whose preorder traversal and top-down traversal coincide.

3.6 Give a full binary tree on $2k + 1$ nodes whose postorder traversal and bottom-up traversal coincide.

Exercises

3.1 The preorder traversal of a tree is also called *depth-first prefix leftmost* traversal, because parents are visited before children and siblings are visited in left-to-right order. In a *depth-first prefix rightmost* traversal, parents are still visited before children but siblings are visited in right-to-left order instead. The depth-first prefix rightmost traversal of a tree is illustrated in Fig. 3.12. Adapt the iterative preorder tree traversal algorithm to perform a depth-first prefix rightmost traversal of a tree. Develop also a double-check of depth-first prefix rightmost tree traversal.

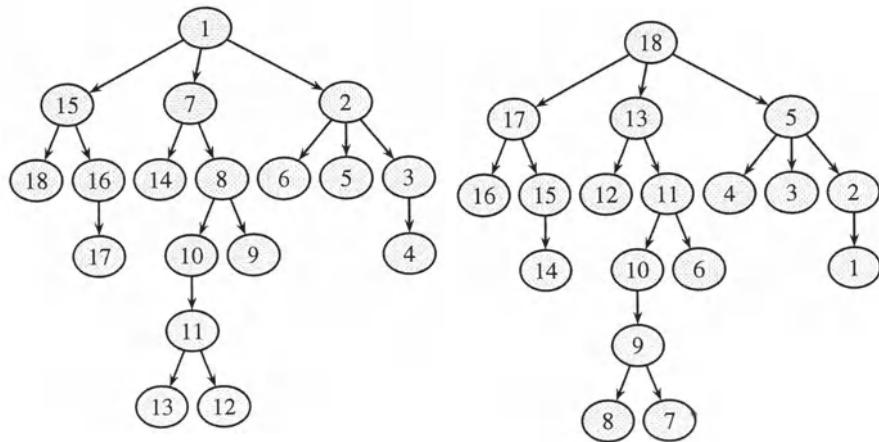


Fig. 3.12. Depth-first prefix (left) and postfix (right) rightmost traversal of a rooted tree. Nodes are numbered according to the order in which they are visited during the traversal.

3.2 The postorder traversal of a tree is also called *depth-first postfix leftmost* traversal, because parents are visited after children and siblings are visited in left-to-right order. In a *depth-first postfix rightmost*

traversal, parents are still visited after children but siblings are visited in right-to-left order instead. The depth-first postfix rightmost traversal of a tree is illustrated in Fig. 3.12. Adapt the iterative postorder tree traversal algorithm to perform a depth-first postfix rightmost traversal of a tree. Develop also a double-check of depth-first postfix rightmost tree traversal.

3.3 The depth of a rooted tree was computed in $O(n)$ time in Sect. 3.5 as a by-product of computing the depth of each node, during an iterative preorder traversal of the tree. Extend the recursive preorder traversal algorithm to compute the depth of a rooted tree.

3.4 The depth of a rooted tree was computed in $O(n)$ time in Sect. 3.5 as a by-product of computing the depth of each node, during an iterative preorder traversal of the tree. The depth of a rooted tree can also be computed in $O(n)$ time by a top-down traversal of the tree. Extend the top-down traversal algorithm to compute the depth of a rooted tree.

3.5 The height of a tree was computed in $O(n)$ time in Sect. 3.4 as a by-product of computing the height of each node, during an iterative postorder traversal of the tree and also in Sect. 3.5, during a bottom-up traversal of the tree. Extend the recursive postorder traversal algorithm to compute the height of a rooted tree.

3.6 The depth and the breadth of all nodes in a rooted tree were computed in $O(n)$ time in Sect. 3.5 during an iterative preorder and a bottom-up traversal of the tree, respectively. Show that both the depth and the breadth of all nodes in a rooted tree can be computed in $O(n)$ time during a single, recursive preorder traversal of the tree.

3.7 The inductive definition of binary trees yields a simple recursive algorithm for the symmetric order traversal of a binary tree. Give instead an iterative algorithm for the symmetric order traversal of a *full* binary tree. Develop also a double-check of symmetric order binary tree traversal.

3.8 In the iterative algorithm for the top-down traversal of a tree, a queue is used to hold children nodes while the sibling and cousin nodes of the parent node are visited, because children nodes are only accessible from their parent node. Give instead a recursive algorithm for the top-down traversal of a tree, without using an auxiliary queue.

Give also the time and space complexity of the recursive top-down tree traversal algorithm.

3.9 Implement the simple algorithm for drawing *full* binary trees, in which the vertical coordinate of a node is set proportional to opposite the depth of the node and the horizontal coordinate of a node is set proportional to the symmetric traversal order of the node.

3.10 Extend the layered tree layout algorithm to produce tidier drawings, in which siblings are set as close to each other as possible while keeping a horizontal distance between nodes of at least one. For instance, the simple layered layout of Fig. 3.10 could be improved as in Fig. 3.13. Give also the time and space complexity of the improved algorithm.

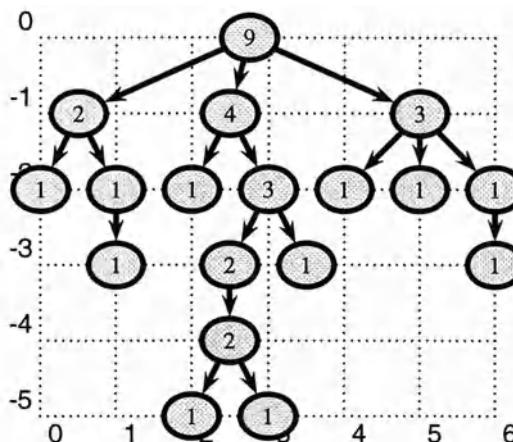


Fig. 3.13. Tidier drawing of the simple layered layout of Fig. 3.10. Nodes are labeled by their breadth.

Part III

Algorithms on Graphs

4. Tree Isomorphism

The method for tree identification seems not to offer much help toward a good algorithm for deciding whether one given rooted tree contains a subgraph identical to another given rooted tree so that roots identify. It would be very interesting to have a good algorithm for this more general identification task.

—Robert G. Busacker and Thomas L. Saaty [63]

Tree identification is the problem of determining whether two trees are isomorphic, and it is a fundamental problem with a variety of interesting applications in diverse scientific and engineering disciplines. The related problems of subtree isomorphism and maximum common subtree isomorphism generalize pattern matching in strings to trees, and find application whenever hierarchical structures, structures described by trees, need to be compared.

Combinatorial algorithms are discussed in this chapter for testing isomorphism of ordered and unordered trees, finding one or all isomorphisms of a tree as a subtree of another tree, and finding one or all maximum common subtree isomorphisms between two trees. These algorithms use the techniques given in Chap. 2, and some of them also build upon the algorithms discussed in Chap. 3 for traversing trees.

4.1 Tree Isomorphism

Tree isomorphism is the problem of determining whether a tree is isomorphic to another tree and, beside being a fundamental problem with a variety of applications, it is also the basis of simple solutions

to the more general problems of subtree isomorphism and maximum common subtree isomorphism.

Since trees can be either ordered or unordered, there are different notions of tree isomorphism.

4.1.1 Ordered Tree Isomorphism

Two ordered trees are isomorphic if there is a bijective correspondence between their node sets which preserves and reflects the structure of the ordered trees—that is, such that the node corresponding to the root of one tree is the root of the other tree, a node v_1 is the parent of a node v_2 if and only if the node corresponding to v_1 is the parent of the node corresponding to v_2 in the other tree, and a node v_3 is the next sibling of a node v_2 if and only if the node corresponding to v_3 is also the next sibling of the node corresponding to v_2 in the other tree.

In most applications, however, further information is attached to nodes and edges in the form of node and edge labels. Then, two ordered labeled trees are isomorphic if the underlying ordered trees are isomorphic and, furthermore, corresponding nodes and edges share the same label. Given that a node in a tree can have at most one incoming edge, though, the information attached to an edge can be attached to the target node of the edge instead, and edge labels are not really needed. The algorithms discussed in this chapter thus deal with trees which are either unlabeled or whose nodes are labeled.

Definition 4.1. *Two ordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ are isomorphic, denoted by $T_1 \cong T_2$, if there is a bijection $M \subseteq V_1 \times V_2$ such that $(\text{root}[T_1], \text{root}[T_2]) \in M$ and the following conditions*

- $\text{first}[v] = \text{first}[w]$ for all nonleaves $v \in V_1$ and $w \in V_2$ with $(v, w) \in M$
- $\text{next}[v] = \text{next}[w]$ for all nonlast children $v \in V_1$ and $w \in V_2$ with $(v, w) \in M$

are satisfied. In such a case, M is an ordered tree isomorphism of T_1 to T_2 .

Isomorphism expresses what, in less formal language, is meant when two trees are said to be the same tree. Two isomorphic trees may be depicted in such a way that they look very different—they are

differently labeled, perhaps also differently drawn, and it is for this reason that they look different.

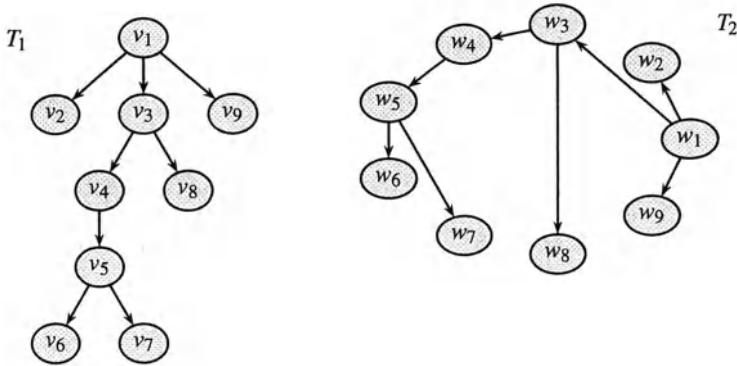


Fig. 4.1. Isomorphic ordered trees. Nodes are numbered according to the order in which they are visited during a preorder traversal.

Example 4.2. The ordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ shown in Fig. 4.1 are isomorphic. The bijection $M \subseteq V_1 \times V_2$ given by $M = \{(v_1, w_1), (v_2, w_2), (v_3, w_3), (v_4, w_4), (v_5, w_5), (v_6, w_6), (v_7, w_7), (v_8, w_8), (v_9, w_9)\}$ is an ordered tree isomorphism of T_1 to T_2 .

An Algorithm for Ordered Tree Isomorphism

A straightforward procedure for testing isomorphism of two ordered trees consists of performing a traversal of both trees using the same traversal method, and then testing whether the node mapping induced by the traversal is an isomorphism between the ordered trees.

Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be ordered trees on n nodes, let $\text{order}_1 : V_1 \rightarrow \{1, \dots, n\}$ be a traversal of T_1 , and let $\text{order}_2 : V_2 \rightarrow \{1, \dots, n\}$ be a traversal of T_2 . The node mapping $M \subseteq V_1 \times V_2$ induced by order_1 and order_2 is given by $M[v] = w$ if and only if $\text{order}_1[v] = \text{order}_2[w]$, for all nodes $v \in V_1$ and $w \in V_2$. Since both order_1 and order_2 are bijections, M is also a bijection.

The following algorithm implements the previous procedure for testing the isomorphism of two ordered trees, based on a top-down traversal of each of the trees. The bijection $M \subseteq V_1 \times V_2$ computed by

the procedure upon two isomorphic ordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ is represented by an array of nodes (of tree T_2) indexed by the nodes of tree T_1 .

Recall first that for a LEDA tree $T = (V, E)$, the label of a node $v \in V$ is denoted by $T[v]$. Unlabeled trees are dealt with in the following algorithms as labeled trees with all node labels set to *nil*, that is, undefined.

154

```
(tree isomorphism 154)≡
  bool simple_ordered_tree_isomorphism(
    const TREE<string,string>& T1,
    const TREE<string,string>& T2,
    node_array<node>& M)
  {
    int n = T2.number_of_nodes();
    if ( T1.number_of_nodes() ≠ n ) return false;

    node_array<int> order1(T1);
    node_array<int> order2(T2);
    top_down_tree_traversal(T1,order1);
    top_down_tree_traversal(T2,order2);

    array<node> disorder2(1,n);
    node v,w;
    forall_nodes(v,T2)
      disorder2[order2[v]] = v;
    forall_nodes(v,T1)
      M[v] = disorder2[order1[v]];

    forall_nodes(v,T1) {
      w = M[v];
      if ( T1[v] ≠ T2[w] ) return false;
      if ( ¬T1.is_leaf(v) ∧
          ( T2.is_leaf(w) ∨
            ( M[T1.first_child(v)] ≠ T2.first_child(w) ) ) )
        return false;
      if ( ¬T1.is_last_child(v) ∧
          ( T2.is_last_child(w) ∨
            ( M[T1.next_sibling(v)] ≠ T2.next_sibling(w) ) ) )
        return false;
    }
    return true;
  }
```

Although the previous procedure is correct, nonisomorphism of two ordered trees could be detected earlier by performing a simultaneous traversal of the two trees, instead of performing a traversal of each of the trees and then testing the induced node mapping for ordered tree isomorphism.

The following algorithm performs a simultaneous preorder traversal of the two ordered trees T_1 and T_2 , collecting also a mapping M of nodes of T_1 to nodes of T_2 .

155a $\langle \text{tree isomorphism 154} \rangle + \equiv$

```

bool ordered_tree_isomorphism(
    const TREE<string,string>& T1,
    const TREE<string,string>& T2,
    node_array<node>& M)
{
    if ( T1.number_of_nodes() ≠ T2.number_of_nodes() ) return false;
    if ( map_ordered_tree(T1,T1.root(),T2,T2.root(),M) ) {
        <double-check ordered tree isomorphism 156>
        return true;
    }
    return false;
}

```

The simultaneous preorder traversal succeeds if T_1 and T_2 are isomorphic, but otherwise fails as soon as either the structure of T_1 and the structure of T_2 differ, or corresponding nodes in T_1 and T_2 do not share the same label.

155b $\langle \text{tree isomorphism 154} \rangle + \equiv$

```

bool map_ordered_tree(
    const TREE<string,string>& T1,
    const node r1,
    const TREE<string,string>& T2,
    const node r2,
    node_array<node>& M)
{
    if ( T1[r1] ≠ T2[r2] ) return false;
    M[r1] = r2;
    int d1 = T1.number_of_children(r1);
    int d2 = T2.number_of_children(r2);
    if ( d1 ≠ d2 ) return false;
    node v1,v2;
    if ( ¬T1.is_leaf(r1) ) {
        v1 = T1.first_child(r1);
        v2 = T2.first_child(r2);
        if ( ¬map_ordered_tree(T1,v1,T2,v2,M) ) return false;
        for ( int i = 2; i ≤ d1; i++ ) {
            v1 = T1.next_sibling(v1);
            v2 = T2.next_sibling(v2);
            if ( ¬map_ordered_tree(T1,v1,T2,v2,M) ) return false;
        }
    }
    return true;
}

```

The following double-check of ordered tree isomorphism, although being redundant, gives some reassurance of the correctness of the implementation. It verifies that M is an ordered tree isomorphism of T_1 to T_2 , according to Definition 4.1.

156 ⟨double-check ordered tree isomorphism 156⟩≡

```

{ node v;
  forall_nodes(v,T1) {
    if ( M[v] ≡ nil ∨ T1[v] ≠ T2[M[v]] )
      error_handler(1,
        "Wrong implementation of ordered tree isomorphism");
    if ( ¬T1.is_leaf(v) ∧
        ( T2.is_leaf(M[v]) ∨
          ( M[T1.first_child(v)] ≠ T2.first_child(M[v]) ) ))
      error_handler(1,
        "Wrong implementation of ordered tree isomorphism");
    if ( ¬T1.is_last_child(v) ∧
        ( T2.is_last_child(M[v]) ∨
          ( M[T1.next_sibling(v)] ≠ T2.next_sibling(M[v]) ) ))
      error_handler(1,
        "Wrong implementation of ordered tree isomorphism");
  }
}
```

Lemma 4.3. *The algorithm for ordered tree isomorphism runs in $O(n)$ time using $O(n)$ additional space, where n is the number of nodes in the trees.*

Proof. Let T_1 and T_2 be ordered trees on n nodes. The algorithm makes $O(n)$ recursive calls, one for each nonleaf node of T_1 and although within a recursive call on some node, the effort spent is not bounded by a constant but is proportional to the number of children of the node, the total effort spent over all nonleaf nodes of T_1 is proportional to n and the algorithm runs in $O(n)$ time. Further, $O(n)$ additional space is used.

The double-check of ordered tree isomorphism runs in $O(n)$ time using $O(1)$ additional space. \square

4.1.2 Unordered Tree Isomorphism

Two unordered trees are isomorphic if there is a bijective correspondence between their node sets which preserves and reflects the structure of the trees—that is, such that the node corresponding to the root of one tree is the root of the other tree, and a node v_1 is the parent of

a node v_2 if and only if the node corresponding to v_1 is the parent of the node corresponding to v_2 in the other tree.

Further, two unordered labeled trees are isomorphic if the underlying unordered trees are isomorphic and corresponding nodes share the same label.

Definition 4.4. Two unordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ are isomorphic, denoted by $T_1 \cong T_2$, if there is a bijection $M \subseteq V_1 \times V_2$ such that $(\text{root}[T_1], \text{root}[T_2]) \in M$ and the following condition

- $\text{parent}[v] = \text{parent}[w]$ for all nonroots $v \in V_1$ and $w \in V_2$ with $(v, w) \in M$

is satisfied. In such a case, M is an unordered tree isomorphism, or just a tree isomorphism, of T_1 to T_2 .

Isomorphism of unordered trees expresses what, in less formal language, is meant when two unordered trees are said to be, up to permuting subtrees rooted at some node, the same tree.

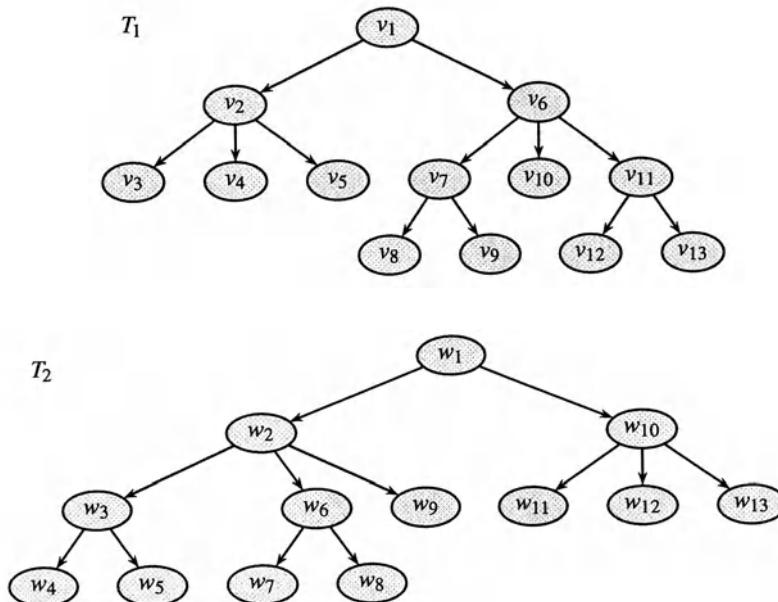


Fig. 4.2. Isomorphic trees. Nodes are numbered according to the order in which they are visited during a preorder traversal.

Example 4.5. The unordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ in Fig. 4.2 are isomorphic. The bijection $M \subseteq V_1 \times V_2$ given by $M = \{(v_1, w_1), (v_2, w_{10}), (v_3, w_{11}), (v_4, w_{12}), (v_5, w_{13}), (v_6, w_2), (v_7, w_3), (v_8, w_4), (v_9, w_5), (v_{10}, w_9), (v_{11}, w_6), (v_{12}, w_7), (v_{13}, w_8)\}$ is a tree isomorphism of T_1 to T_2 .

Remark 4.6. An ordered tree isomorphism between two ordered trees is also a tree isomorphism between the underlying unordered trees.

The tree isomorphism problem is actually harder for unordered trees than for ordered trees, because two isomorphic unordered trees may be represented in many different ways. Given an (ordered or unordered) tree, any nonidentical permutation of the subtrees rooted at any node in the tree yields, in fact, a nonisomorphic ordered tree which is isomorphic as unordered tree to the given tree. That is, many nonisomorphic ordered trees share the same underlying unordered tree.

A simple *certificate* or necessary and sufficient condition for two unordered trees to be isomorphic consists in associating a *unique* isomorphism code to each of the nodes in the two trees. Then, the two trees are isomorphic if and only if their respective root nodes share the same isomorphism code.

The isomorphism code cannot be a single short integer, though. There are $\Omega(2^n/(n\sqrt{n}))$ nonisomorphic unordered trees on n nodes, and encoding such an exponential number of trees would require integers of arbitrary precision, with $\Omega(n)$ bits. Therefore, the isomorphism code for an unordered tree on n nodes is instead a sequence of n integers in the range $1, \dots, n$.

Recall from Sect. 1.1 that in a tree $T = (V, E)$, $\text{size}[v]$ denotes the number of nodes in the subtree of T rooted at node v , for all nodes $v \in V$. Let also $p; q$ denote the concatenation of sequences p and q .

Definition 4.7. Let $T = (V, E)$ be an unordered tree on n nodes. The **isomorphism code** of the root of T is the sequence of n integers in the range $1, \dots, n$ given by $\text{code}[\text{root}[T]] = [\text{size}[\text{root}[T]]]; \text{code}[w_1]; \dots; \text{code}[w_k]$, where nodes w_1, \dots, w_k are the children of the root of T arranged in nondecreasing lexicographic order of isomorphism code. The isomorphism code of an unordered tree is the isomorphism code of the root of the tree.

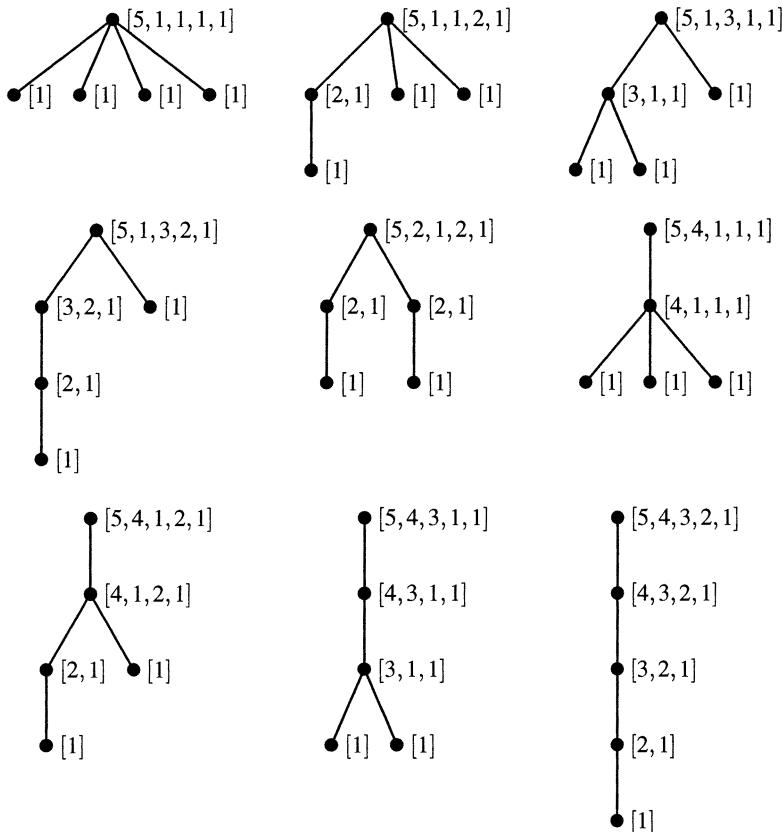


Fig. 4.3. Isomorphism codes of the nine nonisomorphic unordered trees on five nodes. The isomorphism code of each node is shown to the right of the node.

It follows from Definition 4.7 that in an unordered tree $T = (V, E)$, all leaves $v \in V$ share the same isomorphism code, $\text{code}[v] = [1]$.

Example 4.8. The unique isomorphism codes for the nine nonisomorphic unordered trees on five nodes are shown in Fig. 4.3, together with the isomorphism codes for the subtrees rooted at each of their nodes.

Theorem 4.9. Let T_1 and T_2 be unordered trees. Then, $T_1 \cong T_2$ if and only if $\text{code}[T_1] = \text{code}[T_2]$.

Proof. It is immediate from Definition 4.7 that in order for two unordered trees to be isomorphic, it is necessary that their isomorphism codes be identical. Showing sufficiency of the isomorphism codes of

two unordered trees being identical in order for the trees to be isomorphic, is equivalent to showing that, up to isomorphism, a unique unordered tree can be reconstructed from its isomorphism code.

The latter can be shown by induction on the length of the isomorphism code. Let $\text{code}[T]$ be an isomorphism code. If the length of $\text{code}[T]$ is equal to one, then it must be $\text{code}[T] = [1]$ and the unordered tree T consists of a single node.

Assume now that, up to isomorphism, a unique unordered tree T can be reconstructed from an isomorphism code $\text{code}[T]$ of length i , for all $1 \leq i \leq n$. After discarding the first integer, the rest of a sequence $\text{code}[T]$ of length $n + 1$ can be partitioned in a unique way into a sequence of disjoint sequences, because an isomorphism code is a sequence of isomorphism codes arranged in nondecreasing lexicographic order and further, the first integer in an isomorphism code is larger than the others. As a matter of fact, each of these sequences starts with the first integer in the rest of the isomorphism code which is at least as large as the first integer of the previous sequence.

Now, since none of the sequences resulting from partitioning the isomorphism code can have more than n integers, a unique, up to isomorphism, unordered tree can be reconstructed from each of them. Then, the unordered tree T consists of an additional node, the root of the tree, together with these reconstructed unordered trees as subtrees.

□

An Algorithm for Unordered Tree Isomorphism

The isomorphism code of an unordered tree can be obtained by performing a series of permutations of the children of the nodes in the tree, transforming the unordered tree into a *canonical* ordered tree in which the left-to-right order of siblings reflects the nondecreasing lexicographic order of their isomorphism codes, as illustrated in Fig. 4.4. Then, two unordered trees are isomorphic if and only if their canonical ordered trees are isomorphic.

An alternative, simple procedure for testing isomorphism of two unordered trees consists of computing the isomorphism code for each of the nodes in the trees during either a postorder traversal or a bottom-

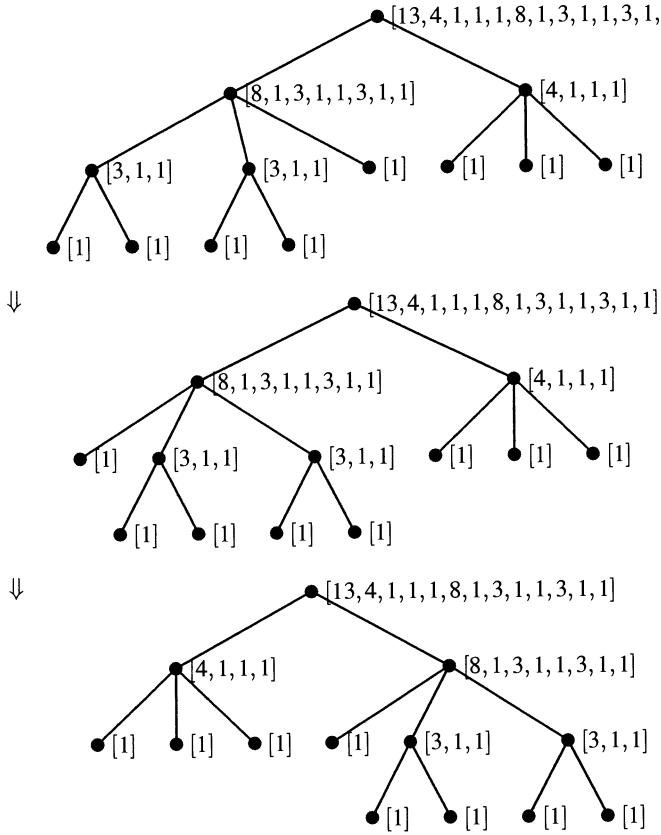


Fig. 4.4. Transformation of an unordered tree into a canonical ordered tree. The isomorphism code of each node is shown to the right of the node.

up traversal, and then comparing the isomorphism codes of their root nodes.

Remark 4.10. Correctness of the simple procedure for unordered tree isomorphism follows from Theorem 4.9.

Example 4.11. Consider the execution of the simple isomorphism procedure for unordered trees, illustrated in Fig. 4.5. During a post-order traversal of a tree, the isomorphism codes for the children of a node are computed before computing the isomorphism code of the node. For instance, the isomorphism code of the eighth node in post-order of tree T_2 , $[8, 1, 3, 1, 1, 3, 1, 1]$, is obtained by arranging in non-

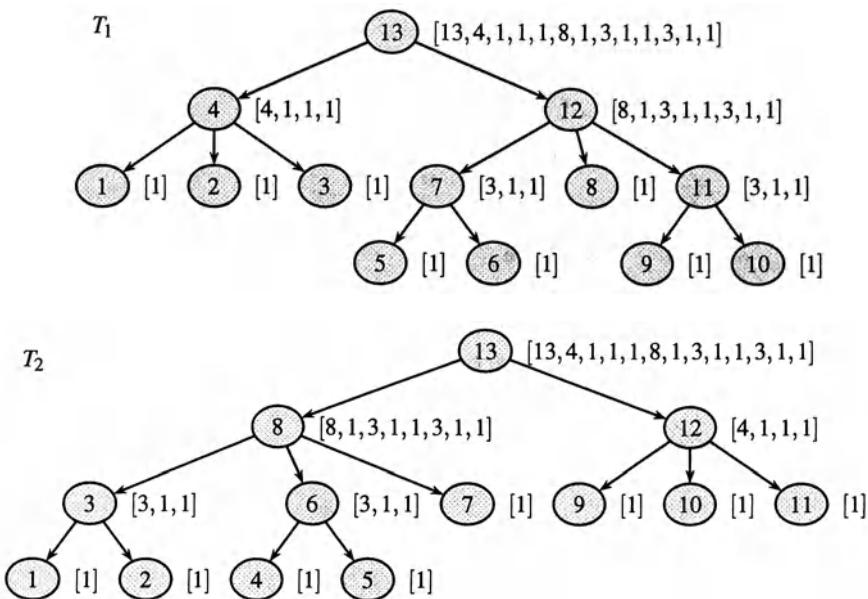


Fig. 4.5. Execution of the simple unordered tree isomorphism procedure upon the trees of Fig. 4.2. Nodes are numbered according to the order in which they are visited during a postorder traversal, and the isomorphism code of each node is also shown to the right of the node.

decreasing lexicographic order the isomorphism codes of the third, sixth, and seventh node of the tree, concatenating them, and preceding them by the size of the subtree of T_2 rooted at the eighth node.

Notice that testing unordered tree isomorphism based on isomorphism codes applies to unlabeled trees only. However, the previous isomorphism procedure for unordered unlabeled trees with n nodes can be extended to unordered trees whose nodes are labeled by integers in the range $1, \dots, n$. See Exercise 4.3.

The following algorithm determines whether two unordered trees T_1 and T_2 are isomorphic, computing isomorphism codes during a postorder traversal to implement the previous procedure. The bijection $M \subseteq V_1 \times V_2$ computed by the procedure upon two isomorphic unordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ is represented by an array of nodes (of tree T_2) indexed by the nodes of tree T_1 .

```

const TREE<string,string>& T1,
const TREE<string,string>& T2,
node_array<node>& M)
{
  if ( T1.number_of_nodes() ≠ T2.number_of_nodes() ) return false;

  node_array<list<int> > code1(T1);
  node_array<list<int> > code2(T2);

  ⟨assign isomorphism codes to all nodes of T1 and T2 in postorder 163b⟩

  if ( code1[T1.root()] ≡ code2[T2.root()] ) {
    ⟨build tree isomorphism mapping 165a⟩
    ⟨double-check unordered tree isomorphism 166⟩
    return true;
  } else {
    return false; // T1 and T2 are not isomorphic
  } }
}

```

The following procedure performs an iterative postorder traversal of an unordered tree T with the help of a stack of nodes, and builds a list L of the nodes of T in the order in which they are visited during the traversal.

163a

```

⟨tree isomorphism 154⟩ +≡
void postorder_tree_list_traversal(
  const TREE<string,string>& T,
  list<node>& L)
{
  L.clear();
  stack<node> S;
  S.push(T.root());
  node v,w;
  do {
    v = S.pop();
    L.push(v);
    forall_children(w,v)
      S.push(w);
  } while ( ¬S.empty() );
}

```

Isomorphism codes are computed by the following procedure for all nodes of T_1 and T_2 , during a postorder traversal of the two trees. Although the isomorphism code of a node could be computed by performing a *destructive* concatenation of the isomorphism codes of the children of the node, the isomorphism codes of all nodes are still needed in order to compute the bijection $M \subseteq V_1 \times V_2$ corresponding to an unordered isomorphism of tree $T_1 = (V_1, E_1)$ to tree $T_2 = (V_2, E_2)$.

163b ⟨assign isomorphism codes to all nodes of T1 and T2 in postorder 163b⟩≡

```

{ node v,w;
array<int> A;
list<array<int>> L;
int code;

list<node> L1;
postorder_tree_list_traversal(T1,L1);

forall(v,L1) {
  if ( T1.is_leaf(v) ) {
    code1[v].append(1);
  } else {
    L.clear();
    code = 1;
    forall_children(w,v) {
      code += code1[w].head();
      list_to_array(code1[w],A);
      L.append(A);
    }
    radix_sort(L);
    code1[v].append(code);
    forall(A,L)
      forall(code,A)
        code1[v].append(code);
  } }

list<node> L2;
postorder_tree_list_traversal(T2,L2);

forall(v,L2) {
  if ( T2.is_leaf(v) ) {
    code2[v].append(1);
  } else {
    L.clear();
    code = 1;
    forall_children(w,v) {
      code += code2[w].head();
      list_to_array(code2[w],A);
      L.append(A);
    }
    radix_sort(L);
    code2[v].append(code);
    forall(A,L)
      forall(code,A)
        code2[v].append(code);
  } } }
```

If the unordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ are isomorphic, the corresponding bijection $M \subseteq V_1 \times V_2$ can be computed from the isomorphism codes of the nodes of T_1 and T_2 . Node $v \in V_1$ is

mapped to node $w \in T_2$, that is, $(v, w) \in M$ if and only if $\text{code}_1[v] = \text{code}_2[w]$ and $(\text{parent}[v], \text{parent}[w]) \in M$.

165a ⟨build tree isomorphism mapping 165a⟩≡

```
{ node v = T1.root();
  node w = T2.root();
  M[v] = w;
  node_array<bool> mapped_to(T2,false);
  list<node> L1;
  preorder_tree_list_traversal(T1,L1);
  L1.pop(); // node v already mapped
  forall(v,L1) {
    forall.children(w,M[T1.parent(v)]) {
      if ( code1[v] ≡ code2[w] ∧ ¬mapped_to[w] ∧
          ( T2.is_root(w) ∨ M[T1.parent(v)] ≡ T2.parent(w) ) ) {
        M[v] = w;
        mapped_to[w] = true;
        break;
      } } } }
```

The following procedure performs an iterative preorder traversal of an unordered tree T with the help of a stack of nodes, and builds a list L of the nodes of T in the order in which they are visited during the traversal.

165b ⟨tree isomorphism 154⟩+≡

```
void preorder_tree_list_traversal(
  const TREE<string,string>& T,
  list<node>& L)
{
  L.clear();
  stack<node> S;
  S.push(T.root());
  node v,w;
  do {
    v = S.pop();
    L.push(v);
    w = T.last_child(v);
    while ( w ≠ nil ) {
      S.push(w);
      w = T.previous_sibling(w);
    }
  } while ( ¬S.empty() );
  L.reverse();
}
```

The following double-check of unordered tree isomorphism, although being redundant, gives some reassurance of the correctness of

the implementation. It verifies that M is an unordered tree isomorphism of T_1 to T_2 , according to Definition 4.4.

166

`(double-check unordered tree isomorphism 166)≡`

```

{ node v;
forall_nodes(v,T1) {
  if( M[v] ≡ nil ∨ T1[v] ≠ T2[M[v]] )
    error_handler(1,
      "Wrong implementation of unordered tree isomorphism");
  if( ¬T1.is_root(v) ∧
      ( T2.is_root(M[v]) ∨
        ( M[T1.parent(v)] ≠ T2.parent(M[v]) ) ) )
    error_handler(1,
      "Wrong implementation of unordered tree isomorphism");
}
}
```

Theorem 4.12. *The algorithm for unordered tree isomorphism runs in $O(n^2)$ time using $O(n)$ additional space, where n is the number of nodes in the trees.*

Proof. Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be unordered trees on n nodes. The effort spent in sorting the isomorphism codes of the children of a node in lexicographic order is proportional not only to the number of children of the node, but also to the length of the isomorphism codes to be sorted, that is, to the size of the subtree rooted at each of the children nodes.

Now, radix sorting a list of arrays of integers takes time linear in the total length of the arrays, even if the arrays to be sorted are of different lengths. Then, the total effort spent in radix sorting the isomorphism codes of the children of all nodes in the trees is proportional to the sum over all nodes in the trees, of the size of the subtree rooted at the node. This sum is bounded by the recurrence $T(1) = 1$, $T(n) = T(n-1) + n$, which solves to give $T(n) = O(n^2)$.

Therefore, the algorithm runs in $O(n^2)$ time using $O(n)$ additional space. (Taking the construction of the bijection $M \subseteq V_1 \times V_2$ into account, though, the algorithm runs in $O(n^2)$ time using $O(n^2)$ additional space.) The double-check of unordered tree isomorphism runs in $O(n)$ time using $O(1)$ additional space. \square

References to more efficient algorithms for unordered tree isomorphism are given in the bibliographic notes, at the end of the chapter.

A few implementation details still need to be filled in, though. Radix sorting a list of isomorphism codes cannot be performed straight upon the list itself, because isomorphism codes are themselves lists (of integers) but radix sort requires direct access to each of the integers in an isomorphism code. Therefore, isomorphism codes are converted to arrays of integers before radix sorting them, and converted back to lists of integers thereafter. The following procedure implements the conversion of a list of elements to an array of elements of the same type.

167a ⟨tree isomorphism 154⟩+≡

```
template<class E>
void list_to_array(
    const list<E>& L,
    array<E>& A)
{
    A.resize(1,L.length());
    int i = 1;
    E x;
    forall(x,L) A[i++] = x;
}
```

Most LEDA data types have a default linear order, that is, a reflexive, antisymmetric, and transitive relation defined, but there is no default linear order defined in LEDA for comparing two lists of elements.

The following procedure defines a default linear order for LEDA lists of integers. Given two lists of integers L_1 and L_2 , it returns -1 if L_1 is smaller in lexicographic order than L_2 , 0 if the lists L_1 and L_2 are identical, and 1 if L_1 is larger in lexicographic order than L_2 .

167b ⟨tree isomorphism 154⟩+≡

```
int compare(
    const list<int>& L1,
    const list<int>& L2)
{
    list_item p = L1.first();
    list_item q = L2.first();
    while (p ≠ nil ∧ q ≠ nil) {
        if (L1[p] < L2[q]) return -1;
        if (L1[p] > L2[q]) return 1;
        p = L1.succ(p);
        q = L2.succ(q);
    }
    if (q ≠ nil) return -1; // p == nil && q != nil
    if (p ≠ nil) return 1; // p != nil && q == nil
```

```

return 0; // p == nil && q == nil
}

```

Now, the following Boolean comparison operator for lists of integers makes it possible to test equality of isomorphism codes.

168a ⟨tree isomorphism 154⟩ +≡

```

bool operator≡(
    const list<int>& L1,
    const list<int>& L2)
{
    return compare(L1,L2) ≡ 0;
}

```

Last, but not least, LEDA does not provide an implementation of radix sort, although it does provide an efficient implementation of bucket sort. A simple LEDA implementation of radix sort is given in Sect. A.5.

4.1.3 Interactive Demonstration of Tree Isomorphism

The algorithms for ordered and unordered tree isomorphism are both integrated next in the interactive demonstration of graph algorithms. A simple checker for isomorphic trees that provides some visual reassurance consists of redrawing one of the trees according to the tree isomorphism found, to match the layout of the other tree.

168b

```

⟨demo tree isomorphism 168b⟩ ≡
    void gw_ordered_tree_isomorphism(
        GraphWin& gwI)
    {
        GRAPH<string,string>& G1 = gwI.get_graph();
        TREE<string,string> TI(G1); // tree T1(G1);

        GRAPH<string,string> G2;
        GraphWin gw2(G2,500,500, "Ordered Tree Isomorphism");
        gw2.display();
        gw2.message("Enter second tree. Press done when finished");
        gw2.edit();
        gw2.del_message();

        TREE<string,string> T2(G2); // tree T2(G2);

        node_array<node> M(T1);
        bool isomorph = ordered_tree_isomorphism(T1,T2,M);

        panel P;

```

```

if ( isomorph ) {
    make_proof_panel(P, "These trees are isomorphic",true);
    if ( gw1.open_panel(P) ) { // proof button pressed
        gw1.save_all_attributes();
        node_array<point> pos1(T1);
        node_array<point> pos2(T2);
        ⟨show tree isomorphism 170⟩
        gw1.wait();
        gw1.restore_all_attributes();
    } else {
        make_proof_panel(P, "These trees are \\\red not isomorphic",false);
        gw1.open_panel(P);
    }
}

```

169

```

⟨demo tree isomorphism 168b⟩+≡
void gw_unordered_tree_isomorphism(GraphWin& gw1)
{
    GRAPH<string,string>& GI = gw1.get_graph();
    TREE<string,string> T1(GI);

    GRAPH<string,string> G2;
    GraphWin gw2(G2,500,500,"Unordered Tree Isomorphism");
    gw2.display();
    gw2.message("Enter second tree. Press done when finished");
    gw2.edit();
    gw2.del_message();

    TREE<string,string> T2(G2);

    node_array<node> M(T1);
    bool isomorph = unordered_tree_isomorphism(T1,T2,M);

    panel P;
    if ( isomorph ) {
        make_proof_panel(P, "These trees are isomorphic",true);
        if ( gw1.open_panel(P) ) { // proof button pressed
            gw1.save_all_attributes();
            node_array<point> pos1(T1);
            node_array<point> pos2(T2);
            ⟨show tree isomorphism 170⟩
            gw1.wait();
            gw1.restore_all_attributes();
        } else {
            make_proof_panel(P, "These trees are \\\red not isomorphic",false);
            gw1.open_panel(P);
        }
    }
}

```

Tree T_2 is redrawn according to the tree isomorphism mapping M found, to match the layout of tree T_1 , by updating the position of every node $M[v] \in V_2$ in the layout of T_2 to be identical with the position of

node $v \in V_1$ in the layout of T_1 , and removing any edge bends in the layout of both trees.

170

```
<show tree isomorphism 170>≡
  gw1.get_position(pos1);
  gw1.set_layout(pos1); // remove edge bends
  gw2.get_position(pos2);
  gw2.set_layout(pos2); // remove edge bends
  node v;
  forall_nodes(v,T1)
    pos2[M[v]] = pos1[v];
  gw2.set_position(pos2);
  gw2.set_layout(pos2);
```

4.2 Subtree Isomorphism

An important generalization of tree isomorphism is known as subtree isomorphism. Subtree isomorphism is the problem of determining whether a tree is isomorphic to a subtree of another tree, and is a fundamental problem with a variety of applications in engineering and life sciences. Trees can be either ordered or unordered and, further, there are several different notions of subtree.

In the most general sense, a subtree of a given unordered tree is a connected subgraph of the tree, while in a more restricted sense, a *bottom-up subtree* of a given unordered tree is the whole subtree rooted at some node of the tree. Further, a connected subgraph is called a *top-down subtree* if the parent of all nodes in the subtree also belongs to the subtree.

Definition 4.13. Let $T = (V, E)$ be an unordered tree, and let $W \subseteq V$. Let also $\text{children}[v]$ denote the set of children of node v , for all nodes $v \in V$. An unordered tree (W, S) is a **subtree** of T if $S \subseteq E$. A subtree (W, S) is a **top-down subtree** if $\text{parent}[v] \in W$, for all nodes $v \in W$ different from the root, and it is a **bottom-up subtree** if $\text{children}[v] \subseteq W$, for all nonleaves $v \in W$.

For ordered trees, the different notions of subtree become those of *leftmost* subtrees. An ordered subtree is a subtree in which the previous sibling (if any) of each of the nodes in the subtree also belongs to

the subtree, and the same holds for top-down and bottom-up ordered subtrees.

Recall from Sect. 1.1 that in an ordered tree $T = (V, E)$, $\text{previous}[v]$ denotes the previous sibling of node v , for all nonfirst children nodes $v \in V$.

Definition 4.14. Let $T = (V, E)$ be an ordered tree, and let $W \subseteq V$. Let also $\text{children}[v]$ denote the set of children of node v , for all nodes $v \in V$. An ordered tree (W, S) is an **ordered subtree** of T if $S \subseteq E$ and furthermore, $\text{previous}[v] \in W$ for all nonfirst children nodes $v \in W$. An ordered subtree (W, S) is a **top-down ordered subtree** if $\text{parent}[v] \in W$, for all nodes $v \in W$ different from the root, and it is a **bottom-up ordered subtree** if $\text{children}[v] \subseteq W$, for all nonleaves $v \in W$.

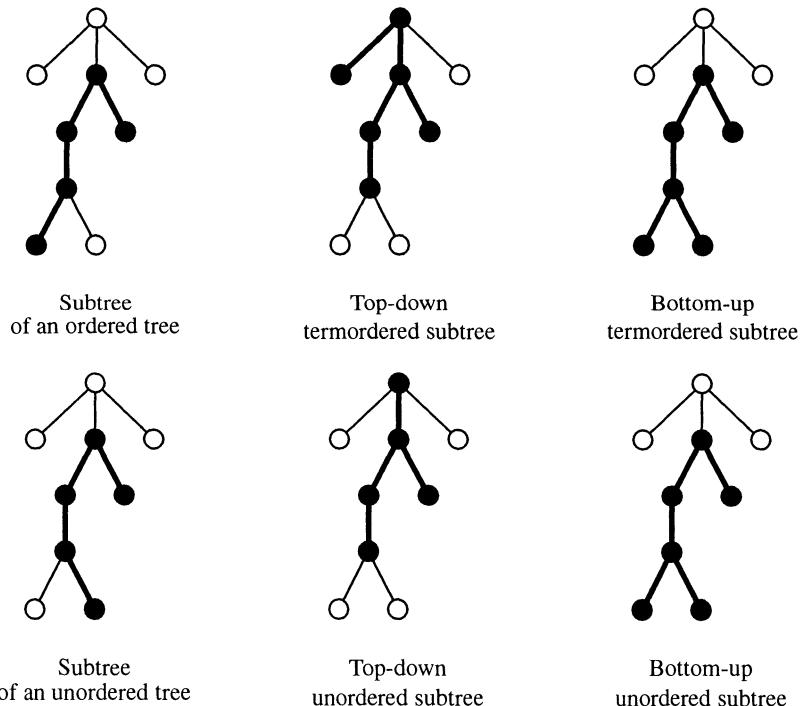


Fig. 4.6. A connected subgraph, a top-down subtree, and a bottom-up subtree of an ordered and an unordered tree.

The difference between a connected subgraph, a top-down subtree, and a bottom-up subtree of an ordered and an unordered tree is illustrated in Fig. 4.6. Notice that a subtree (connected subgraph) of an ordered tree is also a subtree of the underlying unordered tree, and a top-down subtree of an ordered tree is also a top-down subtree of the underlying unordered tree, but the converse does not always hold. The notion of bottom-up subtree, however, is the same for both ordered and unordered trees, because a bottom-up subtree of a given tree contains all of the nodes in the subtree rooted at some node of the tree.

Remark 4.15. A top-down subtree of a tree is a connected subgraph of the tree which is rooted at the root of the tree.

Because of the previous remark, connected subgraph isomorphism problems on trees can be reduced to top-down subtree isomorphism problems upon the subtrees (of enough size and height) rooted at each node of the given tree. Therefore, more attention will be paid in the rest of this chapter to top-down and bottom-up subtree isomorphism problems.

4.2.1 Top-Down Subtree Isomorphism

An ordered tree T_1 is isomorphic to a top-down subtree of another ordered tree T_2 if there is an injective correspondence of the node set of T_1 into the node set of T_2 which preserves the ordered structure of T_1 —that is, such that the node of T_2 corresponding to the root of T_1 is the root of T_2 , the node of T_2 corresponding to the parent in T_1 of a node v_1 is the parent in T_2 of the node corresponding to v_1 , and the node of T_2 corresponding to the next sibling in T_1 of a node v_2 is also the next sibling in T_2 of the node corresponding to v_2 .

Definition 4.16. An ordered tree $T_1 = (V_1, E_1)$ is **isomorphic to a top-down subtree** of another ordered tree $T_2 = (V_2, E_2)$ if there is an injection $M \subseteq V_1 \times V_2$ such that the following conditions

- $(\text{root}[T_1], \text{root}[T_2]) \in M$
- $(v, w) \in M$ for all nonleaves $v \in V_1$ and $w \in V_2$ such that $(\text{first}[v], \text{first}[w]) \in M$

- $(v, w) \in M$ for all nonlast children $v \in V_1$ and $w \in V_2$ such that $(\text{next}[v], \text{next}[w]) \in M$

are satisfied. In such a case, M is a top-down ordered subtree isomorphism of T_1 into T_2 .

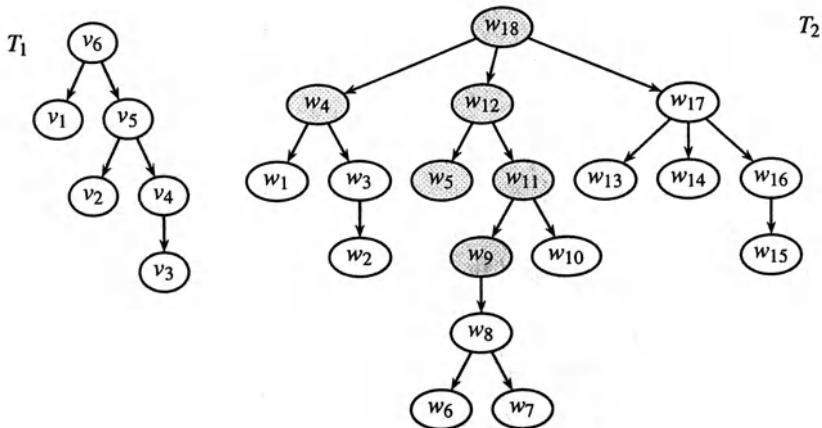


Fig. 4.7. An ordered tree which is isomorphic to a top-down subtree of another ordered tree. Nodes are numbered according to the order in which they are visited during a postorder traversal. The subtree of T_2 which is isomorphic to T_1 is shown highlighted.

Example 4.17. The ordered tree $T_1 = (V_1, E_1)$ in Fig. 4.7 is isomorphic to a top-down subtree of the ordered tree $T_2 = (V_2, E_2)$. The injection $M \subseteq V_1 \times V_2$ given by $M = \{(v_1, w_4), (v_2, w_5), (v_3, w_9), (v_4, w_{11}), (v_5, w_{12}), (v_6, w_{18})\}$ is a top-down ordered subtree isomorphism of T_1 into T_2 .

An Algorithm for Top-Down Ordered Subtree Isomorphism

A top-down subtree isomorphism of an ordered tree T_1 into another ordered tree T_2 can be obtained by performing a simultaneous traversal of the two trees, in much the same way as done in Sect. 4.1 for testing isomorphism of two ordered trees.

The following algorithm performs a simultaneous preorder traversal of the ordered tree T_1 and a top-down subtree of the ordered tree T_2 , also collecting a mapping M of nodes of T_1 to nodes of T_2 .

173 $\langle \text{subtree isomorphism 173} \rangle \equiv$

```

bool top_down_ordered_subtree_isomorphism(
    const TREE<string,string>& T1,
    const TREE<string,string>& T2,
    node_array<node>& M)
{
    if ( T1.number_of_nodes() > T2.number_of_nodes() ) return false;
    if ( map_ordered_subtree(T1,T1.root(),T2,T2.root(),M) ) {
        (double-check top-down ordered subtree isomorphism 174b)
        return true;
    }
    return false;
}

```

The simultaneous preorder traversal succeeds if T_1 is isomorphic to a top-down ordered subtree of T_2 , but otherwise fails as soon as the structure of T_1 and the top-down ordered structure of T_2 differ.

174a $\langle \text{tree isomorphism 154} \rangle + \equiv$

```

bool map_ordered_subtree(
    const TREE<string,string>& T1,
    const node r1,
    const TREE<string,string>& T2,
    const node r2,
    node_array<node>& M)
{
    if ( T1[r1] ≠ T2[r2] ) return false;
    M[r1] = r2;
    int d1 = T1.number_of_children(r1);
    int d2 = T2.number_of_children(r2);
    if ( d1 > d2 ) return false;
    node v1,v2;
    if ( ¬T1.is_leaf(r1) ) {
        v1 = T1.first_child(r1);
        v2 = T2.first_child(r2);
        if ( ¬map_ordered_subtree(T1,v1,T2,v2,M) ) return false;
        for ( int i = 2; i ≤ d1; i++ ) {
            v1 = T1.next_sibling(v1);
            v2 = T2.next_sibling(v2);
            if ( ¬map_ordered_subtree(T1,v1,T2,v2,M) ) return false;
        }
    }
    return true;
}

```

The following double-check of top-down ordered subtree isomorphism, although being redundant, gives some reassurance of the correctness of the implementation. It verifies that M is a top-down ordered subtree isomorphism of T_1 into T_2 , according to Definition 4.16.

174b $\langle \text{double-check top-down ordered subtree isomorphism 174b} \rangle \equiv$

```

{ node v;
  if ( M[T1.root()] ≠ T2.root() )
    error_handler(1,
      "Wrong implementation of top-down subtree isomorphism");
  forall_nodes(v,T1) {
    if ( M[v] ≡ nil ∨ T1[v] ≠ T2[M[v]] )
      error_handler(1,
        "Wrong implementation of top-down subtree isomorphism");
    if ( ¬T1.is_leaf(v) ∧
        ( T2.is_leaf(M[v]) ∨
          ( M[T1.first_child(v)] ≠ T2.first_child(M[v]) ) ) )
      error_handler(1,
        "Wrong implementation of top-down subtree isomorphism");
    if ( ¬T1.is_last_child(v) ∧
        ( T2.is_last_child(M[v]) ∨
          ( M[T1.next_sibling(v)] ≠ T2.next_sibling(M[v]) ) ) )
      error_handler(1,
        "Wrong implementation of top-down subtree isomorphism");
  }
}

```

Lemma 4.18. *Let T_1 and T_2 be ordered trees with respectively n_1 and n_2 nodes, where $n_1 \leq n_2$. The algorithm for top-down ordered subtree isomorphism runs in $O(n_1)$ time using $O(n_1)$ additional space.*

Proof. Let T_1 and T_2 be ordered trees with respectively n_1 and n_2 nodes, where $n_1 \leq n_2$. The algorithm makes $O(n_1)$ recursive calls, one for each nonleaf node of T_1 and although within a recursive call on some node, the effort spent is not bounded by a constant but is proportional to the number of children of the node, the total effort spent over all nonleaf nodes of T_1 is proportional to n_1 and the algorithm runs in $O(n_1)$ time. Further, $O(n_1)$ additional space is used.

The double-check of ordered tree isomorphism runs in $O(n_1)$ time using $O(1)$ additional space. \square

4.2.2 Top-Down Unordered Subtree Isomorphism

An unordered tree T_1 is isomorphic to a top-down subtree of another unordered tree T_2 if there is an injective correspondence of the node set of T_1 into the node set of T_2 which preserves the structure of T_1 —that is, such that the node of T_2 corresponding to the root of T_1 is the root of T_2 , and the node of T_2 corresponding to the parent in T_1 of a node v_1 is the parent in T_2 of the node corresponding to v_1 .

Definition 4.19. An unordered tree $T_1 = (V_1, E_1)$ is **isomorphic to a top-down subtree** of another unordered tree $T_2 = (V_2, E_2)$ if there is an injection $M \subseteq V_1 \times V_2$ such that the following conditions

- $(\text{root}[T_1], \text{root}[T_2]) \in M$
- $(\text{parent}[v], \text{parent}[w]) \in M$ for all nonroot nodes $v \in V_1$ and $w \in V_2$ with $(v, w) \in M$

are satisfied. In such a case, M is a top-down unordered subtree isomorphism of T_1 into T_2 .

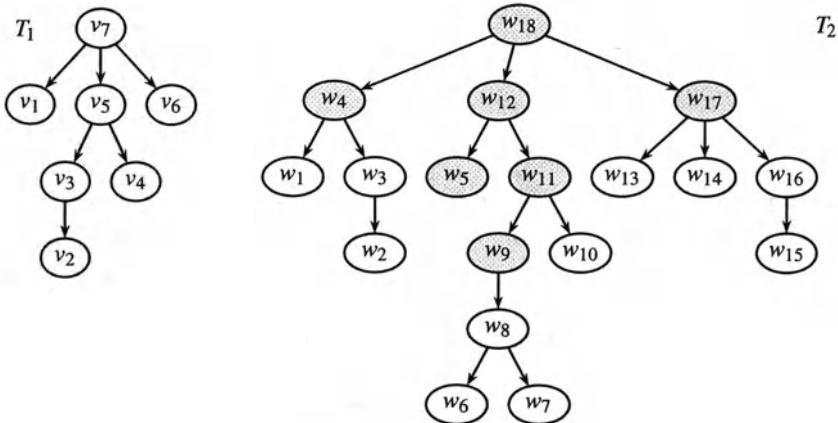


Fig. 4.8. An unordered tree which is isomorphic to a top-down subtree of another unordered tree. Nodes are numbered according to the order in which they are visited during a postorder traversal. The subtree of T_2 which is isomorphic to T_1 is shown highlighted.

Example 4.20. The unordered tree $T_1 = (V_1, E_1)$ in Fig. 4.8 is isomorphic to a top-down subtree of the unordered tree $T_2 = (V_2, E_2)$. The injection $M \subseteq V_1 \times V_2$ given by $M = \{(v_1, w_4), (v_2, w_9), (v_3, w_{11}), (v_4, w_5), (v_5, w_{12}), (v_6, w_{17}), (v_7, w_{18})\}$ is a top-down unordered subtree isomorphism of T_1 into T_2 .

An Algorithm for Top-Down Unordered Subtree Isomorphism

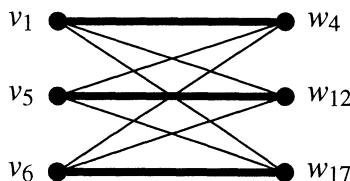
A top-down subtree isomorphism of an unordered tree T_1 into another unordered tree T_2 can be constructed from subtree isomorphisms of

each of the subtrees of T_1 rooted at the children of node v into each of the subtrees of T_2 rooted at the children of node w , provided that these isomorphic subtrees do not overlap. This decomposition property allows application of the divide-and-conquer technique, yielding a simple recursive algorithm.

Consider first the problem of determining whether or not there is a top-down unordered subtree isomorphism of T_1 into T_2 , postponing for a while the discussion about construction of an actual subtree isomorphism mapping M of T_1 into T_2 . If node v is a leaf in T_1 , then it can be mapped to node w in T_2 , that is, (v, w) belongs to some top-down unordered subtree isomorphism mapping of T_1 into T_2 , provided that $T_1[v] = T_2[w]$.

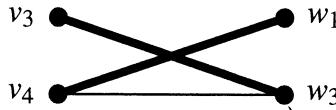
Otherwise, let p be the number of children of node v in T_1 , and let q be the number of children of node w in T_2 . Let also v_1, \dots, v_p and w_1, \dots, w_q be the children of nodes v and w , respectively. Build a bipartite graph $G = (\{v_1, \dots, v_p\}, \{w_1, \dots, w_q\}, E)$ on $p + q$ vertices, with an edge $(v_i, w_j) \in E$ if and only if node v_i can be mapped to node w_j . Then, node v can be mapped to node w if and only if G has a maximum cardinality bipartite matching with p edges.

Example 4.21. Consider the execution of the top-down unordered subtree isomorphism procedure, upon the unordered trees of Fig. 4.8. In order to decide if node v_7 can be mapped to node w_{18} , that is, if T_1 is isomorphic to a top-down unordered subtree of T_2 , the following maximum cardinality bipartite matching problem is solved:



Since there is a solution of cardinality equal to 3, the number of children of node v_7 , node v_7 can, in fact, be mapped to node w_{18} and there is a top-down unordered subtree isomorphism of T_1 into T_2 . However, stating this bipartite matching problem involves (recursively) solving further maximum cardinality bipartite matching problems.

First, in order to decide if node v_5 can be mapped to node w_4 , the following maximum cardinality bipartite matching problem is also solved



which, in turn, requires solving the following maximum cardinality bipartite matching problem, in order to decide if node v_3 can be mapped to node w_3 :



Since the latter (trivial) bipartite matching problem has a solution of cardinality equal to 1, the number of children of node v_3 , node v_3 can be mapped to node w_3 and, furthermore, node v_5 can be mapped to node w_4 , because the former bipartite matching problem has a solution of cardinality equal to 2, the number of children of node v_5 .

Next, in order to decide if node v_5 can be mapped to node w_{12} , the following maximum cardinality bipartite matching problem is also solved



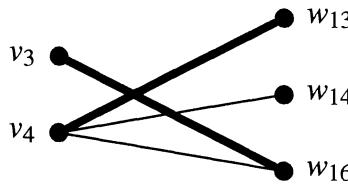
which, in turn, requires solving the following maximum cardinality bipartite matching problem, in order to decide if node v_3 can be mapped to node w_{11} :



Since the latter (trivial) bipartite matching problem has a solution of cardinality equal to 1, the number of children of node v_3 , node v_3 can

be mapped to node w_{11} and, furthermore, node v_5 can be mapped to node w_{12} , because the former bipartite matching problem has a solution of cardinality equal to 2, the number of children of node v_5 .

Finally, in order to decide if node v_5 can be mapped to node w_{17} , the following maximum cardinality bipartite matching problem is also solved



which, in turn, requires solving the following maximum cardinality bipartite matching problem, in order to decide if node v_3 can be mapped to node w_{16} :



Again, since the latter (trivial) bipartite matching problem has a solution of cardinality equal to 1, the number of children of node v_3 , node v_3 can be mapped to node w_{16} and, furthermore, node v_5 can be mapped to node w_{17} , because the former bipartite matching problem has a solution of cardinality equal to 2, the number of children of node v_5 .

Now, the previous decision procedure for top-down unordered subtree isomorphism can be extended with the construction of an actual unordered subtree isomorphism mapping M of T_1 into T_2 , based on the following result. Notice first that the solution to a maximum cardinality bipartite matching problem on a subtree (W_1, S_1) of T_1 and a subtree (W_2, S_2) of T_2 is a set of edges of the corresponding bipartite graph, that is, a set of ordered pairs of nodes $B \in W_1 \times W_2$. Since $W_1 \subseteq V_1$ and $W_2 \subseteq V_2$, it also holds that $B \in V_1 \times V_2$.

Lemma 4.22. *Let $T_1 = (V_1, E_1)$ be an unordered tree isomorphic to a top-down subtree of an unordered tree $T_2 = (V_2, E_2)$, and let $B \subseteq V_1 \times V_2$ be the solutions to all the maximum cardinality bipartite matching*

problems solved during the top-down unordered subtree isomorphism procedure upon T_1 and T_2 . Then, there is a unique top-down unordered subtree isomorphism $M \in V_1 \times V_2$ such that $M \subseteq B$.

Proof. Let $T_1 = (V_1, E_1)$ be an unordered tree isomorphic to a top-down subtree of an unordered tree $T_2 = (V_2, E_2)$, and let $B \subseteq V_1 \times V_2$ be the corresponding solutions of maximum cardinality bipartite matching problems. Showing existence and uniqueness of a top-down unordered subtree isomorphism $M \subseteq B$ is equivalent to showing that, for each node $v \in V_1$ with $(\text{parent}(v), z) \in B$ for some node $z \in V_2$, there is a unique $(v, w) \in B$ such that $\text{parent}(w) = z$, because a unique top-down subtree of T_2 isomorphic to T_1 can then be reconstructed by order of nondecreasing depth. Therefore, it suffices to show the weaker condition that, for all $(v, w_1), (v, w_2) \in B$ with $w_1 \neq w_2$, it holds that $\text{parent}(w_1) \neq \text{parent}(w_2)$.

Let $(v, w_1), (v, w_2) \in B$ with $w_1 \neq w_2$. Then, (v, w_1) and (v, w_2) must belong to the solution to different bipartite matching problems, for otherwise the corresponding edges in the bipartite graph would not be independent. But if (v, w_1) and (v, w_2) belong to the solution to different bipartite matching problems then nodes w_1 and w_2 cannot be siblings, because in each bipartite matching problem all children of some node of T_1 are matched against all children of some node of T_2 . Therefore, $\text{parent}(w_1) \neq \text{parent}(w_2)$. \square

Given the solutions $B \subseteq V_1 \times V_2$ to all the maximum cardinality bipartite matching problems solved during the top-down unordered subtree isomorphism procedure upon unordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$, the corresponding subtree isomorphism mapping $M \subseteq V_1 \times V_2$ can be reconstructed as follows. Set $M[\text{root}[T_1]]$ to $\text{root}[T_2]$ and, for all nodes $v \in V_1$ during a preorder traversal, set $M[v]$ to the unique node w with $(v, w) \in B$ and $(\text{parent}(v), \text{parent}(w)) \in B$.

Example 4.23. Consider the execution shown in Example 4.21 of the top-down unordered subtree isomorphism procedure, upon the unordered trees of Fig. 4.8. The solutions $B \subseteq V_1 \times V_2$ to all the maximum cardinality bipartite matching problems solved are as follows.

v_1	:	w_4		
v_2	:	w_2	w_9	w_{15}
v_3	:	w_3	w_{11}	w_{16}
v_4	:	w_1	w_5	w_{13}
v_5	:	w_{12}		
v_6	:	w_{17}		
v_7	:	w_{18}		

The unique subtree isomorphism mapping $M \subseteq B \subseteq V_1 \times V_2$ is indicated by the encircled nodes of T_2 .

The following result yields a simple improvement of the algorithm. The recursive solution to top-down unordered subtree isomorphism problems on subtrees that actually cannot have a solution, because the subtree of the first tree either does not have enough children, is not deep enough, or is not large enough, can be avoided even before constructing the corresponding maximum cardinality bipartite matching problems.

Lemma 4.24. *Let $M \subseteq V_1 \times V_2$ be a top-down subtree isomorphism of an unordered tree $T_1 = (V_1, E_1)$ into another unordered tree $T_2 = (V_2, E_2)$. Then,*

- $\text{children}(v) \leq \text{children}(w)$
- $\text{height}[v] \leq \text{height}[w]$
- $\text{size}[v] \leq \text{size}[w]$

for all $(v, w) \in M$.

Proof. Let $M \subseteq V_1 \times V_2$ be a top-down subtree isomorphism of an unordered tree $T_1 = (V_1, E_1)$ into an unordered tree $T_2 = (V_2, E_2)$. Suppose that $\text{children}(v) > \text{children}(w)$ for some $(v, w) \in M$. Let also $X_1 = (W_1, S_1)$ and $X_2 = (W_2, S_2)$ be the subtrees of T_1 and T_2 induced respectively by $W_1 = \{v\} \cup \{x \in V_1 \mid (v, x) \in E_1\}$ and $W_2 = \{w\} \cup \{y \in V_2 \mid (w, y) \in E_2\}$, and let $X = \{(x, y) \in M \mid x \in W_1, y \in W_2\}$. Since $|W_1| > |W_2|$, $X \subseteq W_1 \times W_2$ is not an injection, and then M is not an injection either, contradicting the hypothesis that M is a top-down

subtree isomorphism of T_1 into T_2 . Therefore, it must be true that $\text{children}(v) \leq \text{children}(w)$.

Suppose now that $\text{height}[v] > \text{height}[w]$ for some $(v, w) \in M$. Let $X_1 = (W_1, S_1)$ and $X_2 = (W_2, S_2)$ be the subtrees of T_1 and T_2 induced respectively by $W_1 = \{v\} \cup \{\text{parent}[x] \mid x \in W_1, x \neq \text{root}[T_1]\}$ and $W_2 = \{w\} \cup \{\text{parent}[y] \mid y \in W_2, y \neq \text{root}[T_2]\}$, and let $X = \{(x, y) \in M \mid x \in W_1, y \in W_2\}$. Since $\text{height}[v] + 1 = |W_1| > |W_2| = \text{height}[w] + 1$, $X \subseteq W_1 \times W_2$ is not an injection, and then M is not an injection either, contradicting the hypothesis that M is a top-down subtree isomorphism of T_1 into T_2 . Therefore, it must be true that $\text{height}[v] \leq \text{height}[w]$.

Finally, suppose that $\text{size}[v] > \text{size}[w]$ for some $(v, w) \in M$. Let $X_1 = (W_1, S_1)$ and $X_2 = (W_2, S_2)$ be the subtrees of T_1 and T_2 rooted respectively at nodes v and w , and let $X = \{(x, y) \in M \mid x \in W_1, y \in W_2\}$. Since $|W_1| > |W_2|$, $X \subseteq W_1 \times W_2$ is not an injection, and then M is not an injection either, contradicting the hypothesis that M is a top-down subtree isomorphism of T_1 into T_2 . Therefore, $\text{size}[v] \leq \text{size}[w]$. \square

The following algorithm implements the previous procedure for top-down unordered subtree isomorphism. The injection $M \subseteq V_1 \times V_2$ computed by the procedure upon two unordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ is represented by an array of nodes (of tree T_2) indexed by the nodes of tree T_1 .

Notice first that an unordered labeled tree is isomorphic to a top-down subtree of another unordered labeled tree if the unlabeled tree underlying the former labeled tree is isomorphic to the unlabeled tree underlying the subtree of the latter labeled tree and, furthermore, corresponding nodes share the same label.

The correspondence between children nodes in the trees and the vertices of the bipartite graph is represented by two maps of vertices (of the bipartite graph) indexed respectively by the nodes of trees T_1 and T_2 , in one direction, and in the opposite direction by a map of nodes (of trees T_1 and T_2) indexed by the vertices of the bipartite graph.

Recall that LEDA node maps are a dynamic variant of node arrays, implemented by an efficient hashing method based on the internal numbering of the nodes and that unlike node arrays, which

have a declaration cost linear in the number of nodes, declaration of a node map takes $O(1)$ time. Now, the bipartite graph built in order to decide if a node $v \in V_1$ can be mapped to a node $w \in V_2$ has only $\text{children}(v) + \text{children}(w)$ vertices and thus, since $\text{children}(v) + \text{children}(w) \ll n_1 + n_2$ for all but at most an $O(1)$ number of nodes in the trees, node maps are preferred because they save the $O(n_1)$ and $O(n_2)$ declaration costs of node arrays, although at the expense of access and update operations taking expected $O(1)$ time, instead of worst-case $O(1)$ time.

183

```
(subtree isomorphism 173) +≡
  bool top_down_unordered_subtree_isomorphism(
    const TREE<string,string>& T1,
    node r1,
    node_array<int> height1,
    node_array<int> size1,
    const TREE<string,string>& T2,
    node r2,
    node_array<int> height2,
    node_array<int> size2,
    node_array<set<node>>& B)
  {
    if ( T1[r1] ≠ T2[r2] ) return false;
    if ( T1.is_leaf(r1) ) return true;
    int p = T1.number_of_children(r1);
    int q = T2.number_of_children(r2);
    if ( p > q ∨ height1[r1] > height2[r2] ∨ size1[r1] > size2[r2] )
      return false;
    node_map<node> T1G(T1);
    node_map<node> T2G(T2);
    graph G;
    node_map<node> GT(G);
    node v1,v2;
    forall_children(v1,r1) {
      node v = G.new_node();
      GT[v] = v1;
      T1G[v1] = v;
    }
    forall_children(v2,r2) {
      node w = G.new_node();
      GT[w] = v2;
      T2G[v2] = w;
    }
    forall_children(v1,r1) {
      forall_children(v2,r2) {
```

```

if ( top_down_unordered_subtree_isomorphism
      ( $T_1, v_1, \text{height}_1, \text{size}_1, T_2, v_2, \text{height}_2, \text{size}_2, B$ ) )
       $G.\text{new\_edge}(T_1G[v_1], T_2G[v_2]);$ 
} }

list<edge>  $L = \text{MAX\_CARD\_BIPARTITE\_MATCHING}(G);$ 

if (  $L.\text{length}() \equiv p$  ) {
  edge  $e;$ 
  forall( $e, L$ )
     $B[GT[G.\text{source}(e)]].\text{insert}(GT[G.\text{target}(e)])$ ;
  return true;
} else {
  return false;
} }

```

The following algorithm implements the previous procedure for top-down subtree isomorphism of an unordered tree T_1 into another unordered tree T_2 , calling the previous recursive procedure upon the root of T_1 and the root of T_2 .

184a ⟨subtree isomorphism 173⟩ + ≡

```

bool top_down_unordered_subtree_isomorphism
  const TREE<string, string>&  $T_1$ ,
  const TREE<string, string>&  $T_2$ ,
  node_array<node>&  $M$ )
{
  node  $r_1 = T_1.\text{root}();$ 
  node  $r_2 = T_2.\text{root}();$ 
  ⟨compute height and size of all nodes in the first tree 186⟩
  ⟨compute height and size of all nodes in the second tree 187⟩
  node_array<set<node>>  $B(T_1);$ 
  bool isomorph = top_down_unordered_subtree_isomorphism
    ( $T_1, r_1, \text{height}_1, \text{size}_1, T_2, r_2, \text{height}_2, \text{size}_2, B$ );
  if ( isomorph ) {
    ⟨reconstruct unordered subtree isomorphism 184b⟩
    ⟨double-check top-down unordered subtree isomorphism 185⟩
  }
  return isomorph;
}

```

The following procedure reconstructs the top-down unordered subtree isomorphism mapping $M \subseteq V_1 \times V_2$ included in the solutions $B \subseteq V_1 \times V_2$ to all the maximum cardinality bipartite matching problems. The root of T_1 is mapped to the root of T_2 and, during a pre-order traversal, each nonroot node $v \in V_1$ is mapped to the unique node $w \in V_2$ with $(v, w) \in B$ and $(\text{parent}(v), \text{parent}(w)) \in B$.

184b \langle reconstruct unordered subtree isomorphism 184b $\rangle \equiv$

```

 $M[r1] = r2;$ 
 $\{ \text{node } v,w;$ 
 $\text{list} < \text{node} > L;$ 
 $\text{preorder\_tree\_list\_traversal}(T1,L);$ 
 $\text{forall}(v,L) \{$ 
 $\quad \text{if} ( \neg T1.\text{is\_root}(v) ) \{$ 
 $\quad \quad \text{forall}(w,B[v]) \{$ 
 $\quad \quad \quad \text{if} ( M[T1.\text{parent}(v)] \equiv T2.\text{parent}(w) ) \{$ 
 $\quad \quad \quad \quad M[v] = w;$ 
 $\quad \quad \quad \quad \text{break};$ 
 $\quad \quad \} \} \} \}$ 

```

The following double-check of top-down unordered subtree isomorphism, although being redundant, gives some reassurance of the correctness of the implementation. It verifies that M is a top-down unordered subtree isomorphism of T_1 into T_2 , according to Definition 4.19.

185 \langle double-check top-down unordered subtree isomorphism 185 $\rangle \equiv$

```

 $\{ \text{node } v,w;$ 
 $\quad \text{if} ( M[T1.\text{root}()] \neq T2.\text{root}() )$ 
 $\quad \quad \text{error\_handler}(1,$ 
 $\quad \quad \quad \text{"Wrong implementation of top-down subtree isomorphism");}$ 
 $\quad \text{forall\_nodes}(v,T1) \{$ 
 $\quad \quad \text{if} ( M[v] \equiv \text{nil} \vee T1[v] \neq T2[M[v]] )$ 
 $\quad \quad \quad \text{error\_handler}(1,$ 
 $\quad \quad \quad \quad \text{"Wrong implementation of top-down subtree isomorphism");}$ 
 $\quad \quad \text{if} ( \neg T1.\text{is\_root}(v) \wedge$ 
 $\quad \quad \quad ( T2.\text{is\_root}(M[v]) \vee$ 
 $\quad \quad \quad ( M[T1.\text{parent}(v)] \neq T2.\text{parent}(M[v]) ) ) )$ 
 $\quad \quad \quad \text{error\_handler}(1,$ 
 $\quad \quad \quad \quad \text{"Wrong implementation of top-down subtree isomorphism");}$ 
 $\quad \text{forall\_nodes}(w,T1) \{$ 
 $\quad \quad \text{if} ( v \neq w \wedge M[v] \equiv M[w] )$ 
 $\quad \quad \quad \text{error\_handler}(1,$ 
 $\quad \quad \quad \quad \text{"Wrong implementation of top-down subtree isomorphism");}$ 
 $\quad \} \} \}$ 

```

Remark 4.25. Correctness of the algorithm for top-down unordered subtree isomorphism follows from Lemmas 4.22 and 4.24.

Recall that the LEDA implementation of the maximum cardinality bipartite matching algorithm of [165] runs in $O(pq\sqrt{q})$ time using $O(pq)$ additional space, where $p + q$ is the number of nodes in the bipartite graph and $p \leq q$.

Lemma 4.26. *Let T_1 and T_2 be unordered trees with respectively n_1 and n_2 nodes, where $n_1 \leq n_2$. The algorithm for top-down unordered subtree isomorphism runs in $O(n_1 n_2 \sqrt{n_2})$ time using $O(n_1 n_2)$ additional space.*

Proof. Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be unordered trees on respectively n_1 and n_2 nodes, where $n_1 \leq n_2$, and let $k(v)$ denote the number of children of node v . The effort spent on a leaf node of T_1 is $O(1)$, and the total effort spent on the leaves of T_1 is thus bounded by $O(n_1)$. The effort spent on a nonleaf v of T_1 and a nonleaf w of T_2 , on the other hand, is dominated by solving a maximum cardinality bipartite matching problem on a bipartite graph with $k(v) + k(w)$ nodes, where $k(v) \leq k(w)$, and is thus bounded by $O(k(v)k(w)\sqrt{k(w)})$. The total effort spent on nonleaves of T_1 and T_2 is thus bounded by $O(\sum_{v \in V_1} \sum_{w \in V_2} k(v)k(w)\sqrt{k(w)})$. Reconstructing the actual subtree isomorphism mapping M of T_1 into T_2 takes $O(n_1 n_2)$ time. Therefore, the algorithm runs in $O(n_1 n_2 \sqrt{n_2})$ time.

A similar argument shows that the algorithm uses $O(n_1 n_2)$ additional space. The double-check of top-down unordered subtree isomorphism runs in $O(n_1 n_2)$ time using $O(1)$ additional space. \square

The algorithm solves the problem of finding a top-down subtree isomorphism of an unordered tree into another unordered tree. The problem of enumerating all top-down unordered subtree isomorphisms can be solved with the subgraph isomorphism algorithms given in Sect. 7.3.

A few implementation details still need to be filled in. The height and size of all nodes in the trees is computed by the following procedure, during an iterative postorder traversal of the trees instead.

(compute height and size of all nodes in the first tree 186) \equiv

```

node_array<int> heightI(T1);
node_array<int> sizeI(T1);
{ node v,w;
list<node> L;
postorder_tree.list_traversal(T1,L);
forall(v,L) {
    heightI[v] = 0; // leaves have height equal to zero
    sizeI[v] = 1; // and size equal to one
    if ( !T1.is_leaf(v) ) {
        forall_children(w,v) {
            heightI[v] = ledamax(heightI[v],heightI[w]);
        }
    }
}

```

```

    size1[v] += size1[w];
}
height1[v]++;
// one plus the largest height among the children
} } }

```

187 ⟨compute height and size of all nodes in the second tree 187⟩ ≡

```

node_array<int> height2(T2);
node_array<int> size2(T2);
{ node v,w;
list<node> L;
postorder_tree_list_traversal(T2,L);
forall(v,L) {
    height2[v] = 0; // leaves have height equal to zero
    size2[v] = 1; // and size equal to one
    if ( ¬T2.is_leaf(v) ) {
        forall_children(w,v) {
            height2[v] = led_a_max(height2[v],height2[w]);
            size2[v] += size2[w];
        }
        height2[v]++;
        // one plus the largest height among the children
    } } }

```

4.2.3 Bottom-Up Subtree Isomorphism

An ordered tree T_1 is isomorphic to a bottom-up subtree of another ordered tree T_2 if there is an injective correspondence of the node set of T_1 into the node set of T_2 which preserves the ordered structure of T_1 and, furthermore, reflects the ordered structure of a bottom-up subtree of T_2 —that is, such that the node of T_2 corresponding to the first child in T_1 of a node v_1 is the first child in T_2 of the node corresponding to v_1 , the node of T_2 corresponding to the next sibling in T_1 of a node v_2 is also the next sibling in T_2 of the node corresponding to v_2 , and all nodes of T_2 corresponding to leaves of T_1 are also leaves of T_2 .

Definition 4.27. An ordered tree $T_1 = (V_1, E_1)$ is **isomorphic to a bottom-up subtree** of another ordered tree $T_2 = (V_2, E_2)$ if there is an injection $M \subseteq V_1 \times V_2$ such that the following conditions

- $(\text{first}[v], \text{first}[w]) \in M$ for all nonleaves $v \in V_1$ and $w \in V_2$ with $(v, w) \in M$
- $(\text{next}[v], \text{next}[w]) \in M$ for all nonlast children $v \in V_1$ and $w \in V_2$ with $(v, w) \in M$

- for all leaves $v \in V_1$ and nodes $w \in V_2$ with $(v, w) \in M$, w is a leaf in T_2

are satisfied. In such a case, M is a bottom-up ordered subtree isomorphism of T_1 into T_2 .

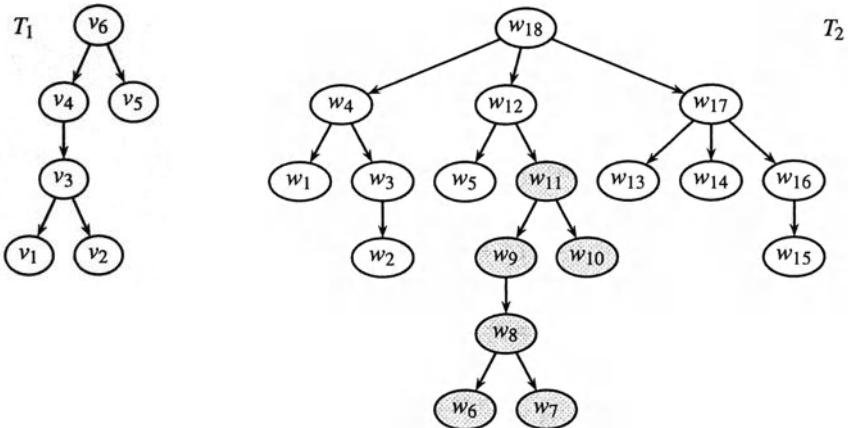


Fig. 4.9. An ordered tree which is isomorphic to a bottom-up subtree of another ordered tree. Nodes are numbered according to the order in which they are visited during a postorder traversal. The subtree of T_2 which is isomorphic to T_1 is shown highlighted.

Example 4.28. The ordered tree $T_1 = (V_1, E_1)$ in Fig. 4.9 is isomorphic to a bottom-up subtree of the ordered tree $T_2 = (V_2, E_2)$. The injection $M \subseteq V_1 \times V_2$ given by $M = \{(v_1, w_6), (v_2, w_7), (v_3, w_8), (v_4, w_9), (v_5, w_{10}), (v_6, w_{11})\}$ is a bottom-up ordered subtree isomorphism of T_1 into T_2 .

An Algorithm for Bottom-Up Ordered Subtree Isomorphism

A bottom-up subtree isomorphism of an ordered tree T_1 into another ordered tree T_2 can be obtained by repeated invocation of the algorithm for ordered tree isomorphism, upon the root of T_1 and each of the nodes of T_2 in turn.

The following result yields a simple improvement of the procedure. Notice that it holds for bottom-up subtrees only, both ordered and unordered, but it does not hold for top-down subtrees.

Lemma 4.29. Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be trees with respectively n_1 and n_2 nodes, where $n_1 \leq n_2$ and $V_1 \cap V_2 = \emptyset$. Then, for all bottom-up subtrees $X_1 = (W_1, S_1)$ and $X_2 = (W_2, S_2)$ of T_2 which are isomorphic to T_1 ,

- $\text{height}[X_1] = \text{height}[X_2] = \text{height}[T_1]$
- $\text{size}[X_1] = \text{size}[X_2] = n_1$
- $W_1 \cap W_2 = \emptyset$ if $X_1 \neq X_2$

Proof. Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be trees with respectively n_1 and n_2 nodes, where $n_1 \leq n_2$ and $V_1 \cap V_2 = \emptyset$. Let also $X_1 = (W_1, S_1)$ and $X_2 = (W_2, S_2)$ be bottom-up subtrees of T_2 isomorphic to T_1 , and let $M_1 \subseteq V_1 \times W_1$ and $M_2 \subseteq V_1 \times W_2$ be bottom-up subtree isomorphisms of T_1 into X_1 and into X_2 , respectively. Being M_1 and M_2 bijections, then $|V_1| = |X_1|$ and $|V_1| = |X_2|$, that is, $n_1 = \text{size}[X_1] = \text{size}[X_2]$.

Suppose now that $\text{height}[T_1] \neq \text{height}[X_1]$. Let $v \in V_1$ be a leaf of largest depth in T_1 , and let $w \in W_1$ be a leaf of largest depth in X_1 . Let also $T'_1 = (V'_1, E'_1)$ and $X'_1 = (W'_1, S'_1)$ be the subtrees of T_1 and X_1 induced respectively by $V'_1 = \{v\} \cup \{\text{parent}[x] \mid x \in V'_1, x \neq \text{root}[T_1]\}$ and $W'_1 = \{w\} \cup \{\text{parent}[y] \mid y \in W'_1, y \neq \text{root}[X_1]\}$, and let $X = \{(x, y) \in M_1 \mid x \in V'_1, y \in W'_1\}$. Since $\text{depth}[v] + 1 = \text{height}[T'_1] + 1 = |V'_1| \neq |W'_1| = \text{height}[X'_1] + 1 = \text{depth}[w] + 1$, $X \subseteq V'_1 \times W'_1$ is not an injection, and then M_1 is not an injection either, contradicting the hypothesis that M_1 is a bottom-up subtree isomorphism of T_1 into X_1 . Therefore, $\text{height}[T_1] = \text{height}[X_1]$. A similar argument shows that $\text{height}[T_1] = \text{height}[X_2]$.

Let now $X_1 = (W_1, S_1)$ and $X_2 = (W_2, S_2)$ be distinct bottom-up subtrees of T_2 which are isomorphic to T_1 . Suppose $W_1 \cap W_2 = \emptyset$, and let $v \in W_1 \cap W_2$ be a node of largest height in $W_1 \cap W_2$. If $v = \text{root}[X_1]$ then $v = \text{root}[X_2]$ as well, because $\text{height}[X_1] = \text{height}[X_2]$ and then $X_1 = X_2$, because X_1 and X_2 are bottom-up subtrees, contradicting the hypothesis that X_1 and X_2 are distinct. Otherwise, there are nodes $u \in W_1$ and $w \in W_2$ such that $(u, v) \in S_1$ and $(w, v) \in S_2$ and then, $\text{parent}[v]$ is not well-defined in T_2 . Therefore, $W_1 \cap W_2 = \emptyset$ if $X_1 \neq X_2$. \square

Now, the repeated invocation of the algorithm for ordered tree isomorphism upon the root of T_1 and each of the nodes v of T_2 with $\text{height}[T_1] = \text{height}[v]$ and $\text{size}[v] = n_1$, is sufficient in order to find all bottom-up subtrees of T_2 which are isomorphic to T_1 .

The following algorithm implements the previous procedure for bottom-up subtree isomorphism of an ordered tree T_1 into another ordered tree T_2 , also collecting a list of mappings M of nodes of T_1 to nodes of T_2 .

190a \langle subtree isomorphism 173 $\rangle + \equiv$

```

void bottom_up_ordered_subtree_isomorphism(
  const TREE<string,string>& T1,
  const TREE<string,string>& T2,
  list<node.array<node>>& L)
{
  node r1 = T1.root();

  {compute height of root of first tree 190b}
  {compute height and size of all nodes in the second tree 187}

  node.array<node> M(T1);
  L.clear();

  node v;
  forall_nodes(v,T2) {
    if (height2[v]  $\equiv$  h1  $\wedge$  size2[v]  $\equiv$  T1.number_of_nodes() ) {
      if (map_ordered_subtree(T1,r1,T2,v,M)) {
        {double-check bottom-up ordered subtree isomorphism 190c}
        L.append(M);
      }
    }
  }
}

```

Only the height and size of the root of the first tree is needed. The following procedure computes the height of the root of the first tree, during a bottom-up traversal of the tree. The size of the root is just the number of nodes in the first tree.

190b \langle compute height of root of first tree 190b $\rangle \equiv$

```

int h1;
{ node.array<int> height(T1);
  bottom_up_tree_height(T1,height);
  h1 = height[r1];
}

```

The following double-check of bottom-up ordered subtree isomorphism, although being redundant, gives some reassurance of the correctness of the implementation. It verifies that M is a bottom-up ordered subtree isomorphism of T_1 into T_2 , according to Definition 4.27.

190c \langle double-check bottom-up ordered subtree isomorphism 190c $\rangle \equiv$

```

{ node v;
forall_nodes(v,T1) {
  if (M[v]  $\equiv$  nil  $\vee$  T1[v]  $\neq$  T2[M[v]] )
    error_handler(1,
}

```

```

    "Wrong implementation of bottom-up subtree isomorphism");
if (  $\neg T1.is\_leaf(v)$   $\wedge$ 
      (  $T2.is\_leaf(M[v]) \vee$ 
        (  $M[T1.first\_child(v)] \neq T2.first\_child(M[v])$  ) ) )
  error_handler(1,
    "Wrong implementation of bottom-up subtree isomorphism");
if (  $\neg T1.is\_last\_child(v)$   $\wedge$ 
      (  $T2.is\_last\_child(M[v]) \vee$ 
        (  $M[T1.next\_ sibling(v)] \neq T2.next\_ sibling(M[v])$  ) ) )
  error_handler(1,
    "Wrong implementation of bottom-up subtree isomorphism");
if (  $T1.is\_leaf(v) \wedge \neg T2.is\_leaf(M[v])$  )
  error_handler(1,
    "Wrong implementation of bottom-up subtree isomorphism");
}
}

```

Remark 4.30. Correctness of the algorithm for bottom-up ordered subtree isomorphism follows from Lemma 4.29.

Lemma 4.31. *Let T_1 and T_2 be ordered trees with respectively n_1 and n_2 nodes, where $n_1 \leq n_2$. The algorithm for bottom-up ordered subtree isomorphism runs in $O(n_2)$ time using $O(n_1)$ additional space.*

Proof. Let T_1 and T_2 be ordered trees with respectively n_1 and n_2 nodes, where $n_1 \leq n_2$, and let k be the number of distinct bottom-up ordered subtrees of T_2 of height $height[T_1]$ and size n_1 . The algorithm makes k calls to the ordered tree isomorphism procedure, and thus takes $O(kn_1)$ time using $O(n_1)$ additional space. Since by Lemma 4.29 these k bottom-up ordered subtrees of T_2 are pairwise node-disjoint, it follows that $kn_1 \leq n_2$. Therefore, the algorithm runs in $O(n_2)$ time using $O(n_1)$ additional space.

The double-check of bottom-up ordered subtree isomorphism runs in $O(n_1)$ time using $O(1)$ additional space. \square

4.2.4 Bottom-Up Unordered Subtree Isomorphism

An unordered tree T_1 is isomorphic to a bottom-up subtree of another unordered tree T_2 if there is an injective correspondence of the node set of T_1 into the node set of T_2 which preserves the structure of T_1 and, furthermore, reflects the structure of a bottom-up subtree of T_2 —that is, such that the node of T_2 corresponding to the parent in T_1 of a node v is the parent in T_2 of the node corresponding to v , and all nodes of T_2 corresponding to leaves of T_1 are also leaves of T_2 .

Definition 4.32. An unordered tree $T_1 = (V_1, E_1)$ is **isomorphic to a bottom-up subtree** of another unordered tree $T_2 = (V_2, E_2)$ if there is an injection $M \subseteq V_1 \times V_2$ such that the following conditions

- for all nonroot nodes $v \in V_1$ such that $(\text{parent}[v], \text{parent}[y]) \in M$ for some nonroot node $y \in V_2$, $(v, w) \in M$ for some nonroot node $w \in V_2$ and, furthermore, $\text{parent}[w] = \text{parent}[y]$
- for all leaves $v \in V_1$ and nodes $w \in V_2$ with $(v, w) \in M$, w is a leaf in T_2

are satisfied. In this case, M is a bottom-up unordered subtree isomorphism of T_1 into T_2 .

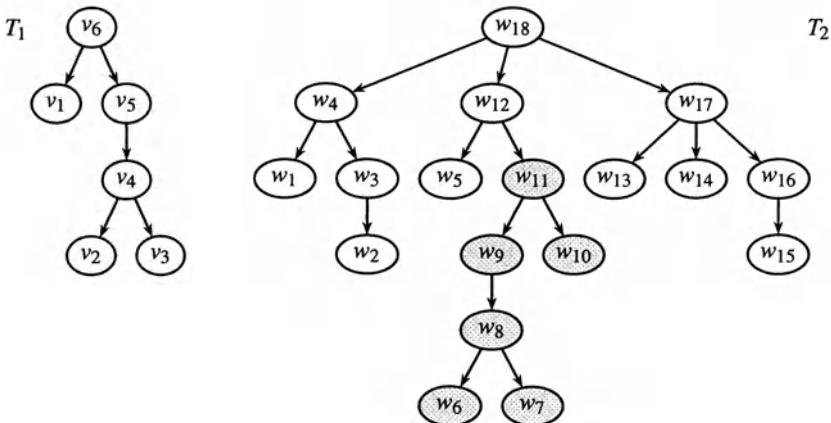


Fig. 4.10. An unordered tree which is isomorphic to a bottom-up subtree of another unordered tree. Nodes are numbered according to the order in which they are visited during a postorder traversal. The subtree of T_2 which is isomorphic to T_1 is shown highlighted.

Example 4.33. The unordered tree $T_1 = (V_1, E_1)$ in Fig. 4.10 is isomorphic to a bottom-up subtree of the unordered tree $T_2 = (V_2, E_2)$. The injection $M \subseteq V_1 \times V_2$ given by $M = \{(v_1, w_{10}), (v_2, w_6), (v_3, w_7), (v_4, w_8), (v_5, w_9), (v_6, w_{11})\}$ is a bottom-up unordered subtree isomorphism of T_1 into T_2 .

The problem of finding all bottom-up subtrees of an unordered tree $T_2 = (V_2, E_2)$ on n_2 nodes which are isomorphic to an unordered tree

$T_1 = (V_1, E_1)$ on n_1 nodes, where $n_1 \leq n_2$, can be reduced to the problem of partitioning $V_1 \cup V_2$ into equivalence classes of bottom-up subtree isomorphism. Two nodes (in the same or in different trees) are equivalent if and only if the bottom-up unordered subtrees rooted at them are isomorphic. Then, T_1 is isomorphic to the bottom-up subtree of T_2 rooted at node $w \in V_2$ if and only if nodes $\text{root}[T_1]$ and w belong to the same equivalence class of bottom-up subtree isomorphism.

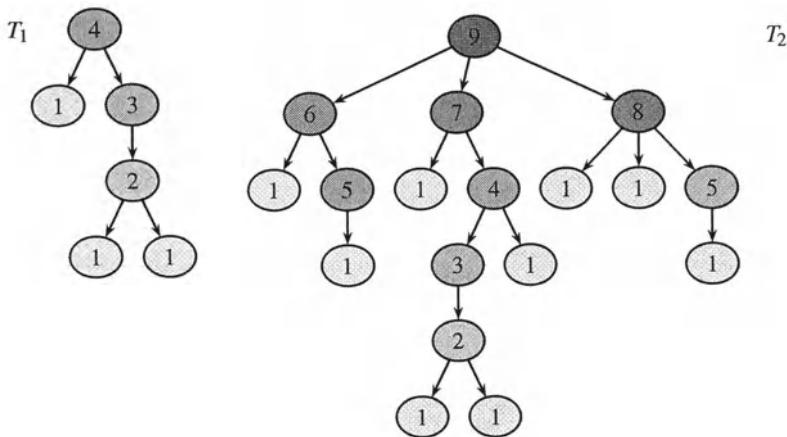


Fig. 4.11. Bottom-up unordered subtree isomorphism equivalence classes for the unordered trees in Fig. 4.10. Nodes are numbered according to the equivalence class to which they belong, and the equivalence classes are shown highlighted in different shades of gray.

Example 4.34. The partition of the unordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ of Fig. 4.10 in equivalence classes of bottom-up subtree isomorphism is illustrated in Fig. 4.11. The nine equivalence classes are numbered from 1 to 9. Tree T_1 is isomorphic to the bottom-up subtree of T_2 rooted at node w_{11} , because nodes v_6 (the root of T_1) and w_{11} both belong to the same equivalence class, the one numbered 4.

Remark 4.35. Notice that partitioning the nodes of two unordered trees in equivalence classes of bottom-up subtree isomorphism is different from assigning isomorphism codes to the nodes in the trees, although isomorphism codes do yield a partition of an unordered tree in bottom-up subtree isomorphism equivalence classes. While the isomorphism code for an unordered tree in Definition 4.7 allows the

reconstruction of a unique (up to isomorphism) unordered tree, the bottom-up subtree isomorphism equivalence classes of the nodes of a tree do not convey enough information to make reconstruction of the tree possible. As a matter of fact, the partition of the nodes of a tree (say, T_2) in bottom-up subtree isomorphism equivalence classes is not a function of T_2 alone, but of tree T_1 as well.

An Algorithm for Bottom-Up Unordered Subtree Isomorphism

The bottom-up subtree isomorphisms of an unordered tree $T_1 = (V_1, E_1)$ into another unordered tree $T_2 = (V_2, E_2)$ can be obtained by first partitioning the set of nodes $V_1 \cup V_2$ in bottom-up subtree isomorphism equivalence classes, and then testing equivalence of the root of T_1 and each of the nodes of T_2 in turn.

A simple procedure for partitioning an unordered tree in equivalence classes of bottom-up subtree isomorphism, consists of maintaining a dictionary of known equivalence classes during a postorder traversal (or a bottom-up traversal) of the trees. Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be unordered trees with respectively n_1 and n_2 nodes, where $n_1 \leq n_2$. The k equivalence classes of bottom-up subtree isomorphism of T_1 into T_2 can be numbered from 1 to k , where $1 \leq k \leq n_1 + n_2$, as follows.

Let the number of known equivalence classes be initially equal to 1, corresponding to the equivalence class of all leaves in the trees. For all nodes v of T_1 and T_2 in postorder, set the equivalence class of v to 1 if node v is a leaf. Otherwise, look up in the dictionary the ordered list of equivalence classes to which the children of node v belong. If the ordered list (key) is found in the dictionary, set the equivalence class of node v to the value (element) found. Otherwise, increment by one the number of known equivalence classes, insert the ordered list together with the number of known equivalence classes in the dictionary, and set the equivalence class of node v to the number of known equivalence classes.

Example 4.36. After partitioning the unordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ of Fig. 4.10 into equivalence classes of bottom-up subtree isomorphism, as illustrated in Fig. 4.11, the dictionary of known equivalence classes contains the entries shown in Fig. 4.12.

key	element	key	element
[1, 1]	2	[1, 5]	6
[2]	3	[1, 4]	7
[1, 3]	4	[1, 1, 5]	8
[1]	5	[6, 7, 8]	9

Fig. 4.12. Contents of the dictionary of known equivalence classes upon partitioning the unordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ of Fig. 4.10 in equivalence classes of bottom-up subtree isomorphism.

Notice that testing of bottom-up unordered subtree isomorphism based on equivalence classes of subtree isomorphism applies to unlabeled trees only. However, the previous bottom-up subtree isomorphism procedure for unordered unlabeled trees with n nodes can be extended to unordered trees whose nodes are labeled by integers in the range $1, \dots, n$. See Exercise 4.7.

The following algorithm implements the previous procedure for bottom-up subtree isomorphism of an unordered tree T_1 into another unordered tree T_2 , collecting a list of mappings M of nodes of T_1 to nodes of T_2 . The bottom-up subtree isomorphism classes are also collected, as mappings of the nodes of T_1 and T_2 to integers, for the purpose of interactive demonstration of the algorithm.

195

```

⟨subtree isomorphism 173⟩ +≡
void bottom_up_unordered_subtree_isomorphism(
    const TREE<string, string>& T1,
    const TREE<string, string>& T2,
    node_array<int>& code1,
    node_array<int>& code2,
    list<node_array<node>>& L)
{
    list<node> L1;
    postorder_tree_list_traversal(T1, L1);

    list<node> L2;
    postorder_tree_list_traversal(T2, L2);

    int num = 1;
    h_array<list<int>, int> CODE;

    bool unordered = true;
    ⟨partition first tree in isomorphism equivalence classes 196a⟩
    ⟨partition second tree in isomorphism equivalence classes 196b⟩

    node r1 = T1.root();
    L.clear();
    node_array<node> M(T1);

```

```

node v;
forall(v,L2) {
  if ( code1[rI] ≡ code2[v] ) {
    <build subtree isomorphism mapping 198a>
    L.append(M);
  } } }

```

Partitioning an unordered tree in equivalence classes of bottom-up subtree isomorphism is done with the help of a variable global to the whole procedure, the number *num* of known equivalence classes, and a dictionary *CODE*, also global to the whole procedure, with lists of integers (equivalence classes of children nodes) as keys and integers (equivalence classes of nodes) as elements. After all nodes of tree T_1 are partitioned into equivalence classes of bottom-up subtree isomorphism, the nodes of tree T_2 are also partitioned, sharing the same variable *num* and dictionary *CODE*.

Given that the equivalence classes of all nodes in the trees are integers falling in a fixed range, the ordered list of equivalence classes to which the children of a node belong can be built in time linear in the number of children plus the range of equivalence classes of the children nodes, by bucket sorting the corresponding unordered list of equivalence classes.

196a

```

⟨partition first tree in isomorphism equivalence classes 196a⟩≡
{ list<int> L;
  node v,w;
  forall(v,L1) {
    if ( T1.is_leaf(v) ) {
      code1[v] = 1;
    } else {
      L.clear();
      forall_children(w,v)
      L.append(code1[w]);
      if ( unordered ) sort_isomorphism_codes(L);

      if ( CODE.defined(L) ) {
        code1[v] = CODE[L];
      } else {
        CODE[L] = ++num;
        code1[v] = num;
      } } } }

```

196b

```

⟨partition second tree in isomorphism equivalence classes 196b⟩≡
{ list<int> L;

```

```

node v,w;
forall(v,L2) {
  if( T2.is_leaf(v) ) {
    code2[v] = 1;
  } else {

    L.clear();
    forall_children(w,v)
      L.append(code2[w]);
    if( unordered ) sort_isomorphism_codes(L);

    if( CODE.defined(L) ) {
      code2[v] = CODE[L];
    } else {
      CODE[L] = ++num;
      code2[v] = num;
    }
  }
}
}

```

Bucket sorting a list of k integers in the range $1, \dots, n$ takes $O(k + n)$ time although for small values of k , sorting by direct comparison of the integers takes $O(1)$ time instead. The following procedure sorts a list of k integers by direct comparison, if $k \leq 3$, relying otherwise on bucket sort.

197

\langle subtree isomorphism 173 $\rangle + \equiv$

```

void sort_isomorphism_codes(
  list<int>& L)
{
  int x,y,z;
  switch ( L.length() ) {
    case 0:
      break;
    case 1:
      break;
    case 2:
      x = L.pop();
      y = L.pop();
      if ( x <= y ) { L.append(x); L.append(y); }
      else { L.append(y); L.append(x); }
      break;
    case 3:
      x = L.pop();
      y = L.pop();
      z = L.pop();
      if ( x <= y ) {
        if ( y <= z ) { L.append(x); L.append(y); L.append(z); }
        else {
          if ( x <= z ) { L.append(x); L.append(z); L.append(y); }
          else { L.append(z); L.append(x); L.append(y); }
        }
      }
    } else {

```

```

if ( y  $\leqslant$  z ) {
  if ( x  $\leqslant$  z ) { L.append(y); L.append(x); L.append(z); }
  else { L.append(y); L.append(z); L.append(x); }
} else { L.append(z); L.append(y); L.append(x); }
}
break;
default:
L.bucket_sort(id);
break;
}
}

```

Now, an actual bottom-up subtree isomorphism mapping $M \subseteq V_1 \times V_2$ of tree $T_1 = (V_1, E_1)$ into the subtree of $T_2 = (V_2, E_2)$ rooted at node v can be constructed by mapping the root of T_1 to node v , and then mapping the remaining nodes in T_1 to the remaining nodes in the subtree of T_2 rooted at node v , such that mapped nodes belong to the same equivalence class of bottom-up subtree isomorphism.

198a \langle build subtree isomorphism mapping 198a $\rangle \equiv$
 $M[r1] = v;$
 $map_unordered_subtree(T1,r1,T2,v,code1,code2,M);$

Mapping the nodes of T_1 to equivalent nodes in the subtree of T_2 during a preorder traversal of T_1 , guarantees that the bottom-up subtree isomorphism mapping preserves the structure of tree T_1 . In the following recursive procedure, each of the children of node $r_1 \in V_1$ is mapped to some unmapped child of node $r_2 \in V_2$ belonging to the same equivalence class.

198b \langle subtree isomorphism 173 $\rangle + \equiv$
void *map_unordered_subtree*(
 const *TREE<string,string>*& *T1*,
 const *node r1*,
 const *TREE<string,string>*& *T2*,
 const *node r2*,
 const *node_array<int>*& *code1*,
 const *node_array<int>*& *code2*,
node_array<node>& *M*)
{
 node v,w;
 list<node> *L2*;
 forall_children(*w,r2*) *L2.append*(*w*);
 forall_children(*v,r1*) {
 list_item it;
 forall_items(*it,L2*) {
 w = L2[it];
 if (*code1[v] ≡ code2[w]*) {
 M[v] = w;
 }
 }
 }
}

```

    L2.del(it); // node w already mapped to
    map_unordered_subtree(T1,v,T2,w,code1,code2,M);
    break;
} } } }
```

The following double-check of bottom-up unordered subtree isomorphism, although being redundant, gives some reassurance of the correctness of the implementation. It verifies that M is a bottom-up unordered subtree isomorphism of T_1 into T_2 , according to Definition 4.32.

199 ⟨double-check bottom-up unordered subtree isomorphism 199⟩≡

```

{ node v,w,z;
  bool found;
  forall_nodes(v,T1) {
    if ( M[v] ≡ nil ∨ T1[v] ≠ T2[M[v]] )
      error_handler(1,
        "Wrong implementation of bottom-up subtree isomorphism");
    forall_nodes(w,T1)
      if ( v ≠ w ∧ M[v] ≡ M[w] )
        error_handler(1,
          "Wrong implementation of bottom-up subtree isomorphism");
    forall_nodes(w,T2) {
      if ( ¬T1.is_root(v) ∧ ¬T2.is_root(w)
          ∧ M[T1.parent(v)] ≡ T2.parent(w) ) {
        found = false;
        forall_children(z,T2.parent(w))
          if ( M[v] ≡ z ) found = true;
        if ( ¬found ) {
          error_handler(1,
            "Wrong implementation of bottom-up subtree isomorphism");
        } } }
      if ( T1.is_leaf(v) ∧ ¬T2.is_leaf(M[v]) )
        error_handler(1,
          "Wrong implementation of bottom-up subtree isomorphism");
    } }

```

Remark 4.37. Correctness of the algorithm for bottom-up unordered subtree isomorphism follows from the fact that the equivalence class of bottom-up subtree isomorphism which each node in the trees belongs to is either equal to 1, if the node is a leaf, or is determined by the equivalence classes which the children of the node belong to. During a postorder traversal of each of the trees in turn, known equivalence classes are looked up in a dictionary and new equivalence classes are inserted into the dictionary.

Theorem 4.38. Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be unordered trees with respectively n_1 and n_2 nodes. The algorithm for bottom-up unordered subtree isomorphism runs in $O((n_1 + n_2)^2)$ time using $O(n_1 + n_2)$ additional space.

Proof. Let T_1 and T_2 be unordered trees with respectively n_1 and n_2 nodes, and let $k(v)$ denote the number of children of node v . The effort spent on a leaf node of T_1 or T_2 is $O(1)$, and the total effort spent on the leaves of T_1 and T_2 is bounded by $O(n_1 + n_2)$.

The effort spent on a nonleaf v of T_1 or T_2 , on the other hand, is dominated by bucket sorting a list of $k(v)$ integers, despite the expected $O(1)$ time taken to look up or insert the sorted list of integers in the dictionary. Since all these integers fall in the range $1, \dots, n_1 + n_2$, the time taken for bucket sorting the children codes is bounded by $O(k(v) + n_1 + n_2)$. The total effort spent on nonleaves of T_1 and T_2 is thus bounded by $O(\sum_{v \in V_1 \cup V_2} (k(v) + n_1 + n_2)) = O(\sum_{v \in V_1 \cup V_2} k(v)) + O((n_1 + n_2)^2) = O(n_1 + n_2) + O((n_1 + n_2)^2) = O((n_1 + n_2)^2)$. Therefore, the algorithm runs in $O((n_1 + n_2)^2)$ time using $O(n_1 + n_2)$ additional space.

The double-check of bottom-up unordered subtree isomorphism runs in $O(n_1 n_2)$ time using $O(1)$ additional space. \square

Lemma 4.39. Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be unordered trees of degree bounded by a constant k and with respectively n_1 and n_2 nodes. The algorithm for bottom-up unordered subtree isomorphism can be implemented to run in expected $O(n_1 + n_2)$ time using $O(n_1 + n_2)$ additional space.

Proof. For trees of degree bounded by a constant k , bucket sorting a list of at most k integers can be replaced by appropriate code performing at most k comparisons, and thus taking $O(1)$ time to sort a list of at most k integers, as shown for $k = 3$ in the previous implementation of the algorithm. In this case, the effort spent on a nonleaf v of T_1 or T_2 is dominated instead by the expected $O(1)$ time required to look up or insert the sorted list of k integers in the dictionary, and the algorithm runs in expected $O(n_1 + n_2)$ time, still using $O(n_1 + n_2)$ additional space. \square

References to more efficient algorithms for bottom-up unordered subtree isomorphism are given in the bibliographic notes, at the end of the chapter.

A few implementation details still need to be filled in. Maintaining a dictionary (implemented by a hashing table) with lists of integers as keys, requires the definition of a hash function mapping lists of integers to integers. The default LEDA hash function is redefined by the following procedure to a simple hash function on lists of integers, consisting of summing up the integers in the key.

201a $\langle \text{subtree isomorphism 173} \rangle + \equiv$

```
int Hash(
    const list<int>& L)
{
    int sum = 0;
    int x;
    forall(x,L) sum += x;
    return sum;
}
```

Bucket sorting a list of elements requires a function mapping the elements to integers. Since LEDA does not provide a default identity function for sorting lists of integers, the following procedure implements such an identity function on integers.

201b $\langle \text{subtree isomorphism 173} \rangle + \equiv$

```
inline int id(
    const int& x)
{
    return x;
}
```

4.2.5 Interactive Demonstration of Subtree Isomorphism

The algorithms for top-down and bottom-up subtree isomorphism upon ordered and unordered trees, are all integrated next in the interactive demonstration of graph algorithms. A simple checker for top-down isomorphic subtrees that provides some visual reassurance consists of highlighting the nodes and edges of the top-down subtree of T_2 that is isomorphic to T_1 , and also redrawing tree T_1 according to

the subtree isomorphism found, to match the layout of the isomorphic subtree of T_2 .

202a

```

⟨demo subtree isomorphism 202a⟩≡
void gw_top_down_ordered_subtree_isomorphism(
    GraphWin& gw1)
{
    GRAPH<string,string>& G1 = gw1.get_graph();
    TREE<string,string> T1(G1);

    GRAPH<string,string> G2;
    GraphWin gw2(G2,500,500,
        "Top-Down Ordered Subtree Isomorphism");
    gw2.display();
    gw2.message("Enter second tree. Press done when finished");
    gw2.edit();
    gw2.del_message();

    TREE<string,string> T2(G2);

    node_array<node> M(T1);
    bool isomorph = top_down_ordered_subtree_isomorphism(T1,T2,M);

    panel P;
    if ( isomorph ) {
        make_proof_panel(P,"There is an isomorphic subtree",true);
        if ( gw1.open_panel(P) ) { // proof button pressed
            gw1.save_all_attributes();
            node_array<point> pos1(T1);
            node_array<point> pos2(T2);
            ⟨show subtree isomorphism 202b⟩
            gw1.wait();
            gw1.restore_all_attributes();
        } else {
            make_proof_panel(P,"There is \\\red no isomorphic subtree",false);
            gw1.open_panel(P);
        }
    }
}

```

Tree T_1 is redrawn according to the subtree isomorphism mapping M found, to match the layout of the isomorphic subtree of T_2 , by updating the position of every node $v \in V_1$ in the layout of T_1 to be identical with the position of node $M[v] \in V_2$ in the layout of T_2 , and removing any edge bends in the layout of both trees. Further, the nodes and edges of the top-down subtree of T_2 that is isomorphic to T_1 are also highlighted.

202b

```

⟨show subtree isomorphism 202b⟩≡
{ node v;
  forall_nodes(v,T1)
  gw2.set_color(M[v],blue);
}

```

```

edge e;
forall_edges(e,T2) {
  if ( gw2.get_color(T2.source(e)) ≡ blue ∧
      gw2.get_color(T2.target(e)) ≡ blue ) {
    gw2.set_color(e,blue);
    gw2.set_width(e,2);
  }
}

gw1.get_position(pos1);
gw1.set_layout(pos1); // remove edge bends
gw2.get_position(pos2);
gw2.set_layout(pos2); // remove edge bends
forall_nodes(v,T1)
  pos1[v] = pos2[M[v]];
gw1.set_position(pos1);
gw1.set_layout(pos1);
}
}

```

A simple checker for top-down isomorphic unordered subtrees that provides some visual reassurance consists of highlighting the nodes and edges of a top-down subtree of T_2 that is isomorphic to T_1 .

203

```

⟨demo subtree isomorphism 202a⟩ +≡
void gw_top_down_unordered_subtree_isomorphism(
  GraphWin& gw1)
{
  GRAPH<string,string>& G1 = gw1.get_graph();
  TREE<string,string> T1(G1);

  GRAPH<string,string> G2;
  GraphWin gw2(G2,500,500,
    "Top-Down Unordered Subtree Isomorphism");
  gw2.display();
  gw2.message("Enter second tree. Press done when finished");
  gw2.edit();
  gw2.del_message();

  TREE<string,string> T2(G2);

  node_array<node> M(T1);
  bool isomorph = top_down_unordered_subtree_isomorphism(T1,T2,M);

  panel P;
  if (isomorph) {
    make_proof_panel(P,"There is an isomorphic subtree",true);
    if (gw1.open_panel(P)) { // proof button pressed
      gw1.save_all_attributes();
      node_array<point> pos1(T1);
      node_array<point> pos2(T2);
      ⟨show subtree isomorphism 202b⟩
      gw1.wait();
      gw1.restore_all_attributes();
    }
  }
}

```

```

} } else {
    make_proof_panel(P,"There is \\red no isomorphic subtree",false);
    gw1.open_panel(P);
}
}

```

A simple checker for bottom-up isomorphic subtrees that provides some visual reassurance consists of highlighting the nodes and edges of each of the bottom-up subtrees of T_2 that are isomorphic to T_1 . Since by Lemma 4.29 these isomorphic subtrees must be pairwise node-disjoint, they are all highlighted at the same time.

204 ⟨demo subtree isomorphism 202a⟩ +≡

```

void gw_bottom_up_ordered_subtree_isomorphism(
    GraphWin& gwI)
{
    GRAPH<string,string>& G1 = gwI.get_graph();
    TREE<string,string> T1(G1);

    GRAPH<string,string> G2;
    GraphWin gw2(G2,500,500,
        "Bottom-Up Ordered Subtree Isomorphism");
    gw2.display();
    gw2.message("Enter second tree. Press done when finished");
    gw2.edit();
    gw2.del_message();

    TREE<string,string> T2(G2);

    list<node_array<node> > L;
    bottom_up_ordered_subtree_isomorphism(T1,T2,L);

    gw2.message(string("There are %i isomorphic subtrees",
        L.size()));
    node v;
    node_array<node> M(T1);
    forall(M,L) {
        forall_nodes(v,T1) {
            gw2.set_color(M[v],blue);
        }
        edge e;
        forall_edges(e,T2) {
            if ( gw2.get_color(T2.source(e)) ≡ blue ∧
                gw2.get_color(T2.target(e)) ≡ blue ) {
                gw2.set_color(e,blue);
                gw2.set_width(e,2);
            }
        }
        gw2.wait();
        gw2.del_message();
    }
}

```

A simple checker for bottom-up unordered isomorphic subtrees that provides some visual reassurance consists of highlighting the nodes and edges of each of the bottom-up subtrees of T_2 that are isomorphic to T_1 . Again, since by Lemma 4.29 these isomorphic subtrees must be pairwise node-disjoint, they are all highlighted at the same time.

205

```
(demo subtree isomorphism 202a) +≡
void gw_bottom_up_unordered_subtree_isomorphism(
    GraphWin& gw1)
{
    GRAPH<string,string>& G1 = gw1.get_graph();
    TREE<string,string> T1(G1);

    GRAPH<string,string> G2;
    GraphWin gw2(G2,500,500,
        "Bottom-up Unordered Subtree Isomorphism");
    gw2.display();
    gw2.message("Enter second tree. Press done when finished");
    gw2.edit();
    gw2.del_message();

    TREE<string,string> T2(G2);

    node_array<int> C1(T1);
    node_array<int> C2(T2);
    list<node_array<node>> L;
    bottom_up_unordered_subtree_isomorphism(T1,T2,C1,C2,L);

    gw2.message(string("There are %i isomorphic subtrees",
        L.size()));

    node v,w;
    forall_nodes(v,T1)
        gw1.set_label(v,string("%i",C1[v]));
    forall_nodes(w,T2)
        gw2.set_label(w,string("%i",C2[w]));

    node_array<node> M(T1);
    forall(M,L) {
        forall_nodes(v,T1) {
            if (M[v] ≠ nil) {
                gw2.set_color(M[v],blue);
            }
        }
    }

    edge e;
    forall_edges(e,T2) {
        if (gw2.get_color(T2.source(e)) ≡ blue ∧
            gw2.get_color(T2.target(e)) ≡ blue) {
            gw2.set_color(e,blue);
            gw2.set_width(e,2);
        }
    }
}
```

```

    gw2.wait();
    gw2.del_message();
}

```

4.3 Maximum Common Subtree Isomorphism

Another important generalization of tree isomorphism which also generalizes subtree isomorphism, is known as maximum common subtree isomorphism. The maximum common subtree isomorphism problem consists in finding a largest common subtree between two trees, and is also a fundamental problem with a variety of applications in engineering and life sciences.

Since trees can be either ordered or unordered and, further, there are several different notions of subtree, there are different notions of maximum common subtree isomorphism.

4.3.1 Top-Down Maximum Common Subtree Isomorphism

A top-down common subtree of two ordered trees T_1 and T_2 is an ordered tree T such that there are top-down ordered subtree isomorphisms of T into T_1 and into T_2 . A maximal top-down common subtree of two ordered trees T_1 and T_2 is a top-down common subtree of T_1 and T_2 which is not a proper subtree of any other top-down common subtree of T_1 and T_2 . A top-down maximum common subtree of two ordered trees T_1 and T_2 is a top-down common subtree of T_1 and T_2 with the largest number of nodes.

Definition 4.40. A *top-down common subtree* of an ordered tree $T_1 = (V_1, E_1)$ to another ordered tree $T_2 = (V_2, E_2)$ is a structure (X_1, X_2, M) , where $X_1 = (W_1, S_1)$ is a top-down ordered subtree of T_1 , $X_2 = (W_2, S_2)$ is a top-down ordered subtree of T_2 , and $M \subseteq W_1 \times W_2$ is an ordered tree isomorphism of X_1 to X_2 . A top-down common subtree (X_1, X_2, M) of T_1 to T_2 is *maximal* if there is no top-down common subtree (X'_1, X'_2, M') of T_1 to T_2 such that X_1 is a proper top-down subtree of X'_1 and X_2 is a proper top-down subtree of X'_2 , and it is *maximum* if there is no top-down common subtree (X'_1, X'_2, M') of T_1 to T_2 with $\text{size}[X_1] < \text{size}[X'_1]$.

The following remark applies to top-down and bottom-up common subtrees as well, both ordered and unordered.

Remark 4.41. Let (X_1, X_2, M) be a top-down common subtree of an ordered tree $T_1 = (V_1, E_1)$ to another ordered tree $T_2 = (V_2, E_2)$, where $X_1 = (W_1, S_1)$, $X_2 = (W_2, S_2)$, and $M \subseteq W_1 \times W_2$. Since $W_1 \subseteq V_1$ and $W_2 \subseteq V_2$, the injection $M \subseteq W_1 \times W_2$ is also a partial injection $M \subseteq V_1 \times V_2$.

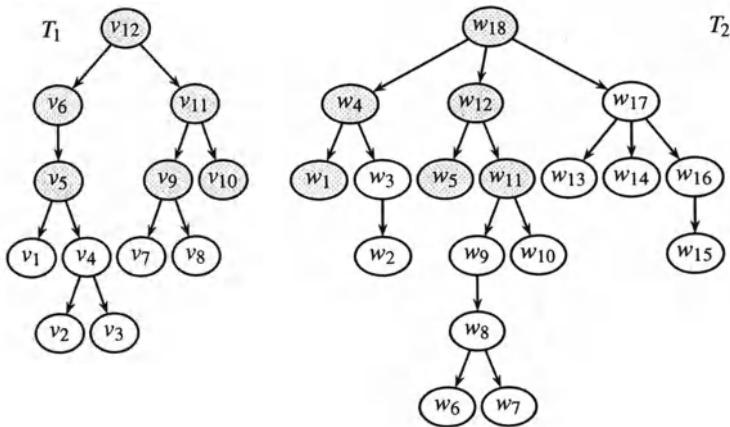


Fig. 4.13. A top-down maximum common subtree of two ordered trees. Nodes are numbered according to the order in which they are visited during a postorder traversal. The common subtree of T_1 and T_2 is shown highlighted in both trees.

Example 4.42. For the ordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ in Fig. 4.13, the partial injection $M \subseteq V_1 \times V_2$ given by $M = \{(v_5, w_1), (v_6, w_4), (v_9, w_5), (v_{10}, w_{11}), (v_{11}, w_{12}), (v_{12}, w_{18})\}$ is a top-down maximum common subtree isomorphism of T_1 to T_2 .

Now, since a top-down ordered subtree is a top-down subtree in which the previous sibling (if any) of each of the nodes in the subtree also belongs to the subtree, the notions of top-down maximal and maximum common subtree isomorphism coincide for ordered trees.

Lemma 4.43. Let (X_1, X_2, M) be a top-down common subtree of an ordered tree $T_1 = (V_1, E_1)$ to another ordered tree $T_2 = (V_2, E_2)$. Then,

(X_1, X_2, M) is a top-down maximal common subtree of T_1 to T_2 if and only if it is a top-down maximum common subtree of T_1 to T_2 .

Proof. Let (X_1, X_2, M) be a maximal top-down common subtree of an ordered tree $T_1 = (V_1, E_1)$ to another ordered tree $T_2 = (V_2, E_2)$, where $X_1 = (W_1, S_1)$, $X_2 = (W_2, S_2)$, and $M \subseteq W_1 \times W_2$, and suppose it is not maximum. Let also (X'_1, X'_2, M') be a maximum top-down common subtree of T_1 to T_2 , where $X'_1 = (W'_1, S'_1)$, $X'_2 = (W'_2, S'_2)$, and $M' \subseteq W'_1 \times W'_2$. Suppose $M \not\subseteq M'$, and let $(v, w) \in M \setminus M'$. Let also $W''_1 = \{v\} \cup \{\text{parent}[x] \mid x \in W'_1, x \neq \text{root}[X'_1]\} \cup \{\text{previous}[x] \mid x \in W'_1, x \neq \text{root}[X'_1], x \neq \text{first}[\text{parent}[x]]\}$, let $W''_2 = \{w\} \cup \{\text{parent}[y] \mid y \in W'_2, y \neq \text{root}[X'_2]\} \cup \{\text{previous}[y] \mid y \in W'_2, y \neq \text{root}[X'_2], y \neq \text{first}[\text{parent}[y]]\}$, and let X''_1 and X''_2 be the top-down subtrees of T_1 and T_2 induced respectively by W''_1 and W''_2 . Let also $M'' = \{(v, w) \in M' \mid v \in W''_1, w \in W''_2\}$.

Then, $(X_1 \cup X''_1, X_2 \cup X''_2, M \cup M'')$ is a top-down common subtree of T_1 to T_2 , contradicting the hypothesis that (X_1, X_2, M) is maximal. Therefore, $M \subseteq M'$ and, since (X_1, X_2, M) is maximal, it must be true that $M = M'$. Therefore, (X_1, X_2, M) is also a top-down maximum common subtree of T_1 to T_2 .

Let now (X_1, X_2, M) be a maximum top-down common subtree of T_1 to T_2 , and suppose it is not maximal. Then, there are nodes $v \in V_1 \setminus W_1$ and $w \in V_2 \setminus W_2$ such that (X'_1, X'_2, M') is a top-down common subtree of T_1 to T_2 , where $X'_1 = (W_1 \cup \{v\}, S_1 \cup \{(\text{parent}[v], v)\})$, $X'_2 = (W_2 \cup \{w\}, S_2 \cup \{(\text{parent}[w], w)\})$, and $M' = M \cup \{(v, w)\}$. But $\text{size}[X'_1] = \text{size}[X_1] + 1$, contradicting the hypothesis that (X_1, X_2, M) is maximum. Therefore, (X_1, X_2, M) is also a top-down maximal common subtree of T_1 to T_2 . \square

An Algorithm for Top-Down Ordered Maximum Common Subtree Isomorphism

A top-down maximum common subtree isomorphism of an ordered tree T_1 to another ordered tree T_2 can be obtained by performing a simultaneous traversal of the two trees, in much the same way as done in Sect. 4.1 for testing isomorphism of two ordered trees and in Sect. 4.2 for finding ordered subtree isomorphisms.

The following algorithm performs a simultaneous preorder traversal of a top-down subtree of the ordered tree T_1 and a top-down subtree of the ordered tree T_2 , collecting also a partial mapping M of nodes of T_1 to nodes of T_2 .

209a \langle maximum common subtree isomorphism 209a $\rangle \equiv$

```
void top_down_ordered_max_common_subtree_isomorphism(
    const TREE<string,string>& T1,
    const TREE<string,string>& T2,
    node_array<node>& M)
{
    if ( map_ordered_common_subtree(T1,T1.root(),T2,T2.root(),M) ) {
        (double-check top-down ordered maximum common subtree isomorphism 210a)
    }
}
```

The simultaneous preorder traversal proceeds as long as a top-down ordered subtree of T_1 is isomorphic to a top-down ordered subtree of T_2 , but otherwise stops as soon as the top-down ordered structures of T_1 and T_2 differ.

209b \langle maximum common subtree isomorphism 209a $\rangle + \equiv$

```
bool map_ordered_common_subtree(
    const TREE<string,string>& T1,
    const node r1,
    const TREE<string,string>& T2,
    const node r2,
    node_array<node>& M)
{
    if ( T1[r1] != T2[r2] ) {
        return false;
    } else {
        M[r1] = r2;
        if ( !T1.is_leaf(r1) & !T2.is_leaf(r2) ) {
            node v = T1.first_child(r1);
            node w = T2.first_child(r2);
            bool res = true;
            while ( res ) {
                res = map_ordered_common_subtree(T1,v,T2,w,M);
                if ( T1.is_last_child(v) ) break;
                v = T1.next_sibling(v);
                if ( T2.is_last_child(w) ) break;
                w = T2.next_sibling(w);
            }
        }
        return true;
    }
}
```

The following double-check of top-down ordered maximum common subtree isomorphism, although being redundant, gives some reassurance of the correctness of the implementation. It verifies that M

is a top-down ordered maximum common subtree isomorphism of T_1 to T_2 , according to Definition 4.40.

210a ⟨double-check top-down ordered maximum common subtree isomorphism 210a⟩ \equiv

```

{ node v;
  if( M[T1.root()] ≠ T2.root() )
    error_handler(1,
      "Wrong implementation of common subtree isomorphism");
  forall_nodes(v,T1) {
    if( M[v] ≠ nil ∧ T1[v] ≠ T2[M[v]] )
      error_handler(1,
        "Wrong implementation of common subtree isomorphism");
    if( M[v] ≠ nil ∧
        ¬T1.is_leaf(v) ∧
        ¬T2.is_leaf(M[v]) ∧
        M[T1.first_child(v)] ≠ nil ∧
        M[T1.first_child(v)] ≠ T2.first_child(M[v]) )
      error_handler(1,
        "Wrong implementation of common subtree isomorphism");
    if( M[v] ≠ nil ∧
        ¬T1.is_last_child(v) ∧
        ¬T2.is_last_child(M[v]) ∧
        M[T1.next_sibling(v)] ≠ nil ∧
        M[T1.next_sibling(v)] ≠ T2.next_sibling(M[v]) )
      error_handler(1,
        "Wrong implementation of common subtree isomorphism");
  }
}
```

Further, the common subtree M found is also verified to be maximal, which implies by Lemma 4.43 that M is, in fact, a maximum common subtree isomorphism of T_1 to T_2 .

210b ⟨double-check top-down ordered maximum common subtree isomorphism 210a⟩ $+≡$

```

{ node x;
  forall_nodes(x,T1) {
    if( M[x] ≠ nil ) {
      if( ¬T1.is_leaf(x) ∧
          M[T1.first_child(x)] ≡ nil ∧
          ¬T2.is_leaf(M[x]) ∧
          T1[T1.first_child(x)] ≡ T2[T2.first_child(M[x])] )
        error_handler(1,
          "Wrong implementation of common subtree isomorphism");
      if( ¬T1.is_last_child(x) ∧
          M[T1.next_sibling(x)] ≡ nil ∧
          ¬T2.is_last_child(M[x]) ∧
          T1[T1.next_sibling(x)] ≡ T2[T2.next_sibling(M[x])] )
        error_handler(1,
          "Wrong implementation of common subtree isomorphism");
    }
  }
}
```

Lemma 4.44. *Let T_1 and T_2 be ordered trees with respectively n_1 and n_2 nodes, where $n_1 \leq n_2$. The algorithm for top-down ordered maximum common subtree isomorphism runs in $O(n_1)$ time using $O(n_1)$ additional space.*

Proof. Let T_1 and T_2 be ordered trees with respectively n_1 and n_2 nodes, where $n_1 \leq n_2$. The algorithm makes $O(n_1)$ recursive calls, one for each nonleaf node of T_1 and although within a recursive call on some node, the effort spent is not bounded by a constant but is proportional to the number of children of the node, the total effort spent over all nonleaf nodes of T_1 is proportional to n_1 and the algorithm runs in $O(n_1)$ time. Further, $O(n_1)$ additional space is used.

The double-check of top-down ordered maximum common subtree isomorphism runs in $O(n_1)$ time using $O(1)$ additional space. \square

4.3.2 Top-Down Unordered Maximum Common Subtree Isomorphism

A top-down common subtree of two unordered trees T_1 and T_2 is an unordered tree T such that there are top-down unordered subtree isomorphisms of T into T_1 and into T_2 . A maximal top-down common subtree of two unordered trees T_1 and T_2 is a top-down common subtree of T_1 and T_2 which is not a proper subtree of any other top-down common subtree of T_1 and T_2 . A top-down maximum common subtree of two unordered trees T_1 and T_2 is a top-down common subtree of T_1 and T_2 with the largest number of nodes.

Definition 4.45. *A **top-down common subtree** of an unordered tree $T_1 = (V_1, E_1)$ to another unordered tree $T_2 = (V_2, E_2)$ is a structure (X_1, X_2, M) , where $X_1 = (W_1, S_1)$ is a top-down unordered subtree of T_1 , $X_2 = (W_2, S_2)$ is a top-down unordered subtree of T_2 , and $M \subseteq W_1 \times W_2$ is an unordered tree isomorphism of X_1 to X_2 . A top-down common subtree (X_1, X_2, M) of T_1 to T_2 is **maximal** if there is no top-down common subtree (X'_1, X'_2, M') of T_1 to T_2 such that X_1 is a proper top-down subtree of X'_1 and X_2 is a proper top-down subtree of X'_2 , and it is **maximum** if there is no top-down common subtree (X'_1, X'_2, M') of T_1 to T_2 with $\text{size}[X_1] < \text{size}[X'_1]$.*

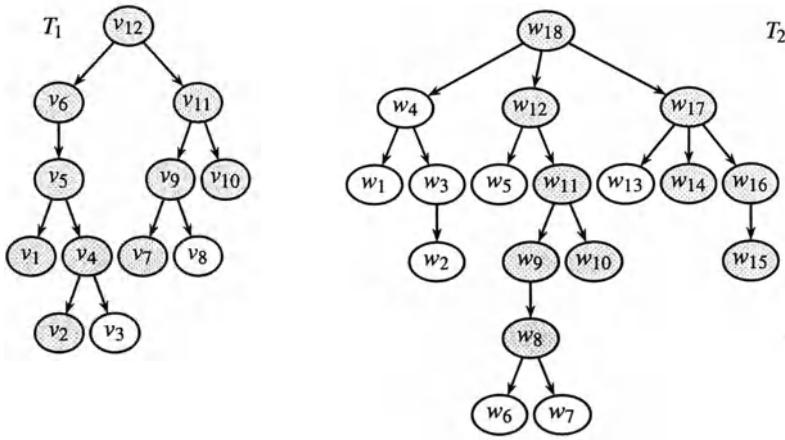


Fig. 4.14. A top-down maximum common subtree of two unordered trees. Nodes are numbered according to the order in which they are visited during a postorder traversal. The common subtree of T_1 and T_2 is shown highlighted in both trees.

Example 4.46. For the unordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ shown in Fig. 4.14, the partial injection $M \subseteq V_1 \times V_2$ given by $M = \{(v_1, w_{10}), (v_2, w_8), (v_4, w_9), (v_5, w_{11}), (v_6, w_{12}), (v_7, w_{15}), (v_9, w_{16}), (v_{10}, w_{14}), (v_{11}, w_{17}), (v_{12}, w_{18})\}$ is a top-down unordered maximum common subtree isomorphism of T_1 and T_2 .

Unlike the case of ordered trees, though, the notions of top-down maximal and maximum common subtree isomorphism do not coincide for ordered trees. Nevertheless, top-down unordered maximum common subtree isomorphisms are still maximal.

Lemma 4.47. *Let (X_1, X_2, M) be a maximum top-down common subtree of an unordered tree $T_1 = (V_1, E_1)$ to another unordered tree $T_2 = (V_2, E_2)$. Then, (X_1, X_2, M) is also a top-down maximal common subtree of T_1 to T_2 .*

Proof. Let (X_1, X_2, M) be a maximum top-down common subtree of an unordered tree $T_1 = (V_1, E_1)$ to another unordered tree $T_2 = (V_2, E_2)$, where $X_1 = (W_1, S_1)$, $X_2 = (W_2, S_2)$, and $M \subseteq W_1 \times W_2$.

Suppose (X_1, X_2, M) is not maximal. Then, there are nodes $v \in V_1 \setminus W_1$ and $w \in V_2 \setminus W_2$ such that (X'_1, X'_2, M') is a top-down common subtree of T_1 to T_2 , where $X'_1 = (W_1 \cup \{v\}, S_1 \cup \{(\text{parent}[v], v)\})$, $X'_2 = (W_2 \cup \{w\}, S_2 \cup \{(\text{parent}[w], w)\})$, and $M' = M \cup \{(v, w)\}$. But

$\text{size}[X'_1] = \text{size}[X_1] + 1$, contradicting the hypothesis that (X_1, X_2, M) is maximum. Therefore, (X_1, X_2, M) is also a top-down maximal common subtree of T_1 to T_2 . \square

An Algorithm for Top-Down Unordered Maximum Common Subtree Isomorphism

A top-down maximum common subtree isomorphism of an unordered tree T_1 into another unordered tree T_2 can be constructed from maximum common subtree isomorphisms of each of the subtrees of T_1 rooted at the children of node v into each of the subtrees of T_2 rooted at the children of node w , provided that these common isomorphic subtrees do not overlap. As in the case of top-down unordered subtree isomorphism, this decomposition property allows application of the divide-and-conquer technique, yielding again a simple recursive algorithm.

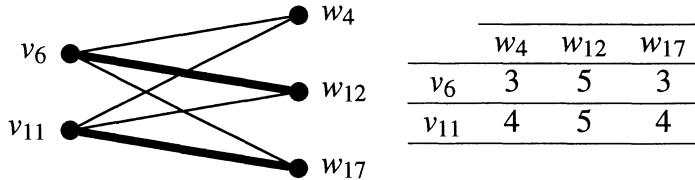
Consider first the problem of determining the size of a top-down unordered maximum common subtree isomorphism of T_1 and T_2 , postponing for a while the discussion about construction of an actual maximum common subtree isomorphism mapping M of T_1 to T_2 . If node v is a leaf in T_1 or node w is a leaf in T_2 , mapping node v to node w yields a maximum common subtree of size 1, provided that $T_1[v] = T_2[w]$.

Otherwise, let p be the number of children of node v in T_1 , and let q be the number of children of node w in T_2 . Let also v_1, \dots, v_p and w_1, \dots, w_q be the children of nodes v and w , respectively. Build a bipartite graph $G = (\{v_1, \dots, v_p\}, \{w_1, \dots, w_q\}, E)$ on $p + q$ vertices, with edge $(v_i, w_j) \in E$ if and only if the size of a maximum common subtree of the subtree of T_1 rooted at node v_i and the subtree of T_2 rooted at node w_j is nonzero and, in such a case, with edge $(v_i, w_j) \in E$ weighted by that nonzero size.

Then, a maximum common subtree of the subtree of T_1 rooted at node v and the subtree of T_2 rooted at node w has size equal to one plus the weight of a maximum weight bipartite matching in G .

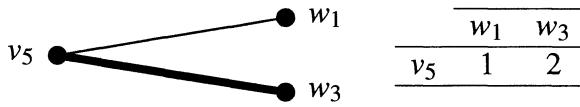
Example 4.48. Consider the execution of the top-down unordered maximum common subtree isomorphism procedure, upon the unordered trees of Fig. 4.14. In order to find the size of a top-down

unordered maximum common subtree of the subtree of T_1 rooted at node v_{12} and the subtree of T_2 rooted at node w_{18} , the following maximum weight bipartite matching problem is solved:

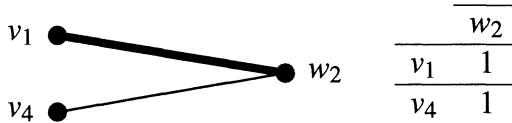


A solution to this maximum weight bipartite matching problem has weight $5 + 4 = 9$ and then, a maximum common subtree of the subtree of T_1 rooted at node v_{12} and the subtree of T_2 rooted at node w_{18} has $9 + 1 = 10$ nodes. However, stating this bipartite matching problem involves (recursively) solving further maximum weight bipartite matching problems.

First, in order to find the size of a maximum common subtree of the subtree of T_1 rooted at node v_6 and the subtree of T_2 rooted at node w_4 , the following maximum weight bipartite matching problem is also solved



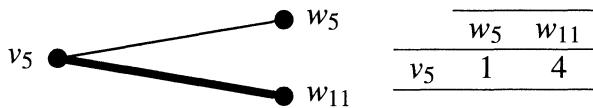
which, in turn, requires solving the following maximum weight bipartite matching problem, in order to find the size of a maximum common subtree of the subtree of T_1 rooted at node v_5 and the subtree of T_2 rooted at node w_3 :



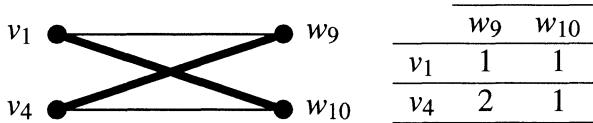
Since the latter (trivial) bipartite matching problem has a solution of weight 1, the former bipartite matching problem has a solution of weight 2, and a maximum common subtree of the subtree of T_1 rooted

at node v_6 and the subtree of T_2 rooted at node w_4 has, in fact, $2 + 1 = 3$ nodes.

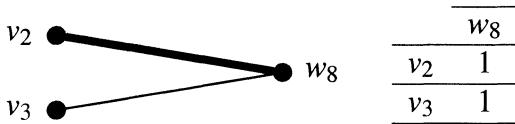
Next, in order to find the size of a maximum common subtree of the subtree of T_1 rooted at node v_6 and the subtree of T_2 rooted at node w_{12} , the following maximum weight bipartite matching problem is also solved



which, in turn, requires solving the following maximum weight bipartite matching problem, in order to find the size of a maximum common subtree of the subtree of T_1 rooted at node v_5 and the subtree of T_2 rooted at node w_{11} :



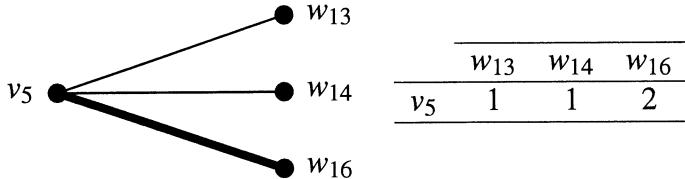
This, in turn, requires solving the following maximum weight bipartite matching problem, in order to find the size of a maximum common subtree of the subtree of T_1 rooted at node v_4 and the subtree of T_2 rooted at node w_9 :



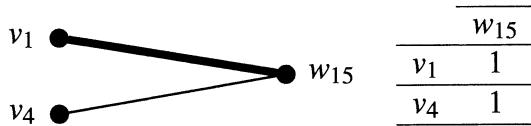
Since the latter (trivial) bipartite matching problem has a solution of weight 1, the previous bipartite matching problem has a solution of weight $2 + 1 = 3$, the former bipartite matching problem has a solution of weight 4, and a maximum common subtree of the subtree of T_1 rooted at node v_6 and the subtree of T_2 rooted at node w_{12} has thus $4 + 1 = 5$ nodes.

Now, in order to find the size of a maximum common subtree of the subtree of T_1 rooted at node v_6 and the subtree of T_2 rooted at

node w_{17} , the following maximum weight bipartite matching problem is also solved

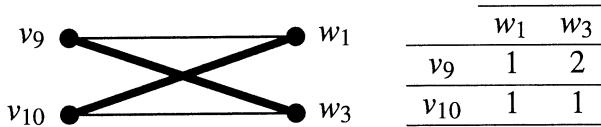


which, in turn, requires solving the following maximum weight bipartite matching problem, in order to find the size of a maximum common subtree of the subtree of T_1 rooted at node v_5 and the subtree of T_2 rooted at node w_{16} :

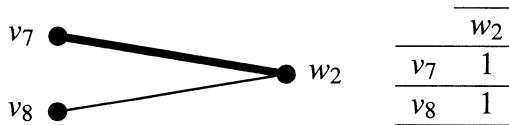


Since the latter (trivial) bipartite matching problem has a solution of weight 1, the former bipartite matching problem has a solution of weight 2, and a maximum common subtree of the subtree of T_1 rooted at node v_6 and the subtree of T_2 rooted at node w_{17} has thus $2 + 1 = 3$ nodes.

In the same way, in order to find the size of a maximum common subtree of the subtree of T_1 rooted at node v_{11} and the subtree of T_2 rooted at node w_4 , the following maximum weight bipartite matching problem is solved

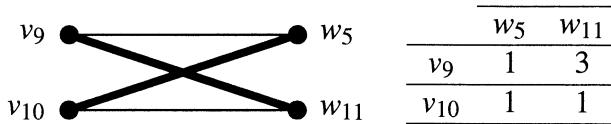


which, in turn, requires solving the following maximum weight bipartite matching problem, in order to find the size of a maximum common subtree of the subtree of T_1 rooted at node v_9 and the subtree of T_2 rooted at node w_3 :

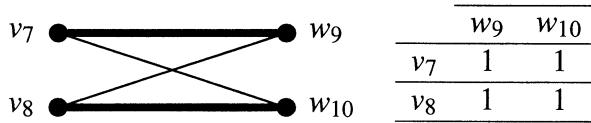


Since the latter (trivial) bipartite matching problem has a solution of weight 1, the former bipartite matching problem has a solution of weight $2 + 1 = 3$, and a maximum common subtree of the subtree of T_1 rooted at node v_{11} and the subtree of T_2 rooted at node w_4 has thus $3 + 1 = 4$ nodes.

Now, in order to find the size of a maximum common subtree of the subtree of T_1 rooted at node v_{11} and the subtree of T_2 rooted at node w_{12} , the following maximum weight bipartite matching problem is solved

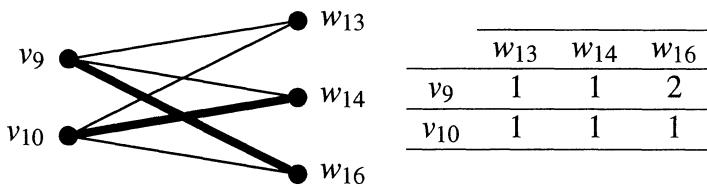


which, in turn, requires solving the following maximum weight bipartite matching problem, in order to find the size of a maximum common subtree of the subtree of T_1 rooted at node v_9 and the subtree of T_2 rooted at node w_{11} :

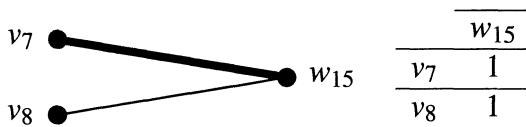


Since the latter (trivial) bipartite matching problem has a solution of weight $1 + 1 = 2$, the former bipartite matching problem has a solution of weight $3 + 1 = 4$, and a maximum common subtree of the subtree of T_1 rooted at node v_{11} and the subtree of T_2 rooted at node w_{12} has thus $4 + 1 = 5$ nodes.

Finally, in order to find the size of a maximum common subtree of the subtree of T_1 rooted at node v_{11} and the subtree of T_2 rooted at node w_{17} , the following maximum weight bipartite matching problem is also solved



which, in turn, requires solving the following maximum weight bipartite matching problem, in order to find the size of a maximum common subtree of the subtree of T_1 rooted at node v_9 and the subtree of T_2 rooted at node w_{16} :



Since the latter (trivial) bipartite matching problem has a solution of weight 1, the former bipartite matching problem has a solution of weight $2 + 1 = 3$, and a maximum common subtree of the subtree of T_1 rooted at node v_9 and the subtree of T_2 rooted at node w_{16} has thus $3 + 1 = 4$ nodes.

Now, the previous procedure can be extended with the construction of an actual top-down unordered maximum common subtree isomorphism mapping M of T_1 into T_2 , based on the following result. Notice first that the solution to a maximum weight bipartite matching problem on a subtree (W_1, S_1) of T_1 and a subtree (W_2, S_2) of T_2 is a set of (weighted) edges of the corresponding bipartite graph, that is, a set of ordered pairs of nodes $B \in W_1 \times W_2$. Since $W_1 \subseteq V_1$ and $W_2 \subseteq V_2$, it also holds that $B \in V_1 \times V_2$.

Lemma 4.49. *Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be unordered trees, and let $B \subseteq V_1 \times V_2$ be the solutions to all the maximum weight bipartite matching problems solved during the top-down unordered maximum common subtree isomorphism procedure upon T_1 and T_2 . Then, there is a unique top-down unordered maximum common subtree isomorphism $M \in V_1 \times V_2$ such that $M \subseteq B$.*

Proof. Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be unordered trees, and let $B \subseteq V_1 \times V_2$ be the corresponding solutions to maximum weight bipartite matching problems. Showing existence and uniqueness of a top-down unordered maximum common subtree isomorphism $M \subseteq B$ is equivalent to showing that, for each node $v \in V_1$ with $(\text{parent}(v), z) \in B$ for some node $z \in V_2$, there is a unique $(v, w) \in B$ such that $\text{parent}(w) = z$, because a unique top-down subtree of T_2 isomorphic to a top-down subtree of T_1 can then be reconstructed by order of nondecreasing depth. Therefore, it suffices to show the weaker condition that, for all $(v, w_1), (v, w_2) \in B$ with $w_1 \neq w_2$, it holds that $\text{parent}(w_1) \neq \text{parent}(w_2)$.

Let $(v, w_1), (v, w_2) \in B$ with $w_1 \neq w_2$. Then, (v, w_1) and (v, w_2) must belong to the solution to different bipartite matching problems, for otherwise the corresponding edges in the bipartite graph would not be independent. But if (v, w_1) and (v, w_2) belong to the solution to different bipartite matching problems then nodes w_1 and w_2 cannot be siblings, because in each bipartite matching problem all children of some node of T_1 are matched against all children of some node of T_2 . Therefore, $\text{parent}(w_1) \neq \text{parent}(w_2)$. \square

Given the solutions $B \subseteq V_1 \times V_2$ to all the maximum weight bipartite matching problems solved during the top-down unordered maximum common subtree isomorphism procedure upon unordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$, the corresponding maximum common subtree isomorphism mapping $M \subseteq V_1 \times V_2$ can be reconstructed as follows. Set $M[\text{root}[T_1]]$ to $\text{root}[T_2]$ and, for all nodes $v \in V_1$ in preorder, set $M[v]$ to the unique node w with $(v, w) \in B$ and $(\text{parent}(v), \text{parent}(w)) \in B$.

Example 4.50. Consider the execution shown in Example 4.48 of the top-down unordered maximum common subtree isomorphism procedure, upon the unordered trees of Fig. 4.14. The solutions $B \subseteq V_1 \times V_2$ to all the maximum weight bipartite matching problems solved are as follows.

$v_1 : w_2$	w_{10}	w_{15}		$v_7 : w_2$	w_9	w_{15}
$v_2 : w_8$				$v_8 : w_{10}$		
$v_3 :$				$v_9 : w_3$	w_{11}	w_{16}
$v_4 : w_9$				$v_{10} : w_1$	w_5	w_{14}
$v_5 : w_3$	w_{11}	w_{16}		$v_{11} : w_{17}$		
$v_6 : w_{12}$				$v_{12} : w_{18}$		

The unique maximum common subtree isomorphism mapping $M \subseteq B \subseteq V_1 \times V_2$ is indicated by the encircled nodes of T_2 .

The following algorithm implements the previous procedure for top-down maximum common subtree isomorphism of an unordered tree T_1 into another unordered tree T_2 . The partial injection $M \subseteq V_1 \times V_2$ computed by the procedure upon two unordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ is represented by an array of nodes (of tree T_2) indexed by the nodes of tree T_1 .

Notice first that a top-down subtree of an unordered labeled tree is isomorphic to a top-down subtree of another unordered labeled tree if the unlabeled trees underlying the subtrees of the labeled trees are isomorphic and, furthermore, corresponding nodes share the same label.

Further, as in the case of top-down unordered subtree isomorphism, the correspondence between children nodes in the trees and the vertices of the bipartite graph is represented by two maps of vertices (of the bipartite graph) indexed respectively by the nodes of trees T_1 and T_2 , in one direction, and in the opposite direction by a map of nodes (of trees T_1 and T_2) indexed by the vertices of the bipartite graph.

220

```
(maximum common subtree isomorphism 209a)+≡
int top_down_unordered_max_common_subtree_isomorphism(
    const TREE<string,string>& T1,
    node r1,
    const TREE<string,string>& T2,
    node r2,
    node_array<set<node>>& B)
{
    if ( T1[r1] ≠ T2[r2] ) return 0;

    if ( T1.is_leaf(r1) ∨ T2.is_leaf(r2) ) return T1[r1] ≡ T2[r2];

    int p = T1.number_of_children(r1);
    int q = T2.number_of_children(r2);
```

```

node_map<node> T1G(T1);
node_map<node> T2G(T2);
graph G;
node_map<node> GT(G);
edge_map<int> WEIGHT(G);

list<node> U;
node v1,v2;
forall_children(v1,r1) {
    node v = G.new_node();
    U.append(v);
    GT[v] = v1;
    T1G[v1] = v;
}
list<node> W;
forall_children(v2,r2) {
    node w = G.new_node();
    W.append(w);
    GT[w] = v2;
    T2G[v2] = w;
}
edge e;
forall_children(v1,r1) {
    forall_children(v2,r2) {
        int res = top_down_unordered_max_common_subtree_isomorphism
            (T1,v1,T2,v2,B);
        if (res ≠ 0) {
            e = G.new_edge(T1G[v1],T2G[v2]);
            WEIGHT[e] = res;
        }
    }
}

node_array<int> POT(G);
list<edge> L = MAX_WEIGHT_BIPARTITE_MATCHING(G,U,W,WEIGHT,POT);

int res = 1; // mapping of r1 to r2
forall(e,L) {
    B[GT[G.source(e)]].insert(GT[G.target(e)]);
    res += WEIGHT[e];
}
return res;
}

```

The following algorithm implements the previous procedure for top-down maximum common subtree isomorphism of an unordered tree T_1 into another unordered tree T_2 , calling the previous recursive procedure upon the root of T_1 and the root of T_2 .

```

const TREE<string,string>& T1,
const TREE<string,string>& T2,
node_array<node>& M)
{
    node r1 = T1.root();
    node r2 = T2.root();
    if ( T1[r1] ≡ T2[r2] ) {
        node_array<set<node>> B(T1);
        B[r1].insert(r2);
        top_down_unordered_max_common_subtree_isomorphism(T1,r1,T2,r2,B);
        <reconstruct max common unordered subtree isomorphism 222a>
        <double-check top-down unordered maximum common subtree isomorphism 222b>
    } }
}

```

The following procedure reconstructs the top-down unordered maximum common subtree isomorphism mapping $M \subseteq V_1 \times V_2$ included in the solutions $B \subseteq V_1 \times V_2$ to all the maximum weight bipartite matching problems. The root of T_1 is mapped to the root of T_2 and, during a preorder traversal, each nonroot node $v \in V_1$ is mapped to the unique node $w \in V_2$ with $(v, w) \in B$ and $(\text{parent}(v), \text{parent}(w)) \in B$.

222a

<reconstruct max common unordered subtree isomorphism 222a>≡

```

M[r1] = r2;
{ node v,w;
list<node> L;
preorder_tree_list_traversal(T1,L);
L.pop(); // node r1 already mapped
forall(v,L) {
    forall(w,B[v]) {
        if ( M[T1.parent(v)] ≡ T2.parent(w) ) {
            M[v] = w;
            break;
        } } } }
}

```

The following double-check of maximum common subtree isomorphism, although being redundant, gives some reassurance of the correctness of the implementation. It verifies that M is a top-down unordered common subtree isomorphism of T_1 into T_2 , according to Definition 4.45. Notice that it is not checked that M is a top-down unordered *maximum* common subtree isomorphism of T_1 into T_2 , because this would require enumeration of all top-down unordered common subtree isomorphisms of T_1 into T_2 .

222b

<double-check top-down unordered maximum common subtree isomorphism 222b>≡

```

{ node v,w;
if ( M[T1.root()] ≠ T2.root() )
    error_handler(1,
}

```

```

    "Wrong implementation of common subtree isomorphism");
forall_nodes(v,T1) {
  if ( M[v] ≠ nil ) {
    if ( T1[v] ≠ T2[M[v]] )
      error_handler(1,
        "Wrong implementation of common subtree isomorphism");
    if ( ¬T1.is_root(v) ∧
        ¬T2.is_root(M[v]) ∧
        M[T1.parent(v)] ≠ T2.parent(M[v]) )
      error_handler(1,
        "Wrong implementation of common subtree isomorphism");
  }
}

```

Remark 4.51. Correctness of the algorithm for top-down unordered maximum common subtree isomorphism follows from Lemma 4.49.

Recall that the LEDA implementation of the maximum weight bipartite matching algorithm runs in $O(n(m + n \log n))$ time using $O(m)$ additional space. On a bipartite graph with $n = p + q$ nodes and $m = pq$ edges, the maximum weight bipartite matching algorithm runs in $O((p + q)(pq + (p + q) \log(p + q)))$ time using $O(pq)$ additional space.

Lemma 4.52. Let T_1 and T_2 be unordered trees with respectively n_1 and n_2 nodes, where $n_1 \leq n_2$. The algorithm for top-down unordered maximum common subtree isomorphism runs in $O((n_1 + n_2)(n_1 n_2 + (n_1 + n_2) \log(n_1 + n_2)))$ time using $O(n_1 n_2)$ additional space.

Proof. Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be unordered trees on respectively n_1 and n_2 nodes, and let $k(v)$ denote the number of children of node v . The effort spent on a leaf node of T_1 is $O(1)$, and the total effort spent on the leaves of T_1 is thus bounded by $O(n_1)$. The effort spent on a nonleaf v of T_1 and a nonleaf w of T_2 , on the other hand, is dominated by solving a maximum weight bipartite matching problem on a bipartite graph with at most $k(v) + k(w)$ nodes, where $k(v) \leq k(w)$, and is thus bounded by $O((k(v) + k(w))(k(v)k(w) + (k(v) + k(w))\log(k(v) + k(w))))$. The total effort spent on nonleaves of T_1 and T_2 is thus bounded by $O(\sum_{v \in V_1} \sum_{w \in V_2} (k(v) + k(w))(k(v)k(w) + (k(v) + k(w))\log(k(v) + k(w))))$.

$k(w) + (k(v) + k(w)) \log(k(v) + k(w)))$. Therefore, the algorithm runs in $O((n_1 + n_2)(n_1 n_2 + (n_1 + n_2) \log(n_1 + n_2)))$ time.

A similar argument shows that the algorithm uses $O(n_1 n_2)$ additional space. The double-check of top-down unordered subtree isomorphism runs in $O(n_1 n_2)$ time using $O(1)$ additional space. \square

The algorithm solves the problem of finding a top-down maximum common subtree isomorphism of an unordered tree into another unordered tree. The problem of enumerating all top-down unordered maximum common subtree isomorphisms can be solved with the maximum common subgraph isomorphism algorithms given in Sect. 7.3.

4.3.3 Bottom-Up Maximum Common Subtree Isomorphism

A bottom-up common subtree of two ordered trees T_1 and T_2 is an ordered tree T such that there are bottom-up ordered subtree isomorphisms of T into T_1 and into T_2 . A maximal bottom-up common subtree of two ordered trees T_1 and T_2 is a bottom-up common subtree of T_1 and T_2 which is not a proper subtree of any other bottom-up common subtree of T_1 and T_2 . A maximum bottom-up common subtree of two ordered trees T_1 and T_2 is a bottom-up common subtree of T_1 and T_2 with the largest number of nodes.

Definition 4.53. A *bottom-up common subtree* of an ordered tree $T_1 = (V_1, E_1)$ to another ordered tree $T_2 = (V_2, E_2)$ is a structure (X_1, X_2, M) , where $X_1 = (W_1, S_1)$ is a bottom-up ordered subtree of T_1 , $X_2 = (W_2, S_2)$ is a bottom-up ordered subtree of T_2 , and $M \subseteq W_1 \times W_2$ is an ordered tree isomorphism of X_1 to X_2 . A bottom-up common subtree (X_1, X_2, M) of T_1 to T_2 is **maximal** if there is no bottom-up common subtree (X'_1, X'_2, M') of T_1 to T_2 such that X_1 is a proper bottom-up subtree of X'_1 and X_2 is a proper bottom-up subtree of X'_2 , and it is **maximum** if there is no bottom-up common subtree (X'_1, X'_2, M') of T_1 to T_2 with $\text{size}[X_1] < \text{size}[X'_1]$.

Example 4.54. For the ordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ in Fig. 4.15, the partial injection $M \subseteq V_1 \times V_2$ given by $M = \{(v_9, w_6), (v_{10}, w_7), (v_{11}, w_8), (v_{12}, w_9), (v_{13}, w_{10}), (v_{14}, w_{11})\}$ is a bottom-up maximum common subtree isomorphism of T_1 to T_2 .

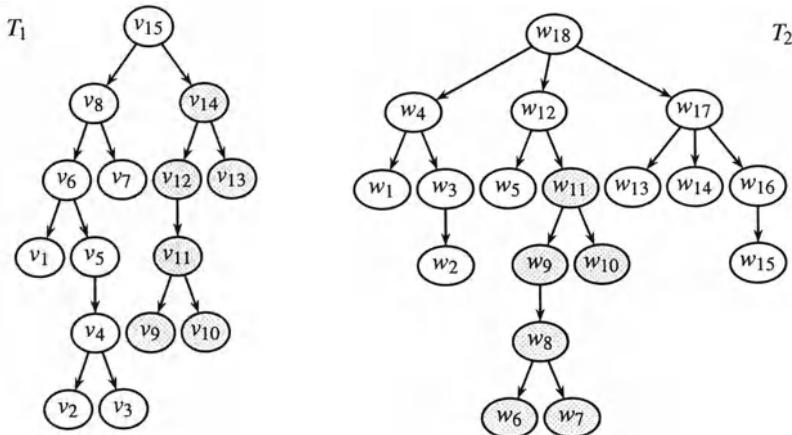


Fig. 4.15. A bottom-up maximum common subtree of two ordered trees. Nodes are numbered according to the order in which they are visited during a postorder traversal. The common subtree of T_1 and T_2 is shown highlighted in both trees.

The problem of finding a bottom-up maximum common subtree of an ordered tree $T_2 = (V_2, E_2)$ to an ordered tree $T_1 = (V_1, E_1)$, can also be reduced to the problem of partitioning $V_1 \cup V_2$ into equivalence classes of bottom-up subtree isomorphism. Two nodes (in the same or in different trees) are equivalent if and only if the bottom-up ordered subtrees rooted at them are isomorphic. Then, the bottom-up subtree of T_1 rooted at node $v \in V_1$ is isomorphic to the bottom-up subtree of T_2 rooted at node $w \in V_2$ if and only if nodes v and w belong to the same equivalence class of bottom-up subtree isomorphism.

Example 4.55. The partition of the ordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ of Fig. 4.15 in equivalence classes of bottom-up subtree isomorphism is illustrated in Fig. 4.16. The twelve equivalence classes are numbered from 1 to 12. The bottom-up subtree of T_1 rooted at node v_{14} is isomorphic to the bottom-up subtree of T_2 rooted at node w_{11} , because nodes v_{14} and w_{11} both belong to the same equivalence class, the one numbered 6.

An Algorithm for Bottom-Up Ordered Maximum Common Subtree Isomorphism

A bottom-up ordered maximum common subtree isomorphism of an ordered tree $T_1 = (V_1, E_1)$ to another ordered tree $T_2 = (V_2, E_2)$ can

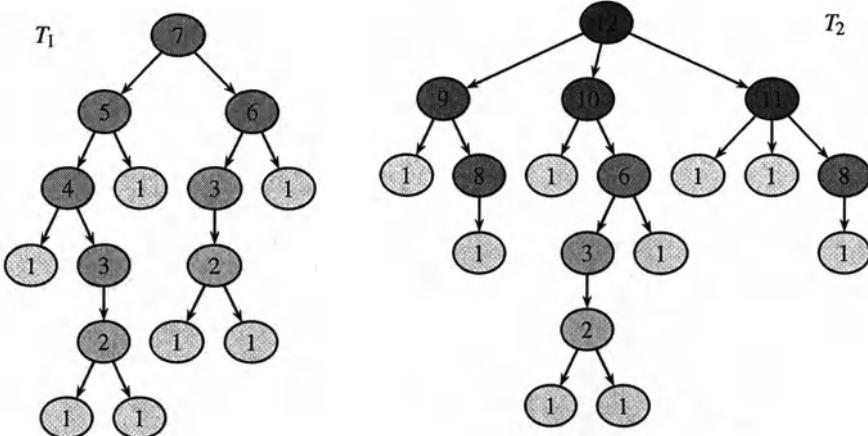


Fig. 4.16. Bottom-up ordered subtree isomorphism equivalence classes for the ordered trees in Fig. 4.15. Nodes are numbered according to the equivalence class to which they belong, and the equivalence classes are shown highlighted in different shades of gray.

be obtained by first partitioning the set of nodes $V_1 \cup V_2$ in bottom-up subtree isomorphism equivalence classes, and then finding equivalent nodes $v \in V_1$ and $w \in V_2$ of largest size.

Notice that finding bottom-up ordered maximum common subtree isomorphisms based on equivalence classes of subtree isomorphism applies to unlabeled trees only. However, the bottom-up maximum common subtree isomorphism procedure for ordered unlabeled trees with n nodes can be extended to ordered trees whose nodes are labeled by integers in the range $1, \dots, n$.

The following algorithm implements the procedure for bottom-up maximum common subtree isomorphism of an ordered tree T_1 to another ordered tree T_2 , collecting a mapping M of nodes of T_1 to nodes of T_2 . The bottom-up subtree isomorphism classes are also collected, as mappings of the nodes of T_1 and T_2 to integers, for the purpose of interactive demonstration of the algorithm.

226 (maximum common subtree isomorphism 209a) +≡
void *bottom_up_ordered_max_common_subtree_isomorphism*(
 const *TREE<string,string>*& T_1 ,
 const *TREE<string,string>*& T_2 ,
node_array<int>& $code1$,
node_array<int>& $code2$,
node_array<node>& M)
{
list<node> L_1 ;

```

postorder_tree_list_traversal(T1,L1);

list<node> L2;
postorder_tree_list_traversal(T2,L2);

int num = 1;
h_array<list<int>,int> CODE;

bool unordered = false;
⟨partition first tree in isomorphism equivalence classes 196a⟩
⟨partition second tree in isomorphism equivalence classes 196b⟩

node v,w;
⟨find largest common subtree 227⟩
map_ordered_subtree(T1,v,T2,w,M);
⟨double-check bottom-up ordered maximum common subtree isomorphism 228⟩
}
}

```

Now, given a partition of the nodes of the ordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ in equivalence classes of bottom-up subtree isomorphism, equivalent nodes $v \in V_1$ and $w \in V_2$ of largest size can be found by a simultaneous traversal of a list of nodes of T_1 and a list of nodes of T_2 , sorted by nonincreasing order of subtree size and, within size, by (nondecreasing) order of equivalence classes of bottom-up subtree isomorphism.

The following procedure uses instead priority queues of nodes of T_1 and T_2 , implementing the simultaneous traversal by selectively deleting nodes of largest subtree size until the nodes with highest priority in both queues belong to the same equivalence class. The required ordering of nodes is accomplished by setting the priority of a node to opposite the subtree size of the node, together with the equivalence class of bottom-up subtree isomorphism to which it belongs.

227

```

⟨find largest common subtree 227⟩ ≡
  node_array<int> size1(T1);
  node_array<int> size2(T2);
  {
    node_pq<two_tuple<int,int> > Q1(T1);
    node_pq<two_tuple<int,int> > Q2(T2);

    forall(v,L1) {
      size1[v] = 1; // leaves have size equal to one
      if (¬T1.is_leaf(v)) {
        forall_children(w,v) {
          size1[v] += size1[w];
        } }
        two_tuple<int,int> prio(−size1[v],code1[v]);
    }
  }
}

```

```

    Q1.insert(v,prio);
}

forall(v,L2) {
    size2[v] = 1; // leaves have size equal to one
    if ( ¬T2.is_leaf(v) ) {
        forall.children(w,v) {
            size2[v] += size2[w];
        }
    }
    two_tuple<int,int> prio(-size2[v],code2[v]);
    Q2.insert(v,prio);
}

while ( ¬Q1.empty() ∧ ¬Q2.empty() ) {
    v = Q1.find_min();
    w = Q2.find_min();
    if ( code1[v] ≡ code2[w] ) break;
    if ( Q1.prio(v) < Q2.prio(w) ) {
        Q1.del_min();
    } else {
        Q2.del_min();
    }
}

M[v] = w;
}

```

The following double-check of maximum common subtree isomorphism, although being redundant, gives some reassurance of the correctness of the implementation. It verifies that M is a bottom-up ordered maximum common subtree isomorphism of T_1 to T_2 , according to Definition 4.53.

```

⟨double-check bottom-up ordered maximum common subtree isomorphism 228⟩ ≡
{ node v,w;
  forall.nodes(v,T1) {
    if ( M[v] ≠ nil ∧
        ¬T1.is_leaf(v) ∧
        ¬T2.is_leaf(M[v]) ∧
        M[T1.first_child(v)] ≠ nil ∧
        M[T1.first_child(v)] ≠ T2.first_child(M[v]) )
      error_handler(1,
                    "Wrong implementation of common subtree isomorphism");
    if ( M[v] ≠ nil ∧
        ¬T1.is_last_child(v) ∧
        ¬T2.is_last_child(M[v]) ∧
        M[T1.next_sibling(v)] ≠ nil ∧
        M[T1.next_sibling(v)] ≠ T2.next_sibling(M[v]) )
      error_handler(1,
                    "Wrong implementation of common subtree isomorphism");
  }
}
```

Further, the common subtree found is verified to be a maximum common subtree of T_1 and T_2 , by checking that no other subtrees of T_1 and T_2 belonging to the same equivalence class of bottom-up ordered subtree isomorphism are larger than the maximum common subtree found.

229a $\langle \text{double-check bottom-up ordered maximum common subtree isomorphism 228} \rangle + \equiv$

```

{ node x,y;
  forall_nodes(x,T1) {
    forall_nodes(y,T2) {
      if ( code1[x] ≡ code2[y] ∧ sizeI[x] > sizeI[y] )
        error_handler(1,
                      "Wrong implementation of common subtree isomorphism");
    } } }
```

Theorem 4.56. *Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be ordered trees with respectively n_1 and n_2 nodes, where $n_1 \leq n_2$. The algorithm for bottom-up ordered maximum common subtree isomorphism runs in $O(n_2 \log n_2)$ time using $O(n_1 + n_2)$ additional space.*

Proof. Let T_1 and T_2 be ordered trees with respectively n_1 and n_2 nodes, where $n_1 \leq n_2$. Despite the expected $O(n_1 + n_2)$ time needed to partition the nodes of T_1 and T_2 in equivalence classes of bottom-up subtree isomorphism, the effort spent on the nodes of T_1 and T_2 is dominated by the n_1 and n_2 insertions and $O(n_2)$ deletions from node priority queues with respectively n_1 and n_2 elements, which are implemented in LEDA by binary heaps and node arrays. Therefore, the algorithm runs in $O(n_2 \log n_2)$ time using $O(n_1 + n_2)$ additional space.

The double-check of bottom-up unordered subtree isomorphism runs in $O(n_1 n_2)$ time using $O(1)$ additional space. \square

The only implementation detail that still needs to be filled in is the definition of a less-than comparison on ordered pairs of integers. The default LEDA less-than comparison operator is redefined by the following procedure to the (default) lexicographic ordering for two-tuples of integers.

229b $\langle \text{maximum common subtree isomorphism 209a} \rangle + \equiv$

```

bool operator<(<
```

```

const two_tuple<int,int>& p,
const two_tuple<int,int>& q)
{
    return compare(p,q) ≡ -1;
}

```

4.3.4 Bottom-Up Unordered Maximum Common Subtree Isomorphism

A bottom-up common subtree of two unordered trees T_1 and T_2 is an unordered tree T such that there are bottom-up unordered subtree isomorphisms of T into T_1 and into T_2 . A maximal bottom-up common subtree of two unordered trees T_1 and T_2 is a bottom-up common subtree of T_1 and T_2 which is not a proper subtree of any other bottom-up common subtree of T_1 and T_2 . A maximum bottom-up common subtree of two unordered trees T_1 and T_2 is a bottom-up common subtree of T_1 and T_2 with the largest number of nodes.

Definition 4.57. A *bottom-up common subtree* of an unordered tree $T_1 = (V_1, E_1)$ to another unordered tree $T_2 = (V_2, E_2)$ is a structure (X_1, X_2, M) , where $X_1 = (W_1, S_1)$ is a bottom-up unordered subtree of T_1 , $X_2 = (W_2, S_2)$ is a bottom-up unordered subtree of T_2 , and $M \subseteq W_1 \times W_2$ is an unordered tree isomorphism of X_1 to X_2 . A bottom-up common subtree (X_1, X_2, M) of T_1 to T_2 is **maximal** if there is no bottom-up common subtree (X'_1, X'_2, M') of T_1 to T_2 such that X_1 is a proper bottom-up subtree of X'_1 and X_2 is a proper bottom-up subtree of X'_2 , and it is **maximum** if there is no bottom-up common subtree (X'_1, X'_2, M') of T_1 to T_2 with $\text{size}[X_1] < \text{size}[X'_1]$.

Example 4.58. For the unordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ in Fig. 4.17, the partial injection $M \subseteq V_1 \times V_2$ given by $M = \{(v_1, w_{10}), (v_2, w_6), (v_3, w_7), (v_4, w_8), (v_5, w_9), (v_6, w_{11}), (v_7, w_5), (v_8, w_{12})\}$ is a bottom-up maximum common subtree isomorphism of T_1 to T_2 .

Again, the problem of finding a bottom-up maximum common subtree of an unordered tree $T_2 = (V_2, E_2)$ on n_2 nodes to an unordered tree $T_1 = (V_1, E_1)$ on n_1 nodes, where $n_1 \leq n_2$, can be reduced to the

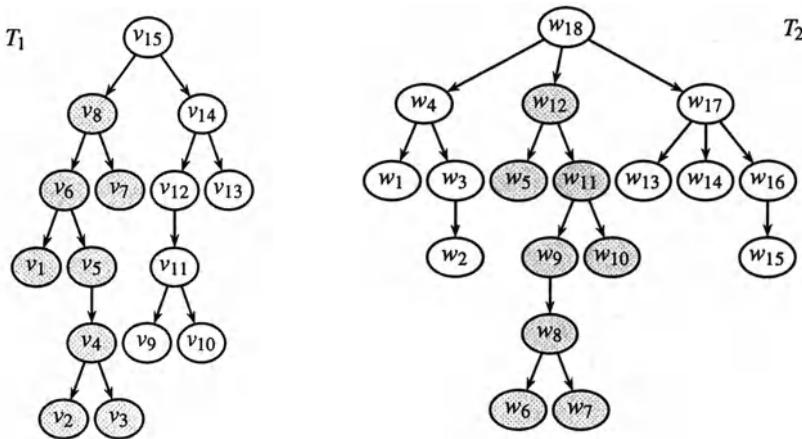


Fig. 4.17. A bottom-up maximum common subtree of two unordered trees. Nodes are numbered according to the order in which they are visited during a postorder traversal. The common subtree of T_1 and T_2 is shown highlighted in both trees.

problem of partitioning $V_1 \cup V_2$ into equivalence classes of bottom-up subtree isomorphism. Two nodes (in the same or in different trees) are equivalent if and only if the bottom-up unordered subtrees rooted at them are isomorphic. Then, the bottom-up subtree of T_1 rooted at node $v \in V_1$ is isomorphic to the bottom-up subtree of T_2 rooted at node $w \in V_2$ if and only if nodes v and w belong to the same equivalence class of bottom-up subtree isomorphism.

Example 4.59. The partition of the unordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ of Fig. 4.17 in equivalence classes of bottom-up subtree isomorphism is illustrated in Fig. 4.18. The ten equivalence classes are numbered from 1 to 10. The bottom-up subtree of T_1 rooted at node v_8 is isomorphic to the bottom-up subtree of T_2 rooted at node w_{12} , because nodes v_8 and w_{12} both belong to the same equivalence class, the one numbered 5.

An Algorithm for Bottom-Up Unordered Maximum Common Subtree Isomorphism

A bottom-up unordered maximum common subtree isomorphism of an unordered tree $T_1 = (V_1, E_1)$ to another unordered tree $T_2 = (V_2, E_2)$ can also be obtained by first partitioning the set of nodes $V_1 \cup V_2$ into

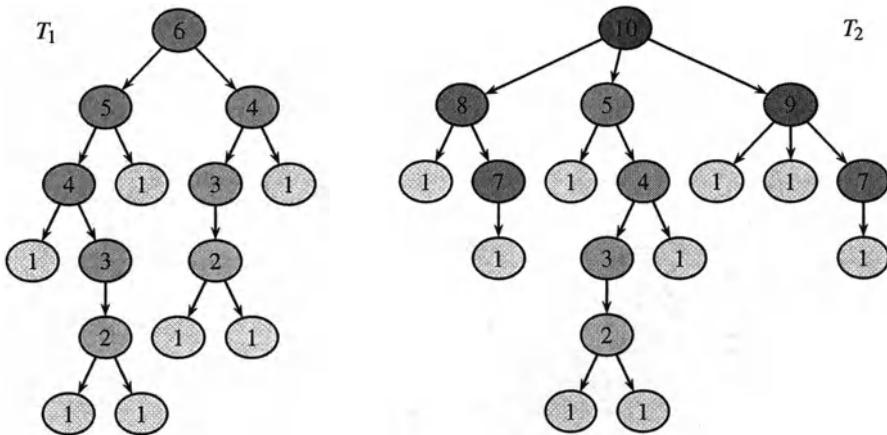


Fig. 4.18. Bottom-up unordered subtree isomorphism equivalence classes for the unordered trees in Fig. 4.17. Nodes are numbered according to the equivalence class to which they belong, and the equivalence classes are shown highlighted in different shades of gray.

bottom-up subtree isomorphism equivalence classes, and then finding equivalent nodes $v \in V_1$ and $w \in V_2$ of largest size.

Again, notice that finding bottom-up unordered maximum common subtree isomorphisms based on equivalence classes of subtree isomorphism applies to unlabeled trees only. However, the bottom-up maximum common subtree isomorphism procedure for unordered unlabeled trees with n nodes can be extended to unordered trees whose nodes are labeled by integers in the range $1, \dots, n$.

The following algorithm implements the procedure for bottom-up maximum common subtree isomorphism of an unordered tree T_1 to another unordered tree T_2 , collecting a mapping M of nodes of T_1 to nodes of T_2 . Again, the bottom-up subtree isomorphism classes are also collected, as mappings of the nodes of T_1 and T_2 to integers, for the purpose of interactive demonstration of the algorithm.

```
<maximum common subtree isomorphism 209a>+≡
void bottom_up_unordered_max_common_subtree_isomorphism(
    const TREE<string,string>& T1,
    const TREE<string,string>& T2,
    node_array<int>& code1,
    node_array<int>& code2,
    node_array<node>& M)
{
    list<node> L1;
    postorder_tree_list_traversal(T1,L1);
```

```

list<node> L2;
postorder_tree_list_traversal(T2,L2);

int num = 1;
h_array<list<int>,int> CODE;

bool unordered = true;
⟨partition first tree in isomorphism equivalence classes 196a)
⟨partition second tree in isomorphism equivalence classes 196b)

node v,w;
⟨find largest common subtree 227)

preorder_subtree_list_traversal(T1,v,L1);
L1.pop(); // node v already mapped
node_array<bool> mapped_to(T2,false);
{ node v,w;
forall(v,L1) {
forall_children(w,M[T1.parent(v)]) {
if ( code1[v] ≡ code2[w] ∧ ¬mapped_to[w] ) {
M[v] = w;
mapped_to[w] = true;
break;
} } } }

⟨double-check bottom-up unordered maximum common subtree isomorphism 234a)

}

```

The following procedure performs an iterative preorder traversal of the subtree of an unordered tree $T = (V, E)$ rooted at node $v \in V$, with the help of a stack of nodes, and builds a list L of the nodes of the subtree of T in the order in which they are visited during the traversal.

233

⟨maximum common subtree isomorphism 209a⟩ + ≡

```

void preorder_subtree_list_traversal(
const TREE<string,string>& T,
const node r,
list<node>& L)
{
  L.clear();
  stack<node> S;
  S.push(r);
  node v,w;
  do {
    v = S.pop();
    L.push(v);
    w = T.last_child(v);
    while ( w ≠ nil ) {
      S.push(w);

```

```

    w = T.previous_sibling(w);
}
} while (  $\neg S.empty()$  );
L.reverse();
}
}
```

The following double-check of maximum common subtree isomorphism, although being redundant, gives some reassurance of the correctness of the implementation. It verifies that M is a bottom-up unordered maximum common subtree isomorphism of T_1 to T_2 , according to Definition 4.57.

```

234a ⟨double-check bottom-up unordered maximum common subtree isomorphism 234a⟩≡
    { node v,w,z;
      bool found;
      forall_nodes(v,T1) {
        if ( M[v] ≠ nil ) {
          forall_nodes(w,T1)
            if ( v ≠ w ∧ M[v] ≡ M[w] )
              error_handler(1,
                "Wrong implementation of common subtree isomorphism");
          forall_nodes(w,T2) {
            if ( ¬T1.is_root(v) ∧
                ¬T2.is_root(w) ∧
                M[T1.parent(v)] ≡ T2.parent(w) ) {
              found = false;
              forall_children(z,T2.parent(w))
                if ( M[v] ≡ z ) found = true;
              if ( ¬found ) {
                error_handler(1,
                  "Wrong implementation of common subtree isomorphism");
            } } } } } }

```

Further, the common subtree found is verified to be a maximum common subtree of T_1 and T_2 , by checking that no other subtrees of T_1 and T_2 belonging to the same equivalence class of bottom-up unordered subtree isomorphism are larger than the maximum common subtree found.

```

234b {double-check bottom-up unordered maximum common subtree isomorphism 234a} +≡
      { node x,y;
        forall_nodes(x,T1) {
          forall_nodes(y,T2) {
            if( code1[x] ≡ code2[y] ∧ size1[x] > size1[v] )
              error_handler(1,
                "Wrong implementation of common subtree isomorphism");
          }
        }
      }

```

Theorem 4.60. *Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be unordered trees with respectively n_1 and n_2 nodes. The algorithm for bottom-up unordered maximum common subtree isomorphism runs in $O((n_1 + n_2)^2)$ time using $O(n_1 + n_2)$ additional space.*

Proof. Let T_1 and T_2 be unordered trees with respectively n_1 and n_2 nodes, and let $k(v)$ denote the number of children of node v . The effort spent on a leaf node of T_1 or T_2 is $O(1)$, and the total effort spent on the leaves of T_1 and T_2 is bounded by $O(n_1 + n_2)$.

The effort spent on a nonleaf v of T_1 or T_2 , on the other hand, is dominated by bucket sorting a list of $k(v)$ integers, despite the expected $O(1)$ time taken to look up or insert the sorted list of integers in the dictionary. Since all these integers fall in the range $1, \dots, n_1 + n_2$, the time taken for bucket sorting the children codes is bounded by $O(k(v) + n_1 + n_2)$. The total effort spent on nonleaves of T_1 and T_2 is thus bounded by $O(\sum_{v \in V_1 \cup V_2} (k(v) + n_1 + n_2)) = O(\sum_{v \in V_1 \cup V_2} k(v)) + O((n_1 + n_2)^2) = O(n_1 + n_2) + O((n_1 + n_2)^2) = O((n_1 + n_2)^2)$. Therefore, the algorithm runs in $O((n_1 + n_2)^2)$ time using $O(n_1 + n_2)$ additional space.

The double-check of bottom-up unordered subtree isomorphism runs in $O(n_1 n_2)$ time using $O(1)$ additional space. \square

Lemma 4.61. *Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be unordered trees of degree bounded by a constant k and with respectively n_1 and n_2 nodes. The algorithm for bottom-up unordered subtree isomorphism can be implemented to run in expected $O(n_1 + n_2)$ time using $O(n_1 + n_2)$ additional space.*

Proof. For trees of degree bounded by a constant k , bucket sorting a list of at most k integers can be replaced by appropriate code performing at most k comparisons, and thus taking $O(1)$ time to sort a list of at most k integers, as shown for $k = 3$ in the previous implementation of the algorithm. In this case, the effort spent on a nonleaf v of T_1 or T_2 is dominated instead by the expected $O(1)$ time required to look up or insert the sorted list of k integers in the dictionary, and the algorithm runs in expected $O(n_1 + n_2)$ time, still using $O(n_1 + n_2)$ additional space. \square

4.3.5 Interactive Demonstration of Maximum Common Subtree Isomorphism

The algorithms for top-down and bottom-up maximum common subtree isomorphism upon ordered and unordered trees, are all integrated next in the interactive demonstration of graph algorithms. A simple checker for top-down ordered maximum common isomorphic subtrees that provides some visual reassurance consists of highlighting the nodes and edges of the subtrees of T_1 and T_2 that are isomorphic to each other.

236a ⟨demo maximum common subtree isomorphism 236a⟩≡

```

void gw_top_down_ordered_maximum_common_subtree_isomorphism(
    GraphWin& gw1)
{
    GRAPH<string,string>& G1 = gw1.get_graph();
    TREE<string,string> T1(G1);

    GRAPH<string,string> G2;
    GraphWin gw2(G2,500,500,
        "Top-Down Ordered Maximum Common Subtree Isomorphism");
    gw2.display();
    gw2.message("Enter second tree. Press done when finished");
    gw2.edit();
    gw2.del_message();

    TREE<string,string> T2(G2);

    node_array<node> M(T1,nil);
    top_down_ordered_max_common_subtree_isomorphism(T1,T2,M);

    gw1.save_all_attributes();
    ⟨show maximum common subtree isomorphism 236b⟩
    gw2.wait();
    gw1.restore_all_attributes();
}

```

The nodes and edges of the isomorphic top-down maximum common subtrees of T_1 and T_2 are highlighted next.

236b ⟨show maximum common subtree isomorphism 236b⟩≡

```

{ node v;
  forall_nodes(v,T1) {
    if (M[v] ≠ nil) {
      gw1.set_color(v,blue);
      gw2.set_color(M[v],blue);
    }
  }
  edge e;
  forall_edges(e,T1) {

```

```

if ( gw1.get_color(T1.source(e)) ≡ blue ∧
    gw1.get_color(T1.target(e)) ≡ blue ) {
  gw1.set_color(e,blue);
  gw1.set_width(e,2);
}
forall_edges(e,T2) {
  if ( gw2.get_color(T2.source(e)) ≡ blue ∧
    gw2.get_color(T2.target(e)) ≡ blue ) {
    gw2.set_color(e,blue);
    gw2.set_width(e,2);
}
}

```

As in the case of top-down ordered maximum common subtree isomorphism, a simple checker for top-down unordered maximum common isomorphic subtrees that provides some visual reassurance consists of highlighting the nodes and edges of the subtrees of T_1 and T_2 that are isomorphic to each other.

237

```

⟨demo maximum common subtree isomorphism 236a⟩+≡
void gw_top_down_unordered_maximum_common_subtree_isomorphism(
  GraphWin& gw1)
{
  GRAPH<string,string>& G1 = gw1.get_graph();
  TREE<string,string> T1(G1);

  GRAPH<string,string> G2;
  GraphWin gw2(G2,500,500,
    "Top-Down Unordered Maximum Common Subtree Isomorphism");
  gw2.display();
  gw2.message("Enter second tree. Press done when finished");
  gw2.edit();
  gw2.del_message();

  TREE<string,string> T2(G2);

  node_array<node> M(T1,nil);
  top_down_unordered_max_common_subtree_isomorphism(T1,T2,M);

  gw1.save_all_attributes();
  ⟨show maximum common subtree isomorphism 236b⟩
  gw2.wait();
  gw1.restore_all_attributes();
}

```

A simple checker for bottom-up ordered maximum common isomorphic subtrees that provides some visual reassurance consists of numbering the nodes according to the equivalence class of bottom-up ordered subtree isomorphism to which they belong, and also high-

lighting the nodes and edges of the subtrees of T_1 and T_2 that are isomorphic to each other.

238a

```
<demo maximum common subtree isomorphism 236a>+≡
void gw_bottom_up_ordered_maximum_common_subtree_isomorphism(
    GraphWin& gw1)
{
    GRAPH<string,string>& G1 = gw1.get_graph();
    TREE<string,string> T1(G1);

    GRAPH<string,string> G2;
    GraphWin gw2(G2,500,500,
        "Bottom-Up Ordered Maximum Common Subtree Isomorphism");
    gw2.display();
    gw2.message("Enter second tree. Press done when finished");
    gw2.edit();
    gw2.del_message();

    TREE<string,string> T2(G2);

    node_array<int> C1(T1);
    node_array<int> C2(T2);
    node_array<node> M(T1,nil);
    bottom_up_ordered_max_common_subtree_isomorphism(T1,T2,C1,C2,M);

    gw1.save_all_attributes();
    node v,w;
    forall_nodes(v,T1)
        gw1.set_label(v,string("%i",C1[v]));
    forall_nodes(w,T2)
        gw2.set_label(w,string("%i",C2[w]));
    <show maximum common subtree isomorphism 236b>
    gw2.wait();
    gw1.restore_all_attributes();
}
```

Again, a simple checker for bottom-up unordered maximum common isomorphic subtrees that provides some visual reassurance consists of numbering the nodes according to the equivalence class of bottom-up unordered subtree isomorphism to which they belong, and also highlighting the nodes and edges of the subtrees of T_1 and T_2 that are isomorphic to each other.

238b

```
<demo maximum common subtree isomorphism 236a>+≡
void gw_bottom_up_unordered_maximum_common_subtree_isomorphism(
    GraphWin& gw1)
{
    GRAPH<string,string>& G1 = gw1.get_graph();
    TREE<string,string> T1(G1);

    GRAPH<string,string> G2;
```

```

GraphWin gw2(G2,500,500,
    "Bottom-Up Unordered Maximum Common Subtree Isomorphism");
gw2.display();
gw2.message("Enter second tree. Press done when finished");
gw2.edit();
gw2.del_message();

TREE<string,string> T2(G2);

node_array<int> C1(T1);
node_array<int> C2(T2);
node_array<node> M(T1,nil);
bottom_up_unordered_max_common_subtree_isomorphism(T1,T2,C1,C2,M);

gw1.save_all_attributes();
node v,w;
forall_nodes(v,T1)
    gw1.set_label(v,string("%i",C1[v]));
forall_nodes(w,T2)
    gw2.set_label(w,string("%i",C2[w]));
⟨show maximum common subtree isomorphism 236b⟩
gw2.wait();
gw1.restore_all_attributes();
}

```

4.4 Applications

Isomorphism problems on trees find application whenever structures described by trees need to be identified or compared. In most application areas, tree isomorphism, subtree isomorphism, and maximal common subtree isomorphism can be seen as some form of pattern matching or information retrieval. Such an application will be further addressed in Chap. 7, within the broader context of isomorphism problems on graphs.

The application of the different isomorphism problems on trees to the comparison of RNA secondary structure in computational molecular biology is discussed next. See the bibliographic notes below for further applications of tree isomorphism and related problems.

An RNA (ribonucleic acid) molecule is composed of four types of nucleotides, also called *bases*. These are: adenine (A), cytosine (C), guanine (G), and uracil (U). Nucleotides A and U, as well as nucleotides C and G, form stable chemical bonds with each other and are thus called *complementary nucleotides*.

The structure of an RNA molecule can be described at different levels of abstraction. The *primary structure*, on the one hand, is the linear sequence of nucleotides in an RNA molecule. An RNA sequence is thus a finite sequence or string over the alphabet $\{A, C, G, U\}$. The *secondary structure*, on the other hand, is a simplified two-dimensional description of the three-dimensional structure of the RNA molecule in which some complementary nucleotides get paired, making the molecule fold in three-dimensional space. The secondary structure of an RNA molecule is thus the collection of all nucleotide pairs that occur in its three-dimensional structure. Almost all RNA molecules form secondary structure.

Now, the secondary structure of an RNA molecule can be represented by an undirected, connected, labeled graph, with one vertex for each nucleotide in the RNA sequence and where consecutive nucleotides in the sequence, as well as paired nucleotides, are joined by an edge. Vertices are labeled by the corresponding nucleotide, while edges are not labeled.

Example 4.62. Consider the following RNA sequence of 120 nucleotides from an Annelida species,

```
GUCUACGGCC AUACCACGUU GAAAGCACCG GUUCUCGUCC
GAUCACCGAA GUUAAGCAAC GUCGGGCCCG GUUAGUACUU
GGAUGGGUGA CCGCCUGGGGA AUACCGGGUG CUGUAGACUU
```

and the following RNA sequence, also of 120 nucleotides, from a Hemichordata species:

```
GCCUACGGCC AUACCACGUA GAAUGCACCG GUUCUCGUCC
GAUCACCGAA GUUAAGCUGC GUCGGGCGUG GUUAGUACUU
GCAUGGGAGA CCGGCUGGGGA AUACCACGUG CCGUAGGCUU
```

Both sequences correspond to the same gene, the 5S rRNA. Their graphs of secondary structure are shown in Fig. 4.19.

The graph of an RNA secondary structure is very tree-like. As a matter of fact, it can be uniquely decomposed into structural elements called stacks, loops, and external nodes. Stacks are formed by paired nucleotides which are consecutive in the RNA sequence. Loops may differ in size (number of unpaired nucleotides) and branching degree

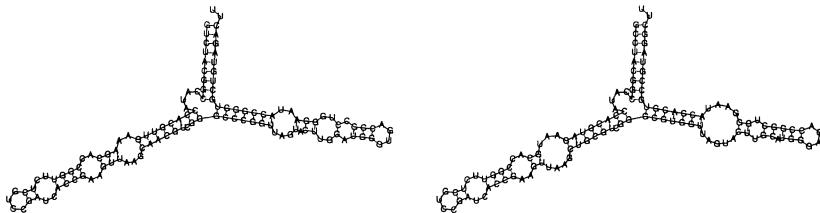


Fig. 4.19. Usual representation of the predicted secondary structure of RNA sequences from an Annelida species (left) and a Hemichordata species (right). The sequences are composed of 120 nucleotides each.

(hairpin loops have degree one, and internal loops have degree two or more). External nodes are nucleotides that belong neither to a stack nor to a loop.

Now, the graph of an RNA secondary structure can also be represented by an ordered labeled tree, with nonleaves corresponding to nucleotide pairs and leaves corresponding to unpaired nucleotides. Since paired nucleotides are contracted to single nodes, stacks appear as chains of nonleaves, possibly ending at a leaf, while loops appear as bushes of leaves. An additional root node is added, as parent of the external nodes. Node labels are used in order to distinguish between leaves representing paired and unpaired nucleotides. In schematic diagrams, the label of a node standing for paired nucleotides is often represented by a black circle, while the label of a leaf standing for an unpaired nucleotide is represented by white circles. The tree representation of the predicted RNA secondary structures of Fig. 4.19 is shown in Fig. 4.20, where nodes standing for paired nucleotides are labeled by white squares instead of black circles.

Tree isomorphism, subtree isomorphism and, in general, maximum common subtree isomorphism find application in the comparison of RNA secondary structures. Maximum common subtrees of the RNA secondary structures of Annelida and Hemichordata of Fig. 4.20 is shown in Fig. 4.21. While the trees representing these RNA secondary

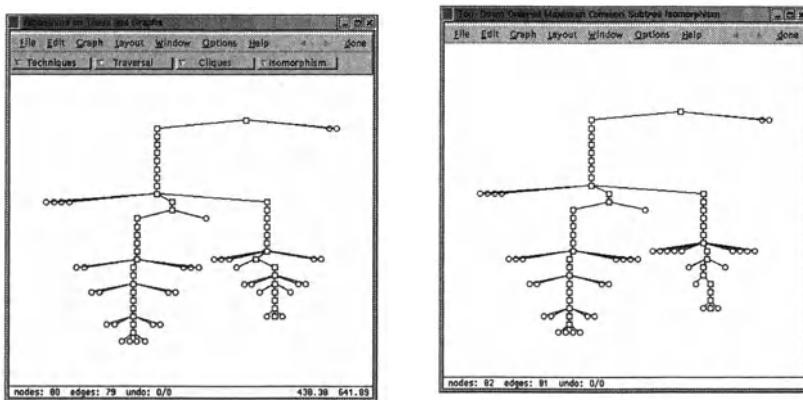


Fig. 4.20. Tree representation of the predicted RNA secondary structures of Annelida (left) and Hemichordata (right) of Fig. 4.19.

structures have respectively 80 and 82 nodes, a top-down ordered maximum common subtree has 37 nodes, a top-down unordered maximum common subtree has 74 nodes, and a bottom-up ordered maximum common subtree which, in this case, is also a bottom-up unordered maximum common subtree, has only 21 nodes.

Summary

Several isomorphism problems on ordered and unordered trees were addressed in this chapter. They form a hierarchy of pattern matching problems, where maximum common subtree isomorphism generalizes subtree isomorphism, which in turn generalizes tree isomorphism. Simple algorithms are given in detail for enumerating all solutions to these problems. Some of the algorithms for tree and subtree isomorphism are based on the methods of tree traversal discussed in Chap. 3, while some of the maximum common subtree algorithms are based on the divide-and-conquer technique reviewed in Sect. 2.4. References to more sophisticated algorithms are given in the bibliographic notes below. Computational molecular biology is also discussed as a prototypical application of isomorphism problems on trees.

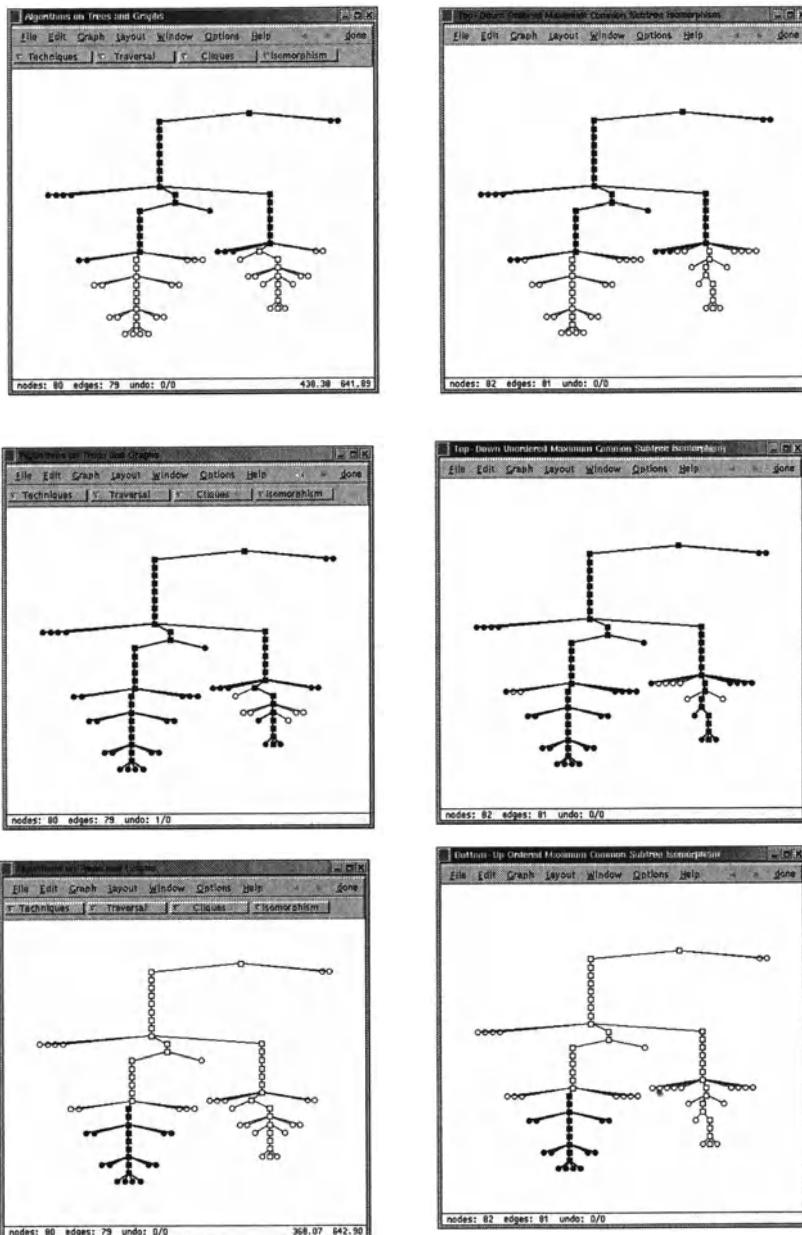


Fig. 4.21. A top-down ordered maximum common subtree isomorphism (top row), a top-down unordered maximum common subtree isomorphism (middle row), and a bottom-up ordered maximum common subtree isomorphism of the RNA secondary structures of Annelida (left) and Hemichordata (right) of Fig. 4.20.

Bibliographic Notes

The algorithm for unordered tree isomorphism is based on [63, pp. 196–199]. Another, more efficient algorithm running in $O(n)$ time was proposed in [2, pp. 84–86]. Both algorithms for unordered tree isomorphism were actually found by J. Edmonds. See also Exercise 4.4.

Tree isomorphism is closely related to the problem of generating trees without repetitions. Algorithms for enumerating all nonisomorphic rooted ordered trees on n nodes were given in [106, 378], and for enumerating all nonisomorphic unordered trees on n nodes, either rooted or free, in [41, 214] and also in [367, Ch. 5]. See also Problem 4.2 and Exercise 4.1. Further, a certificate for free unordered trees was given in [203, Sect. 7.3.1]. See also Problem 4.3 and Exercise 4.2. Notice that enumerating nonisomorphic trees is different from enumerating *labeled* trees, that is, trees $T = (V, E)$ whose n nodes are labeled by a bijection $V \rightarrow \{1, 2, \dots, n\}$. Algorithms for enumerating unordered labeled trees are often based on Prüfer sequences. See, for instance, [91] and [203, Sect. 3.3].

An algorithm for subtree isomorphism of binary trees was proposed in [40, Sect. 5d]. See also Exercise 4.5. The algorithm for top-down unordered subtree isomorphism is based on [224, 273, 348]. A more efficient algorithm was proposed in [79] which runs in $O(n_1\sqrt{n_1}n_2)$ time, and it was further improved in [296] to run in $O(n_1\sqrt{n_1}n_2/\log n_1)$ time. The algorithm for top-down unordered maximum common subtree isomorphism is also based on [224].

The simple algorithm for bottom-up ordered subtree isomorphism is based on [345]. Further algorithms were proposed in [100, 140, 141, 221]. The algorithm for bottom-up unordered subtree isomorphism is based on [333], where the idea from [113] is exploited that a procedure for dynamically maintaining a global table of unique identifiers allows the compacted directed acyclic graph representation of a binary tree to be determined in expected $O(n)$ time. Further algorithms were proposed in [95, 142]. The algorithms for bottom-up ordered and unordered maximum common subtree isomorphism are also based on [333].

Isomorphism problems on trees are also closely related to the problem of transforming or *editing* trees. The edit distance between two

labeled trees is the cost of a least-cost sequence of elementary edit operations, such as deletion and insertion of bottom-up subtrees (and, in particular, leaves) and modification of node or edge labels, that allows one to transform a given tree into another given tree. Several such tree edit distances have been proposed [167, 169, 293, 313, 316, 333, 377]. Further, a hierarchy among several of these tree edit distances was given in [355].

The elementary edit operations often have a cost or weight associated to them, and under the assumption that the cost of modifying a node label is always less than the cost of deleting a node with the old label and inserting a node with the new label, the edit distance can be shown to correspond with a maximal common subtree, as shown in [59] in the more general context of elementary edit operations on graphs. Several algorithms were proposed for computing the edit distance between ordered trees [218, 301, 313, 382]. Further measures of edit distance were proposed for ordered trees, either based on the elementary edit operation of edge rotation [86, 379] or node splitting and merging [219].

Computing the edit distance between unordered trees is an NP-complete problem [383], even for trees of bounded degree $k \geq 2$, although under the constraint that deletion and insertion operations be made on leaves only [293] or that disjoint subtrees be mapped to disjoint subtrees [381], the distance can be computed in polynomial time. Some restricted edit distances between unrooted and unordered trees can also be computed in polynomial time [315]. Further algorithms were proposed for computing the edit distance between rooted unordered trees [300] and free unordered trees [384].

Tree isomorphism and related problems being a form of pattern matching, they should not be confused with the related problem of pattern matching in trees, where leaves in the smaller tree are labeled with variables (wildcards) and the pattern matching problem consists of finding all subtrees of the larger tree that are isomorphic to some extension of the smaller tree, which is obtained by replacing these variables by appropriate subtrees of the larger tree. Several algorithms were proposed for pattern matching in ordered trees [81, 99, 163, 201].

The RNA sequences given in Example 4.62 for the 5S rRNA gene are taken from [306, 307] for the Annelida species, and from [251] for the Hemichordata species. The usual representation of their predicted secondary structure shown in Fig. 4.19 was produced using the RNAfold program [161], which is based on the dynamic programming algorithm of [385] for computing minimum free energy secondary structures. See [13, 359] for an introduction to computational molecular biology, see [162, 360] for further details about RNA secondary structures and their mathematical properties, and [284] for an introduction to sequence comparison algorithms. See also [145] for dynamic programming algorithms on DNA and RNA sequences.

Beside the comparison of RNA secondary structures based on maximum common subtree isomorphism, discussed in this chapter, tree edit distance was used for comparing RNA secondary structures in [297].

Another important application area of tree isomorphism and related problems is the comparison and retrieval of structured documents. Documents often display a structure, and the structural knowledge expressed in text documents marked up with languages such as SGML and XML can be exploited for more effective information retrieval, and detecting similarities and differences between structured documents is the basis of structured information retrieval. Tree editing was used in [377] to highlight differences between versions of the same computer program, and in [70, 71, 72] to find differences between structured documents.

Review Problems

4.1 Determine whether the unordered trees T_1 and T_2 of Fig. 4.22 are isomorphic. Give also the isomorphism code for the root of each of them, as well as their canonical ordered trees.

4.2 Let $[v_1, v_2, \dots, v_n]$ be the sequence of the n nodes of a tree T in the order in which they are visited during a preorder traversal. The *depth sequence* of T is the sequence of n integers $[\text{depth}[v_1], \text{depth}[v_2], \dots, \text{depth}[v_n]]$; for instance, the depth sequences of the 14 nonisomorphic ordered trees on five nodes are shown in Fig. 4.23. Prove that the depth

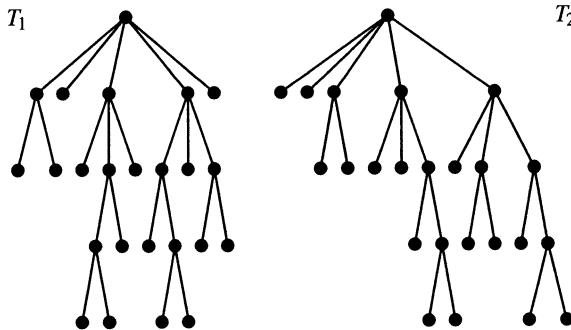


Fig. 4.22. Unordered trees for Problem 4.1.

sequence of an ordered tree is a certificate for ordered tree isomorphism.

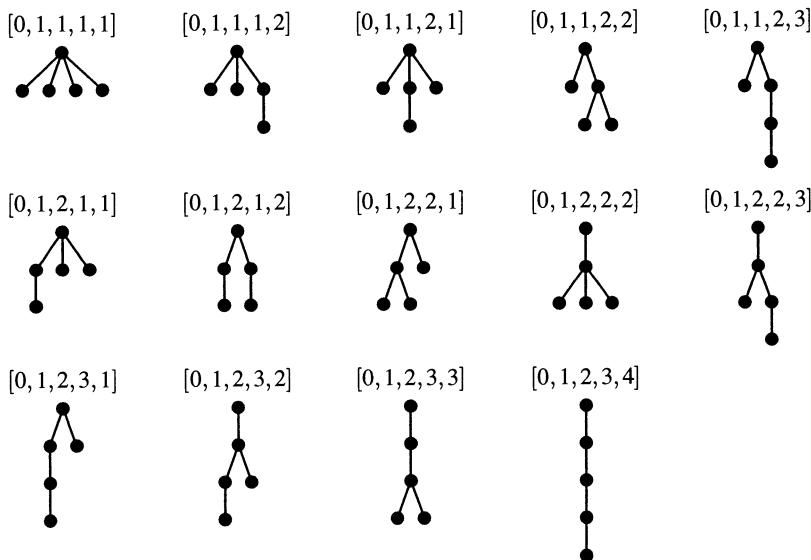


Fig. 4.23. Depth sequences of the 14 nonisomorphic ordered trees on five nodes, arranged in lexicographic order.

4.3 Let the parenthesis string of an unordered tree on n nodes be the string of zeros and ones of length $2n$ defined as follows. The parenthesis string of a leaf node is 01, and the parenthesis string associated

to a nonleaf node is obtained by concatenating the set of strings of the children of the node, sorted in lexicographic order, preceded by an additional 0 and followed by an additional 1. For instance, the parenthesis string of the nine nonisomorphic unordered trees on five nodes are shown in Fig. 4.24. Prove that the parenthesis string of the root of an unordered tree is a certificate for unordered tree isomorphism.

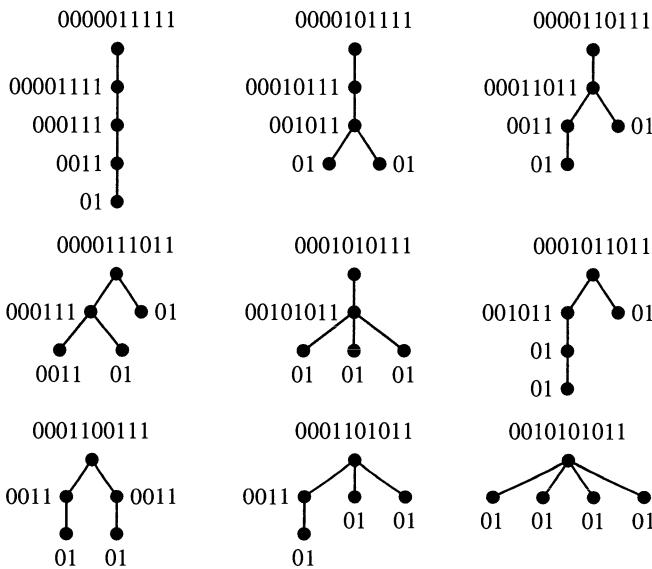


Fig. 4.24. Parenthesis strings of the nine nonisomorphic unordered trees on five nodes, arranged in lexicographic order. The parenthesis string of each node is also shown next to the node.

4.4 Give upper and lower bounds on the number of subtree isomorphisms of an ordered tree on n_1 nodes into an ordered tree on n_2 nodes, where $n_1 \leq n_2$.

4.5 Find top-down and bottom-up maximum common subtree isomorphisms of the ordered trees T_1 and T_2 of Fig. 4.25. Find also top-down and bottom-up maximum common subtree isomorphisms of their underlying unordered trees.

4.6 A distance measure between trees, also called a *metric* over trees, is a real function δ satisfying the following conditions,

- $\delta(T_i, T_j) \geq 0$,

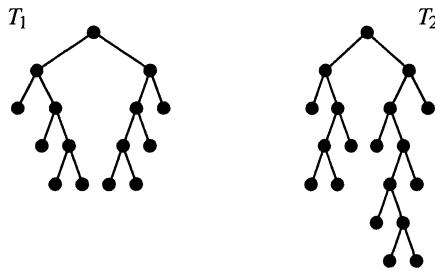


Fig. 4.25. Ordered trees for Problem 4.5.

- $\delta(T_i, T_j) = 0$ if and only if T_i and T_j are isomorphic,
- $\delta(T_i, T_j) = \delta(T_j, T_i)$, and
- $\delta(T_i, T_j) \leq \delta(T_i, T_k) + \delta(T_k, T_j)$

for all trees T_i, T_j, T_k . Define tree distance measures between ordered and unordered trees, based on top-down and bottom-up maximum common subtree isomorphism. Prove them to be metric.

Exercises

4.1 Give an algorithm for ordered tree isomorphism based on the depth sequences discussed in Problem 4.2. Give also the time and space complexity of the algorithm.

4.2 Give an algorithm for unordered tree isomorphism based on the parenthesis strings discussed in Problem 4.3. Give also the time and space complexity of the algorithm.

4.3 Extend the algorithm for unordered tree isomorphism, in order to test isomorphism of labeled trees. Assume that node labels are integers in the range $1, \dots, n$, where n is the number of nodes in each of the trees. Give also a correctness proof, together with the time and space complexity of the extended algorithm.

4.4 Consider the following procedure for testing isomorphism of unordered trees. Given two unordered trees, label first all the leaves in the two trees with the integer zero. If the number of leaves in the two trees differ, then the trees are not isomorphic.

Otherwise, determine for each tree the set of unlabeled nodes having all of their children already labeled. Tentatively, label each such node with the sequence of labels of its children, sorted in nondecreasing order. If the multisets of tentative labels for these nodes in the two trees differ, then the trees are not isomorphic. Otherwise, substitute the tentative labels by *new* integer labels in a consistent way, that is, such that nodes with the same tentative label get the same new label. Repeat the procedure, until all nodes have been labeled. In such a case, the trees are isomorphic.

Give an algorithm for unordered tree isomorphism implementing the previous procedure. Give also the time and space complexity of the algorithm.

4.5 All top-down subtree isomorphisms of an ordered tree $T_1 = (V_1, E_1)$ on n_1 nodes into an ordered tree $T_2 = (V_2, E_2)$ on n_2 nodes, where $n_1 \leq n_2$, can be obtained by repeated application of the top-down ordered subtree isomorphism algorithm, upon all bottom-up subtrees of T_2 of size and height at least as large as those of T_1 . Consider, though, an alternative procedure based on the following remark.

For all nodes $v \in V_1$, let $Q(v) \subseteq V_2$ denote the set of all nodes $w \in V_2$ such that a path in T_2 originating at node w exactly replicates the path in T_1 from the root down to node v . Then, if some node $w \in V_2$ belongs to $Q(v)$ for *all the leaves* $v \in V_1$, there are paths in T_2 originating at node w that constitute a complete replica of T_1 , that is, node w is the root of a top-down subtree of T_2 isomorphic to T_1 . Therefore, the set of all nodes $w \in V_2$ that are roots of top-down subtrees of T_2 isomorphic to T_1 is just the intersection $\cap_{v \in V'_1} Q(v)$, where $V'_1 \subseteq V_1$ is the set of the leaves of T_1 .

Give an algorithm for finding all top-down ordered subtree isomorphisms of an ordered tree T_1 on n_1 nodes into an ordered tree T_2 on n_2 nodes, with $n_1 \leq n_2$, based on the previous remark. Give also the time and space complexity of the algorithm.

4.6 Give algorithms for bottom-up subtree isomorphism on both ordered and unordered trees, based on the top-down subtree isomorphism algorithms. Give also a correctness proof, together with the time and space complexity of the algorithms.

- 4.7** Extend the algorithm for bottom-up unordered subtree isomorphism, in order to test subtree isomorphism of labeled trees. Assume that node labels are integers in the range $1, \dots, n$, where n is the number of nodes in each of the trees. Give also a correctness proof, together with the time and space complexity of the extended algorithm.
- 4.8** Improve the algorithm for bottom-up ordered maximum common subtree isomorphism, replacing the use of node priority queues by bucket sorting. Give also the time and space complexity of the improved algorithm.
- 4.9** Give an efficient algorithm for computing tight upper and lower bounds on the size of a top-down maximum common subtree of two unordered trees. Assume the trees are unlabeled. Give also a correctness proof, together with the time and space complexity of the algorithm.
- 4.10** Extend to labeled trees the algorithms for bottom-up ordered and unordered maximum common subtree isomorphism. Assume that node labels are integers in the range $1, \dots, n$, where n is the total number of nodes in the trees. Give also a correctness proof, together with the time and space complexity of the extended algorithm.

5. Graph Traversal

A good idea has a way of becoming simpler and solving problems other than that for which it was intended.

—Robert E. Tarjan [299]

Most algorithms on graphs require a systematic method of visiting the vertices of a graph. Two basic and common methods of exploring a graph are the generalization to graphs of the preorder traversal and the top-down traversal of trees.

- In a *depth-first* traversal of a graph, also known as *depth-first search*, the vertices reachable from a given, initial vertex are visited before their adjacent vertices, and those vertices which a visited vertex is adjacent to are visited in first-to-last order.
- In a *breadth-first* traversal of a graph, also known as *breadth-first search*, the vertices reachable from a given, initial vertex are visited in order of nondecreasing distance from the initial vertex.

These systematic methods of visiting the vertices of a graph are the subject of this chapter.

5.1 Depth-First Traversal of a Graph

A traversal of a graph $G = (V, E)$ with n vertices is just a bijection $\text{order} : V \rightarrow \{1, \dots, n\}$. In an operational view, a traversal of a graph consists of visiting first the vertex v with $\text{order}[v] = 1$, then the vertex w with $\text{order}[w] = 2$, and so on, until visiting last the vertex z with $\text{order}[z] = n$.

The generalization of tree traversal to graph traversal does not come for free, though. Graph traversal algorithms need to take into account the possibility that a vertex being visited had already been visited along a different path during the traversal, while in (rooted) tree traversal, a node can only be accessed along the unique path from the root to the node. Nevertheless, there is a close relationship between tree and graph traversal, which will be explored in detail in this chapter.

As with tree traversal, the order in which the arcs going out of a given vertex are considered (and, therefore, the order in which the vertices adjacent to a given vertex—the children of a node, in the case of trees—are considered) is significant for an ordered graph, and it is also fixed by the representation adopted for both unordered and ordered graphs. Further, the order in which the vertices of the graph are considered, as well as the order in which the arcs coming into and going out of a vertex are considered, are also significant.

The following assumption about the relative order of the vertices and arcs of a graph will, without loss of generality, make it possible to give a precise notion of a depth-first and breadth-first spanning forest of a graph.

Assumption 5.1. *The order of the vertices of a graph is the order fixed by the representation of the graph, the order of the arcs coming into a vertex is the order of their source vertices, and the order of the arcs going out of a vertex is the order of their target vertices.*

The actual LEDA representation of a graph to be traversed will be later rearranged in order to meet the previous assumption about the relative order of vertices and arcs.

The counterpart to the preorder traversal of a tree is the depth-first traversal of a graph, which is the preorder traversal of a depth-first spanning forest of a graph. Recall that a spanning forest of a graph is an ordered set of pairwise-disjoint subgraphs of the graph which are rooted trees and which, together, span all the vertices of the graph.

Definition 5.2. *Let $G = (V, E)$ be a graph with n vertices. A spanning forest F of G is a **depth-first spanning forest** of G if the following conditions are satisfied,*

- for all trees $T = (W, S)$ in F and for all arcs $(v, w) \in S$, there is no arc $(x, y) \in E \cap (W \times W)$ such that $\text{order}[v] < \text{order}[x] < \text{order}[w] \leq \text{order}[y]$
- for all trees $T_i = (W_i, S_i)$ and $T_j = (W_j, S_j)$ in F with $i < j$, there is no arc $(v, w) \in E$ with $v \in W_i$ and $w \in W_j$

where $\text{order} : W \rightarrow \{1, \dots, k\}$ is the preorder traversal of a rooted tree $T = (W, S)$ with $k \leq n$ nodes.

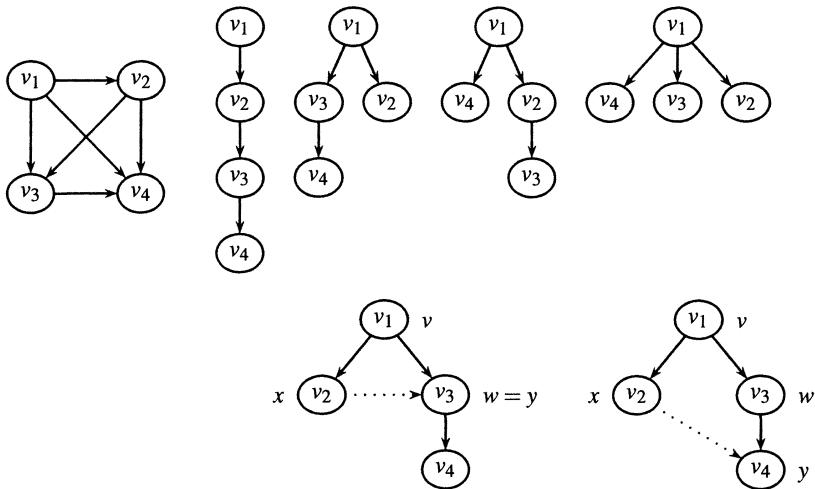


Fig. 5.1. Depth-first spanning forest of a graph. All the depth-first spanning trees rooted at vertex v_1 of the graph to the left of the top row are also shown in the top row, to the right. The spanning tree shown twice in the bottom row is not a depth-first spanning tree.

Example 5.3. The notion of a depth-first spanning forest of a graph is illustrated in Fig. 5.1. Since all vertices of the graph are reachable from vertex v_1 , every depth-first spanning forest of the graph whose initial vertex is v_1 must consist of a single tree, which is therefore a spanning tree of the graph. The spanning tree shown twice in the bottom row of Fig. 5.1 is not a depth-first spanning tree of the graph, because there is an arc (v, w) in the tree and an arc (x, y) not in the tree with $\text{order}[v] < \text{order}[x] < \text{order}[w] \leq \text{order}[y]$. In the picture to the left, $\text{order}[v] = 1$, $\text{order}[x] = 2$, and $\text{order}[w] = \text{order}[y] = 3$, while

in the picture to the right, $\text{order}[v] = 1$, $\text{order}[x] = 2$, $\text{order}[w] = 3$, and $\text{order}[y] = 4$.

Now, a depth-first traversal of a graph is just a preorder traversal of each of the trees in the depth-first spanning forest of the graph.

Definition 5.4. Let $G = (V, E)$ be a graph with n vertices, and let F be the depth-first spanning tree of G . A bijection $\text{order} : V \rightarrow \{1, \dots, n\}$ is a **depth-first traversal** of G if for all trees $T = (W, S)$ in F , the restriction of the bijection to $W \subseteq V$ is a preorder traversal of T .

In an operational view, a depth-first tree in a depth-first spanning forest of a graph is grown out of a single, initial vertex by adding arcs leading to vertices not yet in the tree, together with their target vertices. The next arc to be added goes out of the latest-added vertex which is adjacent to some vertex not in the tree, and comes precisely into some vertex not in the tree. Then, the first condition in the previous definition ensures that the tree is grown in a depth-first fashion, that is, that arcs going out of “deeper” vertices are added before arcs going out of “shallower” vertices. The second condition guarantees that the tree is grown as much as possible, that is, that it spans all the vertices reachable in the graph from the initial vertex.

Example 5.5. In the spanning tree shown in the bottom row of Fig. 5.1, arc (v, w) cannot be added to the tree before having added arc (x, y) , because vertex x is “deeper” than vertex v , that is, vertex x is visited after vertex v during a preorder traversal of the tree.

Now, the following result gives a constructive characterization of a depth-first tree in a particular depth-first spanning forest of a graph: the one generalizing to graphs the depth-first prefix leftmost traversal of a tree.

Lemma 5.6. Let $G = (V, E)$ be a graph, let $u \in V$, let $W \subseteq V$ be the set of vertices reachable in G from vertex u , and let k be the number of vertices in W . Let also the subgraph $(W, S) = (W_k, S_k)$ of G be defined by induction as follows:

- $(W_1, S_1) = (\{u\}, \emptyset)$
- $(W_{i+1}, S_{i+1}) = (W_i \cup \{w\}, S_i \cup \{(v, w)\})$ for $1 \leq i < k$, where

- j is the largest integer between 1 and i such that there are vertices $v \in W_j$ and $w \in V \setminus W_i$ with $(v, w) \in E$
- $\{v\} = W_j$ if $j = 1$, otherwise $\{v\} = W_j \setminus W_{j-1}$
- w is the smallest (according to the representation of the graph) such vertex in $V \setminus W_i$ with $(v, w) \in E$

Then, (W, S) is a depth-first spanning tree of the subgraph of G induced by W .

The following result, which will be used in the proof of Lemma 5.6, justifies the previous claim that the notion of depth-first spanning forest of a graph, indeed, generalizes the preorder traversal of a tree.

Lemma 5.7. *Let $G = (V, E)$ be a graph, and let $T = (W, S)$ be the depth-first spanning tree of the subgraph of G induced by W . Let also $\text{order} : W \rightarrow \{1, \dots, k\}$, where k is the number of vertices in W , be the bijection defined by $\text{order}[v] = i$ if $\{v\} = W_i \setminus W_{i-1}$ for all vertices $v \in W$, where $W_0 = \emptyset$. Then, the bijection $\text{order} : W \rightarrow \{1, \dots, k\}$ is the preorder traversal of $T = (W, S)$.*

Proof. It has to be shown that in the inductive definition of depth-first spanning tree of Lemma 5.6, a tree is grown by adding vertices in the order given by the preorder traversal of the tree. The tree is grown by adding a vertex adjacent to the latest-added vertex (which is adjacent to some vertex not yet in the tree) and since by Assumption 5.1, the relative order among the vertices adjacent to a given vertex coincides with the relative order among the children of the vertex in the spanning tree, previous siblings are added before next siblings when growing the tree. Therefore, the tree is grown by adding vertices in preorder. \square

Proof (Lemma 5.6). Let $G = (V, E)$ be a graph with n vertices, let $u \in V$, let $W \subseteq V$ be the set of vertices reachable in G from vertex u , let k be the number of vertices in W , and let (W, S) be the subgraph of G given by the inductive definition in Lemma 5.6. Since $S_{i+1} \setminus S_i$ contains exactly one arc (v, w) going out of a vertex $v \in W_i$ and coming into a vertex $w \in W_{i+1} \setminus W_i$, for $1 \leq i < k$, it follows that (W, S) is a tree, rooted at vertex $u \in W \subseteq V$ and spanning the subgraph of G induced by W .

Now, suppose there are arcs $(v, w) \in S$ and $(x, y) \in E \cap (W \times W)$ such that $\text{order}[v] < \text{order}[x] < \text{order}[w] \leq \text{order}[y]$, and let $W_0 = \emptyset$. Let $j = \text{order}[v]$, let $\ell = \text{order}[x]$, and let $i = \text{order}[w] - 1$. By Lemma 5.7, $\{v\} = W_j \setminus W_{j-1}$, $\{x\} = W_\ell \setminus W_{\ell-1}$, and $\{w\} = W_{i+1} \setminus W_i$. Furthermore, since $\text{order}[y] \geq \text{order}[w]$ and $w \notin W_i$, then $y \in V \setminus W_i$.

Then, $j < \ell \leq i$, $x \in W_\ell$, $y \in V \setminus W_i$, and $(x, y) \in E$, contradicting the hypothesis that j is the largest integer between 1 and i such that there is an arc in E from a vertex in W_j to a vertex in $V \setminus W_i$. Therefore, (W, S) is a depth-first spanning tree of the subgraph of G induced by W . \square

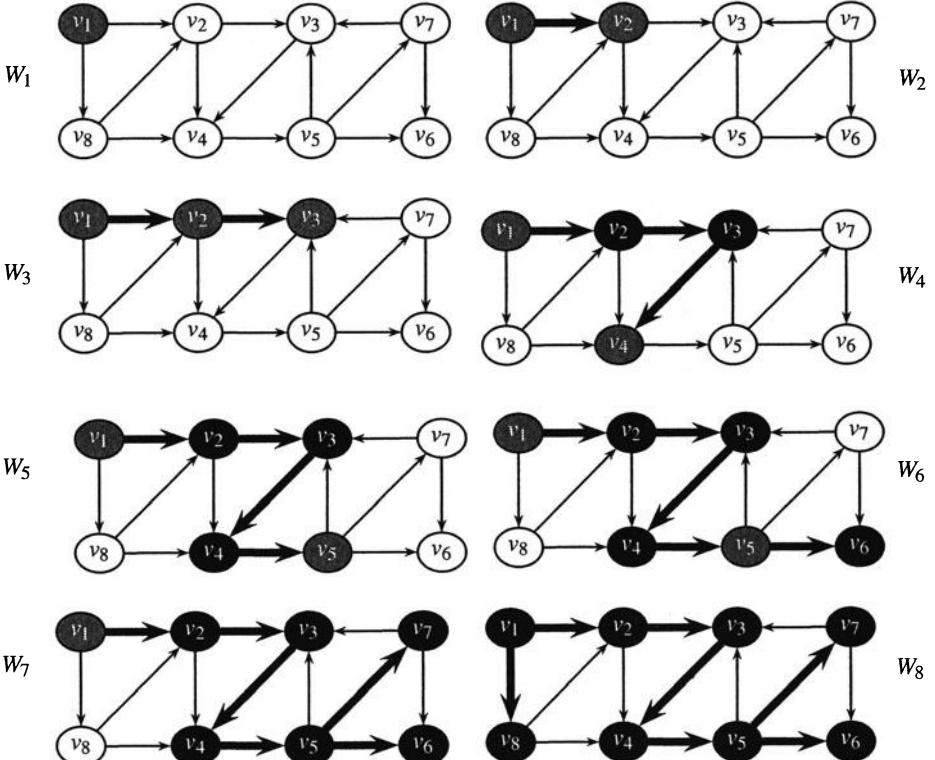


Fig. 5.2. Construction of a depth-first spanning forest $T = (V, S)$ of a graph $G = (V, E)$ with $n = 8$ vertices. Vertices are numbered according to the order in which they are added to the spanning forest. The relative order of the vertices adjacent to a given vertex corresponds to the counterclockwise ordering of the outgoing edges of the vertex in the drawing of the graph.

Example 5.8. The construction given in Lemma 5.6 for obtaining a depth-first spanning forest of a graph is illustrated in Fig. 5.2. Vertices are numbered according to the order in which they are added to the spanning forest. Since all of the vertices of the graph are reachable from the first-numbered vertex, the spanning forest consists of a single tree. Vertices not yet in the tree are shown in white, and vertices already in the tree are either shown in gray, if they are adjacent to some vertex which is not yet in the tree, otherwise they are shown in black.

Now, a simple procedure for the depth-first traversal of a graph consists of performing a preorder traversal upon each of the depth-first trees in the depth-first spanning forest of the graph. The procedure differs from preorder tree traversal, though.

- During the depth-traversal of a graph, a vertex being visited may have already been visited along a different path in the graph. Vertices need to be marked as either unvisited or visited.
- Only those vertices which are reachable from an initial vertex are visited during the traversal of the depth-first tree rooted at the initial vertex. The procedure may need to be repeated upon still unvisited vertices, until all vertices have been visited.

As in the case of tree traversal, the stack implicit in the recursive algorithm for preorder tree traversal can be made explicit, yielding a simple iterative algorithm in which a stack of vertices holds those vertices which are still waiting to be traversed.

Initially, all vertices are marked as unvisited, and the stack contains the first (according to the representation of the graph) vertex of the graph. Every time a vertex is popped and (if still unvisited) visited, those unvisited vertices adjacent to the popped vertex are pushed, one after the other, starting with the last adjacent vertex, following with the previous adjacent vertex, and so on, until having pushed the first adjacent vertex of the popped vertex. When the stack has been emptied and no vertices remain to be pushed, the next (according to the representation of the graph) still unvisited vertex of the graph, if any, is pushed and the procedure is repeated, until no unvisited vertices remain in the graph.

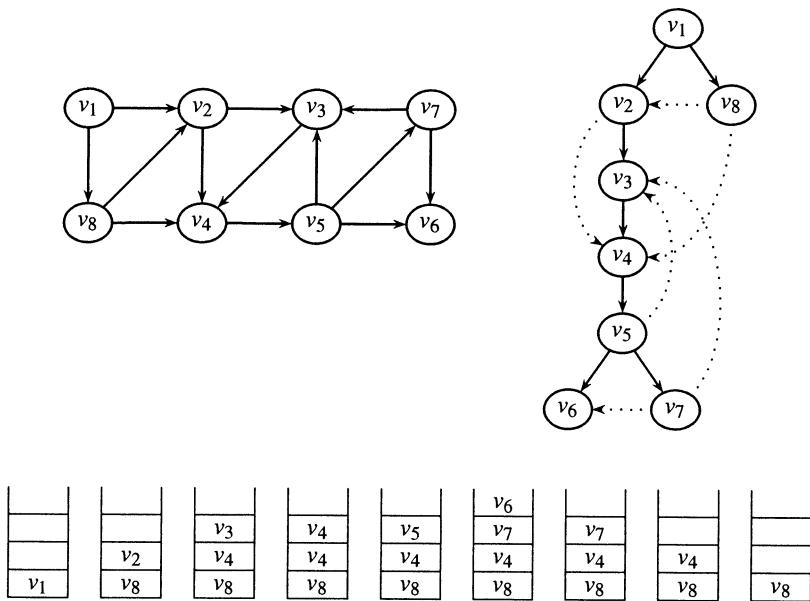


Fig. 5.3. Depth-first traversal, depth-first spanning forest, and evolution of the stack of vertices during execution of the depth-first traversal procedure, upon the graph of Fig. 5.2. Vertices are numbered according to the order in which they are first visited during the traversal. The relative order of the vertices adjacent to a given vertex corresponds to the counterclockwise ordering of the outgoing edges of the vertex in the drawing of the graph.

Example 5.9. Consider the execution of the depth-first traversal procedure with the help of a stack of vertices, illustrated in Fig. 5.3. The evolution of the stack of vertices right before a vertex is popped shows that vertices are, as a matter of fact, popped in depth-first traversal order: v_1, v_2, \dots, v_8 .

The following algorithm performs a depth-first traversal of a graph $G = (V, E)$, using a stack of vertices S to implement the previous procedure. The order in which vertices are visited during the depth-first traversal is stored in the array of vertices $\text{order} : V \rightarrow \text{int}$, where $\text{order}[v] = -1$ if vertex v has not yet been visited, for all vertices $v \in V$.

```
<graph traversal 262>≡
void depth-first_traversal(
    const graph& G,
    node_array<int>& order)
{
    node v,w;
    edge e;
```

```

forall_nodes(v,G)
order[v] = -1;

int num = 1;
stack<node> S;
forall_nodes(v,G) {
    if (order[v] ≡ -1) {
        S.push(v);
        while (¬S.empty()) {
            v = S.pop();
            if (order[v] ≡ -1) {
                order[v] = num++; // visit vertex v
                e = G.last_adj_edge(v);
                while (e ≠ nil) {
                    w = G.opposite(v,e);
                    if (order[w] ≡ -1) S.push(w);
                    e = G.adj_pred(e);
                }
            }
        }
    }
}

⟨double-check depth-first forest 263⟩
}

```

The following double-check of depth-first traversal, although being redundant, gives some reassurance of the correctness of the implementation. It verifies that *order* is a depth-first traversal of graph G , according to Definition 5.2.

263 ⟨double-check depth-first forest 263⟩ ≡
 { node v,w,x,y ;
 edge e,ee ;
 node_array<node> root(G);
 list<edge> S ;

```

<find roots and arcs in depth-first forest 264>

forall_edges(e,G) {
  v = G.source(e);
  w = G.target(e);
  if ( order[root[v]] < order[root[w]] ) {
    error_handler(1,
      "Wrong implementation of depth-first traversal");
  }
}

forall(e,S) {
  v = G.source(e);
  w = G.target(e);
  forall_edges(ee,G) {
    x = G.source(ee);
    y = G.target(ee);
    if ( order[v] < order[x] &

```

```

order[x] < order[w] ∧
order[w] ≤ order[y] )
error_handler(1,
"Wrong implementation of depth-first traversal");
} } }

```

The root of the tree in the forest to which each vertex belongs is collected in an array $root$ of vertices indexed by the vertices of G , and the arcs in the depth-first spanning forest of G are collected in a list S of arcs, as follows. The first-numbered vertex is the root of a tree in the forest, as are those vertices whose indegree is equal to zero or which has no incoming arcs from lower-numbered vertices. Those vertices v which are not the root of a tree in the forest, have an incoming tree arc going out of the highest-numbered vertex w which is not numbered higher than vertex v itself.

264

\langle find roots and arcs in depth-first forest 264 $\rangle \equiv$

```

{ int n = G.number_of_nodes();
array<node> disorder(1,n);
forall_nodes(v,G)
disorder[order[v]] = v;

root[disorder[1]] = disorder[1];
for (int i = 2; i ≤ n; i++) {
v = disorder[i];
if (G.indeg(v) ≡ 0) {
root[v] = v;
} else {
bool nonroot = false;
forall_in_edges(e,v) {
if (order[G.source(e)] < order[v]) {
nonroot = true;
break;
}
}
if (¬nonroot) {
root[v] = v;
} else {
forall_in_edges(e,v)
if (order[G.source(e)] < order[v])
break;
forall_in_edges(ee,v)
if (order[G.source(ee)] > order[G.source(e)] ∧
order[G.source(ee)] < order[v])
e = ee;
S.append(e);
root[v] = root[G.source(e)];
}
}
}
}

```

Lemma 5.10. *The algorithm for depth-first traversal of a graph runs in $O(n + m)$ time using $O(m)$ additional space, where n is the number of vertices and m is the number of arcs in the graph.*

Proof. Let $G = (V, E)$ be a graph with n vertices and m arcs. Since each vertex of the graph is pushed and popped at most once for each incoming arc, the loop is executed $n + m$ times and the algorithm runs in $O(n + m)$ time. Further, since each vertex is pushed at most once for each incoming arc, the stack cannot ever contain more than m vertices and the algorithm uses $O(m)$ additional space.

The double-check of depth-first graph traversal runs in $O(m^2)$ time using $O(1)$ additional space. \square

The depth-first traversal procedure can also be applied to those vertices which are reachable from an initial vertex, in order to perform a traversal of the depth-first tree rooted at the initial vertex. The following procedure, which is based on the previous depth-first graph traversal algorithm, not only computes the order in which vertices are visited during the traversal, but also the depth-first spanning tree (W, S) of the subgraph of a graph $G = (V, E)$ induced by the set of vertices $W \subseteq V$ reachable from an initial vertex $v \in V$.

265

```

⟨graph traversal 262⟩ +≡
void depth_first_spanning_subtree(
    const graph& G,
    node v,
    node_array<int>& order,
    set<node>& W,
    set<edge>& S)
{
    node w;
    edge e;
    forall_nodes(w,G)
        order[w] = -1;

    int num = 1;
    W.insert(v);
    order[v] = num++; // visit vertex v

    stack<edge> Z;
    e = G.last_adj_edge(v);
    while (e ≠ nil) {
        w = G.opposite(v,e);
        Z.push(e);
        e = G.adj_pred(e);
    }
}
```

```

while (  $\neg Z.empty()$  ) {
   $e = Z.pop();$ 
   $v = G.target(e);$ 
  if (  $order[v] \equiv -1$  ) {
     $W.insert(v);$ 
     $S.insert(e);$ 
     $order[v] = num++;$  // visit vertex  $v$ 
     $e = G.last\_adj\_edge(v);$ 
    while (  $e \neq nil$  ) {
       $w = G.opposite(v,e);$ 
      if (  $order[w] \equiv -1$  )
         $Z.push(e);$ 
         $e = G.adj\_pred(e);$ 
    } } }
} 
```

⟨double-check depth-first tree 266⟩

The following double-check of depth-first tree, although being redundant, gives some reassurance of the correctness of the implementation. It verifies that $order$ is a depth-first traversal of the depth-first tree (W,S) of G , according to Definition 5.2.

266 ⟨double-check depth-first tree 266⟩ ≡

```

{ node v,w,x,y;
  edge e,ee;
  forall(e,S) {
    forall_edges(ee,G) {
      v = G.source(e);
      w = G.target(e);
      x = G.source(ee);
      y = G.target(ee);
      if (  $order[v] < order[x] \wedge$ 
             $order[x] < order[w] \wedge$ 
             $order[w] \leqslant order[y]$  )
        error_handler(1,
          "Wrong implementation of depth-first traversal");
    } } } 
```

5.1.1 Interactive Demonstration of Depth-First Traversal

The algorithm for depth-first traversal is integrated next in the interactive demonstration of graph algorithms. A simple checker for depth-first traversal that provides some visual reassurance consists of numbering the vertices according to the order in which they are first vis-

ited during the traversal, highlighting also the vertices and arcs of the graph as the depth-first traversal proceeds.

267a
 ⟨demo graph traversal 267a⟩≡
 void gw_depth_first_traversal(
 GraphWin& gw)
{
graph& G = gw.get_graph();
⟨put ordered graph in standard representation 269⟩
gw.update_graph();
node_array<int> order(G);
depth_first_traversal(G,order);
⟨show depth-first graph traversal 267b⟩
}

A depth-first graph traversal can be demonstrated by highlighting each of the vertices of the graph in turn, according to the order defined by the traversal. A pause of half a second between vertices should suffice to get the picture of how the traversal proceeds.

267b
⟨show depth-first graph traversal 267b⟩≡
{ gw.save_all_attributes();
color gray = grey3;

node v,w,x;
forall_nodes(v,G) {
gw.set_color(v,white);
gw.set_label(v,string("%i ",order[v]));
}

int n = G.number_of_nodes();
array<node> disorder(1,n);
forall_nodes(v,G)
disorder[order[v]] = v;

edge e,ee;
for (int i = 1; i ≤ n; i++) {
v = disorder[i];

⟨highlight discovered vertex 268a⟩
⟨highlight incoming depth-first tree arc 268b⟩
⟨highlight predecessor vertices 268c⟩

leda_wait(0.5);
}

gw.wait();
gw.restore_all_attributes();
}

Vertices are highlighted in gray when they are *discovered*, that is, when they are reached for the first time during the traversal, and they are highlighted in black when their processing has finished, that is, when all their adjacent vertices have already been discovered.

268a ⟨highlight discovered vertex 268a⟩≡

```
gw.set_color(v,gray);
{ bool finished = true;
forall_adj_nodes(w,v)
  if ( gw.get_color(w) ≡ white ) finished = false;
if (finished) {
  gw.set_color(v,black);
  led_a_wait(0.5);
} }
```

Those vertices v which are not the root of a tree in the depth-first forest, have an incoming depth-first tree arc (w, v) going out of the highest-numbered visited vertex w .

268b ⟨highlight incoming depth-first tree arc 268b⟩≡

```
{ int pred_num = 1;
bool pred = false;
forall_in_edges(e,v) {
  w = G.opposite(v,e);
  if ( gw.get_color(w) ≠ white ∧
       order[w] ≥ pred_num ) {
    ee = e;
    pred_num = order[w];
    pred = true;
  }
}
if ( pred ) gw.set_width(ee,5);
led_a_wait(0.5);
}
```

Predecessor gray vertices whose processing has also finished, that is, all of whose adjacent vertices have already been visited, are highlighted in black.

268c ⟨highlight predecessor vertices 268c⟩≡

```
forall_in_edges(e,v) {
  w = G.opposite(v,e);
  if ( gw.get_color(w) ≡ gray ) {
    bool finished = true;
    forall_out_edges(ee,w)
      if ( gw.get_color(G.opposite(w,ee)) ≡ white )
        finished = false;
    if (finished) {
      gw.set_color(w,black);
      led_a_wait(0.5);
    }
  }
}
```

The actual LEDA representation of a graph to be traversed can be easily rearranged in order to meet Assumption 5.1 about the relative order of vertices and arcs in the graph, as follows. Recall that in the LEDA representation of a graph, each vertex is assigned a unique *index* integer according to the order of the vertices fixed by the representation.

Sorting the arcs in the representation of a graph according to the order of their source vertex and, for those arcs going out of the same vertex, also according to the order of their target vertex, guarantees that the order of the arcs coming into a vertex is the order of their source vertices, and that the order of the arcs going out of a vertex is the order of their target vertices.

Now, the representation of a graph $G = (V, E)$ with n vertices and m arcs can be rearranged in $O(n + m)$ time using $O(n + m)$ additional space by bucket sorting the arcs of G twice, first according to the index of their target vertex and then according to the index of their source vertex, because the index of a vertex $v \in V$ is an integer between zero and $n - 1$, and bucket sort is a stable sorting method.

269

`(put ordered graph in standard representation 269)≡`

```

{ edge e;
  edge_array<int> S(G),T(G);
  forall_edges(e,G) {
    S[e] = index(G.source(e));
    T[e] = index(G.target(e));
  }
  G.bucket_sort_edges(T);
  G.bucket_sort_edges(S);
}
```

Notice that, after having rearranged the LEDA representation of a graph, the relative order of the vertices adjacent to a given vertex may no longer correspond to the counterclockwise ordering of the outgoing arcs of the vertex in the layout of the graph. See Exercise 5.4.

Now, the algorithm for traversing a depth-first tree is also integrated next in the interactive demonstration of graph algorithms. A checker for depth-first tree traversal that provides some visual reassurance consists of numbering the vertices according to the order in which they are first visited during the traversal, highlighting also the vertices and arcs of the depth-first tree and producing a separate layered layout of the depth-first tree as well.

270a ⟨demo graph traversal 267a⟩ + ≡

```

void gw_depth_first_spanning_subtree(
    GraphWin& gw)
{
    graph& G = gw.get_graph();
    ⟨put ordered graph in standard representation 269⟩
    gw.update_graph();
    node v = gw.ask_node();
    node_array<int> order(G);
    set<node> W;
    set<edge> S;
    depth_first_spanning_subtree(G,v,order,W,S);
    ⟨show depth-first spanning subtree 270b⟩
}

```

The layered layout of the depth-first tree is produced using the simple layered layout algorithm for rooted trees given in Sect. 3.5. Since the LEDA operations for hiding and restoring vertices and arcs would change the ordered structure of the graph, a rooted tree representation of the subgraph corresponding to the depth-first tree is obtained instead by deleting from a copy of the whole graph, those vertices and arcs which do not belong to the depth-first tree.

270b ⟨show depth-first spanning subtree 270b⟩ ≡

```

{ gw.save_all_attributes();
  color gray = grey3;
  forall_nodes(v,G)
    gw.set_label(v," ");
  forall(v,W) {
    gw.set_color(v,gray);
    gw.set_label(v,string("%i",order[v]));
  }
  edge e;
  forall(e,S)
    gw.set_width(e,3);

  GRAPH<node,edge> H;
  CopyGraph(H,G);

  forall_nodes(v,H)
    if (  $\neg$ W.member(H[v]) )
      H.del_node(v);
  forall_edges(e,H)
    if (  $\neg$ S.member(H[e]) )
      H.del_edge(e);
}

```

```

tree T(H);
GraphWin gw2(T,500,500,"Depth-First Spanning Subtree");
gw2.display();

```

```

forall_nodes(v,T)
  gw2.set_label(v,gw.get_label(H[v]));

node_array<double> xcoord(T);
node_array<double> ycoord(T);
layered_tree.layout(T,xcoord,ycoord);
gw2.adjust_coords_to_win(xcoord,ycoord);
gw2.set_layout(xcoord,ycoord);
gw2.display();

gw.wait();
gw.restore_all_attributes();
}

```

5.1.2 Leftmost Depth-First Traversal of a Graph

An interesting particular case of the depth-first traversal of an undirected graph consists of performing the traversal according to the ordered structure of the graph. Recall from Sect. 1.1 that an ordered graph is a graph that has been embedded in a certain surface, that is, such that the relative order of the adjacent vertices is fixed for each vertex. As a matter of fact, there is always a relative order of the vertices adjacent to each vertex, fixed by the graph representation.

The leftmost depth-first traversal of an undirected graph is related to the problem of finding an Euler trail through the graph. Recall from Sect. 1.1 that an undirected graph is Eulerian if it has an Euler cycle, that is, a closed trail containing all the vertices and edges of the graph, and that a nontrivial, connected undirected graph is Eulerian if and only if every vertex in the graph has even degree.

Now, an Euler cycle through a nontrivial, connected bidirected graph can be constructed by traversing each edge exactly once in each direction, which guarantees that the degree of each vertex is even and, therefore, the bidirected graph is indeed Eulerian.

Such a traversal is called a **leftmost depth-first traversal**, since the edges are explored in left-to-right order (if drawn downwards) for any vertex of the graph and, more generally, the whole graph is explored in a left-to-right fashion.

A simple method for the leftmost depth-first traversal of a bidirected graph is based on maze traversal methods. Starting with an edge traversed in one of its directions,

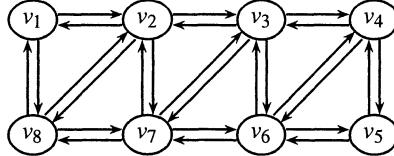


Fig. 5.4. A leftmost depth-first traversal of a bidirected, ordered graph. The relative order of the vertices adjacent to a given vertex reflects the counterclockwise ordering of the outgoing edges of the vertex in the drawing. Vertices are numbered according to the order in which they are first visited during the traversal.

- When an unvisited vertex is reached, the next (in the counterclockwise ordering of the edges around the vertex) edge is traversed.
- When a visited vertex is reached along an unvisited edge, the same edge is traversed again but in the opposite direction.
- When a visited vertex is reached along a visited edge, the next (in the counterclockwise ordering of the edges around the vertex) unvisited edge, if any, is traversed.

Example 5.11. The execution of the leftmost depth-first traversal procedure, starting in turn with each of the arcs of the ordered graph of Fig. 5.4, is illustrated in Fig. 5.5. For instance, the closed trail $[v_1, v_2, v_8, v_1, v_8, v_7, v_6, v_5, v_4, v_3, v_2, v_3, v_7, v_3, v_6, v_3, v_4, v_6, v_4, v_5, v_6, v_7, v_2, v_7, v_8, v_2, v_1]$ corresponds to the leftmost depth-first traversal starting with arc (v_1, v_2) .

The following recursive algorithm performs a leftmost depth-first traversal of a bidirected graph $G = (V, E)$, implementing the previous procedure. The arcs of the graph are stored in the list of arcs L , in the leftmost depth-first order in which they are traversed.

```

graph traversal 262) +≡
void leftmost_depth_first_traversal(
    const graph& G,
    edge e,
    list<edge>& L)
{
    node_array<bool> node_visited(G, false);
    edge_array<bool> edge_visited(G, false);
    node_visited[G.source(e)] = true;
    leftmost_depth_first_traversal(G,
        node_visited, edge_visited, e, L);
}

```

```

[v1,v2,v8,v1,v8,v7,v6,v5,v4,v3,v2,v3,v7,v3,v6,v3,v4,v6,v4,v5,v6,v7,v2,v7,v8,v2,v1]
[v1,v8,v7,v6,v5,v4,v3,v2,v1,v2,v8,v2,v7,v2,v3,v7,v3,v6,v3,v4,v6,v4,v5,v6,v7,v8,v1]
[v2,v1,v8,v7,v6,v5,v4,v3,v2,v3,v7,v3,v6,v3,v4,v6,v4,v5,v6,v7,v2,v7,v8,v2,v8,v1,v2]
[v2,v8,v1,v2,v1,v8,v7,v6,v5,v4,v3,v2,v3,v7,v3,v6,v3,v4,v6,v4,v5,v6,v7,v2,v7,v8,v2]
[v2,v7,v8,v2,v8,v1,v2,v1,v8,v7,v6,v5,v4,v3,v2,v3,v7,v3,v6,v3,v4,v6,v4,v5,v6,v7,v2]
[v2,v3,v7,v2,v7,v8,v2,v8,v1,v2,v1,v8,v7,v6,v5,v4,v3,v4,v6,v4,v5,v6,v3,v6,v7,v3,v2]
[v3,v2,v1,v8,v7,v6,v5,v4,v3,v4,v6,v4,v5,v6,v3,v6,v7,v3,v7,v2,v7,v8,v2,v8,v1,v2,v3]
[v3,v7,v2,v3,v2,v1,v8,v7,v8,v2,v8,v1,v2,v7,v6,v5,v4,v3,v4,v6,v4,v5,v6,v3,v6,v7,v3]
[v3,v6,v7,v3,v7,v2,v3,v2,v1,v8,v7,v8,v2,v8,v1,v2,v7,v6,v5,v4,v3,v4,v6,v4,v5,v6,v3]
[v3,v4,v6,v3,v6,v7,v3,v7,v2,v3,v2,v1,v8,v7,v8,v2,v8,v1,v2,v7,v6,v5,v4,v5,v6,v4,v3]
[v4,v3,v2,v1,v8,v7,v6,v5,v4,v5,v6,v4,v6,v3,v6,v7,v3,v7,v2,v7,v8,v2,v8,v1,v2,v3,v4]
[v4,v6,v3,v4,v3,v2,v1,v8,v7,v6,v7,v3,v7,v2,v7,v8,v2,v8,v1,v2,v3,v6,v5,v4,v5,v6,v4]
[v4,v5,v6,v4,v6,v3,v4,v3,v2,v1,v8,v7,v6,v7,v3,v7,v2,v7,v8,v2,v8,v1,v2,v3,v6,v5,v4]
[v5,v4,v3,v2,v1,v8,v7,v6,v5,v6,v4,v6,v3,v6,v7,v3,v7,v2,v7,v8,v2,v8,v1,v2,v3,v4,v5]
[v5,v6,v4,v5,v4,v3,v2,v1,v8,v7,v6,v7,v3,v7,v2,v7,v8,v2,v8,v1,v2,v3,v6,v3,v4,v6,v5]
[v6,v5,v4,v3,v2,v1,v8,v7,v6,v7,v3,v7,v2,v7,v8,v2,v8,v1,v2,v3,v6,v3,v4,v6,v4,v5,v6]
[v6,v4,v5,v6,v5,v4,v3,v2,v1,v8,v7,v6,v7,v3,v7,v2,v7,v8,v2,v8,v1,v2,v3,v6,v3,v4,v6]
[v6,v3,v4,v6,v4,v5,v6,v5,v4,v3,v2,v1,v8,v7,v6,v7,v3,v7,v2,v7,v8,v2,v8,v1,v2,v3,v6]
[v6,v7,v3,v6,v3,v4,v6,v4,v5,v6,v5,v4,v3,v2,v1,v8,v7,v8,v2,v8,v1,v2,v7,v2,v3,v7,v6]
[v7,v6,v5,v4,v3,v2,v1,v8,v7,v8,v2,v8,v1,v2,v7,v2,v3,v7,v3,v6,v3,v4,v6,v4,v5,v6,v7]
[v7,v3,v6,v7,v6,v5,v4,v3,v4,v6,v4,v5,v6,v3,v2,v1,v8,v7,v8,v2,v8,v1,v2,v7,v2,v3,v7]
[v7,v2,v3,v7,v3,v6,v7,v6,v5,v4,v3,v4,v6,v4,v5,v6,v3,v2,v1,v8,v7,v8,v2,v8,v1,v2,v7]
[v7,v8,v2,v7,v2,v3,v7,v3,v6,v7,v6,v5,v4,v3,v4,v6,v4,v5,v6,v3,v2,v1,v8,v1,v2,v8,v7]
[v8,v1,v2,v8,v2,v7,v8,v7,v6,v5,v4,v3,v2,v3,v7,v3,v6,v3,v4,v6,v4,v5,v6,v7,v2,v1,v8]
[v8,v7,v6,v5,v4,v3,v2,v1,v8,v1,v2,v8,v2,v7,v2,v3,v7,v3,v6,v3,v4,v6,v4,v5,v6,v7,v8]
[v8,v2,v7,v8,v7,v6,v5,v4,v3,v2,v3,v7,v3,v6,v3,v4,v6,v4,v5,v6,v7,v2,v1,v8,v1,v2,v8]
}
```

Fig. 5.5. Execution of the leftmost depth-first traversal procedure, starting in turn with each arc of the ordered graph of Fig. 5.4.

}

The arrays of vertices and arcs hidden from the user by the previous procedure are needed by the leftmost depth-first traversal procedure to keep track of traversed arcs and visited vertices along the traversed arcs. The following recursive algorithm implements the actual leftmost depth-first traversal procedure.

```

⟨graph traversal 262⟩ +≡
void leftmost_depth_first_traversal(
    const graph& G,
    node_array<bool>& node_visited,
    edge_array<bool>& edge_visited,
    edge e,
    list<edge>& L)
{
    node v = G.target(e);
    edge eprime;
    edge erev = G.reversal(e);
}
```

```

L.append(e);
if ( node_visited[v] ) {
    if ( edge_visited[erev] ) {
        eprime = errev;
        do {
            eprime = G.cyclic_adj_succ(eprime);
        } while ( eprime ≠ errev ∧ edge_visited[eprime] );
        if ( edge_visited[eprime] ) return;
    }
    else {
        eprime = errev;
    }
} else {
    eprime = G.cyclic_adj_succ(errev);
}
edge_visited[e] = true;
node_visited[v] = true;
return leftmost_depth_first_traversal(G,
    node_visited,edge_visited,eprime,L);
}
}

```

Interactive Demonstration of Leftmost Depth-First Traversal

The algorithm for leftmost depth-first traversal is integrated next in the interactive demonstration of graph algorithms. The user is asked to select an initial arc for the traversal.

A simple checker for leftmost depth-first traversal that provides some visual reassurance consists of numbering the vertices according to the order in which they are first visited during the traversal, highlighting also the vertices and arcs of the bidirected graph as the leftmost depth-first traversal proceeds.

274

```

(demo graph traversal 267a) +≡
void gw_leftmost_depth_first_traversal(
    GraphWin& gw)
{
    graph& G = gw.get_graph();

    panel P;
    if ( !Is_Bidirected(G) ) {
        make_proof_panel(P,"This graph is \\\'red not bidirected",false);
        gw.open_panel(P);
        return;
    }

    G.make_map(); // set edge reversal information

    list<edge> L;

```

```

edge e = gw.ask_edge();
leftmost_depth_first_traversal(G,e,L);
⟨show leftmost depth-first graph traversal 275⟩
}

```

The vertices are numbered and highlighted in the order in which they are first visited during the traversal, and the arcs are also highlighted in the leftmost depth-first order in which they are traversed. A pause of half a second between arcs should suffice to get the picture of how the traversal proceeds.

275 ⟨show leftmost depth-first graph traversal 275⟩≡

```

{ gw.save_all_attributes();

node_array<int> order(G);
node v;
forall_nodes(v,G)
order[v] = -1;
edge e = L.head();
int num = 1;
order[G.source(e)] = num++;
forall(e,L)
if ( order[G.target(e)] ≡ -1 )
order[G.target(e)] = num++;

forall_nodes(v,G) {
gw.set_color(v,white);
gw.set_label(v,string("%i",order[v]));
}

gw.set_color(G.source(L.front()),blue);
forall(e,L) {
gw.set_width(e,2);
gw.set_color(e,red);
if ( gw.get_color(G.target(e)) ≡ white )
gw.set_color(G.target(e),blue);
leda_wait(0.5);
}

gw.wait();
gw.restore_all_attributes();
}
```

5.2 Breadth-First Traversal of a Graph

The counterpart to the top-down traversal of a tree is the top-down traversal of a graph, which is the top-down traversal of a breadth-first spanning forest of a graph. Recall that a spanning forest of a graph is an ordered set of pairwise-disjoint subgraphs of the graph which are rooted trees and which, together, span all the vertices of the graph.

Definition 5.12. Let $G = (V, E)$ be a graph with n vertices. A spanning forest F of G is a **breadth-first spanning forest** of G if the following conditions are satisfied,

- for all trees $T = (W, S)$ in F and for all arcs $(v, w) \in S$, there is no arc $(x, y) \in E \cap (W \times W)$ such that $\text{order}[x] < \text{order}[v] < \text{order}[w] \leq \text{order}[y]$
- for all trees $T_i = (W_i, S_i)$ and $T_j = (W_j, S_j)$ in F with $i < j$, there is no arc $(v, w) \in E$ with $v \in W_i$ and $w \in W_j$

where $\text{order} : W \rightarrow \{1, \dots, k\}$ is the top-down traversal of a rooted tree $T = (W, S)$ with $k \leq n$ nodes.

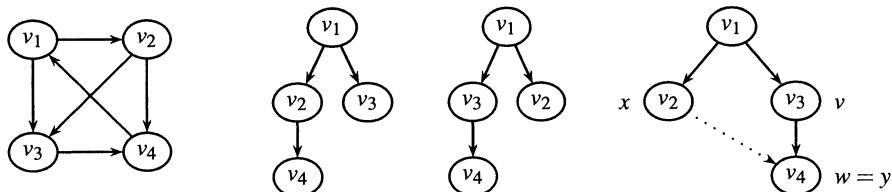


Fig. 5.6. Breadth-first spanning forest of a graph. All the breadth-first spanning trees rooted at vertex v_1 of the graph to the left of the figure are also shown in the middle of the figure. The spanning tree shown to the right of the figure is not a breadth-first spanning tree.

Example 5.13. The notion of a breadth-first spanning forest of a graph is illustrated in Fig. 5.6. Since all vertices of the graph are reachable from vertex v_1 , every breadth-first spanning forest of the graph whose initial vertex is v_1 must consist of a single tree, which is therefore a spanning tree of the graph. The spanning tree shown to the right of Fig. 5.6 is not a breadth-first spanning tree of the graph, because

there is an arc (v, w) in the tree and an arc (x, y) not in the tree with $\text{order}[x] = 2$, $\text{order}[v] = 3$, and $\text{order}[w] = \text{order}[y] = 4$, that is, with $\text{order}[v] < \text{order}[x] < \text{order}[w] \leq \text{order}[y]$.

Now, a breadth-first traversal of a graph is just a top-down traversal of each of the trees in the breadth-first spanning forest of the graph.

Definition 5.14. Let $G = (V, E)$ be a graph with n vertices, and let F be the breadth-first spanning tree of G . A bijection $\text{order} : V \rightarrow \{1, \dots, n\}$ is a **breadth-first traversal** of G if for all trees $T = (W, S)$ in F , the restriction of the bijection to $W \subseteq V$ is a top-down traversal of T .

In an operational view, a breadth-first tree in a breadth-first spanning forest of a graph is grown out of a single, initial vertex by adding arcs leading to vertices not yet in the tree, together with their target vertices. The next arc to be added goes out of the soonest-added vertex which is adjacent to some vertex not in the tree, and comes precisely into some vertex not in the tree. Then, the first condition in the previous definition ensures that the tree is grown in a breadth-first fashion, that is, that arcs going out of “shallow” vertices are added before arcs going out of “deep” vertices. The second condition guarantees that the tree is grown as much as possible, that is, that it spans all the vertices reachable in the graph from the initial vertex.

Example 5.15. In the spanning tree shown to the right of Fig. 5.6, arc (v, w) cannot be added to the tree before having added arc (x, y) , because vertex x is “shallow” than vertex v , that is, vertex x is visited before vertex v during a top-down traversal of the tree.

The following result gives a constructive characterization of a breadth-first tree in a particular breadth-first spanning forest of a graph: the one generalizing to graphs the breadth-first leftmost traversal of a tree.

Lemma 5.16. Let $G = (V, E)$ be a graph, let $u \in V$, let $W \subseteq V$ be the set of vertices reachable in G from vertex u , and let k be the number of vertices in W . Let also the subgraph $(W, S) = (W_k, S_k)$ of G be defined by induction as follows:

- $(W_1, S_1) = (\{u\}, \emptyset)$
- $(W_{i+1}, S_{i+1}) = (W_i \cup \{w\}, S_i \cup \{(v, w)\})$ for $1 \leq i < k$, where
 - j is the smallest integer between 1 and i such that there are vertices $v \in W_j$ and $w \in V \setminus W_i$ with $(v, w) \in E$
 - $\{v\} = W_j$ if $j = 1$, otherwise $\{v\} = W_j \setminus W_{j-1}$
 - w is the smallest (according to the representation of the graph) such vertex in $V \setminus W_i$ with $(v, w) \in E$

Then, (W, S) is a breadth-first spanning tree of the subgraph of G induced by W .

The following result, which will be used in the proof of Lemma 5.16, justifies the previous claim that the notion of breadth-first spanning forest of a graph, indeed, generalizes the top-down traversal of a tree.

Lemma 5.17. *Let $G = (V, E)$ be a graph, and let $T = (W, S)$ be the breadth-first spanning tree of the subgraph of G induced by W . Let also $\text{order} : W \rightarrow \{1, \dots, k\}$, where k is the number of vertices in W , be the bijection defined by $\text{order}[v] = i$ if $\{v\} = W_i \setminus W_{i-1}$ for all vertices $v \in W$, where $W_0 = \emptyset$. Then, the bijection $\text{order} : W \rightarrow \{1, \dots, k\}$ is the top-down traversal of $T = (W, S)$.*

Proof. It has to be shown that in the inductive definition of breadth-first spanning tree of Lemma 5.16, a tree is grown by adding vertices in the order given by the top-down traversal of the tree. The tree is grown by adding a vertex adjacent to the soonest-added vertex (which is adjacent to some vertex not yet in the tree) and since by Assumption 5.1, the relative order among the vertices adjacent to a given vertex coincides with the relative order among the children of the vertex in the spanning tree, previous siblings are added before next siblings when growing the tree. Therefore, the tree is grown by adding vertices in top-down order. \square

Proof (Lemma 5.16). Let $G = (V, E)$ be a graph with n vertices, let $u \in V$, let $W \subseteq V$ be the set of vertices reachable in G from vertex u , let k be the number of vertices in W , and let (W, S) be the subgraph of G given by the inductive definition in Lemma 5.16. Since $S_{i+1} \setminus S_i$ contains exactly one arc (v, w) going out of a vertex $v \in W_i$ and coming into a vertex $w \in W_{i+1} \setminus W_i$, for $1 \leq i < k$, it follows that (W, S) is a tree,

rooted at vertex $u \in W \subseteq V$ and spanning the subgraph of G induced by W .

Now, suppose there are arcs $(v, w) \in S$ and $(x, y) \in E \cap (W \times W)$ such that $\text{order}[x] < \text{order}[v] < \text{order}[w] \leq \text{order}[y]$, and let $W_0 = \emptyset$. Let $\ell = \text{order}[x]$, let $j = \text{order}[v]$, and let $i = \text{order}[w] - 1$. By Lemma 5.17, $\{x\} = W_\ell \setminus W_{\ell-1}$, $\{v\} = W_j \setminus W_{j-1}$, and $\{w\} = W_{i+1} \setminus W_i$. Furthermore, since $\text{order}[y] \geq \text{order}[w]$ and $w \notin W_i$, then $y \in V \setminus W_i$.

Then, $1 \leq \ell < j \leq i$, $x \in W_\ell$, $y \in V \setminus W_i$, and $(x, y) \in E$, contradicting the hypothesis that j is the smallest integer between 1 and i such that there is an arc in E from a vertex in W_j to a vertex in $V \setminus W_i$. Therefore, (W, S) is a breadth-first spanning tree of the subgraph of G induced by W . \square

Example 5.18. The construction given in Lemma 5.16 above for obtaining a breadth-first spanning forest of a graph is illustrated in Fig. 5.7. Vertices are numbered according to the order in which they are added to the spanning forest. Since all of the vertices of the graph are reachable from the first-numbered vertex, the spanning forest consists of a single tree. Vertices not yet in the tree are shown in white, and vertices already in the tree are either shown in gray, if they are adjacent to some vertex which is not yet in the tree, otherwise they are shown in black.

Now, a simple procedure for the breadth-first traversal of a graph consists of performing a top-down traversal upon each of the breadth-first trees in the breadth-first spanning forest of the graph. The procedure differs from top-down tree traversal, though.

- During the breadth-traversal of a graph, a vertex being visited may have already been visited along a different path in the graph. Vertices need to be marked as either unvisited or visited.
- Only those vertices which are reachable from an initial vertex are visited during the traversal of the breadth-first tree rooted at the initial vertex. The procedure may need to be repeated upon still unvisited vertices, until all vertices have been visited.

As in the case of top-down tree traversal, a breadth-first traversal of a graph can be easily realized with the help of a queue of vertices, which holds those vertices which are still waiting to be traversed.

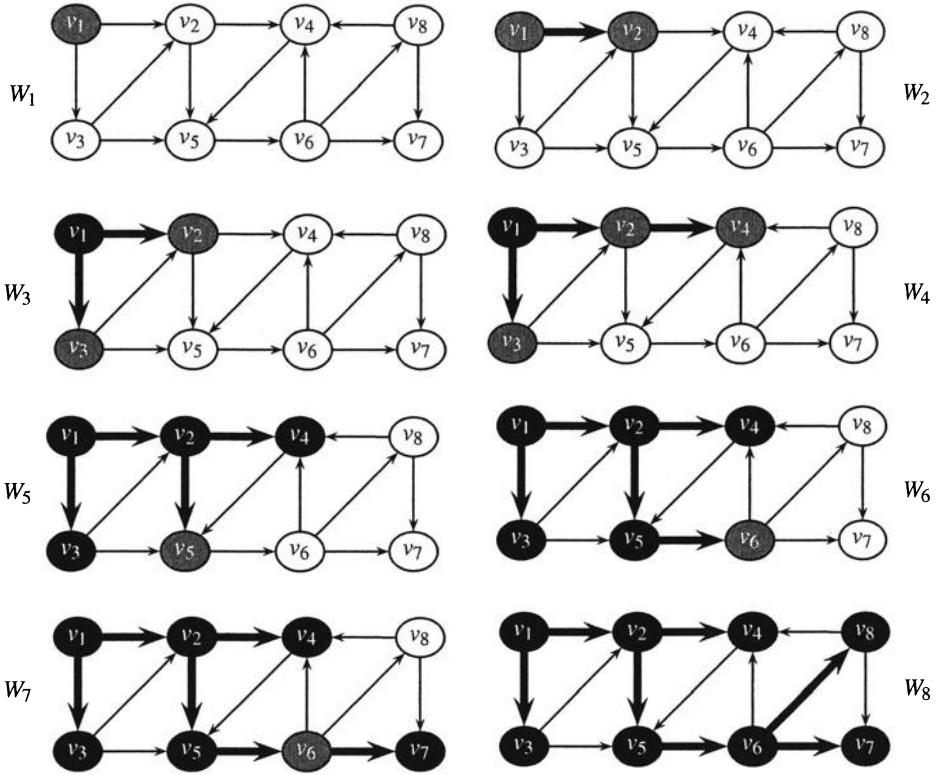


Fig. 5.7. Construction of a breadth-first spanning forest $T = (V, S)$ of a graph $G = (V, E)$ with $n = 8$ vertices. Vertices are numbered according to the order in which they are added to the spanning forest. The relative order of the vertices adjacent to a given vertex corresponds to the counterclockwise ordering of the outgoing edges of the vertex in the drawing of the graph.

Initially, all vertices are marked as unvisited, and the queue contains the first (according to the representation of the graph) vertex of the graph. Every time a vertex is dequeued and (if still unvisited) visited, those unvisited vertices adjacent to the dequeued vertex are enqueued, one after the other, starting with the first adjacent vertex, following with the next adjacent vertex, and so on, until having pushed the last adjacent vertex of the popped vertex. When the queue has been emptied and no vertices remain to be enqueued, the next (according to the representation of the graph) still unvisited vertex of the graph, if any, is enqueued and the procedure is repeated, until no unvisited vertices remain in the graph.

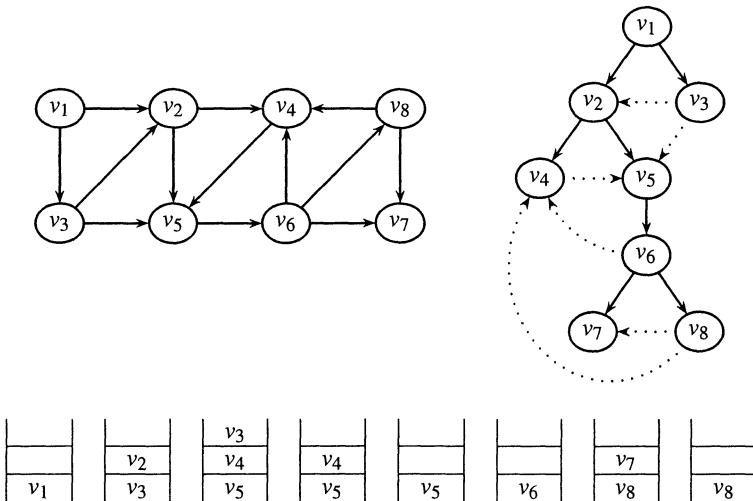


Fig. 5.8. Breadth-first traversal, breadth-first spanning forest, and evolution of the queue of vertices during execution of the breadth-first traversal procedure, upon the graph of Fig. 5.7. Vertices are numbered according to the order in which they are first visited during the traversal. The relative order of the vertices adjacent to a given vertex corresponds to the counter-clockwise ordering of the outgoing edges of the vertex in the drawing of the graph.

Example 5.19. Consider the execution of the breadth-first traversal procedure with the help of a queue of vertices, illustrated in Fig. 5.8. The evolution of the queue of vertices right before a vertex is dequeued shows that vertices are, as a matter of fact, dequeued in breadth-first traversal order: v_1, v_2, \dots, v_8 .

The following algorithm performs a breadth-first traversal of a graph $G = (V, E)$, using a queue of vertices Q to implement the previous procedure. The order in which vertices are visited during the breadth-first traversal is stored in the array of vertices $\text{order} : V \rightarrow \text{int}$, where $\text{order}[v] = -1$ if vertex v has not yet been visited, for all vertices $v \in V$.

```
<graph traversal 262>+≡
void breadth_first_traversal(
    const graph& G,
    node_array<int>& order)
{
    node v,w;
    forall_nodes(v,G)
        order[v] = -1;
```

```

int num = 1;
queue<node> Q;
forall_nodes(v,G) {
  if (order[v] ≡ -1) {
    Q.append(v);
    order[v] = num++; // visit vertex v
    while (¬Q.empty()) {
      v = Q.pop();
      forall_adj_nodes(w,v) {
        if (order[w] ≡ -1) {
          Q.append(w);
          order[w] = num++; // visit vertex w
        }
      }
    }
  }
}

⟨double-check breadth-first forest 282⟩
}

```

The following double-check of breadth-first traversal, although being redundant, gives some reassurance of the correctness of the implementation. It verifies that *order* is a breadth-first traversal of graph *G*, according to Definition 5.12.

282 ⟨double-check breadth-first forest 282⟩≡
{ node v,w,x,y;
edge e,ee;
node_array<node> root(G);
list<edge> S;

```

⟨find roots and arcs in breadth-first forest 283⟩

forall_edges(e,G) {
  v = G.source(e);
  w = G.target(e);
  if (order[root[v]] < order[root[w]]) {
    error_handler(1,
      "Wrong implementation of breadth-first traversal");
  }
}

forall(e,S) {
  v = G.source(e);
  w = G.target(e);
  forall_edges(ee,G) {
    x = G.source(ee);
    y = G.target(ee);
    if (order[x] < order[v] ∧
      order[v] < order[w] ∧
      order[w] ≤ order[y])
      error_handler(1,
        "Wrong implementation of breadth-first traversal");
  }
}

```

The root of the tree in the forest to which each vertex belongs is collected in an array $root$ of vertices indexed by the vertices of G , and the arcs in the breadth-first spanning forest of G are collected in a list S of arcs, as follows. The first-numbered vertex is the root of a tree in the forest, as are those vertices whose indegree is equal to zero or which has no incoming arcs from lower-numbered vertices. Those vertices v which are not the root of a tree in the forest, have an incoming tree arc going out of the lowest-numbered vertex w which is not numbered higher than vertex v itself.

283

(find roots and arcs in breadth-first forest 283)≡

```

{ int n = G.number_of_nodes();
  array<node> disorder(1,n);
  forall_nodes(v,G)
    disorder[order[v]] = v;

  root[disorder[1]] = disorder[1];
  for ( int i = 2; i ≤ n; i++ ) {
    v = disorder[i];
    if ( G.indeg(v) ≡ 0 ) {
      root[v] = v;
    } else {
      bool nonroot = false;
      forall_in_edges(e,v) {
        if ( order[G.source(e)] < order[v] ) {
          nonroot = true;
          break;
        }
      }
      if ( !nonroot ) {
        root[v] = v;
      } else {
        forall_in_edges(e,v)
          if ( order[G.source(e)] < order[v] )
            break;
        forall_in_edges(ee,v)
          if ( order[G.source(ee)] < order[G.source(e)] ∧
              order[G.source(ee)] < order[v] )
            e = ee;
        S.append(e);
        root[v] = root[G.source(e)];
      }
    }
  }
}

```

Lemma 5.20. *The algorithm for breadth-first traversal of a graph runs in $O(n + m)$ time using $O(m)$ additional space, where n is the number of vertices and m is the number of arcs in the graph.*

Proof. Let $G = (V, E)$ be a graph with n vertices and m arcs. Since each vertex of the graph is enqueued and dequeued at most once for

{double-check breadth-first tree 285a}

}

The following double-check of a breadth-first tree, although being redundant, gives some reassurance of the correctness of the implementation. It verifies that *order* is a breadth-first traversal of the breadth-first tree (W, S) of G , according to Definition 5.12.

Notice the possibility that $\text{order}[x] < \text{order}[v] < \text{order}[w] \leq \text{order}[y]$ for $(v, w) \in S$ and $(x, y) \in E$ with (W, S) still being a breadth-first tree of $G = (V, E)$, just because $x \notin W$ and then, $\text{order}[x] = -1$.

285a {double-check breadth-first tree 285a}≡
 { node v,w,x,y;
 edge e,ee;
 forall(e,S) {
 forall_edges(ee,G) {
 v = G.source(e);
 w = G.target(e);
 x = G.source(ee);
 y = G.target(ee);
 if (order[x] ≠ -1 ∧
 order[x] < order[v] ∧
 order[v] < order[w] ∧
 order[w] ≤ order[y])
 error_handler(1,
 "Wrong implementation of breadth-first traversal");
 } } }

5.2.1 Interactive Demonstration of Breadth-First Traversal

The algorithm for breadth-first traversal is integrated next in the interactive demonstration of graph algorithms. A simple checker for breadth-first traversal that provides some visual reassurance consists of numbering the vertices according to the order in which they are first visited during the traversal, highlighting also the vertices and arcs of the graph as the breadth-first traversal proceeds.

285b {demo graph traversal 267a}+≡
 void gw_breadth_first_traversal(
 GraphWin& gw)
 {
 graph& G = gw.get_graph();
 {put ordered graph in standard representation 269}
 gw.update_graph();

```

node_array<int> order(G);
breadth.first_traversal(G,order);
⟨show breadth-first graph traversal 286a⟩
}

```

A breadth-first graph traversal can also be demonstrated by highlighting each of the vertices of the graph in turn, according to the order defined by the traversal. A pause of half a second between vertices should suffice to get the picture of how the traversal proceeds.

286a ⟨show breadth-first graph traversal 286a⟩≡
 { gw.save_all_attributes();
 color gray = grey3;

```

node v,w,x;
forall_nodes(v,G) {
  gw.set_color(v,white);
  gw.set_label(v,string("%i",order[v]));
}

int n = G.number_of_nodes();
array<node> disorder(1,n);
forall_nodes(v,G)
  disorder[order[v]] = v;

edge e,ee;
for ( int i = 1; i ≤ n; i++ ) {
  v = disorder[i];

  ⟨highlight discovered vertex 268a⟩
  ⟨highlight incoming breadth-first tree arc 286b⟩
  ⟨highlight predecessor vertices 268c⟩

  leda_wait(0.5);
}

gw.wait();
gw.restore_all_attributes();
}
```

Those vertices v which are not the root of a tree in the breadth-first forest, have an incoming depth-first tree arc (w, v) going out of the lowest-numbered visited vertex w .

286b ⟨highlight incoming breadth-first tree arc 286b⟩≡
 { int pred_num = n;
 bool pred = false;
 forall_in_edges(e,v) {
 w = G.opposite(v,e);
 if (gw.get_color(w) ≠ white ∧

```

order[w] < pred_num ) {
ee = e;
pred_num = order[w];
pred = true;
}
if (pred) gw.set_width(ee,5);
leda_wait(0.5);
}
}

```

Now, the algorithm for traversing a breadth-first tree is also integrated next in the interactive demonstration of graph algorithms. A checker for breadth-first tree traversal that provides some visual reassurance consists of numbering the vertices according to the order in which they are first visited during the traversal, highlighting also the vertices and arcs of the breadth-first tree and producing a separate layered layout of the breadth-first tree as well.

287a

```

<demo graph traversal 267a)>+≡
void gw_breadth_first_spanning_subtree(
    GraphWin& gw)
{
    graph& G = gw.get_graph();
    (put ordered graph in standard representation 269)
    gw.update_graph();
    node v = gw.ask_node();
    node_array<int> order(G);
    set<node> W;
    set<edge> S;
    breadth_first_spanning_subtree(G,v,order,W,S);
    <show breadth-first spanning subtree 287b>
}

```

The layered layout of the breadth-first tree is also produced using the simple layered layout algorithm for rooted trees given in Sect. 3.5. Again, since the LEDA operations for hiding and restoring vertices and arcs would change the ordered structure of the graph, a rooted tree representation of the subgraph corresponding to the breadth-first tree is obtained instead by deleting from a copy of the whole graph, those vertices and arcs which do not belong to the breadth-first tree.

287b

```

<show breadth-first spanning subtree 287b)>≡
{ gw.save_all_attributes();
  color gray = grey3;
  forall_nodes(v,G)
    gw.set_label(v,"");
  forall(v,W) {

```

```

    gw.set_color(v,gray);
    gw.set_label(v,string("%i ",order[v]));
}
edge e;
forall(e,S)
    gw.set_width(e,3);

GRAPH<node,edge> H;
CopyGraph(H,G);

forall_nodes(v,H)
    if ( ~W.member(H[v]) )
        H.del_node(v);
    forall_edges(e,H)
        if ( ~S.member(H[e]) )
            H.del_edge(e);

tree T(H);
GraphWin gw2(T,500,500,"Breadth-First Spanning Subtree");
gw2.display();

forall_nodes(v,T)
    gw2.set_label(v,gw.get_label(H[v]));

node_array<double> xcoord(T);
node_array<double> ycoord(T);
layered_tree_layout(T,xcoord,ycoord);
gw2.adjust_coords_to_win(xcoord,ycoord);
gw2.set_layout(xcoord,ycoord);
gw2.display();

gw.wait();
gw.restore_all_attributes();
}

```

5.3 Applications

Depth-first and breadth-first graph traversal are the basis of several fundamental graph algorithms which the reader may be familiar with, including algorithms for finding strong components in a graph and connected components in an undirected graph, finding cycles, topological sorting of an acyclic graph, and finding shortest paths and minimum spanning trees.

A quite different application of graph traversal will be addressed here, related to the isomorphism problems on trees discussed in Chap. 4,

as an appetizer to Chap. 7. Recall that two ordered trees are isomorphic if there is a bijective correspondence between their node sets which preserves and reflects the structure of the ordered trees. In the same sense, two ordered graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic if there is a bijection $M \subseteq V_1 \times V_2$ which preserves and reflects the ordered structure of the graphs—that is, such that the ordered sequence of vertices adjacent to vertex $v \in V_1$ coincides (up to a cyclic rotation) with the ordered sequence of vertices adjacent to vertex $M[v] \in V_2$, for all vertices $v \in V_1$. In such a case, M is an **ordered graph isomorphism** of G_1 to G_2 .

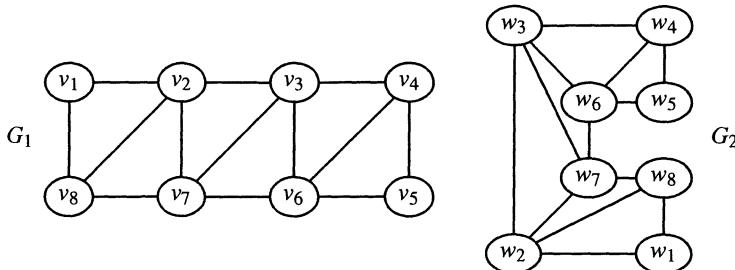


Fig. 5.9. Isomorphic ordered graphs. The relative order of the vertices adjacent to a given vertex reflects the counterclockwise ordering of the outgoing edges of the vertex in the drawing of each of the graphs.

Example 5.21. The ordered graphs of Fig. 5.9 are isomorphic, and $M = \{(v_1, w_1), (v_2, w_2), (v_3, w_3), (v_4, w_4), (v_5, w_5), (v_6, w_6), (v_7, w_7), (v_8, w_8)\}$ is an ordered graph isomorphism of G_1 to G_2 . As a matter of fact, the ordered sequence of vertices $[v_1, v_8, v_7, v_3]$ adjacent to vertex $v_2 \in V_1$ coincides, up to a cyclic rotation, with the ordered sequence $[w_7, w_3, w_1, w_8]$ of vertices adjacent to vertex $M[v_2] = w_2 \in V_2$, and the same holds for all of the vertices in the graphs.

A straightforward procedure for testing isomorphism of two connected, ordered undirected graphs consists of performing a leftmost depth-traversal of the bidirected graphs underlying both undirected graphs, and then testing whether the vertex mapping induced by the traversal is an isomorphism between the ordered undirected graphs. Since a leftmost depth-first traversal of a bidirected graph is only

unique for a given initial arc, though, the leftmost depth-first traversal starting in turn with each of the arcs of one graph, has to be tested against the leftmost depth-first traversal of the other graph starting with some (fixed) arc.

Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be ordered bidirected graphs with n vertices, let $\text{order}_1 : V_1 \rightarrow \{1, \dots, n\}$ be a leftmost depth-first traversal of G_1 , and let $\text{order}_2 : V_2 \rightarrow \{1, \dots, n\}$ be a leftmost depth-first traversal of G_2 . The vertex mapping $M \subseteq V_1 \times V_2$ induced by order_1 and order_2 is given by $M[v] = w$ if and only if $\text{order}_1[v] = \text{order}_2[w]$, for all vertices $v \in V_1$ and $w \in V_2$. Since both order_1 and order_2 are bijections, M is also a bijection.

The following algorithm implements the previous procedure for testing isomorphism of two ordered undirected graphs, based on a series of leftmost depth-first traversals of the underlying bidirected graphs. The bijection $M \subseteq V_1 \times V_2$ computed by the procedure upon two isomorphic ordered undirected graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is represented by an array of vertices (of graph G_2) indexed by the vertices of graph G_1 .

290

```

⟨graph traversal 262⟩ +≡
bool ordered_graph_isomorphism(
    const graph& G1,
    const graph& G2,
    node_array<node>& M)
{
    if ( G1.number_of_nodes() ≠ G2.number_of_nodes() ∨
        G1.number_of_edges() ≠ G2.number_of_edges() )
        return false;

    if ( G1.number_of_edges() ≡ 0 ) {
        node v = G1.first_node();
        node w = G2.first_node();
        while ( v ≠ nil ) { // && w != nil
            M[v] = w;
            v = G1.succ_node(v);
            w = G2.succ_node(w);
        }
        return true;
    }

    edge e1 = G1.first_edge();
    list<edge> L;
    leftmost_depth-first_traversal(G1,e1,L);
}

```

⟨obtain order in which vertices of G1 were visited 292a⟩

```

edge e2;
forall_edges(e2,G2) {
  L.clear();
  leftmost_depth-first_traversal(G2,e2,L);

  ⟨obtain order in which vertices of G2 were visited 292b⟩
  ⟨build ordered graph isomorphism mapping 291a⟩
  ⟨test mapping for ordered graph isomorphism 291b⟩

  if (isomorph) return true;
}
return false;
}

```

An actual ordered graph isomorphism mapping $M \subseteq V_1 \times V_2$ of graph $G_1 = (V_1, E_1)$ to graph $G_2 = (V_2, E_2)$ can be constructed by mapping the i th-visited vertex $v \in V_1$ during the leftmost depth-first traversal of G_1 to the i th-visited vertex $w \in V_2$ during the leftmost depth-first traversal of G_2 , for $1 \leq i \leq n$.

291a ⟨build ordered graph isomorphism mapping 291a⟩≡

```

{ int n = G1.number_of_nodes();
  array<node> disorder1(1,n);
  node v;
  forall_nodes(v,G1)
    disorder1[order1[v]] = v;
  array<node> disorder2(1,n);
  forall_nodes(v,G2)
    disorder2[order2[v]] = v;
  for (int i = 1; i <= n; i++) {
    M[disorder1[i]] = disorder2[i];
  }
}

```

Testing a candidate vertex mapping $M \subseteq V_1 \times V_2$ for ordered graph isomorphism of $G_1 = (V_1, E_1)$ to $G_2 = (V_2, E_2)$ amounts to testing whether the ordered sequence L_1 of vertices adjacent to vertex $v \in V_1$ coincides, up to a cyclic rotation, with the ordered sequence L_2 of vertices adjacent to vertex $M[v] \in V_2$, for all vertices $v \in V_1$.

291b ⟨test mapping for ordered graph isomorphism 291b⟩≡

```

bool isomorph = true;
{ node v,w;
  list<node> L1,L2;
  forall_nodes(v,G1) {
    L1 = G1.adj_nodes(v);
    L2 = G2.adj_nodes(M[v]);
    w = M[L1.front()];
    for (int i = 1; i < L1.length(); i++) {

```

```

if (  $w \equiv L2.front()$  ) {
    break;
} else {
     $L2.move\_to\_rear(L2.first());$ 
} }

list.item  $it1 = L1.first();$ 
list.item  $it2 = L2.first();$ 
while (  $it1 \neq \text{nil}$  ) { // &&  $it2 \neq \text{nil}$ 
    if (  $M[L1.contents(it1)] \neq L2.contents(it2)$  ) {
         $isomorph = \text{false};$ 
        break;
    }
     $it1 = L1.succ(it1);$ 
     $it2 = L2.succ(it2);$ 
} } }

```

The order in which the vertices of G_1 and G_2 are visited is just the order in which they are first reached during the leftmost depth-first traversal of G_1 and G_2 , respectively.

292a ⟨obtain order in which vertices of G_1 were visited 292a⟩≡
 $node_array<\text{int}> order1(G1);$
 $\{ node v;$
 $forall_nodes(v,G1)$
 $order1[v] = -1;$
 $edge e = L.head();$
 $int num = 1;$
 $order1[G1.source(e)] = num++;$
 $forall(e,L) \{$
 $if (order1[G1.target(e)] \equiv -1) \{$
 $order1[G1.target(e)] = num++;$
} } }

292b ⟨obtain order in which vertices of G_2 were visited 292b⟩≡
 $node_array<\text{int}> order2(G2);$
 $\{ node v;$
 $forall_nodes(v,G2)$
 $order2[v] = -1;$
 $edge e = L.head();$
 $int num = 1;$
 $order2[G2.source(e)] = num++;$
 $forall(e,L) \{$
 $if (order2[G2.target(e)] \equiv -1) \{$
 $order2[G2.target(e)] = num++;$
} } }

No double-check of ordered graph isomorphism is needed, because the mapping $M \subseteq V_1 \times V_2$ was already tested for ordered graph isomorphism of $G_1 = (V_1, E_1)$ to $G_2 = (V_2, E_2)$.

Lemma 5.22. *The algorithm for ordered graph isomorphism runs in $O((n+m)m)$ time using $O(n+m)$ additional space, where n is the number of vertices and m is the number of arcs in the ordered graphs.*

Proof. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be ordered graphs with n vertices and m arcs. Since one leftmost depth-first traversal of G_1 and m leftmost depth-first traversals of G_2 are performed, and each leftmost depth-first traversal, as well as the test of a mapping for ordered graph isomorphism, takes $O(n+m)$ time, the algorithm runs in $O((n+m)m)$ time. Further, since only the leftmost depth-first traversal of G_1 and one leftmost depth-first traversal of G_2 are kept at the same time, the algorithm uses $O(n+m)$ additional space.

The algorithm for ordered graph isomorphism is integrated next in the interactive demonstration of graph algorithms. A simple checker for isomorphic graphs that provides some visual reassurance consists of redrawing one of the graphs according to the graph isomorphism found, to match the layout of the other graph.

293

```
<demo graph traversal 267a>+≡
void gw_ordered_graph_isomorphism(
    GraphWin& gw1)
{
    graph& G1 = gw1.get_graph();

    panel P;
    if ( !Is_Bidirected(G1) ) {
        make_proof_panel(P,"This graph is \\\red not bidirected",false);
        gw1.open_panel(P);
        return;
    }

    G1.make_map(); // set edge reversal information
    node_array<node> M(G1,nil);

    graph G2;
    GraphWin gw2(G2,500,500,"Ordered Graph Isomorphism");
    gw2.display();
    gw2.message("Enter second graph. Press done when finished");
    gw2.edit();
    gw2.del_message();

    if ( !Is_Bidirected(G2) ) {
        make_proof_panel(P,"This graph is \\\red not bidirected",false);
        gw2.open_panel(P);
        return;
    }
```

```

}

G2.make_map(); // set edge reversal information

if ( ordered_graph_isomorphism(G1,G2,M) ) {

    make_proof_panel(P,"These graphs are isomorphic",true);
    if ( gw1.open_panel(P) ) { // proof button pressed
        gw1.save_all_attributes();
        node_array<point> pos1(G1);
        node_array<point> pos2(G2);
        ⟨show graph isomorphism 363⟩
        gw1.wait();
        gw1.restore_all_attributes();
    }

} else {

    make_proof_panel(P,"These graphs are \\\red not isomorphic",false);
    gw1.open_panel(P);

}
}

```

Summary

The most common methods of exploring a graph were addressed in this chapter. Simple algorithms are given in detail for two different methods of graph traversal: depth-first traversal and breadth-first traversal. While the former generalizes the depth-first prefix leftmost (preorder) traversal of a rooted tree, the latter is a generalization of breadth-first leftmost (top-down) tree traversal. A particular case of the depth-first traversal of an undirected graph, called leftmost depth-first traversal, is also discussed and detailed algorithms tailored to this particular case are also given. The application of leftmost depth-first graph traversal to the isomorphism of ordered graphs is also discussed in detail.

Bibliographic Notes

Depth-first graph traversal was first described in [317], and is the basis of many fundamental graph algorithms. Breadth-first graph traversal

was first described in [208, 241]. The recognition of depth-first and breadth-first spanning trees was addressed respectively in [200, 258, 270, 285] and in [223].

The interactive demonstration of depth-first traversal is based on [83, Sect. 22.3] and also on [236, Sect. 7.3]. The interactive demonstration of breadth-first traversal is based on [83, Sect. 22.2].

The algorithm for leftmost depth-first traversal of a bidirected graph is based on the maze traversal algorithm found by Trémaux and recalled in [362]. An algorithm for the leftmost depth-first traversal of an undirected graph is included in LEDA [236, Sect. 8.7.2]. The algorithm for isomorphism of ordered graphs is based on [170]. See also [166, 286, 334].

Review Problems

5.1 Give the depth-first tree rooted in turn at each of the vertices of the graph in Fig. 5.10. Explain why the depth-first tree rooted at vertex v_2 differs from the tree rooted at the same vertex v_2 in the depth-first forest of the graph.

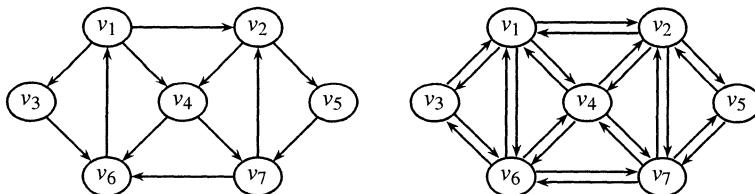


Fig. 5.10. Graph (to the left) and bidirected graph (to the right) for problems 5.1–5.5. The relative order of the vertices adjacent to a given vertex corresponds to the counterclockwise ordering of the outgoing edges of the vertex in the drawing of the graph.

5.2 In the depth-first traversal of a graph $G = (V, E)$ with n vertices, the set of arcs E can be partitioned into four classes, according to order : $V \rightarrow \{1, \dots, n\}$, the order in which their source and target vertices are first visited and also according to comp : $V \rightarrow \{1, \dots, n\}$, the order in which processing of their source and target vertices is completed or finished, where a vertex is completed as soon as all its adjacent vertices have been visited. An arc (v, w) is called a *tree arc* if it

belongs to the depth-first forest of the graph, and it is called a *forward arc* if it is parallel to a path of tree arcs. In both cases, it holds that $\text{order}[v] < \text{order}[w]$ and $\text{comp}[v] > \text{comp}[w]$. An arc (v, w) is called a *backward arc* if it is antiparallel to a path of tree arcs and in this case, $\text{order}[v] \geq \text{order}[w]$ and $\text{comp}[v] \leq \text{comp}[w]$. Finally, an arc (v, w) is called a *cross arc* if it is neither a tree, nor a forward, nor a backward arc, and in this case, $\text{order}[v] > \text{order}[w]$ and $\text{comp}[v] > \text{comp}[w]$. Give the classification of the arcs in the depth-first traversal of the graph in Fig. 5.10.

5.3 Give the leftmost depth-first traversal of the bidirected graph in Fig. 5.10, starting in turn at each of the arcs of the graph. Explain whether there is any relationship holding among all the leftmost depth-first traversals of a bidirected graph.

5.4 Give the breadth-first tree rooted in turn at each of the vertices of the graph in Fig. 5.10. Explain why the breadth-first tree rooted at vertex v_2 differs from the tree rooted at the same vertex v_2 in the breadth-first forest of the graph.

5.5 The same classification of the arcs of a graph explained in Problem 5.2 applies to breadth-first traversal, but there are no forward arcs in the breadth-first traversal of a graph. Give the classification of the arcs in the breadth-first traversal of the graph in Fig. 5.10.

5.6 Determine whether the ordered undirected graphs in Fig. 5.11 are isomorphic and, in this case, give an ordered graph isomorphism of G_1 to G_2 .

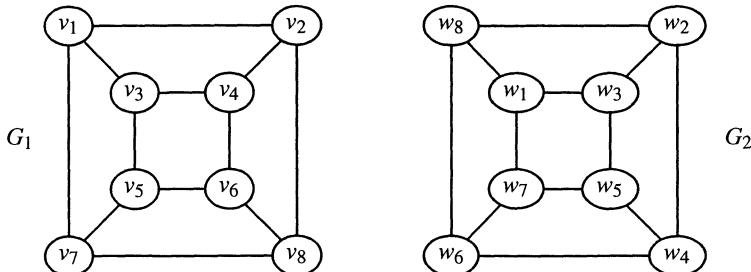


Fig. 5.11. Ordered undirected graphs for problem 5.6. The relative order of the vertices adjacent to a given vertex corresponds to the counterclockwise ordering of the outgoing edges of the vertex in each of the graphs.

Exercises

- 5.1** The depth-first tree and the breadth-first tree rooted at vertex v_2 of the graph in Fig. 5.10 are identical, although the depth-first forest and the breadth-first forest of the graph are different. Characterize those graphs whose depth-first forest and breadth-first forest are identical.
- 5.2** Extend the depth-first traversal algorithm to compute the completion order of all the vertices in a graph. Extend also the interactive demonstration of depth-first traversal to distinguish among tree, forward, backward, and cross arcs.
- 5.3** Modify the depth-first traversal algorithm to operate on the adjacency matrix representation of a graph. Give the time and space complexity of the modified algorithm.
- 5.4** When rearranging the LEDA representation of a graph in order to meet Assumption 5.1 about the order of vertices and arcs, the relative order of the vertices adjacent to a given vertex may no longer correspond to the counterclockwise ordering of the outgoing arcs of the vertex in the layout of the graph. Give an algorithm to rearrange the layout of the graph as well, in order to reflect the standard representation of the graph.
- 5.5** Give an algorithm to compute the bidirected graph underlying a graph, in such a way that the ordered structure of the given graph is preserved. For instance, for the graph in Fig. 5.10, the cyclic sequence of vertices adjacent to vertex v_1 is $[v_2, v_3, v_4]$ in the graph but $[v_2, v_3, v_6, v_4]$ in the underlying bidirected graph.
- 5.6** Modify the breadth-first traversal algorithm to find a shortest path in a graph from a given vertex to every other vertex reachable in the graph from the given vertex.
- 5.7** Extend the breadth-first traversal algorithm to compute the completion order of all the vertices in a graph. Extend also the interactive demonstration of breadth-first traversal to distinguish among tree, backward, and cross arcs.
- 5.8** Modify the breadth-first traversal algorithm to operate on the adjacency matrix representation of a graph. Give the time and space complexity of the modified algorithm.

5.9 Modify the ordered graph isomorphism algorithm given in Sect. 5.3 to compare isomorphism codes, where the isomorphism code of a connected, ordered undirected graph $G = (V, E)$ with m edges is a sequence of $2m$ vertices in the order in which they are visited during a leftmost depth-first traversal of the underlying bidirected graph, and equality of isomorphism codes holds up to a cyclic rotation of the sequences. Give also the time and space complexity of the modified algorithm.

5.10 The ordered graph isomorphism algorithm given in Sect. 5.3 is based on performing a leftmost depth-first traversal of one of the graphs and a series of leftmost depth-first traversals of the other graph, testing in each case the induced vertex mapping for ordered graph isomorphism. Nonisomorphism of two connected, ordered undirected graphs could be detected earlier, though, by performing a series of simultaneous leftmost depth-first traversals of the two graphs, in much the same way as done in Chap. 4 for solving isomorphism problems on ordered trees. Give an algorithm for ordered graph isomorphism based on simultaneous leftmost depth-first traversals of the graphs. Give also the time and space complexity of the algorithm.

6. Clique, Independent Set, and Vertex Cover

If you really need to find the largest clique in a graph, an exhaustive search via backtracking provides the only real solution.

—Steven S. Skiena [304]

Finding complete subgraphs of an undirected graph is another fundamental problem with a variety of interesting applications in diverse scientific and engineering disciplines. Clique algorithms also find application in the comparison of structures described by trees or graphs and, as a matter of fact, finding maximal and maximum cliques is the basis of the algorithms discussed in Chap. 7 for finding maximal and maximum common subgraph isomorphisms between two graphs.

Combinatorial algorithms are discussed in this chapter for enumerating all maximal cliques, and finding a maximum clique in an undirected graph. Further, the dual problem of finding a maximum independent set in an undirected graph is solved by finding a maximum clique in the complement of the graph, and a simple algorithm is given for finding a maximum independent set in a tree. These algorithms use the techniques given in Chap. 2, and the latter also builds upon the algorithms discussed in Chap. 3 for traversing trees. The algorithms for finding a maximum independent set are also used for solving the related problem of finding a minimum vertex cover, in both trees and undirected graphs.

6.1 Cliques, Maximal Cliques, and Maximum Cliques

A clique of an undirected graph is a complete subgraph of the graph. A maximal clique of a graph is a clique of the graph that is not properly

included in any other clique of the graph, and a maximum clique of a graph is a (maximal) clique of the graph with the largest number of vertices.

Recall from Sect. 1.1 that an undirected graph is a complete graph if every pair of distinct vertices in the graph is joined by an edge.

Definition 6.1. A *clique* of an undirected graph $G = (V, E)$ is a set of vertices $C \subseteq V$ such that the subgraph of G induced by C is complete, that is, such that $\{v, w\} \in E$ for all distinct vertices $v, w \in C$. A clique C of an undirected graph $G = (V, E)$ is **maximal** if there is no clique D of G such that $C \subseteq D$ and $C \neq D$, and it is **maximum** if there is no clique of G with more vertices than C . The **clique number** of G , denoted by $\omega(G)$, is the cardinality of a maximum clique of G .

It follows from the previous definition that maximum cliques are also maximal.

Lemma 6.2. Every maximum clique $C \subseteq V$ of an undirected graph $G = (V, E)$ is maximal.

Proof. Let $C \subseteq V$ be a maximum clique of an undirected graph $G = (V, E)$, and suppose C is not maximal. There is, by Definition 6.1, a clique $D \subseteq V$ such that $C \subseteq D$ and $C \neq D$. Then, $D \setminus C \neq \emptyset$, that is, clique D has more vertices than clique C , contradicting the hypothesis that C is a maximum clique. Therefore, C is a maximal clique. \square

Example 6.3. The undirected graph shown in Fig. 6.1 has three maximal cliques: $\{v_1, v_2, v_4, v_7\}$, $\{v_2, v_4, v_5, v_7\}$, and $\{v_1, v_3, v_4, v_6, v_7\}$. The latter is also the only maximum clique of the graph.

Notice that being a clique of an undirected graph is a hereditary graph property, that is, a property shared by all those subgraphs of the given graph which are induced by some subset of the given clique.

Lemma 6.4. Let $C \subseteq V$ be a clique of an undirected graph $G = (V, E)$, and let $B \subseteq C$. Then, B is also a clique of G .

Proof. Let $C \subseteq V$ be a clique of an undirected graph $G = (V, E)$, and let $B \subseteq C$. Then, $\{v, w\} \in E$ for all distinct vertices $v, w \in C$ by Definition 6.1 and, in particular, $\{v, w\} \in E$ for all distinct vertices $v, w \in B$. Therefore, $B \subseteq V$ is also a clique. \square

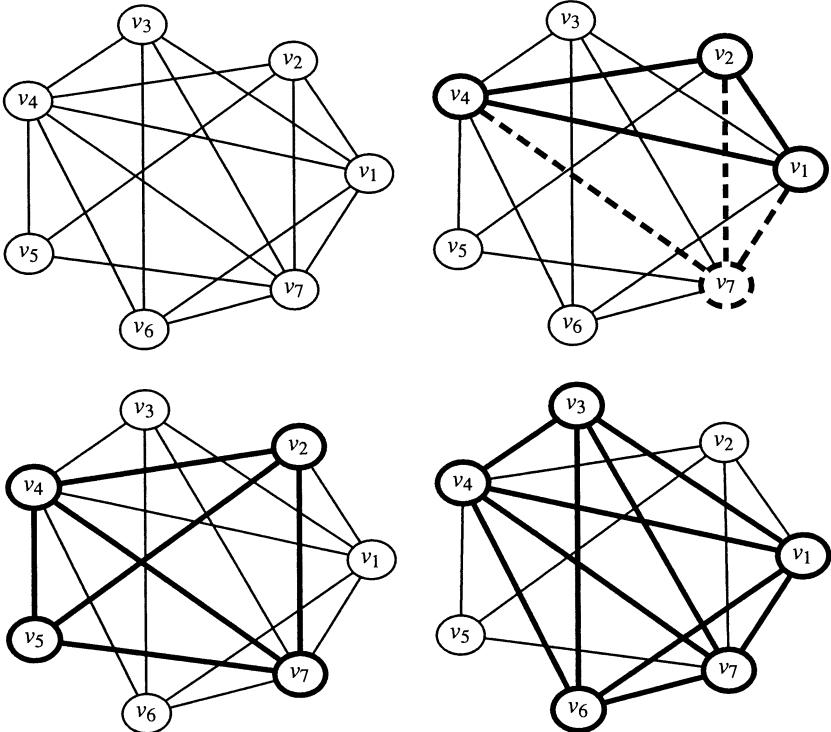


Fig. 6.1. Maximal and maximum cliques of an undirected graph. The clique $\{v_1, v_2, v_4\}$ is not maximal, because it is included in clique $\{v_1, v_2, v_4, v_7\}$. The cliques $\{v_1, v_2, v_4, v_7\}$, $\{v_1, v_3, v_4, v_6, v_7\}$, and $\{v_2, v_4, v_5, v_7\}$ are all maximal. Clique $\{v_1, v_3, v_4, v_6, v_7\}$ is also a maximum clique.

Maximal Cliques of a Graph

Finding all maximal cliques of an undirected graph is an NP-hard problem, because the maximum number of maximal cliques of an undirected graph is exponential in the number of vertices of the graph.

Consider first the problem of finding all cliques of an undirected graph. A clique of an undirected graph can be extended to a larger clique if there is a vertex not in the clique which is adjacent to all of the vertices in the clique.

A simple procedure to extend a clique $C \subseteq V$ of an undirected graph $G = (V, E)$ in all possible ways is the following. Let $P \subseteq V \setminus C$ be the set of *candidate* vertices which can be used to extend C to a larger clique, that is, $P = \{w \in V \setminus C \mid \{v, w\} \in E \text{ for all } v \in C\}$. Then,

all possible extensions of C with vertices from P can be obtained by taking each candidate vertex of P in turn and removing it from P , adding it to C , creating a new set P from the old set by removing those vertices which are not adjacent to the selected candidate vertex, recursively performing the procedure upon the new sets C and P , and then removing the selected candidate vertex from C .

In order to avoid obtaining duplicate cliques, that is, to avoid obtaining all permutations of the vertices of a clique C , though, the set of candidate vertices $P \subseteq V \setminus C$ is considered to include only those vertices which are adjacent to all of the vertices in C and which are greater than the vertices in C , according to the order on the vertices fixed by the representation of the graph. That is, $P = \{w \in V \setminus C \mid \text{order}[v] < \text{order}[w], \{v, w\} \in E \text{ for all } v \in C\}$, where $\text{order} : V \rightarrow \{1, \dots, n\}$ is the order on V fixed by the representation of an undirected graph $G = (V, E)$ with n vertices.

The extension of a clique C to a larger clique by adding a candidate vertex $v \in P$ is illustrated in Fig. 6.2. The new set P of candidate vertices, shown encircled with dashed lines, contains those vertices $w \in P$ which are adjacent to vertex v .

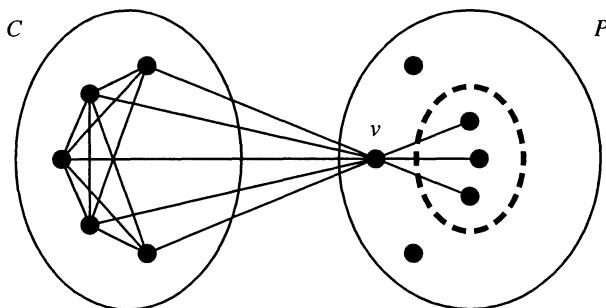


Fig. 6.2. Extending a clique C to a larger clique by adding a candidate vertex $v \in P$. The new set of candidate vertices contains those vertices $w \in P$ which are adjacent to vertex v . Edges joining vertices in C with vertices in $P \setminus \{v\}$ are omitted, for clarity.

Now, the following backtracking algorithm for enumerating cliques extends an (initially empty) clique C in all possible ways, in order to enumerate all cliques $C \subseteq V$ of an undirected graph $G = (V, E)$. The cliques found are collected in a list L of sets of vertices, and the adj-

cency matrix A of graph G will be used for double-checking each of the cliques found.

303a ⟨maximal clique 303a⟩≡

```

void all_cliques(
    const graph& G,
    const node_matrix<bool>& A,
    list<set<node>>& L)
{
    set<node> C;
    set<node> P;
    node v;
    forall_nodes(v,G)
        P.insert(v);
    next_clique(G,A,C,P,L);
}

```

The actual extension of a clique is done by the following recursive procedure, which extends a clique $C \subseteq V$ of the undirected graph $G = (V, E)$ by adding to C each vertex $w \in V \setminus C$ in turn which is adjacent to all those vertices which are already in C .

For each extension of clique C by adding a vertex $v \in P$, a new set PP of candidate vertices is created containing only those vertices $w \in P$ which are adjacent to vertex v , instead of removing from a copy of set P those vertices which are not adjacent to vertex v .

Notice that every vertex and every pair of adjacent vertices constitute a (trivial) clique of an undirected graph. All nontrivial cliques are collected in a list L of sets of vertices.

303b ⟨maximal clique 303a⟩+≡

```

void next_clique(
    const graph& G,
    const node_matrix<bool>& A,
    set<node>& C,
    set<node>& P,
    list<set<node>>& L)
{
    if (C.size() > 2) { // ignore trivial cliques
        double_check_clique(G,A,C);
        L.append(C);
    }

    if (¬P.empty()) {
        node v,w;
        forall(v,P) {
            P.del(v);
            set<node> PP;
            forall_adj_nodes(w,v)

```

```

if ( P.member(w) ) PP.insert(w);
C.insert(v);
next_clique(G,A,C,PP,L);
C.del(v);
} } }

```

The following double-check of the clique, although being redundant, gives some reassurance of the correctness of the implementation. It verifies that the vertices of the set C induce a clique of graph G , by double-checking that every vertex of C is adjacent in G to every other vertex of C .

304 \langle double-check clique 304 $\rangle \equiv$

```

void double_check_clique(
    const graph& G,
    const node_matrix<bool>& A,
    const set<node>& C)
{
    node v,w;
    forall(v,C)
        forall(w,C)
            if ( v  $\neq$  w  $\wedge$  (  $\neg A(v,w) \vee \neg A(w,v)$  ) )
                error_handler(1,"Wrong implementation of clique");
}

```

Example 6.5. Consider the execution of the clique enumeration procedure illustrated in Fig. 6.3 and also in Fig. 6.4. The cliques of the given undirected graph are those for which the set of candidate vertices P with which they can be extended is empty. Only three out of these 24 cliques are maximal, namely: $\{v_1, v_2, v_4, v_7\}$, $\{v_1, v_3, v_4, v_6, v_7\}$, and $\{v_2, v_4, v_5, v_7\}$.

Remark 6.6. Correctness of the backtracking algorithm for finding all cliques follows from the fact that an (initially empty) clique C of an undirected graph is extended in all possible ways, by adding one vertex from P at a time, until it cannot be further extended because P is empty.

Consider now the problem of finding all maximal cliques of an undirected graph. A clique of an undirected graph cannot be extended to a larger clique unless there is a vertex which is not in the clique and is adjacent to all of the vertices in the clique.

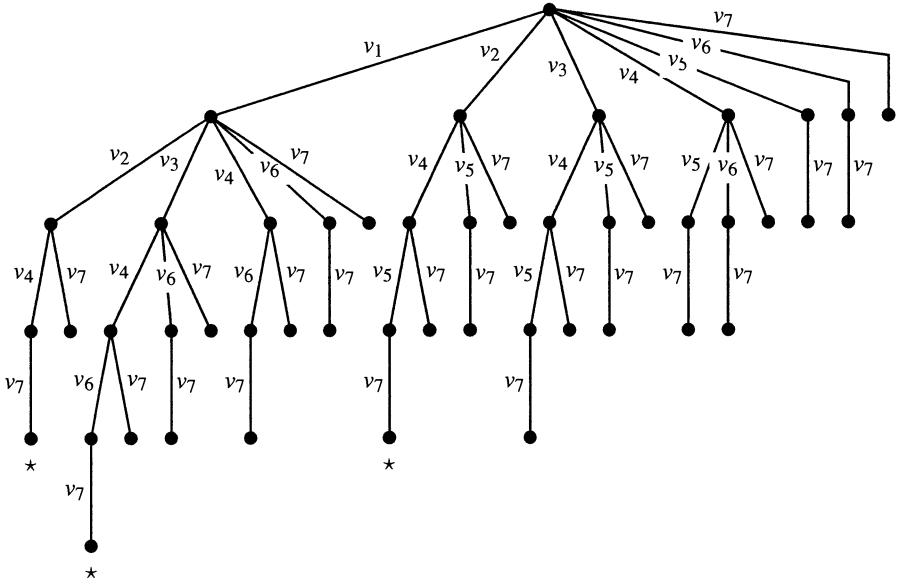


Fig. 6.3. Search tree for the execution of the clique enumeration procedure, upon the undirected graph of Fig. 6.1. Every edge of the tree represents an extension of the current clique by adding one vertex, and the edge is labeled by the vertex added to the current clique. Every path in the tree from the root to some node represents a clique of the graph. Those leaves of the tree representing a maximal clique are marked by a star.

Lemma 6.7. A clique $C \subseteq V$ of an undirected graph $G = (V, E)$ is maximal if and only if there is no vertex $w \in V \setminus C$ such that $\{v, w\} \in E$ for all vertices $v \in C$.

Proof. Let $C \subseteq V$ be a maximal clique of an undirected graph $G = (V, E)$. Then, by Definition 6.1, $\{u, v\} \in E$ for all distinct vertices $u, v \in C$. Suppose now that there is a vertex $w \in V \setminus C$ such that $\{v, w\} \in E$, for all vertices $v \in C$. Then, $\{u, v\} \in E$ for all distinct vertices $u, v \in C \cup \{w\}$, that is, $C \cup \{w\}$ is a clique, contradicting the hypothesis that C is maximal. Therefore, there is no vertex $w \in V \setminus C$ such that $\{v, w\} \in E$ for all vertices $v \in C$.

Now, let $C \subseteq V$ be a clique of an undirected graph $G = (V, E)$ such that there is no vertex $w \in V \setminus C$ with $\{v, w\} \in E$ for all vertices $v \in C$, and let $D \subseteq V \setminus C$ be a set of vertices such that $C \cup D$ is a maximal clique. Then, by Definition 6.1, $\{v, w\} \in E$ for all distinct vertices $v, w \in C \cup D$ and in particular, for all vertices $v \in C$ and $w \in D$, contra-

C	P
	$v_1 v_2 v_3 v_4 v_5 v_6 v_7$
v_1	$v_2 v_3 v_4 v_6 v_7$
$v_1 v_2$	$v_4 v_7$
$v_1 v_2 v_4$	v_7
$v_1 v_2 v_4 v_7$	
$v_1 v_2 v_7$	
$v_1 v_3$	$v_4 v_6 v_7$
$v_1 v_3 v_4$	$v_6 v_7$
$v_1 v_3 v_4 v_6$	v_7
$v_1 v_3 v_4 v_6 v_7$	
$v_1 v_3 v_4 v_7$	
$v_1 v_3 v_6$	v_7
$v_1 v_3 v_6 v_7$	
$v_1 v_3 v_7$	
$v_1 v_4$	$v_6 v_7$
$v_1 v_4 v_6$	v_7
$v_1 v_4 v_6 v_7$	
$v_1 v_4 v_7$	
$v_1 v_6$	v_7
$v_1 v_6 v_7$	
$v_1 v_7$	
v_2	$v_4 v_5 v_7$
$v_2 v_4$	$v_5 v_7$
$v_2 v_4 v_5$	v_7

Fig. 6.4. Execution of the clique enumeration procedure, upon the undirected graph of Fig. 6.1. For each clique C , the corresponding set P contains those vertices which can be used to extend C to a larger clique.

dicting the hypothesis that there is no vertex $w \in V \setminus C$ with $\{v, w\} \in E$ for all vertices $v \in C$. Therefore, C is maximal. \square

In the backtracking algorithm for enumerating cliques, a necessary condition for a clique C to be maximal is that P be empty, for otherwise C can still be extended to a larger clique by adding a vertex from P . Such a necessary condition is not sufficient for C to be a maximal clique, though, because a clique C may be contained in some already obtained clique and still P be empty.

Therefore, in order to extend the backtracking algorithm to the enumeration of maximal cliques, the already obtained cliques need to be recognized. The set S of those vertices such that all extensions of clique C with vertices from S were already obtained, allows the efficient recognition of already obtained cliques.

Let $P \subseteq V \setminus C$ be, again, the set of candidate vertices which can be used to extend C to a larger clique, and let now $S \subseteq V \setminus C \setminus P$ be the set of vertices which cannot be used to extend C to a larger clique, because all cliques D with $C \subseteq D \subseteq C \cup S$ were already obtained.

Then, all possible extensions of C with vertices from P and without vertices from S can be obtained by taking each candidate vertex of P in turn and removing it from P , adding it to C , creating new sets P and S from the old sets by removing those vertices which are not adjacent to the selected candidate vertex, recursively performing the procedure upon the new sets C , P , and S , and then removing the selected candidate vertex from C and adding it to S . Throughout this procedure, a clique C is maximal if both P and S are empty.

Lemma 6.8. *A clique C is maximal if and only if both P and S are empty.*

Proof. Let C be a maximal clique, and suppose $P \neq \emptyset$. Then, there is a vertex $w \in P$ such that $\{v, w\} \in E$, for all vertices $v \in C$, that is, $C \cup \{w\}$ is a clique. Further, since $P \subseteq V \setminus C$, it follows that $C \neq C \cup \{w\}$, contradicting the hypothesis that clique C is maximal. Therefore, $P = \emptyset$.

Suppose now $S \neq \emptyset$. Then, there is a vertex $w \in S$ such that clique $C \cup \{w\}$ was already obtained, contradicting again the hypothesis that C is maximal. Therefore, $S = \emptyset$.

Let now C be a clique such that both $P = \emptyset$ and $S = \emptyset$, and suppose C is not maximal. Then, there is a clique D such that $C \subseteq D$ and $C \neq D$. Let $w \in D \setminus C$. Since D is a clique, $\{v, w\} \in E$ for all vertices $v \in D$ with $v \neq w$ and, in particular, $\{v, w\} \in E$ for all vertices $v \in C$, that is, $w \in P$, contradicting the hypothesis that $P = \emptyset$. Therefore, C is maximal. \square

Now, the following simple backtracking algorithm for enumerating maximal cliques extends an (initially empty) clique C in all possible ways, in order to enumerate all maximal cliques $C \subseteq V$ of an undirected graph $G = (V, E)$. The maximal cliques found are collected in a list L of sets of vertices, and the adjacency matrix A of graph G will be used for double-checking each of the maximal cliques found.

```

const node_matrix<bool>& A,
list<set<node>>& L)
{
    set<node> C;
    set<node> P;
    set<node> S;
    node v;
    forall_nodes(v,G)
        P.insert(v);
    simple_next_maximal_clique(G,A,C,P,S,L);
}

```

The actual extension of a clique is done by the following recursive procedure, which extends a clique $C \subseteq V$ of the undirected graph $G = (V, E)$ by adding to C each vertex $w \in V \setminus C$ in turn which is adjacent to all those vertices which are already in C .

For each extension of clique C by adding a vertex $v \in P$, a new set PP of candidate vertices is created containing only those vertices $w \in P$ which are adjacent to vertex v , and a new set SS of noncandidate vertices is created containing only those vertices $w \in S$ which are adjacent to vertex v .

308

```

⟨maximal clique 303a⟩ +≡
void simple_next_maximal_clique(
    const graph& G,
    const node_matrix<bool>& A,
    set<node>& C,
    set<node>& P,
    set<node>& S,
    list<set<node>>& L)
{
    if ( P.empty() & S.empty() ) {
        double_check_maximal_clique(G,A,C);
        L.append(C);
    } else {
        node v,w;
        forall(v,P) {
            P.del(v);
            set<node> PP;
            set<node> SS;
            forall_adj_nodes(w,v) {
                if ( P.member(w) ) PP.insert(w);
                if ( S.member(w) ) SS.insert(w);
            }
            C.insert(v);
        simple_next_maximal_clique(G,A,C,PP,SS,L);
        C.del(v);
        S.insert(v);
    } } }
}
```

C	P	S	C	P	S
	$v_1 v_2 v_3 v_4 v_5 v_6 v_7$			$v_2 v_4 v_5 v_7$	
v_1	$v_2 v_3 v_4 v_6 v_7$			$v_2 v_4 v_7$	$v_1 v_5$
$v_1 v_2$	$v_4 v_7$			$v_2 v_5$	v_7
$v_1 v_2 v_4$	v_7			$v_2 v_5 v_7$	v_4
$v_1 v_2 v_4 v_7$				$v_2 v_7$	$v_1 v_4 v_5$
$v_1 v_2 v_7$	v_4			v_3	$v_4 v_6 v_7$
$v_1 v_3$	$v_4 v_6 v_7$			$v_3 v_4$	$v_6 v_7$
$v_1 v_3 v_4$	$v_6 v_7$			$v_3 v_4 v_6$	v_7
$v_1 v_3 v_4 v_6$	v_7			$v_3 v_4 v_6 v_7$	v_1
$v_1 v_3 v_4 v_6 v_7$				$v_3 v_4 v_7$	$v_1 v_6$
$v_1 v_3 v_4 v_7$	v_6			$v_3 v_6$	v_7
$v_1 v_3 v_6$	v_7	v_4		$v_3 v_6 v_7$	$v_1 v_4$
$v_1 v_3 v_6 v_7$		v_4		$v_3 v_7$	$v_1 v_4 v_6$
$v_1 v_3 v_7$		$v_4 v_6$		v_4	$v_5 v_6 v_7$
$v_1 v_4$	$v_6 v_7$	$v_2 v_3$		$v_4 v_5$	v_7
$v_1 v_4 v_6$	v_7	v_3		$v_4 v_5 v_7$	v_2
$v_1 v_4 v_6 v_7$		v_3		$v_4 v_6$	v_7
$v_1 v_4 v_7$		$v_2 v_3 v_6$		$v_4 v_6 v_7$	$v_1 v_3$
$v_1 v_6$	v_7	$v_3 v_4$		$v_4 v_7$	$v_1 v_2 v_3 v_5 v_6$
$v_1 v_6 v_7$		$v_3 v_4$		v_5	v_7
$v_1 v_7$		$v_2 v_3 v_4 v_6$		$v_5 v_7$	$v_2 v_4$
v_2	$v_4 v_5 v_7$	v_1		v_6	v_7
$v_2 v_4$	$v_5 v_7$	v_1		$v_6 v_7$	$v_1 v_3 v_4$
$v_2 v_4 v_5$	v_7			v_7	$v_1 v_2 v_3 v_4 v_5 v_6$

Fig. 6.5. Execution of the simple maximal clique enumeration procedure, upon the undirected graph of Fig. 6.1. For each clique C , the corresponding set P contains those vertices which can be used to extend C to a larger clique, and the corresponding set S contains those vertices which cannot be used to further extend C to a larger clique, because all extensions of clique C with vertices from S were already obtained.

Example 6.9. Consider the execution of the simple maximal clique enumeration procedure illustrated in Fig. 6.3 and also in Fig. 6.5. The maximal cliques of the given undirected graph are those cliques C for which the set of candidate vertices P with which they can be extended is empty and the set of vertices S with which all extensions of C were already obtained is also empty. For instance, clique $\{v_1, v_3, v_4\}$ is not maximal, because it can be extended with vertices from $P = \{v_6, v_7\}$ in order to obtain the larger cliques $\{v_1, v_3, v_4, v_6\}$, $\{v_1, v_3, v_4, v_6, v_7\}$, and $\{v_1, v_3, v_4, v_7\}$. Neither is clique $\{v_1, v_3, v_7\}$ maximal, although $P = \emptyset$, because its extensions with vertices from

$S = \{v_4, v_6\}$, namely: $\{v_1, v_3, v_4, v_7\}$ and $\{v_1, v_3, v_6, v_7\}$, were already obtained. The only maximal cliques are $\{v_1, v_2, v_4, v_7\}$, $\{v_1, v_3, v_4, v_6, v_7\}$, and $\{v_2, v_4, v_5, v_7\}$.

The following simple result yields an effective improvement of the maximal clique enumeration procedure.

Lemma 6.10. *No extension of a clique $C \cup \{v\}$ to a larger clique can include vertex u if $\{u, v\} \notin E$, for all vertices $u, v \in P$.*

Proof. Let $u, v \in P$, and let $C \cup \{v\}$ and D be cliques such that $C \cup \{u, v\} \subseteq D$. Suppose $\{u, v\} \notin E$. Then, $C \cup \{u, v\}$ is not a clique, thus by Lemma 6.4 contradicting the hypothesis that D is a clique. \square

The improvement of the maximal clique enumeration procedure consists of considering only those candidate vertices $v \in P$ which are not adjacent to some vertex $u \in P$, in order to extend C to a larger clique.

Again, let $P \subseteq V \setminus C$ be the set of candidate vertices which can be used to extend C to a larger clique, and let $S \subseteq V \setminus C \setminus P$ be the set of vertices which cannot be used to extend C to a larger clique, because all cliques D with $C \subseteq D \subseteq C \cup S$ were already obtained. All possible extensions of C with vertices from P and without vertices from S can also be obtained by taking first some candidate vertex $u \in P$ and then, taking each candidate vertex $v \in P$ in turn which is not adjacent to vertex u and removing vertex v from P , adding vertex v to C , creating new sets P and S from the old sets by removing those vertices which are not adjacent to vertex v , recursively performing the procedure upon the new sets C , P , and S , and then removing vertex v from C and adding vertex v to S . Throughout this procedure, again, a clique C is maximal if both P and S are empty.

The effectiveness of the simple improvement to the maximal clique enumeration procedure is reflected in the reduction of the search tree for the execution of the recursive procedure.

Example 6.11. Consider the execution illustrated in Fig. 6.6 and also in Fig. 6.7 of the improved maximal clique enumeration procedure. The maximal cliques of the given undirected graph are those cliques C for which the set of candidate vertices P with which they can be

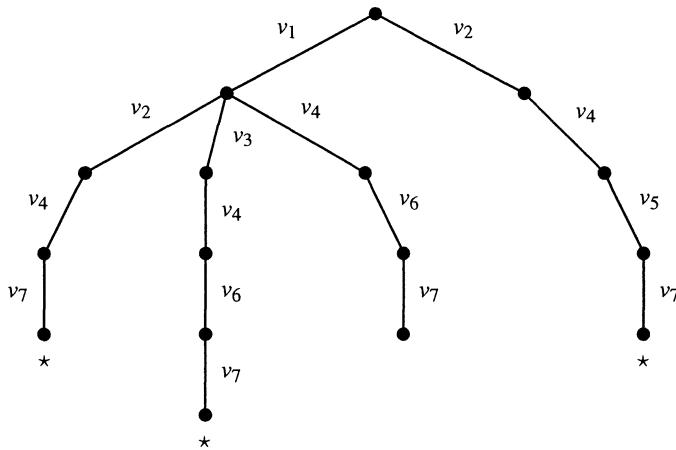


Fig. 6.6. Search tree for the execution of the maximal clique enumeration procedure, upon the undirected graph of Fig. 6.1. Every edge of the tree represents an extension of the current clique by adding one vertex, and the edge is labeled by the vertex added to the current clique. Every path in the tree from the root to a leaf node represents a clique of the graph. Those leaves of the tree representing a maximal clique are marked by a star.

extended is empty and the set of vertices S with which all extensions of C were already obtained is also empty. For instance, clique $\{v_1, v_4, v_6\}$ is not maximal, because it can be extended with a vertex from $P = \{v_7\}$ in order to obtain the larger clique $\{v_1, v_4, v_6, v_7\}$ which is also not maximal, although in this case $P = \emptyset$, because its extension with a vertex from $S = \{v_3\}$, namely: $\{v_1, v_3, v_6, v_7\}$, was already obtained. The only maximal cliques are $\{v_1, v_2, v_4, v_7\}$, $\{v_1, v_3, v_4, v_6, v_7\}$, and $\{v_2, v_4, v_5, v_7\}$.

The effectiveness of the simple improvement to the maximal clique enumeration procedure is reflected in the reduction of the search tree for the execution of the recursive procedure shown in Fig 6.3, to the smaller search tree shown in Fig 6.6. As a matter of fact, the search tree for the improved procedure has 16 nodes, corresponding to 15 nontrivial cliques obtained, and only four leaves, three of which correspond to maximal cliques. The search tree for the simple procedure, on the other hand, had 48 nodes, corresponding to 47 nontrivial cliques obtained, and 24 leaves, only three of which correspond to maximal cliques.

	C	P	S
		$v_1 v_2 v_3 v_4 v_5 v_6 v_7$	
	v_1	$v_2 v_3 v_4 v_6 v_7$	
	$v_1 v_2$	$v_4 v_7$	
	$v_1 v_2 v_4$	v_7	
<i>maximal clique</i>		$v_1 v_2 v_4 v_7$	
	$v_1 v_3$	$v_4 v_6 v_7$	
	$v_1 v_3 v_4$	$v_6 v_7$	
	$v_1 v_3 v_4 v_6$	v_7	
<i>maximum clique</i>		$v_1 v_3 v_4 v_6 v_7$	
	$v_1 v_6$	$v_4 v_7$	v_3
	$v_1 v_4 v_6$	v_7	v_3
	$v_1 v_4 v_6 v_7$		v_3
	v_5	$v_2 v_4 v_7$	
	$v_2 v_5$	$v_4 v_7$	
	$v_2 v_4 v_5$	v_7	
<i>maximal clique</i>		$v_2 v_4 v_5 v_7$	

Fig. 6.7. Execution of the maximal clique enumeration procedure, upon the undirected graph of Fig. 6.1. For each clique C , the corresponding set P contains those vertices which can be used to extend C to a larger clique, and the corresponding set S contains those vertices which cannot be used to further extend C to a larger clique, because all extensions of clique C with vertices from S were already obtained.

The following improved backtracking algorithm for enumerating maximal cliques extends an (initially empty) clique C in all possible ways, in order to enumerate all maximal cliques $C \subseteq V$ of an undirected graph $G = (V, E)$, implementing the improved maximal clique enumeration procedure. The maximal cliques found are collected in a list L of sets of vertices, and the adjacency matrix A of graph G will be used for double-checking each of the maximal cliques found.

```
(maximal clique 303a)+≡
void all_maximal_cliques(
    const graph& G,
    const node_matrix<bool>& A,
    list<set<node>>& L)
{
    set<node> C;
    set<node> P;
    set<node> S;
    node v;
    forall_nodes(v,G)
        P.insert(v);
    next_maximal_clique(G,A,C,P,S,L);
}
```

The actual extension of a clique is done by the following recursive procedure, which extends a clique $C \subseteq V$ of the undirected graph $G = (V, E)$ by adding to C each vertex $w \in V \setminus C$ in turn which is adjacent to all those vertices which are already in C .

Again, for each extension of clique C by adding a vertex $v \in P$, a new set PP of candidate vertices is created containing only those vertices $w \in P$ which are adjacent to vertex v , and a new set SS of noncandidate vertices is created containing only those vertices $w \in S$ which are adjacent to vertex v .

313

```
<maximal clique 303a>+≡
void next_maximal_clique(
    const graph& G,
    const node_matrix<bool>& A,
    set<node>& C,
    set<node>& P,
    set<node>& S,
    list<set<node>>& L)
{
    if ( P.empty() ) {
        if ( S.empty() ) {
            double_check_maximal_clique(G,A,C);
            L.append(C);
        } } else {
            node u = P.choose();
            node v,w;
            forall(v,P) {
                if ( ~A(u,v) ∨ ~A(v,u) ) {
                    P.del(v);
                    set<node> PP;
                    set<node> SS;
                    forall_adj_nodes(w,v) {
                        if ( P.member(w) ) PP.insert(w);
                        if ( S.member(w) ) SS.insert(w);
                    }
                    C.insert(v);
                    next_maximal_clique(G,A,C,PP,SS,L);
                    C.del(v);
                    S.insert(v);
                } } } }
```

The following double-check of the maximal clique, although being redundant, gives some reassurance of the correctness of the implementation. It verifies that the vertices of the set C induce a clique of graph G , by double-checking that every vertex of C is adjacent in G to every other vertex of C , and it also verifies that the clique of G

induced by the vertices of the set C is maximal, by double-checking that no other vertex of G is adjacent to all the vertices of the set C .

314a

```
<double-check maximal clique 314a>≡
void double_check_maximal_clique(
    const graph& G,
    const node_matrix<bool>& A,
    const set<node>& C)
{
    double_check_clique(G,A,C);

    node v,w;
    set<node> REST;
    forall_nodes(v,G)
        if ( ~C.member(v) ) REST.insert(v);
    bool adjacent_all;
    forall(v,REST) {
        adjacent_all = true;
        forall(w,C) {
            if ( ~A(v,w) ∨ ~A(w,v) ) {
                adjacent_all = false;
                break;
            }
        }
        if ( adjacent_all )
            error_handler(1,
                "Wrong implementation of maximal clique");
    }
}
```

Remark 6.12. Correctness of the improved backtracking algorithm for finding all maximal cliques, follows from the fact that an (initially empty) clique C of an undirected graph is extended in all possible ways, by adding one vertex from P at a time, until it cannot be further extended because P is empty. In such a case, Lemma 6.8 ensures that clique C is maximal if and only if S is also empty.

Interactive Demonstration of Maximal Cliques

The backtracking algorithm for finding all cliques is integrated next in the interactive demonstration of graph algorithms. A simple checker for cliques that provides some visual reassurance consists of highlighting the clique as a subgraph of the given graph. Sorting the list L of cliques in nonincreasing order of clique size, allows highlighting the cliques from the largest down to the smallest.

314b

```
<demo maximal clique 314b>≡
void gw_all_cliques(
```

```

GraphWin& gw)
{
graph& G = gw.get-graph();
Make_Bidirected(G);
gw.update-graph();

⟨set up adjacency matrix 316a⟩

list<set<node>> L;
all_cliques(G,A,L);

L.bucket_sort(clique_size);
L.reverse();

panel P;
make_proof_panel(P,
  string("There are %i nontrivial cliques",L.length()),true);
if ( gw.open_panel(P) ) { // proof button pressed

  node v;
  edge e;
  set<node> S;
  forall(S,L) {
    gw.save_all_attributes();
    forall(v,S) {
      gw.set_color(v,blue);
      forall_adj_edges(e,v) {
        if ( S.member(G.opposite(v,e)) ) {
          gw.set_color(G.opposite(v,e),blue);
          gw.set_color(e,blue);
          gw.set_width(e,2);
        }
      }
    }
    gw.redraw();
  panel Q;
  make_yes_no_panel(Q,"Continue",true);
  if ( gw.open_panel(Q) ) { // no button pressed
    gw.restore_all_attributes();
    break;
  }
  gw.restore_all_attributes();
}
}
}
}

```

Bucket sorting a list of cliques by size requires a function mapping a set of vertices to an integer, the size of the clique.

```

<demo maximal clique 314b>+≡
  inline int clique_size(
    const set<node>& C)
  {
    return C.size();
  }
}

```

Recall that graphs are represented in LEDA by adjacency lists. An adjacency matrix A for graph G will also be used in order to test adjacency of a vertex pair in $O(1)$ time.

316a ⟨set up adjacency matrix 316a⟩≡
`node_matrix<bool> A(G,false);
{ edge e;
forall_edges(e,G) {
A(G.source(e),G.target(e)) = true;
} }`

The improved backtracking algorithm for finding all the maximal cliques of an undirected graph is also integrated next in the interactive demonstration of graph algorithms. A simple checker for maximal cliques that provides some visual reassurance consists of highlighting the maximal clique as a subgraph of the given graph. Again, sorting the list L of cliques in nonincreasing order of clique size, allows highlighting the cliques from the largest down to the smallest.

316b ⟨demo maximal clique 314b⟩+≡
`void gw_all_maximal_cliques(
GraphWin& gw)
{
graph& G = gw.get_graph();
Make_Bidirected(G);
gw.update_graph();`
⟨set up adjacency matrix 316a⟩
`list<set<node>> L;
all_maximal_cliques(G,A,L);`
`L.bucket_sort(clique_size);
L.reverse();`
`panel P;
make_proof_panel(P,string("There are %i maximal cliques",L.length()),true);
if (gw.open_panel(P)) { // proof button pressed`
`node v;
edge e;
set<node> S;
forall(S,L) {
gw.save_all_attributes();
forall(v,S) {
gw.set_color(v,blue);
forall_adj_edges(e,v) {
if (S.member(G.opposite(v,e))) {
gw.set_color(G.opposite(v,e),blue);
gw.set_color(e,blue);`

```

    gw.set_width(e,2);
} } }
gw.redraw();
panel Q;
make_yes_no_panel(Q,"Continue",true);
if (gw.open_panel(Q)) { // no button pressed
    gw.restore_all_attributes();
    break;
}
gw.restore_all_attributes();

} } }

```

Maximum Clique of a Graph

The problem of determining whether an undirected graph $G = (V, E)$ with n vertices has a clique with k or more vertices, for a fixed integer k with $0 \leq k \leq n$, belongs to the class of NP-complete problems, meaning that all known algorithms for solving the maximum clique problem upon general graphs (that is, graphs without any restriction on any graph parameter) take time exponential in the size of the graphs, and that it is highly unlikely that such an algorithm will be found which takes time polynomial in the size of the graph.

All maximum cliques of an undirected graph can be obtained by first finding all maximal cliques of the graph, and then selecting the largest among all these maximal cliques. The improved backtracking algorithm for finding all maximal cliques, however, can be easily turned into a practical branch-and-bound algorithm for finding a maximum clique of an undirected graph.

Notice first that the maximum vertex degree in an undirected graph gives an upper bound on the size of a maximum clique of the graph.

Lemma 6.13. *Let $C \subseteq V$ be a maximum clique of an undirected graph $G = (V, E)$. Then, $\text{size}[C] \leq \max_{v \in V} \deg(v) + 1$.*

Proof. Let $C \subseteq V$ be a maximum clique of an undirected graph $G = (V, E)$, let $\text{maxdeg} = \max_{v \in V} \deg(v)$, and suppose $\text{size}[C] > \text{maxdeg} + 1$. Then, by Definition 6.1, $\{v, w\} \in E$ for all distinct vertices $v, w \in C$ and then, $\deg(v) \geq \text{size}[C] - 1 > \text{maxdeg}$ for all vertices $v \in C$, contradicting the hypothesis that $\text{maxdeg} = \max_{v \in V} \deg(v)$. Therefore, $\text{size}[C] \leq \max_{v \in V} \deg(v) + 1$. \square

Now, in the improved backtracking algorithm for finding all maximal cliques, an upper bound on the size of an extension of a clique $C \subseteq V$ of an undirected graph $G = (V, E)$, is given by the total size of the clique C being extended and the set P of candidate vertices with which clique C can be extended.

Lemma 6.14. *Let $D \subseteq V$ be a maximum clique of an undirected graph $G = (V, E)$, and let $C \subseteq D$. Then, $\text{size}[D] \leq \text{size}[C] + \text{size}[P]$.*

Proof. Let $C \subseteq D$ be a clique contained in a maximum clique $D \subseteq V$ of an undirected graph $G = (V, E)$. Since clique C can only be extended by adding vertices from P , it follows that $\text{size}[E] \leq \text{size}[C] + \text{size}[P]$ for any extension E of clique C and, in particular, $\text{size}[D] \leq \text{size}[C] + \text{size}[P]$. \square

Both the upper bound on the size of a maximum clique and the upper bound on the size of an extension of a clique, can be used to turn the backtracking algorithm into a practical branch-and-bound algorithm. The procedure is identical to the backtracking procedure for finding all maximal cliques of an undirected graph, except that a clique C is recursively extended if and only if the size of clique C is still below the upper bound on the size of an extension of the clique and, moreover, the size of the largest clique found so far is below the upper bound given by the maximum vertex degree.

Example 6.15. Consider the execution of the procedure for finding a maximum clique, illustrated in Fig. 6.8 and Fig. 6.9. The maximal cliques of the given undirected graph are those cliques C for which the set of candidate vertices P with which they can be extended is empty and the set of vertices S with which all extensions of C were already obtained is also empty. A clique C is actually extended only if there are enough candidate vertices in P to eventually obtain a clique larger than the current largest clique, and unless the current largest clique is already as large as one plus the maximum vertex degree.

For instance, clique $C = \{v_1, v_3, v_4\}$ is further extended, because the current largest clique is $|\text{MAX}| = \{v_1, v_2, v_4, v_7\}$, $P = \{v_6, v_7\}$, $\text{size}[\text{MAX}] = 4 \leq 3 + 2 = \text{size}[C] + \text{size}[P]$, and also $\text{size}[\text{MAX}] \leq 7$, one plus the maximum vertex degree in the graph.

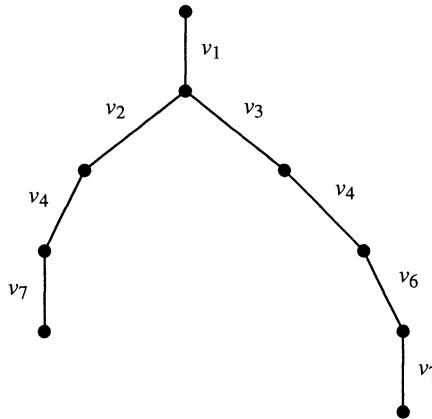


Fig. 6.8. Search tree for the execution of the maximum clique procedure, upon the undirected graph of Fig. 6.1. Every edge of the tree represents an extension of the current clique by adding one vertex, and the edge is labeled by the vertex added to the current clique. Every path in the tree from the root to a leaf node represents a clique of the graph. Both leaves of the tree represent a maximal clique, and the rightmost leaf node also stands for a maximum clique.

	C	P	S
	$v_1 v_2 v_3 v_4 v_5 v_6 v_7$		
v_1		$v_2 v_3 v_4 v_6 v_7$	
$v_1 v_2$		$v_4 v_7$	
$v_1 v_2 v_4$		v_7	
<i>maximal clique</i>	$v_1 v_2 v_4 v_7$		
	$v_1 v_3$	$v_4 v_6 v_7$	
	$v_1 v_3 v_4$	$v_6 v_7$	
	$v_1 v_3 v_4 v_6$	v_7	
<i>maximum clique</i>	$v_1 v_3 v_4 v_6 v_7$		

Fig. 6.9. Execution of the procedure for finding a maximum clique, upon the undirected graph of Fig. 6.1. For each clique C , the corresponding set P contains those vertices which can be used to extend C to a larger clique, and the corresponding set S contains those vertices which cannot be used to further extend C to a larger clique, because all extensions of clique C with vertices from S were already obtained. However, a clique C is actually extended only if there are enough candidate vertices in P to eventually obtain a clique larger than the current largest clique.

The following branch-and-bound algorithm for finding a maximum clique extends an (initially empty) clique C in all possible ways, in order to enumerate all maximal cliques $C \subseteq V$ of an undirected graph $G = (V, E)$, implementing the previous procedure. The adjacency matrix A of graph G will be used for double-checking each of the maximal cliques found.

320a

```

⟨maximum clique 320a⟩≡
void maximum_clique(
    graph& G,
    node_matrix<bool>& A,
    set<node>& MAX)
{
    set<node> C;
    set<node> P;
    set<node> S;
    node v;
    int max_deg = 0;
    forall_nodes(v,G) {
        P.insert(v);
        max_deg = led_max(max_deg,G.outdeg(v));
    }
    MAX.clear();
    next_maximum_clique(G,A,C,P,S,max_deg,MAX);
}

```

The actual extension of a clique is done by the following recursive procedure, which extends a clique $C \subseteq V$ of the undirected graph $G = (V, E)$ by adding to C each vertex $w \in V \setminus C$ in turn which is adjacent to all those vertices which are already in C .

Again, for each extension of clique C by adding a vertex $v \in P$, a new set PP of candidate vertices is created containing only those vertices $w \in P$ which are adjacent to vertex v , and a new set SS of noncandidate vertices is created containing only those vertices $w \in S$ which are adjacent to vertex v .

Further, an extension is only computed if it may eventually lead to a clique larger than the largest clique found so far, that is, if the size of the current largest clique MAX is less than the size of clique C plus the size of the new set PP of candidate vertices with which clique C can be extended, and unless MAX is already as large as one plus the maximum vertex degree.

320b

```

⟨maximum clique 320a⟩+≡

```

```

void next_maximum_clique(
    const graph& G,
    const node_matrix<bool>& A,
    set<node>& C,
    set<node>& P,
    set<node>& S,
    const int max_deg,
    set<node>& MAX)
{
    if ( MAX.size() < C.size() )

```

```

MAX = C; // new largest clique found

if ( P.empty() ) {
  if ( S.empty() ) {
    double_check_maximal_clique(G,A,C);
  } } else {
  node u = P.choose();
  node v,w;
  forall(v,P) {
    if ( ¬A(u,v) ∨ ¬A(v,u) ) {
      P.del(v);
      set<node> PP;
      set<node> SS;
      forall_adj_nodes(w,v) {
        if ( P.member(w) ) PP.insert(w);
        if ( S.member(w) ) SS.insert(w);
      }
      C.insert(v);
      if ( MAX.size() < C.size() + PP.size() ∧
          MAX.size() < max_deg + 1 )
        next_maximum_clique(G,A,C,PP,SS,max_deg,MAX);
      C.del(v);
      S.insert(v);
    } } } }

```

Remark 6.16. Correctness of the branch-and-bound algorithm for finding a maximum clique follows from the fact that an (initially empty) clique C of an undirected graph is extended in all possible ways, by adding one vertex from P at a time, until it cannot be further extended because P is empty. In this case, Lemma 6.8 ensures that clique C is maximal if and only if S is also empty, and clique C being maximal is a necessary condition for C to be a maximum clique, by Lemma 6.2.

Further, clique C is actually extended only if there are enough candidate vertices in P to eventually obtain a clique larger than the current largest clique, according to Lemma 6.14, and unless the current largest clique is already as large as one plus the maximum vertex degree, according to Lemma 6.13.

Interactive Demonstration of Maximum Clique

The branch-and-bound algorithm for finding a maximum clique is integrated next in the interactive demonstration of graph algorithms. A simple checker for maximum clique that provides some visual reas-

surance consists of highlighting the maximum clique as a subgraph of the given graph.

```
322 ⟨demo maximum clique 322⟩≡
  void gw_maximum_clique(
    GraphWin& gw)
  {
    graph& G = gw.get_graph();
    Make_Bidirected(G);
    gw.update_graph();

    ⟨set up adjacency matrix 316a⟩

    set<node> MAX;
    maximum_clique(G,A,MAX);

    gw.save_all_attributes();
    node v;
    edge e;
    forall(v,MAX) {
      gw.set_color(v,blue);
      forall_adj_edges(e,v) {
        if ( MAX.member(G.opposite(v,e)) ) {
          gw.set_color(G.opposite(v,e),blue);
          gw.set_color(e,blue);
          gw.set_width(e,2);
        } }
      gw.redraw();
      gw.wait();
      gw.restore_all_attributes();
    }
  }
```

6.2 Maximal and Maximum Independent Sets

An independent set of an undirected graph is an induced subgraph of the graph which has no edges. A maximal independent set of a graph is an independent set of the graph that is not properly included in any other independent set of the graph, and a maximum independent set of a graph is a (maximal) independent set of the graph with the largest number of vertices.

Definition 6.17. An *independent set* of an undirected graph $G = (V, E)$ is a set of vertices $I \subseteq V$ such that the subgraph of G induced by I has no edges, that is, such that $\{v, w\} \notin E$ for all distinct vertices $v, w \in I$. An *independent set* I of an undirected graph $G = (V, E)$

is maximal if there is no independent set J of G such that $I \subseteq J$ and $I \neq J$, and it is maximum if there is no independent set of G with more vertices than I . The independence number of G , denoted by $\beta(G)$, is the cardinality of a maximum independent set of G .

It follows from the previous definition that maximum independent sets are also maximal.

Lemma 6.18. *Every maximum independent set $I \subseteq V$ of an undirected graph $G = (V, E)$ is maximal.*

Proof. Let $I \subseteq V$ be a maximum independent set of an undirected graph $G = (V, E)$, and suppose I is not maximal. There is, by Definition 6.17, an independent set $J \subseteq V$ such that $I \subseteq J$ and $I \neq J$. Then, $J \setminus I \neq \emptyset$, that is, independent set J has more vertices than I , contradicting the hypothesis that I is a maximum independent set. Therefore, I is also a maximal independent set. \square

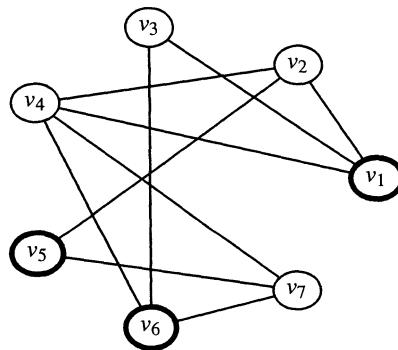


Fig. 6.10. Maximal and maximum independent sets of an undirected graph. The independent set $\{v_1, v_6\}$ is not maximal, because it is included in the independent set $\{v_1, v_5, v_6\}$, which is a maximal and also a maximum independent set.

Example 6.19. The undirected graph shown in Fig. 6.10 has five maximal independent sets: $\{v_1, v_7\}$, $\{v_2, v_6\}$, $\{v_1, v_5, v_6\}$, $\{v_2, v_3, v_7\}$, and $\{v_3, v_4, v_5\}$. The latter three are also maximum independent sets of the graph.

Notice that being an independent set of an undirected graph is another hereditary graph property.

Lemma 6.20. *Let $I \subseteq V$ be an independent set of an undirected graph $G = (V, E)$, and let $H \subseteq I$. Then, H is also an independent set of G .*

Proof. Let $I \subseteq V$ be an independent set of an undirected graph $G = (V, E)$, and let $H \subseteq I$. Then, $\{v, w\} \notin E$ for all distinct vertices $v, w \in I$ by Definition 6.17 and, in particular, $\{v, w\} \notin E$ for all distinct vertices $v, w \in H$. Therefore, $H \subseteq V$ is also an independent set. \square

Maximum Independent Set of a Tree

An independent set of a tree is a set of nodes of the tree which does not include any of the children of the nodes in the independent set or, in an equivalent formulation, a set of nodes of the tree which does not include the parent of any of the nodes in the independent set. A maximal independent set of a tree is an independent set of the tree that is not properly included in any other independent set of the tree, and a maximum independent set of a tree is a (maximal) independent set of the tree with the largest number of nodes.

Definition 6.21. *An independent set of a rooted tree $T = (V, E)$ is a set of nodes $I \subseteq V$ such that $\text{parent}[v] \notin I$ for all nonroot nodes $v \in I$. An independent set I of a rooted tree $T = (V, E)$ is maximal if there is no independent set J of T such that $I \subseteq J$ and $I \neq J$, and it is maximum if there is no independent set of T with more nodes than I . The independence number of T , denoted by $\beta(T)$, is the cardinality of a maximum independent set of T .*

Example 6.22. A maximum independent set of a rooted tree is illustrated in Fig. 6.11. All the leaves of the tree belong to the maximum independent set, together with all those nonleaves none of whose children, but at least one of whose grandchildren, belong to the maximum independent set.

In a rooted tree, the children of a node constitute an independent set, because otherwise the undirected graph underlying the tree would no longer be acyclic. For the same reason, all the leaves of a tree also constitute an independent set of the tree.

A particular maximum independent set of a tree can be obtained as follows. All the leaves of the tree belong to the maximum independent

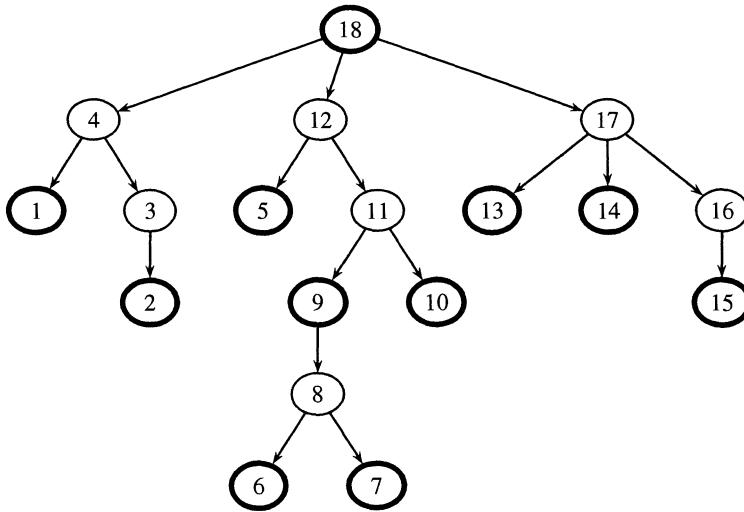


Fig. 6.11. Maximum independent set of the rooted tree of Fig. 3.3. Nodes are numbered according to the order in which they are visited during a postorder traversal.

set, together with all those nonleaves none of whose children, but at least one of whose grandchildren, belong to the maximum independent set.

Lemma 6.23. *Let $T = (V, E)$ be a tree with n nodes, and let $I = I_n \subseteq V$ be the set of nodes defined by induction as follows:*

- $I_1 = \{v \in V \mid (v, w) \notin E \text{ for all } w \in V\}$.
- $I_{i+1} = I_i \cup \{u \in V \setminus I_i \mid v \notin I_i \text{ for all } (u, v) \in E\} \cap \{u \in V \setminus I_i \mid (u, v), (v, w) \in E \text{ for some } w \in I_i\}$, for $1 \leq i < n$.

Then, I is a maximum independent set of T .

Proof. Suppose $I \subseteq V$ is not an independent set of $T = (V, E)$. Then, there are nodes $v, w \in I$ such that $(v, w) \in E$. Now, it must be true that $v \notin I_1$, because $(v, w) \in E$, and it must also be true that $v \notin I_i$ for $2 \leq i \leq n$, because $(v, w) \in E$ and $w \in I$, contradicting the hypothesis that $v \in I$. Therefore, I is an independent set of T .

Suppose now that I is not maximal. Then, there is a node $v \in V \setminus I$ which is neither the parent nor a child of any node $w \in I$, that is, whose parent and all of whose children belong to $V \setminus I$. Then, for some i , with $1 \leq i < n$, it holds that $v \in \{x \in V \setminus I_i \mid y \notin I_i \text{ for all } (x, y) \in E\}$.

$E\}$. Further, for all children w of node v , there must be $(w, z) \in E$ for some node $z \in I$, because otherwise $w \in I$, and then $v \in \{x \in V \setminus I_i \mid (x, y), (y, z) \in E \text{ for some } z \in I_i\}$. Therefore, $v \in I_i$ for some i , with $1 \leq i \leq n$, contradicting the hypothesis that $v \in V \setminus I$. Then, I is a maximal independent set of T .

Finally, notice that the set I_1 contains all the nodes of least height of T and for $1 \leq i < n$, the set I_{i+1} contains all those nodes of least height which are not the parent of any node in I_i . Since the number of nodes of height $i+1$ is less than or equal to the number of nodes of height i , it follows that I is a maximum independent set of T . \square

Now, a maximum independent set of a tree can be obtained by collecting *independent* nodes during a postorder traversal of the tree. All the leaves of the tree are independent, and nonleaves are independent if none of their children belong to the maximum independent set.

The following algorithm finds a maximum independent set $I \subseteq V$ of a tree $G = (V, E)$, implementing the previous procedure.

326

```
<independent set 326>≡
void maximum_independent_set(
    const TREE<string,string>& T,
    set<node>& I)
{
    I.clear();
    list<node> L;
    postorder_tree_list_traversal(T,L);
    node_array<bool> independent(T,true);
    node v,w;
    forall(v,L) {
        if ( ¬T.is_leaf(v) ) {
            forall_children(w,v) {
                if ( independent[w] ) {
                    independent[v] = false;
                    break;
                }
            }
            if ( independent[v] ) I.insert(v);
        }
        double_check_tree_independent_set(T,I);
    }
}
```

The following double-check of a maximal independent set, although being redundant, gives some reassurance of the correctness of the implementation. It verifies that the nodes of the set I constitute an independent set, by double-checking that no node of I is the parent

in T of any other node of I , and it also verifies that the independent set I is maximal, by double-checking that every node of $V \setminus I$ is either the parent or a child of some node of I .

327

```
(double-check independent set 327)≡
void double_check_tree_independent_set(
    const TREE<string,string>& T,
    const set<node>& I)
{
    node v,w;
    forall(v,I) {
        if ( !T.is_root(v) & I.member(T.parent(v)) ) {
            error_handler(1,
                "Wrong implementation of independent set");
        }
    }

    bool independent;
    forall_nodes(v,T) {
        if ( !I.member(v) ) {
            if ( !T.is_root(v) & !I.member(T.parent(v)) ) {
                independent = true; // presumed guilty
                forall_children(w,v) {
                    if ( I.member(w) ) {
                        independent = false; // found innocent
                        break;
                    }
                }
                if ( independent ) { // condemned
                    error_handler(1,
                        "Wrong implementation of maximal independent set");
                }
            }
        }
    }
}
```

Lemma 6.24. *The algorithm for finding a maximum independent set of a rooted tree runs in $O(n \log n)$ time using $O(n)$ additional space, where n is the number of nodes in the tree.*

Proof. Let $T = (V, E)$ be a rooted tree with n nodes. A postorder traversal is made which, by Lemma 3.10, takes $O(n)$ time using $O(n)$ additional space. Further, $O(n)$ insertions in an initially empty set of nodes are made, corresponding to the nodes in $I \subseteq V$. Therefore, the algorithm runs in $O(n \log n)$ time using $O(n)$ additional space. The double-check of maximal independent set also runs in $O(n \log n)$ time using $O(n)$ additional space. \square

Remark 6.25. The tree maximum independent set algorithm can also be implemented to run in $O(n)$ time, where n is the number of nodes in the tree, while still using $O(n)$ additional space. See Exercise 6.4.

Maximum Independent Set of a Graph

The problem of determining whether an undirected graph $G = (V, E)$ with n vertices has an independent set with k or more vertices, for a fixed integer k with $0 \leq k \leq n$, also belongs to the class of NP-complete problems, meaning that all known algorithms for solving the maximum independent set problem upon general graphs (that is, graphs without any restriction on any graph parameter) take time exponential in the size of the graphs, and that it is highly unlikely that such an algorithm will be found which takes time polynomial in the size of the graph.

Finding a maximum independent set of an undirected graph can be reduced to the problem of finding a maximum clique in the complement of the graph. Recall from Sect. 1.1 that the complement of an undirected graph (V, E) is the undirected graph (V, F) , where $\{v, w\} \in F$ if and only if $\{v, w\} \notin E$, for all distinct vertices $v, w \in V$.

Lemma 6.26. *Let $C \subseteq V$ be a maximum clique of the complement of a graph $G = (V, E)$. Then, C is a maximum independent set of graph G .*

Proof. It suffices to show that a clique $C \subseteq V$ of the complement of a graph $G = (V, E)$ is an independent set of graph G . Let (V, F) be the complement of graph G . Then, by Definition 6.1, $\{v, w\} \in F$ for all distinct vertices $v, w \in C$, that is, $\{v, w\} \notin E$ for all distinct vertices $v, w \in C$. Therefore, C is, by Definition 6.17, an independent set of graph G . \square

Example 6.27. The relationship between maximum clique and maximum independent set is illustrated in Fig. 6.12. The set of vertices $\{v_1, v_5, v_6\}$ is shown as a maximum independent set of the graph of Fig. 6.10, and also as a maximum clique of the complement of the graph.

The following algorithm implements the construction of the complement H of an undirected graph G . The correspondence among the vertices of G and H is represented by an array M of vertices (of graph G) indexed by the vertices of graph H .

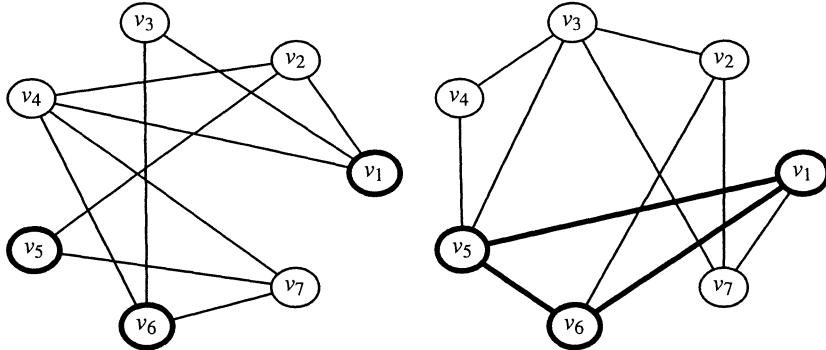


Fig. 6.12. Complement (right) of the undirected graph (left) of Fig. 6.10. A maximum independent set in the graph corresponds to a maximum clique in the complement of the graph.

328

\langle independent set 326 $\rangle + \equiv$

```
void graph_complement(
    const GRAPH<string,string>& G,
    GRAPH<string,string>& H,
    node_array<node>& M)
{
    H.clear(); // reset H to the empty graph
    node_array<node> N(G);
    node v,w;
    forall_nodes(v,G) {
        w = H.new_node();
        H[w] = G[v]; // vertex label
        M[w] = v;
        N[v] = w;
    }
    edge e;
    forall_edges(e,G)
        H.new_edge(N[G.source(e)],N[G.target(e)]);
    graph_complement(H);
}
```

The actual construction of the complement of an undirected graph is implemented by the following algorithm.

329

\langle independent set 326 $\rangle + \equiv$

```
void graph_complement(
    GRAPH<string,string>& G)
{
    Make_Bidirected(G);

    {set up adjacency matrix 316a}

    G.del_all_edges();
```

```

node v,w;
forall_nodes(v,G) {
    forall_nodes(w,G) {
        if( v ≠ w ∧ ¬A(v,w) ) {
            G.new_edge(v,w);
        } } } }

```

Now, the following algorithm implements the reduction of a maximum independent set to a maximum clique. The maximum independent set found is represented by a set I of vertices, and the adjacency matrix A of the complement H of graph G will be used for finding a maximum clique I of H , that is, a maximum independent set I of G .

330a \langle independent set 326 $\rangle + \equiv$

```

void maximum_independent_set(
    const GRAPH<string,string>& G,
    set<node>& I)
{
    GRAPH<string,string> H;
    node_array<node> M(G);
    graph_complement(G,H,M);

    node_matrix<bool> A(H,false);
    { edge e;
        forall_edges(e,H) {
            A(H.source(e),H.target(e)) = true;
        } }
    set<node> C;
    maximum_clique(H,A,C);

    I.clear();
    node v;
    forall(v,C)
        I.insert(M[v]);
    double_check_independent_set(G,I);
}

```

The following double-check of the maximal independent set, although being redundant, gives some reassurance of the correctness of the implementation. It verifies that the vertices of the set I constitute an independent set, by double-checking that no vertex of I is adjacent in G to any other vertex of I , and it also verifies that the independent set I is maximal, by double-checking that every vertex of $V \setminus I$ is adjacent to some vertex of I .

330b \langle double-check independent set 327 $\rangle + \equiv$

```

void double_check_independent_set()

```

```

const GRAPH<string,string>& G,
const set<node>& I)
{
    ⟨set up adjacency matrix 316a⟩

node v,w;
forall(v,I)
    forall(w,I)
        if ( v ≠ w ∧ ( A(v,w) ∨ A(w,v) ) )
            error_handler(1,
                "Wrong implementation of independent set");

bool independent;
forall_nodes(v,G) {
    if ( ¬I.member(v) ) {
        independent = true; // presumed guilty
        forall(w,I) {
            if ( A(v,w) ∨ A(w,v) ) {
                independent = false; // found innocent
                break;
            } }
        if ( independent ) { // condemned
            error_handler(1,
                "Wrong implementation of maximal independent set");
        } } } }

```

Interactive Demonstration of Maximum Independent Set

The algorithm for finding a maximum independent set of a tree is integrated next in the interactive demonstration of graph algorithms. A simple checker for a maximum independent set that provides some visual reassurance consists of highlighting the nodes in the minimum independent set as a subgraph of the given tree.

331

```

<demo independent set 331>≡
void gw_tree_maximum_independent_set(
    GraphWin& gw)
{
    GRAPH<string,string>& G = gw.get_graph();
    TREE<string,string> T(G);
    set<node> I;
    maximum_independent_set(T,I);

    gw.save_all_attributes();
    node v;
    forall(v,I) gw.set_color(v,blue);
    gw.wait();
    gw.restore_all_attributes();
}

```

The algorithm for finding a maximum independent set of a graph is also integrated next in the interactive demonstration of graph algorithms. Again, a simple checker for a maximum independent set that provides some visual reassurance consists of highlighting the vertices in the minimum independent set as a subgraph of the given graph.

332a

```
<demo independent set 331>+≡
void gw_maximum_independent_set(
    GraphWin& gw)
{
    GRAPH<string,string>& G = gw.get_graph();
    Make_Bidirected(G);
    gw.update_graph();

    set<node> I;
    maximum_independent_set(G,I);

    gw.save_all_attributes();
    node v;
    forall(v,I)
        gw.set_color(v,blue);
    gw.wait();
    gw.restore_all_attributes();
}
```

The complement of a graph is illustrated by just showing it in place of the given graph, in the interactive demonstration of graph algorithms.

332b

```
<demo independent set 331>+≡
void gw_graph_complement(
    GraphWin& gw)
{
    GRAPH<string,string>& G = gw.get_graph();
    graph_complement(G);
    gw.update_graph();
}
```

6.3 Minimal and Minimum Vertex Covers

A vertex cover of an undirected graph is a set of vertices covering all of the edges of the graph, that is, such that every edge of the graph is incident with some vertex of the vertex cover. A minimal vertex cover of a graph is a vertex cover of the graph that does not properly include

any other vertex cover of the graph, and a minimum vertex cover of a graph is a (minimal) vertex cover of the graph with the smallest number of vertices.

Definition 6.28. A *vertex cover* of an undirected graph $G = (V, E)$ is a set of vertices $C \subseteq V$ such that $v \in C$ or $w \in C$ for all edges $\{v, w\} \in E$. A vertex cover C of an undirected graph $G = (V, E)$ is **minimal** if there is no vertex cover D of G such that $D \subseteq C$ and $C \neq D$, and it is **minimum** if there is no vertex cover of G with less vertices than C . The *vertex cover number* of G , denoted by $\alpha(G)$, is the cardinality of a minimum vertex cover of G .

It follows from the previous definition that minimum vertex covers are also minimal.

Lemma 6.29. Every minimum vertex cover $C \subseteq V$ of an undirected graph $G = (V, E)$ is minimal.

Proof. Let $C \subseteq V$ be a minimum vertex cover of an undirected graph $G = (V, E)$, and suppose C is not minimal. There is, by Definition 6.28, a vertex cover $D \subseteq V$ such that $D \subseteq C$ and $C \neq D$. Then, $C \setminus D \neq \emptyset$, that is, vertex cover D has less vertices than C , contradicting the hypothesis that C is a minimum vertex cover. Therefore, C is also a minimal vertex cover. \square

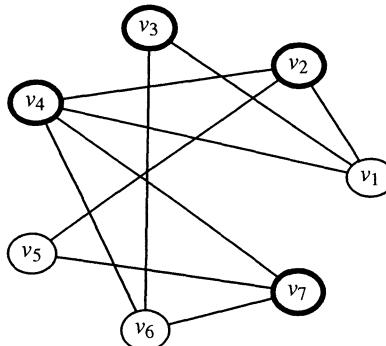


Fig. 6.13. Minimal and minimum vertex covers of an undirected graph. Vertex cover $\{v_1, v_2, v_3, v_4, v_7\}$ is not minimal, because it includes vertex cover $\{v_2, v_3, v_4, v_7\}$, which is a minimal and also a minimum vertex cover.

Example 6.30. The undirected graph shown in Fig. 6.13 has five minimal vertex covers: $\{v_1, v_2, v_6, v_7\}$, $\{v_1, v_4, v_5, v_6\}$, $\{v_2, v_3, v_4, v_7\}$, $\{v_1, v_3, v_4, v_5, v_7\}$, and $\{v_2, v_3, v_4, v_5, v_6\}$. The first three are also minimum vertex covers of the graph.

The notions of vertex cover and independent set are dual to each other, although in a different sense than for cliques and independent sets. Given an independent set of an undirected graph, those vertices of the graph which do not belong to the independent set constitute a vertex cover of the graph.

Lemma 6.31. *A set of vertices $W \subseteq V$ is an independent set of an undirected graph $G = (V, E)$ if and only if the set of vertices $V \setminus W$ is a vertex cover of G .*

Proof. Let $W \subseteq V$ be an independent set of an undirected graph $G = (V, E)$, and suppose $V \setminus W$ is not a vertex cover of G . Then, $v, w \notin V \setminus W$ for some edge $\{v, w\} \in E$, that is, there are vertices $v, w \in W$ with $\{v, w\} \in E$, contradicting the hypothesis that W is an independent set of G . Therefore, $V \setminus W$ is a vertex cover of G .

Now, let $V \setminus W$ be a vertex cover of an undirected graph $G = (V, E)$, where $W \subseteq V$, and suppose W is not an independent set of G . Then, $\{v, w\} \in E$ for some vertices $v, w \in W$, that is, there is an edge $\{v, w\} \in E$ such that neither $v \in V \setminus W$ nor $w \in V \setminus W$, contradicting the hypothesis that $V \setminus W$ is a vertex cover of G . Therefore, W is an independent set of G . \square

There is also a close relationship between the cardinalities of a maximum independent set and a minimum vertex cover of an undirected graph.

Theorem 6.32. *Let $G = (V, E)$ be an undirected graph with n vertices. Then, $\alpha(G) + \beta(G) = n$.*

Proof. Let $G = (V, E)$ be an undirected graph with n vertices, and let $W \subseteq V$ be a maximum independent set of G , that is, an independent set of G with $\beta(G)$ vertices. Since, by Lemma 6.31, $V \setminus W$ is a vertex cover of G , it follows that $\alpha(G) \leq n - \beta(G)$. Let now $W \subseteq V$ be a minimum vertex cover of G , that is, a vertex cover of G with $\alpha(G)$ vertices. Since, by Lemma 6.31, $V \setminus W$ is an independent set of G , it follows that $\beta(G) \geq n - \alpha(G)$. Therefore, $\alpha(G) + \beta(G) = n$. \square

Remark 6.33. Notice that the existence of isolated vertices in an undirected graph does not alter the previous relationship between the independence number and the vertex cover number of the graph. As a matter of fact, isolated vertices must belong to every maximum independent set, but no minimum vertex cover can have an isolated vertex.

Now, given a maximum independent set of an undirected graph, those vertices of the graph which do not belong to the maximum independent set constitute a minimum vertex cover of the graph.

Corollary 6.34. *A set of vertices $W \subseteq V$ is a maximum independent set of an undirected graph $G = (V, E)$ if and only if the set of vertices $V \setminus W$ is a minimum vertex cover of G .*

Proof. Let $W \subseteq V$ be a maximum independent set of an undirected graph $G = (V, E)$, that is, an independent set of G with $\beta(G)$ vertices. Then, by Lemma 6.31, $V \setminus W$ is a vertex cover of G , and it has $n - \beta(G)$ vertices. Now, it follows from Theorem 6.32 that $V \setminus W$ has $\alpha(G)$ vertices and therefore, $V \setminus W$ is a minimum vertex cover of G .

Now, let $W \subseteq V$ be a minimum vertex cover of an undirected graph $G = (V, E)$, that is, a vertex cover of G with $\alpha(G)$ vertices. Then, by Lemma 6.31, $V \setminus W$ is an independent set of G , and it has $n - \alpha(G)$ vertices. Now, it follows from Theorem 6.32 that $V \setminus W$ has $\beta(G)$ vertices and therefore, $V \setminus W$ is a maximum independent set of G . \square

Minimum Vertex Cover of a Tree

A vertex cover of a tree is a set of nodes covering all of the edges of the tree, that is, such that for all nodes in the tree, the node or the parent of the node belong to the vertex cover. A minimal vertex cover of a tree is a vertex cover of the tree that does not properly include any other vertex cover of the tree, and a minimum vertex cover of a tree is a (minimal) vertex cover of the tree with the smallest number of nodes.

Definition 6.35. *A vertex cover of a rooted tree $T = (V, E)$ is a set of nodes $C \subseteq V$ such that $v \in C$ or $\text{parent}[v] \in C$ for all nonroot nodes $v \in V$. A vertex cover C of a rooted tree $T = (V, E)$ is **minimal** if there is no vertex cover D of T such that $D \subseteq C$ and $C \neq D$, and it*

is **minimum** if there is no vertex cover of T with less nodes than C . The **vertex cover number** of T , denoted by $\alpha(T)$, is the cardinality of a minimum vertex cover of T .

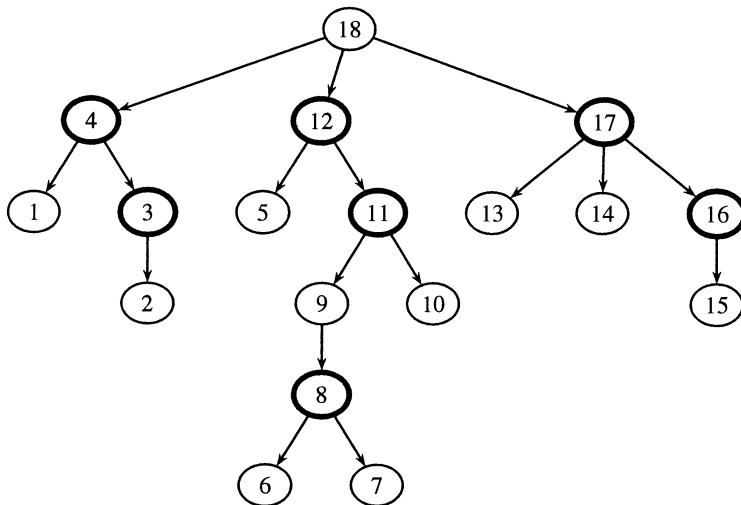


Fig. 6.14. Minimum vertex cover of the rooted tree of Fig. 3.3. Nodes are numbered according to the order in which they are visited during a postorder traversal.

Example 6.36. The set of nodes $\{v_3, v_4, v_8, v_{11}, v_{12}, v_{16}, v_{17}\}$ is a minimum vertex cover $C \subseteq V$ of the rooted tree $T = (V, E)$ shown in Fig. 6.14. The parent of every nonroot node of the tree which does not belong to the minimum vertex cover, does indeed belong to the minimum vertex cover. In fact, $\text{parent}[v_1] = v_4 \in C$, $\text{parent}[v_2] = v_3 \in C$, $\text{parent}[v_5] = v_{12} \in C$, $\text{parent}[v_6] = \text{parent}[v_7] = v_8 \in C$, $\text{parent}[v_9] = \text{parent}[v_{10}] = v_{11} \in C$, $\text{parent}[v_{13}] = \text{parent}[v_{14}] = v_{17} \in C$, and also $\text{parent}[v_{15}] = v_{16} \in C$. Further, root node $v_{18} \notin C$, because all of children nodes $v_4, v_{12}, v_{17} \in C$.

A minimum vertex cover of a tree can be obtained by first finding a maximum independent set of the tree, and then collecting all those nodes of the tree which do not belong to the maximum independent set.

The following algorithm finds a minimum vertex cover $C \subseteq V$ of a tree $G = (V, E)$, implementing the previous procedure.

337a

```
<vertex cover 337a>≡
void minimum_vertex_cover(
    const TREE<string,string>& T,
    set<node>& C)
{
    C.clear();
    if ( T.number_of_nodes() ≡ 1 ) {
        C.insert(T.first_node());
    } else {
        set<node> I;
        maximum_independent_set(T,I);
        node v;
        forall_nodes(v,T) {
            if ( ¬I.member(v) ) C.insert(v);
        }
        double_check_tree_vertex_cover(T,C);
    }
}
```

The following double-check of the minimal vertex cover, although being redundant, gives some reassurance of the correctness of the implementation. It verifies that the nodes of the set C constitute a vertex cover, by double-checking that $v \in C$ or $\text{parent}[v] \in C$ for all nodes $v \in V$, and it also verifies that the vertex cover C is minimal, by double-checking that, for all nodes $v \in C$, either $\text{parent}[v] \notin C$ or $w \notin C$ for some child w of node v .

337b

```
<double-check vertex cover 337b>≡
void double_check_tree_vertex_cover(
    const TREE<string,string>& T,
    const set<node>& C)
{
    if ( T.number_of_nodes() ≡ 1 ) return;

    node v,w;
    forall_nodes(v,T) {
        if ( ¬C.member(v) ∧ ¬T.is_root(v) ∧
            ¬C.member(T.parent(v)) ) {
            error_handler(1,
                "Wrong implementation of vertex cover");
        }
    }

    bool cover = false;
    forall(v,C) {
        if ( T.is_root(v) ∨
            ( ¬T.is_root(v) ∧ ¬C.member(T.parent(v)) ) ) {
            forall_children(w,v) {
                if ( C.member(w) ) {

```

```

    cover = true;
    break;
}
if( !cover ) {
    error_handler(1,
        "Wrong implementation of minimal vertex cover");
}
}
}
```

Lemma 6.37. *The algorithm for finding a minimum vertex cover of a rooted tree runs in $O(n \log n)$ time using $O(n)$ additional space, where n is the number of nodes in the tree.*

Proof. Let $T = (V, E)$ be a rooted tree with n nodes. Finding a maximum independent set $I \subseteq V$ takes $O(n \log n)$ time using $O(n)$ additional space, by Lemma 6.24. Further, $O(n)$ insertions in an initially empty set of nodes are made, corresponding to the nodes in $C = V \setminus I \subseteq V$. Therefore, the algorithm runs in $O(n \log n)$ time using $O(n)$ additional space. The double-check of minimal vertex cover also runs in $O(n \log n)$ time using $O(n)$ additional space. \square

Remark 6.38. The tree minimum vertex cover algorithm can also be implemented to run in $O(n)$ time, where n is the number of nodes in the tree, while still using $O(n)$ additional space. See Exercise 6.6.

Minimum Vertex Cover of a Graph

The problem of determining whether an undirected graph $G = (V, E)$ with n vertices has a vertex cover with k or less vertices, for a fixed integer k with $0 \leq k \leq n$, also belongs to the class of NP-complete problems, meaning that all known algorithms for solving the minimum vertex cover problem upon general graphs (that is, graphs without any restriction on any graph parameter) take time exponential in the size of the graphs, and that it is highly unlikely that such an algorithm will be found which takes time polynomial in the size of the graph.

Finding a minimum vertex cover of an undirected graph can be reduced to the problem of finding a maximum independent set. As a matter of fact, a minimum vertex cover of an undirected graph can be obtained by first finding a maximum independent set of the graph, and then collecting all those vertices of the graph which do not belong to the maximum independent set.

The following algorithm finds a minimum vertex cover $C \subseteq V$ of an undirected graph $G = (V, E)$, implementing the previous procedure.

339a \langle vertex cover 337a $\rangle + \equiv$

```

void minimum_vertex_cover(
    const GRAPH<string,string>& G,
    set<node>& C)
{
    C.clear();
    if ( G.number_of_nodes() == 1 ) {
        C.insert(G.first_node());
    } else {
        set<node> I;
        maximum_independent_set(G,I);
        node v;
        forall_nodes(v,G) {
            if ( !I.member(v) ) C.insert(v);
        }
        double_check_vertex_cover(G,C);
    }
}
```

The following double-check of the minimal vertex cover, although being redundant, gives some reassurance of the correctness of the implementation. It verifies that the nodes of the set C constitute a vertex cover, by double-checking that $v \in C$ or $w \in C$ for all edges $(v, w) \in E$, and it also verifies that the vertex cover C is minimal, by double-checking that all vertices $v \in C$ are adjacent to some vertex $w \notin C$.

339b \langle double-check vertex cover 337b $\rangle + \equiv$

```

void double_check_vertex_cover(
    const GRAPH<string,string>& G,
    const set<node>& C)
{
    if ( G.number_of_nodes() == 1 ) return;

    edge e;
    forall_edges(e,G) {
        if ( !C.member(G.source(e)) & !C.member(G.target(e)) ) {
            error_handler(1,"Wrong implementation of vertex cover");
        }
    }

    bool cover = false;
    node v,w;
    forall(v,C) {
        forall_adj_nodes(w,v) {
            if ( !C.member(w) ) {
                cover = true;
                break;
            }
        }
        if ( !cover ) {
            error_handler(1,

```

```

    "Wrong implementation of minimal vertex cover");
} } }
```

Interactive Demonstration of Minimum Vertex Cover

The algorithm for finding a minimum vertex cover of a tree is integrated next in the interactive demonstration of graph algorithms. A simple checker for a minimum vertex cover that provides some visual reassurance consists of highlighting the nodes in the minimum vertex cover as a subgraph of the given tree.

340a

```

⟨demo vertex cover 340a⟩≡
void gw_tree_minimum_vertex_cover(
    GraphWin& gw)
{
    GRAPH<string,string>& G = gw.get_graph();
    TREE<string,string> T(G);
    set<node> C;
    minimum_vertex_cover(T,C);

    gw.save_all_attributes();
    node v;
    forall(v,C)
        gw.set_color(v,blue);
    gw.wait();
    gw.restore_all_attributes();
}
```

The algorithm for finding a minimum vertex cover of a graph is also integrated next in the interactive demonstration of graph algorithms. Again, a simple checker for minimum vertex cover that provides some visual reassurance consists of highlighting the vertices in the minimum vertex cover as a subgraph of the given graph.

340b

```

⟨demo vertex cover 340a⟩+≡
void gw_minimum_vertex_cover(
    GraphWin& gw)
{
    GRAPH<string,string>& G = gw.get_graph();
    Make_Bidirected(G);
    gw.update_graph();

    set<node> C;
    minimum_vertex_cover(G,C);

    gw.save_all_attributes();
    node v;
```

```

forall(v,C)
    gw.set_color(v,blue);
    gw.wait();
    gw.restore_all_attributes();
}

```

6.4 Applications

Clique, independent set, and vertex cover algorithms find application in the comparison of structures described by trees or graphs. In particular, finding maximal and maximum cliques is the basis of the algorithms discussed in Chap. 7 for finding maximal and maximum common subgraph isomorphisms between two unordered graphs.

The application of maximum clique algorithms to the multiple alignment of nucleic acid or amino acid (protein) sequences in computational molecular biology is discussed next. See the bibliographic notes below for further applications of clique and independent set algorithms.

Recall from Sect. 4.4 that a ribonucleic acid molecule is composed of four types of nucleotides. Protein molecules in animals are instead composed of 20 types of amino acids, also called *residues*. These are: alanine (A), cysteine (C), aspartic acid (D), glutamic acid (E), phenylalanine (F), glycine (G), histidine (H), isoleucine (I), lysine (K), leucine (L), methionine (M), asparagine (N), proline (P), glutamine (Q), arginine (R), serine (S), threonine (T), valine (V), tryptophan (W), and tyrosine (Y). Protein molecules in plants are composed of about 100 types of residues.

Protein molecules are composed of one or more sequences of residues in a specific order, determined by the sequence of nucleotides in the gene encoding the protein.

Example 6.39. Six sequences of 60 residues each, extracted from the Protein Data Bank [38] and corresponding to the hemoglobin protein of different animal species, are shown in Fig. 6.15. The hemoglobin protein consists of four sequences of about 140 residues each.

The edit distance between two protein sequences is the cost of a least-cost sequence of elementary edit operations, such as deletion,

```
MVHLTPEEKSAVTALWGKVNVDEVGGEALGRLLVVYPWTQRFFESFGDLSTPDAVMGNPK
VQLSGEEKAAVLALWDKVNEEVGGEALGRLLVVYPWTQRFFDSFGDLSNPGAVMGNPKV
MVLSPADKTNVKAAGKVGAGAHAGEYGAEALEMFLSFPTTKTYFPHFDSLHGSAQVKGHG
MVLSAADKTNVKAASWKSKVGGHAGEYGAEALEMFLGFPTTCKTYFPHFDSLHGSAQVKAHG
VLSEGEWQLVLHVWAKVEADVGAGHQDILIRLFKSHPETLEKDRFKHLKTEAEMKASED
PIVDTGSVAPLSAAEKTICKRISAWAPVYSTYETSGVDILVKFFTSTPAAQEFPKFGLTT
```

Fig. 6.15. Sample protein sequences.

substitution, and insertion of residues, that allows the transformation of one protein sequence into another. An alignment of two protein sequences is an alternative representation of the transformation of one protein sequence into another.

An alignment of two protein sequences is obtained by first inserting dashes, either into or at the end of the sequences, in such a way that the resulting sequences have the same length, and then placing the resulting sequences one on top of the other, in such a way that every residue or dash in either sequence is opposite to a unique residue or dash in the other sequence. Then, a dash in the sequence at the top represents the insertion of a residue at the same position in the sequence at the bottom, and a dash in the sequence at the bottom represents the deletion of a residue at the same position from the sequence at the top. The absence of dashes at a particular position in both sequences indicates a substitution of the residues, if the residues at that position differ.

```
VQLSGEEKAAVLALWDKVNEE--EVGGEALGRLLVVYPWTQRFFDSFGDLSNPGAVMGNPKV
MVLSPADKTNVKAAGKVGAGAHAGEYGAEALEMFLSFPTTCKTYFPHFDSLHGSAQVKGHG--
```

Fig. 6.16. An alignment of two protein sequences.

Example 6.40. An alignment of the second and the third protein sequences of Fig. 6.15, obtained with the CLUSTAL W program [324], is shown in Fig. 6.16. The dashes in the second sequence (shown on top of the third sequence) represent the insertion of residues A and G into the third sequence, and the dashes at the end of the third sequence represent the deletion of residues K and V from the second sequence. Further, 38 out of the other 58 positions represent substi-

tutions of different residues, and only 20 positions represent identical substitutions.

Alignment is an important form of protein sequence comparison. The biological significance of sequence alignment lies in the fact that, in deoxyribonucleic acid, ribonucleic acid, or amino acid sequences, high sequence similarity usually implies functional or structural similarity.

The alignment of multiple protein sequences, although being a natural generalization of protein sequence alignment, has an even more profound biological significance. Multiple sequence alignment may reveal evolutionary history, reveal common two-dimensional and three-dimensional molecular structure, and suggest common biological function.

A multiple alignment of several protein sequences is obtained by first inserting dashes, either into or at the end of the sequences, in such a way that the resulting sequences have the same length, and then placing the resulting sequences one on top of the other, in such a way that every residue or dash in either sequence is opposite to a unique residue or dash in every other sequence.

```
-----MVHLTPEEKS A VTALWGKV N--VDEVGGEALGRLLVV PWTQRFFESFGDLSTPD AVMGNPK-
-----VQLS GEEKA VAVL ALWD KV N--EEEVGGEALGRLLVV PWTQRFFDSFGDLSNP GAVMGNPKV
-----MVLSPADK TNVKA A WGKV GA HAGEYGA EALER MF LS FPTT KTYF PHF -DLSHGSAQVK GHG-
-----MVL SAA DKTNVKA A WSKV GGHAGEYGA EALER MF LGFPTT KTYF PHF -DLSHGSAQVK AHG-
-----VLSEGEW QLV LHV WA KVEADV AGH QDIL IRLFK SHP ETLEKFDRFKHLK TEAEMKASED-
PIVDTGSVAPLSA A EKTKIRSAWA PVYST YETSGVDI LVKFFTST PAAQEFPKFKGLTT-----
```

Fig. 6.17. A multiple alignment of six protein sequences.

Example 6.41. A multiple alignment of the protein sequences from Fig. 6.15, also obtained with the CLUSTAL W program [324], is shown in Fig. 6.17. Only nine out of the 60 positions represent identical substitutions in all of the protein sequences, that is, residues common to all six sequences.

Most protein sequence alignment algorithms run in $O(n^2)$ time, where n is the number of residues in each of the sequences. By extension to several sequences, the multiple alignment of $k \geq 2$ protein

sequences takes $O(n^k)$ time. That is, the time taken to compute an alignment grows exponentially with the number of sequences to be aligned.

However, since there are $\binom{k}{2} = k(k - 1)/2$ pairwise alignments of $k \geq 2$ protein sequences, and it holds that $O(k^2 n^2) \subseteq O(n^k)$ if $k \leq \log n$, it follows that an approach to multiple sequence alignment based on combining the results of all pairwise alignments could improve the $O(n^k)$ time bound for all practical purposes, that is, unless the number of protein sequences to be aligned exceeds the logarithm of the number of residues in each of the sequences.

A simple and natural way to combine alignments of protein sequences is the following. Given the $k(k - 1)/2$ pairwise alignments of $k \geq 2$ protein sequences with n residues each, their *alignment graph* is a k -partite undirected graph with nk vertices and at most $nk(k - 1)/2$ edges. There is a vertex in the alignment graph for each residue in each sequence, and for each pair of aligned residues in each of the pairwise sequence alignments, there is an edge in the alignment graph between the corresponding vertices.

Now, a multiple sequence alignment is given by the maximum cliques in the alignment graph of the sequences.

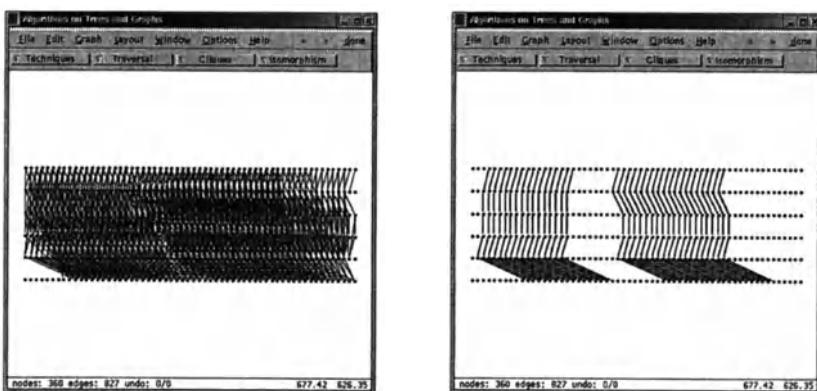


Fig. 6.18. Maximum cliques in the alignment graph for the pairwise alignments of the protein sequences of Fig. 6.15. Only maximum clique edges between consecutive layers in the alignment graph are shown, for clarity. All maximum clique edges are also shown independently, to the right.

Example 6.42. The alignment graph of the protein sequences given in Fig. 6.15 is shown in Fig. 6.16. All pairwise alignments of the protein sequences were obtained with the CLUSTAL W multiple alignment program [324]. The alignment graph has $360 = 60 \cdot 6$ vertices and $827 \leq 900 = 60 \cdot 6 \cdot 5/2$ edges. There are 144 maximal cliques in the alignment graph, only 36 of which are also maximum cliques.

The multiple sequence alignment based on maximum cliques in the alignment graph of the sequences, complements multiple sequence alignments obtained by other methods by emphasizing those residues which are aligned in a consistent way in all of the pairwise alignments of the sequences.

```
-----MVHLTPEEKSAVTALWGKVNVDEVGGEALGRLLVVYPWTQRFFESFGDLSTPDAMGNPK-
||| ||||| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
-----VQLSGEEKAAVLALWDKVN--EEEVGGAEALGRLLVVYPWTQRFFDSFGDLSNPGAVMGNPKV
||| ||||| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
-----MVLSPADKTNVKAAGWKVGAHAGEYGAEEALERMFLLSFPTTKTYFPFH-DLSHGSAQVKGHG-
||| ||||| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
-----MVLSAADKTNVKAAWSKVGGHAGEYGAEEALERMFLLGFPTTKTYFPFH-DLSHGSAQVKAHG-
||| ||||| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
-----VLSEGEWQLVLHVWAKVEADVAGHGQDILIRLFKSHPETLEKFDRFKHLKTEAMKASED-
||| ||||| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
-----PIVDTGSVAPLSAAEKTKIRSAWAPVYSTYETSGVDILVKFFTSTPAAQEFPPFKGLTT-----
```

Fig. 6.19. Multiple sequence alignment for the protein sequences of Fig. 6.15, based on the maximum cliques in the alignment graph of Fig. 6.18.

Example 6.43. The multiple sequence alignment based on the maximum cliques in the alignment graph of Fig. 6.18, corresponding to the protein sequences of Fig. 6.15, is shown in Fig. 6.19. Vertical bars between residues in consecutive protein sequences represent maximum clique edges in the alignment graph, that is, residues that are aligned in a consistent way in all of the pairwise alignments of the sequences. The maximum clique edges in the alignment graph induce a unique padding of the sequences with initial and final dashes, but a nonunique padding of the sequences with inner dashes. The padding shown in Fig. 6.19 is identical to the multiple alignment shown in Fig. 6.17.

Summary

The related problems of finding maximal and maximum cliques, independent sets, and vertex covers of an undirected graph were addressed in this chapter. Simple algorithms are given in detail for enumerating all maximal cliques and finding a maximum clique in an undirected graph. The dual problem of finding a maximum independent set in an undirected graph is solved by finding a maximum clique in the complement of the graph, and a simpler algorithm is given for finding a maximum independent set in a tree. Most of these algorithms are based on the backtracking and branch-and-bound techniques reviewed in Sect. 2.4, while the algorithm for finding a maximum independent set in a tree is based on the methods of tree traversal discussed in Chap. 3. The algorithms for finding a maximum independent set are also used for solving the related problem of finding a minimum vertex cover, in both trees and undirected graphs. Further, the application of maximum cliques to the multiple alignment of protein sequences in computational molecular biology, is also discussed in detail.

Bibliographic Notes

The exponential relation between the maximum number of maximal cliques and the number of vertices in an undirected graph was established in [240].

The backtracking algorithm for enumerating all maximal cliques of an undirected graph is based on the description given in [199] of the branch-and-bound algorithm of [53]. See also [5, 14, 173, 242]. The branch-and-bound algorithm for finding a maximum clique of an undirected graph is based on [66, 257]. Further algorithms for finding maximal and maximum cliques include [17, 18, 19, 20, 125, 186, 254, 372]. See also [51, 172] and [203, Sect. 4.3 and 4.6.3].

The algorithm for finding a maximum independent set of a tree is based on [304, Sect. 8.5.2]. See also [67, 174, 280, 281, 282], and see [27, 168, 181, 275, 322, 328] for maximal and maximum independent set enumeration algorithms upon general graphs. Algorithms for finding a minimum vertex cover include [21, 64, 73, 97, 246].

The application of maximal cliques to multiple sequence alignment is based on [304, Sect. 8.7.8]. See also [135, 277, 289, 349, 351, 352]. A comprehensive comparison of different approaches and programs for multiple sequence alignment can be found in [159, 225, 325]. See also [145, Chap. 14].

Review Problems

- 6.1** Find all nontrivial cliques, all maximal cliques, and all maximum cliques of the graph in Fig. 6.20.

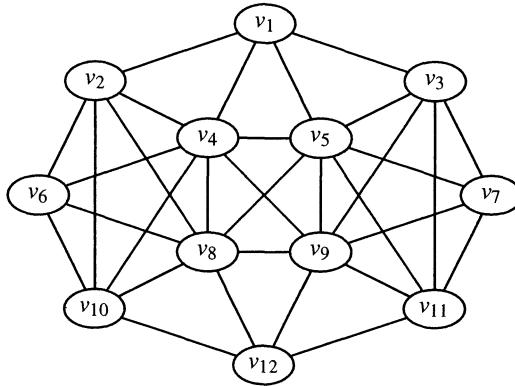


Fig. 6.20. Undirected graph for problems 6.1–6.4.

- 6.2** Find all maximal independent sets and all maximum independent sets of the graph in Fig. 6.20.

- 6.3** Give the complement of the graph in Fig. 6.20, and find all maximal cliques and all maximum cliques of the graph complement.

- 6.4** Find all maximal vertex covers and all maximum vertex covers of the graph in Fig. 6.20.

- 6.5** Give the number of maximal independent sets of the complete bipartite graph $K_{p,q}$ on $p+q$ vertices, as well as the number of vertices in each of them.

6.6 Recall from Chap. 2 that a vertex coloring of an undirected graph is an assignment of a color to each vertex of the graph, such that adjacent vertices are assigned different colors. Recall also that the chromatic number $\chi(G)$ of an undirected graph G is the minimum number of different colors required for a vertex coloring of G . Show that $\chi(G) \geq \omega(G)$, and that $\chi(G) \geq \lceil n/\beta(G) \rceil$.

Exercises

6.1 Modify the maximal clique enumeration algorithm to collect the maximal cliques in a rooted tree, as shown in Fig. 6.6, instead of a list of sets of vertices. Use the simple tree layout algorithm of Sect. 3.5 for the visualization of the search tree of maximal cliques.

6.2 In the maximal clique enumeration algorithm, the generation of duplicate cliques is avoided by only considering those candidate vertices which are greater than the vertices in the current clique, according to the order on the vertices fixed by the representation of the graph. However, the particular representation of the graph may have a significant influence on the performance of the algorithm. Some natural orderings of the vertices of an undirected graph include the following:

- Vertices are ordered from lowest to highest degree.
- Each of the vertices has the highest degree in the subgraph induced by the previous vertices and the vertex of highest degree.
- Vertices are ordered from highest to lowest degree.
- Each of the vertices has the lowest degree in the subgraph induced by the previous vertices and the vertex of lowest degree.
- A random order is imposed on the vertices of the graph.

Perform experiments on random graphs to determine the effect of the previous vertex orderings on the performance of the maximal clique enumeration algorithm.

6.3 A benchmark for maximum cliques was set up for the Second DIMACS Implementation Challenge [172] and is available from

`ftp://dimacs.rutgers.edu/pub/challenge/graph/`

Apply the branch-and-bound maximum clique algorithm to the benchmark graphs.

6.4 The $O(n \log n)$ time taken by the tree maximum independent set algorithm upon a tree with n nodes, comes from the $O(\log n)$ time taken to insert a node in a set of $O(n)$ nodes. However, the fact that the potential elements of the set are known beforehand, allows using a more efficient set data structure. Give an alternative implementation of the tree maximum independent set algorithm running in $O(n)$ time using $O(n)$ additional space, where n is the number of nodes in the tree.

6.5 If the root of a tree belongs to an independent set of the tree, then none of the children of the root may belong to the independent set. Otherwise, if the root does not belong to the independent set, any independent sets of the subtrees rooted at each of the children of the root of the tree, may be combined into an independent set of the tree. The size of a maximum independent set $I \subseteq V$ of a rooted tree $T = (V, E)$ is thus described by the recurrence relations

$$\begin{cases} \beta(T[v]) = \max(\beta'(T[v]), 1 + \sum_{(v,w) \in E} \beta'(T[w])) \\ \beta'(T[v]) = \sum_{(v,w) \in E} \beta(T[w]) \end{cases}$$

for all nonleaves $v \in V$ and, for all leaves $w \in V$,

$$\begin{cases} \beta(T[w]) = 1 \\ \beta'(T[w]) = 0 \end{cases}$$

where $T[v]$ denotes the subtree of T rooted at node $v \in V$. Give an algorithm for computing the independence number $\beta(T) = \beta(T[\text{root}[T]])$ of a rooted tree T . Give also the time and space complexity of the algorithm.

6.6 The $O(n \log n)$ time taken by the tree minimum vertex cover algorithm upon a tree with n nodes, comes from the $O(\log n)$ time taken to insert a node in a set of $O(n)$ nodes. However, the fact that the potential elements of the set are known beforehand, allows using a more efficient set data structure. Give an alternative implementation of the tree minimum vertex cover algorithm running in $O(n)$ time using $O(n)$ additional space, where n is the number of nodes in the tree.

6.7 Adapt the branch-and-bound maximum clique algorithm, using the relationship between the chromatic number and the clique number of an undirected graph of Problem 6.6 as an alternative upper bound on the size of a maximum clique.

6.8 Perform experiments on random graphs to compare the efficiency of the branch-and-bound maximum clique algorithm, using the upper bound on the size of a maximum clique based on the size of the clique being extended and the number of candidate vertices with which it can be extended, on the one hand and using, on the other hand, the upper bound of Problem 6.6 and Exercise 6.7.

6.9 Recall from Sect. 6.4 that the alignment graph of k sequences with n residues each is an undirected graph with nk vertices and at most $nk(k - 1)/2$ edges, where there is a vertex in the alignment graph for each residue in each sequence, and for each pair of aligned residues in each of the pairwise sequence alignments, there is an edge in the alignment graph between the corresponding vertices. Give an algorithm for building the alignment graph of $k \geq 2$ sequences with n residues each, given their $k(k - 1)/2$ pairwise alignments.

6.10 Both the backtracking maximal clique enumeration algorithm and the branch-and-bound maximum clique algorithm take general graphs as input. In the particular case of an alignment graph of $k \geq 2$ sequences, though, the maximal cliques of interest are those having exactly k vertices, which are thus (as long as there is such a k -vertex clique in the alignment graph) maximum cliques. Adapt the branch-and-bound maximum clique algorithm in order to find all k -cliques in an alignment graph of $k \geq 2$ sequences.

7. Graph Isomorphism

A straightforward enumerative algorithm might require 40 years of running time on a very high speed computer in order to compare two 15-node graphs.

—Stephen H. Unger [332]

The concept of *graph isomorphism* lies (explicitly or implicitly) behind almost any discussion of graphs, to the extent that it can be regarded as *the* fundamental concept of graph theory. In particular, the automorphism group of a graph provides much information about symmetries in the graph. The related problems of subgraph isomorphism and maximum common subgraph isomorphism generalize pattern matching in strings and trees to graphs, and find application whenever structures described by graphs need to be compared.

Combinatorial algorithms are discussed in this chapter for testing graph isomorphism, finding the automorphism group of a graph, finding one or all isomorphisms of a graph as a subgraph of another graph, and finding one or all maximal or maximum common subgraph isomorphisms between two graphs. These algorithms use the techniques given in Chap. 2, and some of them also build upon the algorithms discussed in Chap. 6 for finding maximal and maximum cliques.

7.1 Graph Isomorphism

The question of equality or identity is fundamental for the objects of any universe of mathematical discourse. For example, a pair of fractions (which may look different) are the same if their difference is zero, two sets (which may be represented in quite different ways) are

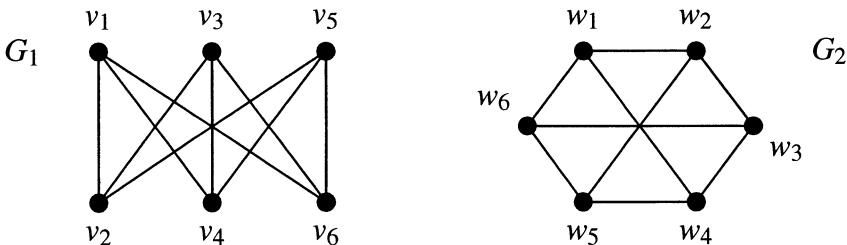
the same if they contain the same elements, etc. In the same vein, a pair of graphs may also look different but actually have the same structure.

Two graphs are isomorphic if there is a bijective correspondence between their vertex sets which preserves and reflects adjacencies—that is, such that two vertices of a graph are adjacent if and only if the corresponding vertices of the other graph are adjacent.

Definition 7.1. Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic, denoted by $G_1 \cong G_2$, if there is a bijection $M \subseteq V_1 \times V_2$ such that, for every pair of vertices $v_i, v_j \in V_1$ and $w_i, w_j \in V_2$ with $(v_i, w_i) \in M$ and $(v_j, w_j) \in M$, $(v_i, v_j) \in E_1$ if and only if $(w_i, w_j) \in E_2$. In such a case, M is a graph isomorphism of G_1 to G_2 .

Isomorphism expresses what, in less formal language, is meant when two graphs are said to be the same graph. Two isomorphic graphs may be depicted in such a way that they look very different—they are differently labeled, perhaps also differently drawn, and it is for this reason that they look different.

Example 7.2. The following two graphs are isomorphic, and $M = \{(v_1, w_1), (v_2, w_2), (v_3, w_3), (v_4, w_4), (v_5, w_5), (v_6, w_6)\}$ is a graph isomorphism of G_1 to G_2 .



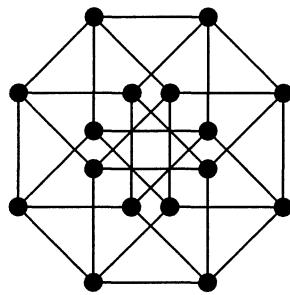
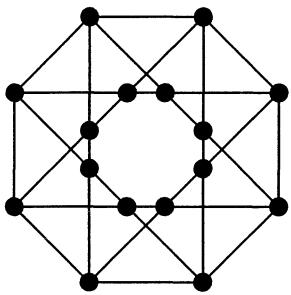
Nonisomorphism of graphs is not usually hard to prove, because several *invariants* or necessary conditions for isomorphism are not difficult to compute. These are properties that do not depend on the presentation or labeling of a graph. For instance, two graphs cannot be isomorphic if they differ in their order, size, or degree sequence.

Remark 7.3. Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ with $V_1 = \{u_1, \dots, u_n\}$ and $V_2 = \{v_1, \dots, v_n\}$, a necessary condition for

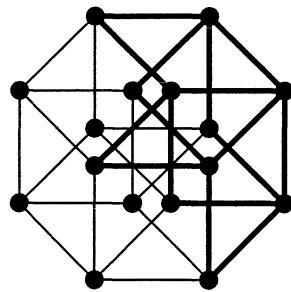
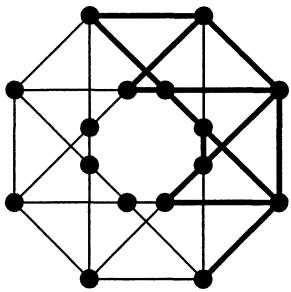
$G_1 \cong G_2$ is that the multisets $\{\Gamma(u_i) \mid 1 \leq i \leq n\}$ and $\{\Gamma(v_i) \mid 1 \leq i \leq n\}$ be equal.

On the other hand, invariants are not sufficient conditions for isomorphism and nothing can be concluded about two graphs which share an invariant.

Example 7.4. The following two graphs are not isomorphic, although they have the same number of vertices, the same number of edges, and are both 4-regular.



There are, in fact, only 10 paths of length 2 starting from any vertex (and only five vertices at distance 2 of any given vertex) of the graph to the left-hand side, but 12 paths of length 2 starting from any vertex (and six vertices at distance 2 of any given vertex) of the graph to the right-hand side:



Isomorphism of graphs is usually much harder to prove than non-isomorphism of graphs, because all known *certificates* or necessary and sufficient conditions for graph isomorphism are as difficult to compute as graph isomorphism itself. The graph isomorphism problem is actually not only of practical interest but also of theoretical

relevance. From a complexity-theoretical point of view, graph isomorphism is one of the few NP problems believed neither to be in P nor to be NP-complete.

7.1.1 An Algorithm for Graph Isomorphism

The general backtracking scheme presented in Sect. 2.2 can be instantiated to yield a simple algorithm for testing graph isomorphism based on the next result, which follows from Definition 7.1.

Notice first that two graphs are isomorphic if there is a bijective correspondence between their vertex sets such that two vertices of a graph are adjacent if and only if the corresponding vertices of the other graph are adjacent, while in most applications further information is attached to vertices and edges in the form of vertex and edge labels. Then, two labeled graphs are isomorphic if the underlying graphs are isomorphic and, furthermore, corresponding vertices and edges share the same label.

Recall that for a LEDA graph $G = (V, E)$, the label of a vertex $v \in V$ is denoted by $G[v]$ and the label of an edge $(v, w) \in E$ is denoted by $G[v, w]$. Unlabeled graphs are dealt with in the following algorithms as labeled graphs with all vertex and edge labels set to *nil*, that is, undefined.

Lemma 7.5. *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs, let $V \subseteq V_1$ and $W \subseteq V_2$, and let $M \subseteq V \times W$ be an isomorphism of the subgraph of G_1 induced by V to the subgraph of G_2 induced by W . For any vertices $v \in V_1 \setminus V$ and $w \in V_2 \setminus W$, $M \cup \{(v, w)\}$ is an isomorphism of the subgraph of G_1 induced by $V \cup \{v\}$ to the subgraph of G_2 induced by $W \cup \{w\}$ if and only if the following conditions*

- $(x, v) \in E_1$ if and only if $(y, w) \in E_2$
- $(v, x) \in E_1$ if and only if $(w, y) \in E_2$

hold for all vertices $x \in V$ and $y \in W$ with $(x, y) \in M$.

Proof. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs, let $V \subseteq V_1$ and $W \subseteq V_2$, and let $M \subseteq V \times W$ be an isomorphism of the subgraph of G_1 induced by V to the subgraph of G_2 induced by W . See Fig. 7.1. Let also $v \in V_1 \setminus V$ and $w \in V_2 \setminus W$.

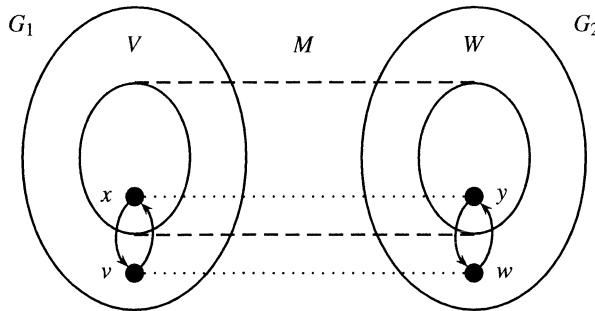


Fig. 7.1. Proof of Lemma 7.5

Suppose $(x, v) \in E_1$ if and only if $(y, w) \in E_2$ and $(v, x) \in E_1$ if and only if $(w, y) \in E_2$, for all $(x, y) \in M$. Then, by Definition 7.1, $M \cup \{(v, w)\}$ is an isomorphism of the subgraph of G_1 induced by $V \cup \{v\}$ to the subgraph of G_2 induced by $W \cup \{w\}$.

Conversely, let $M \cup \{(v, w)\}$ be an isomorphism of the subgraph of G_1 induced by $V \cup \{v\}$ to the subgraph of G_2 induced by $W \cup \{w\}$, and let $x \in V$ and $y \in W$ with $(x, y) \in M$. Since $(u, v), (x, y) \in M \cup \{(v, w)\}$, by Definition 7.1 it holds that $(x, v) \in E_1$ if and only if $(y, w) \in E_2$ and $(v, x) \in E_1$ if and only if $(w, y) \in E_2$. \square

The following backtracking algorithm for graph isomorphism extends an (initially empty) vertex mapping M in all possible ways, in order to enumerate all graph isomorphisms of a graph G_1 to a graph G_2 . Although the enumeration of all graph isomorphisms is sufficient but not necessary for testing whether or not two graphs are isomorphic, it will also be used in Sect. 7.2 for determining the automorphism group of a graph.

Each graph isomorphism mapping M is represented by an array of vertices (of G_2) indexed by the vertices of G_1 .

```
<graph isomorphism 355>≡
void graph_isomorphism(
    const GRAPH<string,string>& G1,
    const GRAPH<string,string>& G2,
    node_array<node>& M,
    list<node_array<node>>& L)
{
    L.clear();
    if( G1.number_of_nodes() ≠ G2.number_of_nodes() ∨
        G1.number_of_edges() ≠ G2.number_of_edges() ) return;
```

```

⟨set up adjacency matrices 356a⟩
node v = G1.first_node();
extend_graph_isomorphism(G1,G2,A1,A2,M,v,L);
}

```

Recall that graphs are represented in LEDA by adjacency lists. Adjacency matrices A_1 and A_2 for graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, respectively, will also be used in order to test adjacency of a vertex pair in $O(1)$ time: given two vertices $v_i, v_j \in V_1$, there is an edge $(v_i, v_j) \in E_1$ if and only if $A(v_i, v_j) \neq \text{nil}$, and the same holds for G_2 .

356a ⟨set up adjacency matrices 356a⟩≡

```

node_matrix<edge> A1(G1,nil);
node_matrix<edge> A2(G2,nil);
{ edge e;
forall_edges(e,G1)
  A1(G1.source(e),G1.target(e)) = e;
forall_edges(e,G2)
  A2(G2.source(e),G2.target(e)) = e;
}

```

The actual extension of a mapping is done by the following recursive procedure, which extends a vertex mapping $M \subseteq V \times W$ of the subgraph of G_1 induced by V to the subgraph of G_2 induced by W by mapping vertex $v \in V_1 \setminus V$ to each vertex $w \in V_2 \setminus W$ in turn, that is, to each vertex which has not been mapped to yet, making a recursive call upon the successor of vertex v in G_1 whenever $M \cup \{(u, w)\}$ is a graph isomorphism of the subgraph of G_1 induced by $V \cup \{v\}$ to the subgraph of G_2 induced by $W \cup \{w\}$.

All such vertex mappings M which are defined for all vertices of G_1 , that is, all isomorphisms of graph G_1 to graph G_2 , are collected in a list L of vertex mappings.

356b ⟨graph isomorphism 355⟩+≡

```

bool extend_graph_isomorphism(
  const GRAPH<string,string>& G1,
  const GRAPH<string,string>& G2,
  const node_matrix<edge>& A1,
  const node_matrix<edge>& A2,
  node_array<node> M,
  node v,
  list<node_array<node> >& L)
{
  list<node> V2 = G2.all_nodes();

```

```

node w;
forall_nodes(w,G1)
  if ( M[w] ≠ nil )
    V2.remove(M[w]); // already mapped vertex
  forall_nodes(w,G2)
    if ( G1[v] ≠ G2[w] )
      V2.remove(w); // different vertex labels
  forall(w,V2) {
    if ( preserves_adjacencies(G1,G2,A1,A2,M,v,w) ) {
      M[v] = w;
      if ( v ≡ G1.last_node() ) {
        double_check_graph_isomorphism(G1,G2,A1,A2,M);
        L.append(M);
      } else {
        extend_graph_isomorphism(G1,G2,A1,A2,M,G1.succ_node(v),L);
      }
    }
  }
}

```

The following double-check of graph isomorphism, although being redundant, gives some reassurance of the correctness of the implementation. It verifies that M is a graph isomorphism of graph G_1 to graph G_2 , where the two graphs have the same number of vertices and the same number of edges, by double-checking that M is a bijection (that is, M is defined for all vertices of G_1 and G_2 , and different vertices of G_1 are mapped by M to different vertices of G_2) and that all edges of G_1 are mapped to edges of G_2 . Notice that, having double-checked that M is a bijection, it is not necessary to verify that all nonedges of G_1 are mapped by M to nonedges of G_2 .

357

```

⟨double-check graph isomorphism 357⟩≡
void double_check_graph_isomorphism(
  const GRAPH<string,string>& G1,
  const GRAPH<string,string>& G2,
  const node_matrix<edge>& A1,
  const node_matrix<edge>& A2,
  const node_array<node>& M)
{
  if ( G1.number_of_nodes() ≠ G2.number_of_nodes() ∨
       G1.number_of_edges() ≠ G2.number_of_edges() )
    error_handler(1,
      "Wrong implementation of graph isomorphism");

  node v,w;
  forall_nodes(v,G1)
    if ( M[v] ≡ nil ∨ G1[v] ≠ G2[M[v]] )
      error_handler(1,
        "Wrong implementation of graph isomorphism");

  forall_nodes(v,G1)
  forall_nodes(w,G1)

```

```

if ( (  $v \neq w \wedge (M[v] \equiv M[w] \vee$ 
       $A1(v,w) \neq \text{nil} \wedge A2(M[v],M[w]) \equiv \text{nil} ) ) \vee$ 
    (  $A1(v,w) \neq \text{nil} \wedge A2(M[v],M[w]) \equiv \text{nil} \wedge$ 
       $GI[A1(v,w)] \neq G2[A2(M[v],M[w])] ) )$ 
  error_handler(1,
    "Wrong implementation of graph isomorphism");
}

```

The following procedure implements the obvious adjacency test: M may be extended by mapping vertex $v \in V_1$ to vertex $w \in V_2$ if and only if vertices v and w have the same indegree and the same outdegree, every (already mapped) predecessor x of vertex v was mapped to a predecessor of vertex w , and every (already mapped) successor x of vertex v was mapped to a successor of vertex w . See Fig. 7.2.

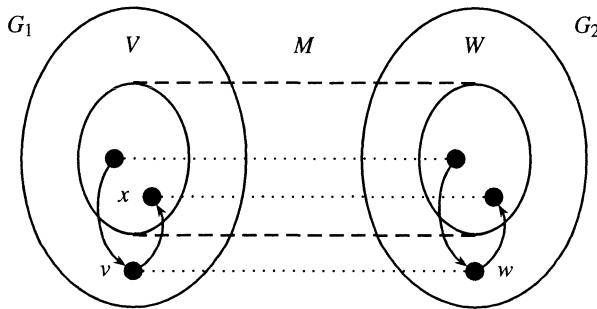


Fig. 7.2. Adjacency test for extending a graph isomorphism.

```

⟨graph isomorphism 355⟩ +≡
bool preserves_adjacencies(
  const GRAPH<string,string>& G1,
  const GRAPH<string,string>& G2,
  const node_matrix<edge>& A1,
  const node_matrix<edge>& A2,
  const node_array<node>& M,
  node v,
  node w)
{
  if (  $G1.\text{indeg}(v) \neq G2.\text{indeg}(w) \vee G1.\text{outdeg}(v) \neq G2.\text{outdeg}(w) )$ 
    return false;
  node x;
  edge e;
  forall_in_edges(e,v) {
     $x = G1.\text{opposite}(v,e);$ 
    if (  $M[x] \neq \text{nil} \wedge$ 

```

```

( A2( $M[x], w$ )  $\equiv$  nil  $\vee$  GI[e]  $\neq$  G2[A2( $M[x], w$ )] ) )
return false;
}
forall_out_edges( $e, v$ ) {
   $x = GI.opposite(v, e)$ ;
  if (  $M[x] \neq$  nil  $\wedge$  A2( $w, M[x]$ )  $\equiv$  nil ) return false;
}
return true;
}

```

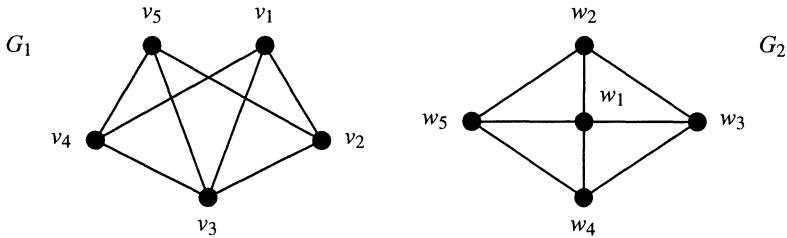


Fig. 7.3. Isomorphic graphs

Example 7.6. The undirected graphs of Fig. 7.3 are isomorphic. Let $[v_1, v_2, v_3, v_4, v_5]$ and $[w_1, w_2, w_3, w_4, w_5]$ be the order among the vertices of the graphs fixed by their representation. Then, the graph isomorphism M of G_1 to G_2 found by the backtracking algorithm for graph isomorphism is $M = \{(v_1, w_2), (v_2, w_3), (v_3, w_1), (v_4, w_5), (v_5, w_4)\}$. The sequence of calls to the recursive procedure *extend graph isomorphism* upon graphs G_1 and G_2 , until a graph isomorphism M of G_1 to G_2 is found, shown to the left of Fig. 7.4, corresponds to a preorder traversal of the search tree shown to the right of Fig. 7.4. The levels of the search tree correspond to the vertices of G_1 , and the nodes stand for vertices of G_2 to which the vertex of G_1 might be mapped to. The rightmost path from the root to a leaf corresponds to the graph isomorphism found, mapping vertices $[v_1, v_2, v_3, v_4, v_5]$ to vertices $[w_2, w_3, w_1, w_5, w_4]$, respectively. The whole search tree that corresponds to finding all isomorphic copies of G_1 in G_2 is shown in Fig. 7.5. Those leaves of the search tree representing an isomorphism of G_1 to G_2 are marked by a star.

M	v
$\{\}$	v_1
$\{(v_1, w_1)\}$	v_2
$\{(v_1, w_2)\}$	v_2
$\{(v_1, w_2), (v_2, w_1)\}$	v_3
$\{(v_1, w_2), (v_2, w_3)\}$	v_3
$\{(v_1, w_2), (v_2, w_3), (v_3, w_1)\}$	v_4
$\{(v_1, w_2), (v_2, w_3), (v_3, w_1), (v_4, w_4)\}$	v_5
$\{(v_1, w_2), (v_2, w_3), (v_3, w_1), (v_4, w_5)\}$	v_5
$\{(v_1, w_2), (v_2, w_3), (v_3, w_1), (v_4, w_5), (v_5, w_4)\}$	v_5

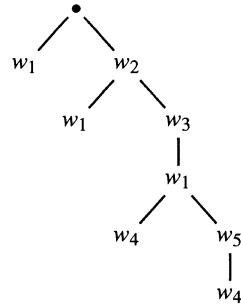


Fig. 7.4. Sequence of calls (to the left) and search tree (to the right) for graph isomorphism.

Remark 7.7. Correctness of the backtracking algorithm for graph isomorphism follows from the fact that a graph isomorphism M of an (initially empty) subgraph of a graph G_1 to an (initially empty) subgraph of a graph G_2 , is extended in all possible ways, one vertex pair at a time, until either M maps all the vertices of G_1 to all the vertices of G_2 (and $G_1 \cong G_2$) or no mapping M can be further extended (and $G_1 \not\cong G_2$).

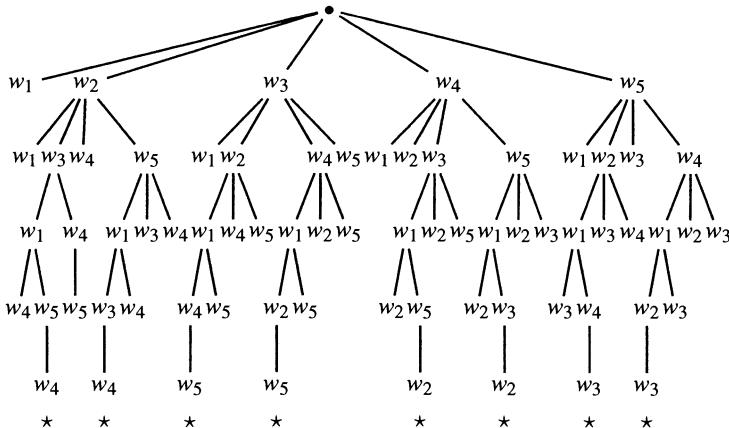


Fig. 7.5. Search tree for graph isomorphism.

Regarding computational complexity of the backtracking algorithm for graph isomorphism, the worst case arises when the graphs are complete and isomorphic. In this situation, the whole search tree

will have to be traversed and, since vertex v_1 may be mapped to each of the n vertices of G_2 , vertex v_2 may be mapped to $n - 1$ vertices (each of the n vertices of G_2 except the vertex of G_2 which vertex v_1 was mapped to), and so on, the search tree will have n nodes on the first level, $n(n - 1)$ nodes on the second level, and $n(n - 1) \cdots (n - (n - 1)) = n!$ nodes on the n -th level. Furthermore, every one of the $n!$ leaves of the search tree will correspond to a graph isomorphism.

Theorem 7.8. *The backtracking algorithm for graph isomorphism runs in worst-case $O((n + 1)!)$ time and $O(n^2)$ space, on connected graphs G_1 and G_2 with n vertices.*

Proof. Let G_1 and G_2 be two graphs with n vertices and m edges, and let $[v_1, v_2, \dots, v_n]$ be the order among the vertices of V_1 fixed by the representation of G_1 . Since vertex v_1 can, in principle, be mapped to any vertex of G_2 , there are at most $O(n)$ recursive calls to procedure *extend graph isomorphism*(G_1, G_2, M, v_1). Now, after having mapped vertex v_1 to some vertex of G_2 , vertex v_2 can, in principle, be mapped to any remaining vertex of G_2 and there are at most $O(n(n - 1))$ recursive calls to *extend graph isomorphism*(G_1, G_2, M, v_2). In general, after having mapped vertices v_1, \dots, v_{i-1} to (pairwise different) vertices of G_2 , vertex v_i can be mapped, in principle, to any of the $n - (i - 1)$ remaining vertices of G_2 , and there are at most $O(n(n - 1) \cdots (n - (i - 1))) = O(n!/(n - i)!)$ recursive calls to procedure *extend graph isomorphism*(G_1, G_2, M, v_i), with $1 \leq i \leq n$. Therefore, the total number of recursive calls to procedure *extend graph isomorphism* is at most $\sum_{i=1}^n O(n!/(n - i)!) = O(\sum_{i=1}^n n!/(n - i)!) = O((n + 1)!)$. The effort done on each recursive call to *extend graph isomorphism*(G_1, G_2, M, v) is not constant, though, but the total effort done in order to make $O(\deg(v))$ further recursive calls is at most $O(\deg(v))$. Therefore, the algorithm runs in $O((n + 1)!)$ time.

Since there is at most one nested recursive call to the procedure *extend graph isomorphism* for each vertex of G_1 , the number of nested recursive calls is at most $O(n)$ and given that for each recursive call, a local copy of mapping M is made, each nested recursive call takes $O(n)$ additional space and the algorithm runs in $O(n^2)$ space. \square

Several improvements to the backtracking algorithm for graph isomorphism will be introduced later, when discussing the more general problems of subgraph isomorphism in Sect. 7.3 and maximal common subgraph in Sect. 7.4.

7.1.2 Interactive Demonstration of Graph Isomorphism

The backtracking algorithm for graph isomorphism is integrated next in the interactive demonstration of graph algorithms. A simple checker for isomorphic graphs that provides some visual reassurance consists of redrawing one of the graphs according to the graph isomorphism found, to match the layout of the other graph.

362

```
<demo graph isomorphism 362>≡
void gw_graph_isomorphism(
    GraphWin& gw1)
{
    GRAPH<string,string>& G1 = gw1.get_graph();
    node_array<node> M(G1,nil);

    GRAPH<string,string> G2;
    GraphWin gw2(G2,500,500,"Graph Isomorphism");
    gw2.display();
    gw2.message("Enter second graph. Press done when finished");
    gw2.edit();
    gw2.del_message();

    list<node_array<node>> L;
    graph_isomorphism(G1,G2,M,L);
    panel P;
    if ( L.length() > 1 ) {

        make_proof_panel(P,
            string("There are %i graph isomorphisms",L.length()),true);
        if ( gw1.open_panel(P) ) { // proof button pressed
            gw1.save_all_attributes();
            node_array<point> pos1(G1);
            node_array<point> pos2(G2);
            forall(M,L) {
                <show graph isomorphism 363>
                panel Q;
                make_yes_no_panel(Q,"Continue",true);
                if ( gw1.open_panel(Q) ) // no button pressed
                    break;
            }
            gw1.restore_all_attributes();
        }

    } else if ( L.length() ≡ 1 ) {
```

```

make_proof_panel(P,"These graphs are isomorphic",true);
if ( gw1.open_panel(P) ) { // proof button pressed
    gw1.save_all_attributes();
    node_array<point> pos1(G1);
    node_array<point> pos2(G2);
    M = L.head();
    <show graph isomorphism 363>
    gw1.wait();
    gw1.restore_all_attributes();
}
} else {
    make_proof_panel(P,"These graphs are \\\red not isomorphic",false);
    gw1.open_panel(P);
}
}

```

Graph G_2 is redrawn according to the graph isomorphism mapping M found, to match the layout of graph G_1 , by updating the position of every vertex $M[v] \in V_2$ in the layout of G_2 to be identical with the position of vertex $v \in V_1$ in the layout of G_1 , and removing any edge bends in the layout of both graphs.

363

```

<show graph isomorphism 363>≡
gw1.get_position(pos1);
gw1.set_layout(pos1); // remove edge bends
node v;
forall_nodes(v,G1)
    pos2[M[v]] = pos1[v];
gw2.set_position(pos2);
gw2.set_layout(pos2);

```

Execution of the interactive demonstration of graph algorithms to find graph isomorphisms is illustrated in Fig. 7.6.

7.2 Graph Automorphism

The automorphism group of a graph reveals information about the structure and symmetries of the graph.

Definition 7.9. An automorphism of a graph G is a graph isomorphism between G and itself.

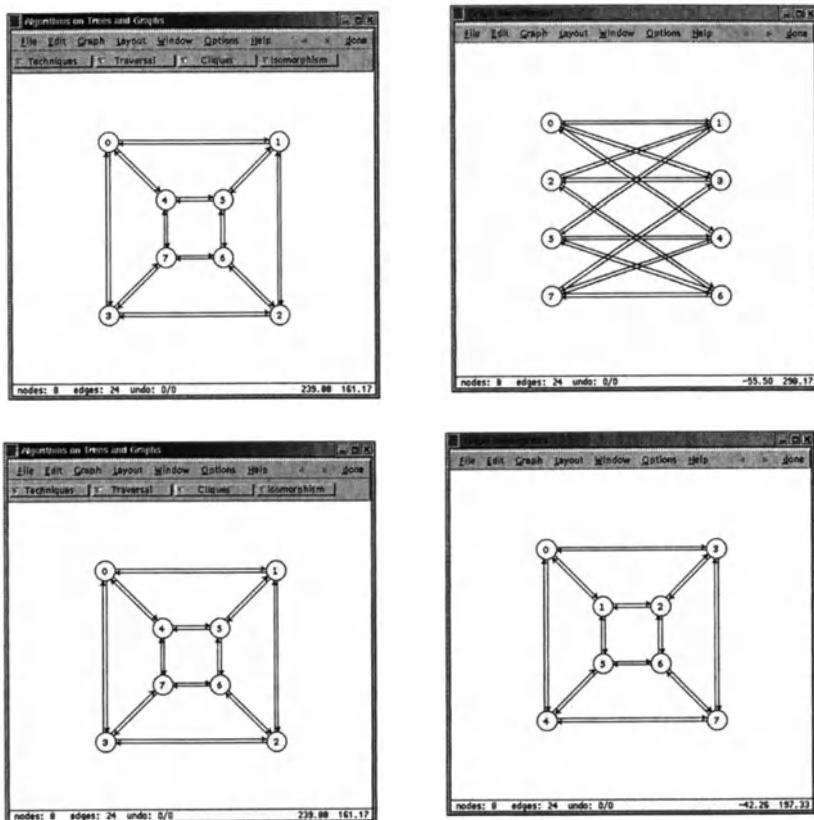


Fig. 7.6. Finding isomorphisms between two graphs.

An obvious automorphism is the identity mapping on the vertices of a graph. The inverse of an automorphism of a graph G is also an automorphism of G , and the composition of two automorphisms of G is an automorphism of G . As a matter of fact, the set of all automorphisms of a graph forms a group under the operation of composition, called the *automorphism group* of the graph.

For instance, every permutation of the vertex set of the complete graph on n vertices K_n corresponds to an automorphism of K_n and then, the automorphism group of K_n is of order $n!$. The automorphism group of the cycle graph C_n on $n \geq 3$ vertices is a group of order $2n$ consisting of n rotations and n reflections, as is the automorphism group of the wheel graph W_{n+1} with $n \geq 4$ outer vertices.

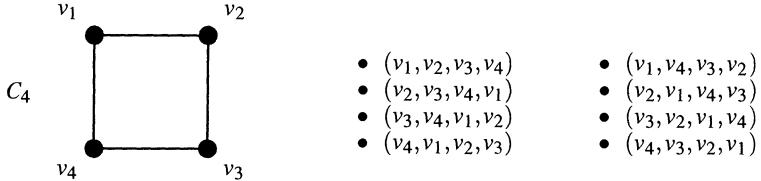


Fig. 7.7. Automorphism group of C_4 .

Example 7.10. The automorphism group of the cycle graph on four vertices C_4 is of order 8. The permutations of the vertex set shown in Fig. 7.7 correspond to the graph automorphisms of C_4 .

7.2.1 Interactive Demonstration of Graph Automorphism

The backtracking algorithm for graph isomorphism is used next for the interactive demonstration of graph automorphism. A simple checker for graph automorphism that provides some visual reassurance consists of redrawing the graph according to each of the graph automorphisms found, to match the original layout of the graph.

365

```
<demo graph automorphism 365>≡
  void gw_graph_automorphism(
    GraphWin& gw)
  {
    GRAPH<string,string>& G = gw.get_graph();
    node_array<node> M(G,nil);
    list<node_array<node>> L;
    graph_isomorphism(G,G,M,L);

    panel P;
    if( L.length() > 1 ) {

      make_proof_panel(P,
        string("The automorphism group has order %i",L.length()),true);
      if( gw.open_panel(P) ) { // proof button pressed
        node_array<point> pos1(G);
        node_array<point> pos2(G);
        node v;
        gw.save_all_attributes();
        forall(M,L) {
          gw.get_position(pos1);
          forall_nodes(v,G)
            pos2[M[v]] = pos1[v];
          gw.set_position(pos2);
          gw.set_layout(pos2);
        panel Q;
        make_yes_no_panel(Q,"Continue",true);
        if( gw.open_panel(Q) ) // no button pressed

```

```

        break;
    }
    gw.restore_all_attributes();
} } else {

make_proof_panel(P,"The automorphism group has order 1",false);
gw.open_panel(P);

}
}

```

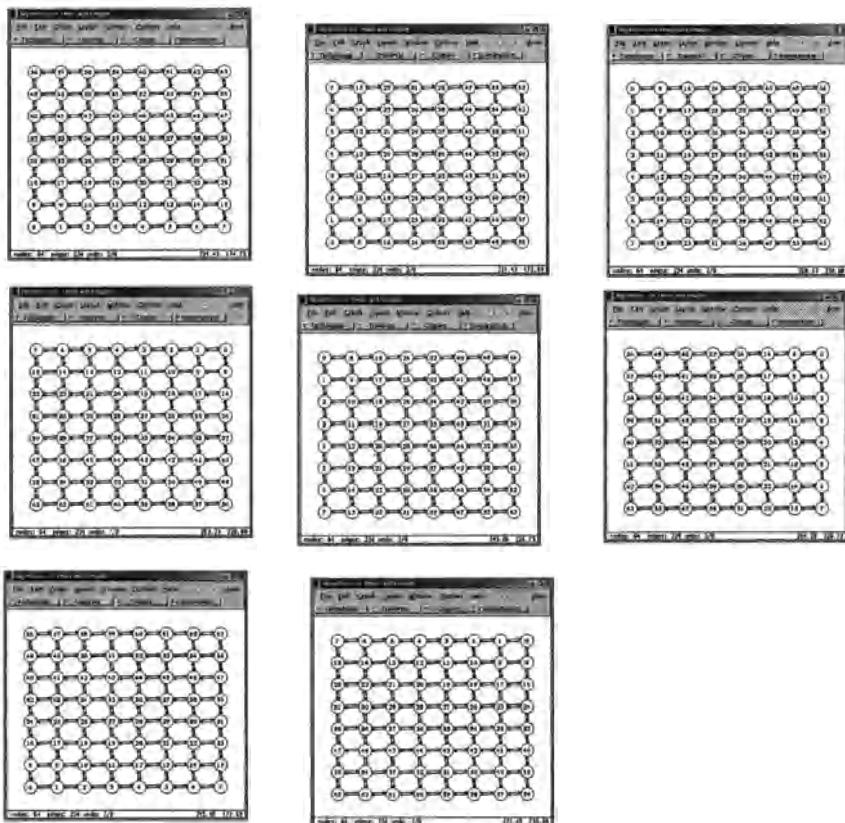


Fig. 7.8. Finding all automorphisms of a graph. The automorphism group of the grid graph on 8×8 vertices has size 8.

Execution of the interactive demonstration of graph algorithms to find all automorphisms of a graph is illustrated in Fig. 7.8.

7.3 Subgraph Isomorphism

An important generalization of graph isomorphism is known as subgraph isomorphism. The subgraph isomorphism problem is to determine whether a graph is isomorphic to a subgraph of another graph, and is a fundamental problem with a variety of applications in engineering sciences, organic chemistry, biology, and pattern matching.

Definition 7.11. A subgraph isomorphism of a graph $G_1 = (V_1, E_1)$ into a graph $G_2 = (V_2, E_2)$ is an injection $M \subseteq V_1 \times V_2$ such that, for every pair of vertices $v_i, v_j \in V_1$ and $w_i, w_j \in V_2$ with $(v_i, w_i) \in M$ and $(v_j, w_j) \in M$, $(w_i, w_j) \in E_2$ if $(v_i, v_j) \in E_1$. In such a case, M is a subgraph isomorphism of G_1 into G_2 .

The problem of determining whether or not a graph is isomorphic to a subgraph of another graph belongs to the class of NP-complete problems, meaning that all known algorithms for solving the subgraph isomorphism problem upon general graphs (that is, graphs without any restriction on any graph parameter) take time exponential in the size of the graphs, and that it is highly unlikely that such an algorithm will be found which takes time polynomial in the size of the graphs.

Most practical applications of subgraph isomorphism require not only determining whether or not a given graph is isomorphic to a subgraph of another given graph, but finding all subgraphs of a given graph which are isomorphic to another given graph. Already the number of subgraph isomorphisms of a graph into another graph can be exponential in the size of the graphs, though.

Example 7.12. There are $q!/(q-p)!$ different subgraph isomorphisms of the complete graph on p vertices $K_p = (V_1, E_1)$ into the complete graph on q vertices $K_q = (V_2, E_2)$, with $p \leq q$. (Since the graphs are complete, any injection $M \subseteq V_1 \times V_2$ will be a subgraph isomorphism of K_p into K_q , and there are $\binom{q}{p} = q!/(q-p)!/p!$ different injections of a set of p elements into a set of q elements, the complete graph on p vertices induced by each of which has $p!$ different automorphisms.) Table 7.1 gives some examples for small graphs.

Remark 7.13. Notice that in the definition of subgraph isomorphism, it is not required that $(v_i, v_j) \in E_1$ if $(w_i, w_j) \in E_2$. Under the additional

Table 7.1. Number of different subgraph isomorphisms of the complete graph on p vertices K_p into the complete graph on q vertices K_q , with $1 \leq p \leq q \leq 12$.

	K_1	K_2	K_3	K_4	K_5	K_6	K_7	K_8	K_9	K_{10}	K_{11}	K_{12}
K_1	1	2	3	4	5	6	7	8	9	10	11	12
K_2		2	6	12	20	30	42	56	72	90	110	132
K_3			6	24	60	120	210	336	504	720	990	1320
K_4				24	120	360	840	1680	3024	5040	7920	11880
K_5					120	720	2520	6720	15120	30240	55440	95040
K_6						720	5040	20160	60480	151200	332640	665280
K_7							5040	40320	181440	604800	1663200	3991680
K_8								40320	362880	1814400	6652800	19958400
K_9									362880	3628800	19958400	79833600
K_{10}										3628800	39916800	239500800
K_{11}											39916800	479001600
K_{12}												479001600

condition of not only preserving the structure of graph G_1 but also reflecting in G_1 the structure of the subgraph of G_2 induced by M , the vertex mapping M is called an induced subgraph isomorphism of G_1 into G_2 .

It follows from Definition 7.11 that if $(v, w) \in M$ in a subgraph isomorphism M of a graph G_1 into a graph G_2 , then the degree of vertex v in G_1 cannot be greater than the degree of vertex w in G_2 .

Lemma 7.14. *Let M be a subgraph isomorphism of a graph G_1 into a graph G_2 . Then, $\deg(v) \leq \deg(w)$ for all $(v, w) \in M$.*

Proof. Let $M \subseteq V_1 \times V_2$ be a subgraph isomorphism of a graph $G_1 = (V_1, E_1)$ into a graph $G_2 = (V_2, E_2)$, and let $(v, w) \in M$. Let also $\Gamma(v) = \{x \in V_1 \mid (x, v) \in E_1\} \cup \{x \in V_1 \mid (v, x) \in E_1\}$, and let $\Gamma(w) = \{y \in V_2 \mid (y, w) \in E_2\} \cup \{y \in V_2 \mid (w, y) \in E_2\}$. Then, $\deg(v) = |\Gamma(v)|$ and $\deg(w) = |\Gamma(w)|$. By Definition 7.11, for all $(x, y) \in M$ with $x \in \Gamma(v)$ it must be true that $y \in \Gamma(w)$. Then, M being an injection, it must be true that $|\Gamma(v)| \leq |\Gamma(w)|$. Therefore, $\deg(v) \leq \deg(w)$. \square

7.3.1 An Algorithm for Subgraph Isomorphism

The general backtracking scheme presented in Sect. 2.2 can also be instantiated to yield a simple algorithm for subgraph isomorphism based on the next result, which follows from Definition 7.11.

Notice first that there is a subgraph isomorphism between two labeled graphs if there is a subgraph isomorphism between the underlying graphs and, furthermore, corresponding vertices and edges share the same label.

Lemma 7.15. *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs, let $V \subseteq V_1$ and $W \subseteq V_2$, and let $M \subseteq V \times W$ be an isomorphism of the subgraph of G_1 induced by V to a subgraph of G_2 with vertex set W . For any vertices $v \in V_1 \setminus V$ and $w \in V_2 \setminus W$, $M \cup \{(v, w)\}$ is an isomorphism of the subgraph of G_1 induced by $V \cup \{v\}$ to a subgraph of G_2 with vertex set $W \cup \{w\}$ if and only if the following conditions*

- if $(x, v) \in E_1$ then $(y, w) \in E_2$
- if $(v, x) \in E_1$ then $(w, y) \in E_2$

hold for all vertices $x \in V_1 \setminus V$ and $y \in V_2 \setminus W$.

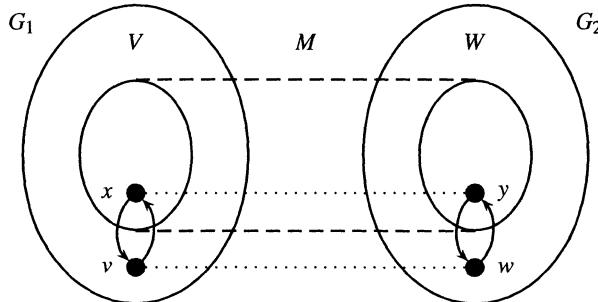


Fig. 7.9. Proof of Lemma 7.15.

Proof. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs, let $V \subseteq V_1$ and $W \subseteq V_2$, and let $M \subseteq V \times W$ be an isomorphism of the subgraph of G_1 induced by V to a subgraph (W, E) of G_2 . See Fig. 7.9. Let also $v \in V_1 \setminus V$ and $w \in V_2 \setminus W$.

Suppose $(y, w) \in E_2$ if $(x, v) \in E_1$ and $(w, y) \in E_2$ if $(v, x) \in E_1$, for all $(x, y) \in M$. Then, by Definition 7.1, $M \cup \{(v, w)\}$ is an isomorphism of the subgraph of G_1 induced by $V \cup \{v\}$ to the subgraph $(W \cup \{w\}, E \cup \{(y, w) \in E_2 \mid y \in W, (x, y) \in M, (x, v) \in E_1\}) \cup \{(w, y) \in E_2 \mid y \in W, (x, y) \in M, (v, x) \in E_1\})$ of G_2 .

Conversely, let $M \cup \{(v, w)\}$ be an isomorphism of the subgraph of G_1 induced by $V \cup \{v\}$ to a subgraph $(W \cup \{w\}, E)$ of G_2 , and let $x \in V$ and $y \in W$ with $(x, y) \in M$. Since $(v, w), (x, y) \in M \cup \{(v, w)\}$, by Definition 7.1 it holds that $(x, v) \in E_1$ if and only if $(y, w) \in E \subseteq E_2$ and $(v, x) \in E_1$ if and only if $(w, y) \in E \subseteq E_2$. \square

The following backtracking algorithm for subgraph isomorphism extends an (initially empty) vertex mapping M in all possible ways, in order to enumerate all subgraph isomorphisms of a graph G_1 into a graph G_2 . Each subgraph isomorphism mapping M is represented by an array of vertices (of G_2) indexed by the vertices of G_1 .

370a

```
<subgraph isomorphism 370a>≡
void subgraph_isomorphism(
    const GRAPH<string,string>& G1,
    const GRAPH<string,string>& G2,
    node_array<node>& MAP,
    list<node_array<node>>& ALL)
{
    ALL.clear();
    (set up candidate vertices 370b)
    (set up adjacency matrices 356a)
    node v = G1.first_node();
    extend_subgraph_isomorphism(G1,G2,ALL,CAN,v,MAP,ALL);
}
```

The vertices of graph G_2 which a vertex of graph G_1 might be mapped to are represented by a set of candidate vertices $CAN[v]$, for each vertex v of G_1 . Lemma 7.14 implies that only those vertices w of G_2 with $\deg(v) \leq \deg(w)$ are candidates which vertex v of G_1 might be mapped to.

370b

```
<set up candidate vertices 370b>≡
node_array<set<node>> CAN(G1);
{ node v,w;
forall_nodes(v,G1) {
    forall_nodes(w,G2) {
        if( G1.indeg(v) ≤ G2.indeg(w) ∧
            G1.outdeg(v) ≤ G2.outdeg(w) ∧
            GI[v] ≡ G2[w] ) {
            CAN[v].insert(w);
        }
    }
}
```

The actual extension of a subgraph isomorphism mapping is done by the following recursive procedure, which extends a vertex mapping M of a subgraph of G_1 to a subgraph of G_2 by mapping vertex $v \in V_1$

to each candidate vertex $w \in V_2$ in turn which has not been mapped to yet, making a recursive call upon the successor of vertex v in G_1 whenever the extension of M mapping v to w preserves adjacencies. All such mappings M which are defined for all vertices of G_1 , that is, all isomorphisms of graph G_1 to a subgraph of G_2 , are collected in a list L of mappings.

371

```

⟨subgraph isomorphism 370a⟩+≡
void extend_subgraph_isomorphism(
    const GRAPH<string,string>& G1,
    const GRAPH<string,string>& G2,
    const node_matrix<edge>& A1,
    const node_matrix<edge>& A2,
    node_array<set<node>>& CAN,
    node v,
    node_array<node>& MAP,
    list<node_array<node>>& L)
{
    node w,x,y;
    forall_nodes(w,G2) {
        if ( CAN[v].member(w) ) { // can map v to w

            node_array<set<node>> NEXT(G1);
            forall_nodes(x,G1) NEXT[x] = CAN[x];
            forall_nodes(x,G1)
                if ( x ≠ v ) NEXT[x].del(w);
            forall_nodes(y,G2)
                if ( y ≠ w ) NEXT[v].del(y);

            if ( refine_subgraph_isomorphism(G1,G2,A1,A2,NEXT,v,w) ) {
                if ( v ≡ G1.last_node() ) {
                    forall_nodes(x,G1) MAP[x] = NEXT[x].choose();
                    double_check_subgraph_isomorphism(G1,G2,A1,A2,MAP);
                    L.append(MAP);
                } else {
                    extend_subgraph_isomorphism(G1,G2,A1,A2,
                        NEXT,G1.succ_node(v),MAP,L);
                }
            }
        }
    }
}

```

The obvious adjacency test, that a subgraph isomorphism mapping M may be extended by mapping vertex $v \in V_1$ to vertex $w \in V_2$ if and only if $\text{indeg}(v) \leq \text{indeg}(w)$, $\text{outdeg}(v) \leq \text{outdeg}(w)$, every (already mapped) predecessor x of vertex v was mapped to a predecessor of vertex w , and every (already mapped) successor x of vertex v was mapped to a successor of vertex w , can be improved in order to earlier detect whether such an extension will lead to a later impossibility to further extend it in any way, because it will cause some unmapped

vertex x to have no candidate vertex y which it could be mapped to, based on the following result.

Lemma 7.16. *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs, let $V \subseteq V_1$ and $W \subseteq V_2$, and let $M \subseteq V \times W$ be a graph isomorphism of the subgraph of G_1 induced by V to the subgraph of G_2 induced by W . A necessary condition for M to be contained in some subgraph isomorphism of G_1 into G_2 is that for all vertices $v \in V_1 \setminus V$, there is some vertex $w \in V_2 \setminus W$ such that $M \cup \{(v, w)\}$ is a graph isomorphism of the subgraph of G_1 induced by $V \cup \{v\}$ to the subgraph of G_2 induced by $W \cup \{w\}$.*

Proof. Let $M \subseteq V \times W$ be a graph isomorphism of the subgraph of $G_1 = (V_1, E_1)$ induced by $V \subseteq V_1$ to the subgraph of $G_2 = (V_2, E_2)$ induced by $W \subseteq V_2$, and let $v \in V_1 \setminus V$ such that, for all vertices $w \in V_2 \setminus W$, $M \cup \{(v, w)\}$ is not an isomorphism of the subgraph of G_1 induced by $V \cup \{v\}$ to the subgraph of G_2 induced by $W \cup \{w\}$, that is, either $G_1[v] \neq G_2[w]$, or there is some $(x, y) \in M$ such that either $(x, v) \in E_1$ but $(y, w) \notin E_2$, or $(v, x) \in E_1$ but $(w, y) \notin E_2$, or $(x, v) \in E_1$ and $(y, w) \in E_2$ but $G_1[x, v] \neq G_2[y, w]$, or $(v, x) \in E_1$ and $(w, y) \in E_2$ but $G_1[v, x] \neq G_2[w, y]$. Then, by Lemma 7.15, $M \cup \{(v, w)\}$ is not an isomorphism of the subgraph of G_1 induced by $V \cup \{v\}$ to the subgraph of G_2 induced by $W \cup \{w\}$. Since this holds for all vertices $w \in V_2 \setminus W$, the vertex mapping M cannot be extended by mapping vertex $v \in V_1 \setminus V$ to any vertex $w \in V_2 \setminus W$ and therefore, M cannot be extended to a subgraph isomorphism of G_1 into G_2 . \square

The extended adjacency test given by the previous lemma turns out to be computationally too expensive to be performed upon every potential extension of a subgraph isomorphism mapping, meaning that it does not pay off in practice to systematically perform such an extended adjacency test. A weaker form of the lemma, consisting of testing only those potential extensions of a subgraph isomorphism mapping upon the neighborhood of the vertex last added to the mapping, turns out to perform much better in practice (and in theory).

Lemma 7.17. *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs, let $V \subseteq V_1$ and $W \subseteq V_2$, and let $M \subseteq V \times W$ be a graph isomorphism of the subgraph of G_1 induced by V to the subgraph of G_2 induced by W .*

A necessary condition for the extension of a mapping $M \cup \{(x, y)\} \subseteq V \times W$ to be contained in some subgraph isomorphism of G_1 into G_2 is that for all vertices $v \in V_1 \setminus V$ which are adjacent to vertex $x \in V$, there is some vertex $w \in V_2 \setminus W$ which is adjacent to vertex $y \in W$ and such that $M \cup \{(x, y), (v, w)\}$ is a graph isomorphism of the subgraph of G_1 induced by $V \cup \{x, v\}$ to the subgraph of G_2 induced by $W \cup \{y, w\}$.

The following procedure detects if the extension of M by mapping vertex v to some vertex $w \in V_2 \setminus W$ cannot lead to any subgraph isomorphism of G_1 into G_2 , that is, if such an extension would leave no candidate vertices $y \in V_2 \setminus (W \cup \{w\})$ which some other vertex $x \in V_1 \setminus (V \cup \{v\})$ could be mapped to, by updating the set of candidate vertices $CAN[x]$ which each vertex $x \in V_1 \setminus (V \cup \{v\})$ might be mapped to, based on the adjacencies of vertex v in G_1 and of vertex w in G_2 , and returns true if and only if all these sets of candidate vertices remain nonempty.

373

```
(subgraph isomorphism 370a) +≡
  bool refine_subgraph_isomorphism(
    const GRAPH<string, string>& G1,
    const GRAPH<string, string>& G2,
    const node_matrix<edge>& A1,
    const node_matrix<edge>& A2,
    node_array<set<node>>& CAN,
    node v,
    node w)
  {
    if ( G1.indeg(v) > G2.indeg(w) ∨ G1.outdeg(v) > G2.outdeg(w) )
      return false;

    node x,y;
    edge e;

    forall_in_edges(e,v) {
      x = G1.opposite(v,e);
      if ( index(x) > index(v) ) // future vertex
        forall_nodes(y,G2)
          if ( A2(y,w) ≡ nil ∨
              GI[e] ≠ G2[A2(y,w)] ) CAN[x].del(y);
    }

    forall_out_edges(e,v) {
      x = G1.opposite(v,e);
      if ( index(x) > index(v) ) // future vertex
        forall_nodes(y,G2)
          if ( A2(w,y) ≡ nil ∨
              GI[e] ≠ G2[A2(w,y)] ) CAN[x].del(y);
    }
  }
}
```

```

}

forall_nodes(x,G1)
  if ( index(x) > index(v) ∧ CAN[x].empty() )
    return false;

return true;
}

```

The following double-check of subgraph isomorphism, although being redundant, gives some reassurance of the correctness of the implementation. It verifies that M is a graph isomorphism of graph G_1 to a subgraph of G_2 , where $n_1 \leq n_2$ and $m_1 \leq m_2$, by double-checking that M is an injection (that is, M is defined for all vertices of G_1 , and different vertices of G_1 are mapped by M to different vertices of G_2) and that all edges of G_1 are mapped to edges of G_2 .

374 ⟨double-check subgraph isomorphism 374⟩ ≡

```

void double_check_subgraph_isomorphism(
  const GRAPH<string,string>& G1,
  const GRAPH<string,string>& G2,
  const node_matrix<edge>& A1,
  const node_matrix<edge>& A2,
  const node_array<node>& M)
{
  if ( G1.number_of_nodes() > G2.number_of_nodes() ∨
       G1.number_of_edges() > G2.number_of_edges() )
    error_handler(1,"Wrong implementation of subgraph isomorphism");

  node v,w;
  forall_nodes(v,G1)
    if ( M[v] ≡ nil ∨ G1[v] ≠ G2[M[v]] )
      error_handler(1,"Wrong implementation of subgraph isomorphism");

  forall_nodes(v,G1)
    forall_nodes(w,G1)
      if ( ( v ≠ w ∧ ( M[v] ≡ M[w] ∨
                         A1(v,w) ≠ nil ∧ A2(M[v],M[w]) ≡ nil ) ) ∨
            ( A1(v,w) ≠ nil ∧ A2(M[v],M[w]) ≡ nil ∧
              G1[A1(v,w)] ≠ G2[A2(M[v],M[w])] ) )
        error_handler(1,"Wrong implementation of subgraph isomorphism");
}

```

7.3.2 Interactive Demonstration of Subgraph Isomorphism

The backtracking algorithm for subgraph isomorphism is integrated next in the interactive demonstration of graph algorithms. A simple checker for isomorphic subgraphs that provides some visual reassurance consists of redrawing graph G_1 according to the subgraph isomorphism found, to match the layout of graph G_2 , as well as highlighting the subgraph of G_2 which is isomorphic to G_1 .

375

```

<demo subgraph isomorphism 375>≡
void gw_subgraph_isomorphism(
    GraphWin& gw1)
{
    GRAPH<string,string>& G1 = gw1.get_graph();
    node_array<node> M(G1,nil);

    GRAPH<string,string> G2;
    GraphWin gw2(G2,500,500,"Subgraph Isomorphism");
    gw2.display();
    gw2.message("Enter second graph. Press done when finished");
    gw2.edit();
    gw2.del_message();

    list<node_array<node> > L;
    subgraph_isomorphism(G1,G2,M,L);

    panel P;
    if (L.length() > 1) {
        make_proof_panel(P,
            string("There are %i subgraph isomorphisms",L.length()),true);
        if (gw1.open_panel(P)) { // proof button pressed
            (set up adjacency matrices 356a)
            gw1.save_all_attributes();
            node_array<point> pos1(G1);
            node_array<point> pos2(G2);
            forall(M,L) {
                (show subgraph isomorphism 376)
                panel Q;
                make_yes_no_panel(Q,"Continue",true);
                if (gw1.open_panel(Q)) { // no button pressed
                    gw1.restore_all_attributes();
                    gw2.restore_all_attributes();
                    break;
                }
                gw2.restore_all_attributes();
            }
            gw1.restore_all_attributes();
        }
    } else if (L.length() ≡ 1) {

```

```

make_proof_panel(P,"There is one subgraph isomorphism",true);
if( gw1.open_panel(P) ) { // proof button pressed
    <setup adjacency matrices 356a>
    gw1.save_all_attributes();
    node_array<point> pos1(G1);
    node_array<point> pos2(G2);
    M = L.head();
    <show subgraph isomorphism 376>
    gw1.wait();
    gw1.restore_all_attributes();
}

} else {
    make_proof_panel(P,"There is \\\red no subgraph isomorphism",false);
    gw1.open_panel(P);
}
}

```

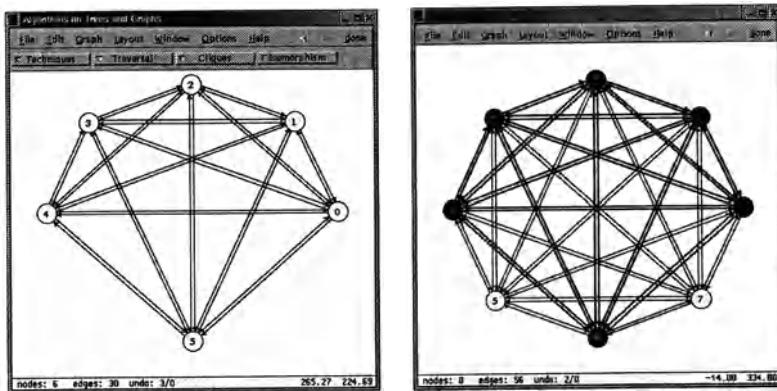


Fig. 7.10. Finding all subgraph isomorphisms of K_6 into K_8 .

Graph G_1 is redrawn according to the subgraph isomorphism mapping M found, to match the layout of graph G_2 , by updating the position of every vertex $v \in V_1$ in the layout of G_1 to be identical with the position of vertex $M[v] \in V_2$ in the layout of G_2 , and removing any edge bends in the layout of both graphs. The subgraph of G_2 isomorphic to G_1 is also highlighted.

```

<show subgraph isomorphism 376>≡
gw2.get_position(pos2);
gw2.set_layout(pos2); // remove edge bends

```

```

node v;
edge e1,e2;
forall_nodes(v,G1)
  pos1[v] = pos2[M[v]];
  gw1.set_position(pos1);
  gw1.set_layout(pos1);
  gw2.save_all_attributes();

forall_nodes(v,G1)
  gw2.set_color(M[v],blue);
forall_edges(e1,G1) {
  e2 = A2(M[G1.source(e1)],M[G1.target(e1)]);
  if( e2 ≠ nil ) {
    gw2.set_color(e2,blue);
    gw2.set_width(e2,2);
  }
}

```

Execution of the interactive demonstration of graph algorithms to find all subgraph isomorphisms of K_6 into K_8 is illustrated in Fig. 7.10.

7.4 Maximal Common Subgraph Isomorphism

Another important generalization of graph isomorphism which also generalizes subgraph isomorphism, is known as maximal common subgraph isomorphism. The maximal common subgraph isomorphism problem consists in finding a common subgraph between two graphs which is not a proper subgraph of any other common subgraph between the two graphs, and is also a fundamental problem with a variety of applications in engineering sciences, organic chemistry, biology, and pattern matching.

Definition 7.18. *A common induced subgraph isomorphism of a graph G_1 to a graph G_2 is a structure (S_1, S_2, M) , where S_1 is an induced subgraph of G_1 , S_2 is an induced subgraph of G_2 , and M is a graph isomorphism of S_1 to S_2 . A common induced subgraph isomorphism is maximal if there is no common induced subgraph isomorphism (S'_1, S'_2, M') of G_1 to G_2 such that S_1 is a proper (induced) subgraph of G_1 and S_2 is a proper (induced) subgraph of G_2 .*

Replacing an induced subgraph by a subgraph in Definition 7.18 leads to the notion of maximal common subgraph isomorphism. Further, a maximum common (induced) subgraph isomorphism of two

graphs is a largest maximal common (induced) subgraph isomorphism of the two graphs.

Example 7.19. A maximum common induced subgraph isomorphism of the following two graphs is $((V_s, E_s), (V_t, E_t), M)$ with

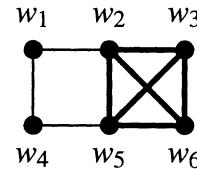
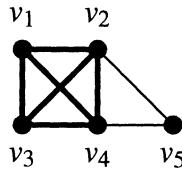
$$V_s = \{v_1, v_2, v_3, v_4\},$$

$$E_s = \{(v_1, v_2), (v_1, v_3), (v_1, v_4), (v_2, v_3), (v_2, v_4), (v_3, v_4), (v_2, v_1), (v_3, v_1), (v_4, v_1), (v_3, v_2), (v_4, v_2), (v_4, v_3)\},$$

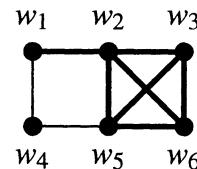
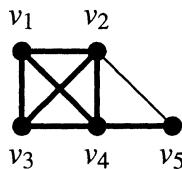
$$V_t = \{w_2, w_3, w_5, w_6\},$$

$$E_t = \{(w_2, w_3), (w_2, w_5), (w_2, w_6), (w_3, w_5), (w_3, w_6), (w_5, w_6), (w_3, w_2), (w_5, w_2), (w_6, w_2), (w_5, w_3), (w_6, w_3), (w_6, w_5)\},$$

$$M = \{(v_1, w_5), (v_2, w_6), (v_3, w_3), (v_4, w_2)\},$$



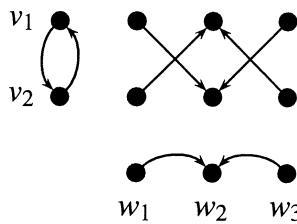
while a maximum common subgraph isomorphism of the two graphs is $((V_s \cup \{v_5\}, E_s \cup \{(v_4, v_5), (v_5, v_4)\}), (V_t \cup \{w_1\}, E_t \cup \{(w_1, w_2), (w_2, w_1)\}), M \cup \{(v_5, w_1)\})$.



Computation of all maximal common induced subgraph isomorphisms of two graphs can be reduced to finding all maximal cliques in the graph product of the two graphs. The vertices in the graph product are ordered pairs (v, w) with $v \in V_1$ and $w \in V_2$, and there is an edge from vertex (v_i, w_i) to vertex (v_j, w_j) in the graph product if and only if $v_i \neq v_j, w_i \neq w_j$, and either there is an edge from vertex v_i to vertex v_j in graph G_1 and an edge from vertex w_i to vertex w_j in graph G_2 , or there is no edge from vertex v_i to vertex v_j in graph G_1 and no edge from vertex w_i to vertex w_j in graph G_2 .

Definition 7.20. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs. The graph product of G_1 and G_2 , denoted by $G_1 \times G_2$, is the graph $G = (V, E)$ with vertex set $V = V_1 \times V_2$ and edge set $E = \{((v_i, w_i), (v_j, w_j)) \in V \times V \mid v_i \neq v_j, w_i \neq w_j, (v_i, v_j) \in E_1, (w_i, w_j) \in E_2\} \cup \{((v_i, w_i), (v_j, w_j)) \in V \times V \mid v_i \neq v_j, w_i \neq w_j, (v_i, v_j) \notin E_1, (w_i, w_j) \notin E_2\}$.

Example 7.21. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs with $V_1 = \{v_1, v_2\}$, $E_1 = \{(v_1, v_2), (v_2, v_1)\}$, $V_2 = \{w_1, w_2, w_3\}$, $E_2 = \{(w_1, w_2), (w_3, w_2)\}$.



The graph product $G_1 \times G_2$ has vertex set $\{(v_1, w_1), (v_1, w_2), (v_1, w_3), (v_2, w_1), (v_2, w_2), (v_2, w_3)\}$ and edge set $\{((v_1, w_1), (v_2, w_1)), ((v_1, w_2), (v_2, w_2)), ((v_1, w_3), (v_2, w_3)), ((v_2, w_1), (v_1, w_2)), ((v_2, w_2), (v_1, w_3))\}$.

Complete subgraphs of the graph product of two graphs can be projected onto induced subgraphs of the respective graphs.

Definition 7.22. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs, and let $C = (V, E)$ be a complete subgraph of $G_1 \times G_2$. The projection of C onto G_1 is the subgraph of G_1 induced by the vertices $v \in V_1$ such that $(v, w) \in V$ for some vertex $w \in V_2$. The projection of C onto G_2 is the subgraph of G_2 induced by the vertices $w \in V_2$ such that $(v, w) \in V$ for some vertex $v \in V_1$.

Remark 7.23. Notice that self-loops are ignored in Definition 7.20, for otherwise there would be two counter-parallel edges in the graph product between every loopless vertex of G_1 and every loopless vertex of G_2 . In order to keep the graph product small, graph are thus assumed not to have self-loops.

Now, maximal common induced subgraph isomorphisms of a graph G_1 to another graph G_2 correspond to maximal cliques in the graph product $G_1 \times G_2$.

Theorem 7.24. Let G_1 and G_2 be two graphs, and let (S_1, S_2, M) be a maximal common induced subgraph isomorphism of G_1 to G_2 . Then, there is a maximal clique C of $G_1 \times G_2$ whose projection onto G_1 is S_1 and whose projection onto G_2 is S_2 .

Conversely, let $C = (X, E)$ be a maximal clique of $G_1 \times G_2$, let S_1 be the projection of C onto G_1 , and let S_2 be the projection of C onto G_2 . Then, there is a maximal common induced subgraph isomorphism (S_1, S_2, M) of G_1 to G_2 .

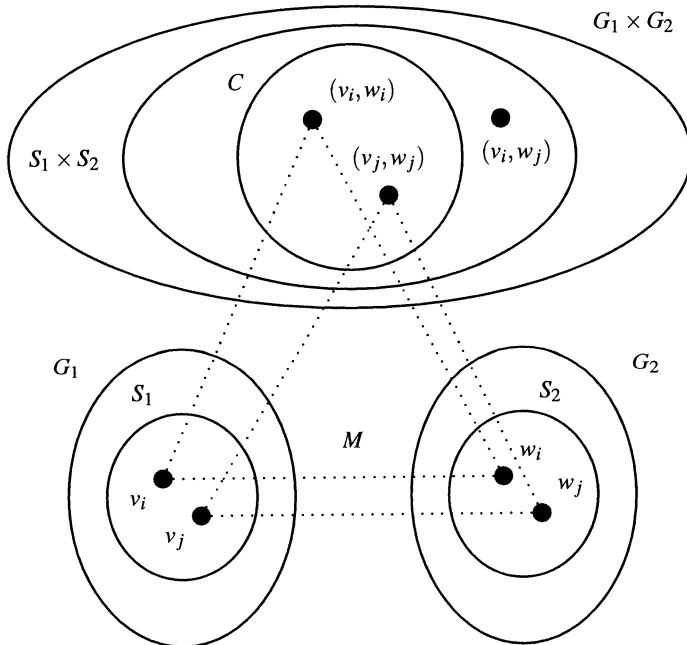


Fig. 7.11. Proof of Theorem 7.24.

Proof. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs, and let (S_1, S_2, M) be a common induced subgraph isomorphism of G_1 to G_2 . Let also $C = (X, E)$ be the subgraph of $S_1 \times S_2$ induced by the vertex set $X = \{(v, w) \in V_1 \times V_2 \mid (v, w) \in M\}$. See Fig. 7.11. Since M is a bijection, $v_i \neq v_j$ and $w_i \neq w_j$ for all distinct vertices (v_i, w_i) and (v_j, w_j) of C . Furthermore, there is an edge from v_i to v_j in S_1 if and only if

there is an edge from w_i to w_j in S_2 , for all (v_i, w_i) and (v_j, w_j) in M , because M is an isomorphism of S_1 to S_2 . Then, by Definition 7.20, there is an edge from (v_i, w_i) to (v_j, w_j) in $S_1 \times S_2$, for all distinct vertices (v_i, w_i) and (v_j, w_j) of $S_1 \times S_2$ which are also in M and, C being an induced subgraph of $S_1 \times S_2$, there is an edge from (v_i, w_i) to (v_j, w_j) in C , for all distinct vertices (v_i, w_i) and (v_j, w_j) of C . Therefore, C is a clique. Furthermore, since S_1 is an induced subgraph of G_1 and S_2 is an induced subgraph of G_2 , it follows from Definition 7.22 that the projection of C onto G_1 is S_1 and the projection of C onto G_2 is S_2 .

Suppose now that the clique $C = (X, E)$ is not maximal, and let $S_1 = (U_1, F_1)$ and $S_2 = (U_2, F_2)$. Then, there is some vertex $(v, w) \in V_1 \times V_2 \setminus X$ such that the subgraph of $G_1 \times G_2$ induced by $X \cup \{(v, w)\}$ is a clique. For all $(x, y) \in X$, since there are edges from (x, y) to (v, w) and from (v, w) to (x, y) in the new clique, it follows from Definition 7.20 that $x \neq v$, $y \neq w$, $(x, v) \in E_1$ if and only if $(y, w) \in E_2$, and $(v, x) \in E_1$ if and only if $(w, y) \in E_2$. Let T_1 be the subgraph of G_1 induced by $U_1 \cup \{v\}$, and let T_2 be the subgraph of G_2 induced by $U_2 \cup \{w\}$. Then, $M \cup \{(v, w)\}$ is a graph isomorphism of T_1 to T_2 and $(T_1, T_2, M \cup \{(v, w)\})$ is a common induced subgraph isomorphism of G_1 to G_2 , contradicting the hypothesis that (S_1, S_2, M) is a maximal common induced subgraph isomorphism of G_1 to G_2 . Therefore, the clique C is maximal.

Conversely, let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs, let $C = (X, E)$ be a maximal clique of $G_1 \times G_2$, and let S_1 and S_2 be the projections of C onto G_1 and G_2 , respectively. For all distinct vertices (v_i, w_i) and (v_j, w_j) of C , there is an edge from (v_i, w_i) to (v_j, w_j) in C and then it follows from Definition 7.20 that $v_i \neq v_j$, $w_i \neq w_j$, and $(v_i, v_j) \in E_1$ if and only if $(w_i, w_j) \in E_2$. Then, $M = \{(v, w) \in V_1 \times V_2 \mid (v, w) \in X\}$ is a graph isomorphism of S_1 to S_2 and therefore, since projections of cliques are induced subgraphs by Definition 7.22, (S_1, S_2, M) is a common induced subgraph isomorphism of G_1 to G_2 .

Suppose now that the common induced subgraph isomorphism (S_1, S_2, M) of G_1 to G_2 is not maximal, and let $S_1 = (U_1, F_1)$ and $S_2 = (U_2, F_2)$. Then, there are vertices $v \in V_1 \setminus U_1$ and $w \in V_2 \setminus U_2$ such that $(T_1, T_2, M \cup \{(v, w)\})$ is a common induced subgraph isomorphism of G_1 to G_2 , where T_1 is the subgraph of G_1 induced by

$U_1 \cup \{v\}$ and T_2 is the subgraph of G_2 induced by $U_2 \cup \{w\}$. By Definition 7.1 and Definition 7.18, $(x, v) \in E_1$ if and only if $(y, w) \in E_2$ and $(v, x) \in E_1$ if and only if $(w, y) \in E_2$, for all vertices $x \in U_1$ and $y \in U_2$ and then, by Definition 7.20 and since $x \neq v$ and $y \neq w$, there are edges from (x, y) to (v, w) and from (v, w) to (x, y) in the subgraph of $G_1 \times G_2$ induced by $X \cup \{(v, w)\}$. Then, the subgraph of $G_1 \times G_2$ induced by $X \cup \{(v, w)\}$ is a clique, contradicting the hypothesis that $C = (X, E)$ is a maximal clique. Therefore, (S_1, S_2, M) is a maximal common induced subgraph isomorphism of G_1 to G_2 . \square

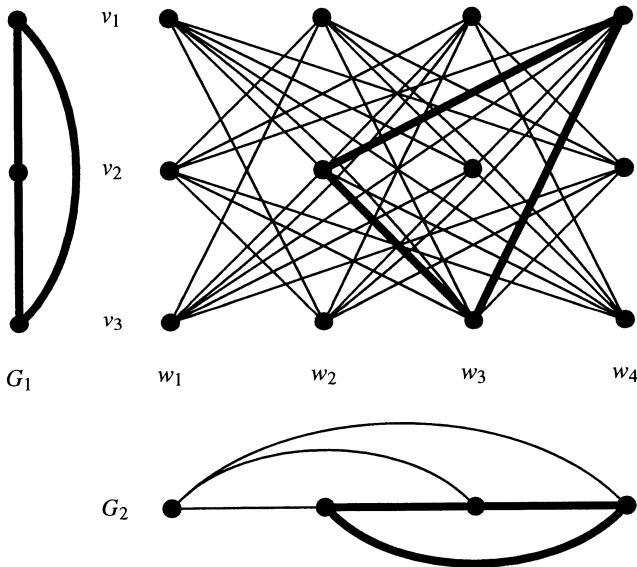


Fig. 7.12. A maximal clique in the product of two graphs.

Example 7.25. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be the two graphs shown in Fig. 7.12, with $V_1 = \{v_1, v_2, v_3\}$ and $V_2 = \{w_1, w_2, w_3, w_4\}$ and where pairs of counter-parallel edges be depicted as undirected edges. The projections of the maximal clique of $G_1 \times G_2$ induced by $\{(v_1, w_4), (v_2, w_2), (v_3, w_3)\}$ are isomorphic, and correspond to a maximal common induced subgraph isomorphism of G_1 to G_2 .

Notice that Theorem 7.24 no longer holds if maximal common subgraph isomorphisms are not induced subgraph isomorphisms. See Exercise 7.8.

7.4.1 An Algorithm for Maximal Common Subgraph Isomorphism

Computation of all maximal common induced subgraph isomorphisms of two graphs proceeds by first computing the graph product of the two graphs, and then finding all maximal cliques in the graph product.

Notice first that the vertices in the product of two labeled graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are ordered pairs (v, w) in the underlying graph product $G_1 \times G_2$ with $v \in V_1$ and $w \in V_2$ sharing the same vertex label, and there is an edge from vertex (v_i, w_i) to vertex (v_j, w_j) in the labeled graph product if and only there is such an edge in the product of the underlying graphs and if there are edges from vertex v_i to vertex v_j in graph G_1 and from vertex w_i to vertex w_j in graph G_2 sharing the same edge label. That is, the graph product of two labeled graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is the graph $G = (V, E)$ with vertex set $V = \{(v, w) \in V_1 \times V_2 \mid G_1[v] = G_2[w]\}$ and edge set $E = \{((v_i, w_i), (v_j, w_j)) \in V \times V \mid v_i \neq v_j, w_i \neq w_j, (v_i, v_j) \in E_1, (w_i, w_j) \in E_2, G_1[v_i, v_j] = G_2[w_i, w_j]\} \cup \{((v_i, w_i), (v_j, w_j)) \in V \times V \mid v_i \neq v_j, w_i \neq w_j, (v_i, v_j) \notin E_1, (w_i, w_j) \notin E_2\}$.

383 ⟨maximal common subgraph isomorphism 383⟩≡

```

void graph_product(
    GRAPH<two_tuple<node,node>,edge>& G, // <edge> not used
    const GRAPH<string,string>& G1,
    const GRAPH<string,string>& G2)
{
    edge e;
    forall_edges(e,G1)
        if ( G1.source(e) ≡ G1.target(e) )
            error_handler(1,"First graph cannot have self-loops");
    forall_edges(e,G2)
        if ( G2.source(e) ≡ G2.target(e) )
            error_handler(1,"Second graph cannot have self-loops");

    G.clear();
    node v,w;
    forall_nodes(v,G1) {
        forall_nodes(w,G2) {
            if ( G1[v] ≡ G2[w] ) {
                two_tuple<node,node> vw(v,w);

```

```

    G.new_node(vw);
} } }
⟨set up adjacency matrices 356a⟩
forall_nodes(v,G) {
forall_nodes(w,G) {
  if ( G[v].first() ≠ G[w].first() ∧
      G[v].second() ≠ G[w].second() ∧
      ( A1(G[v].first(),G[w].first()) ≠ nil ∧
        A2(G[v].second(),G[w].second()) ≠ nil ∧
        GI[A1(G[v].first(),G[w].first())] ≡ G2[A2(G[v].second(),G[w].second())] ∨
        A1(G[v].first(),G[w].first()) ≡ nil ∧
        A2(G[v].second(),G[w].second()) ≡ nil ) )
    G.new_edge(v,w);
} } }

```

The following procedure implements the reduction of maximal common induced subgraph isomorphism to maximal clique.

384

```

⟨maximal common subgraph isomorphism 383⟩+≡
void maximal_common_subgraph_isomorphism(
  const GRAPH<string,string>& G1,
  const GRAPH<string,string>& G2,
  list<node_array<node>>& ALL)
{
  GRAPH<two_tuple<node,node>,edge> G; // <edge> not used
  graph_product(G,G1,G2);

  ⟨set up adjacency matrix 316a⟩

  list<set<node>> L;
  all_maximal_cliques(G,A,L);

  ⟨set up adjacency matrices 356a⟩
  ALL.clear();
  node_array<node> M(G1);
  node v;
  set<node> C;
  forall(C,L) {
    forall_nodes(v,G1)
      M[v] = nil;
    forall(v,C)
      M[G[v].first()] = G[v].second();
    double_check_maximal_common_subgraph_isomorphism(G1,G2,A1,A2,M);
    ALL.append(M);
  } }

```

The following double-check of maximal common subgraph isomorphism, although being redundant, gives some reassurance of the correctness of the implementation. It verifies that M is a graph iso-

morphism of an induced subgraph of G_1 to an induced subgraph of G_2 , and that it is maximal, by double-checking that

- vertices of G_1 are mapped by M to vertices of G_2 having the same label,
- different vertices of G_1 are mapped by M to different vertices of G_2 ,
- edges of G_1 are mapped by M to edges of G_2 having the same label,
- different edges of G_1 are mapped by M to different edges of G_2 ,
- nonedges of G_1 are mapped by M to nonedges of G_2 , and
- no unmapped vertex of G_1 can be mapped to an unmapped vertex of G_2 .

385

```
<double-check maximal common subgraph isomorphism 385>≡
void double_check_maximal_common_subgraph_isomorphism(
    const GRAPH<string,string>& G1,
    const GRAPH<string,string>& G2,
    const node_matrix<edge>& A1,
    const node_matrix<edge>& A2,
    const node_array<node>& M)
{
    node v,w,x;
    forall_nodes(v,G1)
        if ( M[v] ≠ nil ∧ G1[v] ≠ G2[M[v]] )
            error_handler(1,
                "Wrong implementation of common subgraph isomorphism");

    forall_nodes(v,G1)
    forall_nodes(w,G1)
        if ( M[v] ≠ nil ∧ M[w] ≠ nil ∧
            ( v ≠ w ∧ M[v] ≡ M[w] ∨
                A1(v,w) ≠ nil ∧ A2(M[v],M[w]) ≡ nil ∨
                A1(v,w) ≡ nil ∧ A2(M[v],M[w]) ≠ nil ∨
                A1(v,w) ≠ nil ∧ A2(M[v],M[w]) ≠ nil
                ∧ G1[A1(v,w)] ≠ G2[A2(M[v],M[w])] ) )
            error_handler(1,
                "Wrong implementation of common subgraph isomorphism");

    list<node> V1;
    list<node> R1;
    set<node> R2;
    forall_nodes(v,G1) {
        if ( M[v] ≠ nil ) {
            V1.append(v);
        } else {
            R1.append(v);
        }
    }
    forall_nodes(v,G2) R2.insert(v);
    forall(v,V1) R2.del(M[v]);

    forall(v,R1) {
```

```

forall(w,R2) {
    if ( GI[v] ≡ G2[w] ) {
        bool isomorphic = true;
        forall(x,VI) {
            if ( AI(v,x) ≠ nil ∧ A2(w,M[x]) ≡ nil ∨
                AI(v,x) ≡ nil ∧ A2(w,M[x]) ≠ nil ∨
                AI(v,x) ≠ nil ∧ A2(w,M[x]) ≠ nil
                ∧ GI[AI(v,x)] ≠ G2[A2(w,M[x])] ) {
                isomorphic = false;
                break;
            }
            if ( AI(x,v) ≠ nil ∧ A2(M[x],w) ≡ nil ∨
                AI(x,v) ≡ nil ∧ A2(M[x],w) ≠ nil ∨
                AI(x,v) ≠ nil ∧ A2(M[x],w) ≠ nil
                ∧ GI[AI(x,v)] ≠ G2[A2(M[x],w)] ) {
                isomorphic = false;
                break;
            }
        }
        if ( isomorphic ) // can map v to w
            error_handler(1,
                "Wrong implementation of common subgraph isomorphism");
    } } } }
}

```

7.4.2 Interactive Demonstration of Maximal Common Subgraph Isomorphism

The algorithm implementing the reduction of maximal common subgraph isomorphism to maximal clique, is integrated next in the interactive demonstration of graph algorithms. A simple checker for maximal common subgraph isomorphisms that provides some visual reassurance consists of highlighting the induced subgraph of G_1 and the induced subgraph of G_2 which are isomorphic.

```

386 <demo maximal common subgraph isomorphism 386>≡
    void gw_maximal_common_subgraph_isomorphism(
        GraphWin& gw1)
    {
        GRAPH<string,string>& G1 = gw1.get_graph();

        GRAPH<string,string> G2;
        GraphWin gw2(G2,500,500,"Maximal Common Subgraph Isomorphism");
        gw2.display();
        gw2.message("Enter second graph. Press done when finished");
        gw2.edit();
        gw2.del_message();

        list<node_array<node> > L;
        maximal_common_subgraph_isomorphism(G1,G2,L);
    }
}

```

```

panel P;
make_proof_maximum_panel(P,
  string("There are %i maximal common subgraph isomorphisms",
  L.length()),true);
int button = gw1.open_panel(P);
switch(button) {
  case 0 : break; // ok button pressed
  case 1 : { // proof button pressed
    node_array<node> M;
    node v;
    edge e;
    ⟨set up adjacency matrices 356a⟩

    forall(M,L) {
      gw1.save_all_attributes();
      gw2.save_all_attributes();
      ⟨show common subgraph isomorphism 388⟩

      panel Q;
      make_yes_no_panel(Q,"Continue",true);
      if ( gw1.open_panel(Q) ) { // no button pressed
        gw1.restore_all_attributes();
        gw2.restore_all_attributes();
        break;
      }

      gw1.restore_all_attributes();
      gw2.restore_all_attributes();
    }
    break;
  }

  case 2 : { // maximum button pressed
    node_array<node> M;
    node v,w;
    edge e;
    int size;
    int maximum_size = 0;
    node_array<node> MAXIMUM;
    ⟨set up adjacency matrices 356a⟩

    forall(M,L) {
      size = 0;
      forall_nodes(v,GI) {
        if ( M[v] ≠ nil ) {
          size++;
          forall_nodes(w,GI)
            if ( M[w] ≠ nil ∧ AI(v,w) ≠ nil ) size++;
        }
      }
      if ( size > maximum_size ) {
        MAXIMUM = M;
        maximum_size = size;
      }
    }
  }
}

```

```

gw1.save_all_attributes();
M = MAXIMUM;
⟨show common subgraph isomorphism 388⟩
gw1.wait();
gw1.restore_all_attributes();
break;
} } }

```

A maximal common induced subgraph isomorphism is best shown by highlighting all vertices and edges in the induced subgraph of G_1 and the induced subgraph of G_2 , as well as redrawing the induced subgraph of G_1 according to the maximal common induced subgraph isomorphism found, in order to match the layout of the induced subgraph of G_2 .

388 ⟨show common subgraph isomorphism 388⟩≡

```

node_array<point> pos1(G1);
node_array<point> pos2(G2);
gw1.get_position(pos1);
gw1.set_layout(pos1); // remove edge bends
gw2.get_position(pos2);
gw2.set_layout(pos2); // remove edge bends
forall_nodes(v,G1)
  if ( M[v] ≠ nil ) pos1[v] = pos2[M[v]];
gw1.set_position(pos1);
gw1.set_layout(pos1);
forall_nodes(v,G1) {
  if ( M[v] ≠ nil ) {
    gw1.set_color(v,blue);
    gw2.set_color(M[v],blue);
  }
}
forall_edges(e,G1) {
  if ( gw1.get_color(G1.source(e)) ≡ blue ∧
        gw1.get_color(G1.target(e)) ≡ blue ) {
    gw1.set_color(e,blue);
    gw1.set_width(e,2);
  }
}
forall_edges(e,G2) {
  if ( gw2.get_color(G2.source(e)) ≡ blue ∧
        gw2.get_color(G2.target(e)) ≡ blue ) {
    gw2.set_color(e,blue);
    gw2.set_width(e,2);
  }
}

```

Execution of the interactive demonstration of graph algorithms to find all maximal common induced subgraph isomorphisms between two triangulated graphs is illustrated in Fig. 7.13.

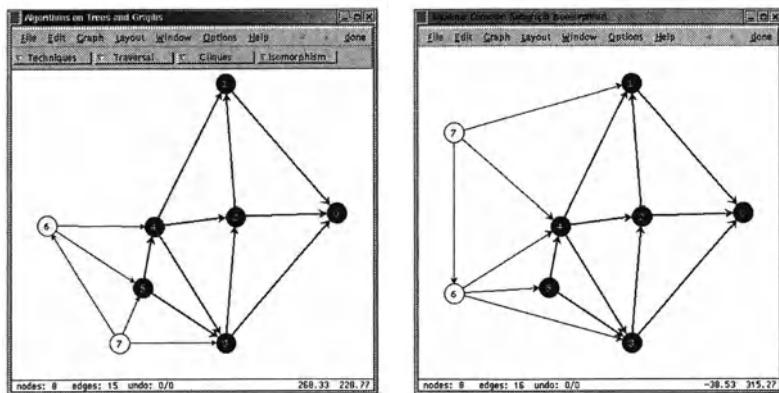


Fig. 7.13. Finding all maximal common induced subgraph isomorphisms between two triangulated graphs.

7.5 Applications

Isomorphism problems on graphs find application whenever structures represented by graphs need to be identified or compared. In most application areas, graph isomorphism, subgraph isomorphism, and maximal common (induced) subgraph isomorphism can be seen as some form of pattern matching or information retrieval.

Roughly stated, given a *pattern* graph and a *text* graph, the pattern matching problem consists of finding all subgraphs of the text that are isomorphic to the pattern. When vertices and edges of the pattern may be labeled with variables, the pattern matching problem consists of finding all subgraphs of the text that are isomorphic to some extension of the pattern, obtained by replacing these variables by appropriate subgraphs of the text. While the former corresponds to the graph or subgraph isomorphism problem, the latter is nothing but a different formulation of the maximal common (induced) subgraph isomorphism problem. All of these pattern matching problems are a form of information retrieval.

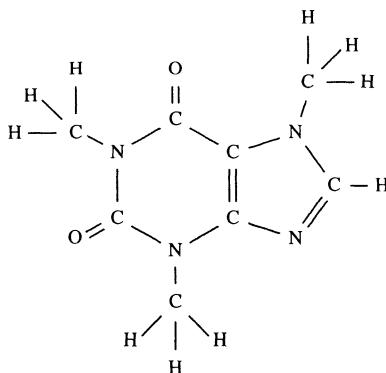
The application of the different isomorphism problems on graphs to the retrieval of information from databases of molecular graphs is discussed next, as it is representative of these pattern matching prob-

lems on graphs. See the bibliographic notes below for further applications of graph isomorphism and related problems.

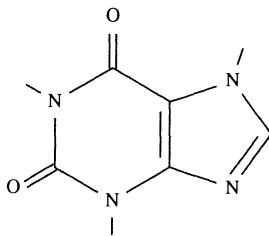
A molecular graph is a connected, labeled, and undirected graph that represents the constitution of a molecular structure. The vertices represent atoms, and the edges represent chemical bonds. Vertices are labeled with the atom type (the atomic symbol of the chemical element), and edges are labeled with the bond order or bond type (single, double, triple, or aromatic).

In schematic diagrams of molecular graphs, a single bond between two atoms is represented by an undirected edge between the atoms, a double bond is represented by a pair of parallel edges, and a triple bond is represented by three parallel edges between the atoms. An aromatic bond is represented by a circle inscribed in the corresponding cyclic chain.

Example 7.26. The chemical structure of caffeine or 1,3,7-trimethylxanthine, which has the chemical formula C₈H₁₀N₄O₂, is represented by the following molecular graph:



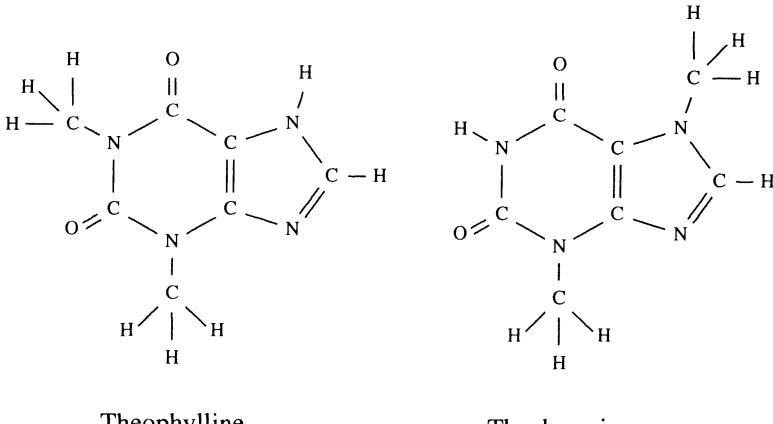
Remark 7.27. A simplified notation for drawing chemical structures is used by chemists in which carbon atoms are represented as a joint between two or more edges, and hydrogen atoms together with their incident edges are left off. Furthermore, methyl (CH₃) groups are also left off while keeping their incident edges. For instance, the chemical structure of caffeine from Example 7.26 has the following simplified representation:



Isomorphism of molecular graphs finds application in the *structure search* of a database of compounds for the presence or absence of a specified compound: when there is a need to retrieve data associated with some existing compound, or when a new compound has been discovered or synthesized.

As a matter of fact, chemical structure search cannot be made solely at the level of the chemical formula, because different compounds (called isomers) may share the same chemical formula but exhibit a different structure, as shown by the following example.

Example 7.28. The molecular graphs of 1,3-dimethylxanthine (theophylline) and 3,7-dimethylxanthine (theobromine) are not isomorphic, although both compounds share the chemical formula $C_7H_8N_4O_2$.



Subgraph isomorphism of molecular graphs finds application in the *substructure search* of a database of compounds for all compounds containing some specified compound as a substructure. Furthermore, maximal common subgraph isomorphism of molecular graphs finds application in the *similarity search* of a database of compounds for

those compounds that are most similar to some specified compound, according to some quantitative definition of structural similarity. The similarity measure used is most often based upon the substructures in common between the specified compound and each compound in the database.

Summary

Several isomorphism problems on graphs were addressed in this chapter which form a hierarchy of pattern matching problems, where maximal common induced subgraph isomorphism generalizes (induced) subgraph isomorphism, which in turn generalizes graph isomorphism. Simple algorithms are given in detail for enumerating all solutions to these problems. The algorithms for graph and subgraph isomorphism are based on the backtracking technique presented in Sect. 2.2, while the maximal common induced subgraph isomorphism problem is reduced to the maximal clique problem and solved using the algorithms discussed in Chap. 6 for finding maximal cliques. References to more sophisticated algorithms are given in the bibliographic notes below. Chemical structure search is also discussed as a prototypical application of the different isomorphism problems on (molecular) graphs.

Bibliographic Notes

A comprehensive survey of the so-called *graph isomorphism disease* can be found in [124, 268]. Complexity aspects of the graph isomorphism problem are treated in much detail in [85, 198, 288]. See also [123].

Several backtracking algorithms have been proposed for the graph isomorphism problem. The first ever backtracking algorithm for graph isomorphism was presented in [267], followed by [39, 90]. One of the best-known backtracking algorithms for graph and subgraph isomorphism, presented in [331], introduced a constraint satisfaction technique that is now known as *really full look ahead*. The backtracking algorithm for subgraph isomorphism is a modern presentation of that algorithm. The algorithms presented in [226] for graph and subgraph

isomorphism and in [227] for maximal common subgraph isomorphism, used the constraint satisfaction technique that is now known as *forward checking*. Again, the backtracking algorithm for maximal common subgraph isomorphism is a modern presentation of that algorithm. Good surveys of constraint satisfaction techniques can be found in [88, 220, 327].

A practical backtracking algorithm for computing the automorphism group of a graph is described in [231]. An alternative approach to the graph isomorphism problem is to iterate a partitioning of the vertex sets (or the edge sets) of the two graphs into classes, based on necessary conditions for graph isomorphism, and to refine the partitioned classes until no further partitioning is possible. The first partitioning algorithm for graph isomorphism was presented in [332], followed by [84, 211, 239, 287]. For a modern description, see [203, Chap. 7].

The reduction of maximum common subgraph isomorphism to cliques was proposed in [4, 25, 126, 210] and further exploited in [199]. Further, the equivalence of graph isomorphisms and maximal cliques in the graph product was proved in [202]. The product of two graphs [37] is also known as conjunction [153], direct product [192, pp. 167–170], Cartesian product [253, pp. 33–36], and Kronecker product [361]. Several distance measures between objects represented by graphs were proposed which are based on the maximum common subgraph [59, 61, 111].

Applications of graph isomorphism and related problems to chemistry were first addressed in [319], and despite applications to chemical computation such as modeling of chemical reactions [228] the most important practical application lies in information retrieval from chemical databases, since every database of chemical compounds must offer some kind of chemical structure and substructure search system, let alone bibliographical searches in specialized databases or over the Internet. Several such chemical information retrieval systems based on graph isomorphism, subgraph isomorphism, and maximal common subgraph isomorphism algorithms were developed for the CAS (American Chemical Society) Registry database [179], which holds over 18 million compounds and 13 million biological sequences, and for the Beilstein database [157, 158], which holds over 8 mil-

lion compounds and 5 million chemical reactions. See also [12, 24, 309, 369, 370]. More recent algorithms for maximal common subgraph isomorphism of molecular graphs are described in [101, 108, 109, 356, 376].

Applications of graph isomorphism and related problems to computational biology include the comparison of two-dimensional and three-dimensional protein structures, as well as the discovery of common structural motifs [137, 238, 283, 350].

Another important application area of subgraph isomorphism is object recognition and computer vision. In structural and syntactical pattern recognition, objects are represented by symbolic data structures, such as strings, trees, or graphs, and relational descriptions of complex objects are also used in computer vision. See [58, 151] for an overview. Similar object recognition problems arise in computer-aided design [54, 215, 252, 259]. Isomorphism and related problems on graphs can also be formulated as *consistent labeling* problems [149, 150] and, as a matter of fact, the *relaxation* and *look-ahead* operators [147] used to improve efficiency of relational matching [148, 185, 184, 298, 371] constitute one of the most important constraint satisfaction techniques.

Last, but not least, graph grammars and graph transformation [104, 105, 278] constitute an important area of application for subgraph isomorphism algorithms [60, 206, 279, 386], because a graph rewriting or graph transformation step succeeds by first finding a subgraph isomorphism of the left-hand side graph of some graph transformation rule into the graph to be rewritten or transformed, and then performing the rewriting or transformation specified by the rule.

Review Problems

7.1 Name at least three invariants for graph isomorphism. Are they also invariants for subgraph isomorphism and for maximal common subgraph isomorphism? Justify your answer.

7.2 Enumerate all the automorphisms of the wheel graph W_5 .

7.3 Give upper and lower bounds on the number of subgraph isomorphisms of a connected graph with n_1 vertices into a connected graph with n_2 vertices, where $n_1 \leq n_2$.

7.4 Give upper and lower bounds on the number of maximal common induced subgraph isomorphisms between two connected graphs with n_1 and n_2 vertices.

7.5 Enumerate all maximal common induced subgraph isomorphisms between the complete bipartite graph $K_{3,4}$ and the wheel graph W_5 .

7.6 Show that computation of all maximal common subgraphs of two graphs can be reduced to finding all maximal cliques in the product of the line graphs of the two graphs. See Exercise 7.8.

Exercises

7.1 Reordering the adjacency lists of the graphs by nondecreasing vertex degree tends to produce deeper search trees, and backtracking tends to perform better on deeper trees than on shallower trees, because *pruning* a search tree at a level close to the root cuts off a larger subtree of a deep search tree than of a shallow search tree. Study the effect of vertex ordering on the performance of the backtracking algorithms for graph and subgraph isomorphism.

7.2 Implement a variant of the backtracking algorithm for graph isomorphism to decide whether or not two graphs are isomorphic, without enumerating all graph isomorphism mappings between the two graphs.

7.3 Replace the refinement procedure in the backtracking algorithm for subgraph isomorphism, which implements the simplified necessary condition for extending a subgraph isomorphism mapping given in Lemma 7.17, by the necessary condition given in Lemma 7.16 and implement the modified algorithm for subgraph isomorphism.

7.4 Perform experiments on random graphs to compare the efficiency of the backtracking algorithm for subgraph isomorphism using the two necessary conditions of Exercise 7.3.

7.5 Implement a variant of the backtracking algorithm for subgraph isomorphism to find one isomorphism of a graph as a subgraph of another graph, instead of enumerating all such subgraph isomorphism mappings.

7.6 Computation of all maximal common subgraphs of two graphs can also be reduced to finding all maximal cliques in the graph product, but in the product of the line graphs of the two graphs, as discussed in [199]. As a matter of fact, with the exception of K_3 and $K_{1,3}$, any two connected graphs with isomorphic line graphs are isomorphic [245, 366]. Implement a variant of the algorithm for maximal common induced subgraph isomorphism to find a maximum common subgraph isomorphism of two graphs. Notice that the line graph product can grow very large, even for small graphs.

7.7 Implement a variant of the algorithm for maximal common subgraph isomorphism to find a maximum common subgraph isomorphism of two graphs, instead of enumerating all maximal common subgraph isomorphisms and then choosing a largest one.

7.8 The most widely used representation of molecular graphs is the connection table, which contains a list of all of the atoms within a chemical structure together with bond information. Hydrogen atoms are often excluded, since their presence or absence can be deduced from the bond orders and atomic types. For instance, the standard representation of a molecular graph adopted by the NIST Chemistry WebBook [222], which is available at <http://webbook.nist.gov/chemistry/>, is a text file consisting of three lines followed by a *count line* containing the number n of nonhydrogen atoms and the number m of bonds, separated by a space and followed by n lines containing the *atom block* and m lines containing the *bond block*. A full description of the PDB format can be found at <http://www.mddi.com/downloads/literature/ctfile.pdf>. Implement procedures to read the connection table of a molecular graph in LEDA and to write a LEDA graph as a connection table.

7.9 A more comprehensive connection table format has been adopted by the Protein Data Bank [38], which is available at <http://www.rcsb.org/pdb/>, in order to represent crystallographic as well as structural information. A full description of the PDB connection table

format can be found at the same Internet address. Implement procedures to read the PDB file of a molecular graph in LEDA and to write a LEDA graph as a PDB file.

7.10 Perform isomorphism experiments on molecular graphs extracted from the NIST Chemistry WebBook and the Protein Data Bank. See Exercises 7.8 and 7.9.

Part IV

Appendices

A. An Overview of LEDA

Libraries are great; use them whenever they do the job. Start with your system library, then search other libraries for appropriate functions.

—Jon Bentley [31]

The detailed exposition of algorithms on trees and graphs made in this book is based on a full C++ implementation of all of the algorithms. These implementations are, in turn, based on the LEDA library of efficient C++ data structures and algorithms. A brief overview of the small subset of LEDA used throughout this book is provided in this appendix, in order to facilitate reading and is aimed at making the book self-contained. The reader is referred to [234, 236] for a more comprehensive description of LEDA.

A.1 Introduction

LEDA is a library of efficient C++ data structures and algorithms, developed over the last decade at the Max-Plank-Institute für Informatik in Saarbrücken, Germany, which is becoming a de facto standard for upper-undergraduate and graduate courses on graph algorithms throughout the world. As a matter of fact, LEDA allows the student, lecturer, researcher, and practitioner to complement algorithmic graph theory with actual implementation and experimentation, building upon a thorough library of efficient implementations of modern data structures and fundamental algorithms.

Another widely used library of C++ data structures and algorithms is STL, the ANSI C++ Standard Template Library [243]. STL includes efficient implementations of sequential container (*deque*, *list*, *stack*,

vector) and associative container (*map*, *set*) classes as well as generic algorithms, providing a large number of operations that can be applied to these classes and to the built-in *array* class as well. In much the same spirit, BGL, the Boost Graph Library [209, 302] that is being considered for adoption into the ANSI C++ Standard, includes efficient implementations of several basic graph algorithms and data structures, designed in the generic programming style of STL.

When it comes to combinatorial and geometric computing, though, LEDA goes much beyond STL and BGL by providing efficient implementations of graphs and their supporting data structures, together with a large number of graph algorithms (including shortest path, minimum spanning tree, bipartite matching, maximum flow, minimum cut, and graph drawing algorithms), while still providing efficient implementations of sequential container (*list*, *queue*, *stack*) and associative container (*dictionary*, *map*, *set*) classes. Furthermore, LEDA provides the most efficient data structures known for implementing each container class, even allowing the user to choose among different implementations for the same class. For instance, the *dictionary* class includes a default implementation by (2,4)-trees [26], together with a choice of several alternative implementations including AVL trees [1], *BB*[α] trees [48, 248], red-black trees [143], randomized search trees [292], and skip lists [262]. Last, but not least, LEDA includes an interface for graphical input and output in several platforms, which facilitates the editing and visualization of graphs and the interactive demonstration and animation of graph algorithms.

A.2 Data Structures

LEDA consists of a set of *container* classes, a set of algorithms to operate over these container classes, and a set of *iterators* to access the contents of a container.

Containers are just objects that hold other objects. A *sequential container* maintains a first element, a second element, and so on through a last element. Examples of sequential containers are the *list* class, which provides a linear list; the *stack* class, which provides a stack; and the *queue* class, which provides a single-ended queue.

An *associative container*, on the other hand, supports fast lookup of an object held in the container. Examples of associative containers are the *dictionary*, *d_array*, *h_array*, *map*, and *map2* classes, which provide access to the information associated with a unique key; and the *set* class, which provides access to unique elements.

Some of the containers provided by LEDA do not fit, though, in the previous distinction between sequential and associative containers. These include the *p_queue* class, which provides a single-ended priority queue; the *two_tuple*, *three_tuple*, and *four_tuple* classes, which provide n -tuples of elements; the *array* and *array2* classes, which provide one-dimensional dynamic arrays and two-dimensional static arrays, respectively; the *sortseq* class, providing sorted sequences, which support most of the operations of the *p_queue* and *dictionary* classes together with further search operations and some operations of the *list* class; and the *graph* class, which provides graphs together with efficient implementations of most fundamental graph algorithms.

Further containers provided by LEDA include the *integer* and *rational* classes, which provide an exact realization of integer and rational numbers, respectively, where the C++ types *int* and *long int* are just an approximation of integer numbers; the *bigfloat* and *real* classes, which provide a better approximation of real numbers than the C++ types *float*, *double*, and *long double*; and the *vector*, *integer_vector*, *matrix*, and *integer_matrix* classes, which provide one-dimensional and two-dimensional arrays of numbers together with basic linear algebra operations.

Algorithms operate on containers, and include capabilities for initializing, sorting, searching, and transforming the content of containers. Most LEDA data structures are implemented as C++ generic classes, which are parametrized by the type of the container, and encapsulate initializing, sorting, searching, and transforming algorithms, which are also implemented as C++ generic functions, parametrized again by the container element type. Most LEDA graph algorithms are implemented as C++ generic functions and wrapped in a separate *graph_alg* class, though, as further discussed below.

The operations shown in Fig. A.1 are common to the sequential container classes *list*, *stack*, and *queue*, and the associative container classes *dictionary* and *set*, as well as the container classes *p_queue*

and *sort_seq*. Furthermore, the associative container classes *d_array* and *h_array* provide *size* and *clear* but no *empty* operation, the *map* and *map2* classes only provide the *clear* operation, the *array* class only provides the *size* operation, and the *graph* class provides all three operations, where the *number_of_nodes* operation gives the order and the *number_of_edges* operation gives the size of the graph.

<i>size()</i>	returns the number of elements in the container
<i>empty()</i>	returns true if the container is empty and false otherwise
<i>clear()</i>	makes the container empty

Fig. A.1. Operations common to most LEDA container classes.

A.2.1 Linear Lists

The *list* class provides a linear list, that is, a sequence of elements. The *list()* constructor creates an empty list. Some of the operations provided by the *list* class are shown in Fig. A.2.

The *list* class includes a default implementation by doubly linked linear lists. Operations *front*, *back*, *push*, *append*, *pop*, *Pop*, *conc*, *size*, *empty* take $O(1)$ time; *rank*, *reverse*, *remove*, *permute*, *merge*, *unique*, *clear* take $O(n)$ time; *bucket_sort* takes $O(n + j - i)$ time; and *sort*, *merge_sort* take $O(n \log n)$ time, where n is the size or length of the list, and i and j are the minimal and maximal values of a given function on the elements of the list. The representation uses $O(n)$ space.

The efficient implementation of some of these operations requires the use of *items*, which are, roughly, addresses of containers. Most of the previous operations have a counterpart which acts upon list items. For instance, *first* and *last* are the counterpart of *front* and *back*, respectively, and return the first and the last item in the list, assuming the list is not empty. Furthermore, operation *contents(it)* returns the element contained at item *it* of the list.

Iteration over the items or the elements of a list is implemented by LEDA macros. In a macro call of the form *forall_items(it,L)*, the items of list *L* are successively assigned to item *it*, and in a macro call of the form *forall(x,L)*, the elements of *L* are successively assigned to variable *x*.

<i>front()</i>	returns the first element in the list, assuming the list is not empty
<i>back()</i>	returns the last element in the list, assuming the list is not empty
<i>rank(x)</i>	returns the rank of element x in the list, that is, the integer position of the first occurrence of x in the list, or zero if x does not occur in the list
<i>push(x)</i>	inserts element x at the front of the list
<i>append(x)</i>	inserts element x at the rear of the list
<i>pop()</i>	deletes and returns the first element in the list, assuming the list is not empty
<i>Pop()</i>	deletes and returns the last element in the list, assuming again that the list is not empty
<i>remove(x)</i>	removes all occurrences of element x from the list
<i>conc(L)</i>	deletes the elements of another list L and inserts them at the rear of the list
<i>reverse()</i>	reverses the sequence of elements of the list
<i>permute()</i>	performs a random permutation of the elements of the list
<i>sort()</i>	sorts the list by quick sort using the default linear ordering on the elements
<i>merge_sort()</i>	sorts the list by merge sort using the default linear ordering on the elements
<i>bucket_sort(f)</i>	sorts the list by bucket sort according to the integer function f defined on the elements
<i>merge(L)</i>	merges the list with L using the default linear ordering on the elements
<i>unique()</i>	removes duplicate elements from the list, assuming the list is already sorted according to the default linear ordering on the elements

Fig. A.2. Some of the operations provided by the *list* class.

A.2.2 Stacks

The *stack* class provides a stack, that is, a sequence of elements which are inserted and deleted at the same end (the top) of the sequence. The *stack()* constructor creates an empty stack. The operations provided by the *stack* class are shown in Fig. A.3.

<i>top()</i>	returns the top element in the stack, assuming the stack is not empty
<i>pop()</i>	deletes and returns the top element in the stack, assuming the stack is not empty
<i>push(x)</i>	inserts element x at the top of the stack

Fig. A.3. Operations provided by the *stack* class.

The *stack* class includes a default implementation by singly linked linear lists. Operations *top*, *pop*, *push*, *size*, *empty* take $O(1)$ time; and *clear* takes $O(n)$ time, where n is the size of the stack. The representation uses $O(n)$ space.

A.2.3 Queues

The *queue* class provides a single-ended queue, that is, a sequence of elements which are inserted at one end (the rear) and deleted at the other end (the front) of the sequence. The *queue()* constructor creates an empty queue. The operations provided by the *queue* class are shown in Fig. A.4.

<i>top()</i>	returns the front element in the queue, assuming the queue is not empty
<i>pop()</i>	deletes and returns the front element in the queue, assuming the queue is not empty
<i>append(x)</i>	inserts element x at the rear end of the queue

Fig. A.4. Operations provided by the *queue* class.

The *queue* class includes a default implementation by singly linked linear lists. Operations *top*, *pop*, *append*, *size*, *empty* take $O(1)$ time; and *clear* takes $O(n)$ time, where n is the size of the queue. The representation uses $O(n)$ space.

A.2.4 Priority Queues

The *p_queue* class provides a single-ended priority queue, that is, a queue of elements with both information and a priority associated with each element, where there is a linear order defined on the priorities. The *p_queue()* constructor creates an empty priority queue, based on the linear order defined by the global *compare* function. The operations provided by the *p_queue* class are shown in Fig. A.5.

The *p_queue* class includes a default implementation by Fibonacci heaps [119]. Operations *prio*, *inf*, *find_min*, *change_inf*, *size*, *empty* take $O(1)$ time; *insert*, *decrease_p* take amortized $O(1)$ time; *del_item*, *del_min* take amortized $O(\log n)$ time; and *clear* takes $O(n)$ time,

<i>prio(x)</i>	returns the priority of element <i>x</i>
<i>inf(x)</i>	returns the information associated with element <i>x</i>
<i>insert(i,p)</i>	inserts and returns an element with information <i>i</i> and priority <i>p</i> in the priority queue
<i>find_min()</i>	returns an element with the minimum priority, or <i>nil</i> if the priority queue is empty
<i>del_item(x)</i>	deletes element <i>x</i> from the priority queue
<i>del_min()</i>	deletes and returns the priority of an element with the minimum priority, assuming the priority queue is not empty
<i>change_inf(x,i)</i>	makes <i>i</i> the new information associated with element <i>x</i> in the priority queue
<i>decrease_p(x,p)</i>	makes <i>p</i> the new priority of element <i>x</i> , assuming <i>x</i> already belongs to the priority queue with a priority not smaller than <i>p</i>

Fig. A.5. Operations provided by the *p_queue* class.

where *n* is the size of the priority queue. The representation uses $O(n)$ space.

Alternative implementations include pairing heaps [118, 308], *k*-ary and binary heaps, lists, buckets, redistributive heaps [3], monotone heaps, and Emde-Boas trees [335]. The default implementation can also be selected by the implementation parameter *f_heap*, and the alternative implementations are selected by *p_heap*, *k_heap*, *bin_heap*, *list_pq*, *b_heap*, *r_heap*, *m_heap*, and *eb_tree*, respectively.

A.2.5 Tuples

The *two_tuple*, *three_tuple*, and *four_tuple* classes provide respectively 2-tuples, 3-tuples, and 4-tuples. The operations provided by the *tuple* classes are shown in Fig. A.6.

<i>first()</i>	returns the first component in the 2-tuple, 3-tuple, or 4-tuple
<i>second()</i>	returns the second component in the 2-tuple, 3-tuple, or 4-tuple
<i>third()</i>	returns the third component in the 3-tuple or 4-tuple
<i>fourth()</i>	returns the fourth component in the 4-tuple

Fig. A.6. Operations provided by the *two_tuple*, *three_tuple*, and *four_tuple* classes.

The comparison operator \equiv is defined for 2-tuples, 3-tuples, and 4-tuples, together with a *compare* lexicographic comparison operator and a *Hash* hashing function. The *tuple* classes include an obvious

default implementation by wrapping the components as private data members of the class. Operations *first*, *second*, *third*, and *fourth* take $O(1)$ time. The representation uses $O(1)$ space times the total size of the components.

A.2.6 Arrays

The *array* class provides one-dimensional dynamic arrays. The *array(a,b)* constructor creates an array of $b - a + 1$ elements indexed by integers in the range $[a..b]$; *array(n)* creates an array of n elements indexed by $[0..n - 1]$; and *array()* creates an empty array of elements. Some of the operations provided by the *array* class are shown in Fig. A.7.

<code>[i]</code>	returns a reference to element $A(i)$ of the array, assuming that $a \leq i \leq b$
<code>resize(a,b)</code>	redefines the array to be indexed by integers in the range $[a..b]$
<code>low()</code>	returns the minimal index a of the array
<code>high()</code>	returns the maximal index b of the array
<code>init(x)</code>	sets all elements in the array to x
<code>swap(i,j)</code>	permutes elements $A(i)$ and $A(j)$ in the array
<code>permute()</code>	performs a random permutation of the elements of the array
<code>sort()</code>	sorts the array by quick sort using the default linear ordering on the elements
<code>binary_search(x)</code>	returns the index i such that $A(i) = x$, or $a - 1$ if element x does not belong to the array, assuming the array is already sorted according to the default linear ordering on the elements

Fig. A.7. Some of the operations provided by the *array* class.

The *array* class includes a default implementation by C++ vectors. Operations `[]`, *low*, *high*, *swap* take $O(1)$ time; *binary_search* takes $O(\log n)$ time; *init*, *permute* take $O(n)$ time; and *sort* takes $O(n \log n)$ time, where $n = b - a + 1$. Further, operation *resize* takes time linear in the maximum between the old and the new size of the array. The representation uses $O(n)$ space times the size of the elements.

Further, the *array2* class provides two-dimensional static arrays. The *array2(a,b,c,d)* constructor creates a two-dimensional array of $b - a + 1$ times $d - c + 1$ elements indexed by integers in the range $[a..b]$ and $[c..d]$; and *array2(n,m)* creates a two-dimensional array of

$n \times m$ elements indexed by $[0..n - 1]$ and $[0..m - 1]$. The operations provided by the *array2* class are shown in Fig. A.8.

$[i,j]$	returns a reference to element $A(i,j)$ of the array, assuming that $a \leq i \leq b$ and $c \leq j \leq d$
<i>low1()</i>	returns the minimal index a of the array
<i>high1()</i>	returns the maximal index b of the array
<i>low2()</i>	returns the minimal index c of the array
<i>high2()</i>	returns the maximal index d of the array

Fig. A.8. Operations provided by the *array2* class.

The *array2* class includes a default implementation by C++ vectors. Operations $[]$, *low1*, *high1*, *low2*, *high2* take $O(1)$ time. The representation uses $O(nm)$ space times the size of the elements, where $n = b - a + 1$ and $m = d - c + 1$.

A.2.7 Dictionaries

The *dictionary* class provides an associative container, consisting of a set of elements with both information and a unique key associated with each element, where there is a linear order defined on the keys and the information associated with an element is retrieved on the basis of its key. The *dictionary()* constructor creates an empty dictionary, based on the linear order defined by the global *compare* function. The operations provided by the *dictionary* class are shown in Fig. A.9.

The *dictionary* class includes a default implementation by (2,4)-trees [26]. Operations *key*, *inf*, *change_inf*, *empty*, *size* take $O(1)$ time; *insert*, *lookup*, *access*, *del*, *del_item* take $O(\log n)$ time; and *clear* takes $O(n)$ time, where n is the size of the dictionary. The representation uses $O(n)$ space.

Alternative implementations include AVL trees [1], *BB*[α] trees [48, 248], red-black trees [143], randomized search trees [292], and skip lists [262]. The default implementation can also be selected by the implementation parameter *ab_tree*, and the alternative implementations are selected by *avl_tree*, *bb_tree*, *rb_tree*, *rs_tree*, and *skiplist*, respectively.

<i>key</i> (<i>x</i>)	returns the key associated with element <i>x</i>
<i>inf</i> (<i>x</i>)	returns the information associated with element <i>x</i>
[<i>x</i>]	returns a reference to the information associated with element <i>x</i>
<i>insert</i> (<i>k</i> , <i>i</i>)	inserts and returns an element with key <i>k</i> and information <i>i</i> in the dictionary, replacing the element (if any) with key <i>k</i>
<i>lookup</i> (<i>k</i>)	returns the element with key <i>k</i> in the dictionary, or <i>nil</i> if there is no such element
<i>access</i> (<i>k</i>)	returns the information associated with key <i>k</i> in the dictionary, assuming there is such an element
<i>del</i> (<i>k</i>)	deletes the element with key <i>k</i> from the dictionary, if there is such an element
<i>del_item</i> (<i>x</i>)	deletes element <i>x</i> from the dictionary, assuming <i>x</i> belongs to the dictionary
<i>change_inf</i> (<i>x</i> , <i>i</i>)	makes <i>i</i> the information associated with element <i>x</i> , assuming <i>x</i> belongs to the dictionary

Fig. A.9. Operations provided by the *dictionary* class.

A.2.8 Dictionary Arrays

The *d_array* class provides a second kind of associative container, also consisting of a set of elements with both information and a unique key associated with each element, where there is a linear order defined on the keys. The *d_array()* constructor creates an empty dictionary array. The operations provided by the *d_array* class are shown in Fig. A.10.

[<i>k</i>]	returns a reference to the information associated with key <i>k</i> in the dictionary array
[<i>k</i>]= <i>i</i>	inserts an element with key <i>k</i> and information <i>i</i> in the dictionary array, replacing the element (if any) with key <i>k</i> , and returns a reference to the information associated with the element
<i>defined</i> (<i>k</i>)	returns true if there is an element with key <i>k</i> in the dictionary array and false otherwise
<i>undefine</i> (<i>k</i>)	deletes the element with key <i>k</i> from the dictionary array and leaves key <i>k</i> undefined

Fig. A.10. Operations provided by the *d_array* class.

The *d_array* class includes a default implementation by randomized search trees [292]. Operation *size* takes $O(1)$ time; [], *defined*, *undefine* take $O(\log n)$ time; and *clear* takes $O(n)$ time, where n is the size of the domain. The representation uses $O(n)$ space.

Alternative implementations include (2,4)-trees [26], AVL trees [1], *BB*[α] trees [48, 248], unbalanced binary trees, red-black trees

[143], and skip lists [262]. The default implementation can also be selected by the implementation parameter *rs_tree*, and the alternative implementations are selected by *ab_tree*, *avl_tree*, *bb_tree*, *bin_tree*, *rb_tree*, and *skiplist*, respectively.

Iteration over defined elements is implemented by LEDA macros. In a macro call of the form *forall_defined(k,A)*, the keys associated with the elements in the dictionary array are successively assigned to variable *k*, and in a macro call of the form *forall(i,A)*, the information associated with the elements of the dictionary array are successively assigned to variable *i*.

A.2.9 Hashing Arrays

The *h_array* class provides a third kind of associative container, consisting of a set of elements with both an information and a unique key associated with each element, where there is a hashing function defined on the keys. The *h_array()* constructor creates an empty hashing array. The operations provided by the *h_array* class are shown in Fig. A.11.

<i>[k]</i>	returns a reference to the information associated with key <i>k</i> in the hashing array
<i>[k]=i</i>	inserts an element with key <i>k</i> and information <i>i</i> in the hashing array, replacing the element (if any) with key <i>k</i> , and returns a reference to the information associated with the element
<i>defined(k)</i>	returns true if there is an element with key <i>k</i> in the hashing array and false otherwise
<i>undefined(k)</i>	deletes the element with key <i>k</i> from the hashing array and leaves key <i>k</i> undefined

Fig. A.11. Operations provided by the *h_array* class.

The *h_array* class includes a default implementation by hashing with chaining. Operation *size* takes $O(1)$ time; *[]*, *defined*, *undefined* take expected $O(1)$ time; and *clear* takes $O(n)$ time, where *n* is the size of the hashing array. The representation uses $O(n)$ space.

There is an alternative implementation by dynamic perfect hashing [93, 117]. The default implementation can also be selected by the implementation parameter *ch_hash*, and the alternative implementation is selected by *dp_hash*.

Again, iteration over defined elements is implemented by LEDA macros. In a macro call of the form `forall_defined(k,A)`, the keys associated with the elements in the hashing array are successively assigned to variable k , and in a macro call of the form `forall(i,A)`, the information associated with the elements of the hashing array are successively assigned to variable i .

A.2.10 Maps

The *map* class provides still a fourth kind of associative container, consisting of a set of elements with both information and a unique key associated with each element, where keys are either of the type *int* or of a *pointer* or *item* type. The `map()` constructor creates an empty map. The operations provided by the *map* class are shown in Fig. A.12.

<code>[k]</code>	returns a reference to the information associated with key k in the map
<code>[k]=i</code>	inserts an element with key k and information i in the map, replacing the element (if any) with key k , and returns a reference to the information associated with the element
<code>defined(k)</code>	returns true if there is an element with key k in the map and false otherwise

Fig. A.12. Operations provided by the *map* class.

The *map* class includes a default implementation by hashing with chaining and table doubling. Operations `[]`, `defined` take expected $O(1)$ time; and `clear` takes $O(n)$ time, where n is the number of elements in the map. The representation uses $O(n)$ space.

Iteration over defined elements is also implemented by LEDA macros. In a macro call of the form `forall_defined(k,A)`, the keys associated with the elements in the map are successively assigned to variable k , and in a macro call of the form `forall(i,A)`, the information associated with the elements of the map are successively assigned to variable i .

Further, the *map2* class provides an associative container consisting of a set of elements with both information and a unique ordered pair of keys associated with each element. The `map2()` constructor creates an empty two-dimensional map. The operations provided by the *map2* class are shown in Fig. A.13.

(k,l)	returns a reference to the information associated with key pair (k,l) in the two-dimensional map
$(k,l)=i$	inserts an element with key pair (k,l) and information i in the two-dimensional map, replacing the element (if any) with key pair (k,l) , and returns a reference to the information associated with the element
$\text{defined}(k,l)$	returns true if there is an element with key pair (k,l) in the two-dimensional map and false otherwise

Fig. A.13. Operations provided by the *map2* class.

The *map2* class includes a default implementation by hashing with chaining and table doubling. Operations $()$, *defined* take expected $O(1)$ time; and *clear* takes $O(n)$ time, where n is the number of elements in the two-dimensional map. The representation uses $O(n)$ space.

A.2.11 Sets

The *set* class provides a set of elements. The *set()* constructor creates an empty set. The operations provided by the *set* class are shown in Fig. A.14.

<i>insert</i> (x)	inserts element x in the set
<i>del</i> (x)	deletes element x from the set
<i>member</i> (x)	returns true if x belongs to the set and false otherwise
<i>choose</i> ()	returns some element of the set, assuming the set is not empty
<i>join</i> (T)	returns the union of the set with set T
<i>diff</i> (T)	returns the difference of the set minus set T
<i>intersect</i> (T)	returns the intersection of the set with set T
<i>syndiff</i> (T)	returns the symmetric difference of the set and set T

Fig. A.14. Operations provided by the *set* class.

The comparison operators \leqslant , \geqslant , \equiv , \neq , $<$, and $>$ are defined for the *set* class. The *set* class is implemented by randomized search trees [292]. Operations *empty*, *size*, *choose* take $O(1)$ time; *insert*, *del*, *member* take expected $O(\log n)$ time; *clear* takes $O(n)$ time; and *join*, *diff*, *intersect*, *syndiff* take expected $O(n \log n)$ time, where n is the size of the set.

Iteration over the elements of a set is also implemented by LEDA macros. In a macro call of the form *forall*(*x,S*), the elements of set *S* are successively assigned to variable *x*.

A.2.12 Partitions

The *partition* class provides a partition of a finite set of items into disjoint subsets, called *blocks*. The *partition()* constructor creates an empty partition of an empty set of elements. The operations provided by the *partition* class are shown in Fig. A.15.

<i>make_block()</i>	returns a new item and inserts a singleton block containing the item in the partition
<i>find</i> (<i>p</i>)	returns a canonical representative item of the block that contains item <i>p</i>
<i>size</i> (<i>p</i>)	returns the size of the block containing item <i>p</i>
<i>number_of_blocks()</i>	returns the number of blocks in the partition
<i>same_block</i> (<i>p,q</i>)	returns true if items <i>p</i> and <i>q</i> belong to the same block of the partition and false otherwise
<i>union_blocks</i> (<i>p,q</i>)	combines the blocks of the partition containing items <i>p</i> and <i>q</i>
<i>split</i> (<i>L</i>)	splits all blocks containing items in a list of items <i>L</i> of the partition into singleton blocks

Fig. A.15. Operations provided by the *partition* class.

The *partition* class includes a default implementation by the union-find data structure with weighted union and path compression [318, 321]. Operation *number_of_blocks* takes $O(1)$ time; a sequence of n operations *make_block* and a total of $m > n$ operations *find*, *size*, *same_block*, *union_blocks* takes $O(m\alpha(m,n))$ time, where the value of $\alpha(m,n)$, the inverse Ackermann function, is less than or equal to 4 for all practical purposes; and operation *split* takes time linear in the size of the blocks. The representation uses $O(n)$ space, where n is now the size of the set.

A.2.13 Sorted Sequences

The *sortseq* class provides a sorted sequence, that is, a sequence of elements with both information and a unique key associated with each

element, where there is a linear order defined on the keys. The *sortseq()* constructor creates an empty sorted sequence, based on the linear order defined by the global *compare* function. Some of the operations provided by the *sortseq* class are shown in Fig. A.16.

<i>key(it)</i>	returns the key associated with item <i>it</i>
<i>inf(it)</i>	returns the information associated with item <i>it</i>
<i>insert(k,i)</i>	inserts and returns an element with key <i>k</i> and information <i>i</i> in the sorted sequence, replacing the element (if any) with key <i>k</i>
<i>lookup(k)</i>	returns the element with key <i>k</i> in the sorted sequence, or <i>nil</i> if there is no such element
<i>del(k)</i>	deletes the element with key <i>k</i> from the sorted sequence, if there is such an element
<i>del_item(it)</i>	deletes item <i>it</i> from the sorted sequence, assuming <i>it</i> belongs to the sequence
<i>change_inf(it,i)</i>	makes <i>i</i> the information associated with item <i>it</i> , assuming <i>it</i> belongs to the sorted sequence

Fig. A.16. Some of the operations provided by the *sortseq* class.

Sorted sequences also offer so-called finger search operations, as well as operations for splitting and merging sorted sequences. The *sortseq* class includes a default implementation by skip lists [262]. Operations *empty*, *size*, *key*, *inf*, *del_item*, *change_inf* take $O(1)$ time; *insert*, *lookup*, *del* take expected $O(\log n)$ time; and *clear* takes $O(n)$ time, where n is the size of the sorted sequence. The representation uses $O(n)$ space times the size of the elements.

Alternative implementations of sorted sequences include (2,4)-trees [26], BB[α] trees [48, 248], red-black trees [143], and randomized search trees [292]. The default implementation can also be selected by the implementation parameter *skiplist*, and the alternative implementations are selected by *ab_tree*, *bb_tree*, *rb_tree*, and *rs_tree*, respectively.

A.2.14 Graphs

The *graph* class provides directed graphs, and also undirected graphs represented by bidirected graphs. The *graph()* constructor creates an empty graph. Some of the operations provided by the *graph* class are shown in Fig. A.17.

<i>outdeg(v)</i>	returns the number of arcs going out of vertex v
<i>indeg(v)</i>	returns the number of arcs coming into vertex v
<i>source(e)</i>	returns the source vertex of arc e
<i>target(e)</i>	returns the target vertex of arc e
<i>opposite(v,e)</i>	returns <i>target(e)</i> if vertex v is the source of arc e and <i>source(e)</i> otherwise
<i>number_of_nodes()</i>	returns the order of the graph
<i>number_of_edges()</i>	returns the size of the graph
<i>choose_node()</i>	returns a vertex of the graph at random, or <i>nil</i> if the graph is empty
<i>choose_edge()</i>	returns an arc of the graph at random, or <i>nil</i> if the graph is empty
<i>all_nodes()</i>	returns a <i>list</i> of all the vertices of the graph
<i>all_edges()</i>	returns a <i>list</i> of all the arcs of the graph
<i>adj.edges(v)</i>	returns a <i>list</i> of all the arcs going out of vertex v
<i>in.edges(v)</i>	returns a <i>list</i> of all the arcs coming into vertex v
<i>adj.nodes(v)</i>	returns a <i>list</i> of all the vertices adjacent to vertex v
<i>new_node()</i>	inserts and returns a vertex in the graph
<i>new_edge(v,w)</i>	inserts and returns an arc in the graph going out of vertex v and coming into vertex w
<i>del_node(v)</i>	deletes vertex v from the graph, together with all those arcs going out of or coming into vertex v
<i>del_edge(e)</i>	deletes arc e from the graph
<i>del_all_nodes()</i>	deletes all vertices from the graph
<i>del_all_edges()</i>	deletes all arcs from the graph
<i>sort_nodes(cmp)</i>	sorts the graph by quick sort according to the linear ordering <i>cmp</i> defined on the vertices
<i>bucket_sort_nodes(f)</i>	sorts the graph by bucket sort according to the integer function <i>f</i> defined on the vertices
<i>sort_edges(cmp)</i>	sorts the graph by quick sort according to the linear ordering <i>cmp</i> defined on the arcs
<i>bucket_sort_edges(f)</i>	sorts the graph by bucket sort according to the integer function <i>f</i> defined on the arcs
<i>empty()</i>	returns true if the graph is empty and false otherwise
<i>clear()</i>	makes the graph empty

Fig. A.17. Some of the operations provided by the *graph* class.

The *graph* class includes a default implementation by doubly linked lists of vertices and arcs. Operations *outdeg*, *indeg*, *source*, *target*, *opposite*, *number_of_nodes*, *number_of_edges*, *choose_node*, *choose_edge*, *new_node*, *new_edge*, *del_node*, *del_edge*, *empty* take $O(1)$ time; *adj.edges*, *adj.nodes* take time linear in the outdegree of the vertex; *in.edges* takes time linear in the indegree of the vertex; *all_nodes*, *all_edges*, *del_all_nodes*, *del_all_edges*, *bucket_sort_nodes*, *bucket_sort_edges*, *clear* take $O(n + m)$ time; operation *sort_nodes* takes $O(n \log n)$ time; and *sort_edges* takes $O(m \log m)$ time, where

n is the order and m is the size of the graph. The representation uses $O(n + m)$ space.

There are further operations supporting iteration over the vertices and arcs of a graph. An alternative form of iteration is implemented by LEDA macros. In a macro call of the form *forall_nodes*(v, G), the vertices of graph G are successively assigned to variable v ; in a macro call *forall_edges*(e, G), the arcs of graph G are successively assigned to variable e ; in a macro call *forall_rev_nodes*(v, G), the vertices of graph G are successively assigned to variable v in reverse order; in a macro call *forall_rev_edges*(e, G), the arcs of graph G are successively assigned to variable e in reverse order; in a macro call *forall_out_edges*(e, v), the arcs going out of vertex v are successively assigned to variable e ; in a macro call *forall_in_edges*(e, w), the arcs coming into vertex w are successively assigned to variable e ; and in a macro call of the form *forall_adj_nodes*(w, v), the vertices adjacent to vertex v are successively assigned to variable w .

Information can be associated with the vertices and arcs of a graph by defining appropriate arrays, matrices, or maps of vertices and arcs. While the former are static data structures, valid only for the vertices and arcs contained in the graph at the moment of creation of the array or matrix, the latter are dynamic data structures, which are also valid for vertices and arcs inserted later in the graph.

The *node_array* class provides a static array of information associated with the vertices of a graph. The *node_array()* constructor creates an empty static array of information to be associated with the vertices of a graph, the *node_array*(G) constructor creates a static array of information indexed by the vertices of graph G , and the *node_array*(G, x) constructor creates a static array of information indexed by the vertices of graph G and initializes all entries to the value of variable x . Some of the operations provided by the *node_array* class are shown in Fig. A.18.

The *node_array* class includes a default implementation by C++ vectors and an internal numbering of the vertices of the graph. Operations *get_graph*, [] take $O(1)$ time; and *init* takes $O(n)$ time, where n is the order of the graph. The representation uses $O(n)$ space times the size of the information associated with the vertices.

<code>get_graph()</code>	returns a reference to the graph which the array of vertices is associated with
<code>[v]</code>	returns a reference to the information associated with vertex v
<code>init(G)</code>	makes the array valid for all vertices of graph G
<code>init(G, x)</code>	makes the array valid for all vertices of graph G and initializes all entries to the value of variable x

Fig. A.18. Some of the operations provided by the *node_array* class.

The *edge_array* class provides a static array of information associated with the arcs of a graph. The *edge_array()* constructor creates an empty static array of information to be associated with the arcs of a graph, the *edge_array(G)* constructor creates a static array of information indexed by the arcs of graph G , and the *edge_array(G, x)* constructor creates a static array of information indexed by the arcs of graph G and initializes all entries to the value of variable x . Some of the operations provided by the *edge_array* class are shown in Fig. A.19.

<code>get_graph()</code>	returns a reference to the graph which the array of arcs is associated with
<code>[e]</code>	returns a reference to the information associated with arc e
<code>init(G)</code>	makes the array valid for all arcs of graph G
<code>init(G, x)</code>	makes the array valid for all arcs of graph G and initializes all entries to the value of variable x

Fig. A.19. Some of the operations provided by the *edge_array* class.

The *edge_array* class includes a default implementation by C++ vectors and an internal numbering of the arcs of the graph. Operations *get_graph*, [] take $O(1)$ time; and *init* takes $O(m)$ time, where m is the size the graph. The representation uses $O(m)$ space times the size of the information associated with the arcs.

The *node_matrix* class provides a static two-dimensional array of information associated with the vertices of a graph. The *node_matrix()* constructor creates an empty static two-dimensional array of information to be associated with the vertices of a graph, the *node_matrix(G)* constructor creates a static two-dimensional array of information indexed by the vertices of graph G , and the *node_matrix(G, x)* constructor creates a static two-dimensional array of information indexed by the vertices of graph G and initializes all entries to the value of vari-

able x . Some of the operations provided by the *node_matrix* class are shown in Fig. A.20.

<i>get_graph()</i>	returns a reference to the graph which the two-dimensional array of vertices is associated with
$[v]$	returns a reference to the <i>node_array</i> associated with vertex v
(v,w)	returns a reference to the information associated with vertices v and w
<i>init(G)</i>	makes the two-dimensional array valid for all vertex pairs of graph G
<i>init(G,x)</i>	makes the two-dimensional array valid for all vertex pairs of graph G and initializes all entries to the value of variable x

Fig. A.20. Some of the operations provided by the *node_matrix* class.

The *node_matrix* class includes a default implementation by *node_array* vectors and an internal numbering of the vertices of the graph. Operations *get_graph*, $[]$, $()$ take $O(1)$ time; and *init* takes $O(n^2)$ time, where n is the order of the graph. The representation uses $O(n^2)$ space times the size of the information associated with the vertices.

The *node_map* and *edge_map* classes provides dynamic arrays of information associated with the vertices and the arcs of a graph, respectively. The *node_map()* and *edge_map()* constructors create empty dynamic arrays of information to be associated with the vertices and arcs of a graph, the *node_map(G)* and *edge_map(G)* constructors create dynamic arrays of information indexed by the vertices and arcs of graph G , and the *node_map(G,x)* and *edge_map(G,x)* constructors create dynamic arrays of information indexed by the vertices and arcs of graph G and initialize all entries to the value of variable x . Some of the operations provided by the *node_map* and *edge_map* classes are shown in Fig. A.21.

The *node_map* and *edge_map* classes includes a default implementation by hashing based on an internal numbering of the vertices and arcs of the graph. Operations *get_graph*, *init* take $O(1)$ time; and $[]$ takes expected $O(1)$ time. The representation uses $O(n)$ and $O(m)$ space times the size of the information associated with the vertices and arcs, respectively, where n is the order and m is the size of the graph.

<i>get_graph()</i>	returns a reference to the graph which the dynamic array is associated with
$[v]$	returns a reference to the information associated with vertex v
$[e]$	returns a reference to the information associated with arc e
<i>init(G)</i>	makes the dynamic array valid for all vertices or arcs of graph G
<i>init(G,x)</i>	makes the dynamic array valid for all vertices or arcs of graph G and initializes all entries to the value of variable x

Fig. A.21. Some of the operations provided by the *node_map* and *edge_map* classes.

The *node_map2* class provides a dynamic two-dimensional array of information associated with the vertices of a graph. The *node_map2()* constructor creates an empty dynamic two-dimensional array of information to be associated with the vertices of a graph, constructor *node_map2(G)* creates a dynamic two-dimensional array of information indexed by the vertices of graph G , and the *node_map2(G,x)* constructor creates a dynamic two-dimensional array of information indexed by the vertices of graph G and initializes all entries to the value of variable x . Some of the operations provided by the *node_map2* class are shown in Fig. A.22.

<i>get_graph()</i>	returns a reference to the graph which the two-dimensional array of vertices is associated with
(v,w)	returns a reference to the information associated with vertices v and w
<i>defined(v,w)</i>	returns true if there is an entry in the two-dimensional array for vertices v and w and false otherwise
<i>init(G)</i>	makes the two-dimensional array valid for all vertex pairs of graph G
<i>init(G,x)</i>	makes the two-dimensional array valid for all vertex pairs of graph G and initializes all entries to the value of variable x

Fig. A.22. Some of the operations provided by the *node_map2* class.

The *node_map2* class includes a default implementation by hashing based on an internal numbering of the vertices of the graph. Operations *get_graph*, *init* take $O(1)$ time; and *()*, *defined* take expected $O(1)$ time. The representation uses $O(n + m)$ space times the size of the information associated with the vertices, where n is the order and m is the size of the graph.

An alternative form of associating information with vertices and arcs consists of using graphs parametrized by the type of the information associated with the vertices and arcs of the graph. The *GRAPH* class provides parametrized directed graphs, and also undirected graphs represented by bidirected graphs. The *GRAPH(V,E)* constructor creates an empty graph parametrized by vertex information type *V* and arc information type *E*. The *GRAPH* class is derived from the *graph* class, and inherits all of the operations provided by the *graph* class. Further operations provided by the *GRAPH* class are shown in Fig. A.23.

<i>inf(v)</i>	returns the information associated with vertex <i>v</i>
<i>[v]</i>	returns a reference to the information associated with vertex <i>v</i>
<i>inf(e)</i>	returns the information associated with arc <i>e</i>
<i>[e]</i>	returns a reference to the information associated with arc <i>e</i>
<i>node_data()</i>	makes the information associated with the vertices of the graph available as a <i>node_array</i>
<i>edge_data()</i>	makes the information associated with the arcs of the graph available as an <i>edge_array</i>
<i>assign(v,x)</i>	makes <i>x</i> the information associated with vertex <i>v</i>
<i>assign(e,x)</i>	makes <i>x</i> the information associated with arc <i>e</i>
<i>new_node(x)</i>	inserts and returns a vertex in the graph with information <i>x</i> associated
<i>new_edge(v,w,x)</i>	inserts and returns an arc in the graph going out of vertex <i>v</i> and coming into vertex <i>w</i> with information <i>x</i> associated

Fig. A.23. Some of the additional operations provided by the *GRAPH* class.

The *GRAPH* class includes a default implementation by doubly linked lists of vertices and arcs. Operations *inf*, *[]*, *node_data*, *edge_data*, *assign*, *new_node*, *new_edge* take $O(1)$ time. The representation uses $O(n)$ space times the size of the information associated with the vertices plus $O(m)$ space times the size of the information associated with the arcs, where *n* is the order and *m* is the size of the graph.

LEDA also provides linear lists, priority queues, and partitions of the set of vertices of a graph, which have a more efficient implementation than the corresponding generic classes. The *node_list* class provides a linear list of the vertices of a graph that can contain each vertex of the graph at most once, with the additional restriction that each vertex of a graph is contained in at most one linear list of vertices. The

node_list() constructor creates an empty list of vertices. Some of the operations provided by the *node_list* class are shown in Fig. A.24.

The *node_list* class includes a default implementation by a doubly linked list of vertices, together with a map of vertices indexed by the vertices of the graph. Operations *append*, *push*, *insert*, *pop*, *del*, *head*, *tail*, *succ*, *pred*, *empty* take $O(1)$ time; *member* takes expected $O(1)$ time; and *clear* takes $O(n)$ time, where n is the size or length of the linear list of vertices.

Iteration over the vertices in a linear list of vertices is also implemented by LEDA macros. In a macro call of the form *forall*(x, L), the vertices of L are successively assigned to variable x .

<i>append(v)</i>	appends vertex v to the list of vertices
<i>push(v)</i>	inserts vertex v at the front of the list of vertices
<i>insert(v,w)</i>	inserts vertex v right after vertex w , assuming w belongs to the list
<i>pop()</i>	deletes and returns the first vertex from the list, assuming the list of vertices is not empty
<i>del(v)</i>	deletes vertex v from the list of vertices, assuming there is such a vertex
<i>member(v)</i>	returns true if vertex v belongs to the list of vertices and false otherwise
<i>head()</i>	returns the first vertex in the list of vertices, or <i>nil</i> if the list is empty
<i>tail()</i>	returns the last vertex in the list of vertices, or <i>nil</i> if the list is empty
<i>succ(v)</i>	returns the successor of vertex v in the list of vertices, assuming the list is not empty
<i>pred(v)</i>	returns the predecessor of vertex v in the list of vertices, assuming the list is not empty
<i>empty()</i>	returns true if the list of vertices is empty and false otherwise
<i>clear()</i>	makes the list of vertices empty

Fig. A.24. Some of the operations provided by the *node_list* class.

The *node_pq* class provides a priority queue of the vertices of a graph that can contain each vertex of the graph at most once, with the additional restriction that only one priority queue of vertices may be used for a graph. The *node_pq(G)* constructor creates a priority queue of the vertices of graph G . The operations provided by the *node_pq* class are shown in Fig. A.25.

The *node_pq* class includes a default implementation by priority queues—implemented, in turn, by binary heaps—and arrays of ver-

tices. Operations *prio*, *inf*, *member*, *find_min*, *size*, *empty* take $O(1)$ time; *insert*, *del*, *del_min*, *decrease_p* take $O(\log m)$ time; and *clear* takes $O(m)$ time, where m is the size of the priority queue. The representation uses $O(n)$ space, where n is the order of the graph.

The *node_partition* class provides a partition of the vertices of a graph. The *node_partition(G)* constructor creates a trivial partition of the vertices of graph G , containing a singleton block for each vertex of the graph. The operations provided by the *node_partition* class are shown in Fig. A.26.

<i>prio(v)</i>	returns the priority of vertex v
<i>inf(v)</i>	returns the information associated with vertex v
<i>member(v)</i>	returns true if vertex v belongs to the priority queue and false otherwise
<i>insert(v,p)</i>	inserts vertex v with priority p in the priority queue
<i>find_min()</i>	returns a vertex with the minimum priority, or <i>nil</i> if the priority queue is empty
<i>del(v)</i>	deletes vertex v from the priority queue
<i>del_min()</i>	deletes and returns a vertex with the minimum priority, assuming the priority queue is not empty
<i>decrease_p(v,p)</i>	makes p the new priority of vertex v , assuming v already belongs to the priority queue with a priority not smaller than p

Fig. A.25. Operations provided by the *node-pq* class.

<i>find(v)</i>	returns a canonical representative vertex of the block containing vertex v
<i>size(v)</i>	returns the size of the block containing vertex v
<i>same_block(v,w)</i>	returns true if vertices v and w belong to the same block of the partition and false otherwise
<i>union_blocks(v,w)</i>	combines the blocks of the partition containing vertices v and w
<i>split(L)</i>	splits all blocks containing vertices in a list of vertices L of the partition into singleton blocks
<i>make_rep(v)</i>	makes vertex v the canonical representative of the block containing it

Fig. A.26. Operations provided by the *node-partition* class.

The *node_partition* class includes a default implementation by a partition—implemented in turn by the union-find data structure with weighted union and path compression—together with a static array of items of the partition indexed by the vertices of the graph. Initialization takes $O(n)$ time; operations *find*, *size*, *same_block*, *union_blocks*,

make_rep take amortized $O(\alpha(n))$ time, where n is the order of the graph and the value of $\alpha(n)$, the inverse Ackermann function, is less than or equal to 4 for all practical purposes; and *split* takes time linear in the size of the blocks. The representation uses $O(n)$ space.

Further data structures provided by LEDA include compressed Boolean arrays; strings; random bits, characters, and numbers; data structures for graphical input and output; and computational geometry data structures. The reader is referred to [236] for details.

BFS($G, s, dist$)

computes the distance $dist$ in graph G of all vertices reachable from vertex s during a breadth-first traversal of the graph in $O(n + m)$ time [232] [235, Sect. 7.3], and returns a list of all visited vertices

BICONNECTED_COMPONENTS($G, compnum$)

computes the biconnected components of an undirected graph G in $O(n + m)$ time [74] and returns the number of biconnected components, together with the component number *compnum* to which each edge belongs

COMPONENTS($G, compnum$)

computes the connected components of an undirected graph G in $O(n + m)$ time [232] and returns the number of connected components, together with the component number *compnum* to which each edge belongs

DFS_NUM($G, dfstnum, compnum$)

computes the number *dfstnum* and *compnum* in which the vertices are first and last visited, respectively, during a depth-first traversal of graph G in $O(n + m)$ time [235, Sect. 7.3] [317] and returns a list of tree edges in the corresponding depth-first forest of the graph

STRONG_COMPONENTS

computes the strong components of graph G in $O(n + m)$ time [232] [235, Sect. 7.4.2] and returns the number of strong components, together with the strong component number *compnum* to which each vertex belongs

TOPSORT(G, ord)

computes a topological sort *ord* of an acyclic graph G in $O(n + m)$ time [176] and returns true if the graph is indeed acyclic and false otherwise

TRANSITIVE_CLOSURE(G)

computes the transitive closure of graph G in $O(n + m)$ time [134]

Fig. A.27. Some basic graph algorithms provided by LEDA.

A.3 Fundamental Graph Algorithms

LEDA provides efficient implementations of a large number of graph algorithms, including traversal, connectivity, shortest path, minimum spanning tree, bipartite matching, maximum flow, minimum cut, algorithms for planar graphs, and graph drawing algorithms. These graph

algorithms are implemented as C++ generic functions, and accept both graphs and parametrized graphs as arguments.

Basic graph algorithms provided by LEDA include topological sorting; depth-first and breadth-first traversal; connected, biconnected, and strongly connected components; and transitive closure algorithms. Some of them are summarized in Fig. A.27.

Further graph algorithms provided by LEDA include several single-source shortest path and all-pairs shortest paths algorithms, as well as minimum spanning tree algorithms. Some of them are summarized in Fig. A.28.

ACYCLIC_SHORTEST_PATH($G, s, cost, dist, pred$)

computes the distance $dist$ and the shortest path arcs $pred$ in an acyclic graph G with arc costs $cost$ of all vertices reachable from vertex s in $O(n + m)$ time [235, Sect. 7.5.4]

ALL_PAIRS_SHORTEST_PATHS($G, cost, dist$)

computes the distance $dist$ between all reachable pairs of vertices in graph G with no negative-cost cycles in $O(nm + n^2 \log n)$ time [235, Sect. 7.5.10] and returns true if the graph has indeed no negative-cost cycles and false otherwise

BELLMAN_FORD($G, s, cost, dist, pred$)

computes the distance $dist$ and the shortest path arcs $pred$ in graph G with arc costs $cost$ of all vertices reachable from vertex s in $O(nm)$ time [29] [235, Sect. 7.5.7–7.5.9] and returns false if there is a negative-cost cycle reachable from s in the graph and true otherwise

DIJKSTRA($G, s, cost, dist, pred$)

computes the distance $dist$ and the shortest path arcs $pred$ in graph G with nonnegative arc costs $cost$ of all vertices reachable from vertex s in $O(m + n \log n)$ time [94] [235, Sect. 6.6]

MIN_SPANNING_TREE($G, cost$)

computes a spanning tree of minimum cost of an undirected graph G with edge costs $cost$ in $O(n + m)$ time [204] [235, Sect. 6.8] and returns a list of the edges in the minimum spanning tree

SPANNING_TREE(G)

computes a spanning tree of an undirected graph G in $O(n + m)$ time [232] and returns a list of the edges in the spanning tree

Fig. A.28. Further graph algorithms for shortest paths and minimum spanning trees provided by LEDA.

Additional graph algorithms provided by LEDA include several matching algorithms in bipartite and general graphs, as well as combinatorial optimization algorithms for computing a maximum or minimum flow in a network, and minimum cut algorithms. Some of them are summarized in Fig. A.29.

MAX_CARD_BIPARTITE_MATCHING(G)

computes a matching of maximum cardinality in a bipartite graph G in $O(\sqrt{nm})$ time [7, 165] or in $O(nm)$ time [114], and returns the list of edges in the matching

MAX_CARD_MATCHING(G)

computes a matching of maximum cardinality in an undirected graph G in $O(nm\alpha(n,m))$ time [103, 121], and returns the list of edges in the matching

MAX_FLOW(G,s,t,cap)

computes a maximum (s,t) -flow in a network (G,s,t,cap) with arc capacities cap in $O(n^2\sqrt{m})$ time [128] [235, Sect. 7.10] and returns the value of the flow

MAX_WEIGHT_ASSIGNMENT($G,cost,pot$)

computes a perfect matching of maximum cost in an undirected graph G with edge costs $cost$ in $O(n(m+n\log n))$ time [235, Sect. 7.8] and returns the list of edges in the matching and a correctness certificate pot

MAX_WEIGHT_BIPARTITE_MATCHING($G,cost,pot$)

computes a matching of maximum cost in a bipartite graph G with edge costs $cost$ in $O(n(m+n\log n))$ time [235, Sect. 7.8] and returns the list of edges in the matching, together with a correctness certificate pot

MIN_CUT($G,weight$)

computes a cut of minimum weight in graph G with arc weights $weight$ in $O(nm + n^2\log n)$ time [11, 310] and returns the value of the cut

Fig. A.29. Further graph algorithms for flows in networks and cuts and matchings in graphs provided by LEDA.

Beware that the maximum cardinality bipartite matching algorithms actually transform the representation of the graph by reordering adjacency lists, thereby modifying the combinatorial embedding of the graph. Beware also that some of the layout algorithms for planar graphs included in LEDA transform first the graph into a planar map, modifying again the combinatorial embedding of the graph. In particular, applying a straight-line layout or a visibility representation layout to an ordered tree transforms it into a *different* ordered tree.

LEDA also provides efficient implementations of algorithms for planar graphs and graph drawing algorithms. The reader is referred to [236, Chap. 7–8] for details.

A.4 A Simple Representation of Trees

The LEDA representation of graphs may be used for representing trees as well, although LEDA graphs offer many operations that do not make much sense for trees—for instance, most of those dealing with embedded graphs, because a tree has only one face—and more space-efficient representations for trees could be adopted, as already

discussed in Sect. 1.4. Nevertheless, the choice in this book has been to define a simple *tree* class, which inherits the representation and operations from the *graph* class and also provides additional operations for trees.

The *tree* class provides nonempty rooted trees. The *tree()* constructor creates an empty tree, and the *tree(G)* constructor creates a tree representation of graph G , assuming G is the graph representation of a nonempty rooted tree. All operations provided by the *graph* class can be applied to trees as well. Further operations provided by the *tree* class are shown in Fig. A.30.

<i>parent(v)</i>	returns the parent of node v , or <i>nil</i> if v is the root of the tree
<i>is_root(v)</i>	returns true if node v is the root of the tree and false otherwise
<i>is_leaf(v)</i>	returns true if node v is a leaf of the tree and false otherwise
<i>root()</i>	returns the root node of the tree
<i>first_child(v)</i>	returns the first child of node v in the tree, or <i>nil</i> if v is a leaf node
<i>last_child(v)</i>	returns the last child of node v in the tree, or <i>nil</i> if v is a leaf node
<i>next_sibling(v)</i>	returns the next sibling of node v in the tree, or <i>nil</i> if v is either the root node or a last child of the tree
<i>previous_sibling(v)</i>	returns the previous sibling of node v in the tree, or <i>nil</i> if v is either the root node or a first child of the tree
<i>is_first_child(v)</i>	returns true if node v is a first child in the tree and false otherwise
<i>is_last_child(v)</i>	returns true if node v is a last child in the tree and false otherwise
<i>number_of_children(v)</i>	returns the number of children of node v in the tree

Fig. A.30. Operations provided by the *tree* class.

The *tree* class includes a default implementation derived from the LEDA *graph* class. Initialization from a LEDA graph takes $O(n)$ time; operations *parent*, *is_root*, *is_leaf*, *first_child*, *last_child*, *next_sibling*, *previous_sibling*, *is_first_child*, *is_last_child*, *number_of_children* take $O(1)$ time; *root* takes time linear in the depth of the tree; and *clear* takes $O(n)$ time, where n is the size of the tree. The representation uses $O(n)$ space.

Recall from Sect. 1.1 that a connected undirected graph $G = (V, E)$ with n vertices and m edges is a nonempty undirected tree if and only if $n = m + 1$. As a matter of fact, in an empty undirected tree both $n = 0$ and $m = 0$.

In the case of directed, rooted trees, the previous condition is still necessary but no longer sufficient for a graph to be a tree, because a node could have more than one parent. Recall from Definition 1.36 that a connected graph $T = (V, E)$ is a tree if the underlying undirected graph has no cycles and for all nodes $v \in V$, there is a path in the graph from a distinguished node $\text{root}[T]$ to node v .

Lemma A.1. *A connected graph $T = (V, E)$ with n vertices and m arcs is a tree if and only if $n = m + 1$ and $\text{indeg}(v) \leq 1$ for all vertices $v \in V$.*

Proof. Let $T = (V, E)$ be a connected graph with n vertices and $m = n - 1$ arcs such that $\text{indeg}(v) \leq 1$ for all vertices $v \in V$. Suppose that $\text{indeg}(v) = 1$ for all vertices $v \in V$. Then, $\sum_{v \in V} \text{indeg}(v) = \sum_{v \in V} 1 = n$. But $\sum_{v \in V} \text{indeg}(v) = m$ by Theorem 1.5, contradicting the assumption that $n = m + 1$. Therefore, there is a vertex $u \in V$ with $\text{indeg}(u) = 0$.

Now, since the graph is connected there is, for all vertices $v \in V$, an undirected path from vertex u to vertex v . Suppose that the path from vertex u to some vertex $v \in V$ is not a directed path. Then, there must be some vertex $w \in V$ along the path with $\text{indeg}(w) \geq 2$, contradicting the assumption that $\text{indeg}(v) \leq 1$ for all vertices $v \in V$. Therefore, there is a directed path from vertex u to vertex v , for all vertices $v \in V$.

Suppose now that there is a cycle $[v_1, \dots, v_2, \dots, v_1]$ in the underlying undirected graph. If the cycle is directed, vertex u cannot belong to the cycle, because $\text{indeg}(u) = 0$, and there must be some vertex $w \in V$ in the cycle such that the directed path from vertex u up to, but not including, vertex w is disjoint from the cycle. Then, $\text{indeg}(w) \geq 2$, contradicting the assumption that $\text{indeg}(v) \leq 1$ for all vertices $v \in V$. Otherwise, if the cycle is not directed, there must be some vertex $w \in V$ along either the undirected path $[v_1, \dots, v_2]$ or the undirected path $[v_2, \dots, v_1]$ with $\text{indeg}(w) \geq 2$, contradicting again the assumption that $\text{indeg}(v) \leq 1$ for all vertices $v \in V$. Therefore, there are no cycles in the underlying undirected graph, and T is indeed a tree.

Conversely, let $T = (V, E)$ be a tree with n nodes and m arcs. By Theorem 1.43, $n = m + 1$. Suppose now that there is a node $w \in V$ with $\text{indeg}(w) > 1$. Let $v_1, v_2 \in V$ be nodes such that $(v_1, w), (v_2, w) \in E$, and let v be the least common ancestor of nodes v_1 and v_2 in T . Then, the paths $[v, \dots, v_1, w]$ and $[v, \dots, v_2, w]$ together constitute a cycle in

the underlying undirected graph, contradicting the assumption that T is a tree. Therefore, $\text{indeg}(v) \leq 1$ for all nodes $v \in V$. \square

The following procedure *is_tree* returns true if the graph is the representation of a rooted tree and false otherwise. Since the LEDA procedure *Is_Connected* takes $O(n)$ time to determine whether the graph is connected, procedure *is_tree* also takes $O(n)$ time.

429a ⟨tree graph representation 429a⟩≡

```
bool is_tree(
    const graph& G)
{
    node v;
    forall_nodes(v,G)
        if ( G.indeg(v) > 1 ) return false; // nonunique parent
    return ( G.number_of_nodes() == G.number_of_edges() + 1
        & Is_Connected(G) );
}
```

The following *tree* class is derived from the LEDA *graph* class. The destructor, copy constructor, and copy assignment operator, as well as the operations on LEDA graphs, are all inherited from the *graph* class.

429b ⟨tree graph representation 429a⟩+≡

```
class tree : public graph {
public:
    tree() : graph() { }

    tree( const graph& G ) : graph( G )
    {
        if ( !is_tree(G) )
            error_handler(1,"Graph is not a tree");
    }

    node parent( const node v ) const;
    bool is_root( const node v ) const;
    bool is_leaf( const node v ) const;
    node root() const;
    node first_child( const node v ) const;
    node last_child( const node v ) const;
    node next_sibling( const node v ) const;
    node previous_sibling( const node v ) const;
    bool is_first_child( const node v ) const;
    bool is_last_child( const node v ) const;
    int number_of_children( const node v ) const;
};
```

In the graph representation of a tree $T = (V, E)$, the parent of a node $v \in V$ is the unique node $u \in V$ such that $(u, v) \in E$.

430a $\langle\text{tree graph representation 429a}\rangle + \equiv$
`node tree::parent(const node v) const
{
 if ((*this).indeg(v) == 0) return nil;
 return (*this).source((*this).first_in_edge(v));
}`

The root of a tree $T = (V, E)$ is the only node $v \in V$ which is not the target of any arc of the form $(u, v) \in E$, that is, the only node $v \in V$ with $\text{indeg}(v) = 0$.

430b $\langle\text{tree graph representation 429a}\rangle + \equiv$
`bool tree::is_root(const node v) const
{
 return (*this).indeg(v) == 0;
}`

On the other hand, a node $v \in V$ is a leaf of a tree $T = (V, E)$ if there is no arc of the form $(v, w) \in E$, that is, if $\text{outdeg}(v) = 0$.

430c $\langle\text{tree graph representation 429a}\rangle + \equiv$
`bool tree::is_leaf(const node v) const
{
 return (*this).outdeg(v) == 0;
}`

The root of a tree can be found in time linear in the depth of the tree by starting off with some node—for instance, a node chosen at random—and then following the path up to the root of the tree.

Notice that operation *root* could be implemented to take $O(1)$ time instead, by just extending the graph representation of a tree with an explicit pointer to the root node. However, the root of a tree must still be found when reading the tree from a LEDA graph window; for instance, in the interactive demonstration of graph algorithms. Besides, all algorithms on trees discussed in this book take $\Omega(n)$ time anyway, the time needed to either generate or input the tree.

430d $\langle\text{tree graph representation 429a}\rangle + \equiv$
`node tree::root() const
{
 node v = (*this).choose_node();
 while (!is_root(v))
 v = parent(v);
}`

```

return v;
}

```

The first and the last child of a node in a tree are the target of the first and the last arc going out of the node, respectively, according to the relative order of the children of the node fixed by the graph representation of the tree.

431a ⟨tree graph representation 429a⟩ +≡
node tree::first_child(const node v) const
{
 if ((*this).outdeg(v) ≡ 0) **return** nil;
 return (*this).target((*this).first_adj_edge(v));
}

431b ⟨tree graph representation 429a⟩ +≡
node tree::last_child(const node v) const
{
 if ((*this).outdeg(v) ≡ 0) **return** nil;
 return (*this).target((*this).last_adj_edge(v));
}

In the same sense, the next and the previous sibling of a node in a tree are the target of the next and the previous arc going out of the parent of the node, respectively, according to the relative order of the arcs going out of the parent node fixed by the graph representation of the tree.

431c ⟨tree graph representation 429a⟩ +≡
node tree::next_sibling(const node v) const
{
 if ((*this).indeg(v) ≡ 0) **return** nil;
 edge e = (*this).adj_succ((*this).first_in_edge(v));
 if (e ≡ nil) **return** nil;
 return (*this).target(e);
}

431d ⟨tree graph representation 429a⟩ +≡
node tree::previous_sibling(const node v) const
{
 if ((*this).indeg(v) ≡ 0) **return** nil;
 edge e = (*this).adj_pred((*this).first_in_edge(v));
 if (e ≡ nil) **return** nil;
 return (*this).target(e);
}

Now, the first child of the parent of a node is said to be a *first child* node, and the last child of the parent of a node is said to be a *last child* node.

432a ⟨tree graph representation 429a⟩+≡
bool tree::*is_first_child*(**const** node *v*) **const**
{
 if ((*this).*is_root*(*v*)) **return** **false**;
 return (*this).*first_child*((*this).*parent*(*v*)) ≡ *v*;
}

432b ⟨tree graph representation 429a⟩+≡
bool tree::*is_last_child*(**const** node *v*) **const**
{
 if ((*this).*is_root*(*v*)) **return** **true**;
 return (*this).*last_child*((*this).*parent*(*v*)) ≡ *v*;
}

The number of children of a node in a tree is just the outdegree of the corresponding vertex in the graph representation of the tree.

432c ⟨tree graph representation 429a⟩+≡
int tree::*number_of_children*(**const** node *v*) **const**
{
 return (*this).*outdeg*(*v*);
}

Iteration over all children nodes $w \in V$ of node $v \in V$ in a tree $T = (V, E)$ is implemented by the LEDA macro for iterating over the target vertex w of all arcs of the form $(v, w) \in E$ in the graph representation of the tree. In a macro call of the form *forall_children*(*w,v*), the children of node *v* are successively assigned to node *w*.

432d ⟨tree graph representation 429a⟩+≡
#define *forall_children*(*w,v*) *forall_adj_nodes*(*w,v*)

A *TREE* class is also defined, providing nonempty rooted trees parametrized by the type of the information associated with the nodes and edges of the tree. The *TREE* class is derived from the *GRAPH* class which is, in turn, derived from the *graph* class. The destructor, copy constructor, and copy assignment operator, as well as the operations on LEDA parametrized graphs, are all inherited from the *GRAPH* class. Further operations provided by the *TREE* class are identical to the additional operations provided by the *tree* class.

432e ⟨tree graph representation 429a⟩+≡

```

template<class V,class E>
class TREE : public GRAPH<V,E> {
public:
    TREE<V,E>() { }

    TREE( const GRAPH<V,E>& G) : GRAPH<V,E>( G )
    {
        if ( notis_tree(G) ) error_handler(1,"Graph is not a tree");
    }

    node parent( const node v ) const
    {
        if ( (*this).indeg(v) == 0 ) return nil;
        return (*this).source((*this).first_in_edge(v));
    }

    bool is_root( const node v ) const
    {
        return (*this).indeg(v) == 0;
    }

    bool is_leaf( const node v ) const
    {
        return (*this).outdeg(v) == 0;
    }

    node root() const
    {
        node v = (*this).choose_node();
        while ( notis_root(v) )
            v = parent(v);
        return v;
    }

    node first_child( const node v ) const
    {
        if ( (*this).outdeg(v) == 0 ) return nil;
        return (*this).target((*this).first_adj_edge(v));
    }

    node last_child( const node v ) const
    {
        if ( (*this).outdeg(v) == 0 ) return nil;
        return (*this).target((*this).last_adj_edge(v));
    }

    node next_sibling( const node v ) const
    {
        if ( (*this).indeg(v) == 0 ) return nil;
        edge e = (*this).adj_succ((*this).first_in_edge(v));
        if ( e == nil ) return nil;
        return (*this).target(e);
    }
}

```

```

node previous_sibling( const node v ) const
{
    if ( (*this).indeg(v) == 0 ) return nil;
    edge e = (*this).adj_pred((*this).first_in_edge(v));
    if ( e == nil ) return nil;
    return (*this).target(e);
}

bool is_first_child( const node v ) const
{
    if ( (*this).is_root(v) ) return false;
    return (*this).first_child((*this).parent(v)) == v;
}

bool is_last_child( const node v ) const
{
    if ( (*this).is_root(v) ) return true;
    return (*this).last_child((*this).parent(v)) == v;
}

int number_of_children( const node v ) const
{
    return (*this).outdeg(v);
}

```

};

A.5 A Simple Implementation of Radix Sort

Some of the algorithms for tree isomorphism and related problems discussed in Chap. 4, required radix sorting a list of arrays of integers. Radix sort is based on bucket sort, and consists of sorting the list of arrays of integers in several stages, using bucket sort at each stage.

Bucket Sort

LEDA does not provide an implementation of radix sort, although it includes an efficient implementation of bucket sort which is, however, of a rather low level. The following, straightforward implementation of bucket sort is a more convenient starting point for a simple LEDA implementation of radix sort.

Let $\text{ord} : E \rightarrow \text{int}$ be a function with $\text{ord}(x) \in [i..j]$ for all elements x of list L . Bucket sorting L consists of maintaining an array $bucket$

of $j - i + 1$ lists of elements, initially empty; appending each element x of L to bucket number $k = \text{ord}(x) - i + 1$; and then collecting the elements in sorted order by concatenating all the buckets back into a single list.

The following algorithm sorts the elements of list L using bucket sort, implementing the previous procedure. The elements of L are distributed into buckets, and the buckets are concatenated back into the sorted list L . The identity function is provided as default value for the ord parameter. Space efficiency of the implementation follows from both pop and conc operations being destructive.

435

```
<subroutines 40>+≡
template<class E>
void straight_bucket_sort(
    list<E>& L,
    int i,
    int j,
    int (*ord)(const E&) = id)
{
    int n = j - i + 1; // number of buckets needed
    array<list<E>> bucket(1,n);
    while (  $\neg L.\text{empty}()$  ) {
        E x = L.pop();
        int k = ord(x) - i + 1; // element x belongs in bucket k
        if (  $k \geq 1 \wedge k \leq n$  ) {
            bucket[k].append(x);
        } else {
            error_handler(1,"bucket sort: value out of range");
        }
    }
    for ( i = 1; i  $\leq n$ ; i++ )
        L.conc(bucket[i]); // destructive
    <double-check bucket sort 436b>
}
```

Remark A.2. Notice that bucket sort is a *stable* sorting method, that is, if $\text{ord}(x) = \text{ord}(y)$ and x precedes y in L , then x also precedes y in the bucket sorted list L , for all elements x and y of L . As a matter of fact, in the previous bucket sorting algorithm, the elements of L are appended to appropriate buckets in order, and this order is preserved when concatenating the buckets.

Bucket sort is often called just on a list of elements. A procedure call of the form *straight_bucket_sort(L,ord)* is the same as the procedure call *straight_bucket_sort(L,i,j,ord)*, where i and j are respectively

the minimum and maximum values of $\text{ord}(x)$ as x ranges over the elements of list L . Again, the identity function is provided as default value for the ord parameter.

436a

```
<subroutines 40>+≡
template<class E>
void straight_bucket_sort(
    list<E>& L,
    int (*ord)(const E&) = id)
{
    if (L.empty()) return;
    int i = ord(L.head());
    int j = i;
    E x;
    forall(x,L) {
        int k = ord(x);
        if (k < i) i = k;
        if (k > j) j = k;
    }
    straight_bucket_sort(L,i,j,ord);
}
```

The following double-check of bucket sort, although being redundant, gives some reassurance of the correctness of the implementation. It verifies that L is sorted, that is, that $\text{ord}(x) \leq \text{ord}(\text{succ}(x))$ for all but the last element x of list L .

436b

```
<double-check bucket sort 436b>≡
{ list_item it;
forall_items(it,L) {
    if (it ≠ L.last() ∧ ord(L[it]) > ord(L[L.succ(it)])) {
        error_handler(1,"Wrong implementation of bucket sort");
    } } }
```

Lemma A.3. *The algorithm for bucket sorting runs in $O(n + j - i)$ time using $O(j - i)$ additional space, where n is the length of list L and i, j are respectively the minimum and maximum values of $\text{ord}(x)$ as x ranges over the elements of L .*

Proof. The first loop ranges over the elements of L , and thus takes $O(n)$ time, and the second loop ranges over the buckets, thus taking $O(j - i)$ time. Therefore, the algorithm runs in $O(n + j - i)$ time. Further, the algorithm uses $O(j - i)$ additional space, because an array of $j - i + 1$ buckets is first allocated, the elements of L are popped from L

and appended to their respective bucket, and the buckets are concatenated back into L , where both *pop* and *conc* operations are destructive.

The double-check of bucket sorting runs in $O(n)$ time using $O(1)$ additional space. \square

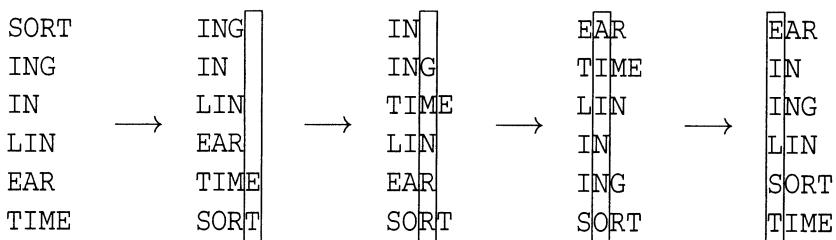
It follows that $j - i \leq n$ is a sufficient condition for bucket sort to run in $O(n)$ time.

Corollary A.4. *The algorithm for bucket sorting runs in $O(n)$ time upon a list L of n elements if $j - i \leq n$, where $\text{ord}(x) \in [i..j]$ for all elements x of L .*

Radix Sort

Radix sort is a stable sorting method consisting of k passes of bucket sort, on the last element, the previous-to-last element, and so on, until bucket sorting on the first element of each list, where k is the maximum among the lengths of the lists of integers to be sorted. Since such a procedure requires direct access to individual elements in each list, though, the list of lists of integers to be sorted is represented by a list of arrays of integers instead.

Example A.5. Radix sorting the list of arrays of integers $[[83, 79, 82, 84], [73, 78, 71], [73, 78], [76, 73, 78], [69, 65, 82], [84, 73, 77, 69]]$, which are the ASCII codes of the characters in the list of character strings [SORT,ING, IN, LIN, EAR, TIME], proceeds as follows.



The leftmost column shows the list of character strings corresponding to the list of arrays of integers to be sorted, and the remaining columns show the result of bucket sorting on increasingly significant character positions. The character position bucket sorted on to produce each list of character strings from the previous one is highlighted.

Remark A.6. The previous algorithm is known as LSD (least significant digit) radix sort, because the least significant element in the arrays of integers is bucket sorted first. MSD (most significant digit) radix sort, on the contrary, consists of bucket sorting on the most significant element first, and then radix sorting each group of array suffixes sharing a common first element. While with LSD radix sort all elements of the arrays are inspected, only the distinguishing prefixes of the arrays are inspected with MSD radix sort, although at the expense of decomposing the sorting problem into a large number of subproblems. With LSD radix sort, bucket sorting is applied several times to the same list of arrays of integers instead.

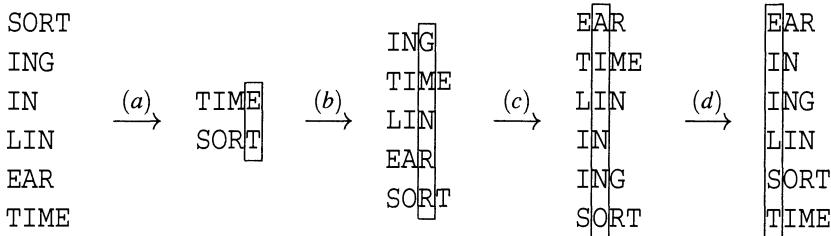
Now, given a list L of n arrays of at most k integers each, where $\text{ord}(x) \in [i..j]$ for all integers x in each array in list L and $j - i \leq n$, the previous radix sorting algorithm still makes k passes of bucket sort, each taking $O(n)$ time, and therefore runs in $O(kn)$ time. This is efficient when all arrays have approximately the same length, because the total length of the arrays of integers to be sorted is $\sum_{i=1}^n O(k) = O(\sum_{i=1}^n k) = O(kn)$. However, radix sorting a list L of n arrays of integers, one of them of length n and the other ones of only a few integers, would take $O(n^2)$ time, although the total length of the arrays of integers to be sorted is, in this case, $O(n) + \sum_{i=2}^n O(1) = O(n)$.

The reason for the lack of efficiency is that in the previous radix sorting algorithm, a large number of empty buckets might be inspected at each pass of bucket sort. Nevertheless, the previous algorithm can be extended to a simple radix sorting algorithm running in time linear in the total length of the arrays of integers to be sorted, as follows.

Let maxlen be the maximum length among all arrays of integers to be sorted. Inspecting empty buckets can be avoided by building for each position i in the arrays, with $1 \leq i \leq \text{maxlen}$, a list $\text{LEN}[i]$ of the arrays of length i and also a sorted list $\text{FULL}[i]$ of the integers appearing at the i th position in the arrays.

Then, bucket sorting the list L of arrays on the i th component proceeds by concatenating at the front of L the list $\text{LEN}[i]$ of arrays of length i , distributing the resulting list L into buckets according to the i th component and, finally, concatenating back into L all nonempty buckets, that is, those buckets corresponding to $\text{FULL}[i]$.

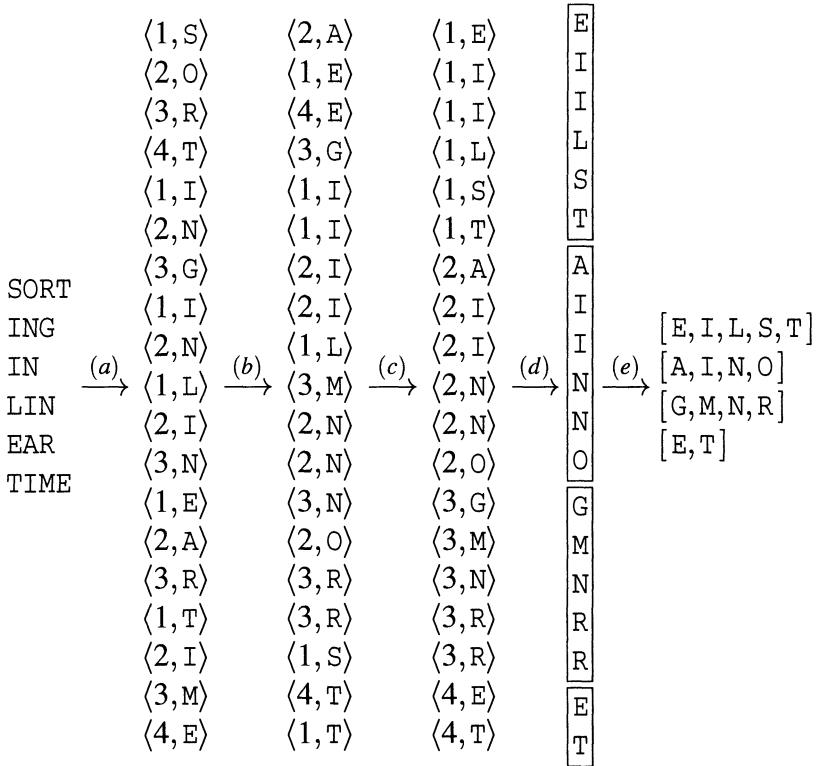
Example A.7. More efficient radix sorting of the list of arrays of integers of Example A.5, which are the ASCII codes of the characters in the list of character strings [SORT, ING, IN, LIN, EAR, TIME].



(a) List $\text{LEN}[4]; L = [\underline{\text{S}\text{O}\text{R}\text{T}}, \underline{\text{T}\text{I}\text{M}\text{E}}]; []$ is distributed into buckets according to the fourth component, but only those buckets corresponding to $\text{FULL}[4] = [\underline{\text{E}}, \underline{\text{T}}]$ are concatenated back into L . (b) List $\text{LEN}[3]; L = [\underline{\text{I}\text{N}\text{G}}, \underline{\text{L}\text{I}\text{N}}, \underline{\text{E}\text{A}\text{R}}]; [\underline{\text{T}\text{I}\text{M}\text{E}}, \underline{\text{S}\text{O}\text{R}\text{T}}]$ is distributed into buckets according to the third component, but only the buckets corresponding to $\text{FULL}[3] = [\underline{\text{G}}, \underline{\text{M}}, \underline{\text{N}}, \underline{\text{R}}]$ are concatenated back into L . (c) List $\text{LEN}[2]; L = [\underline{\text{I}\text{N}}]; [\underline{\text{I}\text{N}\text{G}}, \underline{\text{T}\text{I}\text{M}\text{E}}, \underline{\text{L}\text{I}\text{N}}, \underline{\text{E}\text{A}\text{R}}, \underline{\text{S}\text{O}\text{R}\text{T}}]$ is distributed into buckets according to the second component, but only those buckets corresponding to $\text{FULL}[2] = [\underline{\text{A}}, \underline{\text{I}}, \underline{\text{N}}, \underline{\text{O}}]$ are concatenated back into L . (d) List $\text{LEN}[1]; L = []; [\underline{\text{E}\text{A}\text{R}}, \underline{\text{T}\text{I}\text{M}\text{E}}, \underline{\text{L}\text{I}\text{N}}, \underline{\text{I}\text{N}}, \underline{\text{I}\text{N}\text{G}}, \underline{\text{S}\text{O}\text{R}\text{T}}]$ is distributed into buckets according to the first component, but only those buckets corresponding to $\text{FULL}[1] = [\underline{\text{E}}, \underline{\text{I}}, \underline{\text{L}}, \underline{\text{S}}, \underline{\text{T}}]$ are concatenated back into L .

Nonempty buckets can be found, as a matter of fact, by bucket sorting. Let P be a list of pairs $\langle i, A[i] \rangle$ for all arrays A in list L , with $1 \leq i \leq \text{maxlen}$. After bucket sorting P on the second position and then on the first position, a sorted list (without duplicates) $\text{FULL}[i]$ of the integers in the i th position of the arrays can be obtained, indicating which buckets will be nonempty during the radix sort, when bucket sorting on the i th position of the arrays.

Example A.8. Finding nonempty buckets for a more efficient radix sorting of the list of arrays of integers of Example A.7, which are the ASCII codes of the characters in the list of character strings [SORT, ING, IN, LIN, EAR, TIME].



- (a) Extracting a list P of pairs $\langle i, A[i] \rangle$ for all arrays A in list L , with $1 \leq i \leq 4$. (b) Bucket sorting list P by the second position. (c) Bucket sorting the resulting list P by the first position. (d) Extracting a sorted list $FULL[i]$ of the integers in the i th position of the arrays. (e) Removing duplicates from each sorted list $FULL[i]$.

The following algorithm sorts the arrays of integers of list L using radix sort, implementing the previous procedure. In order to avoid using a new bucket array for each pass of bucket sort, a single array $bucket$ of lists of arrays of integers is used throughout the whole procedure, and the list P of pairs $\langle i, A[i] \rangle$ is also represented by a list of arrays of integers.

\langle subroutines 40 $\rangle + \equiv$

```
void radix_sort(
    list<array<int>>& L,
    int min,
    int max)
{
    if (L.empty()) return;
    <compute number n of buckets needed 441a>
```

```

array<list<array<int>>> bucket(1,n);
⟨find nonempty buckets 441b⟩
⟨distribute arrays by length 442c⟩
for (int i = maxlen; i ≥ 1; i–) {
    ⟨bucket sort list L of arrays on ith component 443a⟩
}
⟨double-check radix sort 443c⟩
}

```

The number of buckets needed is the largest of $\max - \min + 1$ and maxlen , the maximum length among all arrays A in list L .

441a ⟨compute number n of buckets needed 441a⟩≡
int maxlen = 0;
 $\text{array} < \text{int} > A, T;$
forall(A,L)
 maxlen = leda_max(maxlen,A.size());
int n = leda_max(max - min + 1,maxlen);

Nonempty buckets are found by (a) building a list P of pairs $\langle i, A[i] \rangle$ for all arrays A in list L , with $1 \leq i \leq \text{maxlen}$; (b) bucket sorting list P on the second component and then on the first component; (c) collecting the second component of each pair of the sorted list P in a sorted list $FULL[i]$ of integers; and (d) making each lists $FULL[i]$ unique by removing duplicates.

Notice that duplicates can be removed in time linear in the length of the list using the LEDA procedure *unique*, because lists $FULL[i]$ are sorted.

441b ⟨find nonempty buckets 441b⟩≡
 ⟨make list P of pairs $(i, A[i])$ 441c⟩
 ⟨bucket sort list P of pairs by $A[i]$ 442a⟩
 ⟨bucket sort list P of pairs by i 442b⟩
 $\text{array} < \text{list} < \text{int} > > FULL(1,maxlen);$
 forall(A,P)
 $FULL[A[1]].append(A[2]);$
 for (**int** i = 1; i ≤ maxlen; i++)
 $FULL[i].unique();$

Recall that the list of pairs $\langle i, A[i] \rangle$ is represented by a list P of arrays $T = [i, A[i]]$ of integers, in order to use the same array *bucket* of lists of arrays of integers for bucket sorting throughout the radix sorting algorithm.

441c ⟨make list P of pairs $(i, A[i])$ 441c⟩≡
 $list < \text{array} < \text{int} > > P;$

```

forall(A,L) {
  for (int i = 1; i <= A.size(); i++) {
    array<int> T(1,i,A[i]);
    P.append(T);
  }
}

```

When bucket sorting the list P of arrays T of integers by the second component, array $T = [i, A[i]]$ belongs in bucket $A[i] - \min + 1$.

442a ⟨bucket sort list P of pairs by $A[i]$ 442a⟩≡

```

while ( !P.empty() ) {
  T = P.pop();
  int k = T[2] - min + 1; // T belongs in bucket k
  if ( k >= 1 & k <= n ) {
    bucket[k].append(T);
  } else {
    error_handler(1,"radix sort: value out of range");
  }
  for (int i = 1; i <= n; i++)
    P.conc(bucket[i]); // destructive
}

```

When bucket sorting the list P of arrays T of integers by the first component, on the other hand, array $T = [i, A[i]]$ belongs in bucket i .

442b ⟨bucket sort list P of pairs by i 442b⟩≡

```

while ( !P.empty() ) {
  T = P.pop();
  int k = T[1]; // T belongs in bucket k
  if ( k >= 1 & k <= n ) {
    bucket[k].append(T);
  } else {
    error_handler(1,"radix sort: value out of range");
  }
  for (int i = 1; i <= n; i++)
    P.conc(bucket[i]); // destructive
}

```

Distributing the list L of arrays A of integers into lists $LEN[i]$ of arrays of length i is straightforward. An array A of length i belongs in $LEN[i]$.

442c ⟨distribute arrays by length 442c⟩≡

```

array<list<array<int>>> LEN(1,maxlen);
while ( !L.empty() ) {
  A = L.pop();
  LEN[A.size()].append(A);
}

```

Now, bucket sorting the list L of arrays at component i can be done by first concatenating all arrays of length i at the front of L , then distributing L into buckets according to the integers at the i th component and, finally, concatenating all nonempty buckets back into L .

443a ⟨bucket sort list L of arrays on i th component 443a⟩≡
L.conc(LEN[i],LEDA::before); // destructive
while (¬L.empty()) {
A = L.pop();
int k = A[i] - min + 1; // array A belongs in bucket k
if (k ≥ 1 ∧ k ≤ n) {
bucket[k].append(A);
} else {
error_handler(1,"radix sort: value out of range");
} }
int x;
forall(x,FULL[i])
L.conc(bucket[x - min + 1]); // destructive

Radix sort is often called just on a list of arrays of elements. A procedure call of the form *radix_sort(L)* is the same as the procedure call *radix_sort(L,i,j)*, where i and j are respectively the minimum and maximum elements among all arrays in list L .

443b ⟨subroutines 40⟩+≡
void radix_sort(
list<array<int> >& L)
{
if (L.length() ≤ 1) return;
if (L.head().size() ≡ 0) return;
int i = L.head()[1];
int j = i;
array<int> A;
int x;
forall(A,L) {
forall(x,A) {
i = leda_min(i,x);
j = leda_max(j,x);
} }
radix_sort(L,i,j);
}

The following double-check of radix sort, although being redundant, gives some reassurance of the correctness of the implementation. It verifies that L is sorted in lexicographic order, that is, that $A \leqslant \text{succ}(A)$ for all but the last array of integers A of the list of arrays of integers L .

443c \langle double-check radix sort 443c $\rangle \equiv$
 $\{$ list.item it;
 \forall all_items(it,L) {
 \quad if (it \neq L.last() \wedge L[it] $>$ L[L.succ(it)]) {
 $\quad\quad$ error_handler(1,"Wrong implementation of radix sort");
 $\quad}$ } }

Remark A.9. Correctness of radix sorting follows from bucket sort being a stable sorting method.

Lemma A.10. *The algorithm for radix sorting runs in $O(n + j - i)$ time using $O(\max(n, j - i))$ additional space, where n is the total length of the arrays in list L and i and j are respectively the minimum and maximum elements among all arrays in list L .*

Proof. Let L be the list of m arrays of integers to be sorted, let n be the total length of the arrays in list L , and let i and j be respectively the minimum and maximum elements among all arrays in list L . Let also P be a list of pairs $\langle k, A[k] \rangle$ for all arrays A in list L .

List P is built in $O(n)$ time, and nonempty buckets are found in $O(n + j - i)$ time by two passes of bucket sort upon list P . Then, for each component k in the arrays, bucket sorting list L on the k th component takes time linear in the number of arrays A with $\text{length}[A] \geq k$ and therefore, bucket sorting list L on all array components takes a total of $O(n + j - i)$ time. Further, $O(\max(n, j - i))$ additional space is used for storing the array of buckets, which is shared by all passes of bucket sort.

The double-check of radix sorting makes m array comparisons, where each comparison takes time linear in the length of the arrays, and therefore runs in $O(n)$ time, using $O(1)$ additional space. \square

A few implementation details still need to be filled in, though. The double-check of radix sort requires the lexicographic comparison of arrays of integers, and the following procedure defines, then, a default linear order for LEDA arrays of integers. Given two arrays of integers A_1 and A_2 , it returns -1 if A_1 is smaller in lexicographic order than A_2 , 0 if the arrays A_1 and A_2 are identical, and 1 if A_1 is larger in lexicographic order than A_2 .

444 ⟨default procedures 444⟩ ≡

```
int compare(
    const array<int>& A1,
    const array<int>& A2)
{
    int n = leda_min(A1.size(),A2.size());
    for ( int i = 1; i ≤ n; i++ ) {
        if ( A1[i] < A2[i] ) return -1;
        if ( A1[i] > A2[i] ) return 1;
    }
    if ( A1.size() < A2.size() ) return -1;
    if ( A1.size() > A2.size() ) return 1;
    return 0;
}
```

Now, the following Boolean comparison operator for arrays of integers makes it possible to test an array for being larger than another one.

445 ⟨default procedures 444⟩ + ≡

```
bool operator>(
    const array<int>& A1,
    const array<int>& A2)
{
    return compare(A1,A2) ≡ 1;
}
```

Bibliographic Notes

See [234, 236] for a comprehensive description of LEDA. The simple implementation of radix sort is based on [233, Sect. 2.2]. See also [8, 9, 36, 87, 195, 230, 373, 374].

B. Interactive Demonstration of Graph Algorithms

The interactive demonstration of graph algorithms is put together in this appendix. The different program modules are first integrated in a C++ main program, with a LEDA graph window providing access to the program modules.

447

```
(the main program 447)≡
void main() {
    GRAPH<string,string> G;
    GraphWin gw(G,500,500,"Algorithms on Trees and Graphs");

    int techniques_menu = gw.add_menu("Techniques");
    gw.add_simple_call(gw_tree_edit,
        "Tree Edit",techniques_menu);
    gw.add_simple_call(gw_backtracking_tree_edit,
        "Backtracking Tree Edit",techniques_menu);
    gw.add_simple_call(gw_branch_and_bound_tree_edit,
        "Branch-and-Bound Tree Edit",techniques_menu);
    gw.add_simple_call(gw_divide_and_conquer_tree_edit,
        "Divide-and-Conquer Tree Edit",techniques_menu);
    gw.add_simple_call(gw_top_down_dynamic_programming_tree_edit,
        "Top-Down Dynamic Programming Tree Edit",techniques_menu);
    gw.add_simple_call(gw_bottom_up_dynamic_programming_tree_edit,
        "Bottom-Up Dynamic Programming Tree Edit",techniques_menu);

    int traverse_menu = gw.add_menu("Traversal");
    gw.add_simple_call(gw_rec_preorder_tree_traversal,
        "Preorder Tree Traversal (Recursive)",traverse_menu);
    gw.add_simple_call(gw_preorder_tree_traversal,
        "Preorder Tree Traversal (Iterative)",traverse_menu);
    gw.add_simple_call(gw_rec_postorder_tree_traversal,
        "Postorder Tree Traversal (Recursive)",traverse_menu);
    gw.add_simple_call(gw_postorder_tree_traversal,
        "Postorder Tree Traversal (Iterative)",traverse_menu);
    gw.add_simple_call(gw_top_down_tree_traversal,
        "Top-Down Tree Traversal",traverse_menu);
    gw.add_simple_call(gw_bottom_up_tree_traversal,
        "Bottom-Up Tree Traversal",traverse_menu);

    gw.add_separator(traverse_menu);
```

```

gw.add_simple_call(gw.preorder_tree_depth,
  "Tree Depth",traverse_menu);
gw.add_simple_call(gw.bottom_up_tree_height,
  "Tree Height",traverse_menu);
gw.add_simple_call(gw.layered_tree_layout,
  "Layered Tree Layout",traverse_menu);

gw.add_separator(traverse_menu);

gw.add_simple_call(gw.depth_first_traversal,
  "Depth-First Traversal",traverse_menu);
gw.add_simple_call(gw.depth_first_spanning_subtree,
  "Depth-First Spanning Subtree",traverse_menu);
gw.add_simple_call(gw.leftmost_depth_first_traversal,
  "Leftmost Depth-First Traversal",traverse_menu);
gw.add_simple_call(gw.breadth_first_traversal,
  "Breadth-First Traversal",traverse_menu);
gw.add_simple_call(gw.breadth_first_spanning_subtree,
  "Breadth-First Spanning Subtree",traverse_menu);

gw.add_separator(traverse_menu);

gw.add_simple_call(gw.ordered_graph_isomorphism,
  "Ordered Graph Isomorphism",traverse_menu);

int clique_menu = gw.add_menu("Cliques");
gw.add_simple_call(gw.all_cliques,
  "All Cliques",clique_menu);
gw.add_simple_call(gw.all_maximal_cliques,
  "All Maximal Cliques",clique_menu);
gw.add_simple_call(gw.maximum_clique,
  "Maximum Clique",clique_menu);

gw.add_separator(clique_menu);

gw.add_simple_call(gw.graph_complement,
  "Graph Complement",clique_menu);

gw.add_separator(clique_menu);

gw.add_simple_call(gw.tree_maximum_independent_set,
  "Maximum Independent Set in a Tree",clique_menu);
gw.add_simple_call(gw.maximum_independent_set,
  "Maximum Independent Set",clique_menu);

gw.add_separator(clique_menu);

gw.add_simple_call(gw.tree_minimum_vertex_cover,
  "Minimum Vertex Cover in a Tree",clique_menu);
gw.add_simple_call(gw.minimum_vertex_cover,
  "Minimum Vertex Cover",clique_menu);

int isomorph_menu = gw.add_menu("Isomorphism");

```

```

int tree_isomorph_menu =
    gw.add_menu("Tree Isomorphism",isomorph_menu);
    gw.add_simple_call(gw_ordered_tree_isomorphism,
        "Ordered",tree_isomorph_menu);
    gw.add_simple_call(gw_unordered_tree_isomorphism,
        "Unordered",tree_isomorph_menu);

int subtree_isomorph_menu =
    gw.add_menu("Subtree Isomorphism",isomorph_menu);
    gw.add_simple_call(gw_top_down_ordered_subtree_isomorphism,
        "Top-Down Ordered",subtree_isomorph_menu);
    gw.add_simple_call(gw_top_down_unordered_subtree_isomorphism,
        "Top-Down Unordered",subtree_isomorph_menu);
    gw.add_simple_call(gw_bottom_up_ordered_subtree_isomorphism,
        "Bottom-Up Ordered",subtree_isomorph_menu);
    gw.add_simple_call(gw_bottom_up_unordered_subtree_isomorphism,
        "Bottom-Up Unordered",subtree_isomorph_menu);

int max_common_subtree_isomorph_menu =
    gw.add_menu("Maximum Common Subtree Isomorphism",isomorph_menu);
    gw.add_simple_call(
        gw_top_down_ordered_maximum_common_subtree_isomorphism,
        "Top-Down Ordered",max_common_subtree_isomorph_menu);
    gw.add_simple_call(
        gw_top_down_unordered_maximum_common_subtree_isomorphism,
        "Top-Down Unordered",max_common_subtree_isomorph_menu);
    gw.add_simple_call(
        gw_bottom_up_ordered_maximum_common_subtree_isomorphism,
        "Bottom-Up Ordered",max_common_subtree_isomorph_menu);
    gw.add_simple_call(
        gw_bottom_up_unordered_maximum_common_subtree_isomorphism,
        "Bottom-Up Unordered",max_common_subtree_isomorph_menu);

    gw.add_separator(isomorph_menu);

    gw.add_simple_call(gw_graph_isomorphism,
        "Graph Isomorphism",isomorph_menu);
    gw.add_simple_call(gw_graph_automorphism,
        "Graph Automorphism",isomorph_menu);
    gw.add_simple_call(gw_subgraph_isomorphism,
        "Subgraph Isomorphism",isomorph_menu);
    gw.add_simple_call(gw_maximal_common_subgraph_isomorphism,
        "Maximal Common Subgraph Isomorphism",isomorph_menu);

    gw.display(window::min,window::min);

int h_menu = gw.get_menu("Help");
gw.add_simple_call(gw_about_book,"About the book",h_menu);

gw.edit();
}

```

The structure of the main program is, in fact, determined by the * program module, which integrates header files for the LEDA modules needed, the tree graph representation described in Sect. A.4, template files, default procedures, the program modules for the algorithms on trees and graphs, and the actual main C++ program.

450a `(* 450a)≡
 ⟨header files to include 450b⟩
 ⟨tree graph representation 429a⟩
 ⟨templates 452a⟩
 ⟨default procedures 444⟩
 ⟨subroutines 40⟩
 ⟨the main program 447⟩`

The LEDA modules needed are graphs, graph algorithms, and graph windows, together with two-dimensional static arrays, dictionary arrays, hashing arrays, vertex priority queues, and sets.

450b `⟨header files to include 450b⟩≡
 #include <LEDA/graph.h>
 #include <LEDA/graph_alg.h>
 #include <LEDA/graphwin.h>

 #include <LEDA/array2.h>
 #include <LEDA/d_array.h>
 #include <LEDA/h_array.h>
 #include <LEDA/node_pq.h>
 #include <LEDA/set.h>`

The following simple window panels are used throughout the interactive demonstration of graph algorithms, in order to get yes-no and ok-proof answers to short questions.

450c `⟨subroutines 40⟩+≡
 static void make_proof_panel(
 panel& P,
 string s,
 bool proof = false)
 {
 P.text_item(" \\\bf \\\blue " + s + ".");
 P.button("ok");
 if (proof) P.fbutton("proof",1);
 }

 static void make_yes_no_panel(
 panel& P,
 string s,
 bool no = false)
 {`

```

P.text_item("\bf \blue " + s + "?");
P.button("yes");
if (no) P.button("no",1);
}

static void make_proof_maximum_panel(
    panel& P,
    string s,
    bool proof = false)
{
    P.text_item("\bf \blue " + s + ".");
    P.button("ok",0);
    if (proof) P.button("proof",1);
    P.button("maximum",2);
}

```

The program modules for tree edit, tree traversal, tree isomorphism, graph traversal, cliques, and graph isomorphism algorithms are integrated next into a *subroutines* module.

451

```

⟨subroutines 40⟩+≡
    ⟨double-check tree edit 78b⟩
    ⟨techniques 66⟩
    ⟨demo techniques 68⟩

    ⟨double-check preorder tree traversal 115c⟩
    ⟨preorder tree traversal 115a⟩
    ⟨double-check postorder tree traversal 121c⟩
    ⟨postorder tree traversal 121a⟩
    ⟨double-check top-down tree traversal 127b⟩
    ⟨top-down tree traversal 127a⟩
    ⟨double-check bottom-up tree traversal 133b⟩
    ⟨bottom-up tree traversal 132⟩
    ⟨tree traversal 142a⟩

    ⟨demo preorder tree traversal 118a⟩
    ⟨demo postorder tree traversal 124a⟩
    ⟨demo top-down tree traversal 129⟩
    ⟨demo bottom-up tree traversal 135⟩
    ⟨demo tree traversal 144b⟩

    ⟨tree isomorphism 154⟩
    ⟨subtree isomorphism 173⟩
    ⟨maximum common subtree isomorphism 209a⟩
    ⟨demo tree isomorphism 168b⟩
    ⟨demo subtree isomorphism 202a⟩
    ⟨demo maximum common subtree isomorphism 236a⟩

    ⟨graph traversal 262⟩
    ⟨demo graph traversal 267a⟩

    ⟨double-check clique 304⟩

```

```

⟨double-check maximal clique 314a⟩
⟨maximal clique 303a⟩
⟨maximum clique 320a⟩
⟨double-check independent set 327⟩
⟨independent set 326⟩
⟨demo maximal clique 314b⟩
⟨demo maximum clique 322⟩
⟨demo independent set 331⟩
⟨double-check vertex cover 337b⟩
⟨vertex cover 337a⟩
⟨demo vertex cover 340a⟩

⟨double-check graph isomorphism 357⟩
⟨double-check subgraph isomorphism 374⟩
⟨double-check maximal common subgraph isomorphism 385⟩
⟨graph isomorphism 355⟩
⟨subgraph isomorphism 370a⟩
⟨maximal common subgraph isomorphism 383⟩
⟨demo graph isomorphism 362⟩
⟨demo graph automorphism 365⟩
⟨demo subgraph isomorphism 375⟩
⟨demo maximal common subgraph isomorphism 386⟩

```

The different algorithms on trees and graphs are declared in a separate header file.

452a ⟨templates 452a⟩≡
#include "combin.h"

Finally, the help menu of the LEDA graph window is extended with the following, standard copyright notice.

452b ⟨subroutines 40⟩+≡
void about_book(
 GraphWin& gw)
{
 window& W = gw.get_window();
 panel P;
 P.text_item(" \bf Algorithms on Trees and Graphs");
 P.text_item(""); P.text_item("");
 P.text_item(" by Gabriel Valiente");
 P.text_item(""); P.text_item("");
 P.text_item(" (c) 2002 Springer-Verlag");
 P.text_item(""); P.text_item("");
 P.text_item(" LEDA (c) 1991-2002 Algorithmic Solutions GmbH");
 P.text_item("");
 P.button("ok");
 P.open(W);
}

C. Program Modules

- ⟨* 450a⟩ [450a](#)
- ⟨alternative iteration over all children w of node v in T 45b⟩ [45b](#)
- ⟨assign isomorphism codes to all nodes of T_1 and T_2 in postorder 163b⟩ [162](#), [163b](#)
- ⟨bottom-up tree traversal 132⟩ [132](#), [137](#), [451](#)
- ⟨bucket sort list L of arrays on i th component 443a⟩ [440](#), [443a](#)
- ⟨bucket sort list P of pairs by $A[i]$ 442a⟩ [441b](#), [442a](#)
- ⟨bucket sort list P of pairs by i 442b⟩ [441b](#), [442b](#)
- ⟨build ordered graph isomorphism mapping 291a⟩ [290](#), [291a](#)
- ⟨build subtree isomorphism mapping 198a⟩ [195](#), [198a](#)
- ⟨build tree isomorphism mapping 165a⟩ [162](#), [165a](#)
- ⟨center parent nodes over their children nodes 144a⟩ [142a](#), [144a](#)
- ⟨compute breadth of all nodes 142c⟩ [142a](#), [142c](#)
- ⟨compute depth and preorder rank of all nodes 128⟩ [127b](#), [128](#), [133b](#)
- ⟨compute depth of all nodes 142b⟩ [142a](#), [142b](#)
- ⟨compute height and size of all nodes in the first tree 186⟩ [184a](#), [186](#)
- ⟨compute height and size of all nodes in the second tree 187⟩ [184a](#), [187](#), [190a](#)
- ⟨compute height of all nodes 134⟩ [133b](#), [134](#)
- ⟨compute height of root of first tree 190b⟩ [190a](#), [190b](#)
- ⟨compute number n of buckets needed 441a⟩ [440](#), [441a](#)
- ⟨default procedures 444⟩ [444](#), [445](#), [450a](#)
- ⟨demo bottom-up tree traversal 135⟩ [135](#), [138](#), [451](#)
- ⟨demo graph automorphism 365⟩ [365](#), [451](#)
- ⟨demo graph isomorphism 362⟩ [362](#), [451](#)
- ⟨demo graph traversal 267a⟩ [267a](#), [270a](#), [274](#), [285b](#), [287a](#), [293](#), [451](#)
- ⟨demo independent set 331⟩ [331](#), [332a](#), [332b](#), [451](#)
- ⟨demo maximal clique 314b⟩ [314b](#), [315](#), [316b](#), [451](#)

⟨demo maximal common subgraph isomorphism 386⟩ [386](#), 451
⟨demo maximum clique 322⟩ [322](#), 451
⟨demo maximum common subtree isomorphism 236a⟩ [236a](#), [237](#),
[238a](#), [238b](#), 451
⟨demo postorder tree traversal 124a⟩ [124a](#), [124b](#), 451
⟨demo preorder tree traversal 118a⟩ [118a](#), [118b](#), [136b](#), 451
⟨demo subgraph isomorphism 375⟩ [375](#), 451
⟨demo subtree isomorphism 202a⟩ [202a](#), [203](#), [204](#), [205](#), 451
⟨demo techniques 68⟩ [68](#), [79b](#), [84b](#), [93](#), [105](#), [106](#), 451
⟨demo top-down tree traversal 129⟩ [129](#), 451
⟨demo tree isomorphism 168b⟩ [168b](#), [169](#), 451
⟨demo tree traversal 144b⟩ [144b](#), 451
⟨demo vertex cover 340a⟩ [340a](#), [340b](#), 451
⟨distribute arrays by length 442c⟩ [440](#), [442c](#)
⟨double-check bottom-up ordered maximum common subtree iso-
morphism 228⟩ [226](#), [228](#), [229a](#)
⟨double-check bottom-up ordered subtree isomorphism 190c⟩ [190a](#),
[190c](#)
⟨double-check bottom-up tree traversal 133b⟩ [133b](#), 451
⟨double-check bottom-up unordered maximum common subtree iso-
morphism 234a⟩ [232](#), [234a](#), [234b](#)
⟨double-check bottom-up unordered subtree isomorphism 199⟩ [199](#)
⟨double-check breadth-first forest 282⟩ [281](#), [282](#)
⟨double-check breadth-first tree 285a⟩ [284](#), [285a](#)
⟨double-check bucket sort 436b⟩ [435](#), [436b](#)
⟨double-check clique 304⟩ [304](#), 451
⟨double-check depth-first forest 263⟩ [262](#), [263](#)
⟨double-check depth-first tree 266⟩ [265](#), [266](#)
⟨double-check graph isomorphism 357⟩ [357](#), 451
⟨double-check independent set 327⟩ [327](#), [330b](#), 451
⟨double-check maximal clique 314a⟩ [314a](#), 451
⟨double-check maximal common subgraph isomorphism 385⟩ [385](#),
451
⟨double-check ordered tree isomorphism 156⟩ [155a](#), [156](#)
⟨double-check postorder tree traversal 121c⟩ [121c](#), 451
⟨double-check preorder tree traversal 115c⟩ [115c](#), [116](#), 451
⟨double-check radix sort 443c⟩ [440](#), [443c](#)

- ⟨double-check subgraph isomorphism 374⟩ [374](#), 451
- ⟨double-check that L is a permutation of C 29b⟩ [28](#), [29b](#)
- ⟨double-check that L is sorted 29a⟩ [28](#), [29a](#)
- ⟨double-check that T is a permutation of L 31b⟩ [30](#), [31b](#)
- ⟨double-check that T is sorted 32⟩ [30](#), [32](#)
- ⟨double-check top-down ordered maximum common subtree isomorphism 210a⟩ [209a](#), [210a](#), [210b](#)
- ⟨double-check top-down ordered subtree isomorphism 174b⟩ [173](#), [174b](#)
- ⟨double-check top-down tree traversal 127b⟩ [127b](#), 451
- ⟨double-check top-down unordered maximum common subtree isomorphism 222b⟩ [221](#), [222b](#)
- ⟨double-check top-down unordered subtree isomorphism 185⟩ [184a](#), [185](#)
- ⟨double-check tree edit 78b⟩ [78b](#), 451
- ⟨double-check unordered tree isomorphism 166⟩ [162](#), [166](#)
- ⟨double-check vertex cover 337b⟩ [337b](#), [339b](#), 451
- ⟨enqueue all leaves in top-down traversal order 133a⟩ [132](#), [133a](#)
- ⟨ensure that tree edit mapping is a bijection 77b⟩ [77a](#), [77b](#)
- ⟨ensure that tree edit mapping preserves parent-child relation 77c⟩ [77a](#), [77c](#)
- ⟨ensure that tree edit mapping preserves sibling order 78a⟩ [77a](#), [78a](#)
- ⟨find largest common subtree 227⟩ [226](#), [227](#), 232
- ⟨find nonempty buckets 441b⟩ [440](#), [441b](#)
- ⟨find roots and arcs in breadth-first forest 283⟩ [282](#), [283](#)
- ⟨find roots and arcs in depth-first forest 264⟩ [263](#), [264](#)
- ⟨graph isomorphism 355⟩ [355](#), [356b](#), [358](#), 451
- ⟨graph traversal 262⟩ [262](#), [265](#), [272](#), [273](#), [281](#), [284](#), [290](#), 451
- ⟨header files to include 450b⟩ [450a](#), [450b](#)
- ⟨highlight discovered vertex 268a⟩ [267b](#), [268a](#), 286a
- ⟨highlight incoming breadth-first tree arc 286b⟩ [286a](#), [286b](#)
- ⟨highlight incoming depth-first tree arc 268b⟩ [267b](#), [268b](#)
- ⟨highlight predecessor vertices 268c⟩ [267b](#), [268c](#), 286a
- ⟨implementation correctness of sorting 28⟩ [28](#), [30](#), [31a](#), 33
- ⟨independent set 326⟩ [326](#), [328](#), [329](#), [330a](#), 451
- ⟨iteration over all children w of node v in T 45a⟩ [45a](#)
- ⟨iteration over all vertices v of graph G 35a⟩ [35a](#)

⟨iteration over all vertices w adjacent in G to vertex v 35b⟩ 35b
 ⟨make list P of pairs $(i, A[i])$ 441c⟩ 441b, 441c
 ⟨maximal clique 303a⟩ 303a, 303b, 307, 308, 312, 313, 451
 ⟨maximal common subgraph isomorphism 383⟩ 383, 384, 451
 ⟨maximum clique 320a⟩ 320a, 320b, 451
 ⟨maximum common subtree isomorphism 209a⟩ 209a, 209b, 220, 221,
 226, 229b, 232, 233, 451
 ⟨obtain order in which vertices of G_1 were visited 292a⟩ 290, 292a
 ⟨obtain order in which vertices of G_2 were visited 292b⟩ 290, 292b
 ⟨partition first tree in isomorphism equivalence classes 196a⟩ 195,
 196a, 226, 232
 ⟨partition second tree in isomorphism equivalence classes 196b⟩ 195,
 196b, 226, 232
 ⟨postorder tree traversal 121a⟩ 121a, 121b, 122, 451
 ⟨preorder tree traversal 115a⟩ 115a, 115b, 117, 136a, 451
 ⟨put ordered graph in standard representation 269⟩ 267a, 269, 270a,
 285b, 287a
 ⟨reconstruct max common unordered subtree isomorphism 222a⟩ 221,
 222a
 ⟨reconstruct unordered subtree isomorphism 184b⟩ 184a, 184b
 ⟨refine candidate nodes 77a⟩ 76a, 77a, 84a
 ⟨set initial node coordinates 143⟩ 142a, 143
 ⟨set layout of edit graph 69⟩ 68, 69
 ⟨set up adjacency matrices 356a⟩ 355, 356a, 370a, 375, 383, 384, 386
 ⟨set up adjacency matrix 316a⟩ 314b, 316a, 316b, 322, 329, 330b, 384
 ⟨set up candidate nodes 76b⟩ 75, 76b, 83
 ⟨set up candidate vertices 370b⟩ 370a, 370b
 ⟨set up children arrays 104b⟩ 102a, 104b
 ⟨show breadth-first graph traversal 286a⟩ 285b, 286a
 ⟨show breadth-first spanning subtree 287b⟩ 287a, 287b
 ⟨show common subgraph isomorphism 388⟩ 386, 388
 ⟨show depth-first graph traversal 267b⟩ 267a, 267b
 ⟨show depth-first spanning subtree 270b⟩ 270a, 270b
 ⟨show graph isomorphism 363⟩ 293, 362, 363
 ⟨show leftmost depth-first graph traversal 275⟩ 274, 275
 ⟨show maximum common subtree isomorphism 236b⟩ 236a, 236b,
 237, 238a, 238b

⟨show subgraph isomorphism 376⟩ 375, 376
⟨show subtree isomorphism 202b⟩ 202a, 202b, 203
⟨show tree isomorphism 170⟩ 168b, 169, 170
⟨show tree traversal 119⟩ 118a, 118b, 119, 124a, 124b, 129, 135
⟨subgraph isomorphism 370a⟩ 370a, 371, 373, 451
⟨subroutines 40⟩ 40, 41a, 41b, 435, 436a, 440, 443b, 450a, 450c, 451,
452b
⟨subtree isomorphism 173⟩ 173, 183, 184a, 190a, 195, 197, 198b,
201a, 201b, 451
⟨techniques 66⟩ 66, 67a, 67b, 75, 76a, 79a, 83, 84a, 89, 91, 92, 97, 98,
102a, 102b, 104a, 451
⟨templates 452a⟩ 450a, 452a
⟨test implementation correctness of sorting 33⟩ 33
⟨test mapping for ordered graph isomorphism 291b⟩ 290, 291b
⟨the main program 447⟩ 447, 450a
⟨top-down tree traversal 127a⟩ 127a, 451
⟨tree graph representation 429a⟩ 429a, 429b, 430a, 430b, 430c, 430d,
431a, 431b, 431c, 431d, 432a, 432b, 432c, 432d, 432e, 450a
⟨tree isomorphism 154⟩ 154, 155a, 155b, 162, 163a, 165b, 167a, 167b,
168a, 174a, 451
⟨tree traversal 142a⟩ 142a, 451
⟨vertex cover 337a⟩ 337a, 339a, 451

Bibliography

Abbreviations of journal titles follow the form used in the Mathematical Reviews journal of the American Mathematical Society.

1. G. M. Adel'son-Vel'skii and Y. M. Landis. An algorithm for the organization of information. *Dokl. Akad. Nauk*, 3(146):1259–1262, 1962.
2. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading MA, 1974.
3. R. K. Ahuja, K. Mehlhorn, J. B. Orlin, and R. E. Tarjan. Faster algorithms for the shortest path problem. *J. ACM*, 37(2):213–223, 1990.
4. F. A. Akinniyi, A. K. C. Wong, and D. A. Stacey. A new algorithm for graph monomorphism based on the projections of the product graph. *IEEE Trans. Syst. Man Cyb.*, 16(5):740–751, 1986.
5. E. A. Akkoyunlu. The enumeration of maximal cliques of large graphs. *SIAM J. Comput.*, 2(1):1–6, 1973.
6. L. Alonso and R. Schott. On the tree inclusion problem. *Acta Inform.*, 37(9):653–670, 2001.
7. H. Alt, N. Blum, K. Mehlhorn, and M. Paul. Computing a maximum cardinality matching in a bipartite graph in time $O(n^{1.5}\sqrt{m}/\log n)$. *Inform. Process. Lett.*, 37(4):237–240, 1991.
8. A. Andersson and S. Nilsson. A new efficient radix sort. In *Proc. 35th Annual Symp. Foundations of Computer Science*, pages 714–721, Piscataway NJ, 1994. IEEE.
9. A. Andersson and S. Nilsson. Implementing radixsort. *ACM J. Exp. Algorithmics*, 3(7), 1998.

10. S. R. Arikati, A. Maheshwari, and C. D. Zaroliagis. Efficient computation of implicit representations of sparse graphs. *Discr. Appl. Math.*, 78(1–3):1–16, 1997.
11. S. R. Arikati and K. Mehlhorn. A correctness certificate for the Stoer–Wagner min-cut algorithm. *Inform. Process. Lett.*, 70(5):251–254, 1999.
12. J. E. Ash, W. A. Warr, and P. Willett, editors. *Chemical Structure Systems*. Ellis Horwood, Chichester, 1991.
13. T. K. Attwood and D. G. Parry-Smith. *Introduction to Bioinformatics*. Prentice Hall, London, 1999.
14. J. G. Augustson and J. Minker. An analysis of some graph theoretical cluster techniques. *J. ACM*, 17(4):571–588, 1970.
15. G. Ausiello, P. Crescenzi, G. Gambosi, V. Kahn, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation: Combinatorial Optimization Problems and their Approximability Properties*. Springer-Verlag, Berlin Heidelberg, 1999.
16. A. Avizienis. The n -version approach to fault tolerant software. *IEEE Trans. Soft. Eng.*, 11(12):1491–1501, 1985.
17. E. Balas and J. Xue. Minimum weighted coloring of triangulated graphs, with application to maximum weight vertex packing and clique finding in arbitrary graphs. *SIAM J. Comput.*, 20(2):209–221, 1991.
18. E. Balas and J. Xue. Addendum: Minimum weighted coloring of triangulated graphs, with application to maximum weight vertex packing and clique finding in arbitrary graphs. *SIAM J. Comput.*, 21(5):1000, 1992.
19. E. Balas and J. Xue. Weighted and unweighted maximum clique algorithms with upper bounds from fractional coloring. *Algorithmica*, 15(5):397–412, 1996.
20. E. Balas and C. S. Yu. Finding a maximum clique in an arbitrary graph. *SIAM J. Comput.*, 15(4):1054–1068, 1986.
21. R. Balasubramanian, M. R. Fellows, and V. Raman. An improved fixed-parameter algorithm for vertex cover. *Inform. Process. Lett.*, 65(3):163–168, 1998.
22. J. L. Balcázar and A. Lozano. The complexity of graph problems for succinctly represented graphs. In *Proc. 15th Int. Workshop Graph-Theoretic Concepts in Computer Science*, volume 411 of

- Lecture Notes in Comput. Sci.*, pages 277–285, Berlin Heidelberg, 1989. Springer-Verlag.
- 23. J. Bang-Jensen and G. Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer-Verlag, Berlin Heidelberg, 2001.
 - 24. J. M. Barnard. Substructure searching methods: Old and new. *J. Chem. Inf. Comp. Sci.*, 33(4):532–538, 1993.
 - 25. H. G. Barrow and R. M. Burstall. Subgraph isomorphism, matching relational structures and maximal cliques. *Inform. Process. Lett.*, 4(4):83–84, 1976.
 - 26. R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Inform.*, 1(3):173–189, 1972.
 - 27. R. Beigel. Finding maximum independent sets in sparse and general graphs. In *Proc. 10th Annual ACM-SIAM Symp. Discrete Algorithms*, pages 856–857, New York, 1999. ACM.
 - 28. R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, 1957.
 - 29. R. Bellman. On a routing problem. *Quart. Appl. Math.*, 16(1): 87–90, 1958.
 - 30. R. Bellman and S. E. Dreyfus. *Applied Dynamic Programming*. Princeton University Press, Princeton, 1962.
 - 31. J. L. Bentley. *Programming Pearls*. ACM Press, New York, 2nd edition, 2000.
 - 32. J. L. Bentley and D. Gries. Programming pearls: Abstract data types. *Commun. ACM*, 30(4):284–290, 1987.
 - 33. J. L. Bentley, D. Haken, and J. B. Saxe. A general method for solving divide-and-conquer recurrences. *ACM SIGACT News*, 12(3):36–44, 1980.
 - 34. J. L. Bentley and D. E. Knuth. Programming pearls: Literate programming. *Commun. ACM*, 29(5):364–369, 1986.
 - 35. J. L. Bentley, D. E. Knuth, and M. D. McIlroy. Programming pearls: A literate program. *Commun. ACM*, 29(6):471–483, 1986.
 - 36. J. L. Bentley and M. D. McIlroy. Engineering a sort function. *Soft. Pract. Exper.*, 23(11):1249–1265, 1993.
 - 37. C. Berge. *Graphs and Hypergraphs*. North-Holland, Amsterdam, 2nd edition, 1985.

38. H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, and P. E. Bourne. The protein data bank. *Nucl. Acids Res.*, 28(1):235–242, 2000.
39. A. T. Berztiss. A backtrack procedure for isomorphism of directed graphs. *J. ACM*, 20(3):365–377, 1973.
40. A. T. Berztiss. *Data Structures: Theory and Practice*. Academic Press, New York, 2nd edition, 1975.
41. T. Beyer and S. M. Hedetniemi. Constant time generation of rooted trees. *SIAM J. Comput.*, 9(4):706–712, 1980.
42. J. M. Bishop and K. M. Gregson. Literate programming and the LIPED environment. *Struct. Prog.*, 13(1):23–34, 1992.
43. J. R. Bitner and E. M. Reingold. Backtrack programming techniques. *Commun. ACM*, 18(11):651–656, 1975.
44. M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. In *Proc. 32th Annual Symp. Foundations of Computer Science*, pages 90–99, Piscataway NJ, 1991. IEEE.
45. M. Blum and S. Kannan. Designing programs that check their work. *J. ACM*, 42(1):269–291, 1995.
46. M. Blum, M. Luby, and R. Rubinfeld. Self testing/correcting with applications to numerical problems. *J. Comp. Syst. Sci.*, 47(3):549–595, 1993.
47. M. Blum and H. Wasserman. Reflections on the Pentium division bug. *IEEE Trans. Comput.*, 45(4):385–393, 1996.
48. N. Blum and K. Mehlhorn. On the average number of rebalancing operations in weight-balanced trees. *Theor. Comput. Sci.*, 11(3):303–320, 1980.
49. B. Bollobás. *Graph Theory: An Introductory Course*, volume 63 of *Graduate Texts in Mathematics*. Springer-Verlag, Berlin Heidelberg, 1979.
50. B. Bollobás. *Modern Graph Theory*, volume 184 of *Graduate Texts in Mathematics*. Springer-Verlag, Berlin Heidelberg, 1998.
51. I. Bomze, M. Budinich, P. M. Pardalos, and M. Pelillo. The maximum clique problem. In D.-Z. Du and P. M. Pardalos, editors, *Handbook of Combinatorial Optimization*, volume Supp. A, pages 1–74. Kluwer, Dordrecht, 1999.

52. J. D. Bright, G. F. Sullivan, and G. M. Masson. A formally verified sorting certifier. *IEEE Trans. Comput.*, 46(12):1304–1312, 1997.
53. C. Bron and J. Kerbosch. ACM algorithm 457: Finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577, 1973.
54. A. D. Brown and P. R. Thomas. Goal-oriented subgraph isomorphism technique for IC device recognition. *IEE Proceedings*, 135(6):141–150, 1988.
55. M. Brown and B. Childs. An interactive environment for literate programming. *Struct. Prog.*, 11(1):11–26, 1990.
56. M. Brown and D. Cordes. Literate programming applied to conventional software design. *Struct. Prog.*, 11(2):85–98, 1990.
57. A. Brüggemann-Klein and D. Wood. Drawing trees nicely with \TeX . *Electronic Publishing*, 2(2):101–115, 1989.
58. H. Bunke. Structural and syntactic pattern recognition. In C. H. Chen, L. F. Pau, and P. S. P. Wang, editors, *Handbook of Pattern Recognition and Computer Vision*, chapter 1.5, pages 163–209. World Scientific, Singapore, 1993.
59. H. Bunke. On a relation between graph edit distance and maximum common subgraph. *Pattern Recogn. Lett.*, 18(8):689–694, 1997.
60. H. Bunke, T. Glauser, and T.-H. Tran. An efficient implementation of graph grammars based on the RETE matching algorithm. In *Graph-Grammars and their Application to Computer Science*, volume 532 of *Lecture Notes in Comput. Sci.*, pages 174–189, Berlin Heidelberg, 1991. Springer-Verlag.
61. H. Bunke and K. Shearer. A graph distance metric based on the maximal common subgraph. *Pattern Recogn. Lett.*, 19(3–4):255–259, 1998.
62. W. A. Burkhard. Nonrecursive traversals of trees. *Comput. J.*, 18(3):227–230, 1975.
63. R. G. Busacker and T. L. Saaty. *Finite Graphs and Networks: An Introduction with Applications*. McGraw-Hill, New York, 1965.
64. J. F. Buss and J. Goldsmith. Nondeterminism within P. *SIAM J. Comput.*, 22(3):560–572, 1993.
65. W. Bzyl. Adding native language support to the CWEB package and the \TeX program. *TUGboat*, 19(3):293–297, 1998.

66. R. Carraghan and P. M. Pardalos. An exact algorithm for the maximum clique problem. *Oper. Res. Lett.*, 9(6):375–382, 1990.
67. Y. H. Chang, J. S. Wang, and R. C. T. Lee. Generating all maximal independent sets on trees in lexicographic order. *Inform. Sci.*, 76(3–4):279–296, 1994.
68. G. Chartrand and L. Lesniak. *Graphs and Digraphs*. Chapman and Hall, London, 3rd edition, 1996.
69. G. Chartrand and O. R. Oellermann. *Applied and Algorithmic Graph Theory*. McGraw-Hill, New York, 1993.
70. S. S. Chawathe. Comparing hierarchical data in external memory. In *Proc. 25th Int. Conf. Very Large Data Bases*, pages 90–101, New York, 1999. Morgan Kaufmann.
71. S. S. Chawathe and H. García-Molina. Meaningful change detection in structured data. In *Proc. ACM SIGMOD Int. Conf. Management of Data*, pages 26–37, New York, 1997. ACM.
72. S. S. Chawathe, A. Rajaraman, H. García-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proc. ACM SIGMOD Int. Conf. Management of Data*, pages 493–504, New York, 1996. ACM.
73. J. Chen, I. Kanj, and W. Jia. Vertex cover: Further observations and further improvements. In *Proc. 25th Int. Workshop Graph-Theoretic Concepts in Computer Science*, volume 1665 of *Lecture Notes in Comput. Sci.*, pages 311–324, Berlin Heidelberg, 1999. Springer-Verlag.
74. J. Cheriyan and K. Mehlhorn. Algorithms for dense graphs and networks on the random access computer. *Algorithmica*, 15(6):521–549, 1996.
75. B. Childs. Errata: Literate programming, a practitioner’s view. *TUGboat*, 13(4):457–457, 1992.
76. B. Childs. Literate programming, a practitioner’s view. *TUGboat*, 13(3):261–268, 1992.
77. B. Childs, D. Dunn, and W. Lively. Teaching CS/1 courses in a literate manner. *TUGboat*, 16(3):300–309, 1995.
78. N. Christofides. *Graph Theory: An Algorithmic Approach*. Academic Press, New York, 1975.
79. M.-J. Chung. $O(n^{5/2})$ time algorithms for the subgraph isomorphism problem on trees. *J. Algorithms*, 8(1):106–112, 1987.

80. A. Cockburn. Supporting tailorable program visualisation through literate programming and fisheye views. *Inform. Soft. Technol.*, 43(13):745–758, 2001.
81. R. Cole, R. Hariharan, and P. Indyk. Tree pattern matching and subset matching in deterministic $O(n \log^3 n)$ -time. In *Proc. 10th Annual ACM-SIAM Symp. Discrete Algorithms*, pages 245–254, New York, 1999. ACM.
82. D. Cordes and M. Brown. The literate-programming paradigm. *IEEE Computer*, 24(6):52–61, 1991.
83. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge MA, 2nd edition, 2001.
84. D. G. Corneil and C. C. Gotlieb. An efficient algorithm for graph isomorphism. *J. ACM*, 17(1):51–64, 1970.
85. D. G. Corneil and D. G. Kirkpatrick. A theoretical analysis of various heuristics for the graph isomorphism problem. *SIAM J. Comput.*, 9(2):281–297, 1980.
86. K. Culik and D. Wood. A note on some tree similarity measures. *Inform. Process. Lett.*, 15(1):39–42, 1982.
87. I. J. Davis. A fast radix sort. *Comput. J.*, 35(6):636–642, 1992.
88. R. Dechter. Constraint networks. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, volume 1, pages 276–285, New York, 1992. John Wiley & Sons.
89. P. J. Denning. Announcing literate programming. *Commun. ACM*, 30(7):593, 1987.
90. N. J. Deo, J. M. Davis, and R. E. Lord. A new algorithm for digraph isomorphism. *BIT*, 17(2):16–30, 1977.
91. N. J. Deo and P. Micikevicius. Prüfer-like codes for labeled trees. *Congr. Numer.*, 151(1):65–73, 2001.
92. G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, Englewood Cliffs NJ, 1999.
93. M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994.
94. E. W. Dijkstra. A note on two problems in connection with graphs. *Numer. Math.*, 1(1):269–271, 1959.

95. Y. Dinitz, A. Itai, and M. Rodeh. On an algorithm of Zemlyachenko for subtree isomorphism. *Inform. Process. Lett.*, 70(3):141–146, 1999.
96. B. Dixon, M. Rauch, and R. E. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM J. Comput.*, 21(6):1184–1192, 1992.
97. R. G. Downey, M. R. Fellows, and U. Stege. Parameterized complexity: A framework for systematically confronting computational intractability. In F. Roberts, J. Kratochvil, and J. Nešetřil, editors, *Contemporary Trends in Discrete Mathematics*, volume 49 of *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*, pages 49–99. American Mathematical Society, Providence, RI, 1999.
98. S. E. Dreyfus and A. M. Law. *The Art and Theory of Dynamic Programming*. Academic Press, New York, 1977.
99. M. Dubiner, Z. Galil, and E. Magen. Faster tree pattern matching. *J. ACM*, 41(2):205–213, 1994.
100. P. Dublish. Some comments on the subtree isomorphism problem for ordered trees. *Inform. Process. Lett.*, 36(5):273–275, 1990.
101. P. J. Durand, R. Pasari, J. W. Baker, and C.-C. Tsai. An efficient algorithm for the similarity analysis of molecules. *Internet Journal of Chemistry*, 2(17), 1999.
102. J. Ebert. A versatile data structure for edge-oriented graph algorithms. *Commun. ACM*, 30(6):513–519, 1987.
103. J. Edmonds. Paths, trees, and flowers. *Canad. J. Math.*, 17(1):449–467, 1965.
104. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2: *Applications, Languages, and Tools*. World Scientific, Singapore, 1999.
105. H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 3: *Concurrency, Parallelism, and Distribution*. World Scientific, Singapore, 1999.
106. M. C. Er. Enumerating ordered trees lexicographically. *Comput. J.*, 28(5):538–542, 1985.

107. S. Even. *Graph Algorithms*. Computer Science Press, Rockville MD, 1979.
108. B. T. Fan, A. Panaye, and J.-P. Doucet. Comment on “Isomorphism, automorphism partitioning, and canonical labeling can be solved in polynomial-time for molecular graphs”. *J. Chem. Inf. Comp. Sci.*, 39(4):630–631, 1999.
109. J.-L. Faulon. Isomorphism, automorphism partitioning, and canonical labeling can be solved in polynomial-time for molecular graphs. *J. Chem. Inf. Comp. Sci.*, 38(3):432–444, 1998.
110. J. Feigenbaum, S. Kannan, M. Y. Vardi, and M. Viswanathan. Complexity of problems on graphs represented as ordered binary decision diagrams. In *Proc. 15th Annual Symp. Theoretical Aspects of Computer Science*, volume 1373 of *Lecture Notes in Comput. Sci.*, pages 216–226, Berlin Heidelberg, 1998. Springer-Verlag.
111. M.-L. Fernández and G. Valiente. A graph distance measure combining maximum common subgraph and minimum common supergraph. *Pattern Recogn. Lett.*, 22(6–7):753–758, 2001.
112. U. Finkler and K. Mehlhorn. Checking priority queues. In *Proc. 10th Annual ACM-SIAM Symp. Discrete Algorithms*, pages 901–902, New York, 1999. ACM.
113. P. Flajolet, P. Sipala, and J.-M. Steyaert. Analytic variations on the common subexpression problem. In *Proc. 17th Int. Coll. Automata, Languages, and Programming*, volume 443 of *Lecture Notes in Comput. Sci.*, pages 220–234, Berlin Heidelberg, 1990. Springer-Verlag.
114. L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, 1962.
115. J. Fox. Webless literate programming. *TUGboat*, 11(4):511–513, 1990.
116. C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, Reading MA, 1995.
117. M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984.

118. M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.
119. M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.
120. R. Freivalds. Fast probabilistic algorithms. In *Proc. 8th Int. Symp. Mathematical Foundations of Computer Science*, volume 74 of *Lecture Notes in Comput. Sci.*, pages 57–69, Berlin Heidelberg, 1979. Springer-Verlag.
121. H. N. Gabow. An efficient implementation of Edmonds’ algorithm for maximum matching in graphs. *J. ACM*, 23(2):221–234, 1976.
122. H. Galperin and A. Wigderson. Succint representations of graphs. *Inform. Contr.*, 56(3):183–198, 1983.
123. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to NP-Completeness*. W. H. Freeman, San Francisco CA, 1979.
124. G. Gati. Further annotated bibliography on the isomorphism disease. *Journal of Graph Theory*, 3:95–109, 1979.
125. L. Gerhards and W. Lindenbergh. Clique detection for nondirected graphs: Two new algorithms. *Computing*, 21(4):295–322, 1979.
126. D. E. Ghahraman, A. K. C. Wong, and T. Au. Graph monomorphism algorithms. *IEEE Trans. Syst. Man Cyb.*, 10(4):189–197, 1980.
127. A. Gibbons. *Algorithmic Graph Theory*. Cambridge University Press, Cambridge, 1985.
128. A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988.
129. S. W. Golomb and L. D. Baumert. Backtrack programming. *J. ACM*, 12(4):516–524, 1965.
130. M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.
131. M. Goossens, F. Mittelbach, and A. Samarin. *The L^AT_EX Companion*. Addison-Wesley, Reading MA, 1994.

132. M. Goossens and S. Rahtz. *The L^AT_EX Web Companion: Integrating T_EX, HTML, and XML*. Addison-Wesley, Reading MA, 1999.
133. M. Goossens, S. Rahtz, and F. Mittelbach. *The L^AT_EX Graphics Companion*. Addison-Wesley, Reading MA, 1997.
134. A. Goralcikova and V. Konbek. A reduct and closure algorithm for graphs. In *Proc. 8th Int. Symp. Mathematical Foundations of Computer Science*, volume 74 of *Lecture Notes in Comput. Sci.*, pages 301–307, Berlin Heidelberg, 1979. Springer-Verlag.
135. O. Gotoh. Consistency of optimal sequence alignments. *Bull. Math. Biol.*, 52(4):509–525, 1990.
136. R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, Reading MA, 2nd edition, 1994.
137. H. M. Grindley, P. J. Artymiuk, D. W. Rice, and P. Willett. Identification of tertiary structure resemblance in proteins using a maximal common subgraph isomorphism algorithm. *J. Mol. Biol.*, 229(3):707–721, 1993.
138. J. Gross and T. W. Tucker. *Topological Graph Theory*. John Wiley & Sons, New York, 1987.
139. J. Gross and J. Yellen. *Graph Theory and its Applications*. CRC Press, Boca Raton FL, 1999.
140. R. Grossi. Further comments on the subtree isomorphism for ordered trees. *Inform. Process. Lett.*, 40(5):255–256, 1991.
141. R. Grossi. A note on the subtree isomorphism for ordered trees and related problems. *Inform. Process. Lett.*, 39(2):81–84, 1991.
142. R. Grossi. On finding common subtrees. *Theor. Comput. Sci.*, 108(2):345–356, 1993.
143. L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th Annual Symp. Foundations of Computer Science*, pages 8–21, Piscataway NJ, 1978. IEEE.
144. K. Guntermann and J. Schrod. WEB adapted to C. *TUGboat*, 7(3):134–137, 1986.
145. D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge, 1997.

146. D. R. Hanson. *C Interfaces and Implementations: Techniques for Creating Reusable Software*. Addison-Wesley, Reading MA, 1996.
147. R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, 1980.
148. R. M. Haralick and J. S. Kartus. Arrangements, homomorphisms, and discrete relaxation. *IEEE Trans. Syst. Man Cyb.*, 8(8):600–612, 1978.
149. R. M. Haralick and L. G. Shapiro. The consistent labeling problem: Part I. *IEEE Trans. Pattern Anal. Machine Intell.*, 1(2):173–184, 1979.
150. R. M. Haralick and L. G. Shapiro. The consistent labeling problem: Part II. *IEEE Trans. Pattern Anal. Machine Intell.*, 2(3):193–203, 1980.
151. R. M. Haralick and L. G. Shapiro. *Computer and Robot Vision*, volume 2. Addison-Wesley, Reading MA, 1993.
152. F. Harary. *Graph Theory*. Addison-Wesley, Reading MA, 1969.
153. F. Harary and G. W. Wilcox. Boolean operations on graphs. *Math. Scand.*, 20(1):41–51, 1967.
154. X. He, M.-Y. Kao, and H.-I. Lu. Linear-time succinct encodings of planar graphs via canonical orderings. *SIAM J. Discrete Math.*, 12(3):317–325, 1999.
155. X. He, M.-Y. Kao, and H.-I. Lu. A fast general methodology for information-theoretically optimal encodings of graphs. *SIAM J. Comput.*, 30(3):838–846, 2000.
156. M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. *J. SIAM*, 10(1):196–210, 1962.
157. S. R. Heller, editor. *The Beilstein Online Database: Implementation, Content, and Retrieval*, volume 436 of *ACS Symposium Series*. American Chemical Society, Washington DC, 1990.
158. S. R. Heller, editor. *The Beilstein System: Strategies for Effective Searching*. American Chemical Society, Washington DC, 1998.
159. R. E. Hickson, C. Simon, and S. W. Perrey. The performance of several multiple protein-sequence alignment programs in relation to secondary-structure features for an rRNA sequence. *Mol. Biol. Evol.*, 17(4):530–539, 2000.

160. D. S. Hirschberg and S. S. Seiden. A bounded-space tree traversal algorithm. *Inform. Process. Lett.*, 47(4):215–219, 1993.
161. I. L. Hofacker, W. Fontana, P. F. Stadler, S. B. höf fer, M. Tacker, and P. Schuster. Fast folding and comparison of RNA secondary structures. *Monatshefte f. Chemie*, 125(2):167–188, 1994.
162. I. L. Hofacker, P. Schuster, and P. F. Stadler. Combinatorics of RNA secondary structures. *Discr. Appl. Math.*, 88(1–3):207–237, 1998.
163. C. M. Hoffmann and M. J. O’Donnell. Pattern matching in trees. *J. ACM*, 29(1):68–95, 1982.
164. D. R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books, New York, 20th edition, 1999.
165. J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.
166. J. E. Hopcroft and R. E. Tarjan. An $O(n \log n)$ algorithm for isomorphism of triconnected planar graphs. *J. Comp. Syst. Sci.*, 7(3):323–331, 1973.
167. J. Jansson and A. Lingas. A fast algorithm for optimal alignment between similar ordered trees. In *Proc. 12th Annual Symp. Combinatorial Pattern Matching*, volume 2089 of *Lecture Notes in Comput. Sci.*, pages 232–243, Berlin Heidelberg, 2001. Springer-Verlag.
168. T. Jian. An $O(2^{0.304n})$ algorithm for solving maximum independent set problem. *IEEE Trans. Comput.*, 35(9):847–851, 1986.
169. T. Jiang, L. Wang, and K. Zhang. Alignment of trees — an alternative to tree edit. *Theor. Comput. Sci.*, 143(1):137–148, 1995.
170. X. Y. Jiang and H. Bunke. Optimal quadratic-time isomorphism of ordered graphs. *Pattern Recogn.*, 32(7):1273–1283, 1999.
171. D. S. Johnson and C. C. McGeoch, editors. *Network Flows and Matching: First DIMACS Implementation Challenge*, volume 12 of *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, Providence, RI, 1993.
172. D. S. Johnson and M. A. Trick, editors. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, volume 26 of *DIMACS: Series in Discrete Mathematics and Theo-*

- retical Computer Science*. American Mathematical Society, Providence, RI, 1996.
- 173. H. J. Johnston. Cliques of a graph — variations on the Bron-Kerbosh algorithm. *Int. J. Comput. Inform. Sci.*, 5(3):209–238, 1976.
 - 174. H. Jung and K. Mehlhorn. Parallel algorithms for computing maximal independent sets in trees and for updating minimum spanning trees. *Inform. Process. Lett.*, 27(5):227–236, 1988.
 - 175. D. Jungnickel. *Graphs, Networks and Algorithms*. Springer-Verlag, Berlin Heidelberg, 1999.
 - 176. A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, 1962.
 - 177. S. Kannan, M. Moor, and S. Rudich. Implicit representation of graphs. *Discr. Appl. Math.*, 5(4):596–603, 1992.
 - 178. M. Kao. Multiple-size divide-and-conquer recurrences. *ACM SIGACT News*, 28(2):67–69, 1997.
 - 179. S. V. Kasparek. *Computer Graphics and Chemical Structures: Database Management Systems*. John Wiley & Sons, New York, 1990.
 - 180. D. Kennedy. T_EX adapted to CWEB. *TUGboat*, 9(2):124–125, 1988.
 - 181. P. Kikusts. Another algorithm determining the independence number of a graph. *Elektron. Inf. verarb. Kybern.*, 22(4):157–166, 1986.
 - 182. P. Kilpeläinen and H. Mannila. Ordered and unordered tree inclusion. *SIAM J. Comput.*, 24(2):340–356, 1995.
 - 183. V. King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18(2):263–270, 1997.
 - 184. L. Kitchen. Relaxation applied to matching quantitative relational structures. *IEEE Trans. Syst. Man Cyb.*, 10(2):96–101, 1980.
 - 185. L. Kitchen and A. Rosenfeld. Discrete relaxation for matching relational structures. *IEEE Trans. Syst. Man Cyb.*, 9(12):869–874, 1979.
 - 186. W. Knödel. Bestimmung aller maximalen, vollständigen Teilgraphen eines Graphen g nach Stoffers. *Computing*, 3(3):239–240, 1968.

187. D. E. Knuth. Optimum binary search trees. *Acta Inform.*, 1(1):14–25, 1971.
188. D. E. Knuth. Literate programming. *Comput. J.*, 27(2):97–111, 1984.
189. D. E. Knuth. *T_EX: The Program*, volume B of *Computers and Typesetting*. Addison-Wesley, Reading MA, 1986.
190. D. E. Knuth. *METAFONT: The Program*, volume D of *Computers and Typesetting*. Addison-Wesley, Reading MA, 1986.
191. D. E. Knuth. *Literate Programming*, volume 27 of *CLSI Lecture Notes*. Center for the Study of Language and Information, Stanford, California, 1992.
192. D. E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing*. ACM Press, New York, 1993.
193. D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading MA, 3rd edition, 1997.
194. D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading MA, 3rd edition, 1998.
195. D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading MA, 3rd edition, 1998.
196. D. E. Knuth. *MMIXware: A RISC Computer for the Third Millennium*. Springer-Verlag, Berlin Heidelberg, 2000.
197. D. E. Knuth and R. W. Moore. Estimating the efficiency of back-track programs. *Math. Comp.*, 29(1):121–136, 1975.
198. J. Köbler, U. Schöning, and J. Turán. *The Graph Isomorphism Problem: its Structural Complexity*. Progress in Theoretical Computer Science. Birkhäuser, Boston, 1993.
199. I. Koch. Enumerating all connected maximal common subgraphs in two graphs. *Theor. Comput. Sci.*, 250(1–2):1–30, 2001.
200. E. Korach and Z. Ostfeld. DFS tree construction: Algorithms and characterization. In *Proc. 14th Int. Workshop Graph-Theoretic Concepts in Computer Science*, volume 344 of *Lecture Notes in Comput. Sci.*, pages 87–106, Berlin Heidelberg, 1989. Springer-Verlag.

201. S. R. Kosaraju. Efficient tree pattern matching. In *Proc. 30th Annual Symp. Foundations of Computer Science*, pages 178–183, Piscataway NJ, 1989. IEEE.
202. D. Kozen. A clique problem equivalent to graph isomorphism. *ACM SIGACT News*, 10(1):50–52, 1978.
203. D. L. Kreher and D. R. Stinson. *Combinatorial Algorithms: Generation, Enumeration, and Search*. CRC Press, Boca Raton FL, 1999.
204. J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Amer. Math. Soc.*, 7(1):48–50, 1956.
205. L. Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley, Reading MA, 2nd edition, 1994.
206. J. Larrosa and G. Valiente. Constraint satisfaction algorithms for graph pattern matching. *Math. Struct. Comput. Sci.*, 2002. In press.
207. E. L. Lawler and D. E. Wood. Branch-and-bound methods: A survey. *Oper. Res.*, 14(1):699–719, 1966.
208. C. Y. Lee. An algorithm for path connection and its applications. *IRE Trans. Electronic Computers*, 10(3):346–365, 1961.
209. L.-Q. Lee, J. G. Siek, and A. Lumsdaine. The generic graph component library. In *Proc. ACM SIGPLAN Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 399–414, New York, 1999. ACM.
210. G. Levi. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *Calcolo*, 9(4):1–12, 1972.
211. G. Levi. Graph isomorphism: A heuristic edge-partitioning oriented program. *Computing*, 12(4):291–313, 1974.
212. S. Levy. WEB adapted to C, another approach. *TUGboat*, 8(1): 12–13, 1987.
213. S. Levy and D. E. Knuth. *The CWEB System of Structured Documentation*. Addison-Wesley, Reading MA, 1994.
214. G. Li and F. Ruskey. The advantages of forward thinking in generating rooted and free trees. In *Proc. 10th Annual ACM-SIAM Symp. Discrete Algorithms*, pages 939–940, New York, 1999. ACM.

215. Z. Ling and D. Y. Y. Yun. An efficient subcircuit extraction algorithm by resource management. In *Proc. 2nd Int. Conf. on Application Specific Integrated Circuits*, pages 9–14, Piscataway NJ, 1996. IEEE.
216. C. Lins. A first look at literate programming. *Struct. Prog.*, 10(1):60–62, 1989.
217. C. Lins. An introduction to literate programming. *Struct. Prog.*, 10(2):107–112, 1989.
218. S. Y. Lu. A tree-to-tree distance and its applications to cluster analysis. *IEEE Trans. Pattern Anal. Machine Intell.*, 1(2):219–224, 1979.
219. S. Y. Lu. A tree-matching algorithm based on node splitting and merging. *IEEE Trans. Pattern Anal. Machine Intell.*, 6(2):249–256, 1984.
220. A. K. Mackworth. Constraint satisfaction. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, volume 1, pages 285–293, New York, 1992. John Wiley & Sons.
221. E. Mäkinen. On the subtree isomorphism problem for ordered trees. *Inform. Process. Lett.*, 32(5):271–273, 1989.
222. W. G. Mallard and P. J. Linstrom, editors. *NIST Chemistry Web-Book*. Number 69 in *NIST Standard Reference Database*. National Institute of Standards and Technology, Gaithersburg MD, 2000.
223. U. Manber. Recognizing breadth-first search trees in linear time. *Inform. Process. Lett.*, 34(4):167–171, 1990.
224. D. W. Matula. Subtree isomorphism in $O(n^{5/2})$. *Ann. Discrete Math.*, 2(1):91–106, 1978.
225. M. A. McClure, T. K. Vasi, and W. M. Fitch. Comparative analysis of multiple protein-sequence alignment methods. *Mol. Biol. Evol.*, 11(4):571–592, 1994.
226. J. J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Inform. Sci.*, 19(3):229–250, 1979.
227. J. J. McGregor. Backtrack search algorithms and the maximal common subgraph problem. *Software — Practice and Experience*, 12(1):23–34, 1982.
228. J. J. McGregor and P. Willett. Use of a maximal common subgraph algorithm in the automatic identification of the ostensible

- bond changes occurring in chemical reactions. *J. Chem. Inf. Comp. Sci.*, 21(3):137–140, 1981.
229. J. A. McHugh. *Algorithmic Graph Theory*. Prentice Hall, Englewood Cliffs NJ, 1990.
230. P. M. McIlroy, K. Bostic, and M. D. McIlroy. Engineering radix sort. *Computing Systems*, 6(1):5–27, 1993.
231. B. D. McKay. Practical graph isomorphism. *Congr. Numer.*, 30(1):45–87, 1981.
232. K. Mehlhorn. *Graph Algorithms and NP-Completeness*, volume 2 of *Data Structures and Algorithms*. Springer-Verlag, Berlin Heidelberg, 1984.
233. K. Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms*. Springer-Verlag, Berlin Heidelberg, 1984.
234. K. Mehlhorn and S. Näher. LEDA: A platform for combinatorial and geometric computing. *Commun. ACM*, 38(1):96–102, 1995.
235. K. Mehlhorn and S. Näher. From algorithms to working programs: On the use of program checking in LEDA. In *Proc. 23rd Int. Symp. Mathematical Foundations of Computer Science*, volume 1250 of *Lecture Notes in Comput. Sci.*, pages 84–93, Berlin Heidelberg, 1998. Springer-Verlag.
236. K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, 1999.
237. W. Miller and E. W. Myers. A file comparison program. *Soft. Pract. Exper.*, 15(11):1025–1040, 1985.
238. E. M. Mitchell, P. J. Artymiuk, D. W. Rice, and P. Willett. Use of techniques derived from graph theory to compare secondary structure motifs in proteins. *J. Mol. Biol.*, 212(1):151–166, 1989.
239. H. B. Mittal. A fast backtrack algorithm for graph isomorphism. *Inform. Process. Lett.*, 29(2):105–110, 1988.
240. J. W. Moon and L. Moser. On cliques in graphs. *Israel J. Math.*, 3(1):23–28, 1965.
241. E. F. Moore. The shortest path through a maze. In *Proc. Int. Symp. Theory of Switching*, pages 285–292, Cambridge MA, 1959. Harvard University Press.
242. G. D. Mulligan and D. G. Corneil. Corrections to Bierstone’s algorithm for generating cliques. *J. ACM*, 19(2):244–247, 1972.

243. D. R. Musser and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, Reading MA, 1996.
244. E. W. Myers. An $O(nd)$ difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.
245. V. Nicholson, C.-C. Tsai, M. Johnson, and M. Naim. A subgraph isomorphism theorem for molecular graphs. In *Graph Theory and Topology in Chemistry*, number 51 in *Studies in Physical and Theoretical Chemistry*, pages 226–230. Elsevier, New York, 1987.
246. R. Niedermeier and P. Rossmanith. Upper bounds for vertex cover further improved. In *Proc. 16th Symp. Theoretical Aspects of Computer Science*, volume 1563 of *Lecture Notes in Comput. Sci.*, pages 561–570, Berlin Heidelberg, 1999. Springer-Verlag.
247. J. Nievergelt, R. Gasser, F. Mäser, and C. Wirth. All the needles in a haystack: Can exhaustive search overcome combinatorial chaos? In *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Comput. Sci.*, pages 254–274, Berlin Heidelberg, 1995. Springer-Verlag.
248. J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM J. Comput.*, 2(1):33–43, 1973.
249. T. Nishizeki and N. Chiba. *Planar Graphs: Theory and Algorithms*, volume 140 of *North-Holland Mathematical Studies*. North-Holland, Amsterdam, 1988.
250. A. S. Noetzel and S. M. Selkow. An analysis of the general tree-editing problem. In D. Sankoff and J. B. Kruskal, editors, *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, chapter 8. Center for the Study of Language and Information, Stanford, California, 1999.
251. T. Ohama, T. Kumazaki, H. Hori, and S. Osawa. Evolution of multicellular animals as deduced from 5S rRNA sequences: A possible early emergence of the mesozoa. *Nucl. Acids Res.*, 12(12):5101–5108, 1984.
252. M. Ohlrich, C. Ebeling, E. Ginting, and L. Sather. SubGemini: Identifying subcircuits using a fast subgraph isomorphism algorithm. In *Proc. 30th Annual ACM/IEEE Design Automation Conf.*, pages 31–37, Piscataway NJ, 1993. IEEE.
253. O. Ore. *Theory of Graphs*, volume 38 of *Colloquium Publications*. American Mathematical Society, Providence, RI, 1962.

254. R. E. Osteen. Clique detection algorithms based on line addition and line removal. *SIAM Journal on Applied Mathematics*, 26(1):126–135, 1974.
255. C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover, Mineola, New York, 1998.
256. C. H. Papadimitriou and M. Yannakakis. A note on succinct representations of graphs. *Inform. Contr.*, 71(3):181–185, 1987.
257. P. M. Pardalos and G. P. Rodgers. A branch and bound algorithm for the maximum clique problem. *Comput. Oper. Res.*, 19(5):363–375, 1992.
258. C.-H. Peng, B.-F. Wang, and J.-S. Wang. Recognizing unordered depth-search trees of an undirected graph in parallel. *IEEE Trans. Par. Dist. Syst.*, 11(6):559–570, 2000.
259. M. A. Perkowski, M. Chrzanowska-Jeske, A. Coppola, and E. Pierzchała. An exact algorithm for the technology fitting problem in the application specific state machine device. In *Proc. 1992 IEEE Int. Symp. Circuits and Systems*, pages 1977–1980, Piscataway NJ, 1992. IEEE.
260. S. L. Peyton-Jones and D. R. Lester. *Implementing Functional Languages: A Tutorial*. Prentice Hall, London, 1992.
261. H. Prüfer. Neuer Beweis eines Satzes über Permutationen. *Arch. Math. Phys.*, 27(1):142–144, 1918.
262. W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.
263. P. W. Purdom and C. A. Brown. *The Analysis of Algorithms*. Holt, Rinehart, and Winston, New York, 1985.
264. N. Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, 1994.
265. N. Ramsey and C. Marceau. Literate programming on a team project. *Soft. Pract. Exper.*, 21(7):677–683, 1991.
266. B. Randell. System structure for software fault tolerance. *IEEE Trans. Soft. Eng.*, 1(2):221–232, 1975.
267. L. C. Ray and R. A. Kirsch. Finding chemical records by digital computers. *Science*, 126(3278):814–819, 1957.
268. R. C. Read and D. G. Corneil. The graph isomorphism disease. *Journal of Graph Theory*, 1(4):339–363, 1977.

269. R. C. Read and R. J. Wilson. *An Atlas of Graphs*. Oxford University Press, Oxford, 1998.
270. J. H. Reif. Depth-first search is inherently sequential. *Inform. Process. Lett.*, 20(5):229–234, 1985.
271. E. M. Reingold, J. Nievergelt, and N. J. Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice Hall, Englewood Cliffs NJ, 1977.
272. E. M. Reingold and J. S. Tilford. Tidier drawing of trees. *IEEE Trans. Soft. Eng.*, 7(2):223–228, 1981.
273. S. W. Reyner. An analysis of a good algorithm for the subtree problem. *SIAM J. Comput.*, 6(4):730–732, 1977.
274. J. M. Robson. An improved algorithm for traversing binary trees without auxiliary stack. *Inform. Process. Lett.*, 2(1):12–14, 1973.
275. J. M. Robson. Algorithms for maximum independent sets. *J. Algorithms*, 7(3):425–440, 1986.
276. S. Roura. Improved master theorems for divide-and-conquer recurrences. *J. ACM*, 48(2):170–205, 2001.
277. M. A. Royberg. A search for common patterns in many sequences. *Comput. Appl. Biosci.*, 8(1):57–64, 1992.
278. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1: Foundations. World Scientific, Singapore, 1997.
279. M. Rudolf. Utilizing constraint satisfaction techniques for efficient graph pattern matching. In *Theory and Application of Graph Transformations*, volume 1764 of *Lecture Notes in Comput. Sci.*, pages 215–227, Berlin Heidelberg, 2000. Springer-Verlag.
280. B. E. Sagan. A note on independent sets in trees. *SIAM J. Discrete Math.*, 1(1):105–108, 1988.
281. G. Sajith and S. Saxena. Optimal parallel algorithms for coloring bounded degree graphs and finding maximal independent sets in rooted trees. *Inform. Process. Lett.*, 49(6):303–308, 1994.
282. G. Sajith and S. Saxena. Corrigendum: Optimal parallel algorithms for coloring bounded degree graphs and finding maximal independent sets in rooted trees. *Inform. Process. Lett.*, 54(5):305, 1995.

283. R. Samudrala and J. Moult. A graph-theoretic algorithm for comparative modeling of protein structure. *J. Mol. Biol.*, 279(1):287–302, 1998.
284. D. Sankoff and J. B. Kruskal, editors. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Center for the Study of Language and Information, Stanford, California, 1999.
285. C. A. Schevon and J. S. Vitter. A parallel algorithm for recognizing unordered depth-first search. *Inform. Process. Lett.*, 28(2):105–110, 1988.
286. C. Schlieder. Diagrammatic transformation processes on two-dimensional relational maps. *J. Visual Lang. Comput.*, 9(1):45–59, 1998.
287. D. C. Schmidt and L. E. Druffel. A fast backtracking algorithm to test directed graphs for isomorphism using distance matrices. *J. ACM*, 23(3):433–445, 1976.
288. U. Schöning. Graph isomorphism is in the low hierarchy. *J. Comp. Syst. Sci.*, 37(3):312–323, 1988.
289. G. D. Schuler, S. F. Altschul, and D. J. Lipman. A framework for multiple sequence construction and analysis. *Proteins Struct. Funct. Genet.*, 9(1):180–190, 1991.
290. R. Sedgewick. *Algorithms in C++*. Addison-Wesley, Reading MA, 1992.
291. R. Sedgewick and P. Flajolet. *Analysis of Algorithms*. Addison-Wesley, Reading MA, 1996.
292. R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4–5):464–497, 1996.
293. S. M. Selkow. The tree-to-tree editing problem. *Inform. Process. Lett.*, 6(6):184–186, 1977.
294. E. W. Sewell. How to MANGLE your software: the WEB system for Modula-2. *TUGboat*, 8(2):118–122, 1987.
295. E. W. Sewell. *Weaving a Program: Literate Programming in WEB*. Van Nostrand Reinhold, New York, 1989.
296. R. Shamir and D. Tsur. Faster subtree isomorphism. *J. Algorithms*, 33(2):267–280, 1999.

297. B. A. Shapiro and K. Zhang. Comparing multiple RNA secondary structures using tree comparisons. *Comput. Appl. Biosci.*, 6(4):309–318, 1990.
298. L. G. Shapiro and R. M. Haralick. Relational matching. *Applied Optics*, 26(10):1845–1851, 1987.
299. D. Shasha and C. Lazere. *Our of their Minds: The Lives and Discoveries of 15 Great Computer Scientists*. Springer-Verlag, Berlin Heidelberg, 1998.
300. D. Shasha, J. T.-L. Wang, K. Zhang, and F. Y. Shih. Exact and approximate algorithms for unordered tree matching. *IEEE Trans. Syst. Man Cyb.*, 24(4):668–678, 1994.
301. D. Shasha and K. Zhang. Fast algorithms for the unit cost editing distance between trees. *J. Algorithms*, 11(4):581–621, 1990.
302. J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, Reading MA, 2001.
303. L. Siklòssy. Fast and read-only algorithms for traversing trees without an auxiliary stack. *Inform. Process. Lett.*, 1(4):149–152, 1972.
304. S. Skiena. *The Algorithm Design Manual*. Springer-Verlag, Berlin Heidelberg, 1998.
305. S. Soule. A note on the nonrecursive traversal of binary trees. *Comput. J.*, 20(4):350–352, 1977.
306. T. Specht, N. Ulbrich, and V. Erdmann. Nucleotide sequence of the 5S rRNA from the Annelida species Enchytraeus albidus. *Nucl. Acids Res.*, 14(10):4372, 1986.
307. T. Specht, N. Ulbrich, and V. Erdmann. The secondary structure of the 5S ribosomal ribonucleic acid from the Annelida Enchytraeus albidus. *Endocyt. Cell Res.*, 4(1):205–214, 1987.
308. J. T. Stasko and J. S. Vitter. Pairing heaps: Experiments and analysis. *Commun. ACM*, 30(3):234–249, 1987.
309. R. E. Stobaugh. Chemical substructure searching. *J. Chem. Inf. Comp. Sci.*, 25(1):271–275, 1985.
310. M. Stoer and F. Wagner. A simple min-cut algorithm. *J. ACM*, 44(4):585–591, 1997.

311. G. F. Sullivan, D. S. Wilson, and G. M. Masson. Certification of computational results. *IEEE Trans. Comput.*, 44(7):833–847, 1995.
312. K. J. Supowit and E. M. Reingold. The complexity of drawing trees nicely. *Acta Inform.*, 18(4):377–392, 1983.
313. K.-C. Tai. The tree-to-tree correction problem. *J. ACM*, 26(3):422–433, 1979.
314. M. Talamo and P. Vocca. Representing graphs implicitly using almost optimal space. *Discr. Appl. Math.*, 108(1–2):193–210, 2001.
315. E. Tanaka. A metric between unrooted and unordered trees and its bottom-up computing method. *IEEE Trans. Pattern Anal. Machine Intell.*, 16(12):1233–1238, 1994.
316. E. Tanaka and K. Tanaka. The tree-to-tree editing problem. *Int. J. Pattern Recogn. and Artif. Intell.*, 2(2):221–240, 1988.
317. R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.
318. R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.
319. R. E. Tarjan. Graph algorithms in chemical computation. In R. E. Christoffersen, editor, *Algorithms for Chemical Computations*, volume 46 of *ACS Symposium Series*, chapter 1, pages 1–19. Americal Chemical Society, Washington DC, 1977.
320. R. E. Tarjan. Applications of path compression on balanced trees. *J. ACM*, 26(4):690–715, 1979.
321. R. E. Tarjan. *Data Structures and Network Algorithms*. SIAM, Philadelphia PA, 1983.
322. R. E. Tarjan and A. E. Trojanowski. Finding a maximum independent set. *SIAM J. Comput.*, 6(3):537–546, 1977.
323. H. Thimbleby. Experiences of “literate programming” using CWEB (a variant of Knuth’s WEB). *Comput. J.*, 29(3):201–211, 1986.
324. J. D. Thompson, D. G. Higgins, and T. J. Gibson. CLUSTAL W: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, positions-specific gap penalties and weight matrix choice. *Nucl. Acids Res.*, 22(22):4673–4680, 1994.

325. J. D. Thompson, F. Plewniak, and O. Poch. A comprehensive comparison of multiple sequence alignment programs. *Nucl. Acids Res.*, 27(13):2682–2690, 1999.
326. G. Torán. Succinct representations of graphs. *Discr. Appl. Math.*, 8(1):289–294, 1984.
327. E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London, 1993.
328. S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM J. Comput.*, 6(3):505–517, 1977.
329. S.-H. Tung. A structured method for literate programming. *Struct. Prog.*, 10(2):113–120, 1989.
330. W. T. Tutte. *Graph Theory As I Have Known It*, volume 11 of *Oxford Lecture Series in Mathematics and its Applications*. Oxford University Press, Oxford, 1998.
331. J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.
332. S. H. Unger. GIT — A heuristic program for testing pairs of directed line graphs for isomorphism. *Commun. ACM*, 7(1):26–34, 1964.
333. G. Valiente. An efficient bottom-up distance between trees. In *Proc. 8th Int. Symp. String Processing and Information Retrieval*, pages 212–219, Piscataway NJ, 2001. IEEE Computer Science Press.
334. G. Valiente. A general method for graph isomorphism. In *Proc. 13th Int. Symp. Fundamentals of Computation Theory*, volume 2138 of *Lecture Notes in Comput. Sci.*, pages 428–431, Berlin Heidelberg, 2001. Springer-Verlag.
335. P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Syst. Theory*, 10(2):99–127, 1977.
336. J. van Leeuwen. Graph algorithms. In *Handbook of Theoretical Computer Science*, volume A, chapter 10, pages 525–631. Elsevier, Amsterdam, 1990.
337. C. J. Van Wyk. Literate programming: An assessment. *Commun. ACM*, 33(3):361, 365, 1990.

338. C. J. Van Wyk, E. Hamilton, and D. Colner. Literate programming: Expanding generalized regular expressions. *Commun. ACM*, 31(12):1376–1385, 1988.
339. C. J. Van Wyk, D. R. Hanson, and J. Gilbert. Literate programming: Printing common words. *Commun. ACM*, 30(7):594–599, 1987.
340. C. J. Van Wyk, M. Jackson, and D. W. Wall. Literate programming: Processing transactions. *Commun. ACM*, 30(12):1000–1010, 1987.
341. C. J. Van Wyk, D. C. Lindsay, and H. Thimbleby. Literate programming: A file difference program. *Commun. ACM*, 32(6):740–755, 1989.
342. C. J. Van Wyk and N. Ramsey. Literate programming: Waving a language-independent WEB. *Commun. ACM*, 32(9):1051–1055, 1989.
343. J. G. Vaucher. Pretty-printing of trees. *Soft. Pract. Exper.*, 10(5):553–561, 1980.
344. V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag, Berlin Heidelberg, 2001.
345. R. M. Verma. Strings, trees, and patterns. *Inform. Process. Lett.*, 41(3):157–161, 1992.
346. R. M. Verma. A general method and a master theorem for divide-and-conquer recurrences with applications. *J. Algorithms*, 16(1):67–79, 1994.
347. R. M. Verma. General techniques for analyzing recursive algorithms with applications. *SIAM J. Comput.*, 26(2):568–581, 1997.
348. R. M. Verma and S. W. Reyner. An analysis of a good algorithm for the subtree problem, corrected. *SIAM J. Comput.*, 18(5):906–908, 1989.
349. M. Vihinen. An algorithm for simultaneous comparison of several sequences. *Comput. Appl. Biosci.*, 4(1):89–92, 1988.
350. J. Viksna and D. Gilbert. Pattern matching and pattern discovery algorithms for protein topologies. In *Algorithms in BioInformatics*, volume 2149 of *Lecture Notes in Comput. Sci.*, pages 98–111, Berlin Heidelberg, 2001. Springer-Verlag.

351. M. Vingron and P. Argos. Motif recognition and alignment of many sequences by comparison of dot-matrices. *J. Mol. Biol.*, 218(1):33–43, 1991.
352. M. Vingron and P. A. Pevzner. Multiple sequence comparison and consistency on multipartite graphs. *Adv. in Appl. Math.*, 16(1):1–22, 1995.
353. R. J. Walker. An enumerative technique for a class of combinatorial problems. In R. Bellman and M. Hall, editors, *Combinatorial Analysis*, volume 10, chapter 7, pages 91–94. American Mathematical Society, Providence, RI, 1960.
354. J. Q. Walker II. A node-positioning algorithm for general trees. *Soft. Pract. Exper.*, 20(7):685–705, 1990.
355. J. T.-L. Wang and K. Zhang. Finding similar consensus between trees: An algorithm and a distance hierarchy. *Pattern Recogn.*, 34(1):127–137, 2001.
356. T. Wang and J. Zhou. EMCSS: A new method for maximal common substructure search. *J. Chem. Inf. Comp. Sci.*, 37(5):828–834, 1997.
357. X. Wang and Q. Fu. A frame for general divide-and-conquer recurrences. *Inform. Process. Lett.*, 59(1):45–51, 1996.
358. H. Wasserman and M. Blum. Software reliability via run-time result-checking. *J. ACM*, 44(6):826–849, 1997.
359. M. S. Waterman. *Introduction to Computational Biology: Maps, Sequences, Genomes*. Chapman and Hall, London, 1995.
360. M. S. Waterman and T. F. Smith. RNA secondary structure: A complete mathematical analysis. *Math. Biosci.*, 42(1):257–266, 1978.
361. P. M. Weichsel. The Kronecker product of graphs. *Proc. Amer. Math. Soc.*, 13(1):47–52, 1963.
362. L. Weinberg. A simple and efficient algorithm for determining isomorphism of planar triply connected graphs. *IEEE Trans. on Circuit Theory*, 13(2):142–148, 1966.
363. M. A. Weiss. *Data Structures and Algorithm Analysis in C++*. Addison-Wesley, Reading MA, 2nd edition, 1999.
364. C. Wetherell and A. Shannon. Tidy drawing of trees. *IEEE Trans. Soft. Eng.*, 5(5):514–520, 1979.

365. A. T. White. *Graphs of Groups on Surfaces: Interactions and Models*, volume 188 of *North-Holland Mathematical Studies*. North-Holland, Amsterdam, 2001.
366. H. Whitney. Congruent graphs and the connectivity of graphs. *American J. Mathematics*, 54(1):150–168, 1932.
367. H. S. Wilf. *Combinatorial Algorithms: An Update*. SIAM, Philadelphia PA, 1989.
368. R. Wilhelm. A modified tree-to-tree correction problem. *Inform. Process. Lett.*, 12(3):127–132, 1981.
369. P. Willett. *Three-Dimensional Chemical Structure Handling*. John Wiley & Sons, New York, 1991.
370. P. Willett, J. M. Barnard, and G. M. Downs. Chemical similarity searching. *J. Chem. Inf. Comp. Sci.*, 38(6):983–996, 1998.
371. R. C. Wilson, A. N. Evans, and E. R. Hancock. Relational matching by discrete relaxation. *Image and Vision Computing*, 13(5):411–421, 1995.
372. D. R. Wood. An algorithm for finding a maximum clique in a graph. *Oper. Res. Lett.*, 21(5):211–217, 1997.
373. A. D. Woodall. Algorithm 43: A listed radix sort. *Comput. J.*, 12(4):406–406, 1969.
374. A. D. Woodall. Note on Algorithm 43: A listed radix sort. *Comput. J.*, 13(3):326, 1970.
375. S. Wu, U. Manber, G. Myers, and W. Miller. $O(np)$ sequence comparison algorithm. *Inform. Process. Lett.*, 35(6):317–323, 1990.
376. J. Xu. GMA: A generic match algorithm for structural homomorphism, isomorphism, and maximal common substructure match and its applications. *J. Chem. Inf. Comp. Sci.*, 36(1):25–34, 1996.
377. W. Yang. Identifying syntactic differences between two programs. *Soft. Pract. Exper.*, 21(7):739–755, 1991.
378. S. Zaks. Lexicographic generation of ordered trees. *Theor. Comput. Sci.*, 10(1):63–82, 1980.
379. B. Zelinka. Distances between rooted trees. *Math. Bohemica*, 116(1):101–107, 1991.

380. K. Zhang. Algorithms for the constrained editing distance between ordered labeled trees and related problems. *Pattern Recogn.*, 28(3):463–474, 1995.
381. K. Zhang. A constrained edit distance between unordered labeled trees. *Algorithmica*, 15(3):205–222, 1996.
382. K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262, 1989.
383. K. Zhang, R. Statman, and D. Shasha. On the editing distance between unordered labeled trees. *Inform. Process. Lett.*, 42(3):133–139, 1992.
384. K. Zhang, J. T.-L. Wang, and D. Shasha. On the editing distance between undirected acyclic graphs. *Internat. J. Found. Comput. Sci.*, 7(1):43–57, 1996.
385. M. Zucker and P. Stiegler. Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. *Nucl. Acids Res.*, 9(1):133–148, 1981.
386. A. Zündorf. Graph pattern matching in PROGRES. In *Graph Grammars and their Application to Computer Science*, volume 1073 of *Lecture Notes in Comput. Sci.*, pages 454–468, Berlin Heidelberg, 1996. Springer-Verlag.

Index

- α , *see* vertex cover number
- arc, 3
- β , *see* independence number
- backtracking, 70–80
 - tree, 74
 - pruning, 74
- branch-and-bound, 80–85
 - tree, 81
- clique, 300
 - maximal, 300
 - maximum, 300
 - number, 300
- cycle, 6
- divide-and-conquer, 85–94
 - tree, 89
- domino principle, 73, 81
- dynamic programming, 94–107
 - bottom-up, 95
 - memoization, 95
 - top-down, 95
- edge, 8
 - adjacent, 4
- edit graph, 61
- edit operation, 56
 - deletion, 56
 - insertion, 56
 - substitution, 56
- forest
 - spanning, 21
 - – breadth-first, 276
 - – depth-first, 256
- graph, 3
 - k -partite, 14
 - bidirected, 8
 - bipartite, 13
 - complete, 11
- complete bipartite, 13
- connected, 10
- cycle, 12
- degree sequence, 9
- isomorphism
- – ordered, 289
- labeled, 15
- layout, 139
- order, 3
- path, 12
- regular, 14
- representation
- – adjacency list, 38
- – adjacency matrix, 36
- size, 4
- strongly connected, 10
- traversal, 255
 - – breadth-first, 277
 - – depth-first, 258
 - – depth-first leftmost, 271
- undirected, 8
- wheel, 13
- independence number, 323, 324
- independence principle, 86
- independent set, 322, 324
 - maximal, 323, 324
 - maximum, 323, 324
- optimality principle, 94
- path, 5
 - closed, 6
- subgraph, 7
 - induced, 7
- subtree
 - common
 - – ordered, 206, 224
 - – unordered, 211, 230
 - maximal common
 - – ordered, 206, 224

- unordered, 211, 230
- maximum common
- ordered, 206, 224
- unordered, 211, 230
- ordered, 171
- bottom-up, 171
- isomorphic, 172, 187
- top-down, 171
- unordered, 170
- bottom-up, 170
- isomorphic, 176, 192
- top-down, 170
- trail, 5
- closed, 6
- tree, 16
- depth, 17
- edit distance, 61
- edit operation
- cost, 60
- height, 18
- isomorphism
- code, 158
- layout, 139
- layered, 139
- mapping, 58
- ordered
- isomorphic, 152
- representation
- array-of-parents, 46
- first-child, next-sibling, 47
- root, 16
- spanning, 20
- transformation, 57
- cost, 61
- valid, 59
- traversal, 113
- bottom-up, 130
- postorder, 119
- preorder, 114
- symmetric order, 145
- top-down, 125
- undirected, 11
- unordered
- isomorphic, 157
- vertex, 3
- adjacent, 4
- degree, 5, 9
- incident, 4
- indegree, 4
- outdegree, 4
- source, 4
- target, 4
- vertex cover, 333, 335
- minimal, 333, 335
- minimum, 333, 336
- vertex cover number, 333, 336
- walk, 5
- closed, 6