FIELDS INSTITUTE MONOGRAPHS

THE FIELDS INSTITUTE FOR RESEARCH IN MATHEMATICAL SCIENCES

Efficient Graph Representations

Jeremy P. Spinrad

American Mathematical Society
Providence, Rhode Island

The Fields Institute for Research in Mathematical Sciences

The Fields Institute is named in honour of the Canadian mathematician John Charles Fields (1863–1932). Fields was a visionary who received many honours for his scientific work, including election to the Royal Society of Canada in 1909 and to the Royal Society of London in 1913. Among other accomplishments in the service of the international mathematics community, Fields was responsible for establishing the world's most prestigious prize for mathematics research—the Fields Medal.

The Fields Institute for Research in Mathematical Sciences is supported by grants from the Ontario Ministry of Education and Training and the Natural Sciences and Engineering Research Council of Canada. The Institute is sponsored by McMaster University, the University of Toronto, the University of Waterloo, and York University, and has affiliated universities in Ontario and across Canada.

2000 *Mathematics Subject Classification.* Primary 05C62, 05C17, 05C50, 05C85, 05-00, 05-02, 68R10, 68W01, 68P05, 68Q30, 68-01.

For additional information and updates on this book, visit
www.ams.org/bookpages/fim-19

Library of Congress Cataloging-in-Publication Data

Spinrad, Jeremy P.
Efficient graph representations / Jeremy P. Spinrad.
p. cm. — (Fields Institute monographs ; 19)
Includes bibliographical references and index.
ISBN 0-8218-2815-0 (acid-free paper)
1. Representations of graphs. I. Title. II. Series.

QA166.242.S65 2003 2003045106
511′.–dc21 CIP

Copying and reprinting. Individual readers of this publication, and nonprofit libraries acting for them, are permitted to make fair use of the material, such as to copy a chapter for use in teaching or research. Permission is granted to quote brief passages from this publication in reviews, provided the customary acknowledgment of the source is given.

Republication, systematic copying, or multiple reproduction of any material in this publication is permitted only under license from the American Mathematical Society. Requests for such permission should be addressed to the Acquisitions Department, American Mathematical Society, 201 Charles Street, Providence, Rhode Island 02904-2294, USA. Requests can also be made by e-mail to `reprint-permission@ams.org`.

© 2003 by the American Mathematical Society. All rights reserved.
The American Mathematical Society retains all rights
except those granted to the United States Government.
Printed in the United States of America.

⊗ The paper used in this book is acid-free and falls within the guidelines
established to ensure permanence and durability.
This publication was prepared by The Fields Institute.
Visit the AMS home page at `http://www.ams.org/`

10 9 8 7 6 5 4 3 2 1 08 07 06 05 04 03

Contents

Explanatory Remarks 1

Chapter 1. Introduction 5
1.1. Graph Theory Background and Terminology 6
1.2. Algorithm Background and Terminology 6
1.3. Representation Background 8
1.4. Example of a Nice Representation 11
1.5. Overview of Problems in Graph Representation 12
1.6. Exercises 15

Chapter 2. Implicit Representation 17
2.1. Implicit Representation and Universal Graphs 21
2.2. Generalized Implicit Representation 22
2.3. Representations with Very Short Labels 23
2.4. Distance Labeling of Graphs 25
2.5. Exercises 27

Chapter 3. Intersection and Containment Representations 31
3.1. Chordality and Transitive Orientation 31
3.2. Techniques for Interval Graphs 33
3.3. Generalizations of Interval Graphs 34
3.4. Permutation Graphs and Generalizations 38
3.5. Containment Representations 41
3.6. Overlap Representations 42
3.7. Generalized Intersection Models 44
3.8. Perfect Graphs 46
3.9. Exercises 47

Chapter 4. Real Numbers in Graph Representations 53
4.1. Warren's Theorem 54
4.2. Continuous Nongeometric Variables 56
4.3. Decidability Results 57
4.4. Exercises 57

Chapter 5. Classes Which use Global Information 59
5.1. Paths in Trees 59
5.2. Chordal Comparability Graphs 60
5.3. Fill-in Schemes 65
5.4. Closure Operations 66
5.5. Weakly Chordal Graphs 67

5.6. Other Fill-in Schemes 68
5.7. Exercises 68

Chapter 6. Visibility Graphs 73
6.1. Counting Visibility Graphs 74
6.2. Clique Cover Representation 74
6.3. Induced Visibility Graphs 76
6.4. Optimization on Visibility Graphs 80
6.5. Line of Sight Graphs and Other Variants 80
6.6. Exercises 81

Chapter 7. Intersection of Graph Classes 85
7.1. Some Fundamental Properties 85
7.2. Intersections of Fundamental Properties 87
7.3. Weakly Chordal Comparability Graphs 88
7.4. Other Generalized Classes 90
7.5. Observations Regarding AT-free co-AT-free Graphs 91
7.6. Open Problems on the Generalized Classes 94
7.7. Exercises 95

Chapter 8. Graph Classes Defined by Forbidden Subgraphs 97
8.1. Cographs 97
8.2. Classes Which are too Large to have Efficient Representations 100
8.3. Relation Between Recognition Problems 105
8.4. Classes defined by Forbidding Sets of Induced Subgraphs 105
8.5. Dilworth Number and Poset Width 107
8.6. Exercises 109

Chapter 9. Chordal Bipartite Graphs 111
9.1. Γ-free Matrices 111
9.2. Counting and Representation 112
9.3. Characterizations 118
9.4. Recognition 120
9.5. Optimization Problems on Chordal Bipartite Graphs 123
9.6. Variants of Chordal Bipartite Graphs 125
9.7. Subclasses of Chordal Bipartite Graphs 126
9.8. Perfect Elimination Bipartite Graphs 130
9.9. Bipartite Graphs with Forbidden Induced Subgraphs 131
9.10. Exercises 132

Chapter 10. Matrices 135
10.1. Small Forbidden Classes of Matrices 135
10.2. Linear Matrices 136
10.3. Forbidden 2 by 2 Identity Matrices 138
10.4. Forbidding $\binom{10}{10}$ 139
10.5. Other Classes of Interest 142
10.6. The Problems of Counting and Representation 142
10.7. Other Matrix Properties 145
10.8. Exercises 145

Chapter 11. Decomposition 149

11.1. Substitution Decomposition and Vertex Partitioning 149
11.2. Join Decomposition 160
11.3. Recursively Decomposable Graphs 163
11.4. Clique-width and NLC-width 164
11.5. Clique Separator Decomposition 167
11.6. Skew Partition 170
11.7. 2-Join 174
11.8. Exercises 175

Chapter 12. Elimination Schemes 181
12.1. Distance Hereditary Graphs 181
12.2. Strongly Chordal Graphs 182
12.3. k-Simplicial Elimination Schemes 184
12.4. Doubly and Dually Chordal Graphs 185
12.5. Exercises 187

Chapter 13. Recognition Algorithms 191
13.1. Chordal Graphs 191
13.2. Interval Graphs 193
13.3. Circular-Arc Graph Recognition 197
13.4. Restrictions on Intervals and Arcs 204
13.5. Trapezoid Graphs and Related Classes 208
13.6. Circle Graphs 212
13.7. Circular Permutation Graphs 217
13.8. Weakly Chordal Graphs 218
13.9. Paths in Trees 219
13.10. NP-complete Classes 222
13.11. Open Classes 224
13.12. Exercises 224

Chapter 14. Robust Algorithms for Optimization Problems 231
14.1. Robust Algorithms which are Faster Than Recognition 233
14.2. Problems Helped by a Representation 241
14.3. Open Problems for Robust Algorithms 247
14.4. Complexity Issues 249
14.5. Exercises 252

Chapter 15. Characterization and Construction 257
15.1. Chordal Graphs 257
15.2. Unit Interval Graphs 259
15.3. Unit Circular-Arc Graphs 260
15.4. Construction Problem for NP-complete Classes 261
15.5. Exercises 262

Chapter 16. Applications 265
16.1. Computational Biology 265
16.2. Networks 268
16.3. Programming Languages 272
16.4. A Cautionary Tale 272
16.5. Exercises 273

Glossary 277

Survey of Results on Graph Classes 303

Bibliography 319

Index 337

Explanatory Remarks

The book is intended to be a monograph for researchers and advanced graduate courses. I view the following parts of the book as the most important contributions.

1) I feel that studying representation issues on graphs is very natural, and has not been studied in and of itself before. I have included a large number of open problems in the field, which I hope will stimulate research.
2) The issues raised in the optimization chapter, refining the notions of what it means to solve a problem on a class of graphs, have struck some people I have talked to as radical ideas, but I think that they are completely correct, natural, and important.
3) The recognition chapter gives a much more current view of important algorithmic developments in the field of intersection graph classes than is currently available (the current standard is Golumbic's book from 1980).
4) Some of the individual classes of graphs are important, and not covered adequately in any current text.

Let me now present a 'road map' of the book. Chapter 1 deals with background, and discusses some very general issues regarding representation of graphs. Chapter 2 presents a specific model called implicit representation, which forms one of the main topics of the book. Chapter 3 gives a brief overview of known graph classes which have good representations; these classes will arise repeatedly throughout the book, but the chapter can be used as a reference if the reader desires, rather than as required reading. A general notion of representing a graph by intersection models is studied at the end of the chapter. Together, chapters 1-3 can be viewed as an introduction, with chapter 2 being particularly crucial.

Chapters 4-12 deal with graph classes which seem particularly challenging to represent. In most of these chapters, the unifying concept is a method for defining a graph class, with individual sections devoted to particular graph classes which can be defined in this way. In chapters 6 and 9, we discuss individual graph classes that pose particularly interesting problems with respect to computer representation.

Chapter 13 deals with recognition algorithms for graph classes. Perhaps because I view recognition algorithms as my particular area of expertise, considerable space is devoted to trying to present algorithms for a large number of graph classes where the algorithm has previously been viewed as too complex for presenting in a general text.

Chapter 14 deals with new issues which arise when solving problems on graph classes which have special forms of representation. The concept of a robust algorithm is introduced as a new requirement for an algorithm. Examples of tractable and intractable problems for robust algorithms are presented.

Chapter 15 deals with issues of constructing representations which are different from existence of representation for a graph class, and chapter 16 presents a brief look at some of the applications which have come about since the publication of previous books which in general cover applications of graph classes in more detail than this book.

I feel that a course based on this book should include chapters 2 and chapter 14. Choice of other chapters can be based on which graph classes are judged most interesting by the instructor. As a computer scientist, I spend a great deal of time on chapter 13, but a course designed for mathematicians might place less emphasis on this material. I choose to take at least some material from each chapter in my course, but it is also possible to devote more time to a smaller number of chapters.

A number of chapters include material which has not been published in any form before. Of particular interest in this context is the chapter on matrices, and the section on induced visibility graphs. Much of the material in the chapter on intersection of graph classes is also new, but the results there are much more partial and are included primarily to introduce new open problems.

I have left out many topics that are at least somewhat relevant. For example, the work started by Galparin and Wigderson relating time complexity of very simple problems to graph classes with extremely efficient representations could be considered for inclusion. I may feel particularly guilty about this omission since it was my introduction to thinking about graph representation size, but have currently chosen not to include it since the techniques used are quite different from those stressed in the rest of the book.

The exercises in this book vary considerably in level of difficulty. I have intentionally included many problems which are quite challenging; some of these are marked with a star. It has been a goal to have a sufficient quantity and variety of exercises to be interesting for either a fresh student in the field, or for someone with some research experience. As a collector of problems, both open and homework, I would love to hear of good problems for possible future inclusion from any interested reader. If there is sufficient interest, I intend to maintain a list of open problems relating to this work on the web.

I should add one other general comment on an issue that could disturb some readers. There is a certain type of proof which is important to include in a journal article, but which I think should not be included in a text. For example, suppose that we show that if x is adjacent to y in a graph of a certain type, we can always find a forbidden induced subgraph, and this can be proved by simply examining a large number of cases. An algorithm for working on graphs without these forbidden subgraphs might then assume that x and y are nonadjacent. It is important to know that this result is correct, which is why a proof must be verified in a journal. However, the reader who wants to understand the algorithm only needs to know that x and y are nonadjacent, and if simply stating this as a fact clarifies the description of an algorithm, I prefer to give the result without a proof. Thus, in many sections, proofs are left to the journal and omitted from this book. Of course, I may have omitted (or inadvertently included) individual proofs in a fashion that is not consistent with the rest of the book.

I have many people to thank for helping this book to be written. I would particularly like to thank fellow researchers Stathis Zachos, Ross McConnell, Andreas Brandstädt, Dieter Kratsch, Derek Corneil, Peter Hammer, R. Sritharan, Elaine Eschen, Ryan Hayward, and Lorna Stewart, Anne Berry, as well as several anonymous referees for providing valuable comments. I also appreciate the assistance of students who have pointed out issues and errors in earlier drafts of the book; in particular, the book has been improved by the comments of Ian Stobert, Julie Johnson, Dana Gaston, Erika King, and Vera Rayevskaya. I thank the Vanderbilt Research Council for helping to support me during my sabbatical year in which much of the book was written. Of course, no words can express the support given from my family; thank you for everything Barbara, Demetria, and Thalia.

CHAPTER 1

Introduction

Graph theory has developed into a very broad field since its introduction by Euler in the 18th century. Graphs are an intuitively pleasing and flexible method for dealing with relationships between objects, and are used in so many areas of mathematical research that they need no justification for study in their own right. This book is written from both a mathematical and computer science perspective. In the mathematical tradition, graphs and classes of graphs are studied with no specific applications in mind, although many of the classes discussed arose from important applications. Since my research area is graph algorithms, I place much greater emphasis on algorithms for dealing with problems than is found in most mathematical texts. Hopefully, the reader will agree that synthesizing the two traditions opens a number of interesting new research areas, and new open problems can be found throughout this book.

I have always been interested in representations of graphs. There are at least two natural areas of research which spring to mind if one talks of graph representation. One approach is to think of how to make a graph, typically described by a set of connections between points, into a picture which is easily understood by a human. This area of research is called graph drawing. The field of graph drawing has become very popular recently, and while I enjoy reading books on this area such as [30], I am not the person to write such a book. As well as not being my research area, anyone who has seen my office or choice of clothing would agree that I am not qualified to work in an area in which one must decide whether the result is aesthetically pleasing!

The subject of this book can be viewed as the reverse process of graph drawing. In graph drawing, we imagine a representation of a graph is given which is easy for a machine to work with, and we want to convert this into a representation which can be viewed easily by a human. This book talks about methods for storing graphs which make them easy for the computer to work with. I choose to call this topic efficient graph representation. I will use the term efficient representation informally throughout the book to capture a variety of good properties for a representation. We will develop certain well defined notions of efficient representations in later chapters.

The closest generally accepted area of research to the work here is the field of intersection graphs. Golumbic's book [223] has had a major effect on both the study of intersection graphs, and my own research. In general, I have tried to avoid going deeply into subjects which are well covered by Golumbic. Other good books which are related or have affected my own work include [461], [482], [194], [361].

1.1 Graph Theory Background and Terminology

Definitions of terms used in this book are given in the glossary. This section presents a few fundamental definitions, emphasizing usage of common terms which have alternative definitions in the field.

A graph $G = (V,E)$, where V is a set of vertices, and E is a set of edges. Each edge corresponds to a pair (x,y) of vertices. If the edges are ordered pairs of vertices, we call G a digraph.

In this book, we will not in general deal with multigraphs or loops; that is, there is at most one edge (x,y) in a graph, and there is no edge (x,x). Note that if G is a digraph, there may be both an edge (x,y) and an edge (y,x).

Two vertices x,y of a graph are adjacent if they are connected by an edge. The neighbors of x, sometimes denoted $\mathrm{N}(x)$, are the set y such that $(x,y) \in E$.

The subgraph of $G = (V,E)$ induced by a set S of vertices is the graph $H = (S,E_H)$, where E_H is the set of edges of E such that both endpoints are in S.

We will refer to the number of vertices in a graph as n, and the number of edges as m. We will assume that vertices of a graph are labeled as integers from 1 through n, unless noted otherwise.

If C is a class of graphs, the class of co-C graphs is defined to be those graphs G such that the complement $\overline{G}$ of G is in C. The complement $\overline{G}$ of G has the same vertex set as G, and x is adjacent to y in $\overline{G}$ if and only if x and y are nonadjacent in G.

1.2 Algorithm Background and Terminology

This book is intended to be of interest to both mathematicians and computer scientists. In most cases, algorithmic topics will be presented separately from other representation issues, but there is one notational device from the world of computer science which is absolutely essential. We will rely heavily on order notation, as described below.

Order notation is a form of mathematical notation devised to ignore constant factors, and preserve larger differences between functions. We will use the following order notation symbols: O, o, Ω, Θ, which respectively mean $\leq$, $<$, $\geq$, and $=$, if constant factors are ignored. We give a formal definition of O below. $\mathrm{f}(n)$ is $\Theta(\mathrm{g}(n))$ if $\mathrm{f}(n)$ is $\mathrm{O}(\mathrm{g}(n))$ and $\mathrm{g}(n)$ is $\mathrm{O}(\mathrm{f}(n))$. $\mathrm{f}(n)$ is $\mathrm{o}(\mathrm{g}(n))$ if $\mathrm{f}(n)$ is $\mathrm{O}(\mathrm{g}(n))$, but $\mathrm{g}(n)$ is not $\mathrm{O}(\mathrm{f}(n))$. Finally, we use $\mathrm{f}(n)$ is $\Omega(\mathrm{g}(n))$ to denote that $\mathrm{g}(n)$ is $\mathrm{O}(\mathrm{f}(n))$.

I should note that there are slight variants of definition in the literature. Using the definitions below it is possible to have functions $\mathrm{f}(n)$ and $\mathrm{g}(n)$ such that $\mathrm{f}(n)$ is not $\mathrm{O}(\mathrm{g}(n))$, and $\mathrm{f}(n)$ is not $\Omega(\mathrm{g}(n))$; an example would be $\mathrm{f}(n) = n$ for even values of n and n^2 for odd values of n, while $\mathrm{g}(n) = n^2$ for even values of n and n for odd values. This can be avoided by changing the definition of Ω to require that $\mathrm{f}(n) \geq c\mathrm{g}(n)$ infinitely often, and this usage is more meaningful in some contexts. For example, I would argue that an algorithm which runs in n^2 time for odd n and n time for even n should be called $\Theta(n^2)$, $\mathrm{O}(n^2)$, and $\Omega(n^2)$, and for this the definition of Ω using infinitely often is necessary. Although the best definitions are debatable in general, there are no functions used in this book where the different definitions affect any problems or algorithms. I use the definitions above so that the reader who is unfamiliar with order notation can understand all other definitions from the formal definition of O, which we give below.

We say f(n) is O(g(n)) if there are constants c and N such that for all $n > N$, f(n) $\leq cg(n)$.

Measures of time complexity in graph algorithms are based on the number of vertices and the number of edges in the input graph. For graphs, we use the term linear time to mean O(n+m) time; the term generally means time proportional to the size of the input.

Some readers might notice that while we present upper bounds in the form of algorithms with certain time complexities for solving problems, we never present lower bounds which say that the problem requires a certain amount of time. There is a good reason for this; we know next to nothing about proving nonlinear lower bounds for practical problems, unless we place restrictions on the type of operation which is allowed.

For example, the well known $\Omega(n\log n)$ lower bound on sorting applies only if we assume that the algorithm is restricted to using comparisons between elements to get information on the ordering. Teaching this sorting lower bound may even mislead some students into thinking that we can find proofs that our nonlinear algorithms are optimal.

In my opinion, finding superlinear lower bounds for polynomially solvable problems with unrestricted operations allowed is one of the great challenges in computer science. Thus, any nonlinear algorithm in the book is theoretically a candidate for improvement.

Our only 'lower bounds' on polynomially solvable problems will be pointing out when an improvement might be difficult to find, since it would improve the time complexity of a problem which has been studied extensively. For example, we will show that linear time algorithms for certain problems, including the recognition problem for a number of graph classes, would imply linear time algorithms for determining whether a graph has a triangle. This will not prove that a linear time algorithm for these problems is impossible. However, since triangle-free graphs have been studied extensively and no linear time algorithm for recognizing them is known, it is useful to realize that it may be very difficult to find a linear time algorithm for these problems.

Our other notion of a 'lower bound' is NP-completeness. Readers who want more information on the subject of NP-completeness are advised to study [206]. A decision (yes/no) problem is in P if it can be solved in polynomial time. A decision problem is in NP if it can be solved in polynomial time by a nondeterministic Turing machine; for such problems, you can give a polynomial length proof for any yes instance of the problem. A problem is NP-complete if the problem is in NP, and all problems in NP can be polynomially transformed into the problem. If a problem is NP-complete, we can at least say that finding a polynomial time solution for the problem would require a magnificent breakthrough in research.

Important NP-complete problems on graphs include the clique problem, the chromatic number (coloring) problem, independent set, clique cover, Hamilton cycle; many others can be found in [206]. We will define the problems which come up repeatedly in the book at this point; many of these problems are tractable for classes of graphs studied in the book. Other optimization problems will be defined when they arise.

A clique C in a graph is a set of vertices such that each pair of vertices in C is connected by an edge. The clique problem takes a graph G and an integer k, and asks whether G contains a clique of size k. The maximum clique problem involves

finding the size of a largest clique in the graph. A clique C is maximal if no vertex can be added to C such that the resulting set is still a clique.

An independent set (also called a stable set) is the complement of a clique; that is, every pair of vertices in the set must be nonadjacent. Maximum and maximal independent sets are defined analogously to the definitions above.

A coloring of a graph is an assignment of numbers (called colors) to vertices such that no pair of adjacent vertices is assigned the same color. The chromatic number of G is the minimum number of colors used in any valid coloring of G. The coloring, or chromatic number problem, takes a graph G and an integer k and asks whether G can be colored with at most k colors.

A clique cover of a graph is a partition of the vertices into subsets such that each subset induces a clique. The clique cover number of a graph is the minimum number of sets in such a partition.

A Hamilton cycle of a graph is a cycle which includes every vertex of the graph exactly once.

We now must deal with one issue that can cause confusion, known as the unit-cost assumption. In this assumption, standard operations such as addition, multiplication, and finding the element at position i in an array are assumed to take constant time if the input numbers are of size $O(n^k)$, that is, the number of bits in the input is $O(\log n)$. I will not discuss the reasoning behind the unit-cost assumption; if we used different assumptions, our time bounds for standard algorithms would differ from those used in the literature.

We bring up the unit-cost assumption here because we will *not* be using the unit cost assumption with respect to space complexity. When we discuss a storage mechanism for a class of graphs, we will be counting the number of bits in the representation rather than the number of integers in the range 1..O(n). We must count bits of space, since we are discussing what the limits are in a compact representation of a graph. These limits come easily from the number of bits; any set with $2^{f(n)}$ members must require a bit string of length f(n) if we require that each member has a different name. However, if we did not count bits, it would not be accurate to say that f(n) space was required; the precise amount of space would depend on how many bits are allowed to be treated as a constant unit of space.

We still measure space complexity in terms of order notation, since this makes our analysis much cleaner and clarifies a number of open problems. Note that it might be more consistent to use bit complexity for measurements of time as well as space. However, being forced to say (for example) that binary search runs in $O(log^2 n)$ time when all other sources use $\log n$ as the time bound would cause a great deal of confusion, and I feel that this confusion outweighs the benefits of using the same measurement system for time and space.

1.3 Representation Background

As a researcher in computer representation of graphs, the treatment in some introductory discrete mathematics and data structures texts is particularly irritating. Most texts introduce the following two forms of graph representation, which we will also use in this book.

The adjacency matrix A of a graph G is a matrix with n rows and n columns, such that $A[x,y] = 1$ if and only if there is an edge from x to y. Adjacency matrices may represent either directed graphs or undirected graphs.

Adjacency lists store a graph by keeping a list of the neighbors of each vertex. Generally, one stores these as an array of lists, so that one can have access to the list of neighbors of i in constant time. A graph is stored by storing the adjacency lists of all vertices. In standard terminology, an adjacency list keeps an unordered list of neighbors of each vertex; the lists can be sorted in linear time if desired (exercise 1.1).

These definitions are presented adequately in most texts. However, the explanation that goes with them is often faulty. Combining misstatements from several texts, we get the following explanation.

Straw Man's[1] comments on graph representations: There are two methods for representing a graph in a computer; adjacency matrices, and adjacency lists. It is faster to work with adjacency matrices, but they use more space than adjacency lists, so you will choose one or the other depending on which resource is more important to you.

Of course, there are many interesting forms of graph representation; that is one of the points of this book! The notion of time/space tradeoff between the two forms of representation given is also very misleading, since there are problems which can be provably solved faster on adjacency lists than adjacency matrices, and the amount of space used to store a random graph is larger with adjacency lists than with adjacency matrices. Let us look at the time/space tradeoff a bit more carefully.

If all information about the input graph comes from queries of the form 'is there an edge between x and y?', then adjacency matrices are clearly faster, since a query can be answered in constant time, whereas this can involve traversing a long list if adjacency lists are used. However, many algorithms use operations such as 'find a neighbor of x' or 'get the next neighbor of x'. These operations take constant time if an adjacency list is used, but can take time proportional to n if you are only given an adjacency matrix as input. Thus, most linear time solvable problems on graphs, such as asking whether an input graph is bipartite, connected, acyclic, ..., require $\Omega(n^2)$ time if input is in adjacency matrix form (exercise 1.4).

We mention in passing a well known theorem about time required to test properties if the input is in adjacency matrix form. A graph property is nontrivial if the answer may be either yes or no, and monotone if adding edges to a graph which has the property always produces a graph which has the property. Thus, for example, connectivity is a monotone property, while properties such as being disconnected or acyclic are not. Any nontrivial monotone property requires $\Omega(n^2)$ time to test if the input is given in adjacency matrix form [409].

Before dealing with space complexity of the representations, we should let the reader be aware of a well known (to people in the field) trick in graph representation which allows some problems to be solved in linear time. To the best of my knowledge, this approach is first used in problem 2.12 of [11]. One might think that any algorithm which uses adjacency matrices requires $\Omega(n^2)$ time, since the matrix takes that much time to initialize. Exercise 1.2 shows that it is possible to use the adjacency matrix effectively to test adjacency in constant time without paying initialization costs. Exercise 1.3 gives a good example of the use of this representation; it is easy to develop a linear time algorithm to recognize series-parallel

[1] A straw man is a rhetorical construction, in which you create a weak counter-argument to your viewpoint, and proceed to knock it down.

graphs if you know the representation trick, but if you do not know this technique it is quite difficult. In fact, several journal papers and theses have been written recognizing the class in linear time; these algorithms manage to solve the problem using $O(m+n)$ space, but it is quite tricky to do so!

Now let us address space complexity of adjacency matrices and adjacency lists. Adjacency matrices use the same amount of space for every graph. Although there are slight variants which are useful for some applications, such as storing only elements above the diagonal for undirected graphs, there are clearly $\Theta(n^2)$ entries in the matrix, and each entry is a single bit, so $\Theta(n^2)$ bits are used to store the graph.

Adjacency lists will have a total of $2m$ entries for an undirected graph, since each edge appears on two lists. Each entry holds a number in the range $1..n$, and thus takes $\log n$ bits, making the total space complexity $\Theta(m \log n)$ bits.

The number of graphs on n vertices is $2^{n(n-1)/2}$, since there are $n(n\text{-}1)/2$ possible edges and we can choose to include or exclude each edge. Thus, any form of representation for the class of all graphs must use at least $n(n\text{-}1)/2$ bits; this result holds for both the worst case and the average case. Since every form of representation must use $\Omega(n^2)$ bits, and adjacency matrices use $O(n^2)$ space, adjacency matrices come within a constant factor of the minimum possible amount of space. For the purposes of this book, if a set has $2^{\Theta(f(n))}$ elements, any representation of members of the set which uses $O(f(n))$ bits will be called space optimal; adjacency matrices are a space optimal representation of graphs. On the other hand, adjacency lists are not space optimal; they use $\Theta(n^2 \log n)$ bits on complete graphs, since every entry uses $\log n$ bits. In fact, if all of the $2^{n(n-1)/2}$ graphs on n vertices are considered equally likely, the average number of edges in a graph is $n(n\text{-}1)/4$, and adjacency lists use $\Theta(n^2 \log n)$ bits in the average case as well. Thus, while adjacency lists save space on sparse graphs, in our formal sense they use more space than adjacency matrices.

Note that when we counted the number of graphs above, we did not deal with the issue of counting isomorphic copies of the same graph multiple times. Technically, this is unnecessary, since by definition we are discussing labeled graphs; thus isomorphic copies are considered to be different graphs. This argument, like many arguments in this book, can be modified to apply to unlabeled graphs with only minor changes. There are at most $n!$ isomorphic copies of an unlabeled graph; thus, the number of unlabeled graphs on n vertices is at least $2^{n(n-1)/2}/n!$, which is $2^{\Omega(n^2)}$. In general, as long as a graph class has at least $2^{(1+\epsilon)nlogn}$ members, a space optimal representation for a class of labeled graphs is also space optimal for the corresponding class of unlabeled graphs.

Many modifications of general adjacency list or adjacency matrix representation have been proposed. Dahlhaus, Gustedt, and McConnell [146] have introduced the notion of a 'partially complemented' representation of a graph, which can save space for representing many graphs. In this form of representation, there is a list associated with each vertex, and a special bit which indicates whether this list is of the neighbors of the vertex or the nonneighbors of the vertex. They use this to show that many operations on a graph can be solved in time which is linear in the partially complemented representation. For example, the fact that you can perform depth first search and breadth first search in linear time with respect to the partially complemented representation means that you can determine whether

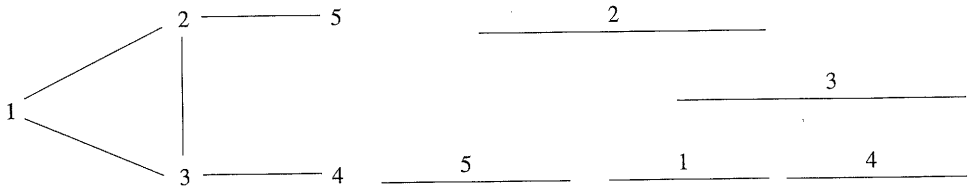


Figure 1.1 An interval graph and its model

the complement of a graph is connected in $O(m+n)$ time. Similarly, one can tell whether the complement is chordal, interval, and many other properties in linear time using this strategy. There is no problem which can be done in linear time in the size of the adjacency lists, and is known to require superlinear time on the partially complemented lists; recall that linear time means in the size of both the input and the output (so that marking all edges of G is not considered to be superlinear with respect to the partially complemented list).

This book examines many aspects of representations of graphs. There are many possible qualities we might look for in a representation, such as space optimality, speed for adjacency testing, ease of finding the representation, help with solving optimization problems, correct modeling of a type of problem, and others. We will formalize some of these notions, and examine aspects of graph representation for many graph classes which have been studied in the literature.

1.4 Example of a Nice Representation

We introduce a well known class of graphs called interval graphs , which have played a key role in defining this field. Interval graphs are particularly satisfying in that they have an important real application, nice (but not obvious) mathematical characterizations, and are defined by a simple geometric rule. For a more complete look at all aspects of interval graphs, see [194].

In an interval graph, vertices correspond to intervals on the real line, and two vertices are adjacent if and only if the corresponding intervals have a nonempty intersection. An example of a graph and its interval model is given in Figure 1.1.

Benzer's prize[2] winning work on arrangement of genes in the chromosome is directly related to interval graphs; for more on the application, see [32]. He speculated that genes were arranged in a simple linear fashion. Mutations would correspond to a damaged segment of the chromosome; this damaged segment would be an interval on the chromosome. Each mutation could damage a different set of genes. At that time, the only information that could be gathered was the set of deformities caused by a mutation. A graph can be formed from this data, by making each mutation into a vertex and adding an edge between two vertices if the mutations share a common deformity. If Benzer was correct in his supposition that genes were arranged in linear fashion, this graph will be an interval graph. He found that the graphs were interval graphs, and this was considered to be important evidence supporting his theory.

[2]It is an interesting academic urban legend that Benzer won the Nobel prize for this work. Although he won the prestigious Lasker and Crafoord prizes, he did not get a Nobel prize. In fact, this may be a double urban legend, since I suspect that we all give Benzer a Nobel prize in order to tell our students about the urban legend of why there is no Nobel prize in mathematics.

This led to a great deal of work on interval graphs and variants of interval graphs, some of which we will look at in chapter 3. For now, let us think of how we would represent an interval graph on the computer. Clearly, we could use adjacency matrices or adjacency lists. This does not seem to capture any special properties of interval graphs. For example, since a clique of size n is an interval graph, an adjacency list representation will not be space efficient. However, the interval model itself gives us a natural form of representation.

Label the endpoints of intervals in increasing order of their appearance on the line. This assigns to every vertex two integers in the range $1..2n$. For each vertex v, we store the labels of the endpoints of v; this uses $O(\log n)$ bits per vertex. Suppose that we want to test whether x and y are adjacent. It is easy to determine this simply from the labels stored at x and y. Thus, we can completely reconstruct the graph from the $O(\log n)$ bits at each vertex, and interval graphs can be stored in $O(n \log n)$ space. Testing whether x and y are adjacent also takes constant time with this form of representation, so we have managed to combine a space efficient representation and fast adjacency testing for this class of graphs. The generalization of this type of representation to other classes of graphs is the subject of chapter 2.

This raises a number of other problems, which are discussed in various chapters. Suppose that we are given an interval graph, but are not given the interval representation (as in, for example, Benzer's problem); can we find this representation efficiently? It is natural to think of letting vertices correspond to other geometric objects; how does this change the answer above? What can we do efficiently on this special class of graphs? How else can we represent a graph class with a small amount of space, and still test adjacency efficiently? This book looks both at the formalizations of these questions, and specific classes of graphs with interesting forms of representation.

Specifically, we try to provide one formal model of efficient representation in chapter 2, and make the questions we have about a class more precise. Chapter 3 surveys some of the well known classes of graphs defined by variants on the interval model above, as well as presenting some general work on intersection models. Chapters 4-12 examine classes of graphs which pose especially interesting problems with respect to finding good representations. Separate chapters are devoted to the recognition problem (chapter 13) on a large number of these graph classes, and a new view of optimization problems which comes from focusing on issues of representation (chapter 14). The final chapters deal with problems involving constructing a representation (chapter 15) and applications (chapter 16).

1.5 Overview of Problems in Graph Representation

No book can hope to deal with all aspects of the classes of graphs we discuss in this book. For example, this book will ignore problems dealing with extremal graphs for particular classes, and special theorems for the particular classes of graphs. We will focus on the following problems which are particularly tied to the problem of representing the graphs efficiently. These problems are closely related to each other, so it is not always possible to say we are dealing with each problem in a separate section, but it is helpful to remember that each problem on a class of graphs is potentially different.

1) Counting

If we want a good representation of a graph class, we need to know the number of graphs in the class. For the purposes of this book, we are satisfied with a result that says the number of graphs in the class is $2^{\Theta(f(n))}$, which implies that we can hope for a representation which uses O(f(n)) bits. Although the counting problem is difficult in general, in Theorem 8.3 we see a general method for determining whether a hereditary class of graphs has $2^{\Theta(n^2)}$ members.

2) Representation

Representation is the primary focus of this book. In general, we like representations which are asymptotically space optimal and allow fast adjacency testing; we will also introduce a specific model in chapter 2 called a generalized implicit representation. For some classes, such as chordal bipartite graphs and visibility graphs, for which no representation is known which meets these criteria, we discuss forms of representations which come close to these goals.

We will call the representation problem the question of whether a particular form of representation exists for graphs in the class. This is distinct from the question of whether such a representation can be constructed efficiently, leading to our next form of problem.

3) Constructing Representations

Some classes of graphs have efficient representations, often as a consequence of the definition of the class, but these representations may be very hard to construct. For such classes, a problem may be much easier to solve if we are given a good representation than the same problem on the same graph if the input is given in standard adjacency matrix or adjacency list form. Thus, we are interested in the question of efficient construction of a representation, as well as the question of the existence of a representation.

The construction problem is often related to the recognition problem discussed below, but this is not always the case. It is sometimes possible to construct an efficient form of representation, even when a class of graphs is difficult to recognize. This may be done by constructing a representation for a class of graphs which contains the class in question. For example, even though planar Hamiltonian graphs are difficult to recognize, it is possible to construct a very efficient representation for the more general class of planar graphs. There are many classes which are known to have efficient representations, but no fast method for constructing efficient representations is known; this topic is discussed in chapter 15.

Just as there are classes which cannot be recognized efficiently but have good representations, there are classes such that the recognition problem has a fast algorithm, but no good representation is known for the class. For example, the chordal bipartite graphs discussed in chapter 9 can be recognized efficiently, but the algorithm does not produce an efficient representation.

4) Recognition

The recognition problem for a class of graphs is the problem of determining whether an input graph is in the class. Sometimes, we can use algorithms for recognizing whether a graph is in the class directly to efficient algorithms for constructing a representation for the class. In other cases, we may suspect that an input graph is from a class which allows a good representation or can be optimized

efficiently. In such cases, we often first perform a recognition algorithm to see if the graph is in the class, and if it is in the class we use a different algorithm designed to construct a representation or solve the optimization problem correctly given the assumption that the input is in the specified class of graphs.

5) Characterization

In the characterization problem, we are looking for a representation which is possible if and only if the graph is in the class.

Open Problem 3.1 may help to make the distinction between characterization and representation more concrete. That problem deals with a class of graphs called chordal comparability graphs (definitions of the classes are contained in chapter 3, but the issue of characterization as opposed to representation does not require knowing these definitions). If a graph G is a chordal comparability graph, it is possible to construct a nice geometric representation of G. However, many graphs which are not chordal comparability graphs also can be represented using the same geometric structure; thus, the representation is not a characterization of chordal comparability graphs. This case is particularly interesting, since chordal comparability graphs can be recognized in polynomial time, while graphs which can be represented using the more general geometric model are NP-complete to recognize; thus, the representation does not capture in some fundamental sense the structure of the class of chordal comparability graphs.

6) Optimization

One reason to study special classes of graphs is that we often find that problems which are intractable on general graphs are solvable efficiently on special classes of graphs. Sometimes efficient algorithms on a class of graphs can only be found after a deep study of properties of the graphs; optimization algorithms for perfect graphs due to Grötschel, Lovász and Schrijver [237] are among the deepest results in the study of algorithms. In this book, we restrict our attention to algorithms which arise fairly directly from the representations of graph classes being studied.

For classes of graphs which have good representations, one must make the distinction between solving the optimization problem if the representation is given as input, and solving the problem if the input is the adjacency list representation of a graph which happens to be in the class. This distinction is often ignored, even in the literature of algorithms for special classes of graphs.

Suppose that the input graph is given in adjacency list form. The assumption is often made that in order to solve an optimization problem on a class of graphs which has a special representation, we must first run a recognition algorithm for the class, then construct a representation, and perform the optimization procedure on the representation. Of course, this assumption is made because there are many cases for which the representation is the key to constructing an efficient algorithm for dealing with the class of graphs. We will encounter examples in Chapter 14 where this assumption is completely invalid; there are classes of graphs which are NP-complete to recognize, but natural optimization problems can be solved in polynomial time even if the input is given in adjacency list form.

Surprisingly, the author knows of no natural problems which can be shown to require a special representation in order to find a polynomial algorithm. Thus, I give the following open problem, which is not precisely stated; I freely acknowledge that the words 'natural' and 'well-known' are subjective, but are nevertheless important

to making a solution to the problem interesting. I prefer to leave the definition imprecise rather than formal, since there are a number of 'trick methods' for finding such a problem which do not address the real issues involved. For example, the term nontrivial usually means that the answer is not always 'yes' or always 'no'; I want to use a slightly broader meaning here. This rules out such 'solutions' as asking to construct an interval dimension k representation of a graph (which is NP-hard if G is given in adjacency list form) with and without an interval dimension k representation. We will return to this open problem in the chapter on optimization.

Open Problem 1.1: Find any well-known NP-complete problem on a class of graphs with a natural efficient representation such that the problem is intractable under standard complexity theory assumptions if the input graph is in the class and the representation is not given, but is polynomial and nontrivial if the representation is given.

In some sense, an implicit representation which is the subject of chapter 2 is one method for capturing the notion that a graph class can be stored using little space, and so that adjacency testing can be done nicely. Other approaches are also possible. For example, Ian Stobert has proposed using the size of the representation times the time needed for an adjacency test as a measure of the effectiveness of a representation. Other factors, such as the time necessary for constructing the representation, could also be used as part of a measure of efficiency of a representation.

1.6 Exercises

1) Normally, adjacency lists are viewed as unsorted, i.e. the neighbors of vertex x may occur in any order on x's list. Suppose we want to arrange the lists so that neighbors of each vertex occur in ascending order on its adjacency list. Show that all adjacency lists of G can be sorted in $O(n+m)$ time.

2) Suppose that you are given the set of edges in a graph. Show that you can use $O(n^2)$ space (for this problem only, a number in the range $1..n^2$ is considered to take constant space), and $O(n+m)$ time initialization so adjacency testing is $O(1)$. Note that your problem is to distinguish whether the entry you look up is a valid entry of the matrix, or garbage left in the matrix since you did not clean it out during initialization. This is a challenging problem without a hint; if you need a hint, see problem 2.12 of [11].

3) A series reduction in a graph takes a degree two vertex y with neighbors x and z, eliminates y, and adds an edge from x to z, if that edge does not already exist. We will call a graph series-parallel if all edges can be eliminated through a sequence of series reductions and removals of degree 1 vertices. Design a linear time algorithm to test whether a graph is series-parallel. Note: the name comes from the fact that this is usually defined as a class of multigraphs which can be made edgeless using a sequence of series and parallel reductions; a series reduction would add the edge (x,z) whether or not it exists already, and a parallel reduction eliminates one of two edges between the same pair of vertices. Degree one vertices are sometimes omitted by assuming that the input graph is biconnected. For a relation between series-parallel graphs and cographs studied in chapter 9, see [475].

4) Show that if the input is given in adjacency matrix form, determining whether G is connected takes $\Omega(n^2)$ time.

5) A vertex v is a supersink in a directed graph if v has in-edges from all other vertices, and no out-edges. Show that we can determine whether G has a supersink from its adjacency matrix in $O(n)$ time; note that this is not a monotone property.

6) Show that almost all graphs are not interval graphs.

7) Show that interval graphs cannot contain chordless cycles of size greater than 3.

8a) Show how to perform depth first search on a graph given in the form of a partially complemented adjacency list (i.e., for each vertex v you are given either the neighbors of v or the nonneighbors of v) in linear time.

8b) Show how to perform breadth first search on a graph given in the form of a partially complemented adjacency list in linear time.

9*) The following problem was posed to me by Andreas Stathopolous. Suppose that, to save space, an undirected graph is stored in the following form. For every edge (x, y), instead of storing x on the adjacency list of y and y on the adjacency list of x, you store only one of these two vertices on the adjacency list of the other. Show that it is possible to determine whether G is connected in $O(n+m)$ time given input of this form, where you are only allowed to use $O(nlogn)$ extra bits of space and are not allowed to modify the space used for storing the input.

CHAPTER 2

Implicit Representation

This chapter introduces a form of graph representation which is quite general, and includes most of the representation schemes for graphs which are easy to work with. The ideas in this chapter were discovered independently by Muller [376] and Kannan, Naor and Rudich [290].

The interval graphs discussed in the previous chapter are a good starting point for understanding the appeal of implicit representation. The most obvious characteristics we want from a computer representation of a graph are that we use a small amount of space, and are able to test adjacency quickly. Consider storing the endpoints of each interval with a vertex as discussed in chapter 1; an example is given in Figure 2.1. This representation uses O(log$n$) space per vertex, and thus O($n$log$n$) space overall. Note that since any clique is an interval graph, standard adjacency matrices or adjacency lists would use $\Omega(n^2)$ space to store an interval graph on n vertices. It is also easy to test whether two vertices are adjacent using this representation in constant time. Intervals (i,j) and (k,l) are adjacent if and only if i or j is in the range k..l, or k or l is in the range i..j. Thus, we use O($n$log$n$) space to store the graph, and can test adjacency in constant time.

If we try to generalize the reason that adjacency testing of intervals is efficient, we come naturally to the notion of an implicit representation. The interval representation works because we are storing only a small number of small integers at each vertex, and adjacency between two vertices x and y can be tested using only the numbers stored at x and y. More formally, we make the following definition.

R is an implicit representation of a graph G if R assigns O(logn) bits to each vertex of G, and there is an adjacency testing algorithm which decides whether x and y are adjacent based only on the bits stored at x and y.

We note that this is still not a completely formal definition; we will delay a more technical definition of the adjacency testing mechanism until later in this chapter.

First, let us compare and contrast implicit representations with representations which are space optimal. It is easy to see (at least for the labeled case; the unlabeled case is treated in exercise 2.1) that the number of interval graphs is $2^{\Omega(nlogn)}$; this

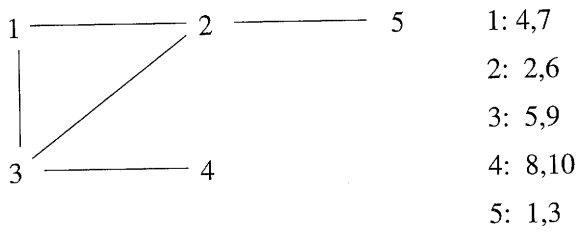


Figure 2.1 Efficient Representation of the Graph in Figure 1.1

implies that the implicit representation is space optimal. Consider graphs in which each of the vertices in the set 1,..,$n/2$ is matched with exactly one vertex in the set $n/2$+1,..,n. Each such graph is an interval graph, and there are $(n/2)!$ such graphs. Since

$$\begin{aligned}(n/2)! &= (n/2)(n/2-1)..(n/4)(n/4-1)..1 \\ &> n/2^*(n/2-1)..(n/4+1)(n/4) > (n/4)^{n/4} \\ &= 2^{(n/4)\log(n/4)}\end{aligned}$$

which is $2^{\Omega(nlogn)}$, representing all interval graphs requires $\Omega(n\log n)$ bits. We achieve $O(n\log n)$ bits with the implicit representation, so the implicit representation is (asymptotically) space optimal.

However, it is easy to invent a generic form of space optimal representation which would not be a good mechanism for storing a graph. Suppose we know that there are f(n) labeled interval graphs on n vertices. This number could be determined (in theory) by running through all possible graphs on n vertices, and testing whether each one is an interval graph; this test can be performed in $O(n+m)$ time, as we will see in chapter 13. We can place an ordering on all graphs with n vertices; for example, graphs which contain the edge (1,2) come before graphs which do not contain (1,2), then divide each set of graphs depending on whether they contain (1,3), and so forth. An interval graph could be given the number i if there are exactly i-1 interval graphs on n vertices which come before this graph in the ordering. There are f(n) interval graphs, and each interval graph is given a number between 1 and f(n); this number has log(f(n)) bits, so it is clearly a space optimal representation. It is possible to test whether two vertices are adjacent using this representation, though the natural method is unbelievably slow, since it would mean constructing all possible graphs on n vertices, and testing whether each is an interval graph, before we could deduce that we have the graph that we are seeking and can answer questions about adjacency.

In an implicit representation, all information is distributed to the vertices themselves. We need to require adjacency to be tested using only information which is 'local' to the vertices in question, to avoid representations as in the previous paragraph, in which bits of the number representing the graph are distributed to the vertices, but reconstructing the graph would still involve using bits from all of the vertices. This property of local adjacency testing is what makes adjacency tests efficient for all known forms of implicit representation, and led Muller to call this form of graph representation a 'local structure' representation.

The first possible question regarding implicit representation is whether every graph class has an implicit representation. The answer to this must be no, since there are classes of graphs which have $2^{\Theta(n^2)}$ members. Any representation of graphs in such classes must use $\Omega(n^2)$ bits for the average graph in the class, while an implicit representation uses $O(n\log n)$ bits. Examples of such classes of graphs include the class of all graphs, bipartite graphs, triangle-free graphs, asteroidal-triple-free graphs, and many more classes which will be defined later in this book.

It is also possible to show that there are graph classes with $2^{O(nlogn)}$ members which do not have implicit representations. Consider the class of graphs which contain at most $7n$ (7 could be replaced with any other constant) edges for all values of n. We will call these graphs 7-sparse. A standard adjacency list representation

uses $O(n\log n)$ bits to represent any such graph, so there are $2^{O(n\log n)}$ graphs on n vertices in this class.

Theorem 2.1 [376] *The 7-sparse graphs do not have an implicit representation.*

Proof Suppose that the 7-sparse graphs have an implicit representation. Consider the set of all graphs on n vertices; we know that there are $2^{\Theta(n^2)}$ such graphs. Let G be an arbitrary graph on n vertices. Add n^2 isolated vertices to G, and we have a graph G' which is 7-sparse. G' must have an implicit representation. Adjacencies between vertices in an implicit representation can be determined using only the bits stored at the vertices, so G can be reconstructed from the bits stored at the vertices of G within the representation of G'. There are $O(\log(n^2+n))$ bits stored at each vertex of G'; thus G can be reconstructed from a set of $O(n\log n)$ bits. This implies that there are $2^{O(n\log n)}$ graphs on n vertices, which is certainly false, so there cannot be an implicit representation of the 7-sparse graphs. □

The construction above is based on a particular property of 7-sparse graphs which some people find intuitively unsatisfying. We can take an arbitrary (non 7-sparse) graph, add vertices, and get a graph which is 7-sparse. If we think of 7-sparse graphs as a potentially 'easy' class of graphs, it seems strange that we can take a 'hard' graph, add vertices to it, and get an 'easy' graph. To formalize this concept, we need a new definition.

A class of graphs is hereditary if for every graph G in the class, every induced subgraph of G is also in the class. Interval graphs, planar graphs, and triangle-free graphs are hereditary classes of graphs; deleting vertices will not disturb the interval and planar representation, and deleting a vertex cannot make a triangle in a graph. Connected graphs and trees are not hereditary classes of graphs, but forests are a hereditary class.

Consider the problem of finding an implicit representation for uniformly c-sparse graphs, that is, graphs such that every subgraph on x vertices has no more than $cx\log n/\log x$ edges. Uniformly c-sparse graphs are contained in the c-sparse graphs, but one cannot use the construction above to embed an arbitrary graph into a uniformly c-sparse graph. In fact, it is known that a graph has an orientation in which every vertex has indegree at most k if and only if G is uniformly k-sparse [133]. This orientation leads easily to an implicit representation, and the distinction from the general class of sparse graphs shows the importance of a property being hereditary.

Kannan, Naor, and Rudich [290] made the following very interesting conjecture, which remains open.

Open Problem 2.1; Implicit Graph Conjecture: Every hereditary class of graphs which has $2^{O(n\log n)}$ graphs on n vertices has an implicit representation.

In the course of this book, we will introduce classes which are open with respect to the implicit graph conjecture, and explain the difficulties involved in representing each type of class.

Let us look at some classes which can be represented implicitly, for obvious and nonobvious reasons. An implicit representation of interval graphs follows very easily from the definition of interval graphs; the same holds true for some other well known classes of graphs which will be discussed in chapter 3, such as circular-arc graphs, permutation graphs, boxicity k graphs, dimension k posets, circle graphs,

and many other intersection graphs mentioned in books by Golumbic [223] and Brandstädt, Le, and Spinrad [74].

Let us look at some other well known classes of graphs. It is obvious that there are $2^{O(nlogn)}$ trees on n vertices, since an adjacency list representation uses $O(nlogn)$ bits. However, this does not immediately imply an implicit representation, since we cannot store all adjacencies of a vertex at that vertex, as vertices can have arbitrarily high degree. A simple implicit representation of a tree can be obtained by arbitrarily choosing a root for the tree, and storing only the parent of each vertex v at v. Two vertices v and w are adjacent if and only if v is the parent of w or w is the parent of v, so this storage technique gives an implicit representation.

Planar graphs are candidates for having an implicit representation, since it is well known that the number of edges in a planar graph is $O(n)$. An implicit representation of planar graphs comes from the property that every planar graph has a vertex of degree at most 5. We can repeatedly remove vertices of degree at most 5 from the graph, storing all remaining neighbors of the vertex with the vertex. Given any pair of vertices x and y, x and y are adjacent if and only if y is stored at x or x is stored at y.

The planar graph representation above can be generalized further. Any hereditary class of graphs with at most cn edges has an implicit representation. Suppose a graph class is guaranteed to have at most cn edges for a graph with n vertices. The total size of all adjacency lists is $2cn$, which implies there must be some vertex with at most $2c$ neighbors. We remove this vertex, storing all neighbors. Since the class is hereditary, the remaining graph has at most $c(n\text{-}1)$ edges, the total size of adjacency lists is at most $2c(n\text{-}1)$, and there is again a vertex with at most $2c$ neighbors. Repeating this process until all vertices are removed gives an implicit representation with at most $2c$ integers stored at each vertex.

Of course, many classes of graphs which can have large number of edges, such as interval graphs, also have implicit representations. Not all implicit representations are obvious from the definition. Consider, for example, the class of distance hereditary, or completely separable graphs, first studied in [27].

Distance hereditary graphs are graphs in which all induced paths between pairs of vertices have the same length. It is not clear that the number of such graphs is $2^{O(nlogn)}$, so that an implicit representation is even possible. Bandelt and Mulder showed [27] that distance hereditary graphs could be built from a single vertex by repeatedly adding either a 'pendant vertex' adjacent to exactly one old vertex, or a 'twin vertex' of another vertex; twins have the same adjacencies to the old vertices, and may or may not be adjacent to each other.

This characterization shows that the number of distance hereditary graphs is $2^{O(nlogn)}$; the graph can be reconstructed from the sequence, and each part of the sequence can be stored using $O(logn)$ bits, since there is only one number in the range $1..n$ to store. It is not possible to determine whether x and y are adjacent given only information stored with x and y; for example, if x is a twin of a and y is a twin of b, we need to know whether a and b are adjacent. Since distance hereditary graphs are a hereditary class of graphs, we might wonder whether they contradict the implicit graph conjecture. However, we will find that the distance hereditary graphs are a subset of the circle graphs, which have an implicit representation as a consequence of their definition, so the distance hereditary graphs satisfy the implicit graph conjecture.

Before continuing with new material, we give one more example of a class of graphs which satisfies the implicit graph conjecture for nonobvious reasons, to show that implicit representations do not always come directly from the definition of a graph class. The intersection graphs of paths in a tree, which are discussed at greater length in chapter 5, are defined as follows. A graph G is in the class if there exists a tree T such that vertices of G can be mapped to paths in T, such that the 2 paths in T share a common vertex if and only if the corresponding vertices are adjacent. In exercise 2.7, we see that if there is such a tree, then there is a tree with $O(n)$ vertices. This implies that the number of graphs in the class is $2^{O(nlogn)}$, since the graph can be reconstructed from an n vertex tree and the 2 endpoints corresponding to each vertex of the graph. However, this does not give an implicit representation, since global information (the tree) is necessary to determine adjacencies. We leave the construction of an implicit representation for this class of graphs to exercise 2.8.

Other ways to define graph classes which do not give immediate implicit representations include graph classes defined by forbidden subgraphs or by taking intersections of known classes of graphs. For example, if we forbid triangles as induced subgraphs, we still have $2^{\Theta(n^2)}$ graphs in the class, and cannot have an implicit representation, while if we forbid induced paths on 4 vertices there is an implicit representation, although this is not obvious from the definition. If we take well known classes of graphs with many members and define a new class by intersection, we may or may not get a class which is small enough to have an implicit representation. For example, both chordal graphs and comparability graphs (these classes will be defined in chapter 3) contain $2^{\Theta(n^2)}$ members, but there are $2^{\Theta(n^2)}$ graphs which are both chordal and co-chordal (such graphs are called split graphs), while there are only $2^{O(nlogn)}$ graphs which are both comparability and co-comparability graphs. The latter class is equivalent to a geometrically defined class of graphs called permutation graphs, and has an implicit representation, though this is not obvious when the class is defined by intersection of the two larger classes. Classes of graphs defined by intersection of larger classes and by forbidden subgraphs are studied in chapters 7 and 8.

2.1 Implicit Representation and Universal Graphs

Although implicit representation is a fairly new concept, there is a connection between implicit classes of graphs and the older field of universal graphs with respect to a class. In order to clarify this connection, we need to give a more formal definition of an implicit representation.

In our earlier discussion, we stated the requirement of a test for determining whether 2 vertices are adjacent, using only bits stored at the 2 vertices. Formally, we require that there is a Turing machine M which takes 2 binary strings B_1 and B_2 as input, and produces either yes or no as output. We interpret this to mean that vertices with bit strings B_1, B_2 stored at x and y are adjacent iff M answers yes on input B_1, B_2.

Kannan, Naor, and Rudich (but not Muller) add the restriction that M is a polynomial time Turing machine. This has the desirable property of guaranteeing that an implicit representation is not only efficient with respect to space complexity, but that adjacency testing is also fast. Since there are $O(\log n)$ bits stored at x and y, and M is polynomial, the adjacency test is at worst polylogarithmic. When we

generalize the notion of an implicit representation in the next section, polynomial time Turing machines may not necessarily give fast adjacency testing, so it is not clear whether to retain this restriction or whether we should allow M to be any recursive Turing machine. Since the issue does not arise on any known class of graphs, we will retain the polynomial restriction for our adjacency test, though it is not known whether this affects which classes of graphs are representable. Muller also considers the possibility of inputs which do not correspond to valid labels of vertices, and what the Turing machine is allowed to do on invalid pairs of inputs.

Let S be a set of graphs. G is vertex-induced universal if every graph in S is an induced subgraph of G. Let C_n be the set of graphs in the class C which contain at most n vertices. Kannan, Naor, and Rudich [290] showed that if a class C has an implicit representation, then for every n there is a vertex-induced universal graph for C_n which contains $O(n^k)$ vertices. Muller [376] adds conditions so that a class has a variant of an implicit representation if and only if the universal graphs fulfill certain conditions.

The universal graph U can be created as follows. Each bit string corresponding to a vertex in an implicit representation of a graph in C_n contains $O(\log n)$ bits; that is, at most $c\log n$ bits for some value of c. The vertices of U correspond to n copies of all possible bit strings of size at most $c\log n$; there are at most $n2^{c\log n} = n^{c+1}$ such vertices. Add an edge between two vertices x,y in U if and only if the Turing machine answers yes when given the bit strings corresponding to x and y as input. This graph can be constructed in polynomial time.

Let G be a member of C_n, and let R be an implicit representation of G. To get G as an induced subgraph of U, label vertices of G from 1 to n and select for vertex i of G the ith copy of the bit string corresponding to i in R. Each vertex x of G is mapped to a vertex with the same bit string x has in R, and vertices in U are adjacent iff vertices with these bit strings are adjacent in R, so G is an induced subgraph of U.

We note that the proof of [290] reduces the number of vertices in U to n^c by assuming that every vertex is assigned a unique bit string. We leave this problem to exercise 2.13.

2.2 Generalized Implicit Representation

By definition, implicit representations are only possible for classes of graphs which have $2^{O(n\log n)}$ members on n vertices. This restriction was probably made to let implicit representation be closely tied to other 'good' representations; almost all known implicit representations also allow constant time adjacency testing. However, there is nothing which prevents us from generalizing the notion of implicit representation to larger classes of graphs, keeping the property that our representation should be asymptotically space optimal, and that adjacency testing can be performed using only information stored at the vertices being tested.

Let C be a class of graphs. Suppose that C has $2^{f(n)}$ members with n vertices. We want our representation to be asymptotically space optimal, so it should use $O(f(n))$ space overall. Distributing information to the vertices means that we allow $O(f(n)/n)$ bits of information to be stored at each vertex. Local adjacency testing means that adjacencies between x and y can be tested using only the bits stored at x and y. This form of representation is called a generalized implicit representation. A more general form of the implicit graph conjecture remains open.

Open Problem 2.2: Generalized Implicit Graph Question Does every hereditary class of graphs have a generalized implicit representation?

For example, let us consider the class of all graphs; the same argument holds for any class of graphs with $2^{\Omega(n^2)}$ members. We are allowed to use $O(n^2)$ bits overall to represent such graphs, distributing $O(n)$ bits to each vertex. A simple generalized implicit representation can be obtained by storing with each vertex the corresponding row of its adjacency matrix.

We note that storing the adjacency list of each vertex rather than the row of the adjacency matrix will not give a generalized implicit representation. An entry of the adjacency list uses as many as $\log n$ bits, and storing, for example, the complete graph uses $\Theta(n^2 \log n)$ bits if adjacency lists are used.

By using generalized implicit representations, we will be able to consider larger classes of graphs. Many parameters used to characterize classes of intersection extend naturally to arbitrarily high dimension, as we shall see in chapter 3. We will deal with open problems on well known classes of graphs and matrices which have more than $2^{O(nlogn)}$ members in chapters 9 and 10. For some other classes, the number of graphs in the class is unknown, but we would hope for a generalized implicit representation even if the number of graphs proves to be larger than $2^{O(nlogn)}$.

A natural problem related to generalized implicit representation is studied in [452]. This paper attempts to find a representation of an arbitrary graph using a number of bits based on the degree of v, such that adjacency can be tested in constant time using only the bits stored at the corresponding vertices. The paper is designed to create such a representation using $O(\text{degree}(v) log^2 n)$ bits for each vertex v. Note that it is possible to use specially devised hash functions which allow constant time adjacency testing and require small space, but these are somewhat unsatisfactory. Some technical reasons for preferring their scheme are presented in [452], but my feeling is that the fundamental objections are not technical, but based on the fact that the hash functions are in some sense too disassociated from any intuitive understanding of the graph. The question of finding a natural representation which uses the space optimal $O(\text{degree}(v)\log(v))$ bits per vertex, and such that adjacency can be tested in constant time using only the bits stored at the corresponding vertices, is an open problem.

2.3 Representations with Very Short Labels

The implicit graph question may seem overly ambitious. There are many ways to define graph classes with $2^{O(nlogn)}$ members but which have no obvious implicit representation, as we will see during the course of this book. What reason is there to believe that all of these classes can be represented implicitly? Indeed, the implicit graph question is very much open; there is no consensus in the field that it should be true or false.

This section presents a result due to Scheinerman [423] and Scheinerman and Zito [427] which show that if we place what seems to be a small restriction on the graph class, then the class can be represented extraordinarily efficiently. Specifically, any hereditary class of graphs which has at most n^{kn} members, where k is less than $1/2$, has an implicit representation which uses only a constant number of bits per vertex! Since n^{kn} is $\Theta(2^{\Theta(nlogn)})$, this implies that every hereditary class of graphs which has $2^{o(nlogn)}$ members has $2^{O(n)}$ members. We prove a slightly

weaker version of their result, showing that this is true for a smaller value of k; every hereditary graph class C such that C has at most $(n/2)!$ graphs on n vertices must have a representation which uses a constant number of bits per vertex.

To prove this, the authors introduce a relationship between vertices which we will call a *twin relationship*. Formally, two vertices v,w are twins if $N(v) - w$ is equal to $N(w) - v$. The reader familiar with the concept of modules or homogeneous sets, which will be defined in later chapters, may recognize twins as vertex pairs which form a module of size 2. The twin relationship [1] is an equivalence relation on the set of vertices (exercise 2.17), and this equivalence relation partitions the vertices into sets, each of which must form either an independent set or a clique. The key result is that if the number of graphs in the hereditary class is small, then the number of equivalence classes is bounded by a constant.

For this result, we need to introduce a basic result from Ramsey theory. For every n, there is a value N such that every graph on N vertices contains either an independent set of size n, or a clique of size n. This result is not trivial, so readers unfamiliar with Ramsey theory must take this result on faith; for a good book on Ramsey theory, see [235].

Theorem 2.2 [423] *Let C be a hereditary class of graphs such that for all $n > k$ for some constant k, the number of n vertex graphs in C is smaller than $(n/2)!$. Then C has an implicit representation using a constant number of bits per vertex.*

Proof The proof divides into two cases. If the maximum number of equivalence classes of the twin relationship for graphs in C is unbounded, we will show that the number of graphs in the class grows too quickly. If there is a constant c such that the number of equivalence classes of the twin relationship for graphs in c is never larger than c, we show graphs in C have an implicit representation using a constant number of bits per vertex.

Suppose that the number of equivalence classes of the twin relationship for a class of graphs is unbounded. For any n, there is some graph G which has at least N vertices which are not twins of each other, where N is the Ramsey theoretic bound discussed above. Let S be a set with one vertex from each equivalence class of G. The graph induced by S must contain a clique or independent set of size at least n; let C be a clique or independent set of size $n/2$ in the subgraph of G induced by S.

Choose a minimal set of vertices Y from G such that for any pair of vertices u,v in S, there is a vertex in Y adjacent to exactly one of u and v. It is not hard to see that Y has size smaller than $n/2$ (exercise 2.18). Add vertices arbitrarily to S and Y to get a set of n vertices, and consider the subgraph F induced by this set. F has at least $n/2$ equivalence classes with respect to the twin relationship, and thus at least $(n/2)!$ different labelings of the vertices. This can be seen by fixing all labels except those in S, and any of the $(n/2)!$ labelings of S produces a different labeled graph, since there is some vertex of a fixed label which distinguishes each

[1] It is interesting to note that Ed Scheinerman, who wrote about the twin relationship in this paper, regards me as his 'evil twin'. This is because Ed and I both did graduate work on special classes of intersection graphs at Princeton at the same time; however, since he was in math and I was in computer science, we did not know of each other's existence until we were almost finished with our degrees. Our work has continued along similar paths, with mine stressing algorithms for intersection graph classes, and his dealing with nonalgorithmic issues on similar classes.

pair of vertices. Since the class of graphs is hereditary, F must be a graph in the class, and thus all relabelings of F produce a graph in the class.

Note that S is chosen to be a clique or independent set in the proof so that vertices of S cannot distinguish other vertices in S; this restriction is not strictly necessary, but may make the proof easier for some readers to see.

Therefore, if the number of equivalence classes of the twin relationship for a class is unbounded, then for every sufficiently large value of n we have at least $(n/2)!$ different labeled graphs in the class.

Thus, if the number of graphs in C is smaller than $(n/2)!$, the number of equivalence classes for graphs in C is bounded above by a constant c. We did not argue that all graphs in C have the same (that is, isomorphic) set of equivalence classes. Therefore, we cannot use the simplest strategy of assigning each vertex a number from 1 to c, and having our adjacency test know whether vertices in a particular pair of equivalence classes are adjacent or nonadjacent. This idea can be modified easily, however, to give a representation with a constant number of bits per vertex.

Let G be any graph in C. Let e be the number of equivalence classes of the twin relationship for G. Construct a matrix M which has a row and column for each equivalence class of the twin relationship for G, with the entry $M[i,j] = 1$ if and only if vertices in classes i and j are adjacent; the diagonal element $M[i,i] = 1$ if and only if equivalence class i is a clique. Store with each vertex v the matrix M and the number of the equivalence class of v. This form of representation uses $O(e^2+\log e)$ bits for a general graph, where e is the number of equivalence classes with respect to the twin relationship. Since e is bounded above by a constant c for the class, we store a constant number of bits per vertex, and adjacency can be tested locally. □

Theorem 2.2 seems to depend heavily on the fact that graphs are labeled, so that certain simple subgraphs correspond to an enormous number of different labeled graphs. It would be interesting to see whether the results of this section can be extended to the unlabeled case as well. For example, there is something inherently unsatisfying about using the fact that the number of labeled matching graphs (graphs in which each vertex has degree 1) is $(n/2)!$, and that thus any class which contains the matching graphs is large; nevertheless, this is completely consistent with the definition of implicit representation.

2.4 Distance Labeling of Graphs

Our goal in implicit representation is to store a graph efficiently in such a way that adjacency testing can be performed efficiently. An extension of this concept called distance labeling has been studied in several recent papers [396, 210]. In distance labeling, one must store information at each vertex so that the distance between any pair of vertices can be computed using only the bits stored at the vertex. It is easy to determine adjacency from a distance labeling (vertices are adjacent if and only if distance$(x,y) = 1$), but more bits are often necessary for distance labeling than adjacency testing. Distance labeling has been suggested as being useful for local message routing in distributed systems.

It is clear that we can find a distance labeling of a graph which uses $O(n\log n)$ bits per vertex, by storing the distance to every other vertex with each vertex v.

Since there are $2^{\Omega(n^2)}$ graphs on n vertices and G can be computed from a distance labeling, any distance labeling must use $\Omega(n^2)$ bits overall, and thus $\Omega(n)$ bits per vertex. It is not obvious that distance labeling can be done for general graphs with $O(n)$ bits per vertex; storing the row of the adjacency matrix with a vertex does not allow us to determine distance between nonadjacent vertices u and v using only the bits stored at u and v. Nevertheless, we will see below that it is possible to find a distance labeling of an arbitrary graph using $O(n)$ bits per vertex.

We state without proof here some of the key theorems of the first papers; since the field is still developing, there are likely to be new results which have been obtained since this book was written. Note, for example, that the bounds for distance labeling of planar graphs are not tight at this point in time.

Theorem 2.3 [396, 294, 210] *The class of all graphs can be distance labeled using $O(n)$ bits per vertex. Trees, permutation graphs, and interval graphs can be distance labeled using $O(log^2 n)$ bits per vertex. Planar graphs can be distance labeled using $O(n^{\cdot 5} logn)$ bits per vertex. Trees require $\Omega(log^2 n)$ bits per vertex for any distance labeling. Degree 3 graphs require $\Omega(n^{1/2})$ bits in any distance labeling. Planar degree 4 graphs require $\Omega(n^{1/3})$ bits per vertex in any distance labeling.*

We now present an ingenious method for finding an $O(n)$ bit per vertex distance labeling of a graph. Details of the labeling and proof of correctness can be found in [210]. We note that there were other $O(n)$ bit distance labelings of graphs known previously. Winkler [487] showed that there is a labeling of vertices in an arbitrary graph with a vector of length n-1, each entry of which is 0, 1, or *, such that the distance between x and y is the number of positions in which one vertex has a 0 while the other has a 1. The labeling scheme described below is the first $O(n)$ bit per vertex labeling scheme which uses $o(n)$ time to reconstruct the distances from the distance labelings of two vertices.

Theorem 2.4 [210] *There is an $O(n)$ bit per vertex distance labeling scheme for arbitrary graphs.*

Proof It is easy to take an arbitrary connected graph G and find for any given i a set S of $n/2^i$ vertices such that every vertex is at distance at most 2^i from some vertex in S (exercise 2.21). Let us assume that we are given a collection of these sets S_i for each value of i from 0 to log$n$; S_0 consists of all vertices of the graph.

Create a tree T from the sets S_i by letting the parent of s_i in S_i be a vertex of S_{i+1} of minimum distance from s_i in G.

Consider two nodes u_i, v_i from S_i with parents u_{i+1}, v_{i+i}. Suppose that we know the distance D between u_{i+1} and v_{i+1} in G. The distance between u_i and v_i is at most $D + 2^{i+2}$, and at least $D - 2^{i+2}$, since u_i and v_i have distance at most 2^{i+1} from u_{i+1} and v_{i+1} respectively. This leads to a new idea for storing distance between u_i and v_i; we store for each such pair the distance between u_i and v_i mod 2^{i+3}.

The label of a node of the tree at level i corresponds to the concatenation of the label of its parent, the position of its parent in the previous level, the position of the node in its level, and the distance to each other node in level i mod 2^{i+3}. We will argue that it is possible for each pair of nodes u_i, v_i to deduce the distance between u_i and v_i given only the labels of these nodes.

Clearly, we can deduce the distance between all nodes of S_{logn} of the tree, since we have only 1 node at this level. Assume that we can deduce the distance between

nodes of S_i from their labels; we will give a procedure for deducing distance between nodes of S_{i-1}.

Let u_{i-1}, v_{i-1} be nodes of S_{i-1}. Since the label of a node contains the label of its parent, we may assume that we can deduce the distance D between the parents of u_{i-1} and v_{i-1}. The label also contains the distance between u_{i-1} and v_{i-1} mod 2^{i+3}; as shown above, this allows us to deduce the distance between u_{i-1} and v_{i-1}.

We now argue that nodes of S_0 are labeled with $O(n)$ bits; since all vertices of G are in S_0 and we can deduce distances from pairs of labels, this means that we have an $O(n)$ bit per vertex distance labeling of G.

The root of T is labeled with a constant number of bits; since the label of each node includes the label of its parent, we only need to consider the total number of bits added to labels along a root to leaf path of the tree. At each level, we add the position of the child in its level and the position of the parent in its level. This adds at most log$n$ bits for each of the log$n$ levels of the tree; the total number of bits added in this way is $O(log^2 n)$. In addition, for a level corresponding to S_i, we add the distance mod 2^{i+3} to each of the 2^i other vertices on the level; each distance is represented by a string of $i+3$ bits. Thus, for a node in S_0, another $3n$ bits are added to the label, for a node in S_1 another $4n/2$ bits are added, and so on. This can easily be shown to sum to $O(n)$. Note that each term in this sum is less than or equal to the corresponding term in $3(n + 2(n/2) + 3(n/4) + 4(n/8) + ... + logn(n/n))$, and the terms inside the parentheses are the sum of the heights of nodes in a complete binary tree; most algorithms texts show this to be $O(n)$ as part of the analysis of the time used to build a heap.

Therefore, this labeling scheme gives a correct distance labeling of an arbitrary graph using $O(n)$ bits per vertex. □

2.5 Exercises

1) Show that the number of unlabeled interval graphs is $2^{\Omega(nlogn)}$.

2) The line graph of G is a graph G' such that vertices of G' correspond to edges of G, with two vertices being adjacent in G' iff the corresponding edges share a common endpoint. Show that line graphs have an implicit representation.

3) A caterpillar is a graph in which the vertices of degree greater than 1 form a single induced path. Show that caterpillars have an implicit representation.

4) A k-tree is defined as follows. A clique of size k is a k-tree, and if G is a k-tree then the graph formed by making a new vertex adjacent exactly to some k-clique in G is a k-tree. A partial k-tree is a graph which can be obtained from a k-tree by deleting 0 or more edges. Show that for all k, the class of partial k-trees has an implicit representation.

5) A poset is N-free if there is no set of 4 elements a, b, c, d which induce an 'N' in the Hasse diagram (transitive reduction); i.e. there are no set of 4 vertices a, b, c, d such that the edges $(a, b), (c, b), (c, d)$ are the only edges present between these vertices in the transitive reduction.

The following characterization of N-free posets comes from [242]. Let P_1, P_2 be two posets, A a subset of sources of P_1 and B a subset of sinks of P_2. A new poset is created by a 'quasi-series-parallel' operation, keeping adjacencies between vertices from the same poset, and adding edges from each vertex in B to each vertex in A (as well as those implied by transitivity from this addition). Habib and Jegou show that a poset is N-free if and only if it can be constructed from single vertex posets using quasi-series-parallel operations.

a) Use the characterization above to show that the number of N-free posets on n vertices is $2^{O(nlogn)}$

b) Show that the set of N-free posets does not have an implicit representation. Hint: any poset can be transformed into an N-free poset by introducing a new vertex inside every edge of the original poset.

6) Let C be the class of graphs which are the transitive closure of rooted trees. In other words, a member of C corresponds to a tree with edges added from all ancestors to all descendants. Give an implicit representation for this class of graphs.

7) Consider the intersection graphs of paths in a tree. Show that if a graph G with n vertices is the intersection graph of paths in a tree, then G is the intersection graph of paths in a tree T with $O(n)$ vertices.

8) Show that intersection graphs of paths in a tree have an implicit representation.

9) A graph is a cograph if it can be constructed using the following rules. A single vertex is a cograph. If $G_1 = (V_1,E_1)$ and $G_2 = (V_2,E_2)$ are cographs then $G_3 = (V_1 \cup V_2,E_1 \cup E_2)$ and $G_4 = (V_1 \cup V_2,E_1 \cup E_2 \cup V_1 \times V_2)$ are cographs. Show that the number of cographs is $2^{O(nlogn)}$.

10*) Show that every cograph is both a comparability graph and a co-comparability graph (see chapter 3 for the definitions of these classes); this implies using results from chapter 3 that cographs have an implicit representation.

11) The total graph $T(G)$ of G has a vertex for each vertex and edge of G. Two vertices of $T(G)$ which correspond to vertices of G are adjacent iff the vertices are adjacent in G, and vertices which correspond to edges of G are adjacent iff they share a common endpoint (as in the line graph). If v_t is a vertex of $T(G)$ corresponding to a vertex v of G and v_e is a vertex of $T(G)$ corresponding to edge e of G, then v_t is adjacent to v_e iff v is an endpoint of e. Show that the total graphs do not have an implicit representation.

12*) An autograph is a graph which can be derived from a set of n positive integers by making a vertex for each integer, and letting i and j be adjacent iff $|i - j|$ is also in the set. Show that the autographs do not have an implicit representation.

13) Show how to modify any implicit representation so that no pair of vertices receives the same label.

14) Show that comparability graphs, chordal graphs, and chordal co-chordal (also known as split) graphs all have $2^{\Omega(n^2)}$ members. What does this imply about the existence of implicit and generalized implicit representations for these classes?

15*) Let C_k be the class of graphs defined by forbidding paths of length k as induced subgraphs. Can you determine the first value of k such that C_k does not have an implicit representation?

16*) A graph is c-decomposable if for every induced subgraph H there is a set S of at most c vertices such that no component of G - S contains at most 2/3 of the vertices in H. Show that for any value of c, every c-decomposable graph has an implicit representation.

Note: Many of the exercises above come from examples in [376].

17) Show that the 'equivalent neighborhood' relationship is an equivalence relationship, and thus partitions the vertex set of a graph. Show how to partition the vertex set into these equivalence classes in $O(n+m)$ time.

18) Let G be an arbitrary graph, and let U be a set of vertices such that no pair of vertices in U are twins. Show that there is a set S of at most $|U|$-1 vertices such that any pair of vertices in U is distinguished by a vertex in S.

19) Show that every graph with at least 6 vertices contains either an independent set of size 3 or a clique of size 3, and that this is not true for some 5 vertex graph.

20) In this book, we generally deal only with labeled graphs; this largely avoids the difficulties of dealing with problems relating to the number of labelings of isomorphic copies of the same graph. Show that there is a constant c such that if a graph class contains at least 2^{cnlogn} members on n vertices, a graph class has $2^{O(f(n))}$ labeled members if and only if the class has $2^{O(f(n))}$ unlabeled members.

21) Show that for any i and any connected graph G, there is a linear time algorithm which finds a set S of at most $\lceil n/i \rceil$ vertices such that every vertex in G has distance at most i from a vertex in S.

22) Design an $O(log^2 n)$ bit per vertex distance labeling scheme for trees.

23) Show that any distance labeling scheme for split graphs requires $\Omega(n)$ bits per vertex.

24) Analyze the time for computing the distance between x and y given the $O(n)$ bit distance labels described in Theorem 2.4.

CHAPTER 3

Intersection and Containment Representations

In this chapter, we discuss classes of graphs defined by simple intersection and containment models. Most of these classes have implicit representations, as an immediate consequence of their definitions. It is necessary to introduce the common classes of graphs with an implicit representation at this point, since one technique for showing a class of graphs has a good representation is to prove that they are a subset of a known class of graphs which has a good representation.

We will introduce a variety of types of graphs, and discuss techniques which apply to the classes. Recognition of many of these graph classes is studied in chapter 13. We start by introducing two graph theoretic properties which a good number of these graph classes satisfy, and allow various optimization problems to be solved efficiently.

Many of the classes defined by intersection and containment models are subclasses of a large class called perfect graphs. We review some relevant information on perfect graphs briefly at the end of the chapter.

3.1 Chordality and Transitive Orientation

A chord in a cycle is an edge which goes between two vertices which are not consecutive in the cycle. A graph G is chordal if there are no chordless cycles in G of length greater than three. Chordal graphs always contain a vertex v such that the neighborhood of v is a clique [160]; such a vertex is called a simplicial vertex. This gives rise to a 'perfect elimination scheme' $v_1\ v_2\ ..\ v_n$ in which each v_i is simplicial in the graph induced by v_i,$v_{i+1}\ ..\ v_n$. Many algorithms on chordal graphs can be solved efficiently by stepping through the graph in the order defined by the perfect elimination scheme. For example, to find the maximum independent set in a graph, we can greedily select vertices from the perfect elimination scheme if they have no previous neighbor in the independent set; in the graph of Figure 3.2, we select 1,2,3 and reject 4 and 5.

Although chordal graphs are not defined by an intersection model, they can be characterized by an important intersection model, which will be used at various points throughout the book. Chordal graphs correspond exactly to intersection graphs of subtrees of a tree [476, 216, 84]. An example is given in figure 3.1. This model is sometimes called a clique tree model, since nodes of the tree correspond to maximal cliques of G, and each vertex v corresponds to a subtree of cliques which contain v. Chapter 15 gives an algorithm for constructing the intersection model for a chordal graph.

A graph G is a comparability graph if the edges of G can be directed to produce a transitive digraph G'; in other words, whenever $x \rightarrow y \rightarrow z$ in G', there must be an edge from x to z. The directions assigned to edges are called a transitive orientation of G.

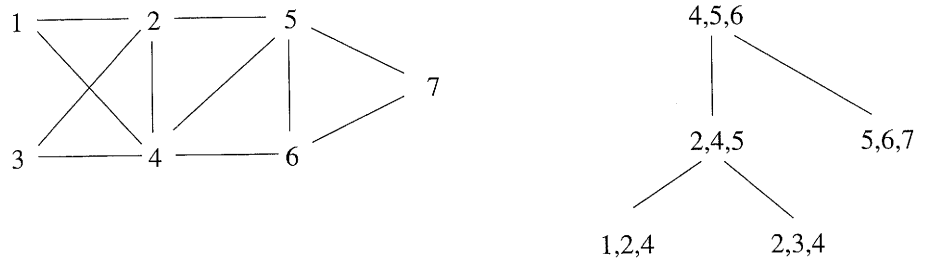


Figure 3.1 A Chordal graph and its Clique Tree Model

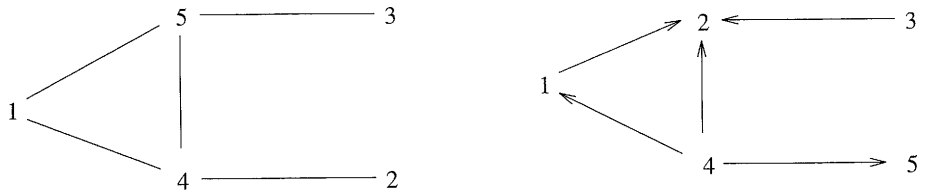


Figure 3.2 Perfect Elimination Scheme 1 2 3 4 5, and Transitive Orientation

A transitively ordered graph corresponds naturally to a partial order, so we will use some terms which come from the literature on partial orders. A linear extension of a partial order is a total order on the elements which includes every edge of the partial order; for our purposes, this is the same as a topological sort of the vertices of the transitively oriented graph.

We can solve certain problems efficiently on comparability graphs by stepping through the vertices in an order defined by any linear extension of the transitive orientation of G. To color a comparability graph optimally, we take a linear extension of the transitive orientation, and give each vertex the smallest color not used on an earlier neighbor. If we use the extension 3 4 5 1 2 in the graph of Figure 3.2, we color 3 and 4 with color 1, 5 with color 2, 1 with color 2, and 2 with color 3. This coloring has the property that if v receives color i, there is a directed path of i vertices which ends at v. Since a path in a transitive graph corresponds to a clique, a vertex receiving color i is part of a clique of size i, which implies that the coloring is optimal.

Some interesting generalizations of chordal and co-comparability graphs are studied in chapter 7. Specifically, we generalize chordal graphs to weakly chordal (also called weakly triangulated) graphs, in which chordless cycles of length four are allowed, but no longer chordless cycles are allowed in G or its complement. It has been observed that various problems seem to be more easily solvable on co-comparability graphs than on comparability graphs. Attempts to explain this phenomenon have led to the study of asteroidal triple-free graphs, which contain the co-comparability graphs. These graphs will be defined and studied in chapter 7.

Chordal graphs and comparability graphs, as well as their complements, are subclasses of perfect graphs, which are discussed at the end of this chapter. Although this implies polynomial algorithms for a number of problems using the algorithm of [237], these problems are solved much more easily on the smaller classes.

The clique and independent set problems are solvable in linear time on chordal graphs; these algorithms come from the fact that the perfect elimination scheme can be constructed in linear time, as described in chapter 13. The clique problem can be solved in linear time on comparability graphs. This comes from the fact that a transitive orientation can be found in linear time, although some interesting issues arise from the fact that comparability graphs cannot yet be recognized in linear time; see chapters 11 and 14 for details. The independent set problem can be solved on comparability graphs by a reduction to bipartite matching, and it might be difficult to improve upon this bound since improvements would lead to improvements in the time for finding the size of a maximum bipartite matching. See exercise 14.8 for discussion of the algorithm and consequences of an algorithm which was faster than bipartite matching.

We will see below that a number of interesting graph classes are defined on the basis of whether the graph and/or its complement is chordal and/or comparability graphs. Interval graphs are equivalent to graphs which are both chordal and co-comparability graphs. Permutation graphs are equivalent to graphs which are both comparability and co-comparability graphs. It is natural in this context to consider graphs which are both chordal and comparability graphs. These graphs were shown to have an implicit representation in [356] where it is shown that they were a subset of comparability graphs of posets which have dimension at most 4. However, this representation is not a characterization; many comparability graphs of 4 dimensional posets are not chordal comparability graphs. In fact, comparability graphs of 4 dimensional posets are NP-complete to recognize, while chordal comparability graphs can be recognized efficiently. To many researchers, this is intuitively unsatisfying; the representation may be good as a storage mechanism, but it does not get to the heart of what makes the class easy to deal with.

Open Problem 3.1: Find an implicit representation which characterizes chordal comparability graphs.

3.2 Techniques for Interval Graphs

Interval graphs are a particularly nice class of graphs in that many different algorithmic techniques produce efficient algorithms for interval graphs. In the following sections, we will study a variety of generalizations of interval graphs. In the more general classes, some of the techniques which lead to efficient algorithms for interval graphs will remain effective, while others will not work on the larger classes.

It is not hard to see that interval graphs are both chordal and co-comparability graphs, and thus any problems solvable for the larger classes are solvable on interval graphs. In fact, interval graphs are exactly equal to chordal co-comparability graphs [220].

Many problems which are difficult on general graphs can be solved on interval graphs using a simple greedy approach. Consider the independent set problem. It is easy to see that the interval which ends first can be included in a maximum independent set. To solve the independent set problem on interval graphs when we are given a representation, we can place this interval in our independent set, delete the vertex and its neighbors, and continue until all vertices are deleted.

A graph class defined by a geometric intersection model has the Helly property if for every clique C, there is some single point p such that every vertex of C includes

the point p. Interval graphs have the Helly property; clearly, this means that the maximum clique problem is solvable efficiently given an intersection model, since we only need to check how many vertex intervals cover each endpoint of an interval. It is not hard to see that the Helly property implies that there are $O(n)$ maximal cliques in an interval graph. There are algorithms which enumerate all maximal cliques in time polynomial with respect to the number of cliques [464]. This implies that there is a polynomial algorithm to solve the maximum clique problem on any class of graphs with a model which guarantees a polynomial number of maximal cliques, even if the model cannot itself be constructed efficiently.

Interval graphs also are amenable to other algorithmic techniques. Since the greedy technique is so effective on interval graphs, relatively few algorithms in the literature use the more complex techniques on interval graphs, but we will present them on interval graphs and discuss whether they are effective or ineffective for various generalizations of interval graphs.

Dynamic programming is also an effective technique for many problems on interval graphs and their generalizations. In dynamic programming, we build up a solution for a large problem from a table of solutions on subproblems. For this to be effective, the number of subproblems must be reasonably small (for example, we cannot solve an exponential number of subproblems), and the solution to a large problem must be able to be deduced quickly from the solutions to all the smaller problems.

Once we have the idea of using dynamic programming on interval graphs, we can bound the number of subproblems as follows. Consider the points of the real line which correspond to an endpoint of at least one interval of the interval graph, and number these from 1 to (at most) 2n on the basis of position in the line. For each pair of integers $i < j \leq 2n$, we can define a subgraph by restricting ourselves to the region between i and j. Depending on the problem, we may look either at intervals such that both endpoints occur within the interval (i,j), or intervals with at least one endpoint in (i,j). In either case, there are $O(n^2)$ subproblems to consider, and subproblems are considered in increasing order of j - i.

A relatively new algorithmic technique, which is really a type of dynamic programming on graphs, is based on minimal separators. A set S of vertices separates a from b if there is no path from a to b in G - S; a separator is minimal if removing any vertex from S leaves a path from a to b. In general, the number of minimal separators of a graph is exponential (exercise 3.22). For interval graphs, any minimal separator corresponds to the set of vertices which pass through some point in the interval model (exercise 3.23). This implies that interval graphs have $O(n)$ minimal separators, and these can be found efficiently. For general graphs, all minimal separators can be enumerated in $O(n^3R)$ time, where R is the number of minimal separators in G [304]. Some problems that can be solved efficiently using this technique are given in [312, 310]. A particularly interesting result due to Bouchitté and Todinca shows that the treewidth and minimum fill-in problems are solvable in polynomial time for all graphs with a polynomial number of minimal separators [65].

3.3 Generalizations of Interval Graphs

There is a great deal of literature on interval graphs, which have important practical applications as well as interesting mathematical structure. Since interval

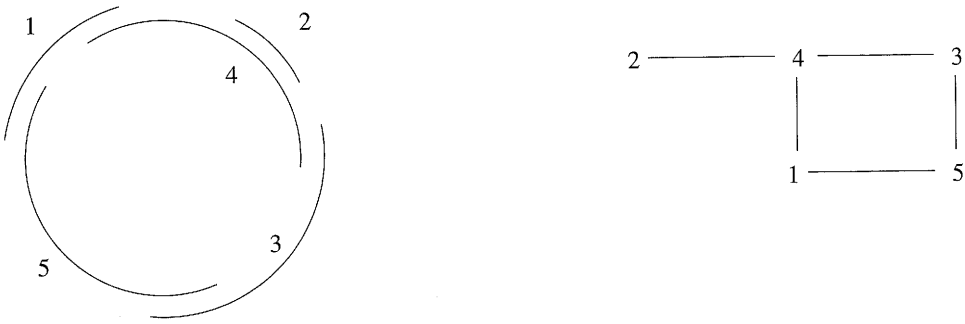


Figure 3.3 Circular-arc model for a graph

graphs have proved to be very easy to deal with, many attempts to generalize the class in different directions have been made. We will not attempt to cover all the work on generalizations of interval graphs, but we will try to familiarize the reader with some of the more popular representations which have been attempted, and introduce some open problems related to them. See [74] for more examples of such classes of graphs.

If an interval graph is viewed in a certain light, it is easy to conceive of a number of generalizations. In an interval graph, each vertex is a line segment, and these line segments are placed on the real line. We will start by considering generalizations of the real line, where each vertex will still correspond to a single segment.

3.3.1 Circular-Arc Graphs. One popular generalization of interval graphs is the class of circular-arc graphs, in which vertices correspond to arcs of a circle rather than intervals on a line; a circular-arc model is shown in figure 3.3. Circular-arc graphs are not perfect, since cycles of any length are circular-arc graphs.

Natural greedy algorithms are more complex than or not as effective for circular-arc graphs as they are for interval graphs. Coloring, which can be solved optimally by a greedy algorithm for interval graphs (exercise 3.8) is NP-complete for circular-arc graphs; however, it remains polynomial to determine whether a circle graph is k-colorable when k is fixed.

Consider the independent set problem for circular-arc graphs. If we start at a point p on the circle, and there are a number of mutually overlapping arcs which cross p, we cannot decide locally which arc to include in the maximum independent set. There are algorithms which solve this problem in $O(n)$ time given the model; these step through the model in order, maintaining the greedy independent set if we start from each candidate starting point in an efficient manner [225], [363],[271] but getting this time bound is much harder than for interval graphs.

Many problems are solved on circular-arc graphs by reducing the problem to an interval graph problem. For example, the independent set problem can be solved easily in $O(dm)$ time, where d is the minimum degree of any vertex in G, even if the model is not given. This can be done because removing all neighbors of any vertex in a circular-arc graph leaves an interval graph. If v is the minimum degree vertex, we try including v and each neighbor of v in the independent set, removing all neighbors of the vertex included, and solving the problem optimally on the remaining interval graph. Independent set on circular-arc graphs if the model is not given is also discussed in chapter 14.

The number of minimal separators in a circular-arc graph remains polynomial, since any minimal separator corresponds to the set of arcs which pass through two points of the circle (exercise 3.10). This allows problems such as treewidth and minimum fill-in [303], and vertex ranking [152] to be solved in polynomial time on circular-arc graphs. These algorithms are similar conceptually to dynamic programming on circular-arc graphs, since the natural dynamic programming approach is to deal with regions defined by cutting the circle at two points.

3.3.2 Boxicity. Boxicity is a natural way of generalizing interval graphs to higher dimension. A graph has boxicity k if it is the intersection graph of isooriented (i.e. aligned with the axes) boxes in k-dimensional space. It is NP-complete to determine whether a graph has boxicity k for $k > 1$ [492, 320]; for $k = 1$, this is simply interval graph recognition.

However, boxes in k-space still have the Helly property, so the number of maximal cliques is $O(n^k)$ for boxicity k graphs. This implies that the maximum clique problem is polynomial on boxicity k graphs; by contrast, the maximum independent set problem and chromatic number problem are already NP-complete for boxicity 2 graphs [408, 337]. Although it is not immediately obvious, every graph can be represented as the intersection graph of boxes in d-dimensional space for some d; the minimum such d is called the boxicity of G. To see that every graph can be represented, note that two boxes defined by a set of intervals in d dimensions intersect if and only if the intervals intersect in all dimensions. Therefore, we create a dimension for each vertex x. Let x correspond to a point in this dimension which is covered by all neighbors of x and uncovered by nonneighbors, and have all vertices except x intersect outside this point.

3.3.3 Disk Intersection Graphs. These graphs are the intersection graphs of circles in the plane, or more precisely of disks in the plane, since circles are considered to be adjacent if one circle completely contains another. Although disk intersection graphs can be represented by giving the centers and radii of the disks, this representation is not implicit, since there is no obvious bound on the number of bits which must be used for these values to be specified. In chapter 4, we will see a crucial theorem which allows us to count the number of disk intersection graphs. An important special case is unit disk graphs, in which all disks must have the same radius; unit disk graphs have also been called proximity graphs.

Recognition of disk graphs and unit disk graphs is NP-hard [322, 77]. Coloring, independent set, and dominating set are NP-complete even on unit-disk graphs [112]. We discuss an algorithm for the clique problem on unit disk graphs in chapter 14.

For geometrically defined classes in which the number of bits in the representation is not clear, we will talk about the problem of using small integers in the representation rather than real numbers. The question of the maximum necessary size of integers used in representing disk graph is open, and is related to the question of whether recognition is in NP, since if both the centers and radii can be mapped to integers of size at most $2^{p(n)}$ for some polynomial $p(n)$, the recognition problem is in NP. However, even if the integers must have doubly exponential size, it is possible that the disk intersection problem is in NP. For example, it may be possible to decide whether there is an arrangement of points in space which respects some ordering relationship on distance between points; this could be decidable using geometric properties which are independent of the sizes of integers in the model.

Disk representation has also been generalized to higher dimensions; a graph has sphericity k if it is the intersection graph of k dimensional spheres. Every graph has a sphericity, although this is not obvious [360].

3.3.4 Interval Number and Total Interval Number. A graph has interval number k if there is a mapping from each vertex to a set of k intervals on the line, such that two vertices are adjacent in G if and only if at least one pair of the corresponding intervals have a nonempty intersection. Recognition of interval number k graphs is NP-complete for all $k > 1$ [481].

Interval number, like boxicity, is a parameter which can be used to measure 'complexity' in graphs. Clearly every graph has an interval number (exercise 3.19). Since graphs with interval number k can be represented using $O(kn\log n)$ bits, there must be graphs with interval number at least $n/\log n$. In fact, there are graphs with interval number $\Theta(n)$ [462], though the interval number of a random graph is $\Theta(n/\log n)$ [172]. If we want to define a graph class with $2^{\Theta(f(n))}$ members for values of $f(n)$ between $n\log n$ and n^2, we can generally pick an appropriate value for one of these parameters; for example, we can define a class of graphs as having interval number at most $n^{1/2}$. This class cannot have an implicit representation, since there are too many graphs in the class, but the natural representation may be generalized implicit.

Interval number has been generalized to such parameters as rectangle number (each vertex is associated with at most k rectangles in the plane), circular-arc number, and in other similar ways. One open problem which has generally been ignored on such classes is the problem of constructing an efficient representation in polynomial time; this may be possible even if the parameter is NP-complete to compute. We will discuss such problems in more detail in chapter 15.

There are few examples of algorithms which work efficiently on interval number 2 graphs. Since they contain circular-arc graphs (exercise 3.11), chromatic number is NP-complete for the class. Since every degree 3 graph has interval number at most 2 (exercise 3.11), independent set is NP-complete, The complexity of the maximum clique seems to be open for the class. For some applications of the independent set problem in interval number 2 graphs see [289].

Open Problem 3.2: Determine the complexity of the clique problem for graphs with interval number 2.

Most of the graph classes in this chapter are defined by an implicit representation. The next graph class has many of the properties required for an implicit representation, but varies in one key way which makes finding an implicit representation impossible for the class.

Consider a representation of a graph in which each vertex corresponds to a set of intervals, with an edge between two vertices if and only if some intervals from each set have a nonempty intersection. Total interval number differs from interval number in that the number of intervals per vertex is not fixed; bounds are placed instead on the total number of intervals in the set. If we bound the total interval number by cn for some fixed c, an interval representation is space efficient, local in the sense that all information about adjacency between x and y is stored with x and y; however, the number of bits at a single vertex may be larger than $\Theta(\log n)$. In exercise 3.16, it is shown that graphs with total interval number cn do not have an implicit representation.

3.3.5 Path Graphs. We can also view an interval graph as the intersection graphs of paths in a larger path. Circular-arc graphs generalize this by being intersection graphs of paths in a cycle. We can also think of intersection graphs of paths within more complex graphs. For these classes it can make a difference (sometimes a surprisingly large difference) as to whether we define path intersection as sharing a common vertex or sharing a common edge. For example, path graphs are the intersection graphs of paths in a tree if we define intersection as meeting at a common vertex, while EPT (meaning edge intersection of paths in trees) graphs are the intersection graphs of paths on trees if we define intersection to mean sharing a common edge. Path graphs can be recognized in polynomial time (see chapter 13), while EPT recognition is NP-complete [226]. Furthermore, path graphs are chordal, implying that such problems as clique and coloring can be solved in polynomial time on these graphs, while EPT graphs are not even necessarily perfect, and the chromatic number problem is NP-complete on EPT graphs [454].

Instead of paths in trees, we can consider paths on a two dimensional grid. An i by j grid is a marking of the plane in which the only possible points to use in specifying a representation are the points (x, y), where x is an integer in the range $0..i$ and y is an integer in the range $0..j$. A grid graph has vertices for all possible pairs (x, y) as specified above, and edges from (x, y) to $(x\text{-}1,y)$, $(x+1,y)$, $(x,y\text{-}1)$, $(x,y+1)$, when these values are in the range of vertex numbers.

Intersection graphs of paths on grids and paths in planar graphs have also been considered. Intersection graphs of paths, and more generally connected subgraphs, of series-parallel graphs have applications in register allocation [291].

Intersection graphs of paths on trees can clearly be stored using a small amount of space, since paths are uniquely defined by a set of endpoints, while this is not clear for graphs defined by intersection of paths on larger classes, since specifying the path requires using a large amount of space. However, it is not clear that there is an implicit representation even for graph classes defined by paths in trees, since testing adjacency seems to require using the entire tree rather than using only the bits stored at the vertices. Implicit representations for classes defined by paths in tree are discussed in exercise 5.1. Exercise 3.18 shows that there are too many intersection graphs of paths on the grid to hope for an implicit representation.

3.4 Permutation Graphs and Generalizations

A variety of intersection graph classes are defined by making objects correspond to sets defined on two parallel lines, rather than one parallel line as in interval graphs. The simplest, and most heavily studied of these classes, is called the permutation graphs. In a permutation graph, each vertex corresponds to a line segment drawn between the two parallel lines, with vertices being adjacent iff the corresponding segments cross in the model; the model is often called a permutation diagram. An example is given in Figure 3.4. Permutation graphs are exactly equal to graphs which are both comparability and co-comparability graphs [166] [223].

Note that it is more usual to define permutation graphs in terms of a single permutation of the vertex set; this corresponds to labeling vertices from 1 to n on the basis of the order of the endpoints along one line, and specifying only the order of endpoints along the other line.

The next most popular class defined using similar structures is the class of trapezoid graphs. These are the intersection graphs of trapezoids defined by pairs

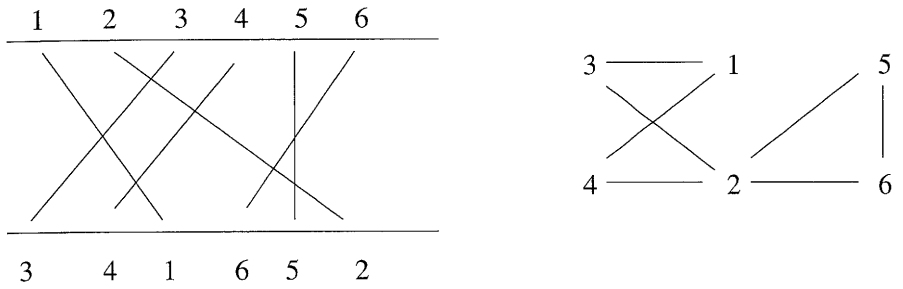


Figure 3.4 A Permutation Diagram and Permutation Graph

of intervals, one on each parallel line. Many dynamic programming and separator based algorithms on permutation graphs also work on trapezoid graphs.

Corneil and Kamula [123] defined a pair of graph classes which lie between the permutation graphs and the trapezoid graphs. In both classes, vertices correspond to triangles with all endpoints on the two parallel lines. In the first class, called PI (for point interval), all triangles have one endpoint on the top line, and two endpoints on the bottom line. In the second class, called PI*, each triangle may have one or two endpoints on the top line. Although there are polynomial time recognition algorithms for permutation graphs and trapezoid graphs, the problems of recognizing PI and PI* graphs remain open.

[188] studies a further generalization to d-trapezoid graphs. In this generalization, we use d parallel lines rather than two. Each vertex is assigned two points on each parallel line; a d- trapezoid is the object created by connecting the left (right) endpoint of each line i with the left (right) endpoint on line $i+1$, and the left and right endpoints on the first and last line.

These classes are all co-comparability graphs, since we can order nonadjacent vertices according to whether the corresponding geometric object is to the left of the other; clearly if x is to the left of y and y is to the left of z, then x is to the left of z. Many problems on these classes are solved using dynamic programming. In its simplest form, we can consider all 'regions' of the diagram defined by cutting each line at two points in increasing order of size; a variety of more complex variants of dynamic programming have also been developed. The number of minimal separators for graphs in these classes is polynomially bounded, since separators correspond to sets that cross a 'scanline' defined by a point on each line. Thus, the algorithmic technique based on minimal separators outlined in the section on interval graphs often applies to these classes as well.

The generalization from intervals on a line to points and intervals on two lines has itself been extended by defining objects which have endpoints on two circles. The first such class studied is called circular permutation graphs, in which vertices map to strings which go between two concentric circles, where no pair of strings is allowed to intersect more than once.. I prefer a containment graph characterization of this class which will be described in the next section; these are containment graphs of arcs in the circle. Circular trapezoid graphs have also been studied [188].

Circle graphs are the intersection graphs of chords in the circle. These generalize permutation graphs, since any permutation diagram can be transformed to a circle graph representation by connecting the two lines at each end to form a

circle. Circle graphs are not necessarily perfect, since a cycle of any length is a circle graph. The chromatic number problem and Hamilton circuit problem are NP-complete for circle graphs [208, 147]. Some problems on circle graphs, such as the maximum clique problem, are easily solved in polynomial time by cutting the circle at appropriate places and solving corresponding permutation graph problems. The recognition problem for circle graphs, however, works rather differently, and is discussed in chapter 13. The number of minimal separators of a circle graph is polynomial [301].

[169] defines an infinite number of classes lying between the permutation graphs and the circle graphs. G is a k-polygon graph if G is the intersection graph of chords in a k-gon, where any chord must have endpoints on two different sides of the k-gon. It is NP-complete in general to decide whether G is a k-polygon graph, though the question can be answered in polynomial time for fixed k [170].

We can generalize circle graphs by allowing vertices to correspond to line segments in the plane. This class is NP-hard to recognize [324], even if lines are restricted to being parallel to the x and y axes [320]; the restricted class is called grid intersection graphs [252]. There are some very interesting issues involving representation for the more general class of lines which are not axis-parallel; Kratochvíl and Matousek [325] show that the representation of such graphs on a grid may require a doubly exponential size grid. Thus, the question of whether the class is in NP is still open; a general technique showing that this problem is in PSPACE is discussed in chapter 4. Independent set is NP-complete for the general class, and remains NP-complete if the segments have a fixed number k of directions ($k = 2$ if we do not require lines to be axis-parallel, $k = 3$ for axis-parallel lines), [326], though it is polynomial for grid intersection graphs. The complexity of the clique problem remains an interesting open question; it is polynomial if the number of directions of the segments is fixed [326].

Perhaps the ultimate generalization within the plane is the string graphs, in which vertices correspond to arbitrary non-self-intersecting strings in the plane. These classes are generally difficult to deal with; in fact, it is only very recently that it was decidable that a graph is a string graph, though the problem is now known to be in NP [421].

Gavril [213] defines an interesting combination of string graphs and interval graphs. Vertices correspond to closed curves connecting two endpoints on a fixed line L, with the restriction that a curve must stay entirely above L on the plane. Graphs which admit such an intersection model are called interval filament graphs. Although no recognition algorithm is known for interval filament graphs, there are polynomial time algorithms for the independent set and clique problems on interval filament graphs, if the model is given as part of the input. This dependence on the model raises issues which will be discussed in Chapter 14. He also generalizes this concept to circular-arc filaments, where vertices correspond to curves which connect to points in the circle, with the restriction that the curve must stay entirely inside the circle.

Instead of varying the circle, it is also possible to make variations on the object inside the circle. According to [321], Fellows first suggested classes defined by mapping vertices to polygons inscribed in the circle, with vertices being adjacent if the polygons intersect. For any fixed k-gon, this gives an implicit representation. The most natural class to study is the intersection graphs of triangles inscribed in the circle; very little is known about this class. The class defined by intersection of

arbitrary k-gons inscribed in the circle is called polygon-circle graphs (also called spider graphs), and is studied in [314, 319]. A proposed polynomial time recognition algorithm is given in [315]; however, the draft that I have seen has serious errors. Since it has been some years since the original draft and a final paper has not appeared, I believe that recognition of spider graphs must be considered to be an open problem at this point in time.

Other classes can also be defined by different forms of intersection; see [74] for more such classes. The reader might notice that the clique problem remains polynomially solvable for many of the classes defined above, and wonder about how far this can be extended. Kratochvíl and Kubena [323] show that co-planar graphs can always be represented as the intersection of convex objects in the plane; this implies that clique is NP-complete for the intersection graphs of convex objects in the plane.

3.5 Containment Representations

We can also define graphs by letting vertices correspond to geometric objects, but letting edges correspond to other types of relationships besides intersection. The most common other type of relationship in the literature is to allow an edge from x to y if object x contains object y. This naturally corresponds to a directed graph, and the graphs which come from the representation are partial orders.

The first containment model to consider is defined by a set of intervals on the line. The corresponding class of partial orders is the two dimensional partial orders. The term dimension will be used in a number of places in this book, and the terms come out of the definition of partial order dimension.

Recall that a linear extension of a partial order is an ordering of the elements so that every edge of the partial order goes forward in the ordering. The dimension of a partial order P is the minimum number of linear extensions of P which produce P as their intersection. In other words, this is the minimum number of orderings of the elements such that for all edges (x,y), x comes before y in all orderings, and for every nonadjacent pair of elements (x,y), x precedes y in some ordering while y precedes x in another. A set of linear extensions which represent the poset in this way is called a realizer. A realizer of size k can be used as the basis of an implicit representation, since we can determine whether x and y are adjacent from the positions of x and y in each of the k linear extensions. Trotter [461] has written an excellent book on partial order dimension theory.

The relationship between two dimensional partial orders and containment graphs of intervals was first noted by Dushnick and Miller [166], and is not difficult to see. If we are given a set of intervals, we form a list of left endpoints of the intervals ordered from left to right, and a list of right endpoints of the intervals ordered from right to left. Then interval i contains interval j if and only if i comes before j in both lists, so all interval containment graphs are two dimensional partial orders. If we are given a realizer consisting of two linear extensions L_1, L_2 of a poset, combine these into a single ordering L_1 followed by the reverse of L_2. For each element x, the corresponding interval goes between the two endpoints in this ordering; clearly interval x contains interval y if and only if x contains y in L_1 and L_2. A partial order is two dimensional if and only if the undirected graph formed by removing all directions is a permutation graph, as defined above, and a graph is a permutation

graph if and only if every transitive orientation is a two dimensional partial order (exercise 3.17).

Containment graphs of arcs on the circle are more commonly associated with an equivalent intersection model, the circular permutation graphs; this is discussed in more detail in chapter 13. Algorithms on this class commonly work by cutting the circle, and turning the problem into a permutation graph problem.

Containment graphs of circles, called circle orders, have received a good deal of attention in the partial order community. A well known open problem asked whether every finite three dimension partial order is a circle order. This was answered negatively in [187]; they show that there are three dimensional partial orders which are not containment graphs of spheres in any dimension. This class poses a challenge to the implicit graph conjecture; as in the case of disk intersection graphs, the natural representation by center and radius of each circle does not map $O(\log n)$ bits to a vertex.

There is a conceptually interesting application of the more general notion of spherical orders in k-space. We say that event a could have caused event b if a precedes b in time by a sufficient amount that a particle could travel from a to b in the intervening time. The corresponding partial order can be modeled by intersection graphs of spheres of appropriate dimension [424]. To the best of my knowledge, polynomial time recognition of circle orders is an open problem.

Other classes of graphs for which containment graphs have been studied include angles on the plane [195] regular polygons, and general polygons in the plane [473].

A question regarding containment graphs of paths in a tree is posed as Open Problem 5.1.

3.6 Overlap Representations

We can also define a graph class on a set of objects by saying that two vertices are adjacent if and only if they 'overlap'; that is, the intersection is nonempty and neither object contains the other. In general, overlap representations are less well studied than either intersection or containment representations.

Once again, the natural model to start with is overlap graphs of intervals on the line. Overlap graphs of intervals are equivalent to the circle graphs discussed above. To turn an interval overlap model into a circle graph model, simply connect the endpoints of the start and finish of the line to create a circle. To turn a circle graph model into an interval overlap model, cut the circle at any point which is not the endpoint of a chord. The proof that this is a valid transformation is deferred to exercise 3.5.

Although overlap graphs of each of the other classes of geometric objects can be defined, these have not been studied in general. Rim and Nakajima study overlap graphs of rectangles in [408], and describe a polynomial algorithm for the maximum clique problem. Cenek and Stewart [91] study the overlap graphs of subtrees of a tree. Algorithms for circular-arc overlap graphs are studied in [293]

Cenek and Stewart also studied some general theorems on overlap representations, one of which is described below.

The starting point for this work is Gavril's paper solving optimization problems on circle graphs, by reducing them to interval graph problems [211]. As one example, let us look at his algorithm for solving the maximum independent set

problem on circle graphs; we will assume that the input is in the form of intervals, in which x is adjacent to y in G if and only if the corresponding intervals overlap.

The algorithm considers intervals in increasing order of interval size, computing the maximum independent set on the intervals contained in each v. We add a single interval to cover all others; when we find the maximum independent set on vertices contained within this longest interval, we are done.

Let v be the current vertex such that we are computing the maximum independent set on vertices contained in v in the overlap model. Let w be an interval contained in v; give w a weight of 1 + the maximum independent set on vertices contained in w. We then compute the weighted maximum independent set on the interval graph (see Exercise 3.40b) of the intervals contained in v, i.e. we consider the intersection model instead of the overlap model. For any independent set I in the intersection model, we can construct an independent set in the overlap model with this weight by finding the maximum independent set contained within each vertex of I. Similarly, any independent set I' in the overlap model corresponds to an independent set in the weighted intersection model, consisting of the maximal intervals in I', with weights corresponding to the weights of the maximum independent sets contained within each of these maximal intervals.

As an example, consider the input 4 7 6 5 3 1 2 4 1 2 9 5 8 6 8 7 9 3 where each i occurs twice in the list, which corresponds to an ordering of the endpoints. The corresponding circle graph model is given in Figure 13.13. Ordering vertices by length of the corresponding intervals means that we process vertices in the order 8,1,2,9,4,5,6,3,7. We add an interval containing all other intervals; we call this vertex 0, and process it last.

Intervals 8,1,2 contain no other intervals, and are given weight 1. Interval 9 contains only 8, and is given the value 2. Interval 4 contains no other interval, and is given the value 1. Interval 5 contains 1 and 2; the maximum weighted independent set on the interval intersection model of 1 and 2 has weight 1, so 5 is given the value 2. Interval 6 contains 1,2, and 5; the maximum weighted independent set on the intersection model of 1,2,5 has weight 2 (vertex 5 by itself), so 6 is given the value 3. Interval 3 contains 1,2,8; the maximum weight of an independent set in the intersection model on intervals 1,2,8 has weight 2 (w(1) + w(8)), so 3 is assigned weight 3. Interval 7 contains 1,2,5,6,8, and the maximum weight of an independent set in the intersection model on these intervals is 3 (= w(6)), so w(7) = 4. Finally, we process vertex 0, which contain all other intervals. The maximum weighted independent set on the intersection model of the intervals is 4 (w(7)), which is the size of the maximum independent set in G.

Cenek and Stewart note that this generalizes to other classes of overlap besides intervals; the algorithm is the natural generalization of the algorithm above to other classes of objects besides intervals.

Theorem 3.1 [91] *Suppose that the weighted independent set problem can be solved in polynomial time on the intersection graphs of a set of objects. The maximum cardinality independent set problem can be solved in polynomial time on overlap graphs of the same objects.*

For example, Theorem 3.1 allows you to solve the independent set problem on the overlap graph of subtrees of a tree; in exercise 3.26, this is shown to contain such classes as chordal graphs, circle graphs, co-comparability graphs, and spider graphs. We note that this assumes that the input is given as an overlap model for

the objects. The problem of finding an algorithm for graphs in the class when input is given in adjacency matrix form is open for some classes, and is a special case of a problem discussed in chapter 14.

A similar, somewhat more complex condition for solving the clique problem on a class defined by an overlap model is also given in [91].

A different generalization of the algorithm for solving weighted independent set in circle graphs is described in [213]. This paper uses the techniques described in the example above to solve the weighted independent set and clique problem for interval filament graphs and circular-arc filament graphs, if the model is given as part of the input.

3.7 Generalized Intersection Models

In this section, we discuss necessary and sufficient conditions due to Scheinerman [425] for a class of graphs to be characterized exactly by some intersection representation. Similar theorems for containment models and overlap models are given in [231].

Note that exact characterization means something quite different from finding a good representation. In order to characterize a class, the objects used in the intersection representation must be able to represent graphs in the class, and not be able to represent any graph not in the class. It is possible for a class of graphs to have an implicit representation in terms of intersection of objects, but not have an intersection characterization which characterizes the class. This can happen when a class has an implicit representation by virtue of being contained in a larger class. Thus in chapter 12 we will see that every distance hereditary graph is the intersection graph of chords of a circle, but the converse is not true; chords on a circle give an implicit intersection representation of the class, but they are not a characterization of the class.

The first necessary condition for a graph to be characterized exactly by an intersection representation is that the class is hereditary. If vertices correspond to sets so that vertices are adjacent if and only if the sets have a nonempty intersection, we can represent any induced subgraph by choosing the corresponding sets from the original intersection representation.

The second simple necessary condition for having a characterization is that the class is closed under vertex expansion, which takes any vertex v and replaces it by a clique C such that each vertex of C is adjacent to the same vertices of G - v. If G is representable, we can clearly represent the new graph by letting each c in C correspond to the same set as the set corresponding to v.

The final condition is slightly harder to see. There must be a countable sequence of graphs G^1, G^2, ... such that each G^i is in the class, each G^i is an induced subgraph of G^{i+1}, and every graph in the class is an induced subgraph of some G^j. The sequence G^1,G^2,... is called a composition sequence.

To see that classes defined by set intersection must have a composition sequence, let G^i be the intersection graph of all sets which correspond to some vertex of a graph G with at most i vertices which is in the class. Since all sets from G^i remain in G^{i+1}, G^i is an induced subgraph of G^{i+1}, and every graph with i vertices in the class is an induced subgraph of G^i.

Scheinerman showed that these necessary conditions are also sufficient for a graph class to have a characterization as the intersection of a collection of sets [425].

We will not present the proof here, but we will give examples which illustrate that it is often quite simple to use these properties to determine whether various classes of graphs are intersection classes.

Connected graphs, for example, do not have an intersection characterization, since connectivity is not a hereditary property. On the other hand, connected graphs clearly are closed under vertex expansion; they also have a composition sequence (exercise 3.32).

Forests (acyclic graphs) clearly are a hereditary class of graphs, but they are not closed under vertex expansion. They also have a composition sequence (exercise 3.32, or more generally by exercise 3.34).

Consider the class of graphs which do not contain both C_4 and C_5 as induced subgraphs. This class is obviously hereditary; it is also closed under vertex expansion (exercise 3.33). However, it cannot have a composition sequence. Otherwise, C_4 would be an induced subgraph of some G^i in the composition sequence, and C_5 would be an induced subgraph of some G^j in the composition sequence; without loss of generality assume that $i < j$. Since G^i is an induced subgraph of G^j, G^j must contain both C_4 and C_5; thus G^j would not be in the class, contradicting the assumption that this is a composition sequence.

The theorems for having a representation in terms of containment of sets, overlap of sets, and disjointness of sets are remarkably similar to the theorem above. A class is closed under vertex multiplication if the graph created by creating a new vertex with the same neighborhood as an old vertex is always in the class; multiplication differs from expansion in that vertices introduced via the latter operation are nonadjacent to their twins, while vertices introduced by expansion form a clique with their twin.

Theorem 3.2 [425, 231] *A graph class is an intersection class if and only if it is hereditary, is closed under vertex expansion, and has a composition sequence. A class of posets is a containment class if and only if it is hereditary, is closed under vertex multiplication, and has a composition sequence. A class of graphs is an overlap class if and only if it is hereditary, is closed under vertex multiplication, and has a composition sequence. A class of graphs is a disjointedness class if and only if it is hereditary, is closed under vertex multiplication, and has a composition sequence.*

The strong similarity between these characterizations may seem surprising, but is understandable when you consider getting from models in one sense to models in another. It is easy to see that an intersection model corresponds to a disjointedness model of the complement; the difference in the characterizations comes from the fact that making a copy of the representation of a vertex gives rise to an independent set in a disjointedness model and a clique in an intersection model. One can turn an intersection model into an overlap model by adding a 'private bit' for each vertex, meaning that each vertex is given a bit in the representation which is shared by no other vertex. Once again, the difference between intersection and overlap comes from the fact that identical sets are adjacent in the intersection model and nonadjacent in the overlap model. Some of the other constructions are more difficult. Note that these constructions are meant to give the reader intuition as to why the classes have similar characterizations; they do not constitute a proof, as is given in [231].

3.8 Perfect Graphs

One class of graphs which will not be studied in this book, but is relevant to the study of some optimization problems on these classes, is the class of perfect graphs. Readers interested in studying perfect graphs are advised to consult [403, 223, 74] and the excellent web-based resource due to Chvátal available at www.cs.rutgers.edu/ chvatal/perfect/problems.html.

A graph G is perfect if for every induced subgraph H of G, the chromatic number of H is equal to the maximum clique size of H. There are many fascinating facts and open problems on perfect graphs.[1] Among these are the following.

Theorem 3.3 [352] *G is perfect if and only if the complement of G is perfect.*

Strong Perfect Graph Conjecture/Theorem [34]: G is perfect if and only if neither G nor its complement contains any induced cycles of length $2k+1$ for $k > 1$.

As I write this, a proof of the strong perfect graph conjecture [102] has just appeared. The discussion of the proof in this book is in Section 11.6, which deals with skew decomposition, which is an important tool in the proof.

Induced cycles of length > 3 will also be called holes. Even though the strong perfect graph conjecture is now known to be true, we still do not know whether perfect graphs can be recognized efficiently, since we do not know how to find 'odd holes' in polynomial time. Thus, the following problems are also open; the first problem is particularly well known, and the problems here are likely to be very difficult to solve.

Open Problem 3.3: Is there a polynomial time algorithm for recognizing perfect graphs?

We note that the problem was known to be in co-NP even before the conjecture had been proved as a consequence of results in [87] and [237]; see also exercise 3.28.

Open Problem 3.4: Is there a polynomial time algorithm to determine whether a graph has an odd hole?

The related problem of determining whether a graph has an even hole can be solved in polynomial time [117]. This was one of a series of results by a group of authors which made the strong perfect graph conjecture seem possible to prove, and showed the general direction that the proof eventually followed. It seems possible that some of these techniques may eventually be extended to odd hole recognition as well.

The following beautiful result due to Grötschel, Lovász, and Schrijver will be relevant to us, since perfect graphs contain a good number of the classes discussed in this book.

Theorem 3.4 [237] *The maximum clique, maximum independent set, chromatic number, and clique cover number problems can be solved in polynomial time for perfect graphs.*

Since perfect graphs cannot at this time be recognized efficiently, it is interesting to ask what this algorithm does if the input graph is not perfect. If the input graph G is not perfect, the algorithm may either provide evidence of the imperfection

[1] As an indication of my interest in this class and/or disassociation from reality, I note that I have a framed T-shirt in my office with classes of perfect graphs, specially treasured because it was signed by Claude Berge.

of G, or solve the optimization problem correctly on G. Thus, the result of the algorithm is reliable, whether or not G is perfect. Algorithms with this property for other classes are studied in chapter 14. We note that although the optimization algorithms for perfect graphs are polynomial, the algorithm is extremely difficult; it uses the ellipsoid algorithm from linear programming theory as a subroutine. No purely combinatorial algorithm is known for solving these problems on perfect graphs, which is why we will study optimization algorithms for subclasses of perfect graphs as well.

3.9 Exercises

1) G is B_1-orientable or fraternally orientable if and only if one can assign directions to the vertices so that if x and y both have an edge to z, there is an edge between x and y. The name B_1-orientable comes from [434], which deals with a number of forbidden configurations, with B_1 being a graph associated with this special case. The paper [474] focuses on this specific orientation, and gives it the name fraternal orientation.

1a) Show that G has an acyclic B_1-orientation if and only if G is chordal.

1b) Prove or disprove: every comparability graph has a B_1-orientation.

1c*) Design an $O(n^3)$ algorithm to find a B_1-orientation of a graph.

1d) Design an algorithm which verifies whether an orientation is a B_1-orientation which runs in $o(n^3)$ time.

2) Show that a co-comparability graph cannot contain any chordless cycles of length > 4.

3) Show that the largest possible number of maximal cliques of a boxicity 2 graph is $O(n^2)$.

4) Show that there are graphs with boxicity 2 which are not circular-arc graphs.

5) Show that overlap graphs of intervals on the line are exactly the circle graphs.

6) Design an $O(n+m)$ algorithm which takes a representation of a circle graph on the circle, and produces an adjacency list representation of the graph.

7) Show how to solve the clique problem on circle graphs in polynomial time; you can assume there is a polynomial time algorithm for the clique problem on permutation graphs.

8) Greedy coloring on interval graphs works as follows. Process vertices in order of leftmost endpoint, and assign v the smallest color not assigned to a neighbor of v. Show that this algorithm colors an interval graph using the optimal number of colors.

9) The dominating set problem looks for the smallest possible set S of vertices such that every vertex of V - S is adjacent to at least one vertex in S.

9a) Design a polynomial time algorithm for solving the dominating set problem on interval graphs, and analyze its time complexity.

9b) Design a polynomial time algorithm for solving the dominating set problem on permutation graphs, and analyze its time complexity.

10) Show that any minimal separator of a circular-arc graph corresponds to the set of vertices which pass through a pair of points on the circle in any arc model.

11a) Show that every circular-arc graph has interval number at most 2.

11b) Show that every degree 3 graph has interval number at most 2.

12a) Show that it is possible to have a clique of size k in a representation of a circular-arc graph in which no point of the circle is covered by more than $k/2 + 1$ arcs.

12b) Show that if a circular-arc graph contains a clique of size k, there must be some point of the circle covered by at least $k/2$ arcs in any circular-arc representation.

Note: The bounds of 12a and 12b can be easily made to match, but I did not want to make the reader deal with the arithmetic issue of whether the clique size is even or odd.

13) Show that there can be cliques in the model of a disk intersection graph such that no point in the model is covered by all circles in the clique.

14) Show that there can be an exponential number of maximal cliques in a disk intersection graph.

15) Show that the interval number of a complete bipartite graph with n vertices on each side is $\Omega(n)$; the precise formula is given in [236].

16a) Show that graphs with total interval number at most cn are not hereditary.

16b) Show that graphs with total interval number at most cn do not have an implicit representation.

17) Show that a partial order has dimension at most 2 if and only if the underlying undirected graph is a permutation graph.

18) Show that there are $2^{\Theta(n^2)}$ intersection graphs of paths on the grid. Nomikos [387] showed that every graph is the intersection graph of paths on a grid.

19) Show that every graph has interval number at most n-1.

20) Prove that every poset has a dimension.

21) A Hiraguchi poset is a height 1 poset with k source vertices and k terminal vertices, such that s_i has an edge to every terminal except for t_i. Show that the Hiraguchi posets have dimension k. In fact, every poset has dimension at most $n/2$, so these are extremal posets for dimension [461].

22) Show that the number of minimal separators in a graph can be exponential.

23) Show that if S is a minimal separator of an interval graph, then every vertex of S passes through a common point on the line.

24) Show that G is partial order if and only if G is the containment graph of substars of a star.

25) Show that G is the overlap graph of sets with an element common to all sets if and only if G is a co-comparability graph.

26) Show that each of the following classes of graphs is contained in the overlap graphs of subtrees of a tree.
a) chordal graphs
b) circle graphs
c) co-comparability graphs
d) spider graphs (i.e. intersection graphs of polygons inscribed in a circle)

27) Show that the interval filament graphs contain the circle graphs.

28) A graph on n vertices is partitionable if $n = \alpha\omega + 1$, and for every vertex x of G, the size of the largest independent set on G-x = the clique cover number of G-x and the cardinality of the largest clique in G-x = the chromatic number of G-x. It can be shown that every minimal imperfect graph is partitionable, and every partitionable graph is imperfect [223]. Use these facts to show that perfect graph recognition is in co-NP, without using the fact that the strong perfect graph conjecture is now known to be true. Note: Shepherd [430] has recently developed a polynomial time algorithm for recognizing partitionable graphs, but this is not necessary for showing perfect graph recognition is in co-NP without resorting to the strong perfect graph theorem.

29) Show that if G is perfect, then either G or its complement has a clique of size at least $n^{1/2}$. Note: It would be interesting to develop fast algorithms for finding a clique or independent set of this size very efficiently. One variant of this problem, which corresponds to restricting the graphs to be permutation graphs, is to find an increasing or decreasing subsequence of length $\Omega(n^{1/2})$ in a permutation in $O(n)$ time; this can be solved in $O(n\text{loglog}n)$ time by using algorithms for finding the largest independent set in a permutation graph when the model is given.

30) Show that there are triangle-free graphs with arbitrarily large chromatic number.

31) The class of graphs with no odd holes of length greater than 3 in either the graph or its complement is called the class of Berge graphs ; the strong perfect graph conjecture is equivalent to saying that every Berge graph is a perfect graph. At this point in time, it is possible to imagine the existence of a Berge/perfect graph recognition algorithm which does not imply a polynomial time algorithm for determining whether a graph has an odd hole. Explain why the following approach does not give a valid algorithm for determining whether G has an odd hole, even if we assume that there is a polynomial algorithm for Berge graph recognition.

To determine whether G has an odd hole, repeat the following steps. While G is nonempty and not a Berge graph, test whether G - x is a Berge graph for each vertex x. If such a vertex x exists, delete x and continue. If no such x exists and x is not Berge, then G is either an odd hole (in which case we find an odd hole) or an odd antihole (in which case we say no odd hole exists).

32a) Show that connected graphs have a composition sequence.

32b) Show that forests have a composition sequence.

33) Show that graphs without C_4 or C_5 as induced subgraphs are closed under vertex expansion.

34) Show that any graph class closed under disjoint union has a composition sequence.

35) Show that comparability graphs and perfect graphs have intersection characterizations.

36) Show that the containment graphs of trapezoids with endpoints along two parallel lines are exactly equal to 4 dimensional partial orders.

37) A k-tree is a graph formed by starting with a clique of size k, and successively adding vertices which are adjacent exactly to a clique of size k-1 in the old graph. Show that every k-tree is chordal, but not every chordal graph is a k-tree.

38) Show that in every trapezoid graph, there is at least one pair of vertices x,y such that N(x) is a subset of N(y); this shows that trapezoid graphs are a subset of domination graphs, which must have such a pair for every induced subgraph.

39) Give an example of a circle graph which is not an interval graph, and an interval graph which is not a circle graph.

40a*) Design an O(n+m) algorithm to find the maximum weighted independent set in a chordal graph.

40b) Design an O(n) algorithm to find the maximum weighted independent set of an interval graph, if the intervals are given as input.

41) Golumbic and Scheinerman also discuss containment classes of graphs; we gave the theorem on containment graphs of posets. For a poset, when $x \rightarrow y$, we require that set x contains set y in a containment representation. There are two possible notion of a containment class of graphs; we can require that there are representations of every transitive orientation by sets in the class, or that there is a representation of some transitive orientation by sets in the class. To illustrate the difference, consider the following example due to Corneil and Golumbic, taken from [231]. An 8-wheel is a cycle on 8 vertices, plus a universal vertex.

41a) Show that there is a transitive orientation of the 8-wheel which is representable as containment of paths in a tree.

41b) Show that there is a transitive orientation of the 8-wheel which is not representable as containment of paths in a tree.

42) An f-diagram, or 2-string graph, generalizes permutation graphs by allowing a vertex to be represented by a 'piece of string' rather than a line, with endpoints on each of two parallel lines as in permutation graphs. Show that G is the intersection graph of an f-diagram if and only if G is a co-comparability graph.

43) Show that not every tree is a permutation graph, but every tree is a circle graph.

44) Show that the class of intersection graphs of triangles inscribed in a circle properly contains the class of circle graphs.

45) In greedy coloring, we fix an ordering of the vertices, and color the next vertex using the smallest color not already assigned to any neighbor of the vertex. Show that if the ordering is the reverse of a perfect elimination scheme, greedy coloring produces an optimal coloring of a chordal graph.

46) Richard Borie considered the class of unit circle graphs, in which every chord has the same length.

46a) Show that there are circle graphs which are not unit circle graphs.

46b) Show that every overlap graph of unit intervals is a unit circle graph, but that the converse is not true.

46c) What other class of graphs is equivalent to overlap graphs of unit intervals?

47) Design an $O(n)$ algorithm for finding the shortest path from s to t in a permutation graph, if the permutation diagram is given as part of the input.

48) Describe a method for taking a containment model for a class, and producing an overlap model for the corresponding co-comparability graphs.

CHAPTER 4

Real Numbers in Graph Representations

This chapter deals with a very natural method for defining a class of graphs which gives us problems when we try to find an implicit, or other space efficient, representation. We will talk about defining graph classes using simple geometric objects such as circles. Although circles can be represented using a small number of variables corresponding to the center and radius of the circle, these variables do not naturally correspond to small integer values. We will discuss the problems of counting, representing, and recognizing graph classes of this type, and point out some of the better known classes for which the problem of efficient representation remains open.

For most of the classes discussed in chapter 3, the intersection, containment, or overlap model for a class of graphs led immediately to an implicit representation. For example, graphs with boxicity k can be represented using $O(k\log n)$ bits per vertex, by giving the order of endpoints in each dimension. However, this approach does not work to give an efficient representation of disk intersection graphs. The reader may feel differently, but I see no particular reason to believe that either of these classes is more complex than the other, if we view them as a set of graphs which happen to have mappings to boxes and spheres. If we think about efficient representation, however, the problem of representing circles and spheres is much more complex than the problem of representing rectangles and boxes.

Let us briefly consider the obvious method of storing a representation for a disk intersection graph; for each vertex, we store an x coordinate, a y coordinate, and the radius of the corresponding circle. If all these values are integers in the range $0..n^k$ for some value of k, then this is an implicit representation. However, these values are not necessarily even rational, and it is not clear at all that there is any grid size large enough so that we can use integer values for all centers and radii. In the previous chapter, the reason we know that the graph classes have $2^{O(n\log n)}$ members has nothing to do with studying the class of graphs; instead, the $O(n\log n)$ bit representation itself tells us that the number of graphs in the class is $2^{O(n\log n)}$. For disk intersection graphs, we do not know any efficient representation, so it is not clear that the graph class is even small enough to hope for an implicit representation. In the next section, we will describe a powerful counting technique which enables us to prove that the number of disk intersection graphs, as well as many other geometrically described classes of graphs, is $2^{O(n\log n)}$. However, no good representation is known, despite the fact that disk intersection graphs are clearly a hereditary class of graphs, leading to the following open problem.

Open Problem 4.1: Is there an implicit representation of the disk intersection graphs?

Open Problem 4.2: Show that there are $2^{O(nlogn)}$ disk intersection graphs using only graph theory, i.e. without resorting to Warren's theorem.

The same problems are open for other classes, such as circle orders, intersection graphs of lines in general position on the plane, and many other classes.

4.1 Warren's Theorem

Warren's theorem is the key mathematical tool which allows us to put good upper bounds on the number of graphs in many geometrically defined classes. Discussing the history of Warren's theorem may put the author in the strange position (for an author of a mathematical text, at least) of being on the fringe of debates about political correctness.[1] Warren's theorem was published in 1968 in a paper entitled 'Lower Bounds for Approximation by Nonlinear Manifolds' as a way of obtaining a bound on how closely polynomials can approximate more general subspaces. Given the subject area, the paper was completely unknown to those of us working with graphs. Warren's paper was later discovered by Alon in the 1980s, who realized that this gave a tool for counting the number of posets such that elements can be described using a fixed number of variables. The same tools have been used to count many other classes of graphs, and it often can give good bounds quite simply. However, the technique is still unknown even to many researchers in the field, since it does not appear in any book at the time of this writing, and one still finds papers which ask whether graph class A is a subset of graph class B, when class B can be shown to be much smaller than class A using Warren's theorem. I would like to add that most researchers, including myself, would not be able to bound the numbers of graphs in the classes without using Warren's theorem, making Warren's theorem one of the deepest and most important results which we will discuss in this book.

Warren's theorem is stated in terms of the number of sign patterns of a set of polynomial equations. A sign pattern is a sequence of +s and -s, corresponding to whether each polynomial equation is nonnegative or negative. For example, suppose that we are given three polynomial equations $5x^2$-$7x$+2, $3x$-9, $4x^2$+$13x$-124. One possible sign pattern for this set of equations is +,-,-, since if $x = 2$ the first equation evaluates to a positive number while the second and third are negative. The number of sign patterns is clearly bounded above by 2^k, where k is the number of equations, but the number is not in general nearly as large as 2^k. In our example, it is easy to see that no sign pattern for this set of equations can begin -,+; if the second equation evaluates to a positive number, then $x > 3$, but it is easy to see that this implies that the first equation evaluates to a positive number. This argument, however, involves looking at the particular set of equations, and says nothing about the number of sign patterns with respect to a general set of polynomial equations.

Before stating Warren's theorem, let us define some parameters which affect the number of possible sign patterns. The number of variables in the set of polynomial

[1] In late 20th century America, there has been an odd point of political contention over the statement 'Columbus discovered America'. How could he have discovered it, says one side, when the Native Americans already lived there? However, I would say that Noga Alon discovered Warren's theorem, in the sense of bringing it into the realm of knowledge of the graph theory community, even though it was clearly already known to others, including Warren; it is in this sense that Columbus 'discovered' America.

equations will be n, and the maximum degree of any individual polynomial equation is d. The total number of equations will be denoted as m.

Warren's Theorem [479]: The number of possible sign patterns is at most $(cdm/n)^n$, where $c = 4e$ and e represents the standard constant, approximately 2.72.

We will make no attempt to prove Warren's theorem here. Typically, we might have a constant number of variables for each vertex, and a single equation to determine adjacency between a pair of vertices. Sometimes, as for visibility graphs, we need n^i for $i > 2$ equations to test adjacency between a pair of vertices. In any case, Warren's theorem tells us that the number of sign patterns, and hence the number of graphs, is at most $(n^k)^{jn}$ for some constants j and k, which is $2^{O(nlogn)}$. This will give a method for showing that a graph class is small without giving an efficient representation.

Let us see how Warren's theorem can be used to show that the number of graphs in a class is small. We will use disk intersection graphs as our example. For each vertex v, we have three variables v_x, v_y, v_r, which represent the x and y coordinates of the center and radius of the circle which corresponds to v. We now want to write a polynomial in these variables which changes signs depending on whether the corresponding circles have a nonempty intersection. For this class, creating such a polynomial is not difficult. For each pair of vertices v, w, we have

$$(v_x - w_x)^2 + (v_y - w_y)^2 - (v_r + w_r)^2.$$

The first two terms of the equation give the square of the distance between the centers; the circles intersect if and only if the sum of the radii is larger than the distance between the centers, so this equation is less than 0 if and only if the circles have a nonempty intersection. This means that if you give me the sign of the result of each equation, I can reconstruct the graph. Since the number of sign patterns is $2^{O(nlogn)}$ by Warren's theorem, the number of disk intersection graphs is $2^{O(nlogn)}$.

No method is known for constructing implicit representations of graph classes which can be shown by Warren's theorem to be sufficiently small so that these might exist. Any such constructive proof would be a major breakthrough.

Alon and Scheinerman [17] characterize classes of partial orders in terms of 'degrees of freedom', which is essentially the number of variables per vertex in the polynomials which describe graphs in the class. This is used to count the number of orders in the class. We could define similar notions of graph classes.

Call a graph class 'Warren representable' if for all n, there is a set of $O(n^k)$ polynomial equations on $O(n)$ variables such that for every graph G in the class, adjacencies in G can be determined from the signs of the polynomials. We could make a conjecture similar to the implicit graph conjecture, that every hereditary class of graphs with $2^{O(nlogn)}$ members is Warren representable. In general, there is no known containment relationship in either direction between implicitly representable graphs and Warren representable graphs.

Note that showing that a graph class is Warren representable is somewhat less useful than showing that it has an implicit representation, since Warren's theorem is a counting tool while implicit representations are useful for efficient storage and fast adjacency testing. Nevertheless, the Warren representable graphs pose a major challenge for the implicit graph conjecture. A number of Warren representable

graph classes are known which are not known to have an implicit representation; these will be discussed at various points in this book.

4.2 Continuous Nongeometric Variables

The size of a number in a representation can also pose a problem for classes of graphs which are not defined geometrically. One of the more natural classes for which this problem arises is representing a graph using a dot product representation. Dot product representations were introduced in [191].

A k-dot product graph is a graph in which each vertex can be mapped to a sequence of k real numbers, where x and y are adjacent if and only if the dot product of the corresponding numbers (that is, the sum of the values obtained by multiplying the ith number for x by the ith number for y) is at least 1.

Since every graph is a k-dot product graph for some value of k (exercise 4.4), there is also a natural notion of dot product dimension; the dot product dimension of G is the smallest k such that G is a k-dot product graph.

If every number is a small integer, a k-dot product representation is clearly implicit. From the definition, however, numbers are not necessarily even rational, much less small integers.

Warren's theorem applies trivially to dot product representations, since each adjacency test is defined by testing whether a polynomial on a set of $2k$ variables is at most 1; subtracting 1 clearly makes the adjacency of two vertices depend only on the sign of the corresponding polynomial. Therefore, for any fixed k, the number of k-dot product graphs on n vertices is $2^{O(nlogn)}$, but it is not clear that any dot product representation is space efficient.

The paper [191] shows that numbers in the dot product representation may be restricted to be rational without loss of generality. However, this bound on the number of digits in the representation is not known to be polynomial, making membership in NP an open problem even for 2-dot product representable graphs. Thus, finding an implicit representation for k-dot product graphs is an open problem. We state a few variants of this problem, mostly chosen from a larger set of questions in [191], which are not equivalent but which may be related to each other. The problems seem to be open for any fixed $k > 1$; dot product dimension 1 graphs are equal to threshold graphs defined below, which have nice representations for a variety of reasons.

G is a threshold graph if one can assign a weight $\mathrm{w}(v)$ to each vertex v such that x and y are adjacent if and only if $\mathrm{w}(x) + \mathrm{w}(y) > t$, where t is called the threshold. For an unusually thorough treatment of threshold graphs, see [361]. Note that the definition given here is not the original definition, instead being a characterization of the class which fits particularly nicely within the framework of this book. Although the weights in threshold graphs are allowed to be arbitrary real numbers, they have an implicit representation since they are a subset of (for example) interval graphs.

Open Problem 4.3: Find an implicit representation for k-dot product representable graphs.

Open Problem 4.4: Can all k-dot product representations use numbers which are bounded by $O(log^i n)$ bits, or failing this a polynomial number of bits?

Open Problem 4.5: Can you determine whether a graph is a k-dot product graph in polynomial time?

Open Problem 4.6: Are there NP-complete problems which are polynomially solvable on k-dot product graphs? This problem may be interesting both if a k-dot product representation is given, and if it is not; see chapter 14 for more on this distinction.

A variety of graph classes in which vertices correspond to intervals and there is a 'tolerance' value associated with each interval have been defined, where vertices x,y are adjacent if the overlap length exceeds some function on the tolerances of x and y. These classes are all open for recognition and implicit representation, and can be shown to have $2^{O(nlogn)}$ members; the most popular of these classes is discussed in exercise 4.6.

4.3 Decidability Results

Many of us with backgrounds in computer science rather than in mathematics are used to the notion that all problems we are likely to encounter are at least decidable. This is probably reinforced by the teaching of NP-completeness as a practical notion of difficulty, relegating decidability problems to a more theoretical course in automata theory.

For these natural recognition problems, the decidability question is by no means obvious. We certainly cannot step through, for example, all possible placements of circles on the plane to decide whether a graph is a disk intersection graph. Just as Warren's theorem can be invoked to show a bound on the number of graphs in a large number of geometrically defined classes, we will present a theorem due to Tarski which will show that the recognition problem on many of the same classes is decidable. Often, this is the only positive result we have on recognition of these classes.

We will state a limited form of Tarski's theorem, which is sufficient for showing that the geometrically defined classes which are defined in this book are decidable. Note that in an inequation, the left hand side is required to be unequal to the right hand side.

Theorem 4.1 [457] *Consider any finite set of polynomial equations, inequalities, and inequations. It is possible to decide in a finite number of steps whether there are real numbers which satisfy all of these.*

Thus, there is an algorithm to determine (for example) whether a graph is a disk intersection graph; we can write polynomial inequalities in the model which corresponding to adjacencies in the graph.

Unfortunately, the methods for determining whether there is a solution are doubly exponential, so that these general techniques are not in general able to produce practical algorithms for recognizing geometrically defined classes of graphs.

4.4 Exercises

1) Consider the class of intersection graphs of a set of lines in the plane, when the lines can take arbitrary positions (i.e. these are not restricted to be parallel to the x and y axes). Show that this class has $2^{O(nlogn)}$ members.

2) One can define a partial order on a set of angles in the plane by $x \rightarrow y$ if and only if the portion of the plane inside of the angle defined by x contains the portion of the plane inside the angle defined by y. P is an angle order if and only if it corresponds to a partial order for some set of angles on the plane. Show that the number of angle orders is $2^{O(nlogn)}$.

3) Show that the number of circle orders is $2^{O(nlogn)}$.

4) Show that every graph is a k-dot product graph for some value k.

5) Show that G is a 1-dot product graph if and only if G is a threshold graph.

6) A tolerance graph is defined by a set of intervals, and a 'tolerance' associated with each interval, which can be an arbitrary number. Two vertices x,y are adjacent if the length of the overlap between x and y is greater than the tolerance of x or is greater than the tolerance of y. Show that there are $2^{O(nlogn)}$ tolerance graphs. Recognizing tolerance graphs and finding an implicit representation for this class are open questions. Hayward and Shamir [255] have shown that the problem is in NP, since the endpoints and tolerances can be restricted to having a polynomial number of bits. Golumbic and Trenk are preparing a book on tolerance representations.

7) It has been shown that tolerance graphs are a subset of weakly chordal graphs. Another subclass of weakly chordal graphs has been called 'domination graphs'; G is a domination graph if every induced subgraph H of G contains a pair of vertices x, y such that N(x) is a subset of N[y] in H.

a) Show that there are domination graphs which are not tolerance graphs.

b) Show that every tolerance graph is a domination graph.

8) A graph is a bounded tolerance graph if it has a tolerance representation in which the tolerance of every interval is less than the length of the interval. A well known open problem is whether every co-comparability graph which is a tolerance graph has a bounded tolerance representation; Felsner [186] has shown that all such graphs are trapezoid graphs.

a) Show that every bounded tolerance graph is a trapezoid graph.

b) Show that there is a tolerance graph which is not a trapezoid graph.

9) Show that every permutation graph is a tolerance graph.

10) In a probe interval graph, vertices correspond to intervals, but every vertex is marked as either a probe vertex or a nonprobe vertex. Two vertices are adjacent if their intervals overlap, and at least one of the vertices is a nonprobe. Show that every probe interval graph is a tolerance graph. Probe interval graphs arise from applications in DNA sequencing; see chapter 16 and [368] for a description of the application.

CHAPTER 5

Classes Which use Global Information

In this chapter, we will study representations of graph classes which have efficient representations with regard to space complexity, but which use global information for adjacency testing. Typically, adjacency testing using natural storage methods will be slow when compared to testing in classes with efficient representations. We will show how to produce implicit representations for these classes, when such a construction is known. Visibility representations also are examples of graph classes which use global information; visibility graphs are covered in chapter 6. Distance hereditary graphs also have a characterization which uses a small amount of space but global information; see chapter 12.

5.1 Paths in Trees

There are a number of classes of graphs defined using paths in a tree. These are probably the clearest examples of graph classes defined with a natural $O(n\log n)$ bit representation, but adjacency testing is not local (or efficient) if this representation is used.

These classes can all be described loosely as the intersection graphs of paths in a tree, with variations coming from alternative definitions of trees, and intersection of paths. Trees can be either undirected, directed, or rooted; in a rooted tree all paths would be on a single path from the root, while in an undirected tree a path may not be contained on a simple path from the root. In a directed but unrooted tree, arbitrary directions may be assigned to edges of an undirected tree. Different notions of intersection arise depending on whether we count two paths which cross at a vertex as intersecting, or whether we require them to share an edge; this distinction comes from viewing a path as a sequence of vertices or as a sequence of edges.

We note that the differences between the graph classes defined by different interpretations of trees and intersection can be considerable. For example, vertex-intersection graphs of undirected trees (called path graphs) can be recognized in polynomial time, while edge-intersection graphs of undirected trees (EPT graphs) are NP-complete to recognize. The isomorphism problem can be solved in polynomial time for vertex-intersection graphs of rooted trees (called rooted directed path graphs), but is isomorphism-complete (i.e. solving the problem in polynomial time for graphs in the class would imply a polynomial time algorithm for the general graph isomorphism problem) for both directed path graphs and for path graphs [25].

We can find implicit representations of all these classes; details are left to exercise 5.1. In all cases, we can show that the tree can be assumed to have size $O(n)$, and we can determine the adjacencies of vertices by storing enough

information to compute ancestor information, and keeping a small set of vertices for each path. The different variants are explored in the exercises.

Note that containment graphs of paths in a tree can be handled similarly. For no particular reason, containment graphs of paths in a tree do not seem to have been studied. Since containment graphs of intervals, which can be viewed as containment graphs of paths in very restricted trees, correspond exactly to two dimensional partial orders, the following question seems natural.

Open Problem 5.1: Is there a constant k such that all containment graphs of paths in a tree have dimension at most k?

5.2 Chordal Comparability Graphs

One of the organizing themes for Golumbic's excellent book [223] is whether a graph G and/or its complement is chordal, and whether it or its complement is a comparability graph. Thus, there are chapters on comparability graphs, chordal graphs, split graphs (= chordal and co-chordal graphs), permutation graphs (= comparability and co-comparability graphs), and interval graphs (= chordal co-comparability graphs). This leads to good characterizations of almost all graph classes defined by a pair of properties from the set chordal, co-chordal, comparability, co-comparability; note that the complement of a graph class, such as the co-chordal comparability graphs, are well characterized as the complements of interval graphs. The only pair of properties from this set which is not dealt with in Golumbic's book is the class of chordal comparability graphs, and their complements.

First, we will show that the number of chordal comparability graphs is $2^{O(nlogn)}$. We will show this by giving a natural O(nlogn) bit representation for any graph in the class. However, the representation is not implicit, because it uses global information to test adjacency. We will then show that chordal comparability graphs are contained in a class which has an implicit representation, which gives an implicit representation for chordal comparability graphs. The representation is not entirely satisfactory; we will see that it does not characterize chordal comparability graphs, and why this is intuitively unsatisfying. The problem of creating a natural implicit representation which exactly characterizes the chordal comparability graphs remains open; see Open Problem 3.1. We note that a partial order which has a chordal graph as its underlying undirected graph is called a cycle-free partial order, and is studied under that name in [461].

The first procedure for finding an O($n$log$n$) bit representation of a chordal comparability graph G is not difficult to understand. The transitive closure of a directed acyclic graph G is a directed graph G' on the same vertex set, such that there is an edge from x to y in G' if and only if there is a path from x to y in G. The transitive reduction G'' of a directed acyclic graph G is a directed graph on the same vertex set such that there is an edge from x to y in G'' if and only if there is an edge from x to y in G, and there is no vertex z such that there is a path from x to z and a path from z to y in G.

Choose any transitive orientation G' of G, and find the transitive reduction G'' of G'. We will show that G'' contains at most $2(n\text{-}1)$ edges. G'' is an O($n$log$n$) bit representation of G, since G' can be reconstructed from G'' by using a transitive closure algorithm, and G can be derived from G' by removing directions from all edges.

Theorem 5.1 [356] *Let G be a chordal comparability graph, G' be any transitive orientation of G, and G'' be the transitive reduction of G'. Then G'' contains at most 2(n-1) edges.*

Proof Consider a perfect elimination ordering of G. Add vertices in the reverse order of the perfect elimination scheme; this guarantees that the next vertex v added is always simplicial in G. Let G_1 be the transitively reduced graph before v is added, and G_2 be the transitively reduced graph after v is added. Any edge of G_2 - G_1 must have v as one endpoint; if an edge was implied by transitivity without v in the graph, it remains implied with v in the graph.

There can be at most one inedge to v and at most one outedge from v in G_2. Suppose v has an edge to x and y in G_2. Since v is simplicial, x and y must be adjacent; without loss of generality, assume that there is an edge from x to y in G'. Since v has an edge to x and there is an edge from x to y in G', there is no edge from v to y in G''.

Since a 1 vertex graph has no edges, and each vertex added adds at most 2 edges, the number of edges in G'' is at most $2(n\text{-}1)$. □

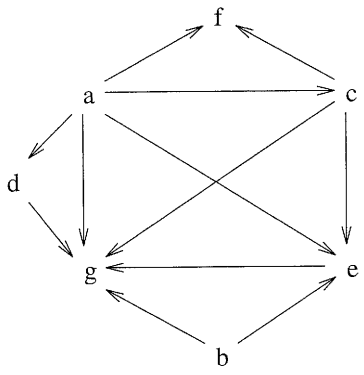
We will use the intersection representation of chordal graphs from chapter 15 to show that chordal comparability graphs are contained in the comparability graphs of 4 dimensional posets. Thus, a chordal comparability graph can be stored by storing with each vertex x the position of x in all linear extensions of a transitive orientation of G, such that x and y are adjacent if and only if the for one of the vertices the ith number stored at that vertex is smaller than the ith number stored at the other vertex for i=1,2,3,4.

Theorem 5.2 *:* [356] *Every chordal comparability graph is the comparability graph of a poset with dimension at most 4.*

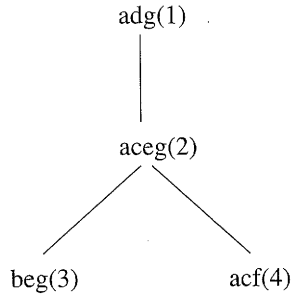
Proof Let P be any transitive orientation of G; we will show that P has dimension at most 4. Suppose that we are given a representation of G as the intersection graph of subtrees of a tree T, and let T_i denote the subtree of T which corresponds to vertex i of G. Let DFS(T) be an arbitrary depth first search ordering of the vertices of T.

To see how the appropriate linear extensions are constructed, we must think of how subtrees corresponding to nonadjacent vertices u, v of G could be traversed. Without loss of generality, assume that the first vertex of T_u is visited before the first vertex of T_v during DFS(T). We say u and v have a type 1 relationship if T_u is traversed entirely before T_v, and that they have a type 2 relationship if we enter T_u first, and at some point leave T_u and traverse all of T_v before reentering T_u. We design linear extensions which reverse the orders of u and v for each of the two types of relationships.

Our linear extensions of P work as follows. We create edge sets E_1, E_2, E_3, E_4; all edge sets include the edges of P. Let u, v be type 1 nonadjacent vertices such that DFS(T) visits T_u before T_v; E_1 adds an edge from u to v, while E_2 adds an edge from v to u. For type 2 nonadjacent vertices x, y such that DFS(T) visits some of T_x, leaves T_x and visits all of T_y and returns to T_x, E_3 adds an edge from x to y, while E_4 adds an edge from y to x. We then create linear extensions L_i for each of the sets E_i.



P



Clique Tree with DFS ordering

Figure 5.1 Chordal Comparability Graph

In the graph of Figure 5.1, the pairs (b, f), (b, d), (c, d), (d, e), (d, f), (e, f), (f, g) have type 1 relationships; (a, b), (b, c) have type 2 relationships. One possible set of linear extensions $L_1 = a\ d\ b\ c\ e\ g\ f$ (note that b may precede c even though c is encountered before b, since (b, c) is not a type 1 relationship), $L_2 = a\ c\ f\ b\ e\ d\ g$, $L_3 = a\ c\ b\ d\ e\ f\ g$, $L_4 = b\ a\ c\ f\ d\ e\ g$.

If x is adjacent to y, then there is an edge (w.l.o.g) from x to y in P, and x comes before y in all extensions. If x and y are nonadjacent, either T_x is visited entirely before T_y, in which x has an edge to y in E_1 while y has an edge to x in E_2, or (w.l.o.g) T_y is visited in the middle of the traversal of T_x, in which case x has an edge to y in E_3 while y has an edge to x in E_4. Therefore, if all edge sets are acyclic, the linear extensions represent all adjacencies properly.

The proofs that E_1 and E_2 have no cycles are essentially the same, as are the proofs that E_3 and E_4 have no cycles; therefore, we only prove that E_1 and E_3 are acyclic. Proof details omitted here can be found in [356].

Claim 5.1: E_1 is acyclic.

Proof Let C be a shortest length cycle in E_1. C cannot contain two consecutive edges from P, since P is transitive. C cannot contain two consecutive edges $v \rightarrow w \rightarrow x$ from E_1 - P, since this would mean that v is visited entirely before w and w is visited entirely before x, so C would have a chord $v \rightarrow x$. This implies there are edges $v \rightarrow w \rightarrow x \rightarrow y$ of C such that $v \rightarrow w$ and $x \rightarrow y$ are in E_1 - P while $w \rightarrow x$ is in P. By the definition of E_1, T_v is traversed entirely before T_w, and T_x shares a node of T with T_w, which implies that the last vertex of T_x is visited after the last vertex of T_v. Since $x \rightarrow y$ is in E_1 - P, the first vertex of T_y is visited after the last vertex of T_x, which implies that T_v is traversed completely before T_y is visited. This creates a chord $v \rightarrow y$, contradicting our assumption that C is a minimum length cycle; this completes the proof of Claim 5.1.

Claim 5.2: E_3 is acyclic.

Let C be a shortest length cycle in E_3. C cannot contain consecutive edges from P, since P is transitive. Suppose that $w \rightarrow x$ and $x \rightarrow y$ are in E_3 - P. By the definition of E_3, DFS(T) will enter T_w, leave T_w, enter T_x, leave T_x, traverse

all of T_y, return to and finish traversing T_x, and return to and finish traversing T_w. This would create an edge $w \rightarrow y$ of E - P, contradicting our assumption that C is a shortest cycle in E_3.

This implies that there are edges $v \rightarrow w \rightarrow x$ of C such that $v \rightarrow w$ is in E_3 - P, and $w \rightarrow x$ is in P. DFS(T) must first enter T_v, leave T_v, and traverse T_w completely before returning to T_v.

Let anc(w) be the last node of T_v encountered by DFS(T) before T_w is visited; since this is a depth first search, anc(w) is an ancestor of every node of T_w. We claim that every node of T_x is a descendant of anc(w). Certainly the node of $T_x \cap T_w$ is a descendant of anc(w). T_x is connected, and there is only one path between nodes of a tree, so the only way that T_x can contain a nonancestor of anc(w) is if T_x contains anc(w). This would create an edge $v \rightarrow x$ (in which case C is not a shortest cycle) or $x \rightarrow v$ (which would imply $w \rightarrow v$ by transitivity of P, making an edge $v \rightarrow w$ of E_3 - P impossible); since neither is possible, E_3 cannot contain a cycle. This completes the proof of Claim 5.2.

Thus L_1, L_2, L_3 and L_4 are valid linear extensions. Since adjacencies between vertices are all reflected by these four lists, every chordal comparability graph is the comparability graph of a partial order with dimension at most 4. □

Kierstead, Trotter, and Qin [297] showed that some chordal comparability graphs need 4 linear extensions, in an elegant use of Ramsey theory. Since Ramsey theory is not one of the assumed prerequisites for this book, we will not cover the proof here. Note that, as is typical with Ramsey theoretic proofs, the proof requires an enormous (27^{27}+1) number of vertices before we can be sure that a 4 dimensional partial order in the class exists. I will not pose the question of finding the smallest chordal comparability graph which requires 4 linear extensions as a numbered open problem, since extremal problems are not the focus of this book; I certainly do not wish to imply that the smallest cycle-free partial order with dimension 4 is of this size!

We will discuss briefly the construction problem for finding a 4 dimensional representation of a chordal comparability graph. Recognizing 4 dimensional partial orders is NP-complete [492] , so we are forced to rely on an algorithm which finds a 4 dimensional representation only for chordal comparability graphs. We will discuss the time complexity involved in constructing the representation by following the proof of Theorem 5.2, doing several depth first searches on the chordal graph representation.

In chapter 15, we show that the intersection model for chordal graphs as subtrees of a tree can be found in O(n+m) time. We will show how to follow the proof of Theorem 5.2 to construct the representation in O(n^2) time. No explicit time bound for the construction is given in the original paper [356]. Recall that the algorithm takes the oriented graph, and adds edges to create four different lists as follows. In list 1 edges are added between 'type 1' nonadjacent vertices, with the edge $x \rightarrow y$ added if T_x is traversed completely before T_y, and edges in the reverse direction are added in list 2. Lists 3 and 4 add edges between 'type 2' nonadjacent vertices; in list 3 the edge $x \rightarrow y$ is added if T_y is traversed in the middle of the traversal of T_x, with list 4 reversing these edges. We will divide the algorithm into two parts. We first identify type 1 and type 2 nonadjacent vertices, and then find the appropriate ordering on each class. The two parts can easily be combined if one

is concerned with constant improvements in running time rather than simplicity of the construction.

For each vertex v of G, there are two nodes of the clique tree (this is the intersection model of subtrees of a tree; see chapter 15 for an explanation of the name) which we want to identify. These will be called v_1, the first node of T_v visited by DFS(T), v_∞, the last node of T_v visited by DFS(T). As we see in chapter 15, the tree representation of the chordal graph is a maximal clique tree, and the sum of the vertices in all maximal cliques of a chordal graph is $O(n+m)$. When we visit each vertex t of T during DFS(T) which corresponds to clique C, we look at every vertex c in C. Set c_∞ to the DFS number of t, and if c_1 is not assigned set c_1 to the depth first search number of t.

It is easy to identify type 1 nonadjacent vertices given these values. Step through the vertices of T in the order of DFS(T). Keep a list F of vertices which are finished with their traversals; F is originally empty. When a new node t of T is visited, label (f,v) as type 1 nonadjacent vertices for all f in F, all v such that $v_1 = t$, and f nonadjacent to v in G. Add all vertices with $t = v_\infty$ to F. Each type 1 pair is labeled once, so the total work in labeling is $O(n^2)$. Since there are $O(n)$ nodes in T, and all work besides labeling pairs done at each node is $O(n)$, type 1 pairs are identified in $O(n^2)$ time. All nonadjacent pairs of vertices which are not labeled type 1 are then assigned type 2.

Orientation of edges between vertex pairs of type 1 (type 2) is easy. For any pair x,y of type 1 (type 2) vertices, if x_1 is less than y_1 then add an edge from x to y in E_1 (E_3) and from y to x in E_2 (E_4).

Since all edges for E_1, E_2, E_3, and E_4 can be found in $O(n^2)$ time, and topological sort can be done in $O(n^2)$ time, we can construct the lists in $O(n^2)$ time.

The representation above is good if we are interested only in efficient storage space and adjacency testing. Nevertheless, we know that we are missing some key properties of chordal comparability graphs if they are stored in this way, since there are problems (in particular, the recognition problem) which can be solved more efficiently on chordal comparability graphs than on the more general comparability graphs of four dimensional partial order. This is intuitively unsatisfying; the representation may be good as a storage mechanism, but it does not get to the heart of what makes the class easy to deal with, as discussed in Open Problem 3.1.

We next consider the recognition problem for chordal comparability graphs. Although chordal graphs can be recognized in linear time, recognizing comparability graphs takes time proportional to matrix multiplication. Recall that this time complexity comes from a two step process; an orientation phase which takes $O(n+m)$ time, and a verification phase which takes time proportional to matrix multiplication. We will show how to use properties of chordal graphs to reduce the verification time to $O(n+m)$.

Let G be a chordal graph, and let G' be the result of running the transitive orientation algorithm on G. Let v_1 v_2 ... v_n be a perfect elimination ordering of G.

Process the vertices in perfect elimination order. Let x be the next vertex visited. We will take $O(|N(x)|)$ time, and either find a transitivity violation in G or determine that removing x will not change whether G' is a comparability graph. If all vertices are eliminated, we declare G to be a comparability graph.

Find all neighbors of x (both inneighbors and outneighbors) in the remaining graph. Let w be the vertex in this set which has an edge to x, and has smallest outdegree of any such vertex. Verify that w has an edge to all vertices in L which

have edges from x, and has edges from all other vertices in the set which have an edge to x. If these conditions hold, then remove x and continue with the algorithm.

It should be clear that if G' is a transitive orientation, we will never declare that a transitivity violation exists. Suppose that G is not a comparability graph. If the algorithm declares G to be a comparability graph, then at some point a vertex v_i was removed so that $v_i \ldots v_n$ induced a nontransitive graph, but $v_{i+1} \ldots v_n$ induces a transitive graph. There must be a transitivity violation $a \rightarrow v_i \rightarrow z$, where a and z both come after v_i in the perfect elimination ordering. However, our algorithm checks for an edge from a to w, where w is the vertex with smallest outdegree which has an edge to v_i, and that w has an edge to z. Therefore, if a does not have an edge to z, then the graph has a transitivity violation even after v_i is removed.

Since each vertex is eliminated in $O(|N(x)|)$ time, we can recognize chordal comparability graphs in $O(n+m)$ time; the first algorithm to achieve this bound is given in [268]. Exercises 5.12 and 5.13 show that a simple variant of lexicographic breadth first search creates a special form of perfect elimination scheme which allows linear time solutions for such problems as dominating set and maximum matching on this class of graphs.

5.3 Fill-in Schemes

A number of algorithms use a graph as a starting point, and add edges to the graph based on certain conditions. Fill-in schemes can be viewed as the reverse process of elimination schemes studied in chapter 12, though fill-in schemes are less well known than elimination schemes. The name is taken from the notion of fill in Gaussian elimination, where 'pivot' elements chosen add nonzero entries to the matrix, so that an originally sparse matrix can end the Gaussian elimination process with many nonzero elements filled in. Minimizing fill in this process is a well known problem; it was one of the original open problems in Garey and Johnson [206] and was proved NP-complete by Yannakakis [491]. Matrices which can be solved without fill can be modeled as chordal graphs (if M is symmetric) [413] or perfect elimination bipartite graphs [224]; Golumbic gives a good overview of these applications [223].

We discuss several notions of creating graph classes on the basis of fill. These classes may have a natural notion of original information to store with each vertex, but computing the edges of the final graph involves use of global information.

The Steinhaus graphs and bipartite Steinhaus graphs are examples of graph classes defined by a fill process. These classes are unusual in that each graph is defined by a string of n bits; thus the number of labeled Steinhaus graphs is only 2^n.

Consider an n by n symmetric matrix defined as follows. Place a 0 in each position (i,i). The first row of the matrix is filled with a string of n-1 bits given as input. Each remaining row in the upper diagonal is computed by setting the value M[i,j] equal to M[i-1,j-1] + M[i-1,j] mod 2; the lower diagonal is filled in to make the matrix symmetric. For instance, filling in a matrix defined by input 1 0 1 1

gives us the matrix

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

A graph is a Steinhaus graph if the adjacency matrix can be built in this way. One can also treat the arrays constructed as bipartite adjacency matrices; for that class, we fill in all entries (rather than just those above the main diagonal) using the set of rules. For either class, it may be natural to think of storing one bit for each vertex, but it is not possible to test adjacency using only the bit stored with each vertex. The adjacencies come from the 'fill-in' of the matrix.

Neither Steinhaus graphs nor bipartite Steinhaus graphs are a hereditary class of graphs. In fact, even though there are only 2^n Steinhaus graphs on n vertices, every graph is an induced Steinhaus graph (exercise 5.4). We will not investigate this class of graphs further, but this is an example of a graph class defined by a fill-in process.

I understand that some researchers of Steinhaus graphs are proud of the fact that the graphs have no practical applications; they arose from a homework problem posed by Steinhaus. I urge the gentle reader to refrain from inflicting on this class a host of optimization algorithms which are not suitable for a graph class with $2^{O(n)}$ members.

5.4 Closure Operations

A general technique for dealing with graphs is to use an operation called k-closure. In a single step, an edge is added between a pair of vertices x, y such that the degree of x plus the degree of y is at least k. The k-closure is formed by repeating this step until no new edges can be added. The closure operation was introduced by Bondy and Chvátal [53]; among other results, they showed that G is Hamiltonian if and only if the n-closure of G is Hamiltonian.

Given a particular G and k, it is not hard to see that the k-closure is uniquely defined (exercise 5.9). Therefore, for any graph class C with $f(n)$ members, we can define a new graph class with at most $f(n)$ members to be the k-closure of graphs in C. Even if C has an implicit representation, it is not clear that the new class will have an implicit representation, since adjacency between x and y in G' may depend on adjacencies between other sets of vertices in G. Thus, for any class of graphs which has an implicit representation, such as planar, degree bounded, or circular-arc graphs, we can define a new class which could be difficult to represent.

We should note that graph classes formed by closure are not necessarily hereditary. For example, consider the class of 10-closures of planar graphs. Clearly, K_5 is not in the class, since K_5 is not planar. However, K_5 is an induced subgraph of a graph in the class (exercise 5.10).

In another variant of the same idea, we can add an edge if the number of vertices adjacent to either x or y is at least k. The k'-closure G' of a graph G is the graph formed by repeating this operation until no new edge can be added. The k'-closure was suggested by Vijay Raghavan for use in fault diagnosis. Consider a graph with an edge between two processors if the processors have tested each other, and each declares that the other is faulty. If we want to find a diagnosis of faulty processors

with fewer than k faults, then if the joint neighborhood of x and y has at least k processors, at least one of x and y must be faulty. Thus, an edge can be added between x and y, and the process can continue until no new edges can be added. To find the faulty processors, we eventually want to solve a vertex cover problem on the k'-closure of the input.

5.5 Weakly Chordal Graphs

Another example of the use of fill-in as an algorithmic technique is recognition of weakly chordal graphs. Recall that a graph is weakly chordal if neither G nor its complement contains a chordless cycle of length at least 5.

The definition of weakly chordal graph implies an $O(n^5)$ algorithm, since one can determine whether an arbitrary graph has a chordless cycle of length at least k in $O(n^k)$ time; this can also be reduced slightly to $O(n^{k-3}MM)$ [441]. The best known algorithm for weakly chordal graph recognition works somewhat differently, based on a fill-in algorithm described below.

A two-pair in a graph G is a pair of vertices x, y such that every chordless path between x and y contains exactly two edges. [254] shows that every weakly chordal graph which is not a clique has a two-pair. In exercise 5.7, one is asked to show that if x, y is a two-pair, then G is weakly chordal if and only if $G + (x,y)$ is weakly chordal. Thus, the test for G being weakly chordal is to perform a fill-in procedure; G is weakly chordal if and only if the final result is a clique. Although it is not known how to find a single two-pair in less than $O(n^3)$ time, one can store enough information from previous steps so that each two-pair after the first can be found in $O(n^2)$ time, which gives an $O(n^4)$ algorithm for recognition of weakly chordal graphs [446]; see exercises 5.15 and 5.16. An alternative efficient recognition algorithm for weakly chordal graphs, developed in [36], is discussed in chapter 12.

It should be noted that it is not possible to add edges between all two-pairs in G at the same time; see exercise 5.14. In order to have a unique notion of fill-in, assume that pairs of vertices are checked in lexicographic order at each step to see whether they are two-pairs. This does not affect the time complexity of the algorithm; we merely want to define a unique result of the fill-in process.

If the fill-in procedure above is performed in one natural manner, as given in [446], the algorithm will run in $O(n^4)$ time but has several drawbacks. First, the algorithm does not produce a chordless cycle in G or its complement when the input is not weakly chordal; indeed, if the graph has no two-pair, we do not seem to get any information on where the hole occurs. Second, in order to find two-pairs efficiently, the algorithm uses $\Theta(n^3)$ space, keeping track of an entire graph representing possible two-pairs with each vertex x. These deficiencies are overcome in [256]; the new algorithm finds each two-pair after the first using $O(n^2)$ time and space, and identifies a hole in G or its complement if the input is not weakly chordal.

The relationship between the input graph and the graph which is the result of the fill process is not well understood. Suppose that we define a class of graphs as follows. G is in the class if G is the result graph of the fill process for some graph G' with maximum degree d. Storing G by storing the corresponding G' is a space efficient representation, and there is a natural way to store O(logn) bits at each vertex, but we cannot test adjacency in G efficiently at all (it seems to take $O(n^4)$

time to test a single adjacency), and adjacency testing is not local. Although it is not common to do so, any fill process can define a graph class in this fashion, and these may pose problems for the implicit graph question; depending on your operations, the graph classes may or may not be hereditary.

5.6 Other Fill-in Schemes

We saw above that we could take an implicitly representable class of graphs, and define an operation called closure so that it is unclear whether the class defined by closures of graphs in the class have an implicit representation. Many operations besides closure can be used to define similar classes. We do not intend to let the reader feel that any of the classes defined here is important in and of itself. Our intention is merely to show methods of defining classes of graphs which are challenging with respect to implicit representation. Hopefully, looking for representations of such classes could lead to new methods for finding implicit representations, or a disproof of the implicit graph conjecture.

For example, a number of problems use the k-neighborhood of a vertex x, meaning the set of vertices at distance at most k from x. This work is often phrased in terms of k-th powers of graphs; see [74] for basic facts regarding powers of graphs. For any graph G, we can define the k-neighborhood graph of G, in which x and y are adjacent if and only if there is a path of length at most k from x to y. Note that this particular class is not hereditary, but it is not clear whether, for example, 2-neighborhood graphs of path graphs would have an implicit representation. Many operations beside k-neighborhood can be used; we can define G' such that x and y are adjacent if $N(x)$ contains $N(y)$ in G, x and y are contained in a common nontrivial module, or any other global property.

We could also use optimization problems to define classes of graphs. To choose one random example, we could take a weighted set of vertices, and define x and y to be adjacent if V - x,y can be partitioned so that the sum of weights in each side of the partition is equal. If the weights are of polynomial size with respect to n, giving the weights gives a small space representation of the graph, but deciding whether x and y are adjacent cannot be solved quickly, so storing a graph in this form is not particularly satisfactory. Again, this is not necessarily a hereditary class; readers can construct their own classes based on their favorite decision or optimization problem.

5.7 Exercises

1) There are many variants of classes defined by intersection of paths in a tree; the tree can be undirected, rooted, or directed but not rooted, and paths can be defined as intersecting if they meet at a vertex or intersecting if they share an edge. Perhaps surprisingly, these variants have very different behavior with respect to recognition and optimization; we will look at some differences in the graph classes below. See [374] for a systematic study of these classes.

1a) G is a path graph if G is the intersection graph of paths on an undirected tree, where paths are defined to be intersecting when they share a common vertex. Show that every path graph is chordal.

1b) G is an EPT graph if G is the intersection graph of paths in a tree, where paths are defined as intersecting if they share a common edge. Show that EPT graphs are not necessarily perfect.

1c) G is a rooted directed path graph if G is the intersection graph of paths in a rooted tree, with intersection defined as in path graphs. Show that the 3-sun (a triangle, plus three independent vertices adjacent to each pair of vertices from the triangle) is a path graph but not a directed path graph.

1d) Show that all the classes defined above have implicit representations.

1e) Show that containment graphs of paths in a tree have an implicit representation.

2) Design an $O(n+m)$ transitive reduction algorithm for chordal comparability graphs.

3) Show that Steinhaus graphs are not hereditary.

4) Show that all graphs are induced Steinhaus graphs; that is, every graph is an induced subgraph of a Steinhaus graph. Use this to show that Steinhaus graphs do not have an implicit representation.

5*) (this problem is due to Ian Stobert) The natural method for determining whether two vertices i, j of a Steinhaus graph are adjacent if one is given the defining bit string would be to reconstruct the entire submatrix which can affect the entry; this can take $\Theta(n^2)$ time. Show that this adjacency can be computed in $o(n^2)$ time; Stobert in an unpublished manuscript designed a method which takes $O(n\log n)$ time.

6) Design an $O(n^k)$ algorithm to determine whether G contains an induced cycle of length at least k. If you are more ambitious, design an $O(n^{k-3}\text{MM})$ algorithm for solving the same problem, where MM represents the time to multiply two n by n matrices.

7) Let x, y be a two-pair in an arbitrary graph G. Show that G is weakly chordal if and only if $G + (x,y)$ is weakly chordal.

8) The algorithm for recognizing weakly chordal graphs repeatedly adds edges between some two-pair x,y; if G is weakly chordal, this leads to a clique. Show that if G is not weakly chordal, the final graph G' created is not unique, i.e. different choices of two-pairs will lead to different G' graphs.

9) Suppose that you start with a graph G, and construct a graph G' by choosing a pair of vertices x, y such that $|N(X) \cup N(y)| > k$ for a fixed k, and repeat this until no new edges can be added. Show that the same graph G' is created, independently of the order in which edges are added.

10) Show K_5 is an induced subgraph of 10-closures of planar graphs.

11) Note that k is not necessarily a constant for the following problem.

11a*) Design an algorithm to compute the k-closure of a graph in time which is linear in the size of the output list.

11b*) The following algorithm seems natural for computing the k-closure of a graph, in that it was discovered independently by a number of students in my course. Repeatedly choose and mark the unmarked vertex v with highest degree in the current graph, and add edges between v and all nonneighbors x of v such that degree(v) + degree(x) is at least k. Show that this algorithm does not compute the k-closure correctly.

11c) Design an $O(n^3)$ algorithm to compute the k'-closure of a graph; recall that the k'-closure differs from the k-closure in that an edge is added between x and y if the size of the joint neighborhood is at least k, as opposed to adding an edge if the degree sum is at least k.

12) A simple elimination ordering is an ordering of the vertices so that when x_i is eliminated, neighbors of x_i form a clique (i.e. x is simplicial), and the closed neighborhoods of the neighbors of x_i are ordered by set containment. Cardinality LBFS is the variant of LBFS discussed in chapter 13 such that a vertex of maximum degree in G from the last set is chosen to be last on the output list. Show that every cardinality LBFS of a chordal comparability graph is a simple elimination ordering. Graphs which have simple elimination orders will be discussed in chapter 12.

13a) Use exercise 12 above to design linear algorithms for the maximum cardinality matching and dominating set problems for chordal comparability graphs.

13b) Show matching is at least as hard as perfect bipartite matching for chordal graphs and for comparability graphs, and that dominating set is NP-complete for both classes.

14) Let G' be formed from G by adding an edge between a two-pair x, y of G.

14a) Show that it is possible to have a two-pair of G' which is not a two-pair of G.

14b) Show that it is possible to have a two-pair of G which is not a two-pair of G'.

15) Design an $O(n^3)$ algorithm for finding a two-pair in an arbitrary graph. Hint: for each vertex x, show how to determine whether x is part of a two-pair in $O(n^2)$ time; this approach appeared first in [22].

16*) The weakly chordal recognition algorithm works by repeatedly adding an edge between two-pairs; G is weakly chordal if this fill-in process produces a clique. Show that if the information computed in exercise 15 for each vertex x is maintained rather than being recomputed, it is possible to find a two-pair in the graph constructed by adding a single edge to an existing graph in $O(n^2)$ time. This approach gives an $O(n^4)$ algorithm for recognizing weakly chordal graphs, as described

in [446].

17) Using the result of exercise 15, explain how to recognize weakly chordal graphs in $O(mn^2)$ time.

18) Suppose that you take an arbitrary graph, and add edges between all pairs of vertices x, y such that x and y are both part of the same chordless cycle. Show that the resulting graph is not necessarily chordal.

19) The following characterization of chordal graphs using a fill-in scheme comes from [37]. Show that G is chordal if and only if G can be obtained from an independent set by successive addition of edges between vertex pairs x, y such that x and y are not connected by any chordless path of length greater than 2.

CHAPTER 6

Visibility Graphs

Visibility graphs are a popular class of graphs studied in the field of computational geometry, and pose some special challenges as far as finding a good representation. Although the number of graphs in the class is small, the natural geometric representation uses both real numbers to store vertices and requires global information to determine adjacency. The class of graphs is not hereditary, but a closely related hereditary class can be defined, which raises some new questions of interest.

Let P be a simple polygon in the plane. The visibility graph G of P is defined by associating a vertex with each endpoint of P such that (x,y) is an edge of G if the line between x and y is either part of the boundary of the polygon, or lies entirely within the polygon. To understand the definition, think of the polygon as walls of a room. Vertices can be thought of as people placed at corners of the room, and two vertices are adjacent if the corresponding people can see each other. Visibility graphs are important to the area of computational geometry; for more on the uses of this class of graphs, see [390], [431]. Recognizing visibility graphs in polynomial time is a major open problem, even if the order of vertices in the polygon is specified. In fact, it is not known whether the problem is in NP. A set of necessary, but not sufficient, conditions for an ordered set of vertices to be a visibility graph is given in exercise 6.3; an open problem related to one of these conditions is stated along with exercise 6.4.

Suppose that we try to represent the visibility graph in its most natural geometric form; at each vertex, we can store the coordinates of the corners of the associated polygon, and the next vertex on the polygon. Using this information, the graph can be reconstructed. However, the number of bits stored at each vertex needs to be large, so this form of representation is not space optimal. If we are also looking for an implicit representation, we have another problem, since we need to know all of the polygon edges before we can determine whether two vertices can see each other, even when their coordinates are known.

In the next sections, we study various interesting properties of visibility graph representations. We first show that the number of visibility graphs is $2^{O(nlogn)}$, by using Warren's theorem. However, the integers for representing coordinates must be at least exponential, leaving an important open problem about just how large these numbers must be. We will look at a form of representation that is good, though not optimal, with respect to space. We investigate the class formed if we change visibility graphs in a natural way to get a hereditary class. We discuss briefly an important optimization problem on visibility graphs, which is treated in more detail in chapter 14.

6.1 Counting Visibility Graphs

We can show that the number of visibility graphs is $2^{O(nlogn)}$ as a straightforward application of Warren's theorem.

Conceptually, variables of our set of polynomial equations correspond to coordinates of endpoints of the polygon. To make our counting argument simpler, we will slightly overcount the number of visibility graphs by not checking whether a set of variables corresponds to a simple polygon. Thus, we have $2n$ variables for the n vertices, since we need an x and y coordinate for each vertex.

We can create a polynomial equation for each pair of vertices u, v and each line L of the polygon, such that the value of the equation is negative if and only if the visibility from u to v is obstructed by L. Although constructing the appropriate equation is not trivial, I do not feel that I can improve on the description in the standard algorithms text [119]; the reader can consult that book to see how to write an equation which evaluates to a positive number if and only if two line segments cross. They also present an equation which evaluates whether two segments meeting at a point turn to the left or right at the point.

We need only one distinction in order to reconstruct the graph entirely from the sign pattern of this set of equations. Some works refer to an external visibility graph of a polygon, in which two vertices are adjacent if and only if the line between the vertices is entirely outside the polygon. If the line from u to v crosses no edge of the polygon, then u and v are adjacent either in the visibility graph or the external visibility graph. This can be distinguished by testing which way the lines of the polygon around u turn with respect to each other, and with respect to the segment (u,v).

Thus the number of visibility graphs is bounded by the number of sign patterns. Using the techniques of chapter 4, since we have $O(n^3)$ equations over $2n$ variables, the number of sign patterns is bounded above by $(cn^3/2n)^{2n}$ which is $O(2^{O(nlogn)})$.

6.2 Clique Cover Representation

Although the most natural form of representation for visibility graphs would be to use coordinates of the points, this is not useful if we are looking for a space efficient representation. Lin and Skiena [349] proved that even very simple visibility graphs require endpoints to have exponential sized integers. However, it is not known whether singly exponential sized integers are sufficient; the question is important because if we could guarantee that the number of bits in the integer is polynomial, then visibility graph recognition is in NP.

Open Problem 6.1: Can all endpoints of a visibility graph be assigned integer coordinates such that the integers use a polynomial number of bits?

There is a natural form of storage studied by Agarwal, Alon, Aronov, and Suri [5] which uses a relatively small number of bits to store a visibility graph. We will study their representation in this section. We note that the representation has its drawbacks; it is neither space optimal, nor can we test for adjacency in constant time, but it is the most significant reduced space representation which is currently known.

The authors of [5] consider the problem of representing a visibility graph as a covering set of cliques and complete bipartite graphs so that every graph in the set is a subset of G, and every edge is contained in at least one of the graphs of the

covering set. Clearly, G can be retrieved from this representation. They measure the size of the representation as the total number of vertices in all the complete bipartite graphs which cover G; in our analysis, we must multiply by a factor of $\log n$ since we are concerned with the actual number of bits in the representation. The fact that the covering set of a visibility graph has small size also leads to efficient algorithms for counting the number of edges in a visibility graph when the polygon is given as input, and finding the longest diagonal of a polygon [7], [8].

The paper [5] describes an algorithm which constructs a covering set which has $O(n\log^4 n)$ bits. It is not clear whether the algorithm ever can construct a set which uses this many bits, however. They also give a lower bound on the size of the cover; they show that there are visibility graphs which have no cover (whether found by their algorithm or not) using $o(n\log n)$ vertices. We note that while it is theoretically possible to have covering sets including a total of $\Theta(n\log n)$ vertices which can be stored in $o(n\log^2 n)$ bits, this lower bound also can be modified to show that any covering set requires $\Omega(n\log^2 n)$ bits on some visibility graphs. Given the gap between upper and lower bounds on this natural form of representation, we have a number of open problems.

Open Problem 6.2: Give a tight bound (with respect to order notation) on the number of bits used in an optimal covering set of a visibility graph.

Open Problem 6.3: Can you find a covering set which matches the above bound in polynomial time?

Note that problem 6.3 is potentially interesting both if we are given the input as a polygon, and if we are given an adjacency matrix of a visibility graph. We would also like tight bounds on the number of bits used for covering sets by the algorithm in [5].

We will not cover the algorithm for constructing the covering sets in detail. The algorithm works roughly as follows. Find a visibility line which divides the polygon into two pieces of approximately equal size using a theorem of Chazelle which will be described in the next section. Find covering sets for edges which are entirely contained within one of the subpolygons recursively. Use a specialized algorithm to cover all edges of the bipartite graph consisting of edges which go from one side of the polygon to the other; this algorithm creates a cover which can be stored using $O(n\log^3 n)$ bits. Standard analysis of recurrences shows this representation uses $O(n\log^4 n)$ bits. It is interesting to note that the covering sets used are all bipartite. This algorithm requires the model as input, which brings up the following open problem.

Open Problem 6.4: Find a small covering set of a visibility graph in polynomial time, if the input is in the form of an adjacency matrix.

We now turn our attention to the clever lower bound argument on the size of the covering set from [5]. Consider a graph H formed by subtracting a perfect matching M from $K_{n,n}$. It is not difficult to show that H is a subset of a visibility graph on $O(n)$ vertices (exercise 6.8).

Theorem 6.1 [5] *Let H be $K_{n,n}$ minus a matching. Let S be a set of complete bipartite subgraphs of H such that every edge of H is contained in at least one subgraph of S. The total number of vertices over all graphs in S is $\Omega(n\log n)$.*

Proof We will call the members of one color class x_1,..., x_n and the other class y_1,...,y_n, with each x_i nonadjacent to y_i. Each member of S contains at most one vertex from each pair x_i,y_i.

Let $|S|$ be the number of graphs in S. Randomly assign the value 0 or 1 to each graph in S. Let E_i be the probability that every graph of S which contains x_i has value 0, while every graph of S which contains y_i has value 1. We will say that an assignment of 0s and 1s to graphs in S contributes to E_i if the above condition is met.

Note that a particular assignment of 0s and 1s cannot contribute to both E_i and E_j. Since (x_i, y_j) must be covered in some graph G of S, G must be labeled 0 in order to contribute to E_i, while G must be labeled 1 to contribute to E_j. It is not hard to see that this implies that the sum of the E_i for a given covering set is at most 1, since each of the $2^{|S|}$ assignments of 0s and 1s contributes a total of at most $1/2^{|S|}$ to the sum.

If covers(i) is the number of graphs of the cover containing x_i or y_i, clearly $E_i = 1/2^{covers(i)}$. Thus, $\sum_{i=1}^{n} 1/2^{covers(i)} \leq 1$.

There are n parts of the sum above, which implies that the average value of $1/2^{covers(i)}$ is at most $1/n$; informally, this means that the average number of graphs in the cover set which contain x_i or y_i is at least log$n$, so the total size of the cover is $\Omega(n\log n)$. This can be made more precise by arguing that for at least $n/2$ values of i, $1/2^{covers(i)}$ is no larger than $2/n$, and following the argument above. Therefore, the total number of vertices over all graphs in the cover is $\Omega(n\log n)$. □

Finally, we might also think of covering the edges with more general subgraphs than cliques or complete bipartite graphs; these are chosen because cliques and complete bipartite graphs on k vertices can be described using $O(k\log k)$ bits. In chapter 14, we will see that there is an efficient method for storing the graph induced by the neighborhood of any 'convex fan' in a visibility graph; this might make it interesting to study the smallest cover of a visibility graph with cliques, complete bipartite graphs, and neighborhoods of convex fans.

6.3 Induced Visibility Graphs

In this section, we study the class of graphs formed if we force the class of visibility graphs to be hereditary in the most natural way. A graph is an induced visibility graph if there is a simple polygon P such that vertices of G map to a subset of vertices of P, and x is adjacent to y iff x and y are neighbors in P, or the line from x to y is entirely within P.

Previous work on induced visibility graphs includes [431], which showed that a number of classes of graphs were not contained in the induced visibility graphs, and [180], which shows that induced visibility graphs are not characterized by a finite set of forbidden induced subgraphs. The problems of counting, recognizing, and representing induced visibility graphs are open. Partial results on counting the number of induced visibility graphs are presented in this section.

With a few exceptions, researchers have had difficulty relating problems on visibility graphs to problems on other well known classes of graphs. In the opinion of this author, this difficulty comes in part from the focus on visibility graphs rather than on induced visibility graphs. For example, the graph on three vertices in figure 6.1-a is not a visibility graph, because every visibility graph must be Hamiltonian.

Figure 6.1 An Induced Visibility Graph which is not a Visibility Graph

Therefore, any graph class which contains this simple graph is not contained in the visibility graphs. However, if we think of visibility between points on a polygon as a representation scheme, we have the power to represent this graph as in figure 6.1-b.

Let n be the number of vertices of an induced visibility graph G. If we hope to use a polygon for representing G, we must hope that the polygon does not need too many extra points. If we are worried about the space complexity of a representation, a polygon with n^2 points would be of no use. We will show that G can be represented on a polygon with at most $O(n\log n)$ points. Using Warren's theorem, this will give an upper bound of $2^{O(n\log^2 n)}$ for the number of induced visibility graphs on n vertices. There is no matching lower bound on the number of induced visibility graphs on n vertices.

Let P be any polygon which represents an induced visibility graph G. We will find another polygon P_2 which represents the visibilities between vertices of G correctly, such that P_2 has $O(n\log n)$ points. The points of P which correspond to vertices of G will have the same coordinates in P and P_2.

Draw lines between all pairs of adjacent vertices from G in P, as in figure 6.2, where the polygon is ordered numerically, with certain points which are not part of G but are in the polygon marked by x. These visibility lines divide the plane into regions. Remove all lines which do not touch the outer region; in figure 6.2, this will remove the line from 4 to 6 and the line from 2 to 5. Note that there are some lines, such as the line from 1 to 3, which touch the outer region at several segments. The outer envelope E of the visibility line segments is formed by eliminating all segments except for those on the outer region.

We can get a polygon which represents G as an induced visibility graph by adding one point for each endpoint of the outer envelope, and one point for each segment of the outer envelope, with points of the second type being infinitely close to the corresponding line of the outer envelope. All visibility lines between adjacent vertices are contained within E, and thus will be contained within the new polygon, since all polygon edges are in the outer region. Consider any nonadjacent pair of vertices x, y. The line between x and y cannot be entirely contained in E; any polygon edge which would cut the visibility between x and y would force some edge of the visibility envelope to be outside the polygon. Thus, the edge between x and y must leave the visibility envelope at some point, between points a and b of E. Since a point is introduced infinitely close to the line between a and b in E, the line from x to y goes outside of the new polygon at this region.

Therefore, if we want to bound the number of points in P_2, we need to get a bound on the number of segments in the outer envelope. We will show that the

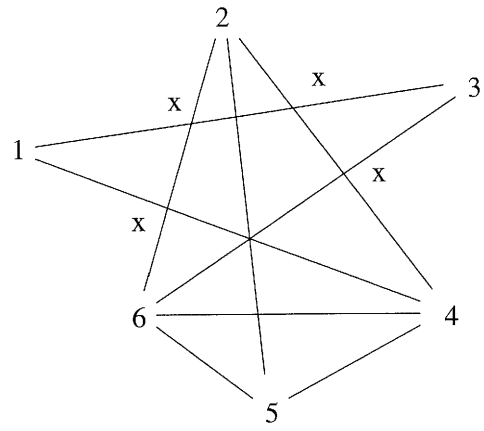


Figure 6.2 Visibility Lines in an Induced Visibility Graph

number of lines which participate in the outer envelope is O($n$log$n$), and invoke a theorem which allows us to say that the number of segments in the outer envelope is O($n$log$n$).

The same bound can be obtained by invoking a more general theorem from [23], which states that for any arrangement of n line segments with at most h distinct endpoints, the complexity of every face is O($h$log$h$). The argument here does not need to rely on the rather powerful theory of Davenport-Schinzel sequences [9] to yield their more general result.

We will get this result using a simple divide and conquer approach. We divide the polygon P into 2 pieces, and argue that the number of edges of the outer envelope which go between the two sides is O(n). If we can divide vertices of G equally between the two pieces of P, and use the same argument recursively, the number of lines of the outer envelope is bounded above by the recurrence E(n) = 2E($n/2$) + O(n), which works out to O($n$log$n$). We will not be able to divide the polygon into pieces of exactly equal size, but we will be able to divide P into reasonably balanced pieces, which is enough to make the recursive approach give a bound of O($n$log$n$).

Suppose that we divide P along some line l, where l is completely inside the polygon. Call the two pieces of the polygon formed by adding l P' and P''. Any line on the visibility envelope E either crosses l, or is on the visibility envelope of P' or P''. We will show that the number of lines of E which cross l is O(n).

For each vertex of G, consider the 2 lines which go to the other half-polygon which are furthest clockwise, and furthest counterclockwise. We claim that all visibility lines on the outer envelope which go between the two half-polygons come from this set of (at most) $2n$ lines. If a line between x in P' and y in P'' is not one of the lines chosen, it is completely enclosed by the 2 lines from x and the 2 lines from y; an example is shown in figure 6.3. The portion of the line which travels inside P' is contained within the two bounding lines from x, while the segment within P'' is contained in the two lines out of y.

To make the recursive argument work, we must show that there is a line l which can cut the polygon so that vertices of G are split roughly equally between P' and

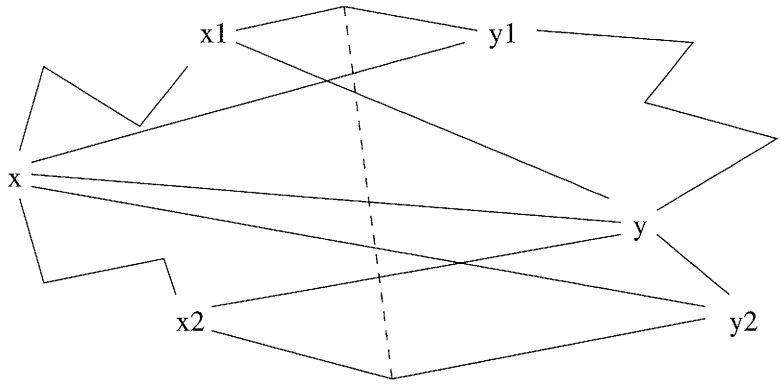


Figure 6.3 The Line from x to y is not on the Outer Envelope

P''. Note that this can be different than finding a line which splits P into equal size pieces, since vertices of G may form only a small fraction of the vertices in P.

Fortunately, we can use a powerful theorem due to Chazelle [99] to find a line which cuts the polygon. Given any polygon with weights assigned to the points of the polygon, there is a line which separates the polygon into two subpolygons, such that the sum of the weights of vertices in each subpolygon is at least 1/3 of the total weight of the points. For our problem, we give weights of 1 to vertices of G in P, and weight 0 to any other points of P; Chazelle's theorem tells us that we can choose l so that both P' and P'' contain at least $n/3$ vertices.

Let E(n) be the number of lines which touch the outer envelope of visibility lines. We know that E(n) $\leq$ E(n_1) + E(n_2) + 2n, where $n_1 + n_2 = n+2$ and both n' and n'' have size at most $2n/3$. It is not difficult to prove inductively from this equation that E(n) is O($n$log$n$).

Although O($n$log$n$) lines touch the outer envelope, it is not immediately obvious that P has O($n$log$n$) points, since a single line may have several segments which touch the outer envelope, and we add a point to P for each segment. The paper [15] shows that the boundary of a set of k segments with O(k) distinct endpoints is of size O(k), which gives us the bound needed for our result.

In summary, we know that any induced visibility graph on n vertices is an induced subgraph of a visibility graph on O($nlogn$) vertices . Since there are $2^{\Theta(nlog^2n)}$ visibility graphs on this number of vertices, it is easy to see that there are $2^{O(nlog^2n)}$ induced visibility graphs on n vertices. However, it is by no means clear that this bound is tight.

Open Problem 6.5: What is the correct function f(n) such that the number of induced visibility graphs on n vertices is $2^{\Theta(f(n))}$?

Clearly, f(n) lies somewhere between $\Theta(n\log n)$ and $\Theta(nlog^2n)$. If the correct answer is $n$log$n$, then the question of whether an implicit representation of visibility graphs can be found is particularly interesting. Resolving the question would allow us to know how much we could hope for in a space optimal representation. Of course, finding a generalized implicit representation is also open for this class of graphs.

6.4 Optimization on Visibility Graphs

The key result on visibility graph optimization is presented in chapter 14, which studies a polynomial time algorithm for the maximum clique problem. This is the most well known (to the author) example of a problem on a graph class which can be solved efficiently if the model is given, but is open if the input was in the form of an adjacency matrix; such problems are discussed at length in chapter 14. The algorithm also can be used to solve the clique problem on induced visibility graphs, if the model is given as input; it is also solvable if only the vertex ordering of the polygon is given, rather than the entire model.

Of somewhat lesser interest is the fact that a Hamilton circuit can be found in a visibility graph if the representation is given; this is an example of a trivial problem, since the answer to the Hamilton circuit question is always yes. The question of finding a Hamilton circuit if a visibility graph is given in adjacency list form is still open.

The obvious question is whether the efficient algorithms are due to the representation or the special properties of the graph class. The major open question, of finding a polynomial time algorithm for the clique problem on visibility graphs and induced visibility graphs if the input is in adjacency list form, is given elsewhere; we pose some related questions here.

Open Problem 6.6: Find a polynomial algorithm for the Hamilton Circuit problem on induced visibility graphs, or show that the problem is NP-complete on the class.

The problem above is open whether the model is given or not.

Lin and Skiena [349] study the complexity of a number of problems on visibility graphs. For example, they show that the independent set and dominating set problem are NP-complete on the class, and that isomorphism is isomorphism complete. However, the following problem is left as an open problem in that paper.

Open Problem 6.7: What is the complexity of the chromatic number problem on visibility graphs, both for the case when the model is given as input and for the problem when input is in the form of an adjacency matrix?

6.5 Line of Sight Graphs and Other Variants

Visibility representations are studied for other classes of objects besides polygons. In the more general problems, vertices map to geometric objects, and two vertices are adjacent if and only if there is some straight line (sometimes these lines are restricted to horizontal and/or vertical lines) between points on the corresponding objects which does not hit any object corresponding to other vertices. We will refer to a representation of this form as a *line of sight* representation, and the corresponding graphs as line of sight graphs for the model. See [30] for a discussion of some graph classes defined by this type of representation.

In one such class, sometimes called bar-representable graphs, vertices are mapped to horizontal segments on the plane with distinct y coordinates, and visibility lines must be vertical. For our purposes, this class is interesting because we no longer have the problem of representing endpoints efficiently for this class. Nevertheless, the natural representation is not implicit, since adjacency between vertices depends on which other lines can obstruct the visibility.

The class defined above has an implicit representation for a different reason. Every such graph is a planar graph [488], and thus has an implicit representation.

Visibility graphs and line of sight graphs can also be generalized to higher dimension. For example, the paper [61] generalizes the class above in a natural way to a higher dimension; vertices correspond to rectangles floating in three dimensional space parallel to the xy-plane, with edges represented by vertical lines of sight. This class contains all planar graphs and all complete bipartite graphs, but cannot contain arbitrarily large cliques (they show how to represent K_{22} and that K_{56} cannot be represented).

These generalizations deserve more study; I do not know of any work on the natural generalization of visibility graphs to higher dimensions. For example, I do not see any reason that the class defined above necessarily has an implicit representation, even though it has a representation using a small amount of space; storing the endpoints of all rectangles is sufficient to reconstruct the graph, but to test adjacency you need to know the endpoints of other rectangles as well. Thus, I propose the following open problem, though since the class is very new there may be a simple solution.

Open Problem 6.8: Is there an implicit representation for the three dimensional line of sight graphs defined above?

6.6 Exercises

1) Show that the number of maximal cliques in a visibility graph can be exponential.

2) Although it seems most natural to consider lines of visibility to be straight, visibility 'pseudolines' have also been considered. Pseudolines are not necessarily straight, but any pair of pseudolines can cross at most once. Although it is not known whether there is a set of $\Omega(n\log n)$ visibility lines all of which touch the outer envelope, show that there is a set of $\Theta(n\log n)$ visibility pseudolines between n points such that each touches the outer envelope. A graph with $\Omega(n\log n)$ edges such that each edge can be drawn as a pseudoline which touches the outer envelope is described in exercise 14.18.

3) This problem deals with some well known necessary conditions due to Ghosh [218] for a Hamilton circuit $h_1\ h_2\ \ldots\ h_n$ to be the outer border of the polygon representing a visibility graph. For each of a,b,c, you may solve either 1) the example question; show a graph and ordering which violate the condition 2) the proof question; show that the condition is necessary 3) the algorithm question; design and analyze an efficient algorithm to test for the condition, or 4) the induced visibility graph question: does the property always hold for induced visibility graphs?

3a) An ordered cycle is a cycle in which vertices are in increasing order. Ghosh showed that every ordered cycle of length at least 4 in a visibility graph contains at least k-3 diagonals.

3b) A blocking vertex for a nonadjacent pair of vertices (i,j) is a vertex b such that all vertices between i and b are nonadjacent to all vertices between b and j on the simple path from i to b to j along the Hamilton cycle. Ghosh showed that every

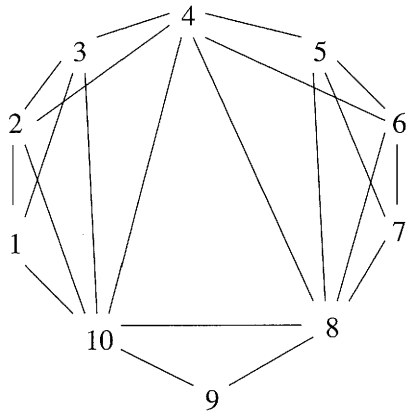


Figure 6.4 A Counterexample to Ghosh's Conditions

invisible pair in a visibility graph must have a blocking vertex.

3c) A nonadjacent pair of vertices x,y is called minimal if they have exactly one blocking vertex. Two pairs u,v and x,y are separable if one can travel from u to v on the Hamilton cycle and avoid both x and y. Ghosh showed that two separable minimal nonadjacent pairs of a visibility graph must contain distinct blocking vertices.

3d*) Show that the graph of Figure 6.4 satisfies Ghosh's conditions, but is not a visibility graph. This counterexample was discovered by Everett and Corneil [180].

4) A graph with an ordering is ordered chordal if every ordered cycle of length at least 4 has a chord. Give a polynomial time algorithm for determining whether an ordered graph is ordered chordal. The following problem seems to be open and very natural:

Open Problem 6.9: Can you determine in polynomial time whether a graph has an ordering so that every ordered cycle has a chord? The same question can be asked of the other two conditions, as well as the problem of counting the number of graphs which satisfy some subset of these conditions; we will return to condition 3c in exercise 14.22.

5) A graph is k-connected if removing any set of at most k-1 vertices leaves a connected graph. Coullard and Lubiw [131] showed that if G is a 3-connected visibility graph, there is an ordering of the vertices so that the first 3 vertices form a clique, and every vertex added is adjacent to some 3-clique of old vertices. Such an ordering can be used to recognize whether there is a polygon P such that G is the visibility graph of P, and all distances between visible points are equal to those given in an input matrix M; fundamentally, adjacency to a 3- clique fixes a point in space.

5a) Design an algorithm to find such an ordering in polynomial time, and analyze its complexity (you should be able to design an $O(n^7)$ algorithm).

5b) Use the above property to show that K_5 - (1,4),(3,5) is not a visibility graph.

6) Show that the number of external visibility graphs is $2^{O(nlogn)}$.

7) The following is an example of a type of problem called an art gallery problem, which is closely related to visibility graphs. For a deeper study of such problems, see [390], [431]. In these problems, a polygon is viewed as the walls of an art gallery, and we consider placing guards at points so that some guard can observe every point of the gallery wall (where a picture may be hanging). There are many variants, depending on the restrictions on the shape of the gallery, and placement and forms of movement allowed for the guards. Suppose that we have a simple polygon, and we want to place stationary 'guards' immediately inside polygon endpoints so that every point on the wall of the gallery can be seen by at least one guard.

7a) Show that for every n, there are polygons which require $\lfloor n/3 \rfloor$ guards, where n is the number of endpoints of the polygon.

7b) Show that every polygon with n points can be guarded by $\lfloor n/3 \rfloor$ guards. This problem, first posed by Klee, was originally solved by Chvátal [103]; for your assignment, you may assume that every triangulation of the plane can be colored with 3 colors.

8) Show that $K_{n,n}$ - a matching, i.e. the complete bipartite graph on $2n$ vertices in which every degree is exactly equal to n-1, is an induced subgraph of a visibility graph on O(n) vertices.

9) Show that $K_{n,n}$ - a matching has a covering set of cliques and complete bipartite graphs containing a total of O($n$log$n$) vertices.

10) This problem comes from [180]. Every visibility graph has an ordered chordal ordering, as described in exercise 6.4. Show that the Grötzsch graph of Figure 6.5 has no ordered chordal ordering, and thus is not an induced visibility graph.

11) A vertex is called convex if the polygon edges touching the corresponding point form an angle of less than 180 degrees. The author conjectured that the neighbors of every convex vertex in a visibility graph must induce a weakly chordal graph; that is, a graph with no induced cycle of length greater than 4 in G or its complement.

11a) Show that it is possible to have a convex vertex v in a visibility graph such that N(v) does not induce a weakly chordal graph.

11b) The conjecture proved false above was made because there are polynomial time algorithms for solving the clique problem on weakly triangulated graphs. Show that if the conjecture had been true, then there would have been a polynomial algorithm for solving the maximum clique problem on visibility graphs even if the model was not given as input. We note that this is not completely trivial, since removing a vertex from a visibility graph does not necessarily give a smaller visibility graph.

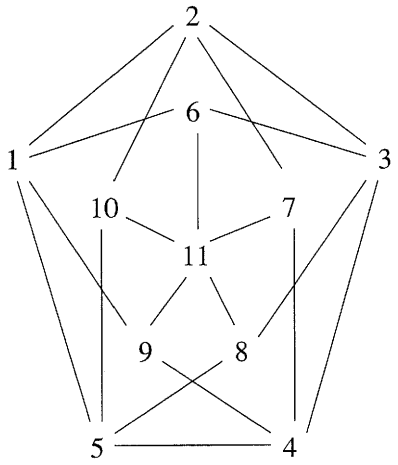


Figure 6.5 The Grötzsch graph

12*) Prove or disprove: if G is a visibility graph with at least four vertices, there is a vertex x in G such that G - x is a visibility graph.

CHAPTER 7

Intersection of Graph Classes

Many interesting classes of graphs have come from requiring graphs to have two separate properties. We will study representation of a number of graph classes defined in this way, and introduce open problems relating to some of these classes.

When one takes the intersection of two graph classes, one does not know whether the number of graphs will stay approximately the size of the smaller class, or whether we will get far fewer graphs than we have in either class individually. When we get a much smaller class, it can be challenging to find a good representation of the new class, since we cannot use the representations of the original graph classes. Therefore, classes defined by intersection of several required properties are good challenges for the implicit graph question.

Many graph classes defined by intersection of larger classes have been studied in the literature. There are papers on intersection of classes defined by pairs of forbidden subgraphs (such as square and trumpet free graphs), classes defined by global properties (such as chordal co-comparability graphs), intersections of global and local properties (cubic planar graphs), and many more. This book can only study a small fraction of classes of this type. We choose to cover a 'consistent' set of classes, which come from a variety of approaches for dealing with interval graphs. We will not repeat material which is well covered in Golumbic [223].

7.1 Some Fundamental Properties

One of the oldest and most satisfying techniques for working with a class C of graphs is to note a property or set of properties which the class obviously has, and to determine that the properties completely characterize C. Euler took the obvious observations that Eulerian graphs (without isolated vertices) were connected, and had no vertex of odd degree, and showed that these two properties exactly characterized Eulerian graphs. Many other useful characterizations have been discovered in this way.

Suppose that we view interval graphs with this approach in mind. One simple property is that interval graphs cannot contain any chordless cycles of length greater than 3, i.e. every interval graph is chordal.

Two other obvious properties of interval graphs are used as parts of characterizations. For any three independent vertices in an interval graph, one must be placed in the middle of the other two, and every path from the first to the third must pass through a neighbor of the middle vertex. A trio of independent vertices is called an asteroidal triple if for each pair of vertices, there is a path between the two which avoids all neighbors of the third. Thus, every interval graph is asteroidal triple-free. We will sometimes use the shortened form AT-free to refer to the property of having no asteroidal triples.

It is also natural to assign an ordering between nonoverlapping intervals, saying that one interval comes before another interval. This relationship is clearly transitive; if x comes before y and y comes before z, then x comes before z. Since these assignments are made exactly between nonadjacent vertices of the corresponding interval graph, we are saying that the complement of an interval graph has a transitive orientation, i.e every interval graph is a co-comparability graph.

It turns out that these simple properties of interval graphs can be used to completely characterize interval graphs. Lekkerkerker and Boland [341] showed that a graph G is an interval graph if and only if G is chordal, and G is asteroidal triple-free. Gilmore and Hoffman [220] showed that G is an interval graph if and only if G is chordal, and G is a co-comparability graph. In general graphs, the class of asteroidal triple-free graphs properly contains the class of co-comparability graphs; see exercises 7.1 and 7.2.

The number of comparability graphs (and thus the number of co-comparability graphs as well) is $2^{\Theta(n^2)}$, since every bipartite graph is a comparability graph. Therefore, the implicit graph question is not relevant for these classes. Recognition of comparability graphs is discussed in chapter 11; to summarize, a transitive orientation of a comparability graph can be found in linear time, but the fastest recognition algorithm for the class takes time proportional to matrix multiplication.

Asteroidal triple-free graphs were first studied as an independent class of graphs in [126], and a number of interesting properties of the class have been found [125], [310]. Let us look briefly at the problem of recognizing asteroidal triple-free graphs, before we deal with combinations of the properties studied here. The reader is asked to provide an $O(n^3)$ algorithm for recognizing asteroidal triple-free graphs in exercise 7.3; the best current time bound for recognizing AT-free graphs is $O(n^{2.82})$, using fast matrix multiplication algorithms as a subroutine [332]. We will show that recognizing asteroidal triple-free graphs is as hard as recognizing graphs without an independent set of size three; thus, any algorithm which did better than matrix multiplication for recognizing graphs in this class would have far-reaching implications.

Let G be a graph with at least four vertices, and suppose that we want to test whether G has an independent set of size 3. Create graph G' by adding n vertices to G, where each v_i is adjacent to every vertex of G' with the exception of the ith vertex of G. If G has an independent set v, w, x of size 3, then G' has an asteroidal triple, since some new vertex is adjacent to each pair of vertices from v,w,x. No new vertices of G' can be in an independent set of size 3, so if G' has an asteroidal triple v, w, x, then v, w and x must form an independent set of size 3 in G. The problem of determining whether G has an independent set of size 3 has been reduced to the problem of determining whether a graph on $2n$ vertices is asteroidal triple-free, which implies that any $O(n^k)$ algorithm where $k < \alpha$ (the exponent in matrix multiplication of two n by n matrices) would improve the time complexity of determining whether a graph has an independent set of size 3.

Thus, finding an $O(n^2)$ algorithm for recognizing asteroidal triple-free graphs may be a very difficult problem. As an open problem, the following question seems more tractable.

Open Problem 7.1: Can you find an algorithm which recognizes asteroidal triple-free graphs in time proportional to matrix multiplication?

A related open problem asks whether there is a good 'certificate' that an input graph is AT-free. A certificate for a property is a proof that the input has the property; we do not care about the time needed to find the proof, but only about the time to verify that the proof is correct. At this point, there seems to be no certificate which can be verified in $o(n^{2.82})$ time. On the other hand, there is no reason to assume that a very good certificate does not exist; note that even the apparently much harder Hamiltonian graph recognition problem has a certificate which can be verified in $O(n)$ time.

The independent set problem was shown to be polynomially solvable for AT-free graphs in [81]; the best current time complexity is $O(n^3)$ [316]. The clique problem is NP-complete on the class [81]. At this point in time, coloring and Hamilton cycle are open for asteroidal triple-free graphs.

In addition to these properties, we will consider classes defined using an important generalization of chordality. Recall that G is weakly chordal if neither G nor the complement of G contains a chordless cycle of length greater than 4. Thus, chordless 4-cycles are allowed in weakly chordal graphs, but longer chordless cycles are not. Forbidding longer chordless cycles in the complement as well makes it possible to solve optimization problems on the graph efficiently [254, 446] and weakly chordal graphs are a subclass of the perfect graphs [253]. In exercise 7.5, the reader is asked to show that chordal graphs are a subclass of weakly chordal graphs. The fastest known recognition algorithm for weakly chordal graphs takes $O(n^4)$ time [446]; one such algorithm is discussed in chapter 5 and another in the recognition chapter.

7.2 Intersections of Fundamental Properties

Combinations of chordality, co-chordality, comparability and co-comparability are well covered in Golumbic [223], with the exception of chordal comparability graphs which we treat in chapter 5. We will summarize the results very briefly before discussing generalizations of the classes. Golumbic devotes separate chapters to each of the classes given below. All of these classes have been studied extensively with respect to optimization problems, as is discussed in the survey of graph algorithms at the end of this book.

G is a split graph if the vertices of G can be partitioned into an independent set and a clique. Split graphs can be obtained from bipartite graphs by making one color class into a clique. It is easy to see that a split graph and its complement are chordal; Földes and Hammer [196] showed that G is a split graph if and only if G is both chordal and co-chordal. This gives an immediate linear time algorithm for recognizing split graphs; see [223] for a recognition algorithm which takes $O(n)$ time if the degrees of all vertices are given as input.

Permutation graphs were introduced in chapter 3, where we defined them to be intersection graphs of line segments which go between two parallel lines. G is a permutation graph if and only if G is both a comparability graph and a co-comparability graph. It is easy to see that these are necessary conditions for permutation graphs. The first statement of the sufficiency of the conditions can be attributed to several authors. I prefer to cite Dushnick and Miller [166]; this is the oldest source, though technically the observation was made on the directed version of the graphs, known as two dimensional partial orders. Permutation graphs can

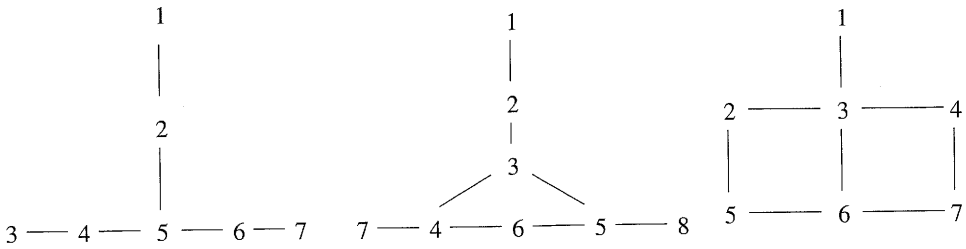


Figure 7.1 Forbidden Subgraphs

be recognized in linear time, as discussed in chapter 11, though this is much more difficult than the linear time recognition of other classes.

The fact that G is an interval graph if and only if G is a chordal co-comparability graph was first proved in [220]. Linear time recognition of interval graphs is discussed in chapter 13.

In this chapter we look at what happens as we generalize these classes defined by pairs of properties by generalizing from chordality to weak chordality, and from co-comparability graphs to AT-free graphs.

Before dealing with these generalizations, let us briefly consider classes obtained by taking the intersection of the fundamental properties and bipartite graphs.

Chordal graphs which are bipartite clearly correspond to forests, while bipartite comparability graphs are the same as bipartite graphs. Bipartite co-comparability graphs are equal to bipartite permutation graphs (exercise 7.6), which are discussed in more detail in chapter 9. Bipartite co-chordal graphs are discussed in exercise 7.7. Weakly chordal bipartite graphs are equal to the class called chordal bipartite (bipartite and no chordless cycles of length > 4) discussed in chapter 9 (exercise 7.8). Bipartite graphs are always co-AT-free, since the complement has no independent set of size 3.

The hardest relationship to see when requiring bipartite graphs to have one of these properties is the equivalence between bipartite AT-free graphs and bipartite permutation graphs. Every permutation graph is a co-comparability graph, and co-comparability graphs are a subclass of AT-free graphs, so it is clear that bipartite permutation graphs are a subset of bipartite AT-free graphs. The simplest way to show the equivalence is to use a forbidden subgraph characterization of comparability graphs due to Gallai [205]; the forbidden subgraphs can also be found in [461]. If G is not a co-comparability graph, then G is either not bipartite, G contains a chordless cycle of length at least 6 or G contains one of the three graphs in Figure 7.1. Since the graphs of Figure 7.1 and long cycles have asteroidal triples, every bipartite graph is either a co-comparability graph or has an asteroidal triple. I do not know of a simple direct proof of the equivalence of these classes, though I imagine that such a proof must exist.

7.3 Weakly Chordal Comparability Graphs

Our first generalization takes the characterization of interval graphs as chordal co-comparability graphs, and replaces the chordality condition with a requirement that the graph is weakly chordal. Since the complement of every weakly chordal graph is weakly chordal, the complement of this class is the class of weakly chordal

comparability graphs. We will show that recognition, representation, and optimization problems on this class can be effectively transformed to the same problem on the subclass of chordal bipartite graphs. Recall that a chordal bipartite graph is not the same as a chordal graph which is bipartite (this would be a forest), but is instead a bipartite graph such that every cycle which can have a chord, i.e. every cycle of length at least 6, has a chord.

Let G be a directed graph. Consider the following natural way of transforming a G into a bipartite graph. Let $B(G)$ be the graph obtained by making two copies, v_1 and v_2, for every vertex v of G, and adding an edge between v_1 and w_2 if and only if there is an edge from v to w in G. This transformation was used by Ford and Fulkerson for finding the maximum independent set in a partial order [198].

Theorem 7.1 [176]*: Let G be a comparability graph, and let G' be a transitive orientation of G. Then $B(G')$ is chordal bipartite if and only if G is weakly chordal.*

Proof Suppose that $B(G')$ has a chordless cycle C of length at least 6. We will show that C corresponds to a chordless cycle in G. To do this, we must rule out the possibility that C contains two vertices which map to a single vertex of G, and the possibility of edges of G between two vertices from the same color class in $B(G')$.

Suppose that there is some vertex v such that both v_1 and v_2 are in C; there must be an edge from some c_1 in C to v_2, which implies by the transitivity of G' that $N(c_1)$ contains $N(v_1)$, so v_1 and c_1 could not both be in C.

Suppose that there is an edge between x and y in G, where x_1 and y_1 are in C. Without loss of generality, assume that $x \rightarrow y$ in G'. By the transitivity of G', $N(x_1)$ contains $N(y_1)$, which cannot be true for a pair of vertices in a chordless cycle. A similar argument shows that there is no edge of G between two vertices x_2, y_2 of C.

Thus, if C has k vertices in $B(G')$, these correspond to k distinct vertices of G, and the edges between the vertices in G are the same as the edges in $B(G')$. The vertices of C form a chordless cycle of length at least 6 in G, so G is not weakly chordal.

We now show that if G is not weakly chordal, $B(G')$ is not chordal bipartite. The complement of a comparability graph cannot have any induced cycles of length greater than 4, and comparability graphs can only have induced cycles of even length, except for triangles [205]. Suppose that a comparability graph G has an induced cycle $C = c_1,c_2,...,c_{2k},c_1$ of length at least 5.

Without loss of generality, assume that the edge (c_1,c_2) is directed from c_1 to c_2 in G'. This will force all edges of C to be directed from odd numbered vertices to even numbered vertices in G'. Then $B(G')$ contains the cycle c_{11},c_{22},c_{31}, $c_{42},...,c_{2k2},c_{11}$, and $B(G)$ is not chordal bipartite.

Therefore, $B(G')$ contains an induced cycle of length at least 6 if and only if G is not weakly chordal. □

Theorem 7.1 implies that a weakly chordal comparability graph can be represented using any representation scheme for a chordal bipartite graph. If there is a generalized implicit representation of chordal bipartite graphs, we can take any weakly chordal comparability graph G, find a generalized implicit representation of $B(G')$ as above, and store the bits of x_1 and x_2 with each vertex x of G. To test

whether x and y are adjacent, we check whether x_1 and y_2 or x_2 and y_1 are adjacent in $B(G')$. Since every chordal bipartite graph is weakly chordal, the questions of whether a generalized implicit representation can be found for the two classes are equivalent.

Theorem 7.1 gives an immediate improvement in the time complexity of recognizing weakly chordal comparability graphs. The obvious algorithm has a bottleneck step of recognizing weakly chordal graphs, which takes $O(n^4)$ time. Theorem 7.1 reduces this to recognizing comparability graphs, which takes O(MM) time, and recognizing chordal bipartite graphs.

We will reduce this time complexity further by eliminating the need for verifying that G is a comparability graph. We cannot simply eliminate this step; there are many orientations of noncomparability graphs such that the bipartite transformation is chordal bipartite (exercise 7.9).

We can improve the time complexity by first testing whether the bipartite transformation $B(G')$ is chordal bipartite, and if it is, using properties of chordal bipartite graphs to help check the transitivity of the orientation. The new ordering of steps is as follows. Given G, find the candidate transitive orientation G', and the bipartite transformation $B(G')$ derived from this. If $B(G')$ is not chordal bipartite, answer that G is not a weakly chordal comparability graph. In chapter 9, we will see that all neighborhood containments in a chordal bipartite graph can be computed in $O(n^2)$ time. To check for transitivity violations, we only need to check each edge $x \rightarrow y$, and verify that $N(x_1)$ contains $N(y_1)$ in $B(G')$. Each edge is checked in constant time for violations. Therefore, weakly chordal comparability graphs can be recognized in $O(n^2)$ time. Other problems which can be solved efficiently on this class by using the fact that the bipartite transformation is chordal bipartite can be found in [176].

7.4 Other Generalized Classes

Other generalizations of fundamental classes have not been studied at this point. We introduce a number of natural new classes defined in this way; counting, representation, and efficient recognition algorithms are all worthy of study for the classes. In particular, these are natural examples of classes defined by intersection of graph classes which could be counterexamples to the generalized implicit graph conjecture.

The classes which it seems most natural to generalize are the split graphs (chordal co-chordal), interval graphs (chordal co-comparability or chordal AT-free), permutation graphs (comparability co-comparability) from Golumbic [223], and the chordal comparability graphs of chapter 5. Split graphs do not generalize to a new class by generalizing from chordality to weak chordality, since weakly chordal graphs contain both the chordal and co-chordal graphs; thus the weakly chordal co-weakly chordal graphs are simply the weakly chordal graphs.

Suppose that we want to generalize the class of interval graphs. We have generalized the chordal co-comparability characterization to weakly chordal co-comparability (actually, we dealt with the complement of this class) in the previous section. Generalizing from chordal co-comparability to chordal AT-free simply gives us the other characterization of interval graphs. However, we get a new graph class if we take the chordal AT-free characterization, generalizing from chordal to weakly chordal; thus, we can study the weakly chordal AT-free graphs.

We now look at generalizations of permutation graphs, using the fact that these are equivalent to comparability co-comparability graphs. We can generalize either G and/or its complement from co-comparability graph to AT-free graph. If we ignore classes which are complements of each other, we get two new classes; AT-free comparability graphs and AT-free co-AT-free graphs.

One generalization of chordal comparability graphs gives us weakly chordal comparability graphs, which we studied earlier. We can also consider generalizing the complement class, and studying co-chordal AT-free graphs.

There are many open problems on these classes of graphs, which have not been studied at this point. Fundamentally, do they behave like the more restricted classes when we deal with problems of counting, representing, and optimizing, or do we get classes which are much larger and harder to deal with? Alternative characterizations, such as geometric intersection models, would also be very interesting.

To summarize this section, we introduce a consistent set of new classes formed by generalizing well known existing classes of graphs. The new classes are weakly chordal AT-free graphs, AT-free comparability graphs, AT-free co-AT-free graphs, and co-chordal AT-free graphs.

7.5 Observations Regarding AT-free co-AT-free Graphs

The new classes of graphs defined in the previous section seem to be worthy of attention; very little is known about them at this point. Given my interest in the classes, I have made some preliminary investigations regarding some of them, which I make available in this section. Readers interested only in general issues of graph representations are advised to skip this section. Readers interested in solving representation and counting questions regarding the new classes might find the observations I have made useful.

Consider the AT-free co-AT-free graphs, which come from generalizing permutation graphs. The major known results on AT-free co-AT-free graphs come from [316], which shows that the number of minimal separators in an AT-free co-AT-free graphs is $O(m^2)$, implying that such problems as treewidth and minimum fill-in are polynomial for the class. It can be seen easily that this class of graphs is not perfect, since C_5 is in the class. However, there cannot be any longer cycles in either G or the complement of G, since every chordless cycle of length at least 6 contains asteroidal triples. The first issue we must deal with if we want to find a generalized implicit representation is to determine the size of the graph class.

The main result of this section is that if the number of graphs in the class is large, it is not because of cycles of length 5. We will use $O(n\log n)$ bits, and derive a graph G' which has no cycles of length 5, such that G can be reconstructed from G' plus the $O(n\log n)$ bits. Thus, the counting problem for AT-free co-AT-free graphs will be reduced to the counting problem for AT-free, co-AT-free, weakly chordal graphs, though representation questions on the two classes can be separate. Our fundamental approach is to find one vertex c that can be eliminated from a C_5 so that adjacencies to every vertex v in G to c can be determined from adjacencies between v and the other vertices in the C_5.

We will present this material so that the algorithm for storing the graph (which is somewhat complex) is described as simply as possible, leaving out the proof that the storage method is correct. We will follow the presentation of the storage method with statements of a series of lemmas which can be used to prove the validity of

the method. The proofs themselves do not use any interesting techniques, in each case showing that if the statement of the lemma is false then there is an asteroidal triple in G or the complement of G. These proofs are therefore left to exercise 7.13.

It is possible to reconstruct a graph from its modular decomposition tree, which has $O(n\log n)$ bits, plus the quotient graphs (i.e. the graph induced by taking one vertex from each child) of its prime modules. Therefore, as a first step in storing an AT-free co-AT-free graph, we perform substitution decomposition as described in chapter 11, and try to find a space efficient representation separately for each quotient graph. In the rest of this section, we will assume that G contains no nontrivial modules.

Let C be an induced C_5 = 1,2,3,4,5,1 in an AT-free co-AT-free graph. The clones of a vertex i, denoted clones(i), are the set of vertices which have the same adjacencies to other members of C as i. For example, clones of 1 are the set of vertices in G which are adjacent to 5 and 2, and nonadjacent to 3 and 4; they may be adjacent or nonadjacent to 1. Clones of different vertices i, j of C are adjacent if and only if i and j are adjacent (unproved lemma 7.2).

If G is AT-free and co-AT-free, a vertex v of G can have the following relationships to C and the clones of C (unproved lemma 7.2). We call v a type i vertex, depending on which condition is met for i.

1) v is a clone of some vertex in C.
2) v is adjacent to no clone of any vertex of C.
3) v is adjacent to every clone of every vertex of C.
4) v is adjacent to some of the clones of exactly one vertex of of C.
5) v is nonadjacent to some of the clones of exactly one vertex of C.

A set S of clones is called a unique neighbor set if there is some type 4 vertex above which is adjacent to a vertex in S. Similarly, a set S of clones is called a unique nonneighbor set if some type 5 vertex is nonadjacent to a vertex in S. Any set of clones which is not a unique neighbor set or a unique nonneighbor set is a module, and can assumed to be reduced to a single vertex by our preprocessing step.

A vertex c of C is called reconstructible if it satisfies the following conditions.

1) The clones of c are not a unique neighbor set or a unique nonneighbor set.
2) If the clones of a vertex c_3 at distance 2 on the cycle from c are a unique neighbor set, then clones of the vertex c_2 between c and c_3 on C are neither a unique neighbor set nor a unique nonneighbor set.
3) If the clones of a vertex c_2 at distance 1 from c are a unique nonneighbor set, then the clones of the vertex c_4 of C which is nonadjacent to both c and c_2 are neither a unique neighbor set nor a unique nonneighbor set.

The fact that every induced 5-cycle of an AT-free co-AT-free graph has a reconstructible vertex is the key to our storage scheme; let us discuss how this apparently strange looking definition arises naturally. We want to be able to reconstruct adjacencies to a reconstructible vertex c from adjacencies to other vertices in C. We store the vertices of C along with vertex c. For each other vertex c_i of C - c, we keep track of whether clones of c_i form a unique neighbor and/or unique nonneighbor set with respect to C. When we discuss vertices of the cycle, we will assume that the cycle is ordered c,c_2,c_3,c_4,c_5. Let v be any vertex of G; we give a procedure for determining adjacency between v and c.

Suppose that v is adjacent to none of $c_2..c_5$. Clearly v must be a type 2 or type 4 vertex; since clones(c) are not a unique neighbor set, v is nonadjacent to c. Similarly, if v is adjacent to $c_2..c_5$, v is a type 3 or type 5 vertex, and must be adjacent to c.

Suppose that v is adjacent to exactly one vertex from $c_2..c_5$. If v is adjacent only to c_2 or c_5, v must be a type 4 vertex, and must be nonadjacent to c. If v is adjacent only to c_3 (the argument for c_4 is equivalent), it might be possible for v to be a clone of c_2 or for v to be a type 3 vertex adjacent only to some clones of c_3; these give different possibilities for adjacency to c. This explains part 2 of the definition of reconstructible vertices; if the clones of c_3 are a unique neighbor set, then clones of c_2 form a module and are eliminated by preprocessing, and we conclude that v is nonadjacent to c. If the clones of c_3 are not a unique neighbor set, then v must be a clone of c_2, and is adjacent to c.

Suppose that v is adjacent to exactly two vertices in C - c. Then v must be a clone vertex, and cannot be a clone of c, since the clones of c form a module and are eliminated by preprocessing. If v is adjacent to c_2 and c_3, or v is adjacent to c_4 and c_5, then v is adjacent to c; otherwise, v is nonadjacent to c.

The case where v is adjacent to exactly 3 vertices in C - c is analogous to the case in which v is adjacent to clones exactly 1 vertex, so we will not present the arguments in detail. If the three adjacent vertices are not consecutive on the cycle, then v must be adjacent to c. Otherwise, assume that v is adjacent to c_2, c_3, and c_4. If clones of c_5 form a unique nonneighbor set, then c is adjacent to v (since c_3 cannot have clones after modular decomposition, by part 3 of the definition of reconstructible vertices); otherwise, c is nonadjacent to v since c_5 cannot be a unique nonneighbor set. Similarly, if v is adjacent to c_3, c_4, and c_5, v is adjacent to c if and only if the clones of c_2 form a unique nonneighbor set.

Therefore, we can deduce the neighbors of c from the graph induced by V - c and the O(log$n$) bits stored with C. This gives a relatively complex elimination scheme for transforming an AT-free co-AT-free graph G into a weakly chordal AT-free co-AT-free graph G' using O($n$log$n$) bits, so that G can be reconstructed from this information. This implies that if the number of AT-free co-AT-free graphs is $2^{\Theta(f(n))}$, then the number of weakly chordal AT-free co-AT-free graphs is also $2^{\Theta(f(n))}$.

We state a series of lemmas which can be used to prove that this representation works. The proofs of these lemmas are the subjects of exercise 7.13. Lemmas 7.2 and 7.4 are used in [178] as part of an efficient algorithm for finding triangles in AT-free graphs.

Lemma 7.2 *Let C be an induced 5-cycle of an AT-free co-AT-free graph. Clones of different vertices i,j of C are adjacent if and only if i and j are adjacent.*

Lemma 7.3 *Let C be an induced 5-cycle in an AT-free co-AT-free graph. Every vertex v is either adjacent to no clones of any vertex in C, adjacent to all clones of every vertex of c, is a clone of some vertex in C, is adjacent to some clones of exactly one vertex in C, or is nonadjacent to some clones of exactly 1 vertex of C.*

Lemma 7.4 *If there are two unique neighborhood sets with respect to C, these must be clones of nonadjacent vertices of C. If there are two unique nonneighborhood sets with respect to C, these must be clones of adjacent vertices of C.*

Lemma 7.5 *If C is an induced 5-cycle of an AT-free co-AT-free graph, then C contains a reconstructible vertex.*

7.6 Open Problems on the Generalized Classes

The construction of the previous section does not resolve the counting question for AT-free co-AT-free graphs, but it may allow us to focus our attention on a possibly more tractable class, the weakly chordal AT-free co-AT-free graphs. It is also possible that the restricted types of relationships to 5-cycles can lead to efficient optimization algorithms for this class of graphs, which seem to be in some sense 'barely' not perfect, in the sense of having only restricted forms of 5-holes, and no longer holes in the graph or its complement. Counting, geometric characterization, generalized implicit representation, and recognition (faster than by recognizing AT-free graphs) is also open for this class.

Open Problem 7.2: Let C be any of the following graph classes; AT-free co-AT-free graphs, weakly chordal AT-free co-AT-free graphs, weakly chordal AT-free graphs, AT-free comparability graphs, co-chordal AT-free graphs. Derive bounds on the number of graphs in C. Find a generalized implicit representation for C. Can C be recognized faster than is implied by the definition? Find a representation which exactly characterizes the class.

For some of these problems, specific approaches seem promising. For example, we discussed a reduction from weakly chordal comparability graphs to chordal bipartite graphs, which allows us to count the number of graphs effectively. To do this, we took a transitive orientation of a graph G, took a natural bipartite transformation, and found that the result was always chordal bipartite. We proved in theorem 7.1 that the result is chordal bipartite if and only if the undirected underlying graph is weakly chordal, if we know that the input is transitive.

If the input is not transitive, other orientations might still yield a chordal bipartite graph. Therefore, it is natural to look for an orientation of a weakly chordal AT-free graph such that the bipartite transformation is chordal bipartite. Orderings of AT-free graphs based on lexicographic breadth first search are studied in [125] ; it is possible that these can be used to solve the counting problem for weakly chordal AT-free graphs.

The following specific conjecture comes from [316]; if it is true, it would immediately solve the counting and representation problems for AT-free comparability graphs.

Open Problem 7.3 [316]: Is the complement of every AT-free comparability graph a trapezoid graph?

More generally, if the interval dimension of the transitive orientation is bounded, i.e. the complement is a k-trapezoid graph for some fixed k, then AT-free comparability graphs have an implicit representation. It is also open as to whether the dimension of an AT-free partial order is bounded by a constant.

The class of AT-free graphs has received a good deal of attention recently; this is in part because they are a large class of graphs which can be recognized in polynomial time, and maintain some of the nice properties of interval graphs. The next paragraphs describe an alternate method for generalizing interval graphs and avoiding asteroidal triples.

Let L be an ordered list of vertices of a graph G. L is a path ordering (this has also been called an AT-destroyer) if for every independent set of 3 vertices $x < y < z$ in L, every path from x to z goes through a neighbor of y.

Although every path orderable graph AT-free, there are AT-free graphs which have no path ordering [317]. The class of AT-free graphs is sometimes justified in terms of an indication of a possible 'linear structure'; perhaps path orderable graphs have a more clear indication of a linear structure. Recently, the problem of determining whether a graph is path orderable was shown to be NP-complete [317].

7.7 Exercises

1) Show that every co-comparability graph is asteroidal triple-free.

2) Show that there are asteroidal triple-free graphs which are not co-comparability graphs.

3) Design an $O(n^3)$ algorithm for recognizing asteroidal triple-free graphs.

4) Show that if the complement of G contains a chordless cycle of length greater than 4, then G contains a chordless cycle of length at least 4.

5) Explain why problem 4 shows that chordal graphs are a subclass of weakly chordal graphs.

6) Show that the class of bipartite co-comparability graphs is equal to the class of bipartite permutation graphs; do not use the forbidden subgraph characterization.

7) Show that a bipartite graph $G = X,Y,E$ is co-chordal if and only if the neighborhoods of color class X are ordered by set containment.

8) Show that G is weakly chordal and bipartite if and only if G is a chordal bipartite graph; i.e., G is bipartite and has no induced cycle of length > 4.

9a) Give an example of an orientation of a noncomparability graph such that the bipartite transformation of section 7.3 is chordal bipartite.

9b) Give an example of a weakly chordal graph and an orientation such that the bipartite transformation of section 7.3 does not produce a chordal bipartite graph.

9c) Show that if G is a sufficiently large co-bipartite graph on $2n$ vertices, with each vertex i from the first co-color class adjacent only to vertex i' from the other co-color class, there is no orientation of G such that the transformation of section 7.3 produces a chordal bipartite graph. I note that question 9b was suggested by Jim Nastos, who also posed question of whether there was any graph which could not be oriented to produce chordal bipartite output, thus leading to problem 9c.

10) Every AT-free comparability graph is weakly chordal, so the transformation described in section 7.3 must give a chordal bipartite graph, if the input is an AT-free comparability graph. This question considers the possibility that if the input is

an AT-free comparability graph, the output of the transformation might correspond to one of the known subclasses of chordal bipartite graphs.

Some well known subclasses of chordal bipartite graphs are bipartite permutation graphs, biconvex graphs (you can order each color class so that neighbors of every vertex from the other color class appear consecutively), and convex graphs (one color class can be ordered as above); these classes are discussed further in chapter 9. It might be natural to hope that a graph is an AT-free comparability graph iff the transformation produces a graph in one of these restricted classes.

Show that it is not true that G is an AT-free comparability graph iff the transformation produces a graph in class C, for C = any of bipartite permutation graphs, biconvex graphs, convex graphs, chordal bipartite graphs.

For problems 11 and 12, you will need to use properties of chordal bipartite graphs, as discussed in chapter 9. In particular, you need to know that a Γ-free ordering of the bipartite adjacency matrix can be constructed in $O(n^2)$ time.

11) Design an $O(n^2)$ algorithm for solving the maximum independent set problem on weakly chordal comparability graphs.

12) Design an $O(n^2 \log n)$ algorithm for computing the transitive closure or reduction of a weakly chordal partial order.

13a-d) Prove each of the unproved lemmas 7.2–7.5 leading to the existence of a reconstructible vertex from any 5-cycle in an AT-free co-AT-free graph.

14) Find a graph which is weakly chordal, AT-free, co-AT-free, and is not a permutation graph.

15*) Show that chordal P_5-free graphs can be recognized in $O(n^3)$ time.

16) Show that the number of co-chordal triangle-free graphs is $2^{O(n \log n)}$.

17) Show that asteroidal triple-free graphs satisfy the conditions of theorem 3.2, and are therefore an intersection class. As far as I know, no nice characterization of asteroidal triple-free graphs as an intersection class is known.

CHAPTER 8

Graph Classes Defined by Forbidden Subgraphs

One common way to define classes of graphs is to specifically exclude certain subgraphs. Almost every small subgraph, and many combinations of several small subgraphs, have been studied in some context. Classes defined by forbidding a fixed set of induced subgraphs will all be hereditary, but there is nothing in the definition which will naturally lead to a good representation. Therefore, these classes are particularly interesting with respect to the implicit graph question.

In the interests of space, we look primarily at graph classes defined by single forbidden induced subgraphs. The key forbidden subgraph here is P_4. If a forbidden subgraph F is a subset of P_4, then the class defined by forbidding F as an induced subgraph has an implicit representation. If F is not a subset of P_4, then the class defined by forbidding F as an induced subgraph contains $2^{\Theta(n^2)}$ members. Those readers who are interested in more precise bounds on the constants in front of the Θ for individual graphs can find more information in [400]. We will discuss the class defined by forbidding P_4 in some detail, then move on to some interesting questions on graph classes which have too many members for the representation problem to be interesting. We will discuss counting, representation, construction, recognition, and optimization problems for various classes in this chapter.

It is interesting to note that the size of poset families defined by forbidding single induced posets has very different behavior than the corresponding questions on graphs. Forbidding each induced height 1 subposet gives a class of posets with $2^{o(n^2)}$ members, and there are many classes defined in this way which have more than 2^{cnlogn} members for any value of c. Results and many open counting problems of this form are contained in [80]. Bipartite families defined by forbidden subgraphs are discussed in chapter 9.

8.1 Cographs

One particularly tractable class of graphs is the class of graphs without any induced P_4. These have been characterized in a number of different ways, and there are efficient algorithms for a wide range of problems which are NP-complete on general graphs.

Graphs without induced P_4s are commonly called cographs because of the following characterization; essentially, it says that P_4-free graphs are exactly equal to the set of graphs which can be built from single vertex graphs using only the union and complement operations. More formally, the class of cographs is defined as follows.

1) A single vertex is a cograph.

2) If G_1 and G_2 are cographs, then $G_1 \cup G_2$ is a cograph.

3) If G is a cograph, then the complement of G is a cograph.

The equivalence between cographs and graphs without induced P_4s is proved in [342].

It can be shown (exercise 8.1) that every cograph on n vertices can be constructed using $O(n)$ of the above operations. Each operation can be described using $O(\log n)$ bits, and the graph can be reconstructed from the sequence of operations. Thus, the number of cographs is $2^{O(n \log n)}$. Representing a cograph by giving a list of these operations has serious drawbacks; adjacency testing would be relatively expensive. Therefore, the question of whether an implicit representation of cographs exists is natural.

Another representation of cographs is used frequently. A *cotree* for G is a tree T such that vertices correspond to leaves in the tree, internal nodes have degree at least 2 and are labeled 0 or 1, and vertices x and y of G are adjacent if and only if the least common ancestor of x and y is labeled 1. The proof that G is a cograph if and only if G has a cotree representation is given in [342]. Cotrees correspond to substitution decomposition trees for cographs, as studied in chapter 11, where series modules of the decomposition tree are labeled 1 and parallel nodes are labeled 0. Thus, cographs are also equivalent to graphs which have no prime modules in their substitution decomposition tree.

Cotrees can be used to design efficient algorithms for many problems on cographs. The fundamental technique is to perform a postorder traversal, and to use the fact that relationships between vertices from different children of the same node are very easy to characterize. For example, suppose that we want to solve the maximum clique problem on cographs using the cotree representation. Mark all leaves as having maximum clique size 1. As we process a cotree in postorder, if we encounter a 1 node v, the maximum clique on descendants of v is equal to the sum of the weight of the maximum cliques of its children, and if v is a 0 node, then the maximum clique on descendants of v is the maximum value of the clique size of any children.

Representing a graph by giving its cotree uses $O(n \log n)$ bits. Once again, testing whether x and y are adjacent is not necessarily efficient, as it requires determining the label of the least common ancestor of x and y. Of course, this is not an implicit representation, since determining whether x and y are adjacent involves looking at information which would not be stored at x or y.

It is easy to use the cotree representation of G to construct a representation of G as a permutation graph. The strategy involves using postorder traversal of the cotree as described for the clique problem above; the complete construction is left for exercise 8.2.

Since algorithms using the cotree for so many problems take $O(n)$ time, it is clear that the bottleneck step for the construction and optimization problems is construction of the cotree. This may seem to be an easy problem, since the substitution decomposition tree can be constructed in linear time, and the cotree is a special case of the substitution decomposition tree. In fact, this argument is circular, since most known linear time substitution decomposition algorithms [366, 136], use the cotree construction algorithm as subroutines; an exception is the algorithm in [145]. We will describe below a linear time cograph recognition and cotree construction algorithm for cographs due to Corneil, Perl and Stewart [128]. The $O(n^2)$ and $O(m \log n)$ algorithms for substitution decomposition given in chapter 11 can also be used for cotree construction.

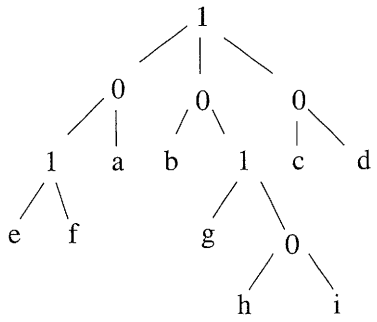


Figure 8.1 Example for the Cograph Recognition Algorithm

The algorithm is incremental, meaning that one vertex is added at each step, and the cotree representation of the current graph is found. Since cographs are hereditary, we stop if at any time we find the current graph has no cotree representation.

We simplify the discussion by assuming that G is connected (if not, the cotree can be constructed by joining all cotrees for connected components under a 0 node), and we add vertices so that the subgraph induced by vertices added at any step is always connected. This can be achieved by adding vertices in an order governed by an arbitrary spanning tree. This restriction allows us to assume that the root is always a 1 node. For some applications, it is desirable to add vertices on-line; the algorithm of Corneil, Perl and Stewart [128] can handle this case as well.

Start with the first u,v, and place them under a 1 node. We now consider the process of adding a new vertex x to a cotree T; we will call the resulting cotree T'.

We use a marking procedure on T as follows. Nodes can be unmarked, partially marked, or fully marked; a node is fully marked if it has been marked by all its children. Originally, leaves which correspond to neighbors of x are marked; since these have no children, they are considered to be fully marked.

Fully marked nodes are placed on a queue, and any fully marked node marks its parent. Consider the cotree of Figure 8.1. Let x be a new vertex adjacent to a, c, d, e, f, g and h. When x is added, the partially marked nodes will be the root, the parent of g, and the parent of h. It is easy to do the marking in time proportional to the number of neighbors of x. Let α, β be the partially marked nodes of T which are at the lowest and second lowest possible levels respectively. In this example, the parent of h is α, and the parent of g is β.

In [128], it is shown that the partially marked nodes, with the exception of α, must have the following structure if the new graph is a cograph. Let PM be the set of partially marked nodes other than α. If PM is empty, then G is a cograph. Otherwise, if G is a cograph, all vertices of PM must be 1 nodes marked by all but 1 child, and they must consist of every other node on the path from β to the root. α must be either a child or grandchild of β, unless β does not exist. These criteria are met in our example.

If the conditions above are met, the new graph is a cograph, and x is added as follows. We note that once the reader understands the algorithm to this point, it may be more instructive to deduce the rules on your own rather than to treat the

rules as following a strange piece of pseudocode. If there are no partially marked nodes, x is added as a child of the root. Otherwise, there are 2 cases.

If α is a 0 node marked by at least 2 children, or a 1 node not marked by at least 2 children, split α by removing all nodes which marked α and place them under a new node y, giving y the same label as α. If α is a 0 node, place y and x under a new 1 node, and make this a child of α. If α is a 1 node with parent p, make a new 0 node with children α and x, make the new node a child of y, and make y a child of p.

If α is a 0 node marked by only 1 child c, or a 1 node not marked by only 1 child c, we have two cases. If c is a leaf, replace c by a new node, which has the opposite label from α, and has c and x as children. If c is not a leaf, add x as a new child of c. In our example, we replace the leaf h by a new 1 node, with children h and x.

The steps of marking the tree and adding the new node to the tree take time proportional to the number of neighbors of x, if standard data structures are used. Data structures used would include a queue of fully marked nodes for the marking procedure, and for each internal node of the tree a variable which stores the number of children which have marked the node, and a list of these children. The time bound comes from the fact that work performed is proportional to the number of fully marked nodes of the tree, and the number of fully marked nodes is $O(N(x))$ (see exercise 8.5).

The only other step which could be a bottleneck for cotree construction is identification of the lowest level partially marked nodes α and β. These can be found by a separate labeling procedure, which takes place after partially marked nodes are identified. Each partially marked node labels its parent and grandparent. If G is a cograph, there will be exactly one partially marked node which is not labeled; this is the lowest level partially marked node α. β, if it exists, is the partially marked node which is the parent or grandparent of α. From this, it is easy to check that the partially marked nodes other than α are equal to the 1 nodes on the path from β to the root, and that these are marked by all but 1 child, by simply following the path from β to the root.

Many generalizations of cographs can be found in the literature, defined by severe restrictions on the number and types of P_4s in G. It is not immediately clear that all of these classes can be represented implicitly. [74] presents definitions and basic theorems for many graph classes defined in this way.

8.2 Classes Which are too Large to have Efficient Representations

In this section, we study classes of graphs defined by forbidden induced subgraphs which contain too many members to have efficient representations. We will point out a number of well known classes of this type, and discuss some natural problems which have not been asked about some of the classes. We will see that for any graph F which is not a subset of P_4, the graph class defined by forbidding F as an induced subgraph has $2^{\Theta(n^2)}$ members. Readers interested in a more precise evaluation of the growth rate of these classes of graphs can find a formula in [400].

8.2.1 Triangle-free Graphs and Matrix Multiplication. We first discuss the class of triangle-free graphs. A graph is triangle-free if it does not contain the subgraph K_3. For this class, we do not need to make the distinction between containing subgraphs or containing induced subgraphs. It is easy to see that the

number of triangle-free graphs is $2^{\Theta(n^2)}$, since every bipartite graph is triangle-free. Therefore, the question of finding a generalized implicit representation for this class is not interesting.

Recognizing triangle-free graphs efficiently does raise some interesting questions, however. Before discussing the best known algorithm for this problem, we introduce a new piece of terminology.

The 0,1 multiplication of two 0/1 matrices A, B is what we get if we perform standard matrix multiplication of $A \times B$, and replace all positive values with a 1. The author has sometimes called this form of multiplication Boolean multiplication in the past; however, Boolean multiplication has other meanings which could confuse the reader. Specifically, one might assume that Boolean arithmetic is being used, in which case $1 + 1 = 0$, so that the result of multiplying two 2 by 2 matrices which have all entries equal to 1 would produce an all 0 matrix, rather than the all 1 matrix desired here.

Triangle-free graphs can be recognized relatively easily in $O(n^\alpha)$ time, where n^α is the time needed to perform 0,1 matrix multiplication of two n by n matrices. At the time of this writing, the best known upper bound for n^α is $O(n^{2.376})$ [118]. Let A be the adjacency matrix of G, and compute $B =$ the result of the 0,1 multiplication of $A \times A$. Then G has a triangle if and only if there is some pair of vertices x,y such that $A[x, y] = 1$ and $B[x, y] = 1$.

It is generally agreed that the asymptotically fastest matrix multiplication algorithms are not practical, since the constants in front of the n^α are astronomical. There is also a less precise reason for objecting to using matrix multiplication on this problem. The steps performed in recognizing triangle-free graphs using matrix multiplication do not translate in any natural way to graph theoretic steps, and are completely divorced from any intuition about what is happening in the graph. The author, at least, feels that an algorithm which used (for example) matching as a subroutine to solve the triangle-free graph recognition in $O(n^{2.5})$ time would be a very interesting result, even if it would not speed up the asymptotic time complexity of triangle-free graph recognition. Therefore, we pose two open problems.

Open Problem 8.1: Design an $O(n^{3-\epsilon})$ algorithm for recognizing triangle-free graphs which is completely graph theoretic; specifically, matrix multiplication routines should not be used.

Of course, an algorithm which used less time than matrix multiplication would be an even more impressive result. On the other hand, we are satisfied with time complexity proportional to matrix multiplication if the problem is actually tied to matrix multiplication. Transitive closure and transitive reduction of digraphs are graph theoretic properties, but it can be shown that they are essentially equivalent in time complexity to matrix multiplication [11], and this is generally viewed as resolving the problem.

In the same way, we would be very interested in a result stating that triangle-free graph recognition is as hard as matrix multiplication. However, it is hard to imagine proving such a result, for the following reason. Triangle-free recognition is essentially a decision problem; there is one bit of output (yes or no). On the other hand, a matrix multiplication routine must produce n^2 bits of output. Since lower bound arguments are generally proved by reduction, it is hard to imagine that we could turn a matrix multiplication question into a question where the answer is

yes or no; each answer yes and no must correspond to many matrix multiplication problems.

Therefore, we propose a potentially easier form of matrix multiplication, which we will call matrix multiplication verification. In this problem, we are given matrices A, B, and C, and are asked whether $A \times B = C$.

Open Problem 8.2: Is triangle-free recognition as hard as 0,1 matrix multiplication verification?

[178] studies the complexity of finding triangles in restricted classes of graphs, designing efficient algorithms for such classes as AT-free graphs and EPT graphs .

Let me add a few notes on triangle-free graph recognition. It is entirely possible that matrix multiplication verification is much easier than matrix multiplication. In fact, standard matrix multiplication can be verified in the randomized sense quite easily in $O(n^2)$ time, as we will discuss below, whereas a randomized $O(n^2)$ matrix multiplication algorithm would be considered to be a great breakthrough. Since 0,1 multiplication cannot be harder than standard matrix multiplication, it may seem counterintuitive that 0,1 verification may well be harder than standard matrix multiplication verification.

To see how this might be true, let us look at a simple technique for randomly verifying standard matrix multiplication of two n by n matrices. Generate randomly a string of n 1s and -1s, and call the resulting vector y. To check that $A \times B = C$, we check whether $(y \times A) \times B = y \times C$; the test uses $O(n^2)$ scalar operations. If $AB = C$, the answer is always yes, by associativity of matrix multiplication. If AB is not equal to C, it will take a remarkable vector y to make the two results equal; formal arguments can be made that the likelihood of y being chosen to make this happen is extraordinarily small [201].

To see why this is not as effective for verifying 0,1 matrix multiplication, look at the following example. Fundamentally, the problem is that a mistake in Boolean matrix multiplication can be overlooked, since all positive values in the result are compressed to a single value. Thus, while different positive values give errors with respect to standard matrix multiplication, they do not give errors in 0,1 multiplication. Suppose that you are multiplying a pair of n by n matrices which contain all 1 entries, and you introduce an error by placing a 0 rather than a 1 in position 1,1 of the result matrix C. The result of $y \times C$ should be 1 everywhere if and only if there are more 1s than -1s in y; otherwise the result should be 0s everywhere. Introducing a 0 in position $C[1,1]$ can only change the result of $y \times C$ if the number of 1s differs from the number of -1s by at most 1; as n becomes large, this possibility becomes increasingly unlikely. The example above could be remedied by randomly generating strings with an equal number of 0s and 1s, but this is not enough to handle general 0,1 verification; see exercise 8.3. Verifying 0,1 matrix multiplication in randomized $O(n^2)$ time would be a very useful and interesting result.

Many other graph theoretic problems have matrix multiplication verification as a bottleneck step. Some of the better known examples include recognizing graphs without simplicial vertices, comparability graphs, and diagrams of posets, and finding all shortest paths in an unweighted graph. We are making some attempts to organize these problems, in terms of which problems are reducible to each other, at the time of this writing, since these relationships have not been explored previously. Some problems which can be shown to be as hard as triangle-free graph recognition are discussed in the exercises. At this time, we note that there are a

number of different types of decision problems which have matrix multiplication as a bottleneck step and seem to be hard to reduce to each other. Our present list of 'basic' decision problems includes recognition of triangle-free graphs, determining whether there is a vertex pair x, y such that $N(x) \subseteq N[y]$, and determining whether a graph has a simplicial vertex. For a further discussion, see [332].

8.2.2 Claw-Free Graphs. There is another well known class of graphs which has a more surprising gap in our knowledge with respect to the recognition problem. A graph is claw-free if it contains no induced subgraph isomorphic to $K_{1,3}$. The independent set problem can be solved in polynomial time for claw-free graphs, as shown by Sbihi [420] for the unweighted case and Minty [370] for the weighted case. Matching on claw-free graphs is discussed in chapter 14.

Once again, it is easy to see that the number of claw-free graphs is $2^{\Theta(n^2)}$. A claw contains an independent set of size 3, so the complement of every bipartite graph is claw-free.

The obvious $O(n^4)$ algorithm for this problem (test whether each subgraph of size 4 is a claw) can be improved easily if we allow the use of matrix multiplication. We can test whether G is claw-free by determining whether the complement of the graph induced by the neighborhood of each vertex is triangle-free. Each test can be done using matrix multiplication as above, so we can test whether a graph in $O(n^{1+\alpha})$ time.

Kloks, Kratsch and Müller make the following observation, which improves the time bound for sparse graphs. If G is claw-free, then every vertex has at most $2m^{1/2}$ neighbors (see exercise 8.6).

Suppose the degree of every vertex is at most $2m^{1/2}$. The time spent checking $N(x)$ for three independent vertices is at most $degree(x)^{\alpha}$, which we rewrite as $degree(x) * degree(x)^{\alpha-1}$. Summing over all vertices, the total time is at most $\sum_{i=1}^{n} degree(i) 2m^{1/2^{\alpha-1}}$. Since the second term in this summation is a constant, this is equal to $2m^{1/2^{\alpha-1}}$ times the sum of the degrees, which is $O(m) * 2m^{1/2^{\alpha-1}}$, which is $O(m^{(\alpha+1)/2})$. However, when G is dense, this is still $n^{1+\alpha}$.

For a 'lower bound', we will show that recognizing claw-free graphs is at least as hard as recognizing triangle-free graphs, assuming that both problems require $\Omega(n^2)$ time. Suppose that we want to test whether G is triangle-free. Create a new graph G_2 by constructing the complement of G, and adding a new vertex which is adjacent to all vertices in the complement. G_2 has a claw if and only if G has a triangle.

The construction above is not, of course, a true lower bound; it is entirely possible that both of these problems can be solved in linear time. Nevertheless, even an ambitious graph algorithms specialist needs to know which improvements would cause improvements in algorithms for well known problems. Improving on n matrix multiplications for this problem seems quite feasible, whereas getting below 1 matrix multiplication may be quite difficult. Therefore, the open problem I will pose in this context is as follows.

Open Problem 8.3: Find an algorithm for recognizing claw-free graphs which takes $o(n^{1+\alpha})$ time, where n^{α} is the time needed to perform 0/1 multiplication of two n by n matrices.

Errata: There is an excellent survey on graph classes in the Handbook of Algorithms and Theoretical Computer Science [338]. However, the time complexity given for

recognition of claw-free graphs is linear, without any further comment. I have spoken with the author, and determined that this is a simple case of an exponent being dropped accidentally during the production of the book.

8.2.3 Forbidding Single Induced Subgraphs. One other class defined by small forbidden subgraphs is interesting, in order to prove that the only small classes of graphs defined by one forbidden subgraph are subclasses of cographs. G is square-free if and only if G contains no induced subgraph isomorphic to C_4.

The number of square-free graphs is $2^{\Theta(n^2)}$, since they contain the class of split graphs. Suppose that a split graph $G = (X,Y,E)$, where X is an independent set and Y is a clique, contains the C_4 u, v, w, x. At most two vertices of the C_4 are from X, since C_4 has no independent set of size 3; similarly, at most 2 vertices are from Y. The two vertices from Y must be neighbors on the cycle, but this would make the other two vertices neighbors, contradicting the fact that these vertices must be in X.

It is now easy to show that any graph F which is not an induced subgraph of P_4 defines a large class of graphs, since any graph is either an induced subgraph of P_4, or contains C_3, C_4, or their complements as induced subgraphs. A more general theorem on hereditary classes of graphs is given as Theorem 8.3.

Theorem 8.1 *Let F be a graph. If F is not an induced subgraph of P_4, then the graph class defined by forbidding F as an induced subgraph has $2^{\Theta(n^2)}$ members.*

Proof Consider any graph F such that F is not an induced subgraph of P_4, and the class defined by forbidding F as an induced subgraph does not have $2^{\Theta(n^2)}$ members. As shown above, F cannot contain K_3, or the class contains the bipartite graphs. F cannot contain C_4, or the class contains the split graphs. F cannot contain the complements of K_3 or C_4; the class would contain the complements of the corresponding classes (here the co-bipartite graphs and the split graphs, since split graphs are self-complementary).

Every 3 vertex graph is an induced subgraph of P_4, or is equal to C_3 or $3K_1$, so F contains at least 4 vertices u, v, w, x. Without loss of generality, since F does not contain 3 independent vertices, assume that u and v are adjacent. If neither w nor x is adjacent to u or v, then F either contains an independent set u,w,x of size 3, or $F = 2K_2$ which is the complement of C_4, and thus defines a class with $2^{\Theta(n^2)}$ members. Assume w is adjacent to v; then w is not adjacent to u, or F contains C_3. Vertex x is adjacent to either u or w, or u, w, x is an independent set of size 3; this implies that x is nonadjacent to v, or F would contain C_3. If x is adjacent to both u and w, then F contains C_4; thus we can assume that u,v,w,x are a P_4.

Since F is not an induced subgraph of P_4, F contains a fifth vertex y. Since u, x, y are not an independent set, we can assume without loss of generality that x and y are adjacent. This implies y is nonadjacent to w, or w, x, y induce C_3; this implies in turn that y is adjacent to u (or u,w,y are independent) and y is nonadjacent to v. The vertices induce a C_5, which means that the class defined by forbidding F as an induced subgraph contains the bipartite graphs. □

8.3 Relation Between Recognition Problems

The following theorem, which seems to be stated first in [384], is not very well known; basically, this says that the K_i-free graphs are at least as hard to recognize as the F-free graphs for any forbidden induced subgraph F on i vertices.

Theorem 8.2 [384] *Let F_i be any forbidden induced subgraph on i vertices. The problem of determining whether a graph G contains F_i as an induced subgraph can be reduced in $O(im)$ time to the problem of determining whether a graph on $O(in)$ vertices contains K_i.*

Proof by Reduction Order the vertices of F_i arbitrarily with numbers from 1 to i. Create a graph $G(F_i)$ as follows. Make i copies $v_1 ... v_i$ of each vertex v in G. Add an edge from v_j to w_k in $G(F_i)$ if j and k are distinct, v and w are distinct, and the relationship (i.e. adjacency or nonadjacency) between v and w in G is the same as the relationship between vertices j and k in F_i.

It is easy to see that a subgraph isomorphic to F_i in G maps to a clique of size i in $G(F_i)$, and that a clique of size i in $G(F_i)$ corresponds to a set of vertices isomorphic to F_i in G. □

Other reductions of this type are also possible; exercise 8.13 asks for a reduction showing that recognizing whether a graph contains C_k as an induced subgraph for $k > 4$ is at least as hard as recognizing whether a graph has a triangle. It is easy to show that for many specific graphs F, recognizing F-free graphs is as hard as recognizing triangle-free graphs. It can be shown that this statement is true for any forbidden graph F on at most 5 vertices, with the exception of C_4. Since there is no general theorem of this type, I state the following open problem, which seems very plausible to me.

Open Problem 8.4: Can you show that if F is not an induced subgraph of P_4, recognizing F-free graphs must be as hard as recognizing triangle-free graphs?

8.4 Classes defined by Forbidding Sets of Induced Subgraphs

Many classes of graphs defined by sets of forbidden subgraphs have been defined; some examples are given in [74]. In general, these classes can form major challenges to the generalized implicit graph conjecture.

Scheinerman and Zito [427] raise the question of whether we can take as input a set of forbidden subgraphs, and solve the counting problem for the class; that is, we would want to find a function f(n) such that the number of graphs in the class is $2^{\Theta(f(n))}$. Currently, we do not seem to even know how to solve this problem for sets of two forbidden subgraphs. There are many reasonable approaches for progress on this problem; as well as attacking the problem for small number of forbidden subgraphs, it would be interesting to know whether the general problem is NP-hard, and if so how many different graphs are necessary to make this difficult. NP-hardness results would not end research in this direction, since it is reasonable to take exponential time in the size of your input graphs for many natural graph classes. For example, if our forbidden subgraphs are two graphs with a total of some small number of vertices, a counting procedure which takes exponential time in the size of the input graphs could also be very interesting, and it is not obvious how to decide this problem in any time bound. Although many problems of this form are open, we will present one problem summarizing these.

Open Problem 8.5: Determine the difficulty of finding procedures which take a finite set of forbidden induced subgraphs as input, and solving the counting problem for the class of graphs defined.

Exercise 8.7 deals with an attempt to generalize cographs in the same way that weakly chordal graphs generalize chordal graphs; we allow P_4 but forbid P_5 in G or its complement, just as weakly chordal graphs allow C_4 but forbid longer cycles in G or its complement. These graphs are studied in [219].

There are many specific open classes defined by sets of forbidden subgraphs. For example, one can consider the class of graphs defined by forbidding $K_{3,3}$ and K_5 as induced subgraphs rather than as minors. Recognition, counting, and representation problems may be interesting on such a class; exercise 8.12 shows that there are at least $2^{n^{1.5}}$ graphs in this class. Other pairs of forbidden subgraphs can be studied; some simple cases are the subjects of exercises 8.7-8.12. The paper [75] shows that for every four vertex graph H except for C_4 and its complement $2K_2$, the H-free co-H-free graphs have bounded clique-width as studied in Chapter 11, and thus have an implicit representation. A forthcoming paper [72] studies clique-width of every graph class defined by a pair of forbidden four vertex subgraphs.

An interesting general theorem comes out of the work of Alekseev; the original paper is in Russian, so I do not know proof details. We will present an alternative proof of the theorem, using results from chapter 10.

Theorem 8.3 [12] *A hereditary class of graphs has $2^{\Theta(n^2)}$ members if and only if it contains all bipartite graphs, all co-bipartite graphs, or all split graphs.*

Proof Let C be a hereditary class of graphs. Clearly, if C contains all bipartite graphs, all co-bipartite graphs, or all split graphs, then C has $2^{\Theta(n^2)}$ graphs on n vertices.

Suppose that C does not contain any of these classes. Then there is a bipartite graph G_b, a co-bipartite graph G_c, and a split graph G_s which are not in C. Let k be a constant at least as large as sum of the number of vertices in G_b, G_s, and G_c.

Ramsey theory tells us that there is a number N such that every graph on at least N vertices contains either an independent set of size k, or a clique of size k. Let G be an arbitrary graph on n vertices, where $n \geq N$. Select and remove a clique or independent set V_1 of size k from G, a clique or independent set V_2 of size k from the remaining vertices, and so on, until there are at most N vertices remaining.

To count the number of graphs with n vertices in C, we first count the number of ways to partition vertices into cliques and independent sets above. Every partition corresponds to an ordering of the vertices, plus a choice of whether each set is a clique or an independent set. It is not hard to see that this implies that there are at most $n!2^{n/k}$ ways to partition vertices into sets.

We now consider a fixed division of vertices into cliques and independent sets of size k as described above. Consider the subgraph G_{ij} induced by any pair of sets V_i, V_j. $G_{i,j}$ is a bipartite graph if V_i and V_j are independent sets, a co-bipartite graph if V_i and V_j are cliques, and a split graph if one is a clique and the other an independent set. Each $G_{i,j}$ can be viewed as a matrix M with vertices of V_i as rows, vertices of V_j as columns. There is at least one forbidden induced submatrix of M, corresponding to G_b, G_c, or G_s being forbidden induced subgraphs of C. The proof of Theorem 10.3 shows that the number of such k by k matrices, and

thus the number of ways to choose $G_{i,j}$, is at most $2^{k^2-\epsilon}$ for some ϵ. Since there at most n/k cliques and independent sets created, the number of ways to choose all connections between sets is $\left(2^{k^2-\epsilon}\right)^{n^2/k^2}$, which is $2^{n^2(1-\delta)}$ for $\delta = \epsilon/k^2$.

This accounts for all choices of adjacencies between vertices from the independent sets and cliques. There are at most N vertices not placed in one of these sets; the number of choices of edges involving these vertices is at most 2^{nN}.

Therefore, the number of possible graphs in C is at most $n!2^{n/k}2^{n^2(1-\delta)}2^{nN}$; since N, k, and δ are constants, a graph class C which has a forbidden induced bipartite graph, a forbidden induced co-bipartite graph, and a forbidden induced split graph cannot contain $2^{\Theta(n^2)}$ members. □

Note that the proof of Theorem 8.3, together with the proof of Theorem 10.3, can be used to derive upper bounds on the number of graphs in a class by finding a small split, bipartite, and co-bipartite graph not in the class. However, these upper bounds may be very much larger than the actual number of graphs in the class.

8.5 Dilworth Number and Poset Width

There are many other variants of forbidden subgraphs, involving issues such as forbidden minors, homeomorphs, and many others; [74] gives a number of different notions and graph classes which fit these notions. This section is devoted to a class defined by forbidding a particular subset which is not an induced subgraph, and has a nonobvious implicit representation.

The Dilworth number of a graph is the maximum number of vertices in the graph such that the neighborhood of one vertex is not contained in the (closed) neighborhood of the other. Graphs with fixed Dilworth number have been studied in a number of papers; thus, for example, it is known that graphs with Dilworth number 1 are exactly the threshold graphs, graphs with Dilworth number 2 are equal to a class called threshold signed graphs studied in [31], graphs with Dilworth number 3 must always have a simplicial or cosimplicial vertex [263] and graphs with Dilworth number 4 are perfect [395].

Dilworth number has a close connection with the width of a poset, which is the maximum number of mutually independent elements in the poset. The Dilworth number of G is exactly equal to the width of the poset constructed by adding an edge from x to y if the neighborhood of y is contained in the neighborhood of x.

The easiest way to see that graphs with Dilworth number k and posets of width k have an implicit representation is to use a famous theorem due to Dilworth; the size of the maximum antichain of a poset is exactly equal to the minimum number of chains needed to partition the elements of the poset. Note that this can be viewed as a proof that comparability graphs are perfect; antichains correspond to independent sets and chain partitions to clique covers.

Given k chains which partition the neighborhood containment poset, we can place every vertex v at an appropriate level of each chain, so that vertices above this level are adjacent to v, and vertices below this level are nonadjacent to v. Thus, for each vertex v we can maintain the position of v with respect to k chains, the chain v is in, and the height of v in its own chain; from this information, it is possible to determine adjacency between x and y using only the bits stored at x and y.

The existence of an implicit representation does not always imply that such a representation can be found efficiently. For example, partial orders of dimension k

have an implicit representation, but it is NP-complete to determine the dimension of a partial order. However, it is possible to construct the implicit representation of a graph with Dilworth number k in $O(n^2)$ time.

In general, Dilworth number can be computed by computing the neighborhood containment poset, and then using bipartite matching to find the width of the poset, as described in exercise 14.9. This would take time proportional to matrix multiplication for finding the poset, and bipartite matching for finding the width. For fixed Dilworth number, these steps can be reduced. The neighborhood containment poset of a graph with Dilworth number k can be found in $O(n^2)$ time for any fixed k [189]; the same paper shows that determining whether a partial order has width k can be solved in $O(n^2)$ time for fixed k.

We will give an outline of the algorithm to determine whether a partial order has width k. The first $O(n^2)$ algorithm for this problem is due to Gavril [217] ; we present the algorithm from [189]. We first find an approximation of the chain partition, i.e. a small but not minimum size set of chains which cover all elements of the poset. Find a maximum chain in the poset; this is simply finding the longest path in an unweighted directed acyclic graph, and can be solved in $O(n^2)$ time. Remove the elements of this chain, and repeat until all elements have been removed. If the poset has width k, this will create a set of at most $log_{(k-1)/k}$ chains; if more chains are needed, we stop and answer that the width is greater than k.

Since each chain is produced in $O(n^2)$ time, the initial set of chains is found in $O(n^2 \log n)$ time. Closer analysis shows that finding the chains takes $O(n^2)$ time. Each chain is actually found in $O({n_i}^2)$ time, where n_i is the number of elements which have not been placed in previous chains. Since $n_{i+1} \leq n_i(k-1)/k$, this leads to a recurrence of the form $T(n) \leq T(((k\text{-}1)/k)n) + O(n^2)$, which is $O(n^2)$.

We use this initial chain partition to determine whether the poset has width k. We actually solve the matching problem on the split of this poset, as defined below. Let $B(P)$ have two vertices x_{in}, x_{out} for each element of P, and edges from x_{out} to y_{in} if and only if there is an edge from x to y in P. Problem 14.9 shows how to get the maximum independent set in P from the maximum matching in $B(P)$.

Our greedy chain decomposition gives a matching in $B(P)$ with $O(\log n)$ unmatched vertices; each x_{out} is matched to y_{in}, where y is the next vertex on x's chain. This leaves two unmatched vertices per chain; s_{in} and t_{out}, where s and t are the first and last vertices on the chain. Standard matching algorithms (see, for example, [198]) operate by finding augmenting paths, where each augmenting path is found in $O(n^2)$ time and adds one edge to the matching. Since we start with $O(\log n)$ unmatched elements, matching can be done for this special case in $O(n^2 \log n)$ time.

The time complexity can be reduced by adding one chain at a time to the graph, always adding back the unused chain which was found last by the greedy chain cover algorithm, and solving the matching problem on the current graph. Each chain added causes at most one new augmenting path, and the algorithm will again have time complexity governed by the recurrence $T(n) \leq T(((k\text{-}1)/k)n) + O(n^2)$, again yielding an $O(n^2)$ algorithm when k is fixed.

8.6 Exercises

1) Show that every cograph can be built using $O(n)$ union and complement operations.

2) Give an $O(n)$ algorithm for constructing a permutation graph representation of a cograph, if the cotree is given as input.

3) In section 8.2, we discussed a method for verifying in the randomized sense that a matrix multiplication $AB = C$ is correct; we assume for simplicity that all matrices are n by n. For standard multiplication, we can generate a vector x of length n, each entry of which is randomly chosen to be 1 or -1, and verify that $Cx = A(Bx)$.

We saw that this approach failed for verifying the result of a 0/1 matrix multiplication. The particular counterexample (introducing a 0 in C when A and B had only 1 entries) might be avoided if we require that x has an equal number of 0 and 1 entries. Show that in general, requiring x to have the same number of 0 and 1 entries is not sufficient if we want to verify the result of a 0/1 matrix multiplication.

4) Show how to determine whether a graph has a simplicial vertex in $O(n^\alpha)$ time, if two n by n matrices can be multiplied in $O(n^\alpha)$ time.

5) Section 8.1 describes a marking procedure for cotrees. A marked node is fully marked if it is a leaf, or if it is marked by all of its children. Neighbors of the new vertex x are marked. Each fully marked node (except for the root) marks its parent, and this procedure continues until there are no fully marked nodes which have not marked their parent. Show that the number of fully marked nodes in the cotree is proportional to the number of neighbors of the new vertex.

6) Turàn's theorem says that a triangle-free graph has at most $n^2/4$ edges. Use Turàn's theorem to show that every vertex in a claw-free graph has at most $2m^{1/2}$ neighbors.

7) This problem deals with the class of graphs which contain no induced P_5 in the graph or its complement. It is clear that these graphs contain no holes or antiholes of length greater than 5. This problem shows that 5-cycles occur only in a very restricted fashion for this class of graphs. Let C be a 5-cycle 1,2,3,4,5 in such a graph.

7a) Show that this class of graphs has $2^{\Theta(n^2)}$ members.

7b) Show that the set of vertices adjacent to at least one and at most 4 vertices of C form a module M.

7c) A clone of a vertex c in C, as described in section 8.5, is adjacent to the two neighbors of c in C, nonadjacent to the two nonneighbors of c in C, and may be adjacent or nonadjacent to c itself. Show that M has exactly 5 maximal submodules, corresponding to 'clones' of each vertex in C.

7d) Use the results from b and c and the fact that independent sets can be found in polynomial time on weakly chordal graphs to argue that independent set can be solved in polynomial time for graphs which contain no induced P_5 or its complement.

8) One approach to working with classes defined by forbidden pairs of subgraphs is to take small graphs which do not rule out P_4s, and see what we get by forbidding pairs of these subgraphs. The smallest such graphs would be C_3 and its complement. Show that the class defined in this way has only a constant number of members.

9) Consider the class of graphs without any induced C_3 or C_4. Show that there are too many graphs in this class to have an implicit representation.

10) Show that triangle-free, claw-free graphs have an implicit representation.

11) Show that there are $2^{o(n^2)}$ square-free (i.e. G does not contain C_4 as an induced subgraph), claw-free graphs.

12*) Show that there are $2^{\Omega(n^{1.5})}$ graphs which contain neither K_5 nor $K_{3,3}$ as induced subgraphs.

13a) Show that recognizing whether a graph contains C_k for $k > 4$ is at least as hard as determining whether a graph has a triangle. For this problem, you are allowed to transform a triangle graph recognition problem to a C_k-free recognition problem with $O(n)$ vertices, and $O(n^2)$ edges.

13b) For $k = 4$, I know only a slightly weaker result. Show that a linear time algorithm for recognizing C_4 free graphs would imply an $O(n^2)$ algorithm for triangle-free recognition, by tranforming a triangle-free recognition problem to a C_4-free recognition problem with $O(m)$ vertices and $O(m)$ edges. A transformation using a linear number of vertices seems to be an open problem.

14a) Show that determining whether G has a triangle is at least as hard as recognizing comparability graphs.

14b) Show that recognizing comparability graphs is at least as hard as recognizing triangle-free graphs.

15) This question could be asked about any fixed graph on 5 vertices; the following is one of the harder cases. The gem is equal to P_4 plus a universal vertex, i.e. a vertex adjacent to every node in the P_4. Show that induced gem-free graph recognition is as hard as triangle-free graph recognition.

CHAPTER 9

Chordal Bipartite Graphs

This chapter deals with chordal bipartite graphs and their generalizations. Chordal bipartite graphs are a useful and natural class of graphs with many interesting properties. We will see that these properties give rise to a form of representation which is fairly natural and is space optimal, but is not implicit. Chordal bipartite graphs are also interesting in that they are one of the most important classes of graphs which are known to have more than $2^{\Omega(nlogn)}$ and at most $2^{o(n^2)}$ members on n vertices.

A graph G is chordal bipartite if G is bipartite, and every cycle of length at least 6 has a chord. The name comes from the idea that these are the bipartite analogue of chordal graph. G is chordal if every cycle which can have a chord (that is, every cycle of length at least 4) has a chord. In bipartite graphs, only cycles of length at least 6 can have chords, so chordal bipartite graphs are bipartite graphs such that every cycle which could have a chord actually has a chord.

Chordal bipartite graphs are analogous to chordal graphs in several other ways. It is known [160] that G is chordal if and only if every minimal separator is a clique, and that a bipartite graph is chordal bipartite if and only if every minimal separator is a complete bipartite graph [224]. An analogue of the perfect elimination scheme characterizations of chordal graphs is contained in exercise 9.3.

9.1 Γ-free Matrices

There is a single important idea which gives rise to interesting representations and efficient algorithms for chordal bipartite graphs. The exact origin of this idea is hard to pin down precisely. It has been attributed to [264], [18], and [354]; I have been advised by several of these researchers that unraveling the mystery of who discovered this first is surprisingly complex.

A Γ in a matrix M is a pair of rows $i < j$ and columns $k < l$ such that $M[i,k] = M[i,l] = M[j,k] = 1$ and $M[j,l] = 0$. A matrix is Γ-free if it has no Γ. A Γ-free ordering of a matrix is a permutation of the rows and columns of the original matrix such that the resulting matrix is Γ-free.

A graph G is chordal bipartite if and only if there is a Γ-free ordering of the bipartite adjacency matrix. One direction of this proof follows immediately from exercise 9.3, which shows that if the bipartite adjacency matrix is Γ-free, there is a vertex which is not in the center of a P_5. No such vertex can be in a hole of size $>$ 5, so that vertex can be removed without eliminating any chordless cycle of size $>$ 5; we can repeat the process until all vertices are eliminated.

Furthermore, a particular ordering scheme called doubly lexical ordering will always produce a Γ-free ordering for a chordal bipartite graph, which leads to efficient algorithms for recognizing chordal bipartite graphs, as will be discussed in section 9.4. Since the proof that chordal bipartite graphs have Γ-free orderings

relies on the existence of a doubly lexical ordering algorithm, we defer this proof to the later section.

Note that some of the work I present here was done originally on the class of strongly chordal (also called sun-free chordal) graphs; these are graphs such that the adjacency matrix augmented by replacing 0s with 1s along the main diagonal has a Γ-free ordering, and are studied in chapter 12. Since representations of both chordal bipartite graphs and strongly chordal graphs come from the Γ-free property, for the purpose of this book almost all theorems on one class translate directly to the other class. I will point out some places where the two classes differ slightly in chapter 12. I choose to work almost exclusively with chordal bipartite graphs here, since we will introduce some natural generalizations which make sense in the bipartite context and give rise to interesting representation problems. However, one interesting method for representing these related classes of graphs is a restricted form of the subtree intersection model of chordal graphs. This representation, which is discussed in theorem 9.6 is more naturally stated on strongly chordal graphs, and is presented as such in the chapter.

The Γ-free property is very useful in designing algorithms, as we will discuss in section 9.5.

9.2 Counting and Representation

In this section, we will see that Γ-free ordering is crucial in designing a space optimal representation for chordal bipartite graphs. We will find that the number of chordal bipartite graphs is $2^{\Theta(nlog^2n)}$, and that there is a method of storing all the information necessary to reconstruct a Γ-free matrix using $O(nlog^2n)$ space.

We present the easier direction first; we will show that the number of Γ-free n by n matrices (and hence the number of chordal bipartite graphs with n vertices in each color class) is $2^{\Omega(nlog^2n)}$.

Theorem 9.1 *The number of chordal bipartite graphs is* $2^{\Omega(nlog^2n)}$.

Proof Let n be a power of 2. Consider how many $2n$ by $2n$ matrices ($2n$ is chosen to make the proof in this section simpler) we can construct using the following method. Divide the matrix into quadrants. Place 1s in every position of the bottom right quadrant. Choose any 'perfect matching' for the top left quadrant; that is, for each row we choose exactly one column in this quadrant to have a 1 entry, filling the rest with 0s, and making sure that no column has two 1 entries. The other two quadrants are constructed recursively using the same technique.

Each matrix constructed is Γ-free by induction. Clearly, a 1 by 1 matrix is Γ-free. Assume that all $k/2$ by $k/2$ matrices constructed using this technique are Γ-free, and let M be any k by k matrix built in this way. Suppose that M has a Γ; by induction, the Γ does not lie entirely within a single quadrant of M. Since there is only one 1 in each row and column of the upper left quadrant, the lower right entry of any Γ which crosses quadrants must be in the bottom right quadrant; this is impossible, since the bottom right quadrant contains no 0 entries.

There are $n!$ choices for the upper left quadrant. In the second stage of the recursion, there are 2 quadrants we are choosing, and we have $(n/2)!$ choices for the perfect matching. Continuing in this fashion, the number of matrices which can be constructed is $n! * (n/2)!^2 * (n/4)!^4 * ... * (n/n)!^n$.

$$\begin{matrix} 1 & 1 & 0 \\ 1 & * & 1 \\ 0 & 1 & * \end{matrix}$$

Figure 9.1 Example Matrix: Redundant 1s Marked with *

Reversing the order and using the fact that $x! \geq x^{x/2}$, we see that this product is at least $1^n * (2^1)^{n/2} * (4^2)^{n/4} * (8^4)^{n/8} * ... * ((n/2)^{n/4})^2 * n^{n/2} = 1^{n/2} * 2^{n/2} * 4^{n/2} * ... * (n/2)^{n/2} * n^{n/2}$. The logarithm of this product is $(n/2)*(0+1+2+3+...+\log n) = (n/2)(\log n(\log n+1))/2$. The logarithm of the number of matrices is at least a constant times $nlog^2n$, so the number of Γ-free n by n matrices is at least $2^{\Omega(nlog^2n)}$. Since each matrix corresponds to a different bipartite adjacency matrix for a chordal bipartite graph, the number of chordal bipartite graphs on $4n$ vertices is $2^{\Omega(nlog^2n)}$. □

The next definition will be used for designing a method of storing a chordal bipartite graph using $O(nlog^2n)$ bits, showing that the lower bound of theorem 9.1 is best possible.

The idea for a natural and space efficient representation for representing chordal bipartite graphs came from the idea of eliminating information from the Γ-free matrix which could be deduced in other ways. Consider storing the 2 by 2 matrix which has a 1 in all positions. We only need to know that the entries in positions (1,1), (1,2), and (2,1) have value 1; if we know that these three entries have value 1 and that the matrix is Γ-free, we can deduce that the entry in position (2,2) is 1. Extending this idea, a 1 entry is redundant in our representation if it can be deduced from other 1s in the matrix; a precise definition is given below. We will show that all but $O(n\log n)$ 1 entries in the matrix are redundant, and that the entire Γ-free matrix can be reconstructed if we know the positions of these $O(n\log n)$ 1 entries. Since the positions can be stored using $O(\log n)$ space per entry, we have a natural $O(nlog^2n)$ space representation. We will show how to find this set of nonredundant 1 entries efficiently, and how to reconstruct the graph from these entries.

We should note that 1s may be redundant as a result of other redundant 1s. For example, in figure 9.1, the nonredundant 1s are at positions (1,1), (1,2), (2,1), (2,3), and (3,2). However, there is no Γ if we replace the 1 at position (3,3) by 0 unless the redundant 1 at position (2,2) is included.

To make the definition of redundancy more precise, we define a 1 entry of a Γ-free matrix to be redundant if replacing the entry by a 0 creates a Γ in the matrix. It is not immediately obvious that storing only the positions of the nonredundant 1 entries allows us to reconstruct the matrix, since as shown in figure 9.1 we cannot simply check whether each entry forms a Γ together with three nonredundant 1s. The next theorem shows that a Γ-free matrix can be reconstructed efficiently from the set of nonredundant 1 entries.

The reduced matrix M' of a Γ-free matrix M is a 0/1 matrix such that the 1 entries of M' are exactly the nonredundant 1 entries of M.

Theorem 9.2 *A Γ-free matrix M can be reconstructed from its reduced matrix M'.*

Proof We reconstruct M from M' as follows. We start by making all 1 entries in M' 1 in M, with all other entries initially set to 0. Process the entries of M in lexicographic order; that is, the order (1,1),(1,2), ... ,(1,n),(2,1), ... (2,n), ..., (n,1), ... ,(n,n). If $M[i,j] = 1$ when it is processed, find the next 1 in row i of M at position (i,k), and the next 1 in column j of M at position (h,j), and set $M[h,k]$ to 1. The proof that every 1 entry we add is necessary for the result to be Γ-free is simple, and is left to the exercises.

Suppose that M is not reconstructed correctly. Let (i,j) be the lexicographically first 1 entry which is given the value 0 by the algorithm above, when it should have the value 1. Then replacing (i,j) with a 0 must create a Γ in M. Consider a 'minimal' Γ caused by replacing (i,j) with 0, in the sense that the rectangle spanned by this Γ does not contain the rectangle spanned by any other Γ. Formally, we can use a lexicographically largest pair (g,h) such that rows g and i, columns h and j form a Γ if (i,j) is replaced by 0. Clearly $g <$ i, $h < j$.

The algorithm will find a 1 in position (g,h), since (i,j) is the lexicographically first 1 value of M which is given a 0 by the algorithm. We will show that the algorithm fills in (i,j) as 1 when (g,h) is examined.

There cannot be any 1 in row g between columns h and j. Suppose (g,x) is a 1 entry where column x is between h and j. Then (i,x) must be a 1, or rows g,i and columns h,x induce a Γ. Replacing (i,j) by 0 creates a Γ induced by rows g and i, columns x and j; this is a smaller Γ than the Γ induced by g,i,h,j, contradicting the assumption that the Γ g,h,i,j is minimal. A similar argument shows that there are no 1s in column h between rows g and i.

When (g,h) is examined, (g,j) must already be known to be 1, since no position of row g will fill in a 1 in position (g,j), and (g,j) is set to be 1 by the algorithm ((i,j) is the first entry incorrectly left 0). Thus, when (g,h) is examined, (g,j) will be found as the next 1 in row g. No entry examined after (g,h) will fill in a 1 in position (i,h) since there is no 1 in column h to be the upper right corner of any Γ filled in. Therefore, when (g,h) is examined, (i,h) is already known to be a 1, and will be the next 1 in column h when (g,h) is examined.

When (g,h) is examined, (g,j) will be the next known 1 entry in row j, and (i,h) will be the next 1 entry in column h. The algorithm will fill in (i,j) with a 1, contradicting the assumption that there is a first 1 entry of the matrix which is not filled in correctly by the algorithm. □

The algorithm for reconstructing a Γ-free matrix from its reduced matrix is given in the proof above. As each 1 value, which corresponds to an edge of G, is visited, we need to be able to find the next 1 in the corresponding row and column, and possibly insert a new 1 value in the matrix. If the 1 values for each row and column are kept in a balanced tree data structure, each edge can be processed in O(log$n$) time. Using the fact that entries in the set are in the range 1..n, this can be reduced to O(loglog$n$) time per edge, giving an O($m$loglog$n$) algorithm for reconstructing the matrix, if we are given only the nonredundant 1s in the matrix.

Theorem 9.2 shows that we can reconstruct a chordal bipartite graph from its reduced matrix. In theorem 9.5, we will show that the reduced matrix has O($n$log$n$) 1 entries; when combined with theorem 9.1, this shows that storing the positions of nonredundant 1s from the bipartite adjacency matrix is a space-optimal representation.

	a	b	c	d	e
1	1	1	1	0	0
2	0	1	0	1	0
3	0	0	1	0	1
4	0	1	0	1	1
5	0	0	1	1	1

Figure 9.2 Example Matrix for Bitonic Paths

We present a simplification of the proof in [435]; the simplified proof is due to McConnell. We note that the proof that there are $2^{O(nlog^2n)}\Gamma$-free matrices is presented in greater detail than most proofs in the book. This is in part because this is the only proof that a class has $2^{f(n)}$ members for an f(n) which is not O($nlogn$) but is o(n^2), and in part because the proof is significantly different from the existing journal proof of the same result. In addition, part of this proof was written by Ross McConnell rather than myself; unfortunately, Ross found himself able to resist my attempts to make him a coauthor in the rest of the book.

Let G be a chordal bipartite graph, and let M be a Γ-free bipartite adjacency matrix of G. A *monotonic path* is any path of length at least two that descends monotonically in the two orders. That is, $(v_1, v_2, ..., v_k)$ is a monotonic path iff $k \geq 3$ and, for each i from 1 to $k-2$, the row or column v_i is always before v_{i+2} in M. A monotonic path corresponds to a zig-zagging sequence of ones in M, where any three consecutive ones in the sequence lie in the union of a row and a column, and the third is always strictly below and to the right of the first. A *bitonic path* is a path consisting of the union of two monotonic paths that share only their first edge. A *bitonic cycle* is a bitonic path, together with an edge joining its endpoints.

Consider the matrix of figure 9.2, which is not Γ-free. There is a monotonic path 1,*b*,2,*d*,5, and another monotonic path *b*,1,*c*,3,*e*. This forms a bitonic path 5,*d*,2,*b*,1,*c*,3,*e*; if we add (5,*e*), we get a bitonic cycle.

Lemma 9.3 *Let (r, c) and (r', c') be two edges of a bitonic path such that $r > r', c < c'$, i.e. (r, c) is below and to the left of (r', c'). If M is a Γ-free matrix, then (r, c') is a 1 entry of M.*

Proof Let P be a bitonic path for which this lemma is false. Let r be te lowest numbered row in such a violation, and let c be the first 1 entry of row r such that (r, c) is below and to the left of some (r', c') on P, where (r, c') is 0 in M. Choose r' to be as small as possible, and c' to be as small as possible, so that (r, c') is 0.

The edge preceding (r', c') in P cannot be of the form (r'_2, c'), since r' was chosen to be as small as possible for edges violating the condition. Thus, the edge preceding (r', c') is (r', c'_2). If $c'_2 > c$, r, c'_2 must be a 1 entry of M, since it is above and to the right of (r, c), and c' was chosen to be minimum among columns violating the condition. Since (r'_2, c'), (r', c') and (r, c'_2) are 1 entries in this case, (r, c') must be 1 since M is Γ-free. Therefore, we can assume $c'_2 < c$.

The edge preceding (r, c) cannot be of the form (r, c_2), since c was chosen to be minimal violating the conditions of the lemma; we thus assume there is an edge (r_2, c) of P. If $r_2 > r'$, the entry r_2, c' must be 1, since r_2, c is below and to the left

of r', c'; this forces (r, c') to be 1 to avoid a Γ induced by r_2, r, c, c'. We therefore assume $r_2 < r'$.

The entry r', c must be a 1, since r', c_2' is below and to the left of (r_2, c). However, this forces r, c' to be 1, or there is a Γ induced by r', r, c, c'. □

Lemma 9.4 *The reduced matrix of a Γ-free matrix has no bitonic cycle.*

Proof Let M' be a reduced matrix of a Γ-free matrix M, and let C be a bitonic cycle in M'. We will show that the bottom rightmost edge (r, c') of C is redundant in M, and thus does not appear in M'.

The neighboring edges of (r, c') in C must be (r, c) and r', c' for some $r > r', c < c'$. The edge preceding r, c is of the form (r_2, c); if $r_2 > r'$ then r_2, c' is a 1 by Lemma 9.3, and (r_2, c), (r, c), (r_2, c') make the entry (r, c') redundant. We therefore assume that $r_2 < r'$.

Similarly, the edge preceding r', c in C is of the form r', c_2'; if $c_2' > c$ then by Lemma 9.3 r, c_2' is a 1 in M, and the edges (r', c_2'), (r', c'), (r, c_2') make r, c' redundant. Therefore, we can assume that $c_2' < c$.

Since (r', c_2') is below and to the left of (r_2, c), Lemma 9.3 allows us to conclude that r', c is a 1 entry of M. Since (r', c), (r, c), and (r', c') are 1 entries of M, (r, c') must be a redundant 1, and cannot be an edge of a cycle in M'. □

Lemma 9.4 is used below to show that a Γ-free matrix has $O(n \log n)$ nonredundant ones. Since we can reconstruct M from the nonredundant ones, and the positions of the nonredundant ones can be represented with $O(nlog^2 n)$ bits, this allows us to give an upper bound on the number of Γ-free matrices which matches the lower bound of theorem 9.1.

Theorem 9.5 *There are $2^{O(nlog^2 n)}$ Γ-free matrices with n rows and columns.*

Proof Let M' be the reduced matrix of a Γ-free matrix M with n rows and columns. We now show that M' has $O(nlogn)$ nonzero entries. Since M' may be specified by using $O(logn)$ bits to give the location of each nonzero entry, M' can be stored using $O(nlog^2 n)$ bits. Since M may be deduced from M', this gives an $O(2^{O(nlog^2 n)})$ bound on the number of Γ-free matrices.

To show that M' has $O(nlogn)$ nonzero entries, we show that the nonzero entries may be partitioned into $O(\log n)$ sets, $X_1, X_2, \ldots, X_k$, each of which has $O(n)$ members. We do this inductively. Suppose $X_1, X_2, \ldots, X_i$ have been selected, where i is even. To obtain X_i, we select the first one in each row that is not already in one of the sets. To obtain X_{i+1}, we select the first two ones in each column that are not already in $X_1 \ldots X_{i+1}$.

Next, at each nonzero entry e of M', we may define a tree rooted at e. Suppose e is in X_i. If i is even, its children are the two members of its column that lie in X_{i-1}. Note that these two children lie above it. If i is odd, then we let the sole child of e be the member of X_{i-1} that lies in its row. Note that this child lies to its left. The subtrees rooted at children of e are obtained recursively.

Let T be the tree rooted at a node in X_i, where i is even. Suppose two descendants d_1 and d_2, both at depth $k > 1$ in T, are in the same column c_1 of the matrix. Let (r_1, c_1) be the position of d_1, and let (r_2, c_1) be the position of d_2; without loss of generality assume $r_1 < r_2$.. Let a_1 and a_2 be the children of the least common ancestor *lca* of d_1 and d_2 in T, where a_1 is an ancestor of d_1 and a_2 is an ancestor of d_2. Since a_1 and a_2 are children of a common node that has two

children in T, they both lie in the same column c_2. Let (r_3, c_2) be the position of a_1 in M', and let (r_4, c_2) be the position of a_2. Finally, let the least common ancestor be located at (r_5, c_2)

Let C be the union of the following paths. Let P_1 be the path $c_1, r_1, ..., r_3, c_2, r_5$, where the path from r_1 to r_5 follows the path from d_1 to lca in T. Let P_2 be the path $r_1, c_1, r_2, ..., r_4, c_2$, where the path from r_2 to c_2 follows the path from d_2 to lca in T. Each of these paths is monotonic, so their union, when combined with the edge (r_5, c_2), is a bitonic cycle. This is a contradiction, by Lemma 9.4.

We may conclude that no two nodes at the same depth of a subtree rooted by any entry e are in the same column of the matrix. Consider a node x in X_{2i}. Node x has 2^j descendants at depth $2j$, each of which is in X_{2i-2j}. There can be at most n descendants at a particular depth, since none share a column column; this implies that $i \leq 2logn$. Since each set X_i clearly contains $O(n)$ entries, the total number of 1s in the reduced matrix is $O(nlogn)$. □

We now know that the reduced Γ-free bipartite adjacency matrix is a space efficient technique for storing a chordal bipartite graph. One natural question is how quickly we can construct this representation. Constructing a Γ-free ordering is the topic of section 9.4; we now show that the reduced matrix can be obtained from the Γ-free ordering in $O(n^2)$ time, an improvement on the obvious $O(n^4)$ algorithm of checking whether each pair of rows and columns to see whether the bottom right 1 entry can be marked as redundant.

The algorithm is essentially the reversal of the process of reconstructing M from the reduced matrix discussed in theorem 9.2; in fact, both algorithms are derived from the algorithm in [354] for determining whether a matrix is Γ-free, which is discussed in section 9.4. Given a matrix M, look at each 1 value in the matrix. If the position of the 1 value is (i, j), we look for the next column k which has a 1 for row i, and the next row l which has a 1 in column j, and mark the 1 value in position (l, k) as a redundant 1.

Every 1 marked as redundant by this algorithm must be redundant, since we find a Γ which has each such element at the bottom right. If (k, l) is redundant, consider the smallest Γ in the sense of theorem 9.2 if the entry at position (k, l) is replaced by a 0. Let (i, j) be the position of the upper left corner of this Γ. If there is a 1 entry in row i, column $j2$, where columns $j < j2 < l$, then the entry at position $(k, j2)$ must be 1 (or rows (i, k) columns $(j, j2)$ induce a Γ), which implies that (i, k) and columns $(j2, l)$ induce a smaller Γ. A similar argument can be used to show that there is no 1 in column j between rows i and k. Therefore, (i, j) will mark (k, l) as a redundant 1.

Representing a chordal bipartite graph by storing the position of all nonredundant ones may be space efficient, but it is not satisfactory for a number of reasons. It is inefficient for testing adjacency, since we must reconstruct the entire matrix in order to test adjacency between a single pair of elements. Since adjacency testing does not rely only on bits stored at individual vertices, this class poses a challenge to the generalized implicit graph conjecture.

Open Problem 9.1: Is there a generalized implicit representation of chordal bipartite graphs?

We note that Dahlhaus [142] has shown that it is possible to add $O(nlog^3n)$ bits to this representation scheme, and test for adjacency between vertices of G

in O(log$n$) time for chordal bipartite graphs; the data structure can be computed in O($nlog^3n$) time from the reduced matrix. This is not a local representation; to compute adjacency between x and y, one must use bits which are globally accessible.

The problem below deals with the notion of partial order dimension, as discussed in chapter 3. There are a number of candidate forms of generalized implicit representations of chordal bipartite graphs. I choose to focus on dimension because dimension is a well studied parameter, and determining that the dimension of a chordal bipartite graphs grows very quickly, which would be discouraging if we are only looking for an efficient representation, would have independent significance.

Open Problem 9.2: What is the maximum dimension as a function of n of a partial order which corresponds to a chordal bipartite graph?

Another perspective on representing chordal bipartite graphs comes from trying to represent the class of graphs constructed in theorem 9.1 efficiently, since this is a large class of chordal bipartite graphs. These were graphs such that the bipartite adjacency matrix could be constructed by placing any perfect matching in the upper left quadrant, all 1 entries in the lower left quadrant, and filling the other two quadrants applying the same technique recursively. It is not at all obvious that this subclass has small dimension. However, there are easy ways to represent this particular subclass efficiently. For example, let the rectangle number of a graph be the minimum number of isooriented rectangles per vertex so that G is the intersection graph of these rectangles. In exercise 9.8, the reader is asked to show that the rectangle number of the graphs constructed in theorem 9.1 is O(log$n$).

Open Problem 9.3: What is the maximum rectangle number of a chordal bipartite graphs, as a function of n?

9.3 Characterizations

Even if we had an implicit representation for chordal bipartite graphs, that representation might not be a characterization. For example, even if the maximum dimension of a chordal bipartite graph is O(log$n$), there would be bipartite graphs which are not chordal bipartite and have dimension O(log$n$), since a simple cycle of length $2k$ has dimension at most 3. The fact that G is chordal bipartite if and only if the bipartite adjacency matrix has a Γ-free ordering is a form of characterization, but is far different from the intersection representation characterizations discussed in chapter 3. In this section, we discuss two other interesting characterizations of chordal bipartite graphs, which are not as well known as some other forms of characterization.

Ideally, we would like a generalized implicit representation which characterizes the class exactly, in the sense that G has a representation if and only if G is in the class. No such representation is known at this time. However, there are two tree models which are exact characterizations of the chordal bipartite graphs. Both of these characterizations use $\Omega(n^2)$ space, but they are interesting nevertheless, and may help some reader design a new space efficient form of characterization.

The first of the two characterizations is from [184]; it is actually a characterization of the related class of strongly chordal graphs. The relationship between strongly chordal graphs and chordal bipartite graphs is discussed in more detail in chapter 12; in particular, a bipartite graph G is chordal bipartite if and only if the graph formed by turning one color class into a clique is strongly chordal. This gives

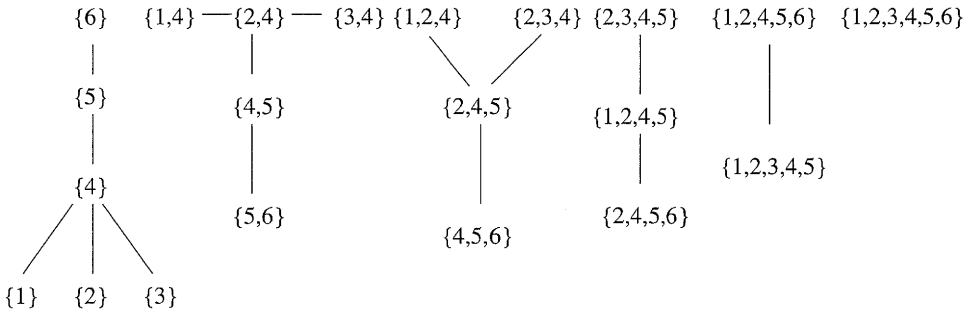


Figure 9.3 Tree Sequence in Characterization

an intersection model for the class by restricting the intersection model of chordal graphs discussed in chapter 15.

Theorem 9.6 [184] *G is strongly chordal if and only if G is the intersection graph of subtrees of a weighted, rooted tree such that for every pair of intersecting subtrees T_x, T_y in the model, vertices in the intersection of $T_x \cap T_y$ are exactly the closest vertices to the root in one of the trees T_x or T_y.*

The second characterization is due to Lehel [339]. This is originally given as a characterization of totally balanced hypergraphs. Connections to graph classes come from the fact that the hypergraph defined by the adjacency matrix of G is totally balanced iff G is strongly chordal, and the hypergraph defined by the bipartite adjacency matrix of G is totally balanced iff G is chordal bipartite; see [74] for details.

It may be helpful to examine the sequence of trees in figure 9.3 before looking at the definition. Given a hypergraph H, there is a sequence of trees T_0,T_1,...,T_{n-1}. Each T_i is a tree on n - i nodes, and each node of T_i corresponds to a set of i+1 vertices of H. The sequence of trees obeys the following rules.

i) Nodes of T_0 correspond to vertices of H. For $i > 0$, each node of T_i comes from a distinct edge e of T_{i-1}, and corresponds to the union of the vertices from either endpoint of e.

ii) Each edge of a tree connects two nodes x,y such that the set corresponding to x can be obtained from the set corresponding to y by replacement of exactly 1 vertex.

iii) Every hyperedge e of H corresponds to some node of one of the trees.

In terms of representing a chordal bipartite graph, the theorem says that for every vertex in X, the set of vertices adjacent to X is exactly the vertices corresponding to some node of one of these trees.

Lehel [339] shows that a hypergraph H is totally balanced if and only if such a sequence of trees exist. I find this to be an elegant characterization, but do not see how to make it space efficient. If we provide the entire sequence of trees, we have used $\Theta(n^2 \log n)$ space. Perhaps some reader will find a way to show how to get from one tree to the next using less space, giving an alternate proof of an upper bound on the number of chordal bipartite graphs. I am not optimistic about this leading to a generalized implicit representation, however; even if the sequence of trees can be encoded efficiently, testing adjacency between vertices seems to depend on much more than the index of nodes in the sequence of trees.

Some characterizations of chordal bipartite graphs which are not defined as forms of representation are given in the exercises. One characterization which was too difficult to phrase as an exercise is due to [164]; they show that a bipartite graph G is chordal bipartite if and only if every induced subgraph H of G contains vertices i and j, where j is a neighbor of i, $N(j)$ contains the neighborhoods of all other neighbors of i (with these neighborhoods restricted to H). This corresponds directly to a characterization of strongly chordal graphs, once again showing the connection between these classes. The fact a bipartite graph G is chordal bipartite if and only if the complement of G is perfectly orderable as studied in chapter 14 was first noted in [107], and is the subject of exercise 14.23.

9.4 Recognition

There is a polynomial time recognition algorithm for chordal bipartite graphs as a consequence of their definition, since we can determine whether an arbitrary graph has an induced cycle of size at least k in $O(n^k)$ time. However, we can do much better; the best known time bounds for this problem are $O(m\log n)$ or $O(n^2)$ [354, 391, 438].

As we will see in chapter 13 for chordal graphs, algorithms for chordal bipartite recognition are divided into two phases. A construction phase produces an ordering of the bipartite adjacency matrix, which may or may not be Γ-free, and a verification phase checks whether the ordering is in fact Γ-free. In the first algorithm designed specifically for recognizing chordal bipartite graphs, both phases took $O(n^3)$ time. Lubiw [354] showed that the verification phase could be solved easily in $O(n+m)$ time as follows. The algorithm is very similar to the algorithm for removing nonredundant 1s from a representation described in section 9.2.

For each 1 entry (i,j) in the matrix, let k be the next row such that the entry at position (k,j) is 1, and let l be the next column such that the entry at position (i,l) is 1. We check that the entry at position (k,l) is a 1. If all these values are 1, then the matrix is Γ-free; the proof is similar to the proofs mentioned above from section 9.2, showing that the smallest Γ in the matrix will be found in this way. If we are given the matrix in the form of an ordered adjacency list, the verification phase can be performed in $O(n+m)$ time; this is easy given the result of exercise 1.2, which allows constant time adjacency testing with input in adjacency list form, if one is willing to use $\Theta(n^2)$ space.

We will describe an $O(n^2)$ algorithm for taking a 0/1 matrix and producing a Γ-free matrix by permuting rows and columns, if such a permutation is possible. The technique used is called doubly lexical ordering. A doubly lexical ordering of an i by j matrix is an ordering of the rows and columns such that if the columns and rows are viewed as i or j digit numbers, both rows and columns occur in increasing order. Unfortunately, for historical reasons, we must read these numbers from right to left and bottom to top; that is, the most significant digit for rows is the rightmost column, and the most significant digit for columns occurs in the bottom row. An example of a matrix and a doubly lexical ordering of the matrix is shown in figure 9.4. Note that this matrix has a doubly lexical ordering, despite the fact that it has no Γ-free ordering; rows 3 and 5, and columns 1 and 2, induce a Γ in the doubly lexical ordering.

The critical theorem on chordal bipartite graphs, attributed variously to [18, 354, 264], states that a matrix can be permuted to be Γ-free if and only if every

	1	2	3	4	5
1	1	0	1	1	0
2	0	1	1	0	0
3	1	1	1	0	0
4	0	1	0	1	1
5	1	0	0	1	0

	3	1	5	2	4
2	1	0	0	1	0
3	1	1	0	1	0
5	0	1	0	0	1
1	1	1	0	0	1
4	0	0	1	1	1

Figure 9.4 Doubly Lexical Ordering

doubly lexical ordering is Γ-free. This led to the development of a series of doubly lexical ordering algorithms. The first to appear [264] had time complexity $O(i^2j)$, where i is the number of rows and j is the number of columns. This was improved to $O(mlog^2n)$ by Lubiw [354] where m is the number of nonzero entries and n is the total number of rows and columns. Paige and Tarjan improved the time bound to $O(mlogn)$. All these algorithms are designed to create a doubly lexical ordering of an arbitrary matrix M, which is not constrained to be 0/1. We will describe an algorithm which works only for the case of 0/1 matrices, with time complexity $O(n^2)$. This algorithm is described in [438], and can be viewed as a particular implementation of Paige and Tarjan's algorithm for the 0/1 case. We will need the $O(n^2)$ time bound for a number of results in chapter 13, where chordal bipartite graphs are used to recognize circular-arc graphs and trapezoid graphs.

As noted above, at a high level algorithms for chordal bipartite recognition resemble algorithms for chordal graph recognition. Both algorithms have a phase devoted to construction of an ordering, and another phase for verifying the ordering. The construction phases use partitioning techniques. To construct a perfect elimination ordering for a chordal graph, we choose any vertex x to go last in the ordering, divide all sets into neighbors and nonneighbors of x, placing neighbors after nonneighbors; any vertex from the last set is chosen as we repeat this procedure. The algorithm for doubly lexical ordering is similar, but it is no longer true that an arbitrary vertex can be chosen to go last. It is easy to see that if $N(x)$ is a proper subset of $N(y)$, x must come before y in a doubly lexical ordering.

Hoffman, Kolen, and Sakarovich dealt with this problem by choosing a special vertex which could go last at each step. Their algorithm successively chooses rows to go last in the row set, and then partitions the column set. For each row r, they keep a tuple corresponding to the number of vertices in each set of the partition which r is adjacent to, and the row with largest lexicographic tuple value (with numbers read from right to left) is chosen to go last and partition the columns.

Let us demonstrate the algorithm on the matrix of figure 9.5. Originally, the columns are in a single block (1,2,3,4,5); row values are 1:3 2:2 3:3 4:3 5:2. Row 1 may be chosen to go last; this divides the columns into neighbors and nonneighbors of row 1 (2,5)(1,3,4). The new values for the remaining rows are 2:1,1 3:1,2 4:2,1 5:0,2. Row 3 has the 'largest' value, and is chosen next; the new column partition is (5)(2)(4)(1,3). Row values become 2:0,1,0,1 4:1,1,1,0 5:0,0,1,1; thus row 5 is chosen next. The column partition becomes (5)(2)(4)(3)(1). Remaining row values are 2:0,1,0,1,0 4:1,1,1,0,0. Thus the resulting matrix is shown in figure 9.5.

	1	2	3	4	5
1	1	0	1	1	0
2	0	1	1	0	0
3	1	1	1	0	0
4	0	1	0	1	1
5	1	0	0	1	0

	5	2	4	3	1
4	1	1	1	0	0
2	0	1	0	1	0
5	0	0	1	0	1
3	0	1	0	1	1
1	0	0	1	1	1

Figure 9.5 Sample Matrix for HKS Algorithm

Some readers may note a similarity to the description of lexicographic breadth first search in [223]. The drawback of this method is that we now have to use time in choosing a vertex which goes last; this increases the time complexity from linear for chordal graphs to n^3 for chordal bipartite graphs.

The algorithms of [354, 391, 438] are based on the same idea, but avoid the bottleneck step of searching for a vertex which can go last; we choose to describe the algorithm from [438]. We keep a current partition of both the rows and the columns. Any row r from the last set of rows is used to partition the last set of columns. However, r cannot be placed at the end of the row list; other rows may have edges to all of r's edges and more. If we call the neighbors of r from the last block of the column partition C_r, we will end up by placing r after all rows from the last block which have any nonneighbors in C_r, before any rows from the last block whose neighbors in the last block properly contain C_r, and together with any rows from the last block whose neighbors in the last block equal C_r.

Let R be the last current block of the row partition, and let C be the last block of the column partition which is not marked as having been split by R. If R has split every block, we remove R from further consideration, placing rows in R in any order on the output list. Rows of R are chosen for processing in any order. Let r be the current row. We step from back to front through subblocks of C which have not previously been split by r until we reach a block B which contains a nonneighbor of r; B is split into N(r) and B - N(r), with N(r) placed after r in the partition list. Store the fact that r has split all blocks which do not precede B. After all rows of R have been processed, divide R on the basis of how many consecutive rows at the end of C are adjacent to the row; rows adjacent to more columns are placed later in the ordering. Each vertex placed in the same subblock of R' has its last nonneighbor from C in the same subblock C' of C; mark R' as having split all blocks which do not precede C'. If the vertices of R' are adjacent to all vertices of C, mark R' as having split all blocks which do not precede C.

Let us examine the algorithm with the input matrix of figure 9.4. Initially, the row partition is (1,2,3,4,5) and the column partition is (1,2,3,4,5). We might choose row 5 first, and obtain the column partition (2,3,5)(1,4). Row 4 might be chosen next, giving the column partition (2,3,5)(1)(4); note that row 4 does not reach the first block of the column partition, and therefore does not split this block. Rows 2 and 3 are nonadjacent to column 4, and do no further splitting. Row 1 is adjacent to both 1 and 4, and thus splits the block (2,3,5), leaving a column partition (2,5)(3)(1)(4). The new row partition becomes (2,3)(4)(5)(1). Row 1 has

split all blocks, and is done; the same is true for row 5. Row 4 now splits earlier blocks; since it is adjacent to both 2 and 5, it does no further splitting. Rows 2 and 3 now compare themselves to the next block (1) of the column partition; the new row partition becomes (2)(3)(4)(5)(1). Row 3 splits earlier blocks of the column partition, and the final column partition becomes (5)(2)(3)(1)(4); the row and column orderings define a doubly lexical ordering of the matrix.

For any pair of vertices r, c, the block containing c is split by r only once, so the total time spent splitting blocks is $O(n^2)$. Sometimes, r is contained in a block R which is used to refine a block C, even though r has already split C. When this happens, we must find the last subblock of C which contains nonneighbors of r, in order to place r in the correct row block R'. This does not cause any problems with respect to the $O(n^2)$ time bound, since any block examined by r at this time will be marked as split by R', so vertices in these blocks will never be compared to r later in the algorithm.

It is interesting to note that none of the algorithms for recognizing chordal bipartite graphs use the fact that G must have a Γ-free ordering during the construction phase of the algorithm. We would be happy to have a linear time algorithm which performed doubly lexical ordering if G was chordal bipartite, and created any ordering if G was not chordal bipartite. The analogy here is with the comparability graph recognition algorithm, which produces an ordering in linear time which is transitive if and only if G is a comparability graph. I feel that there should be a linear time algorithm for chordal bipartite graph recognition based on this approach; note that we can verify that a matrix is Γ-free in linear time.

We now outline the proof that if G is chordal bipartite, every doubly lexical ordering of the bipartite adjacency matrix is Γ-free. Let vertices 1,2 and a, b form a Γ in the doubly lexical ordering, with $1 < 2$ and $a < b$. Since row b comes after row a but 2 is adjacent to a and not b, there must be a vertex 3, $3 > 2$, such that 3 is adjacent to b but not a; similarly, some vertex $c > b$ must be adjacent to 2 but not to 1. We choose vertices 3 and c which are maximum in the ordering meeting these requirements; vertices 3 and c must be nonadjacent or the vertices induce a C_6.

This forms the basis of an inductive proof, in which we have a P_k with the last row and column being inconsistent with a doubly lexical ordering on the induced vertices, necessitating a new later row and column. The completion of the proof that every doubly lexical ordering of a chordal bipartite graph is Γ-free is left to exercise 9.9.

9.5 Optimization Problems on Chordal Bipartite Graphs

A number of problems can be solved efficiently on chordal bipartite graphs due to properties of Γ-free matrices. These will have some consequences for other classes as well, since various problems on such classes as circular-arc graphs [177] and weakly chordal comparability graphs can be transformed to problems on chordal bipartite graphs.

The neighborhood containment problem for chordal bipartite graphs will be especially important for speeding up a number of algorithms. In the neighborhood containment problem, we want to determine for all pairs of vertices x, y whether $N(x)$ is a subset of $N(y)$. In general, the best known neighborhood containment algorithm uses matrix multiplication. Since $A^2[x, y]$ is equal to the number of paths

of length 2 between x and y, $N(x)$ is a subset of $N(y)$ if and only if $A^2[x,y]$ is equal to the degree of x. This algorithm for computing all neighborhood containments in a general graph takes $O(n^{2.376})$ time if very complex algorithms are used [118].

If we are given a doubly lexical ordering of the bipartite adjacency matrix, it is very simple to determine neighborhood containment for a chordal bipartite graph. Suppose that x comes before y in the ordering. Let b be the first neighbor of x in the ordering of the other color class. If y is nonadjacent to b, then $N(x)$ cannot be a subset of $N(y)$. If y is adjacent to b, then $N(x)$ is a subset of $N(y)$, since if there was any later vertex c adjacent to x but not y, then x,y,b,c induce a Γ. If x is after y in the doubly lexical ordering, $N(x)$ is a subset of $N(y)$ if and only if $N(x) = N(y)$. We can determine all 'twin' vertices in linear time for an arbitrary graph (exercise 2.17); alternatively, it is easy to compare successive rows to identify twins from the doubly lexically ordered matrix. Therefore, neighborhood containment in chordal bipartite graphs can be found in $O(n^2)$ time, using graph theoretic algorithms.

The author has used this observation to speed up a number of algorithms on graphs which had neighborhood containment as a bottleneck step.

In chapter 13, we will see some applications of fast computation of neighborhood containments in chordal bipartite graphs. Neighborhood containment calculation was the key bottleneck step in recognizing circular-arc graphs [177]. We were able to transform the problem in this case to computing neighborhood containments on a chordal bipartite graph, achieving an $O(n^2)$ algorithm for this problem. Cogis' algorithm for recognizing graphs with Ferrers dimension 2 [114] had computation of containment relationships as a bottleneck steps, and we were able to show that this could be performed on a chordal bipartite graph, reducing the time complexity of Ferrers dimension 2 digraph recognition, interval dimension 2, or trapezoid graph recognition (these problems are all closely related [357]) to $O(n^2)$.

In chapter 7, we saw a close relationship between chordal bipartite graphs and weakly chordal comparability graphs, which allowed a number of problems including recognition, independent set, and transitive closure and reduction of the corresponding class of directed graphs to be solved efficiently on weakly chordal comparability graphs [176]. The observation that neighborhood containment can be calculated efficiently was also used in an algorithm to compute shortest paths efficiently in strongly chordal graphs [250]. An extension of fast shortest path algorithms to to the larger class of dually chordal graphs, which are studied in chapter 12, is given in [68].

Let x be the first vertex from one of the color classes in a Γ-free ordering. It is easy to show that the neighborhoods of neighbors of x are totally ordered by the subset relation. This gives a nice characterization of chordal bipartite graphs which is analogous to the perfect elimination scheme characterization of chordal graphs. A bipartite graph is chordal bipartite if and only if we can repeatedly remove a vertex which is not in the center of a P_5; note that such vertices can clearly not be part of a chordless cycle of length greater than 5, just as simplicial vertices cannot be part of a chordless cycle of length greater than 3.

The property above immediately gives efficient algorithms for some problems. Consider the matching problem on chordal bipartite graphs. The simple greedy algorithm of repeatedly matching the first remaining row in the ordering with the first adjacent column will always produce a maximum cardinality matching for a chordal bipartite graph. In chapter 12, we study the related class of strongly chordal graphs, in which the augmented adjacency matrix formed by replacing all

diagonal elements by 1s has a Γ-free ordering. Domination is easy on this class, since the neighborhoods of neighbors of the first row are ordered by set containment; this element should clearly be dominated by its last neighbor in the ordering. By contrast, dominating set is NP-complete for chordal bipartite graphs, while the variant in which one color class must be dominated optimally is polynomial for chordal bipartite graphs by using maximum neighbors.

9.6 Variants of Chordal Bipartite Graphs

The following generalization of chordal bipartite graphs seems natural, but has not been studied previously. For any fixed value k, we can define a class of bipartite graphs which allows no induced cycles of length greater than k. Counting, representation, recognition (beating n^k or $O(n^{k-3})$ matrix multiplications)), and optimization problems all seem to be open for these extensions of chordal bipartite graphs.

From the large set of problems defined in this way, I would like to choose one which seems particularly interesting.

Open Problem 9.4: Consider the classes of bipartite graphs defined by forbidding induced cycles of length greater than k. How quickly does the number of graphs in the class increase as k increases?

Note that by theorem 8.3, for any fixed k the number of bipartite graphs with no induced cycles of length greater than k is $2^{o(n^2)}$.

Another natural variant of chordal bipartite graphs forbids induced paths on k vertices in a bipartite graph, rather than forbidding long induced cycles. When k is small, the classes defined are quite restricted. Forbidding P_4 in a bipartite graph allows only disconnected complete bipartite graphs.

Forbidding P_5 in connected bipartite graphs gives only the class called chain graphs, in which neighborhoods of vertices in a color class are ordered by set containment. Suppose a connected bipartite graph is not a chain graph; then there is a pair x_1, x_2 such that x_1 is adjacent to y_1 while x_2 is not, and that x_2 is adjacent to y_2 while x_1 is not. There cannot be a common neighbor y_3 of x_1 and x_2, or the vertices induce a P_5, but that implies that the shortest path from x_1 to x_2 involves at least five vertices, and must contain P_5, This characterization (G is bipartite P_5-free if and only if every connected component is a chain graph) makes it possible to efficiently recognize and implicitly represent graphs in the class, as shown in exercise 9.20.

However, forbidding P_6 gives a class which is incomparable with chordal bipartite graphs; P_6 is chordal bipartite, while C_6 contains no induced P_6. The class obtained by forbidding P_k in a bipartite graph for k is potentially quite interesting. Since each class is defined by a single forbidden subgraph, as opposed to forbidding long induced cycles which correspond to forbidding a set of subgraphs, they fit into a more general set of graph classes discussed in section 9.9. Nevertheless, classes defined by forbidding long induced paths seem particularly natural and worthy of more study.

In chapter 10, we discuss a different form of variant of chordal bipartite graphs, by looking at other forbidden submatrices besides the Γ.

9.7 Subclasses of Chordal Bipartite Graphs

This section discusses a well known hierarchy of subclasses of chordal bipartite graphs. These classes all have implicit representations as a consequence of their definitions. Nevertheless, I feel that the classes are not covered adequately in any current text, and are of sufficient interest to potential readers of this book that I choose to include a section discussing them.

These graph classes have often been been explored while attempting to sharpen the boundary of when a problem becomes NP-complete. Many problems have been shown to be NP-complete on bipartite graphs, and researchers have investigated subclasses of bipartite graphs. One important subclass is the class of chordal bipartite graphs. Many problems remain NP-complete on chordal bipartite graphs, however, so subclasses of chordal bipartite graphs have also been studied. Three important subclasses studied in this context are the bipartite permutation graphs, the biconvex graphs, and the convex graphs.

9.7.1 Bipartite Permutation Graphs. A graph is a bipartite permutation graph if it is both bipartite and a permutation graph.[1] It is not hard to see that bipartite permutation graphs are a proper subclass of chordal bipartite graphs. No permutation graph (and more generally, no co-comparability graph) can contain the complement of a chordless cycle of length greater than four, so bipartite permutation graphs are a subclass of chordal bipartite graphs. There are trees which are not bipartite permutation graphs, so the bipartite permutation graphs are a proper subclass.

Bipartite permutation graphs have been shown to be useful for characterizing a variety of mathematical objects. Visibility graphs of towers, which are polygons consisting of two reflex chains which share a common endpoint, can be characterized as bipartite permutation graphs plus a Hamilton cycle corresponding to edges of the polygon [116]. A bottleneck Monge matrix, as studied in [85, 300] satisfies the property that $\max(a_{ij}, a_{rs}) \leq \max(a_{is}, a_{rj})$ for $i \leq r, j \leq s$; various optimization problems can be solved efficiently if the input is bottleneck Monge [85]. A 0/1 matrix is bottleneck Monge if and only if its bipartite complement is strongly ordered as defined later in this section [100]; [300] uses this relationship to bipartite permutation graphs as a tool to recognize general (i.e. not necessarily 0/1) bottleneck Monge matrices.

Bipartite permutation graphs are also equal to bipartite co-comparability graphs, bipartite AT-free graphs, and bipartite bounded tolerance graphs [336]. They also correspond to bipartite graphs with color classes X and Y such that $x \in X$ is adjacent to $y \in Y$ if and only if the difference in weight between x and y is no more than some threshold t [76], as specified in the application of [155] which gave rise to the notion of a convex graph described below.

[1] This simple definition caused the strangest 'laugh line' I have ever seen in a conference. Immediately before a talk entitled Bipartite Permutation Graphs, Terry McKee gave a talk of a type he calls 'meta-graph-theory'. He disliked the notion that, for example, a bipartite permutation graph is bipartite and a permutation graph, but a chordal bipartite graph is not a chordal graph. He proposed a unifying definition, that an X-bipartite graph should mean that the square of the graph was an X-graph, and discussed what this would be equivalent to for various properties X. The speaker in the next talk had been reviewing the slides outside the room during Terry's talk, and started by saying that they would discuss bipartite permutation graphs, which were of course bipartite and permutation graphs. Most of the audience laughed loudly at this, leaving the speaker and those who missed the preceding talk quite bewildered!

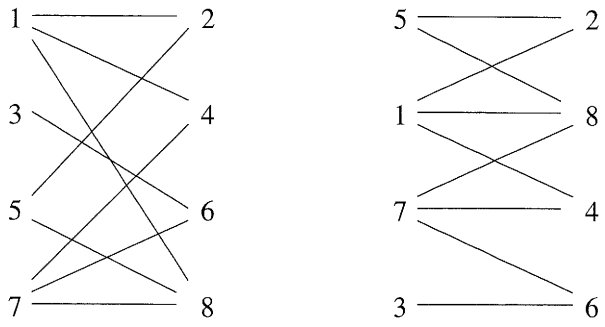


Figure 9.6 Bipartite Permutation Graph, and a Strong Ordering

A bipartite graph has a strong ordering if the two color classes can be ordered $x_1, x_2, .., x_i$ and $y_1, y_2, .., y_j$ such that whenever you have two 'crossing edges' (x_a, y_d) and (x_b, y_c) such that $a < b, c < d$, you must also have the edges (x_a, y_c) and (x_b, y_d). The ordering portrayed on the left side of figure 9.6 is not a strong ordering; the crossing of edges (1,8) and (5,2) is not a problem, since (1,2) and (5,8) are edges, but a crossing such as (3,6) and (7,4) is a violation, since (3,4) is not an edge. The graph does admit a strong ordering, as shown on the right side of the figure.

Theorem 9.7 [445] *A bipartite graph is a bipartite permutation graph if and only if it has a strong ordering.*

Proof We first show that every bipartite permutation graph has a strong ordering. If we assume that we are given a permutation diagram for G, ordering vertices from X by their positions on the first parallel line L_1 and vertices of Y by their positions on the second parallel line L_2 gives a strong ordering. Consider any pair of edges (x_a, y_d) and (x_b, y_c) as in the statement of a strong ordering.

Since x_a is nonadjacent to x_b and y_c is nonadjacent to y_d, x_a precedes x_b on both L_1 and L_2 while y_c precedes y_d on both lines. If y_c precedes x_a on L_1, we have y_c, x_a, x_b on L_1 and x_a, x_b, y_c, y_d on L_2; since x_a is adjacent to y_d, we must have y_d between y_c and x_a on L_1, and we must have both edges required in the theorem. Similarly, if y_c is after x_b on L_1, the orders must be x_a, x_b, y_c, y_d and y_c, y_d, x_a, x_b, giving the extra edges. We cannot have y_c between x_a and x_b on L_1; this would force y_d to occur after x_a on both L_1 and L_2, but these vertices must be adjacent.

Now assume that there is a strong ordering of a bipartite graph. We may assume the graph is connected; otherwise, each component can be shown to be a permutation graph. Place endpoints of X vertices on L_1 as in the strong ordering, and endpoints of Y vertices on L_2 as in the strong ordering. Place endpoints of X vertices on L_2 after their last neighbor in the ordering of Y vertices; if x_a and x_b have the same last neighbor, place x_a before x_b on L_2. Similarly, place endpoints of Y vertices on L_1 after their last neighbors, with tied vertices being ordered as in the strong ordering. We will show that this is a legitimate permutation diagram for the bipartite graph.

Clearly, if x and y are adjacent, x is placed before y in L_1 and y is placed before x in L_2. Suppose that the segments corresponding to x and y cross in the diagram. If x precedes y on L_1, there must be x_b which occurs after x in the ordering with

x_b adjacent to y, and y_d after y in the ordering with x adjacent to y_d, Since the ordering is a strong ordering, we must also have an edge from x to y. Similarly, if y precedes x on L_1, y is adjacent to an x_0 which precedes x, and x is adjacent to a y_1 which precedes y; the strong ordering property implies that x is adjacent to y. Therefore, adjacencies between the color classes are represented correctly in the diagram.

It remains to be shown that there are no crossings between vertices from the same color class in the diagram. The placements of X and Y vertices is symmetric, so we show only that no vertices x_a and x_b can cross in the diagram. Suppose x_a and x_b cross; then the last neighbor of x_a in the ordering is y_d, while the last neighbor of x_b is y_c, where y_c precedes y_d in the ordering. The strong ordering property implies that x_b is adjacent to y_d, contradicting the assumption that y_c is the last neighbor of x_b in the ordering. □

An alternative characterization of bipartite permutation graphs by ordering properties of a single color class, also from [445], is the subject of exercise 9.12.

The first linear time recognition algorithm for bipartite permutation graphs is in [445]. An extremely simple algorithm has been discovered independently several times, as I know from reviewing papers, and appeared first in [448] which also has a correctness proof.

Let y be any vertex of a bipartite graph G. Let z be a vertex of maximum distance from y with minimum possible degree. Divide vertices into levels, based upon their distance from z. Create an ordering of the vertices, such that vertices of level i precede vertices of level i+1. Vertices at level i are ordered as follows; for each x, calculate the value $\mathrm{v}(x)$ = the number of neighbors of x at level i+1 - the number of neighbors of x at level i-1. Order vertices in increasing order of $\mathrm{v}(x)$. G is a bipartite permutation graph if and only if the ordering restricted to each color class is a strong ordering of G.

If we use the graph of figure 9.6, and let y be vertex 7, vertex 2 has the maximum distance from 7 and is chosen as z. Breadth first search from 2 produces the levels (2) (1,5) (4,8) (7) (6) (3). Since 5 is adjacent to 1 vertex at the next level and 1 at the previous level, it is assigned a value 0 and and is placed before 1 which is adjacent to 2 vertices at the next level and 1 at the previous level (giving value 1). Vertex 8, which is adjacent to 1 vertex at the next level and 2 at the previous level (value -1) is placed before 4, which is adjacent to 1 vertex at the next level and 1 at the previous (value 0). The strong ordering derived from this is exactly the one shown in the right hand side of figure 9.6.

It is not hard to see that this can be implemented as a linear time algorithm; details are left for exercise 9.13.

9.7.2 Biconvex and Convex Graphs. An ordering of color class X in a bipartite graph is called convex if for every vertex y, $\mathrm{N}(y)$ occurs consecutively in X. A bipartite graph is a convex graph if one color class has such an ordering, and is called biconvex if both color classes have a convex ordering. Biconvex graphs have also been called doubly convex graphs.

Every strong ordering of a graph with no isolated vertex is doubly convex (exercise 9.14), so bipartite permutation graphs are biconvex. It is not hard to see that a chordless cycle of length at least six has no convex ordering (exercise 9.15), so that both biconvex and convex graphs are subsets of chordal bipartite graphs.

It is slightly more difficult to see that the containments of graph classes are proper, i.e. there are graphs which are biconvex but not bipartite permutation graphs, convex but not biconvex, and chordal bipartite but not convex. Exercise 9.16 gives examples of graphs which show that the containments are proper, and asks the reader to provide proofs that the graphs are in the larger class, but not the smaller.

Convex graphs obviously have an implicit representation, since we can store the number of a vertex from color class X in the convex ordering, and store the first and last neighbors of each vertex from Y in the ordering.

Convex graphs were first introduced in [221], motivated by a manufacturing problem in which left rods could be attached to right rods if the sum of the weights of the two rods was within a fixed range, and the difference in weights of the two sides was within a different fixed range. The class has also been mentioned as applying to certain classes of transportation problems [155].

Recognition of convex graphs is closely tied to recognition of interval graphs. In chapter 13, we discuss PQ-trees; these are a data structure designed to implement restrictions requiring that a set of objects must occur consecutively. PQ-trees can enforce a set of restrictions in linear time with respect to the size of the total restriction set; i.e., a restriction saying elements $x_1, ..., x_i$ must occur consecutively is counted as i units of input. This gives immediately a linear time algorithm for recognizing convex and biconvex graphs.

Tucker [467] shows that color class X has a convex ordering if and only if X contains no asteroidal triple. Note that this does not imply equivalence to bipartite asteroidal triple-free graphs (which are equal to bipartite permutation graphs); there are bipartite graphs which have asteroidal triples, such that all asteroidal triples contain vertices from both color classes (exercise 9.19). Tucker also gives a forbidden subgraph characterization for having a convex ordering for a color class.

Biconvex graphs have some nonobvious properties which have been used in optimization algorithms.

An S-ordering is a generalization of a strong ordering in which for any pair of crossing edges (x_a, y_d), (x_b, y_c) with x_a before x_b, y_c before y_d, at least one of (x_a, y_c) or (x_b, y_d) is an edge of G.

Abbas and Stewart [1] show that G is biconvex if and only if it has has an ordering of the color classes which is both convex for each class, and is an S-ordering. This result was also known to Yu and Chen (it appeared in an early version of one of their papers, but not in the final draft), who emphasize a somewhat different model discussed in the next paragraph. An S-ordering also has a matrix interpretation; if the orderings correspond to rows and columns of a bipartite adjacency matrix, an S-ordering corresponds to a matrix which has no induced 2 by 2 identity submatrix. Matrix properties of this type are studied in chapter 10.

Consider the graph in figure 9.7-a. Place a convex ordering of X along the X axis, and a convex ordering of Y along the Y axis. We can view each adjacency as a grid point; for example, if vertex 3 of X is adjacent to vertex 6 of Y, this intersection corresponds to grid point 3,6. The fact that both orderings are convex means that vertices can be mapped to line segments on the grid, with x and y adjacent if and only if the corresponding segments intersect. I have described a grid intersection model; however, this can also be modified easily to give other models as described in chapter 3. Thus, the model can be interpreted (see exercise 9.17) to give a circle

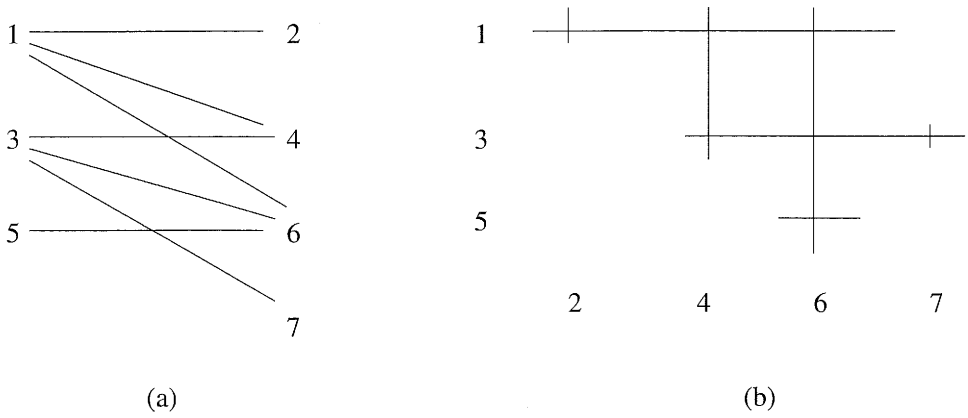


Figure 9.7 A Model of a Biconvex Graph

graph model, a trapezoid graph model , or a 4-polygon model (i.e. intersection graph of segments which have endpoints on different sides of a unit square).

9.8 Perfect Elimination Bipartite Graphs

This section deals with a well known class of graphs which contain the chordal bipartite graphs.

A bisimplicial edge $e = (x, y)$ is an edge such that $N(x) \cup N(y)$ induces a complete bipartite graph. G is a perfect elimination bipartite graph if all edges can be eliminated by successively removing both endpoints of a bisimplicial edge.

Perfect elimination bipartite graphs correspond to graphs in which Gaussian elimination can be performed on nonsymmetric matrices without causing any fill. Golumbic [223] covers this application, and the main theorems on perfect elimination bipartite graphs. For example, it is shown that G is chordal bipartite if and only if every induced subgraph of G is perfect elimination bipartite.

Recognition algorithms for perfect elimination bipartite graphs work by searching greedily for a bisimplicial edge to eliminate. If this is done in a completely naive way, this strategy will be quite expensive. It is not hard to design an $O(n^3)$ recognition algorithm for the class, as first shown in [222]. For each ordered pair of vertices x, y, maintain a variable deficit(x, y), corresponding to the number of vertices which are neighbors of y and nonneighbors of x. It is easy to create the initial set of deficit values in $O(n^3)$ time. Whenever a vertex v is removed, we reduce deficit(x, y) by 1 for all $y \in N(v), x \in V - N(v)$. Clearly, we can update all deficits changed by removing v in $O(n^2)$ time.

We can characterize bisimplicial edges in terms of deficit values. The edge (u, v) is bisimplicial if for every neighbor w of u, deficit$(w, v) = 0$, and for every neighbor t of v, deficit$(t, u) = 0$. For each edge (u, v) we keep the number of neighbors of u and v which violate this condition. When deficit(i, j) is reduced to 0, we decrement the number on any edge (h, j) such that h is a neighbor of i. The total time maintaining these numbers on edges is clearly $O(n^3)$, and we can find the next edge to eliminate in $O(m)$ time given these numbers. Therefore, the class can be recognized in $O(n^3)$ time; this is reduced by a logarithmic factor in [444].

Perfect elimination bipartite graphs are not hereditary. In fact, every bipartite graph is an induced subgraph of a perfect elimination bipartite graph on $2n$ vertices. This implies that there are $2^{\Theta(n^2)}$ perfect elimination bipartite graphs (exercise 9.6), so representation questions are not interesting for the class.

9.9 Bipartite Graphs with Forbidden Induced Subgraphs

As we saw in chapter 8, for any forbidden subgraph F which was not a subset of P_4, the number of F-free graphs is $2^{\Theta(n^2)}$. Although the number of bipartite graphs is also $2^{\Theta(n^2)}$, classes defined by forbidden subgraphs work much differently; forbidding any induced subgraph defines a class with $2^{o(n^2)}$ members.

This can be seen as a consequence of theorem 10.3, which shows that the number of n by n 0/1 matrices which do not contain any fixed induced submatrix is $2^{o(n^2)}$. Since any bipartite graph can be represented by a bipartite adjacency matrix, and forbidding a subgraph corresponds to forbidding more than one induced submatrix (we can rearrange the rows and columns using any permutation, and all the submatrices are forbidden), the bound follows directly.

Brightwell, Grable, and Prömel [80] make a further distinction; they divide the classes defined by forbidden subgraphs based on whether there are $2^{O(nlogn)}$ graphs in the class. There are some differences in their definitions from ours, since they are counting partial orders rather than bipartite graphs. Differences come from the fact that their forbidden subgraphs have vertices labeled as sources and sinks, and from the fact that they allow partial orders of height greater than one in the families of partial orders being counted.

Nevertheless, most results from [80] translate directly to the case of bipartite forbidden subgraphs. They give two maximal forbidden orders F_1, F_2 such that it is known that there are $2^{O(nlogn)}$ F-free partial orders. One class is defined by $F_1 = K_{1,r}$ plus s independent vertices. The second class is defined by $F_2 = K_{1,r} + K_2$. It is easy to see that the F_1-free and F_2-free bipartite graphs have implicit representations (exercise 9.21), so there are $2^{O(nlogn)}$ graphs in the class.

They also give eight minimal forbidden height 1 partial orders F such that the number of F-free posets is known to be larger than $2^{O(nlogn)}$. Seven of these forbidden partial order arguments translate directly to bipartite graphs; the forbidden subgraphs are C_4, C_6, C_8, $3K_2$, $2K_2 + 2K_1$, $P_4 + 2K_1$, and P_4 plus two degree 1 vertices adjacent to the central vertices of the P_4.

Bipartite graphs without C_4 correspond to matrices discussed in section 11.2. It is possible to build a bipartite graph with $\Theta(n^{1.5})$ edges and no C_4; since removing edges cannot create a C_4, the number of bipartite C_4-free subgraphs is not $2^{O(nlogn)}$. Similar known constructions give bipartite graphs with girth 8 and 12 (girth means that there are no cycles of shorter length) and sufficient number of edges ($\Omega(n^{4/3})$ and $\Omega(n^{6/5})$ respectively), so the number of bipartite graphs not containing C_k for any $k \leq 10$ is not $2^{O(nlogn)}$. $3K_2$ is the bipartite complement of C_6 (i.e. the graph by interchanging edges and nonedges between the color classes), which implies that the number of bipartite $3K_2$-free graphs is the same as the number of bipartite C_6-free graphs. For $2K_2 + 2K_1$ and $P_4 + 2K_1$, no matter how you place the independent vertices, you have an induced subgraph with no edges and 2 vertices from each color class; since this is the bipartite complement of C_4, the number of bipartite graphs without this subgraph is not $2^{O(nlogn)}$. The same argument

holds for P_4 plus degree 1 vertices adjacent to the two midpoints of the P_4, since it contains the bipartite complement of C_4.

The eighth minimal height 1 forbidden partial order from [80] which guarantees a large number of partial orders is 2 copies of P_3, with the center of one P_3 as a source, and the center of the other P_3 as a sink. When the assignment to color classes is specified, as assumed in the case of forbidden partial orders, this graph contains the bipartite complement of $2K_2$. However, when treated simply as a forbidden subgraph problem, we are also forbidding $2P_3$s in which the centers are both placed in color class X, and remaining vertices in Y. This is a subgraph of one of the cases left open in [80]; i.e. they were unable to show whether or not there were $2^{O(nlogn)}$ partial orders without this forbidden subposet.

The counting problem was left open in [80] for partial orders which are not contained in $K_{1,r} + sK_1$ or $K_{1,r} + K_2$, but do not contain any of the eight forbidden partial orders which are known to guarantee a large number of orders in the class. This gives four minimal and three maximal forbidden posets such that the number of partial orders defined is open, in the sense that it is not known whether the number is $2^{O(nlogn)}$. Of course, these are also open classes for the generalized implicit representation question.

We note that some of these questions which are open for classes of posets can be solved for classes of bipartite graphs. For example, one open question concerns counting the number of W-free posets. A W is a P_5 1,2,3,4,5 in which vertices 1,3, and 5 are designated as sources, while 2 and 4 are sinks. It can be shown (exercise 9.22) that the number of height 1 W-free posets is $2^{O(nlogn)}$; this does not follow immediately from the fact that the number of P_5-free bipartite graphs is $2^{O(nlogn)}$, since a P_5 with 1,3,5 as sinks and 2,4 as sources is W-free. However, there are also partial orders of height greater than 1 which are W-free, and it is not clear whether there are $2^{O(nlogn)}$ of these. It is worth thinking about which of the other open problems for forbidden posets can be solved for forbidden bipartite graphs; perhaps the most natural questions regard the number of bipartite graphs if we forbid P_6, and if we forbid P_7.

9.10 Exercises

1) In section 9.2, we gave an algorithm for reconstructing a Γ-free matrix from its reduced matrix. Show that every 1 added by this algorithm is necessary; i.e. every Γ-free matrix containing the initial set of 1s must contain every 1 added by the algorithm.

2) Prove that every matrix has a doubly lexical ordering.

3) Show that chordal bipartite graphs are exactly those graphs which are bipartite, and have an elimination scheme based on always removing vertices which are not in the middle of a P_5.

4) Let x be a row corresponding to the first vertex in a Γ-free ordering of the bipartite adjacency matrix.

4a) Show that the neighborhoods of vertices in $N(x)$ are ordered by set containment.

4b) A domination graph is a graph G such that for all induced subgraphs H of G, there are vertices x and y such that N(x) is a subset of N[y] in H. Show that the bipartite domination graphs are exactly equal to the chordal bipartite graphs.

5) Recall that a bisimplicial edge (x, y) in a bipartite graph is an edge such that every neighbor of x is adjacent to every neighbor of y. A bisimplicial edge elimination scheme successively removes bisimplicial edges (not the endpoints, as in the perfect elimination bipartite graphs discussed in the next question, but just the edge itself) until no edges remain. Show that G is chordal bipartite if and only if G has a bisimplicial edge elimination scheme.

6) Show that there are $2^{\Theta(n^2)}$ perfect elimination bipartite graphs.

7) Show that the dimension of a convex graph (considered as a height 1 poset) is at most 3.

8) Show that the rectangle number of the graphs constructed in theorem 9.1 is O(log$n$).

9) Complete the proof from the end of section 9.4 that every doubly lexical ordering of a chordal bipartite graph is Γ-free.

10a) Show that computing neighborhood containments in a chordal graph is as hard as for general graphs, in that an algorithm with running time O(n^k) for neighborhood containment in chordal graphs would yield an O(n^k) algorithm for computing neighborhood containment in general graphs.

10b) Show that we can compute neighborhood containments for all pairs of neighbors in a chordal bipartite graph in O(n^2) time.

11) If we subdivide every edge of a chordal graph, we must get a bipartite graph with no induced cycles of length > 6. Since there are 2^{cn^2} chordal graphs, why doesn't this show that there are 2^{cn^2} bipartite graphs with no induced cycles of length > 6?

12) An ordering of the vertices of color class X in a bipartite graph has the enclosure property if for every pair of vertices y_1, y_2 such that $N(y_1) \subset N(y_2)$ vertices of N(y_2) - N(y_1) occur consecutively in X. Show that G is a bipartite permutation graph if and only if X has an ordering which is both convex , and has the enclosure property.

13) Show that the recognition algorithm for bipartite permutation graphs described at the end of section 9.7.1 can be implemented to run in linear time.

14) Show that every bipartite permutation graph is biconvex.

15) Show that every convex graph is chordal bipartite.

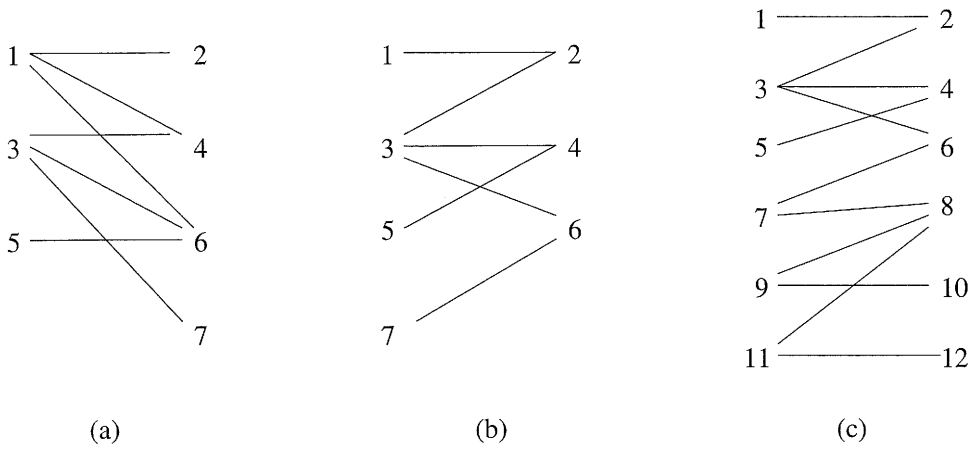


Figure 9.8 Example Graphs for Exercise 16

16a) Show that the graph of figure 9.8-a is biconvex, but is not a bipartite permutation graph.

16b) Show that the graph of figure 9.8-b is convex, but is not biconvex.

16c) Show that the graph of figure 9.8-c is chordal bipartite, but not convex.

17) Show that every biconvex graph is both a circle graph, and a trapezoid graph.

18) Show that every convex graph is a tolerance graph. Show that the graph of figure 9.8-c is a tolerance graph. This implies that bipartite tolerance graphs lie strictly between the convex graphs and chordal bipartite graphs; by contrast, bipartite bounded tolerance graphs are equal to bipartite permutation graphs.

19) Show that there are bipartite graphs which have asteroidal triples, but such that all asteroidal triples contain at least one vertex from both color classes.

20a) Give an $O(n+m)$ recognition algorithm for chain graphs, i.e. bipartite graphs with no induced $2K_2$.

20b) Describe an implicit representation for chain graphs.

21) Let F_1 be the graph $K_{1,r} + sK_1$, and let F_2 be the graph $K_{1,r} + K_2$. Show that both the F_1-free bipartite graphs and the F_2-free bipartite graphs have an implicit representation.

22) Show that the number of height 1 W-free posets is $2^{O(nlogn)}$.

23) Show that the number of W-free partial orders is $2^{O(nlog^2n)}$. Hint: look at properties of the comparability graphs of W-free partial orders.

CHAPTER 10

Matrices

This chapter is the result of a different way of generalizing questions on chordal bipartite graphs, by generalizing the matrix characterization rather than generalizing the graph theoretic definition. The concept of implicit representation is extended from graphs to matrices, and we raise open questions regarding counting and representing matrices with various forbidden submatrices. It is possible to phrase these questions in terms of bipartite graphs with vertex orderings, but to do so would be artificial, while the questions on matrices seem quite natural. We restrict our attention for the purposes of this book to 0/1 matrices, which are the most closely related to graphs. Matrices will be assumed to be n by n unless stated otherwise. Many of the classes of matrices discussed in this chapter are taken from [299].

Defining an implicit, generalized implicit, or space optimal representation of a class of matrices is analogous to the graph theoretic definition. A class of matrices has an implicit representation if we can store O(log$n$) bits of information with each row and column, such that the entry in row x, column y can be deduced entirely from the bits stored at x and y. A class of matrices with $2^{f(n)}$ members has a generalized implicit representation if we can store O(f(n)/n) bits of information at each row and column so that the entry at row x, column y can be deduced from the bits stored at x and y, while a space optimal representation uses O(f(n)) bits.

A forbidden submatrix F for a class of matrices is a matrix which is not an induced submatrix (i.e. the result of deleting rows and/or columns) of any matrix in the class. A matrix M is in the class PM(F) if we can permute the rows and columns of M so that the result does not contain F as an induced submatrix.

Thus, PM(Γ) corresponds to bipartite adjacency matrices of chordal bipartite graphs. This class has $2^{\Theta(nlog^2n)}$ members, and we know of a way to construct space efficient representations efficiently, while no implicit representation is known for the class. Similar questions can be asked about other classes defined by forbidden submatrices; we will discuss some of the most obvious specific matrices to consider, and describe what is known about counting the size of the classes in the more general case. In contrast with forbidding induced subgraphs, which is discussed in chapter 8, we will find that forbidding any single induced submatrix reduces the number of matrices in the class to $2^{o(n^2)}$, and thus each forbidden matrix has the potential to have its own interesting form of generalized implicit representation.

10.1 Small Forbidden Classes of Matrices

For a more complete description of what is known about recognizing classes defined by forbidding small induced submatrices, the reader is invited to see [299].

Of course, forbidding 1 by 1 matrices is uninteresting, leaving only 1 matrix (all 0s or all 1s) for each value of n. Therefore, the first forbidden matrices we consider

are the 2 by 1 forbidden submatrices; because of the obvious symmetries we only need to consider the classes of matrices PM(1 1) and PM(1 0). PM(1 1) is a trivial class of matrices, exactly equal to the class of matrices which have at most one 1 entry in each row.

PM(1 0) is not a particularly interesting class of matrices. Clearly the ordering of rows does not matter, and for any row we must have all 0s before all 1s. There is clearly an implicit representation of such matrices; store with each row the number of 0s in the row, and store with each column the position of the column in the permuted ordering. Recognizing matrices in the class is easy, and will be left for the exercises.

We then come to two by two forbidden submatrices. We have already dealt with one such class, since chordal bipartite graphs correspond to forbidding Γ in the permuted matrix. Symmetries would map any other forbidden two by two matrix with exactly one 0 to chordal bipartite graphs, and exactly three 0s in the matrix could be handled in representations by interchanging the roles of 0 and 1.

10.2 Linear Matrices

Forbidding all 1 and all 0 matrices are equivalent for our purposes. Assume that our forbidden submatrix is a 2 by 2 matrix with every entry equal to 1, which we will call $1_{2,2}$. These matrices are called linear matrices by Berge [35]. Since row and column permutations do not affect the forbidden matrix, we can interpret this completely as a graph theoretic problem on bipartite adjacency matrices; these matrices correspond to bipartite graphs without cycles of length four. For this class, we can count the number of matrices, and get a local representation which is 'close' to generalized implicit, in that the number of bits per vertex exceeds the number necessary for a generalized implicit representation by a logarithmic factor.

The key idea for counting this class of matrices comes from the theory of projective planes. It is possible to construct for certain n an n by n matrix M such that each row has $\Omega(n^{1/2})$ 1 entries, and no pair of rows has a 1 in the same pair of columns. M cannot contain $1_{2,2}$, and replacing any 1 entries by 0s will not create an induced $1_{2,2}$. Therefore, the number of matrices in the class is at least $2^{n^{1.5}}$, since we can include or not include each one value from M in the matrix.

We can also show that every linear matrix has $O(n^{1.5})$ 1 entries. The total number of 1s in rows with at most $2n^{1/2}$ 1 entries is $O(n^{1.5})$, so these can be ignored. We will show that there are fewer than $n^{1/2}$ rows with more than $2n^{1/2}$ 1 entries, which shows that the total number of 1 entries is $O(n^{1.5})$. Consider a set of $n^{1/2}$ rows, each of which has more than $2n^{1/2}$ 1 entries. For any pair of rows, there is at most 1 column which has a 1 in both rows, or the submatrix induced by two columns and this pair of rows is the forbidden submatrix. Thus, each row has a 1 in at least $n^{1/2}+1$ columns for which no other row in this set has a 1. There are only n columns altogether, so there cannot be $2n^{1/2}$ such rows. This implies that the number of 1 entries is $O(n^{1.5})$.

The fact that at most $n^{1/2}$ rows have more than $O(n^{1/2})$ 1 entries also allows us to store the matrix locally. Since row and column permutations do not affect this class, let us assume that the rows are ordered in decreasing order of number of 1 entries. For all but the first $n^{1/2}$ rows, store the positions of the row which contain a 1. For each column, store the first $n^{1/2}$ bits of the matrix. This uses $O(n^{1/2}\log n)$ bits per row, and $O(n^{1/2})$ bits per column.

This leaves the question open as to whether there are $2^{n^{1.5}logn}$ matrices of this form, or whether a representation using $O(n^{1/2})$ bits per row and column is possible. Kleitman and Winston [298] are able to show that there are $2^{O(n^{3/2})}$ such matrices. The problem of finding a generalized implicit representation of linear matrices is therefore open, though the gap between the size of representation which can be found and the best possible representation size is relatively small compared to some other classes defined by forbidden submatrices.

We now consider the recognition problem for linear matrices. Let m be the number of nonzero entries in the matrix; we may assume that m is $O(n^{1.5})$. Since the forbidden matrix is impervious to row and column transformations, we get an immediate $O(n^4)$ algorithm by checking each pair of rows and each pair of columns. This can easily be improved using a number of different strategies. It is easy to determine whether a pair of rows contains a forbidden induced submatrix in $O(n)$ time, reducing the time complexity to $O(n^3)$. This can be reduced to $O(nm)$ by noting that we can test whether there is any forbidden submatrix including row r in $O(m)$ time (exercise 10.6). Matrix multiplication can be used to get an algorithm with time complexity $O(n^{2.376})$ (exercise 10.7). We can bring this down to $O(n^2)$ by keeping an n by n array which has a 1 at position i,j if and only if we have seen a 1 in the same column for rows i and j. When a new column is considered, we mark every pair of rows which have a 1 in this column; if a pair is marked a second time, we have found the forbidden matrix. Every unit of work can be charged to a 0 entry of the array, since any attempt to mark a 1 entry of the array causes us to answer that the forbidden matrix is contained in M. We will refine this method below to give an $O(mn^{1/2})$ algorithm for recognizing this class of matrices.

We will show how to recognize these matrices in $O(mn^{1/2})$ time. We know that there are at most $n^{1/2}$ rows which contain more than $2n^{1/2}$ 1 entries. For each such row r, we can use $O(m)$ time to check whether there is a violation involving r and some other row. After checking for forbidding submatrices in these rows, we may assume that every remaining row has at most $2n^{1/2}$ 1 entries.

We can now use the array based method on the remaining rows to solve the recognition problem, but we get a better time bound. When a new row is considered, we do work proportional to the square of the number of 1 entries in the row, and never look at entries in the row again. Since each remaining row has $O(n^{1/2})$ 1 entries, we mark $O(n^{1/2})$ array positions for each 1 entry in the matrix; thus the total work is $O(mn^{1/2})$. This method uses $\Theta(n^2)$ space; exercise 10.8 deals with solving the same problem in $O(mn^{1/2})$ space.

We note that the problem of recognizing these matrices is also equivalent in time complexity to the problem of determining whether an arbitrary graph contains a C_4 as a subgraph, where we do not require the subgraph to be an induced subgraph. Clearly, an algorithm for finding C_4s in an arbitrary graph can be used to find C_4s in a bipartite graph, which is the same as finding the forbidden submatrix if M is interpreted as a bipartite adjacency matrix. For any graph G, we can create a bipartite graph $B(G)$ by creating two vertices v_1, v_2 for each vertex v in G, and adding an edge (v_1,w_2) to $B(G)$ if and only if v and w are adjacent in G. Exercise 10.13 shows that G contains a C_4 if and only if $B(G)$ contains a C_4, so the problem of finding C_4s in an arbitrary graph can be transformed to a bipartite problem using linear time and space. An algorithm which takes $O(m\ a(G))$ for finding C_4s, where $a(G)$ is the arboricity of G (as defined in the exercises), is presented in [101].

The arboricity of a C_4-free (in the non-induced sense) graph cannot be more than $n^{1/2}$ (exercise 10.14), so the algorithm of [101] is always at least as good as the algorithm presented above.

10.3 Forbidden 2 by 2 Identity Matrices

One class of forbidden 2 by 2 matrices is particularly interesting, and not very well understood. This is the class of 0/1 matrices such that rows and columns can be permuted so that there is no induced 2 by 2 identity matrix, which we will call I_2. There are a number of interesting questions with respect to the class.

PM(I_2) has been studied under the name of connected matrices. This terminology comes from the following definition of matrix connectivity [33]. For any pair of rows i, j, consider the sequence which comes from subtracting the entry at row j from the entry at row i, as we read the rows from left to right; the number of times this sequence changes signs (we ignore 0s in the sequence) is called signchange(i, j). A matrix is k-connected if the maximum value of signchange(i, j) is at most k; a 1-connected matrix is called a connected matrix.

It is not hard to show, as given in [299, 33] that forbidding I_2 is the same as forbidding the pair of matrices

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

Clearly, if I_2 is forbidden then neither of these matrices can occur, so every matrix in PM(I_2) is connected. On the other hand, I_2 is a valid ordering for this class, so PM(I_2) might be a proper subset of connected matrices.

Suppose that we can permute the matrix so that neither of the above submatrices are induced submatrices; let M be the permuted matrix. Changing the order of rows, while leaving columns in the same order, cannot create either of the forbidden submatrices. We will choose an ordering of the rows so that the result matrix is I_2-free.

Consider each row as a binary number, read from right to left. Order the rows in increasing numerical order. Suppose that the result matrix contains an I_2 involving rows a and b, and columns y and z. There must be a column after z which has a 1 in row a and a 0 in row b, or row a would be larger than row b. This creates the forbidden matrix

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

This implies that M contains one of the forbidden submatrices. Thus, if M is connected, we can permute the rows of M to be I_2-free, and the connected matrices are exactly equal to PM(I_2).

One of the major advantages of the characterization in terms of forbidding the two matrices is that row permutation does not affect these forbidden submatrices; this makes it easier to rule out possible arrangements, as is required in exercise 10.3.

The major papers dealing with this class of matrices appear only in Russian. The paper [33] gives a recognition algorithm for the class which works in $O(i^2j^2)$ time, where i and j are the number of rows and columns in M. Although I was able to get this paper translated, there are some crucial steps of the translation which I fail to understand. Thus, I am unable to present the algorithm in this section, though it is very clearly of interest. It would be a service for some reader who is

capable of working in both English and Russian to present an English version of this paper.

Thus, the counting, representation, and (for non-Russian speakers) recognition problems remain open on this class of matrices.

Open Problem 10.1: What is the number of I_2-free n by n matrices?

10.4 Forbidding $\binom{10}{10}$

There is one class of forbidden 2 by 2 matrices which we have not covered, defined by forbidding a matrix with 1s in column 1, and 0s in column 2; we will call this matrix F in this section. Very little work has been done on this class of matrices.

[299] points out that matrices in this class can be recognized in polynomial time. The matrices are affected by column transformations, but not row transformations. We make a graph G with columns as vertices, which reflects required orderings of the columns. For each ordered pair of columns c_1,c_2, we can check whether these contain the forbidden matrix if c_1 is placed before c_2; if they do, add an edge from c_2 to c_1 in G. M can be permuted to avoid F if and only if G is acyclic, and any topological sort of an acyclic G gives a valid ordering of the columns. A straightforward implementation of this algorithm takes $\Theta(n^3)$ time.

We will present a linear time algorithm for recognizing this class of matrices. First, let us note that it is trivial to find a correct ordering of the columns, if such an ordering exists. If there is a valid column ordering, then ordering columns in increasing order of the number of 1s in the column is a valid ordering. Exercise 10.10 asks the reader to show that if the 1s in column c_1 are a proper subset of the 1s in column c_2, any ordering with c_1 after c_2 is still valid if c_1 and c_2 are swapped. The following theorem strengthens this, giving a trivial ordering algorithm for this class of matrices. An ordering of the columns is said to be 1-increasing if whenever $i < j$, the number of 1 entries in column i is no larger than the number of 1 entries in column j.

Theorem 10.1 *Let M be a matrix in PM(F). Every 1-increasing ordering of the columns is F-free.*

Proof By exercise 10.10, we may assume that there is an F-free ordering which respects neighborhood containment, i.e. if the 1s in column i are a proper subset of the 1s in column j, then i precedes j. We first show that every F-free ordering which respects neighborhood containment is 1-increasing.

Consider any columns c_1, c_2 such that the number of 1s in c_1 is smaller than the number of 1s in c_2. If there is a row r such that r has a 1 in column c_1 and a 0 in column c_2, then there must be two rows which have a 1 in column c_2 and a 0 in column c_1, and c_1 must precede c_2 in any ordering.

Thus, we may assume that there is a 1-increasing F-free ordering A. If two columns have the same number of 1 entries, they either differ on at most two rows, in which case any ordering of the columns is valid, or they differ on at least three rows, in which case no ordering is valid. Therefore, order of columns which have the same number of 1 values does not affect whether the matrix is F-free. We can obtain any 1-increasing order from A by permuting columns with the same number of 1s, so every 1-increasing ordering of the columns is F-free. □

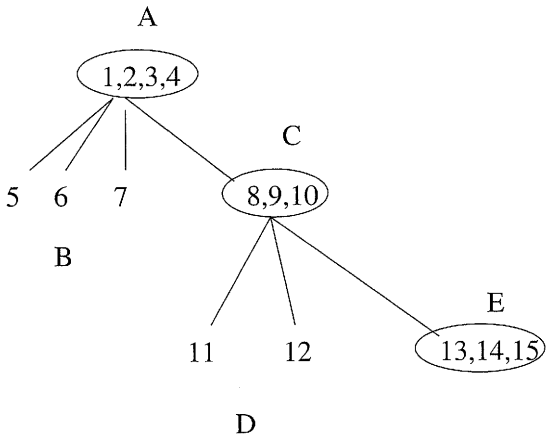


Figure 10.1 Set Labels for Tree

We describe an algorithm to test whether a matrix is F-free below. Using theorem 10.1, this also allows us to recognize whether a matrix is in PM(F).

Our recognition algorithm reads one column at a time, and maintains a very restricted form of tree. Each node of the tree is a set of rows, and each row which has at least one 1 entry in the set of columns read up to this point appears once in the tree. The tree has height at most 2; at each level, at most one node is not a leaf corresponding to a single row of the matrix. The tree is made to have the property that two rows have a 1 value in a common column if and only if they are on a single root to leaf path in the tree. When a new column c is read, the ordering of columns seen up to this point is still valid if c has at most one 0 entry on every path from the root of the tree to a leaf. We first show how to check this condition on a tree, and then show how to update the tree based on the values for column c.

For convenience in the description of the algorithm, let us give names to some of the node types. Call the set of rows contained in the root A, and the set of leaf children of A which are single rows of the matrix B. The single nonleaf consisting of more than one row of M child of A will be called C; the one row children of C are called D, and lone child of C with more than one row is called E. A picture of this general tree is given in figure 10.1; we note that some of these sets may be empty.

Given the restricted form of the tree, it is easy to check whether a new column has at most one 0 entry on every root to leaf path in the tree. If A contains a row with a 0 in column c, verify that all other rows in the tree have a 1 in column c. If A contains no such row, check whether there is a row in C which has a 0 in column c. If there is a 0 row in C, check that all other rows in C, D, and E have a 1 in column c. If there is no 0 row in A or C, verify that E has at most 1 row with a 0 in column c. These steps clearly take time proportional to the number of 1 entries in c.

The key step is modifying the tree to reflect changes made by the addition of a column. Call F the set of rows which have a 1 in column c, and no 1 in any columns seen previously.

If c has a 0 in row a of A, create a height 1 tree with root containing all rows which have had 1 entries except a and F, and make the two children of the root a and F. Otherwise, let B_1 be the rows of B which have a 0 in column c, and let B_2 be the rows of B which have a 1 in column c. If c has a 0 in column x of C, leave A as the root, leave each row of B_1 as a child of A, make the rows of B_2,C-x,D,E a child C' of A, and make x and F the two children of C'. If c has no 0s in A or C, let D_1 (D_2) be the rows of d which have a 0 (1) in column c, and let e be the row of E which has a 0 in column c. Leave A as the root, and make each row of B_1 a child of A, and leave C as the other child of A. Make each row of D_1 and e children of C, and make E-e,D_2,B_2,F a single child of C.

All tree operations take time proportional to the number of 1 entries in c, so matrices in PM(F) can be recognized in linear time.

The algorithm above also can be used to solve the counting problem for this class of matrices, though it does not solve the representation problem. We want to describe the sequence of tree operations using O($n$log$n$) bits. At first, this may seem difficult to do, since it is not possible to describe the changes in the tree caused by a single column c in O(log$n$) bits. First, the set of new rows F for a single column can be arbitrarily large. This is not a major problem, since each row is new only once; thus, the total number of bits needed to describe additions of new rows is O($n$log$n$).

Our other apparent difficulty is to deal with changes to B and D, since any subset of rows from B and D may be removed from its location, and placed in C or E. However, at most 2 rows are added to B and to D in a single step; this implies that the total number of removals from B and D is O(n).

This allows us to construct G from the following information. For each column c, store all row numbers which have their first 1 in column c, and any row from A, C, or E which has a 0 in c. If there are no 0s in any row in A, store all rows of B which have a 1 in column c. If there are no 0s in any row of A or C, store all rows of D which have a 1 in column c. Each successive tree can be built from the previous tree and the information stored with c, so all trees can be constructed, and we can reconstruct column c from the information stored at c and the previous tree. Thus, M can be reconstructed from the information stored at each column.

Each row is stored as a 'new' row for one column, so the total space used storing new rows is O($n$log$n$). At most 1 row value is stored with c from each of A, C, and E, so the bits used for storing rows from these sets is O($n$log$n$). Each row of B or D stored with c is removed from B or D during this step; as argued above, this shows that the total number of bits stored for rows of B and D is O($n$log$n$).

Thus, we have shown that there are at most $2^{O(n\log n)}$ matrices which do not contain this forbidden matrix. The question of finding an implicit representation is left open at this time. I believe that an implicit representation should not be too difficult to find for this class; perhaps finding one could help us understand how to produce good representations of classes of matrices defined by other forbidden submatrices. A simpler proof of the upper bound on the number of matrices which does not immediately give a recognition algorithm is given in exercise 10.11; we present the version in this chapter since it goes together with the recognition algorithm.

10.5 Other Classes of Interest

We mention in this section a number of other classes of matrices which have appeared in the literature. The counting and representation problem does not seem to have been studied for many of these classes.

We noted in section 10.3 that forbidding I_2 is equivalent to forbidding the two matrices

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}.$$

The latter characterization is part of a more general definition; a matrix has been called k-connected if it avoids the pair of 2 by $k+2$ matrices

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 1 & .. \\ 0 & 1 & 0 & 1 & 0 & .. \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & .. \\ 1 & 0 & 1 & 0 & 1 & .. \end{pmatrix}.$$

There are polynomial time algorithms for solving a number of NP- complete problems on 1 and 2-connected matrices; these require that the input is already permuted to give a valid ordering.

Recognition of k-connected matrices is open for all $k > 1$, as are the counting and representation problems.

Monge matrices are matrices such that $M_{ij} + M_{pq} \leq M_{iq} + M_{pj}$ whenever $i \leq p, j \leq q$. There are many papers which use Monge matrices in combinatorial optimization; for a particularly alliterative example, see [397]. Binary (i.e. 0/1) Monge matrices have an obvious characterization in terms of forbidden 2 by 2 subgraphs which violate the property. Binary Monge matrices are completely characterized in [299], who show that removing identical rows and columns from a Monge matrix leads a matrix of size at most 3 by 4; from this, it is easy to see that the number of binary Monge matrices is $2^{O(nlogn)}$ (exercise 10.16).

A class of graphs called X-graphs is studied in [241]. These are graphs in which there is an ordering of the vertices which avoids the two matrices

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

Recognizing X-graphs in polynomial time remains open. The counting and representation for these has not been examined, though it is hard to imagine that these do not have $2^{O(nlogn)}$ members. The problems are also open and of potential interest for other combinations of small submatrices.

10.6 The Problems of Counting and Representation

As we saw in chapter 8, forbidding individual graphs, with a few exceptions, gives graph classes which are too large to have an efficient representation. The situation for forbidden submatrices is quite different; any forbidden submatrix of fixed size will reduce the number of matrices in the class sufficiently that we can hope for space efficient representations. Thus, every forbidden submatrix gives an open question for the generalized implicit graph question, while the same is not true for forbidden subgraphs.

Before developing a general theorem, let us examine the matrices defined by forbidding a single class. We will derive an upper bound on the number of n by n I_2-free matrices.

Consider a submatrix which contains more columns than rows, and such that every column is distinct. There must be some pair of columns c_1, c_2 such that the

one entries in each column are not a subset of the one entries in the other column. This means that there must be some pair of rows r_1, r_2 such that r_1 has values 1 and 0 on c_1 and c_2 respectively, while r_2 has the values 0 and 1 on c_1 and c_2.

We will use this fact to show that each set of $n^{1/4}$ rows in an I_2-free matrix can be stored using O($n$log$n$) bits, rather than the $n^{5/4}$ bits which are required in general. By storing each set of $n^{1/4}$ rows in this way, we use O($n^{7/4}$log$n$) bits to store an I_2-free matrix.

If there are fewer than $n^{3/4} + n^{1/4}$ distinct columns on this set of $n^{1/4}$ rows, we can store all rows in O($n$log$n$) bits. All distinct column values are stored, and given an index; this clearly uses O(n) bits. Each column can be represented by the corresponding index value for that column; this uses O(log$n$) bits for each column, and therefore O($n$log$n$) bits to store all columns on these rows.

If there are more then $n^{3/4}$ distinct column values, we will show that the matrix contains an induced I_2. Group the distinct columns into consecutive sets of $n^{1/4}+1$ columns. Within each set of columns, select some pair of rows r_i, r_j and columns c_k, c_l such that M$[i,k] = 1$, M$[i,l] = 0$, M$[j,k] = 0$, M$[j,l] = 1$.

Since there are $n^{1/2}$ sets of columns and only $n^{1/4}$ rows, some pair of column sets C_1, C_2 must select exactly the same pair of rows r_1, r_2. Assume that all columns in C_1 are to the left of all columns in C_2, and that r_1 is above r_2. From C_1, choose the column c_1 which has a 1 in r_1 and a 0 in r_2, and from C_2 choose column c_2 which has a 0 in r_1 and a 1 in r_2. These induce an I_2, which implies that there are at most $n^{3/4}$ distinct columns with respect to these rows; thus, all I_2-free matrices can be stored using O($n^{7/4}$log$n$) bits.

Although this argument may seem to be very dependent on the fact that we have a 2 by 2 identity matrix as our forbidden subgraph, it can be extended to any forbidden matrix. We will use the following theorem of Anstee and Füredi [19], which is proved by a more generalized form of the argument given above. A matrix is defined to be simple if it has no repeated columns.

Theorem 10.2 [19] *Let A be an m by n simple matrix, and F a k by l 0/1 matrix. Assume A does not have an induced submatrix F. Then m is $O(n^{2k-1})$.*

We note that the constants in the O term are stated more precisely in [19]. In particular, the number of columns l of F does not appear in the bound stated here, since it is a fixed multiple in the O term. We use the bound of theorem 10.2 to generalize our argument on the number of I_2-free matrices to a bound on the number of F-free matrices for an fixed forbidden submatrix F.

Theorem 10.3 *Let F be a k by l forbidden submatrix. The number of F-free n by n matrices is $2^{o(n^2)}$*

Proof Partition the columns of any F-free matrix M into sets of size f(n). For each set of f(n) columns, the number of distinct rows with respect to these columns is O($f(n)^{2k-1}$). The entries in each set of columns can be reconstructed if we write down each row value with respect to these columns, and for each row of the matrix we give the index of the appropriate row value on these columns. We use f(n) * O($f(n)^{2k-1}$) bits writing down the row values, and ($n$log$n$) bits giving the appropriate index for each row of M on these columns. Since there are O(n/f(n)) sets of columns, the total number of bits used to store the matrix is O((n/f(n))($f^{2k}(n)$+$n$log$n$)).

To minimize the number of bits used, we balance the terms $nf^{2k-1}(n)$ and $n^2 \log n / \mathrm{f}(n)$ by setting these to be equal, getting the equation $f^{2k}(n) = n \log n$, or $\mathrm{f}(n) = n^{1/2k} log n^{1/2k}$. The number of bits used to represent M in this way is $\mathrm{O}(n^{2-1/2k} log n)$. □

This argument is sufficient to show that forbidding any fixed induced submatrix F reduces the number of matrices to $2^{o(n^2)}$. Note that [19] and others were working on bounds for a much different problem, bounding the number of different rows with respect to the number of columns in a matrix without F as an induced submatrix. By chopping the matrix into groups of columns, we completely ignore copies of F which are not contained in a single block of the column partition, and thus are dramatically overcounting the number of possible matrices. If we want to find generalized implicit representations of matrices, we will need to be able to get more accurate estimations of the number of matrices which do not contain F. For example, this argument shows that the number of Γ-free matrices is at most $2^{n^{7/4} log^{3/4} n}$, while we know from chapter 9 that the actual number of Γ-free matrices is $2^{\Theta(n log^2 n)}$.

We note that it is conjectured [19] that the true bounds on the number of distinct columns for a k by l forbidden submatrix is $\mathrm{O}(c^k)$ rather than $\mathrm{O}(c^{2k-1})$, and this bound has been proven for many specific submatrices, including all 2 by 2 forbidden submatrices. However, we do not want to focus on the exact bounds known for the specific classes of matrices discussed above, since the bounds derived from this form of argument are almost certainly very far from optimal. The bounds on the number of distinct columns were not intended to be used to place bounds on the number of matrices, and I believe that better, as well as more intuitively appealing, bounds will come from trying to count the number of matrices in the classes directly. Readers interested in bounds on column size for other specific classes of matrices are referred to [19].

One interesting aspect of the bounds on the number of matrices in a class which are based on number of distinct rows is that these give natural representations using $\mathrm{o}(n^2)$ space which are fast for testing adjacency, but do not have any form of local structure. For each set of columns, we store all distinct rows in the matrix on that set of columns, and for each row we maintain the index of the corresponding distinct row on this column set. To test the entry at position (i,j) of M, we might find that i corresponds to row number 243 on this set of columns, and then look up the value of row 243 in column number j. However, this is not local; in order to make the space of the representation small, we cannot store all values of row 243 with column j.

Although the general desired result would be to take an arbitrary forbidden matrix F, and describe an implicit representation for matrices which do not contain F, we will state a number of relevant problems which are potentially easier.

Open Problem 10.2: If we know only that a forbidden submatrix F has i rows and j columns, improve the upper bound on the number of possible n by n matrices which do not contain F as an induced submatrix.

We are also interested in counting the number of matrices possible for specific values of F, such as I_2. Representation is open for these classes as well as counting; one representation question is posed below.

Open Problem 10.3: Find a way to take a forbidden matrix F, and design a storage scheme using $o(n)$ bits per row and column such that the entry at position i,j of any matrix without F as an induced submatrix can be determined only from the bits stored at i and j.

Of course, finding an implicit representation may be even harder. Both the counting and representation problem remain open for almost all matrices; exceptions are generally matrices with particularly simple or regular structure, and Γ.

Extending these bounds to higher dimensional matrices may be interesting, but will not be discussed here since it has relatively little connection to graph theory.

10.7 Other Matrix Properties

This section discusses classes of matrices which are not defined by specific forbidden submatrixes. A number of these classes are directly related to classes of intersection graphs.

A matrix has the consecutive 1s property for rows if the columns can be permuted so that the 1 values in each row occur consecutively. These matrices are closely related to interval graphs, since G is an interval graph if and only if the vertex/maximal clique incidence matrix has the consecutive 1s property. In fact, this is another way of describing Booth and Lueker's recognition algorithm for interval graphs, which uses PQ-trees to enforce maximal cliques containing a vertex to occur consecutively.

Recall that a bipartite graph is convex if the vertices of one color class can be ordered so that for each vertex x of the other color class, neighbors of x occur consecutively; a bipartite graph is biconvex if both color classes can be ordered in this fashion. Clearly G is biconvex if and only if the bipartite adjacency matrix has the consecutive 1s property for both rows and columns, while G is convex if the matrix has the consecutive 1s property for either rows or columns.

Matrices with the consecutive 1s property for rows have an implicit representation. Order columns by their position in their permuted matrix, and keep for each row r the value of the first and last columns which have a 1 in position r.

It seems natural to consider the class of matrices which can be permuted so that no pair of 1s occur consecutively in any column as well, though I have never seen any work on this class. Exercise 10.4 and 10.5 deal with this class of matrices.

10.8 Exercises

1) Design and analyze an efficient recognition algorithm for matrices in PM(1 0).

2) Show that not every I_2-free matrix can be permuted to avoid Γ.

3*) Show that not every Γ-free matrix can be permuted to avoid I_2.

4) Show that $2^{\Theta(n^2)}$ matrices have the nonconsecutive 1 property for rows.

5) Show that it is NP-complete to determine whether the columns of a matrix can be permuted so that the resulting matrix has the nonconsecutive 1s property for rows.

6) Design a simple $O(nm)$ algorithm for recognizing linear matrices by testing whether each 1 entry is part of a forbidden matrix in $O(m)$ time.

7) Design a simple algorithm which recognizes linear matrices in $O(n^2)$ time plus the time used for a single multiplication of two n by n matrices.

8) Design an algorithm which uses $O(mn^{1/2})$ time and space for recognizing linear matrices.

9) To recognize Γ-free matrices, we first found a doubly lexical ordering M' of M. From each 1 entry at position (i, j) of M', we then looked for the next 1 in row i and in column j, seeing whether the four rows and columns for these entries induced a Γ. Show that a similar strategy (seeing whether the four rows and columns induce $1_{2,2}$) does not work to recognize linear matrices.

10) If $N(c_1)$ is a subset of $N(c_2)$ and c_1 comes after c_2 in a $\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}$ free ordering, show that swapping c_1 and c_2 leaves the result $\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}$ free.

11) A simpler method for counting the number of $\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}$ free matrices works as follows. For each column, store the changes from column i to column $i+1$ (i.e. which 1 values become 0, and which 0 values become 1). Show that for any matrix in this class, the total number of changes is at most $2n$.

12*) Let M be a $1_{2,2}$-free matrix in which rows are ordered by decreasing number of 1 entries in a row. Show that it is possible to take rows sequentially, and verify in $O(n)$ time that each row does not induce a $1_{2,2}$ together with any previous rows of the matrix.

13) Show that an arbitrary graph G contains a (not necessarily induced) C_4 if and only if $B(G)$ (defined in section 10.2) contains a C_4.

14) The arboricity of a graph is the minimum number of forests necessary to cover all edges of G. Show that if a bipartite graph has no C_4, it has arboricity at most $n^{1/2}$.

15) Show that there are $2^{\Omega(nlogn)}$ I_2-free matrices, even if matrices which can be reached from each other using row and column permutations are counted as a single matrix.

16) Monge matrices are matrices such that $M_{ij} + M_{pq} \leq M_{iq} + M_{pj}$ whenever $i < p, j < q$. The paper [299] shows that if identical rows and columns are removed from a 0/1 Monge matrix until no identical rows or columns remain, the resulting matrices are of size at most 3 by 4.

17a) Find the four 3 by 4 0/1 Monge matrices with no identical rows or columns.

17b) Show that the results of [299] as stated in exercise 16 imply that the number of 0/1 Monge matrices is $2^{O(nlogn)}$.

CHAPTER 11

Decomposition

Another method of describing a graph class which gives a space efficient representation which is not implicit is to define the class in terms of a decomposition tree. This chapter explores a number of different forms of graph decomposition. We cover the substitution decomposition in detail here, since it is particularly important for algorithms which arise in later chapters. Other important decompositions covered in this chapter include join decomposition, clique separator decomposition, clique-width, and skew partition.

11.1 Substitution Decomposition and Vertex Partitioning

This section deals with an operation which has proved to be useful in a number of algorithms for graphs with a variety of intersection models. The operation, called vertex partitioning, is used in finding the transitive orientation of a graph, recognizing permutation graphs, and finding the substitution decomposition of a graph, recognizing circular-arc graphs and trapezoid graphs, and other problems. We will try to clarify these various algorithms by focusing on the unifying idea behind them. We will discuss two different implementations of the partitioning algorithm, which yield time bounds proportional to n^2 and $m\log n$.

Partitioning is a fundamental algorithmic technique; an early survey of the field by Tarjan [455] mentions partitioning along with such methods as greedy algorithms, divide and conquer, etc., as a design method for algorithms. For some reason, partitioning has become much less well known as a technique than the other methods; it is very useful for dealing with geometrically defined classes of intersection graphs.

I should mention that Ross McConnell and myself [367] have recently discovered an $O(n+m)$ algorithm for vertex partitioning, which speeds up all the algorithms described in this section. However, the new algorithm is currently too complex to describe easily, while I think that the reader can understand the versions described here. The first $O(n^2)$ partitioning algorithms [437], [377] were similarly complex; I hope and believe that in a number of years the linear algorithm can be simplified as well.

We first define the operation which we call vertex partitioning. The name partitioning is chosen because of similarities to the set partitioning problem described in [11]. This operation has been called 'refinement' in a number of papers by this author. In general, partitioning is a way of looking at operations on vertex sets, and this is often more efficient than thinking of operations on edges, as we will see for one important problem below.

Suppose that we are given a graph $G = (V,E)$ and an initial partition P of V into disjoint blocks B_1, ..., B_k. The vertex partitioning problem asks us to find the *coarsest* (having fewest blocks) partition of V, say $P' = C_1$, ..., C_j such that:

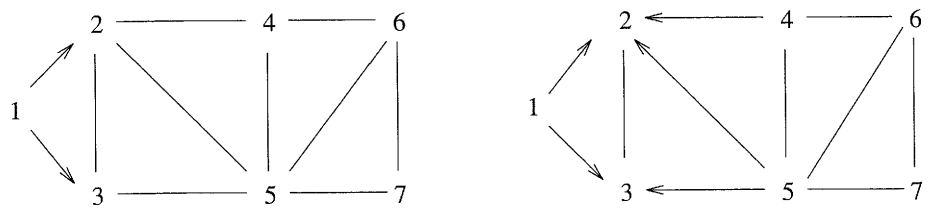


Figure 11.1 Sample Graph for Transitive Orientation via Partitioning

1) Each C_i is a subset of some B_h.
2) $x, y \in C_i$ implies N(x) - C_i = N(y) - C_i.

The second condition can be restated as follows. Each block C_i must be *indistinguishable* by any vertex x in V - C_i; i.e., x is either adjacent to every vertex in C_i, or x is adjacent to no vertex of C_i. For the reader familiar with the substitution decomposition, this corresponds to saying that each C_i is a module. A module is sometimes called a homogeneous set .

Before describing how to solve the vertex partitioning problem, I would like to describe a typical use. We will first look at a sample application, to the transitive orientation problem, and then examine the principle behind the application. Later, we will go into this specific application in greater detail.

11.1.1 Transitive Orientation: Overview. In the transitive orientation problem, we are given an undirected graph $G = (V,E)$, and we want to assign a direction to each edge so that if $a \rightarrow b$, $b \rightarrow c$, then in the new graph $a \rightarrow c$. The classic algorithm for this problem works by orienting an edge, and then looking at which orientations are 'forced' from this. For each edge, you look at all neighbors, which leads to an O(nm) algorithm.

Using vertex partitioning, we take a different approach to orientation. Suppose that you have somehow oriented all edges which go between two sets of vertices X, Y in a manner consistent with a transitive orientation of G. Edges which are entirely within X, or entirely within Y, remain unoriented. Let x be any vertex of X. Vertex x is adjacent to some set of vertices Y1 from Y, and is nonadjacent to some set of vertices Y2 from Y. We can now orient all edges (y_1,y_2) which go between Y1 and Y2. If $x \rightarrow y_1$, the edge must be directed from y_2 to y_1, while if $x \leftarrow y_1$, then we must have $y_1 \rightarrow y_2$ to preserve transitivity. This leads to having all orientations between Y1 and Y2. The algorithm can continue in this manner until each block of the partition is indistinguishable.

Consider the graph of figure 11.1. We start with an initial partition 1, 234567. Since 1 splits the block 234567 into 23, 4567, we can direct all edges between these subblocks as in the figure; note that many edges are assigned directions in a single step. In a later split, 2 could divide 4567 into 45, 67, causing edges to be directed from 4 to 6 and from 5 to 6 and 7.

In a general use of vertex partitioning, you begin with an 'approximation' of your final solution, which completely specifies the relationship between vertices from different blocks of the partition, while leaving unspecified the relationship between vertices in the same block. In the case of transitive orientation, this relationship is the orientation of edges. While doing the partitioning, you achieve a better

approximation of the solution by subdividing the blocks, maintaining the property that all relationships between vertices in different blocks are specified.

One difficult portion of the algorithms which use partitioning is to find the initial sets X and Y, such that all relationships between X and Y can be computed. This step is relatively problem specific, and is not a part of the vertex partitioning operation itself. Fortunately, for many problems, finding the sets X and Y is not the bottleneck step of the algorithm. For example, when we make the algorithm more precise in a later section, we will see that there is a simple linear time algorithm to find a source in a comparability graph.

Another question arises on what to do once the partitioning is finished. In general, some blocks can contain many vertices, and the relationships between vertices in the same block remains unspecified. Later in this section we will describe a form of graph decomposition called substitution decomposition. The graph is decomposed into pieces by this decomposition before the partitioning starts, and each piece has the property that partitioning stops with every block containing a single vertex. For problems such as transitive orientation, it is easy to find a transitive orientation of the entire graph if you are given transitive orientations of the individual pieces.

11.1.2 Partitioning Algorithms. We now describe two algorithms for performing vertex partitioning. Algorithm A solves the vertex partitioning problem in $O(n^2)$ time. We assume that G is stored as an adjacency matrix for algorithm A. A vertex v *splits* a block B when v is used to divide B into two subblocks, $B \cap N(v)$ and B - $N(v)$. For each vertex v, we keep a list ACTIVE[v] consisting of vertices which are in different blocks from v, but whose blocks have not yet been split by v. We choose any vertex w in ACTIVE[v], and v splits the block containing w. The relationship between any pair of vertices x, y is examined twice; once when the block containing y is split by x (and y is removed from ACTIVE[x]), and once when the block containing x is split by y (and x is removed from ACTIVE[y]).

Algorithm A:

```
Partition (B1,...,Bj);
  for each block Bi do
    for each vertex x in Bi do
      ACTIVE[x] := V - Bi;
  SPLITTER := V;
  while (SPLITTER not equal ∅) do begin
    v := any vertex in SPLITTER;
    C := the block containing the first member of ACTIVE[v];
    C1 := C - N(v);
    C2 := C - C1;
    ACTIVE[v] := ACTIVE[v] - C;
    if (ACTIVE[v] = ∅) then SPLITTER := SPLITTER - v;
    for each vertex x in C1 do ACTIVE[x] := ACTIVE[x] ∪ C2;
    for each vertex x in C2 do ACTIVE[x] := ACTIVE[x] ∪ C1;
    if C1 and C2 are nonempty then SPLITTER := SPLITTER ∪ C;
  end;
```

Algorithm B partitions a graph in O($m$log$n$) time. The algorithm chooses a vertex v which either i) has not been used previously as a splitting element, ii) is in a block B which is at most one half the size of the block which contained v the last time v was chosen as a splitting element, or iii) is in a block which will not be split further. Thus, a vertex v can be chosen as a splitting element at most log$n$ + 2 times. Vertex v is used to split every block which does not contain v.

The splitting caused by v can be performed in O(N(v)) time. Each block B is stored as a doubly linked list, and the number of vertices in the block is stored at a special node at the beginning of the block. Each vertex has a pointer to the front of its block. To split B, we first check that v is not in B; we then place neighbors of v in a new block B', which is kept linked to B until the end of the split caused by v.

If no vertex satisfies criteria i or ii for choosing a splitting element, any vertex from the largest active block C remaining will satisfy criterion iii. Every other vertex v has tried to split C, since v was separated from C at some point, and is now in a block which is at most the size of C.

Algorithm B:

```
Partition(B1,...,Bj);
  for each block Bi do lastused[Bi] := n
  READY := B1,...,Bj;
  while (G is nonempty) do begin
    while (READY not equal to ∅) do
      begin
        C := any block from READY;
        lastused[C] := |C|;
        for each vertex x in C do begin
          split all blocks except C on their adjacency to x;
          for each block B split into B,B2 by x do begin
            lastused[B2] := lastused[B];
            if |B| ≤ lastused[B]/2 then add B to READY;
            if |B2| ≤ lastused[B2]/2 then add B2 to READY;
          end
        end
      end;
    C := the largest remaining block;
    output C as part of the final partition;
    use each vertex from C to split other blocks as before;
    G := G - C;
  end;
```

11.1.3 Substitution Decomposition. As was mentioned earlier, vertex partitioning is closely related to the substitution decomposition. Most of the uses of vertex partitioning for developing best known algorithms start by constructing a tree which represents the substitution decomposition. This tree can be constructed in O(m+n) time [366, 136, 145]. We will describe an O(n^2) algorithm.

The substitution decomposition of a graph can be defined as follows. A module is a set M of vertices such that vertices of V - M are adjacent to either all vertices

of M, or no vertex of M. We decompose a graph recursively into subsets, each of which is a module of G. The procedure stops when all sets have a single vertex.

A module which induces a disconnected graph is called a parallel module. If the current set S is disconnected, partition S into connected components; make the decomposition tree for each connected component a child of a new internal node, which is labeled as a parallel module. If the complement of a module M is disconnected, M is called a series module. If the current set S is a series module, make the decomposition tree for each connected component of the complement of S a child of a new node, and label the new node as a series module.

Modules which are both connected and complement-connected are called prime modules. If S is a prime module, divide S into maximal proper submodules; exercise 11.5 shows that every vertex of a prime module M is in a unique maximal proper submodule of M, so this decomposition is unique. The decomposition tree for each maximal proper submodule is made a child of a new node, which is labeled as a prime module.

The quotient graph of a module M is the graph induced by taking one vertex from each child of M in the decomposition tree. It is easy to see that the choice of vertices from the children does not affect the structure of the quotient graph.

Let us first note that there is an obvious polynomial time algorithm for finding the substitution decomposition of a graph. Whenever a graph or its complement is disconnected, decompose the graph into connected components in the graph or its complement. Assume the graph is connected. For each pair of vertices x, y, it is not difficult to find in $O(m+n)$ time the smallest module which contains both x and y (exercise 11.1). Any time a module M is found, we can decompose $(G - M) \cup x$, and replace x in this structure by a recursive decomposition of M. A simple analysis shows that we take $O(n^2m)$ time on a single decomposition step, and thus $O(n^3m)$ time to decompose G. This can be reduced by the observation that if x and y are not contained in a nontrivial module, they cannot be contained in a nontrivial module in any subgraph. Thus, no pair x, y needs to be tested more than once, which reduces the time complexity to $O(n^2m)$ time.

The first $O(n^3)$ algorithms for computing the substitution decomposition are [83] and [244]. We will describe an $O(n^3)$ substitution decomposition algorithm, and modify this to get an $O(n^2)$ decomposition algorithm. The first $O(n^2)$ algorithm for this problem appears in [437]; we describe a simplification of the algorithm in [377].

Consider adding one vertex v at a time to the decomposition tree. If v is adjacent to 0 nodes in the tree, we add v either as a new child of the root if the root is a P node, or add a new root labeled P with v as one child and the rest of the tree as the other; the case where v is adjacent to all nodes of the tree is similar, with S replacing P in the description.

Otherwise, we start at the root. For each child c of the current node, we can decide whether v belongs in a submodule together with descendants of c in $O(n)$ time, by taking any descendant x of c, and testing whether v has exactly the same adjacencies to nondescendants of c as x. If we find that v belongs in a module with a single child c, we recursively place v in c. If we find that v belongs in a module with at least 2 children of c, this must be an S or P node, and v is added as a new child of the node.

Suppose that v does not belong in a module with any single child of the current node p. If p is a series or parallel node, we have a number of simple cases. We treat

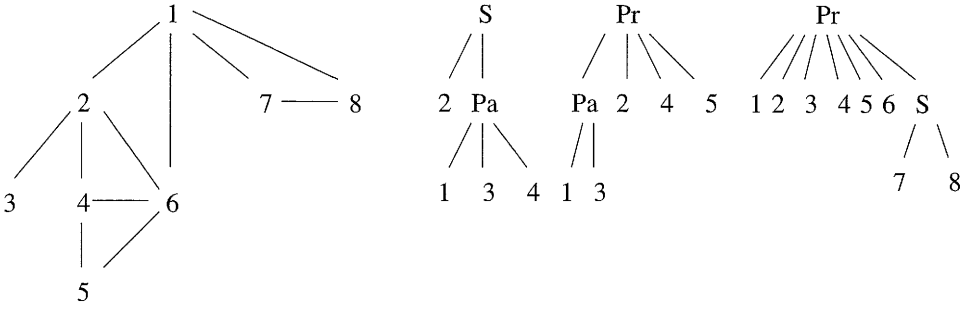


Figure 11.2 Sample Graph for Substitution Decomposition

only the case of a parallel node; the series node is handled symmetrically. Children of p which are completely nonadjacent to v are removed from p and grouped under a parallel node x_1. Replace p by x_1 in the tree, and make p a child of x_1. Children of p which are completely adjacent to v are grouped under a parallel node x_2, which becomes a child of p. Make v a child of p. If no child of p is split by v, label p as a series node. If some child of p is split by v, label p as a prime node, and perform the procedures for prime graphs as described starting with the next paragraph.

Suppose that p is a prime node, or that v splits some child of p and does not belong in a submodule with any child. We make p a prime node with v as one child, and split each child c of the node as follows. First, mark all nodes in the subtree of descendants of c by whether all leaf descendants are adjacent to v, all nonadjacent to v, or whether some are adjacent to v while others are nonadjacent; this marking clearly can be done in time proportional to the size of the subtree rooted at c.

Traverse the subtree rooted by each child c of p in postorder. Do nothing if we reach a node which is completely adjacent to v or completely nonadjacent to v. Suppose we reach an internal node i that contains both neighbors and nonneighbors of v. If i is a prime node, make all remaining children of i children of p. If i is a series or parallel node, split i into a set of children which are completely adjacent to v, and a set which are completely nonadjacent to v. Place each set under a new node marked with the same label as i, and make these new nodes children of p.

Finally, we traverse the tree in postorder to remove internal nodes of degree 1; any such node encountered is replaced by its child.

An example graph is given in figure 11.2. When vertex 5 is added and does not belong in a submodule with any child of the root, the root becomes prime since 5 splits the module 1,3,4; that set is split by a postorder traversal into a subset completely adjacent to 5, and another subset nonadjacent to 5. When 6 is added, it splits the module 1,3, and the tree reduces to a height 1 prime tree. Adding 7 simply makes 7 a child of the root; 8 belongs in a submodule with 7, creating the final tree shown in the figure.

Each vertex is added to the tree in $O(n^2)$ time, so the algorithm takes $O(n^3)$ time. After we decide where v is placed, we take $O(n)$ time to construct the tree, since this is a simple postorder traversal on a number of nonintersecting subtrees of a node. Thus, we focus on the placement phase of the algorithm. To make our task simpler, assume that we have preprocessed the tree so that every internal node has

a marker indicating whether descendants of the node are completely nonadjacent to v, completely adjacent to v, or are split by v.

Placing v in a parallel or series module is quite easy. We place v in a child of p if and only if v splits that child, and v is completely nonadjacent to all other children if p is a parallel module, or completely adjacent to all other children if p is a series module. Thus, the placement work takes time proportional to the number of children of p; the regrouping of children during the placement step is also clearly doable within this time bound.

However, placing children in a prime module is more expensive. If we compare v's adjacency list to the adjacency lists of all children of p, we take time proportional to the square of the number of children of p. For random graphs, the structure is very likely to be a single prime node with no submodules [372], so adding a vertex would take $\Theta(n^2)$ time. We need to keep information which distinguishes the adjacencies of different children of a prime node more efficiently. A complex data structure for keeping this information is described in [377]; the simplification here is due to McConnell.

For each pair of children c_1, c_2 of a prime node, we store a vertex which is not a descendant of c_1 or c_2, and which is adjacent to descendants of exactly one of c_1,c_2. Suppose that we have this information stored. We want to know which child v should be placed in, if any. Originally, we place all children in a set of candidates. While greater than two candidates remain, take any pair of candidates c_1, c_2, and let w be the vertex which is adjacent to descendants of exactly one of c_1,c_2. We remove whichever candidate disagrees with v on w from the candidate set. When we have one candidate c left, we check whether v agrees with c on all descendants of p which are not in c. If v disagrees with c on some such vertex, v is added as a new child of c, and v is added to c if the two agree.

The time taken to place c in p is proportional to the number of children of p, so c is placed in $O(n)$ time. Our only remaining task is to find vertices which distinguish pairs of children in a prime module. There is at most one prime module which has children added to it during the addition of v; this is the module with parent p in which v is placed as a child. We will show how to compute distinguishing vertices for children of p.

We start by choosing distinguishing vertices for v and each other child of p. When we removed each child c of p as a candidate module for containing v, we found a vertex w which could distinguish v from c; we store the value of w with c. Each node moved to be a child of p during the traversal of descendants of c is distinguished from v by w.

Now consider other pairs of nodes moved to be children of p by the addition of v to the graph. Note that if two vertices are in different submodules of a prime module, they will never be merged into a common child of p. Therefore, if a pair of vertices already has a distinguishing vertex, we do not need to update its value. Suppose that our traversal of descendants of c splits a parallel or series module M into a set S_1 of children which are completely adjacent to v, and a set of children S_2 which are completely nonadjacent to v; S_1 and S_2 are made children of p. We make v the distinguishing vertex for each pair $s_1 \in S_1$, $s_2 \in S_2$. If the parent of M is a series or parallel module, let w' be any descendant of the parent. We then climb the tree from M until we reach a prime module; let x be the next internal node encountered. If x has a different label from M, then all vertices in $S_1 \cup S_2$ are distinguished from descendants of x by any node from the other set (i.e. s_1 or

s_2), and if x has the same label as M, then these vertices are distinguished from vertices in M by w'.

The time taken to find distinguishing pairs of vertices is proportional to the number of vertex pairs distinguished. Each pair of vertices is distinguished at most once, so the entire cost of finding distinguishing pairs is $O(n^2)$.

We note that the algorithm has been phrased as if we need distinguishing pairs of vertices for internal nodes. It is trivial to maintain a single leaf descendant with each internal node as we run the algorithm; since each internal node is a module, two internal nodes are distinguished by the distinguishing vertex for their leaf descendants.

There are also relatively simple forms of substitution decomposition which take $O(m\log n)$ time based on modifying the $O(m\log n)$ partitioning algorithm described earlier; see [366] for a description.

11.1.4 Transitive Orientation. The following algorithm produces a transitive orientation of a graph, if such an orientation exists. We start by decomposing the graph with respect to the substitution decomposition.

Traverse the substitution decomposition tree in postorder; i.e. the children of x are visited before x is visited. When a node which represents a module M is visited, we orient all edges which have both endpoints in M.

If M is a parallel module, we do nothing; all edges in M have already been oriented. If M is a series module, we choose any linear order $c_1,...,c_k$ of the children of M, and orient an edge from a vertex in c_i to a vertex in c_j if and only if $i < j$. If M is a prime module, we choose one vertex v_i from each child c_i of M, and find a transitive orientation T of the induced subgraph; in other words, we find a transitive orientation of the quotient graph. We orient an edge from a vertex in child i to a vertex in child j if and only if $v_i \rightarrow v_j$ in T.

Since the least common ancestor of the two endpoints of every edge can be found in linear time [251], the problem can be reduced to the time to find the substitution decomposition tree, plus the time to orient the quotient graphs of prime modules. We note that finding the quotient graphs can be done easily in $O(n^2)$ or $O(m\log n)$ time, if simpler algorithms are preferred. To get an $O(n^2)$ algorithm, traverse the tree in postorder, maintaining a list of all descendants of a node with the node. When an internal node is visited, we check each pair of vertices from different children, and assign any edges found to this internal node. We leave the task of finding an $O(m\log n)$ algorithm for this problem to exercise 11.7.

As we saw earlier in this section, if we are given two sets X and Y with a valid transitive orientation on the edges which go between X and Y, we can use vertex partitioning algorithm and maintain the property that all edges between blocks are consistent with a transitive orientation of G. The result of partitioning any subgraph induced by taking one node from each prime module must have one vertex in each block, so at the end of the partitioning process all edges will be oriented. The only remaining problem is to find the initial sets X and Y, together with a transitive orientation on the edges which go between these sets.

In [436], a complex $O(n + m)$ algorithm is described for this subproblem, but there is a very simple algorithm for finding a source s in transitively orientable graph which is prime with respect to the substitution decomposition in linear time. Once s is identified, we create partitions s and V - s, orienting all edges out of s, and run the vertex partitioning algorithm.

Choose any vertex v as a start vertex. Create a set C of candidate source vertices, originally defined to be V - v. While more than two vertices remain in C, choose any vertex w of V - C. If w has at least one neighbor in C, remove all nonneighbors of w from C. When C has one vertex remaining, make this vertex into a source s. The proof of correctness for this simple algorithm is left for exercise 11.2; since every adjacency list is scanned at most once, the time taken to find s is $O(n+m)$.

For example, in the graph of figure 11.1, we might start with vertex 4 as v. Our initial candidate set is 2,5,6. If we use 3 to refine this set, we are left with 2,5 as candidate sources. If 7 is used next, we get 5 as our source, while if 1 is used we will use 2 as a source.

While we are performing partitioning, it is easy to maintain a linear extension of the current orientation. Initially, we start with s in front of the block V - s. Suppose that a block of B of the partition is split into neighbors and nonneighbors of some vertex w. If w occurs before B in the ordering, place $B \cap N(w)$ after B - $N(w)$; place these in the reverse order if w occurs after B. The final linear extension can be used to find the transitive orientation.

For example, suppose that 5 is used as a candidate source in figure 11.1. Our initial partition is (5)(1,2,3,4,6,7). When 5 splits the other block, we get (5)(1)(2,3,4,6,7). The next partition is (5)(1)(4,6,7)(2,3). If 2 splits (4,6,7) next, we get (5)(1)(4)(6,7)(2,3). If 4 splits (2,3) and (6,7), we get a linear extension (5)(1)(4)(7)(6)(3)(2), from which we can easily derive a candidate transitive orientation.

Suppose that we want to perform transitive orientation on the complement of G rather than G itself. If we take the complement of G and perform transitive orientation, we clearly pay a $\Theta(n^2)$ price. However, the steps of finding a linear extension for the complement involve repeatedly dividing sets into neighbors and nonneighbors. These steps can be done by looking at adjacency lists in G rather than adjacency lists in the complement. In fact, the algorithm of [367] finds a linear extension of a transitive orientation either of G or its complement in linear time, and this can be be used to recognize a number of graph classes efficiently.

The procedure described in this section finds an orientation even if no transitive orientation is possible. However, the orientation produced is transitive if G is a comparability graph. If we must check whether the orientation is transitive, the best known method is to compute A^2, where A is the adjacency matrix of A, to see whether this is equal to A. The current best matrix multiplication algorithm takes $O(n^{2.376})$ time; thus, this is the best known bound for comparability graph recognition. The open problem of recognizing comparability graphs more efficiently is discussed in exercise 8.14; the difference in time complexities between the verification procedure and the orientation procedure has been a great source of inspiration and frustration to me since I first designed an algorithm for efficient orientation many years ago. Indeed, in my first invitation to another university, my host said that when he first saw my transitive orientation algorithm he was surprised that I was smart enough to design a faster algorithm for transitive orientation, but that I could be so stupid as not to use it to solve the 'easier' problem of verifying that the result is transitive. In the next section, we see examples of problems for which the orientation can be checked in a different way, and thus algorithms which use transitive orientation can take less time than comparability graph recognition. More examples of such uses of transitive orientation appear in chapter 13.

We will also discuss briefly an alternative method for verifying transitivity; this was first communicated to me by McConnell, though the idea is fairly old. It is well known that the constants involved in the $O(n^{2.376})$ matrix multiplication are very large; some people feel that no algorithms yet developed which are faster than $O(n^{2.79})$ will ever be practical. We will discuss a method which is good for verifying sparse graph transitivity. It is trivial to verify transitivity in $O(nm)$ time, by doing a graph traversal from each vertex; we will show how to improve this to $O(m^{3/2})$, which is better for sparse graphs.

The technique is very simple. For each edge $x \rightarrow y$ in the transitive orientation, we look at all edges from y; whenever $y \rightarrow z$, we verify that there is an edge from x to z, and declare that the orientation is not transitive if we find some violation. The amount of work is clearly proportional to the number of triangles in G.

It is not so obvious that the number of triangles in a graph is $O(m^{3/2})$. For any triangle x,y,z, assign a cost of 1 to the edge between the two smallest degree vertices, where ties are broken arbitrarily. We will see that every edge accumulates $O(m^{1/2})$ charges, so that the total number of triangles is $O(m^{1.5})$.

Consider any edge x,y. Each charge assigned to this edge corresponds to a distinct neighbor z of x such that the degree of z is at least as large as the degree of x. If x has degree at most $m^{1/2}$, there cannot be more than $m^{1/2}$ such vertices z. On the other hand, there are $O(m^{1/2})$ vertices in a graph of degree greater than $m^{1/2}$ (exercise 11.11). Thus, if x has degree at least $m^{1/2}$, x,y also accumulates $O(m^{1/2})$ charges.

11.1.5 Permutation Graphs and Two Dimensional Partial Orders. The transitive orientation algorithm of the previous section immediately gives an $O(n^2)$ algorithm for permutation graph recognition and two dimensional partial order recognition. If we use the linear time algorithm for finding a transitive orientation, the time bound for each of these problems becomes linear. We will describe the algorithm for two dimensional partial order recognition.

A partial order represented by the transitive graph G is two dimensional if and only if $\bar{G}$, the complement graph of G, has a transitive orientation. If $\bar{G}$ has a transitive orientation, then a realizer for G can be constructed by adding G and a transitive orientation T of $\bar{G}$ to get the first list, and adding G to the reverse of T to get the second [223]. We can then check whether the two lists represent G by checking whether the relative position of each pair of vertices x, y in the two lists represents the adjacency of x and y correctly. We note that if the original graph G is not transitive, then one of the two lists created is not a linear order; this can be tested easily. To test whether an undirected graph G is a permutation graph, we transitively orient G and test whether the result is a two dimensional partial order.

The procedure described above requires $\Omega(n^2)$ time, since it involves taking the complement graph. Suppose that we have a transitive directed graph G, and a linear extension L of the transitive orientation of $\bar{G}$. It is easy to place each vertex v in the correct position of the two lists in time proportional to $|N(v)|$, which means that the lists can be constructed in $O(m+n)$ time.

Let $out_>(v)$ ($in_>(v)$) be the number of outneighbors (inneighbors) of v which come after v in L, and let $p(v)$ be the position of v in L. In the first list, v comes before all outneighbors of v, and all vertices after v in L which are neither outneighbors nor inneighbors of v in G. In the second list, v comes before all outneighbors of v, and all vertices before v in L which are neither outneighbors

nor inneighbors of v in G. Since these positions can be computed easily from the variables $out_>(v)$, $in_>(v)$, the number of inneighbors and outneighbors of v in G, and $p(v)$, we can find the position of all vertices v in the two lists in linear time.

We can then check whether the two lists represent G correctly in $O(n+m)$ time (exercise 11.13). Thus, given linear time algorithms for finding linear extensions of a transitive orientation for G and $\bar{G}$, two dimensional partial orders and permutation graphs can be recognized in linear time. This also allows us to construct a representation as containment of intervals in linear time, as described in chapter 3. Although the linear time algorithms are difficult at this time, the $O(n^2)$ and $O(m \log n)$ are described completely above.

11.1.6 Other Uses of Partitioning. The transitive orientation algorithm itself is useful for a number of other problems. Linear time transitive orientation for G and $\bar{G}$ gives linear algorithms for a number of optimization problems on comparability graphs (chapter 14), and linear time recognition of co-interval graphs (chapter 13). Recognition of a number of other graph classes reduces to recognition of permutation graphs; some of these are discussed in chapter 13.

Partitioning is used in recognition algorithms for circle graphs and circular-arc graphs ([177] and chapter 13). For these classes, and other classes with geometric models, we find an approximate geometric layout, and use partitioning to force correct intersections between sets.

Partitioning variants are used to decompose graphs with respect to the join decomposition [359] and to produce an ordering for chordal bipartite graphs. The chordal bipartite application is described in chapter 9.

In general, partitioning has been used to find algorithms by considering how an operation (such as transitive orientation) might be applied to a set of vertices rather than to individual edges or vertices. This technique for algorithm design was directly responsible for improvements in time complexities on a number of problems, including transitive orientation and the other uses described above.

The problem of recognizing comparability graphs without using matrix multiplication has been of interest to me for a long time. The following natural approach, designed by myself and Gustedt, may be of interest; note, however, that this problem is as hard as determining whether a graph is triangle free (exercise 8.14)

Assume G is prime; otherwise, each prime component of G is tested separately. Use partitioning to create an orientation G' of G. Create a different orientation G'' of G, by randomly making other choices each time we have a choice of splitting element during partitioning. If G' and G'' are not identical up to reversal, answer that G is not a comparability graph. If we repeat this some number of times and always reach the same orientation, answer that G is a permutation graph.

Exercise 11.17 shows that if G is not a comparability graph, there are choices which make G' and G'' different, but it may be the case that some orientations occur with very high frequency. Therefore, to make this algorithm efficient, we need to ask whether there is a small function $f(n)$ such that if G is not a comparability graph, after $f(n)$ steps, the probability that we will find two different orientations is greater than $1/2$. Alternatively, it would be interesting to exhibit a class of graphs for which the expected number of steps before finding a violation is $\Omega(n)$. As a weaker first step involving two notions of randomness, can we show that this algorithm is efficient in the randomized sense for proving that a random graph with fixed edge probabilities is or is not a comparability graph?

11.1.7 Restricted Substitution Decomposition. The starting point for many graph classes based on decomposition is the substitution decomposition. G can be reconstructed from the decomposition tree, plus quotient graphs for each module, where a quotient graph is the graph induced by taking one vertex from each child in the decomposition tree.

The ultimate restriction of substitution decomposition is to allow no prime modules in the decomposition tree. This gives exactly the class of cographs, discussed in chapter 8. As stated in that chapter, cographs have an implicit representation by virtue of being contained in the permutation graphs.

We can extend the above graph class and still have a graph class with $2^{O(nlogn)}$ members if we allow prime modules to have at most k children; this class is called the class of graphs with bounded substitution decomposition diameter. It is not clear whether graphs with bounded decomposition diameter have an implicit representation.

We can also allow other operations which reduce a graph G to G' in such a way that G can be reconstructed from G' plus $O(\log n)$ bits to be combined with bounded diameter decomposition. For example, I have seen papers which allow removal of degree 1 vertices to precede decomposition into at most k submodules; removing vertices which are not in any P_4s has also been considered.

This can allow more graphs to be represented, as shown in exercise 11.25. For space efficient representation, this could be extended in many ways; it is easy to see that allowing removal of degree k vertices for any fixed k still leads to an $O(n\log n)$ bit representation. Many other operations which can be represented efficiently could also be considered, and these pose potential challenges to the implicit graph conjecture.

11.2 Join Decomposition

A join decomposition algorithm is discussed in chapter 13 as part of a circle graph recognition algorithm. Recently, a variant of the join decomposition called 2-join has proved to be important in proving the strong perfect graph conjecture; this variant is discussed later in this chapter, along with skew partitions which were also used as part of the proof of the conjecture.

In join decomposition, vertices of a graph G are partitioned into V_1, V_2 such that both V_1 and V_2 have at least two vertices, and edges which go between the two sets induce a complete bipartite graph. The graph is then decomposed into subgraphs induced by $V_1 \cup m$ and $V_2 \cup m$, where m is a new 'marker' vertex adjacent to those vertices which have a neighbor in the other set; each of these subgraphs is decomposed recursively.

Since the join decomposition can be viewed as an extension of the substitution decomposition, it seems natural to try to restrict join decomposition in the same types of ways that substitution decomposition has been restricted. However, there is relatively little work dealing with restricted cases of join decomposition. This may be because the decomposition is less well known, or because join decomposition is not as helpful when solving optimization problems; one can compare the solutions for the independent set problems in join decomposable graphs [138] and graphs decomposable with respect to the substitution decomposition [373].

The one major class defined by restricting join decomposition is the class of graphs such that every induced subgraph on at least four vertices is decomposable by

the join decomposition. This class is exactly equal to the class of distance hereditary graphs discussed in chapter 12 [246], which explains why distance hereditary graphs have also been called completely separable graphs.

A natural generalization which has not been studied is to restrict the number of vertices on one side of a decomposition step to at most k for larger fixed values of k; in completely separable graphs, k is at most 2. It would still be possible to reconstruct the graph from the decomposition tree, so the number of graphs in the class is $2^{O(nlogn)}$, but this would not give an implicit representation of the class. As far as I know, the class of graphs defined by this restriction on join decomposition has not been studied. Many problems can be posed on this class of graphs; optimization, representation and characterization problems are possible candidate problems to study. I pose the following question not because implicit representation of these graphs is of particular importance in itself, but because I feel that the graph classes defined in this way are worthy of attention.

Open Problem 11.1: Is there an implicit representation of the class of graphs such that every induced subgraph can be partitioned into A, B such that $2 \leq |A| \leq k$, and the edges which go between A and B form a complete bipartite subgraph?

Variants of the above restriction are also possible, capturing different notions of a graph being completely decomposable. In general, there are at least 3 notions of completely decomposing a graph with respect to a given decomposition method. The above definition, requiring that every induced subgraph is decomposable, yields the smallest class of graphs, but defines a hereditary class by definition and thus is most relevant for questions of implicit representation. Since this is the strictest requirement, we choose to use this as our notion of graphs being completely decomposable with respect to a decomposition.

Two other natural notions of complete decomposability deal with decomposing a graph recursively until every prime component is of fixed size. For some purposes, it is most reasonable to define a graph to be completely decomposable if there exists a sequence of decomposition steps which make every prime component small, while for other applications one prefers requiring every prime component to be small in every possible decomposition of the graph. We will say that a graph is deterministically decomposable if every choice of decomposition step yields a complete decomposition, and that a graph is nondeterministically decomposable if there is a sequence of decomposition steps which yield a complete decomposition. It should be clear that every completely decomposable graph is deterministically decomposable, and every deterministically decomposable graph is nondeterministically decomposable, though the converse of these statements does not hold in general.

11.2.1 Generalized Join Decomposition. We will describe an extension of join decomposition due to Hsu [267] which raises a number of interesting open problems, and try to create restrictions which can have efficient representation schemes.

In the join decomposition, all vertices which have edges to the other side of the partition must have the same neighbors in the other side of the partition. In the generalized join decomposition, vertices can have different neighbors on the other side, but these neighborhoods must be ordered by set containment. More formally, vertices are partitioned into V_1, V_2 so that the edges which cross the split are a

chain graph; that is, there would be no induced $2K_2$ on the subgraph formed by these edges. We then add a marker vertex for each different neighborhood. In order to make the subgraphs of the partition smaller than G, Hsu requires the graphs induced by V_1 and V_2 to each contain at least 1 edge; other conditions, such as requiring some vertex in each subgraph to have no neighbors in the other subgraph, or requiring some pair of vertices to have the same neighborhood in the other subgraph, could also be used. Hsu was able to show that G is perfect if and only if all subgraphs found by the generalized join decomposition are perfect; the idea comes from the fact that any long hole or antihole will continue to exist in one of the components after the decomposition.

This decomposition raises a number of interesting questions. Can you find a split with respect to the generalized join decomposition in polynomial time? More generally, we can define a graph decomposition by forbidding any fixed forbidden bipartite subgraph (or set of subgraphs) in the graph formed by taking all edges which cross a split.

It can be shown [417] that it is NP-complete to determine whether a graph is decomposable with respect to the following forbidden subgraphs: $2K_2$ (i.e. generalized join) and $K_{2,2}$. Another natural decomposition which is NP-complete to find requires that the edges connecting the two sets form a matching [105]. On the other hand, it is possible to find a $K_{1,i}$ free decomposition for any fixed i if this means only that vertices in color class 1 must have at most i neighbors on the other side for any fixed i in polynomial time (exercise 11.27). Note that forbidding a graph with 1 vertex in A, 2 vertices in B, and one edge from an a vertex to a B vertex is equivalent to requiring that B is a module.

Open Problem 11.2: Can you specify completely which forbidden subgraphs define a decomposition which can be found in polynomial time?

Note that if we have a split V_1, V_2 defined by the generalized join decomposition, the number of bits needed to represent G is $nlogn$ + the number of bits to represent the subgraphs induced by V_1 and V_2 (exercise 11.24); if V_1 and V_2 are both large this is a vast improvement over the $\Theta(n^2)$ that we would need in general. Thus, an interesting class for representation problems arises if we require that a generalized is balanced, i.e. the number of vertices in V_1 is at most a constant times the number of vertices in V_2. Other interesting classes come from different ways of restricting the generalized join. The following restrictions seem natural, and may lead to graph classes which can be represented efficiently through the decomposition tree, but do not obviously have an implicit representation. Recognizing any of these restricted classes in polynomial time would be interesting.

A first restriction uses the various definitions of total decomposability described in the previous section, where we require that final prime graphs come from a small set of primitive graphs. The notions of completely decomposable, deterministically decomposable, and nondeterministically decomposable graphs are all of potential interest; we will focus here on the most restrictive class.

G is completely decomposable with respect to the generalized join if every induced subgraph with at least four vertices has a generalized join decomposition.

I believe that the definition above is presented here for the first time. Exercise 11.23 shows that all permutation graphs are completely decomposable. Again, we select one open problem to feature from a large number of potential problems on the class.

Open Problem 11.3: Is there a polynomial algorithm for recognizing graphs which are completely decomposable with respect to the generalized join decomposition?

Another restriction which reduces the amount of information which must be shared between subgraphs of a split is to allow only a constant number of marker vertices on each side of the split; i.e. a constant number of different neighborhoods when restricted to the other subgraph. This notion is similar to one which arises during our discussion of algorithms for dealing with graphs having bounded clique-width below.

11.3 Recursively Decomposable Graphs

This section deals with an attempt by Borie, Parker, and Tovey to find a unifying idea behind graph classes defined in terms of decomposition schemes. By focusing on a unifying approach, the authors raised some interesting issues dealing with the distinction between having an efficient representation and finding an efficient representation. The results in this section are taken from [58].

A number of graph classes are defined using the following type of construction, which can generally be viewed either as a composition or as a decomposition. Graphs are composed or decomposed in such a way that interactions between the graphs is localized to a small subset of vertices, which are called terminals. A k-terminal graph is a graph in which at most k vertices are designated as terminals.

Viewing the operation as a composition, we combine k-terminal graphs according to one of a fixed set of rules. The composition rules may combine the graphs by identifying terminals of one graph with terminals of the other graph, and choosing a new set of terminals from the set of terminals of the two subgraphs. Edges in each of the original subgraphs are preserved in the new graph. A graph class is defined by allowing all graphs which can be built up using these rules from a finite set of base graphs.

We can view this process as a graph decomposition as well, by allowing a graph G to be decomposed into G_1, G_2 if G_1 and G_2 can be combined to get G using a composition rule. This gives rise naturally to a decomposition tree; an internal node is added for the decomposition step (typically labeled by the type of rule used), with recursive decompositions of G_1 and G_2 as the children of the node. A graph G is in the class if G has a decomposition tree such that all leaves are base graphs. Thus, classes which can be defined in this way are called recursively decomposable classes of graphs.

Examples of well known recursively decomposable graph classes are trees, series parallel graphs (see problem 1.3), Halin graphs and partial k-trees; see [58] for these and other examples. Many optimization problems can be solved easily if we are given a decomposition tree/composition sequence. This can be done since interactions between graphs are restricted to the terminals. For example, let us assume that we want to solve the maximum independent set problem, and we are given a composition sequence. For each graph in the sequence, we compute and store 2^k values, where k is the maximum number of terminals; each value corresponds to the maximum independent set if we are restricted to use a particular subset of the terminals. It is easy to compute the values for the combined graph if we know the values of the subgraphs, and we know which composition rule was used to get the new graph.

Note that the decomposition tree of a recursively decomposable graph gives a space efficient representation, which is not necessarily implicit. In fact, graph classes defined using this technique must be hereditarily sparse (every subgraph contains a small degree vertex), and have an implicit representation for this reason. Although this may seem to resemble the notion of bounded substitution decomposition diameter, neither notion is stronger than the other (exercise 11.29).

By focusing on a general technique rather than on a particular class, Borie, Parker, and Tovey noted an important distinction between different decomposition rules. Since many problems are efficiently solvable on recursively decomposable graphs given a decomposition sequence, we want to be able to take an input graph G and find a decomposition, if it exists. The natural approach, which works on many classes, is to find any composition rule which breaks G into G_1 and G_2, and to attempt to decompose G_1 and G_2 recursively. Since there are a fixed number of rules, and a fixed number of ways to choose terminals for each subgraph, it is not difficult to make the argument that we can find a decomposition into G_1, G_2 in polynomial time if such a decomposition exists (exercise 11.30).

However, finding a single legal decomposition step efficiently does not necessarily imply that an entire decomposition sequence can be found efficiently. It is possible that if one rule is used at the first step, the graph decomposes into G_1 and G_2, where both G_1 and G_2 are recursively decomposable, but that if another rule is used, G may be decomposed into G'_1 and G'_2, where one or both of G'_1,G'_2 is not recursively decomposable using these rules. Readers familiar with the Church-Rosser property will be comfortable with this distinction. [58] calls a class of graphs defined by a set of decomposition rules deterministic if membership can be decided by arbitrary application of any valid rule to decompose a graph. Recognition and optimization problems can be solved in polynomial time for all deterministic classes, while this is not necessarily true for recursively defined classes which are not deterministic.

Borie, Parker, and Tovey classify decomposition rules as follows. A (k,u,r) operation works by decomposing a k-terminal graph G into two k-terminal graphs so that at most r vertices of G were terminals in either subgraph, with at most u of these becoming nonterminals in G. The class $[k,u,r]$ is defined by allowing all (k,u,r) operations. They then show that the class $[k,u,r]$ is deterministic if and only if it has one of the following forms; $[2,r\text{-}2,r]$, $[3,r\text{-}3,r]$, $[k,0,k]$, or $[k,1,k+1]$.

The immediate consequence is that if all rules are of one of the forms given above, you know that the class is deterministic; this holds for such classes as Halin graphs and biconnected series-parallel graphs.

11.4 Clique-width and NLC-width

The following interesting notions of graph decomposition were first introduced in [134]. Certain types of logical queries can be answered in polynomial time if we are given this form of decomposition for graphs of bounded clique-width or NLC-width; this leads to open problems which will be discussed in the optimization chapter.

The clique-width of G is the minimum number of labels we need to construct G from a set of isolated vertices, if the following operations are allowed. We describe a method for building graphs of clique-width at most k.

Start with any graph on at most k vertices; give each vertex its own label. At any time, we may make a label change as follows: change all vertices with label i in a clique-width k graph to have label j. At any time, we may add edges as follows: for some pair of distinct labels i, j, add undirected edges from all vertices with label i to all vertices with label j. We may also combine two graphs with clique-width k using the disjoint union operation. The graph can be completely described by a composition tree, in which internal nodes have a description of the operation performed on the subtree of descendants of that node.

Clique-width is closely related to a parameter called NLC-width. In building a graph with NLC-width k, we again are allowed to use k labels on vertices, and can at any time perform a relabeling procedure exactly as in the definition of clique-width. The primary difference between the classes comes in the way in which edges are introduced; for NLC-width k graphs, edges are added between subgraphs when the union operation is performed. To combine two NLC-width k graphs, we give a k by k bipartite graph, and add edge between vertices with label i from graph 1 to vertices with label j from graph 2 if and only if there is an edge from i to j in the given bipartite graph. We are also allowed to perform a relabeling operation, telling us how labels should be assigned in the new merged graph.

In exercise 11.31, you are asked to show that the clique-width and NLC-width can differ by at most a factor of 2. Since our concern is primarily with the fact that the number of labels is constant, we will be able to speak interchangeably of graphs with bounded clique-width and graphs with bounded NLC-width. Graphs with clique-width 2 (or NLC-width 1) are exactly the cographs (exercise 11.32).

Distance hereditary graphs have clique-width at most 3 [230], while some surprisingly small classes such as unit interval graphs, grids, and bipartite permutation graphhs have unbounded clique-width [230, 73].

Since two graphs are combined using operations which can be described using a constant number of bits for graphs of fixed clique-width, the total number of bits needed to reconstruct a graph of fixed clique-width is $O(n)$ for unlabeled graphs, and $O(n\log n)$ for labeled graphs. However, this representation is not implicit. We will describe an implicit representation for clique-width k graphs below; we note that this is the first time we will create an implicit representation that does not immediately give a constant time adjacency testing algorithm.

Both the recognition and construction problems are solved for clique-width 3 [122], but are open for graphs of clique-width k for all $k > 3$. Since constructing the decomposition allows you to solve optimization problems for graphs of fixed clique-width, even an approximation (for example, giving a construction sequence which never uses more than $2k$ labels for graphs of clique-width k) would be very useful. An algorithm which gives an $O(\log n)$ clique-width decomposition for graphs with bounded clique-width is given in [281].

Open Problem 11.4: Find a polynomial time recognition algorithm for graphs with clique-width k, when $k > 3$.

Open Problem 11.5: Find the appropriate construction sequence, or a good approximation, for graphs of fixed clique-width $k > 3$.

We now discuss a new approach which gives an implicit representation of graphs with bounded clique-width, and allows certain optimization problems to be solved in polynomial time on the class even when the graph is given in adjacency matrix form.

A set S of vertices is a k-module if S can be partitioned into $S_1, S_2, ..., S_k$ such that for each S_i, S_i is a module in the graph induced by V - $S \cup S_i$; i.e., the vertices of S_i are indistinguishable to vertices of V - S. This is a generalization of the concept of homogeneous pair, as discussed in [181].

A balanced k-module is a k-module of size at least $|V|/3$, and at most $2|V|/3$. In exercises 11.33 and 11.34, we ask the reader to show that every graph with clique-width k has a balanced k-module, and that the balanced k-module can be found in polynomial time given only the adjacency matrix for G.

We can use balanced k-modules to get an implicit representation of clique-width k graphs as follows. Construct a tree, which we will call a balanced k-module decomposition tree, from G by recursively finding a balanced k-module in the current set, and placing the k-module in the left subtree and other vertices in the right subtree. Since the size of each set is at most $2/3$ the size of the set at the previous level, the height of such a tree is $O(\log n)$.

For a vertex v, we store a constant number of bits for each level of the balanced k-module decomposition tree. We store one bit which tells us whether v is placed in the right or left child at this level. For a vertex in the left child (i.e. part of the k-module) we store the value i such that S_i contains v. For a vertex v in the right child, we store all subset numbers corresponding to subsets adjacent to v; since k is constant, this is a constant number of bits.

We can determine whether vertices x, y are adjacent by finding the first level at which exactly one of x and y is a left child, and then determining whether the vertex placed in the right child is adjacent to the subset S_i containing the vertex placed in the left child. It is easy to determine this from the bits stored at x and y, so this is an implicit representation. Note, however, that the time needed to determine adjacency in this way is $O(\log n)$, leaving the following open problem.

Open Problem 11.6: Find an implicit representation of clique-width k graphs such that adjacency can be determined in constant time.

We can use the fact that every clique-width k graph has a balanced k-module to solve a number of optimization problems on the class in polynomial time, if the decomposition is not given as input. A very nice general scheme for solving optimization problems if the decomposition is given was presented in [135]; we note that this is essentially a dynamic programming approach, keeping track of all 'relevant' solutions for a portion of the decomposition tree.

The balanced k-module approach uses recursion rather than dynamic programming. We will illustrate the technique on the vertex cover problem; this is generalized to a variety of problems in [57].

Let S be a balanced k-module, and let $S_1, S_2, ..., S_k$ be the subsets of S. We can determine whether a set is a vertex cover in G if we know which vertices of V - S are in the set, and which sets S_i are completely contained in the vertex cover. This leads to a recursive technique in which we consider all of the 2^k possibilities as to which sets S_i are completely contained in the cover. For each possibility, we solve two problems recursively; find the optimal cover of the sets S_i not contained in the cover, and find the optimal cover of vertices of V - S which are not adjacent to any sets S_i which are not fully contained in the cover. The cost of the possible choice of sets S_i is then the sum of the costs of these two subproblems, plus the number of vertices in the sets S_i contained in the cover and the number of vertices of V - S which are adjacent to some set S_i which is not fully contained in the cover.

This leads to a recursion of the form $T(n) \leq 2^{k+1}T(2n/3) + O(n^2)$, which gives a polynomial solution for any fixed value of k.

A number of problems can be solved directly from this recursive approach. Previously, it had been shown that every problem posed in second order monadic logic with the restriction that quantification is allowed only over sets of vertices could be solved on clique-width k graphs, when the decomposition tree was given as part of the input [132]. In [57] we show that certain problems posed in a more restricted form of second order monadic logic proposed can be solved in polynomial time using balanced k-modules; this eliminates the need for the decomposition tree as part of the input. There is a price to be paid for not having the decomposition tree; the time bound goes from linear (i.e. $f(k)m$) to $O(n^{f(k)})$. Other problems solvable given the decomposition tree are discussed in chapters 14 and 18.

11.5 Clique Separator Decomposition

This form of graph decomposition is useful for solving optimization problems on some graph classes defined in this book, such as EPT graphs, as will be shown in chapter 14. We give an efficient algorithm for finding the decomposition, and define a new restriction which could give rise to classes which have efficient representations.

A clique separator (also called a clique cutset) is a clique C such that the subgraph induced by G - C is disconnected. The clique separator decomposition takes any clique separator C of G, splits G into subgraphs consisting of C plus each connected component of G - C, and decomposes each subgraph recursively. The first polynomial algorithm for finding a clique separator was due to Whitesides [483] and found a single separator in $O(nm)$ time; this implies an $O(n^4)$ clique separator decomposition algorithm (exercise 11.35). The following algorithm is due to Tarjan, and although a single clique separator still is found in $O(nm)$ time, the entire clique separator decomposition is computed in $O(nm)$ time. Proofs of the correctness of the algorithm are contained in [454]. The fastest known algorithm for finding a single clique separator runs in $O(n^{2.69})$ time, using fast matrix multiplication as a subroutine [332], and is based on Whitesides' algorithm, which is outlined in exercise 11.36.

Tarjan's clique separator decomposition algorithm uses the idea of fill added to a matrix during Gaussian elimination. Given an ordering of the vertices, one computes fill by stepping through the vertices in order, and adding edges between all neighbors of the current vertex which occur after the vertex in the ordering. Note that fill edges can themselves be used to create further fill; for example, suppose that we are given a P_4 3,1,2,4 and an ordering 1,2,3,4. Vertex 1 causes a fill edge (2,3), and 2 causes a fill edge (3,4) since 2 has both 3 and 4 as neighbors when it is processed.

The problem of finding an ordering which causes the minimum possible amount of fill has applications in Gaussian elimination, and thus was one of the twelve featured open problems in the first edition of Garey and Johnson's well known book on NP-completeness [206]. The minimum fill problem was shown to be NP-complete by Yannakakis [491].

The minimal fill problem asks for an ordering such that the fill created by this ordering does not properly contain the fill created by any other ordering; thus every minimum fill ordering is a minimal fill ordering, but not vice-versa.

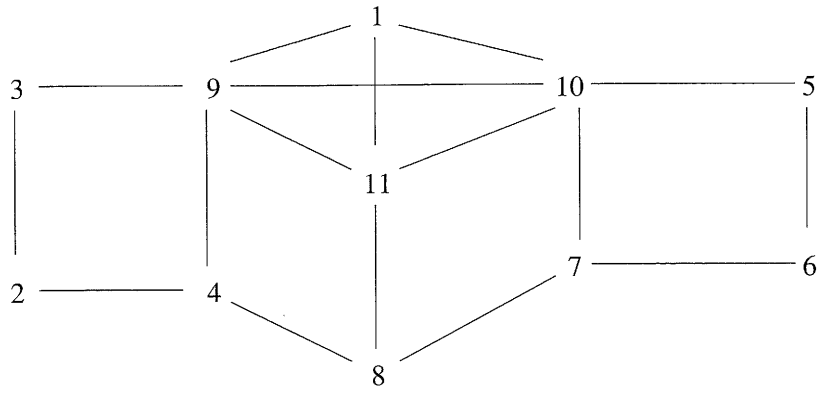


Figure 11.3 Sample Graph for Fill Edges

Consider the graph of figure 11.3, taken from [454]. Fill edges for the ordering 1,2,3,...,11 are (3,4) (from 2), (8,9) (from 4), (6,10) (from 5), (8,10) (from 7). Tarjan's algorithm first computes an ordering which gives minimal fill, and uses this to find clique separators. Note again the difference between minimal and minimum fill, since finding an ordering with minimum fill is NP-complete.

The following algorithm, due to Rose, Tarjan and Lueker, finds a minimal fill ordering in $O(nm)$ time. An alternative $O(nm)$ algorithm for finding a minimal fill ordering is presented in [388]. The algorithm is related to the lexicographic breadth first search algorithm described in chapter 13; in that algorithm, each vertex v chosen divides sets into neighbors and nonneighbors of the currently chosen vertex, placing neighbors after nonneighbors. The additional work in this algorithm comes from the fact that the algorithm is trying to divide sets into neighbors and nonneighbors of v not in the original graph, but in the filled graph; thus it computes the known fill involving v before dividing the set using a characterization of fill given in exercise 11.38.

Start with all vertices in a single set. Repeatedly remove a vertex v from the last set, and place it in the output list (the element chosen at step i is placed at position n-i+1 of the output list, as in lexicographic breadth first search). Let knownfilled(v) be the set of vertices w such that there is a path from v to w in G using only intermediate vertices which come from sets which precede w in the current ordering; note that all neighbors of v are included in this group, since no intermediate vertices are used. Divide every current set S into S - knownfilled(v), knownfilled(v) $\cap$ S, and place the former set before the latter. It is not difficult to compute this ordering in $O(nm)$ time.

Consider running this procedure on the graph of figure 11.3. We can start by placing vertex 11 last; this creates the ordering (2,3,4,5,6,7)(1,8,9,10) 11. Suppose 10 is picked next; the path 10,7,8 uses intermediate vertices only in sets before 8, so the new ordering is (2,3,4,6)(5,7)(1,8,9) 10 11. If 9 is chosen next, we get (2,6)(3,4)(5,7)(1,8) 9 10 11; note that the path 9,4,8 causes 8 to be treated as a neighbor of 9. If 8 is chosen next, we get (2,6)(3)(4)(5)(7)(1) 8 9 10 11. The first vertex which distinguishes 2 from 6 is 7; we thus get the ordering 2 6 3 4 5 7 1 8 9 10 11. Edges of the fill from this ordering are (3,4) (caused by 2), (5,7) (from 6), (8,9) (from 4), (8,10) (from 7).

We now assume that L is a minimal fill ordering. For each v, let C(v) be the set of vertices w such that w comes after v in L, and (v,w) is either an edge of G or part of the fill caused by this ordering.

Let A be the connected component of the graph induced by V - C(v) which contains v, and let $B = V$ - (C(v) $\cup$ A). If C(v) is a clique in G and B is nonempty, then we have found a successful decomposition into $A \cup$ C(v) and $B \cup$ C(v).

Tarjan shows that if there is a clique separator of G, some clique separator is of the form C(v); this gives an O(nm) algorithm for finding a single clique cutset. He then makes several observations regarding further decomposition of the subgraphs which reduce the running time of finding the entire clique separator decomposition, even though finding a single separator has the same time bound as Whitesides' algorithm. These all come from the fact that the relative ordering of vertices produced by Tarjan's algorithm in each subgraph can be the same as the ordering in L, and the sets C(v_i) in the subgraphs are the same as the sets C(v_i) in G if this ordering is used.

The first observation which improves the time bound of clique separator decomposition using this algorithm is that after a successful decomposition step, the subgraph $A \cup$ C(v) is indecomposable [454]; in other words, the algorithm always chooses to separate a minimal set containing v. This implies that the number of atoms of a subgraph decomposed in this fashion is O(n); each decomposition step removes at least one vertex from the subgraph which needs to be decomposed further.

The second observation is that if the decomposition step defined by C(v) is unsuccessful in G, it cannot be successful in either subgraph formed by a successful decomposition step. Therefore, each vertex v is examined once to determine whether C(v) forms a clique separator, which means the entire algorithm runs in O(nm) time.

Thus, for the graph of figure 11.3 and the ordering 1,2,3,..,11, we find that 9,10,11 separates 1 from the rest of the graph and is a clique, decomposing the graph into the subgraphs induced by 1,9,10,11 and 2,3,4,5,6,7,8,9,10,11. C(2) = 3,4 is not a clique, but C(3) = 4,9 separates 3,2 from the rest of its component. The next clique C(v) is C(6) = 7,10, which splits 5 and 6 from its component. The remaining clique C(v) is C(8) = 9,10,11; this is not a clique separator since the component remains connected with the separator removed. Thus, the clique separator decomposition tree found from this minimal fill ordering is shown in figure 11.4.

For the coloring, clique, independent set, and minimum fill problem, if there are efficient algorithms for atoms found by the clique separator decomposition, the problem can be solved efficiently for the entire graph. Clique separator decomposition is used in chapter 14 for dealing with EPT graphs, and in chapter 13 for recognizing path graphs.

Although even chordal graphs, which are completely decomposable with respect to clique separator decomposition, are too large to have an efficient representation, we can define a variant of this decomposition which could be interesting with respect to counting and representation. The following notion was proposed by Ma. Define a simple clique separator (the analogy is with simple vertices in strongly chordal graphs) C to be a clique separator such that the neighborhoods of vertices in C are ordered by set containment within each component. The fundamental idea is that while $\Theta(n^2)$ information must be maintained to reconstruct G from a

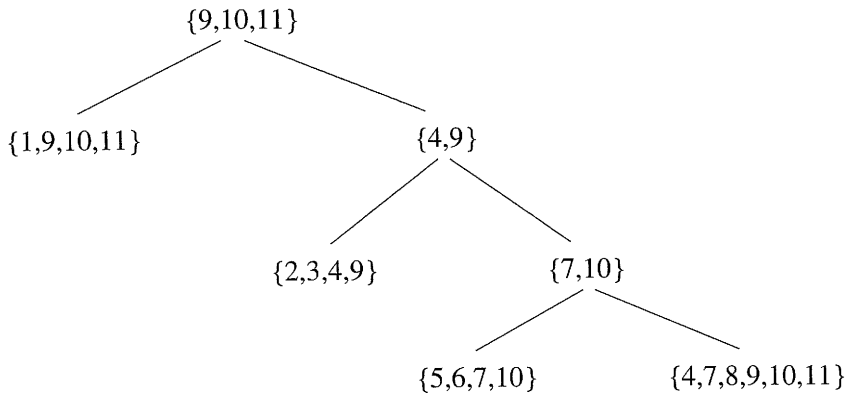


Figure 11.4 Decomposition Tree for the Graph of Figure 11.3

standard clique separator and the graphs induced by the corresponding connected components, we need only $O(n\log n)$ bits to reconstruct G from a simple clique separator. I do not know of any work on this; it might be of interest to study graphs which are completely decomposable using simple clique separators.

11.6 Skew Partition

The skew partition of a graph has suddenly become very interesting to many researchers, since it is critical to a recently announced proof of the strong perfect graph conjecture [102], and poses one of the last obstacles toward recognizing perfect graphs in polynomial time. Therefore, I include this in the decomposition chapter, although it is less closely tied to efficient representation of graphs than the other decompositions.

The strong perfect graph conjecture is proved roughly as follows. We will call certain classes of graphs basic perfect graphs. Basic perfect graphs include bipartite graphs, co-bipartite graphs, line graphs of bipartite graphs, and complements of line graphs of bipartite graphs, as well as one more obscure class of graphs called bicographs in [102]. These bicographs are formed from a matching M and the complement of a matching C, such that for every matched pair m_i, m_j of M and every anti-matched pair c_k, c_l of C, m_i is adjacent to exactly one of c_k or c_l, and m_j is adjacent only to the other vertex in the anti-matched pair.

Every perfect graph which is not basic , is decomposable (in either the graph or its complement) using either a decomposition called 2-join, a decomposition called skew partition, or a decomposition called an M-join. No minimal imperfect Berge graph is decomposable in this way (technically, the authors introduced a restricted form of skew partition called balanced skew partition which we will see later, since they were unable to show directly that a minimal imperfect graph cannot have a general skew partition; details of the proof are beyond the scope of this book), which implies that there is no minimal imperfect Berge graph and thus that every Berge graph is perfect.

At first glance, it might seem that existing polynomial algorithms for finding 2-joins, M-joins, and skew partitions, together with algorithms for testing whether a graph is in each of the basic classes, would imply perfect graph recognition is

polynomial. If a graph is basic, answer that the graph is perfect. If the graph is neither basic nor decomposable, answer that the graph is imperfect. Otherwise, the graph must be decomposable. If we could simply test for perfection in each block of a partition according to the decomposition, we would be able to recognize perfect graphs in polynomial time. Unfortunately, tests for perfection in graphs decomposable using skew partitions and 2-join is not as simple as this. We deal with issues involving skew partitions in this section, and 2-joins and M-joins in the next section.

A skew partition is a partition of the vertex set into four nonempty subsets A, B, C, D such that every vertex of A has an edge to every vertex of B, while every vertex of C is nonadjacent to every vertex of D. In other words, there is a cutset $(A \cup B)$ which induces a subgraph which is co-disconnected.

A special case of a skew partition is a star cutset, in which $|A| = 1$. Star cutsets were studied in [106], which proved that no minimal imperfect graph has a star cutset, posing and leaving open the problem for the more general case of skew partitions. The same paper shows that star cutsets can be recognized in polynomial time, using the following theorem.

Theorem 11.1 [106] *G has a star cutset if and only if G contains a vertex w such that G - $N[w]$ induces a disconnected subgraph, or G contains nonadjacent vertices u, v such that $N(u) \subseteq N(v)$.*

Clearly, the theorem above shows that star cutsets can be found in $O(nm)$ time. The paper [332] reduces the time complexity for testing the conditions of this theorem to $(n^{2.79})$.

It is considerably more difficult to find a polynomial time skew partition algorithm. A subexponential algorithm was discovered in [185], and eventually a polynomial time algorithm in [192]. We will give a brief outline of the latter algorithm; note that the notation has been changed from the original paper.

For each subset of A, B, C, D, there is a set of vertices of G which are candidates for all of these sets. We create n^4 subproblems, choosing to place one vertex a, b, c, d in each of A, B, C, D; this allows us to ignore the possibility that some set of the partition might become empty. Initially, since every vertex of G can be in any set of the skew partition, all vertices of G (with the exception of the four vertices chosen to be in specific sets) are placed in a set called S_{ABCD}; this indicates that vertices in this set are eligible for placement in A, B, C, or D. The algorithm enforces the property that for every vertex a which can only be in A (B), no nonneighbor of a has set B (A) as a possible placement, and that for every vertex c which can only be in C (D), no neighbor of c has set D (C) as a possible final set. Note that some vertices are removed from S_{ABCD} to enforce this restriction, based on adjacencies to a, b, c, d. Some vertices might even be forced into a single possible set, which causes further refinements.

The algorithm uses recursion to successively eliminate the set S which gives the most possibilities to vertices in S. Thus, the algorithm initially eliminates the set S_{ABCD} by creating a polynomial number of restricted choices for each vertex in S_{ABCD}. The same strategy is then used to eliminate possibilities for vertices which can be in 3 subsets of the skew partition, until we have a polynomially solvable special case as described later. The recursions are all of the same basic form; any recursion of the form $f(n) = cf(n/k)$ has a polynomial solution. We will discuss the

procedure from [192] for eliminating S_{ABCD}, by creating a polynomial number of subproblems; eliminating other sets is dealt with by similar procedures.

The simplest case arises when there is a vertex v which is adjacent to at least $1/10$ of the vertices in S_{ABCD}, and is nonadjacent to at least $1/10$ of the vertices in S_{ABCD}. We can try placing any such v in each of the 4 sets; placing v in any set eliminates at least $1/10$ of the vertices from being in some set (placing v in A eliminates nonadjacent vertices from B, placing v in C eliminates adjacent vertices from D, and so on), and we get 4 subproblems in which S_{ABCD} is at most $9/10$ of its current size.

If no such vertex v exists, let W be the members of S_{ABCD} adjacent to more than $9/10$ of the other vertices in S_{ABCD}, and let X be the members of S_{ABCD} adjacent to fewer than $1/10$ of the other vertices in S_{ABCD}. For the next subcase, assume that both W and X contain at least $1/10$ of the members of S_{ABCD}.

At least one of A, B, C, D has more than $1/10$ of the members of S_{ABCD}. We consider each of these possibilities; if A (B for the second case) has more than $1/10$ of the vertices, then no vertex from X can be in B (A in the second case), while if C (D) has more than $1/10$ of the vertices, no vertex from W can be in D (C). Therefore, we have 4 subproblems, and S_{ABCD} is at most $9/10$ of its previous size in each subproblem.

The trickiest case is when one of W, X is much larger than the other; the cases are completely symmetric for the algorithm, so assume that W contains at least $9/10$ of the elements of S_{ABCD}.

Move any vertex v from W to a new set O, and remove nonneighbors of v from W; let $W' = W - v - \mathrm{N}(v)$. Repeatedly add a vertex u from $W - W' - O$ to O, and remove nonneighbors of u from W', until either $W - W'$ contains at least $1/10$ of the vertices of S_{ABCD} or all vertices from W are in either O or W'.

Suppose that this halts when $W - W'$ contains at least $1/10$ of the vertices of S_{ABCD}. It is clear both that W' contains more than $1/10$ of the vertices of S_{ABCD}, and that every vertex of W' is adjacent to every vertex of O. Therefore, if any vertex of O is in C (similarly D), no member of W' can be in D (C), so the size of S_{ABCD} will be at most $9/10$ its previous size. Otherwise, assume that the first vertex v of O is in A; the case in which v is placed in B is symmetric. Every vertex added to O was nonadjacent to a vertex previously placed in O, and thus (since we have restricted vertices of O to A and B) each vertex added to O must be in A. Similarly, every vertex removed from W' was nonadjacent to a vertex from O, and thus cannot be in B; since we stopped with $1/10$ of the vertices of S_{ABCD} in $W - W'$, this reduces the size of S_{ABCD} to at most $9/10$ its previous size. Therefore, we again have 4 subproblems (placement of a vertex of O in C, in D, $O \subseteq A$, $O \subseteq B$, and in each case at least $1/10$ of the vertices of S_{ABCD} are eliminated because they cannot be placed in one of the sets.

Suppose the procedure halts because every vertex of W is in O or W'. Note that every vertex of O has an edge to every vertex of W', and that O is complement-connected. We simply restart the procedure, choosing to remove any vertex of W' to a set O_2, and continue until either at least $1/10$ of the vertices of S_{ABCD} have been removed from W' by the current O set (in which case we act as if this was the first O set chosen, and create 4 subproblems as described above), or the procedure stops growing the current O set and the total number of vertices in the O sets is at least $1/10$ of the vertices in S_{ABCD}. Since all edges exist between vertices in the O sets and vertices in W', and both sets have at least $1/10$ of the vertices in S_{ABCD},

we can consider the cases in which C contains a member of an O set (in which case no member of W' is in D), and C contains no members of an O set, in which case at least $1/10$ of the vertices (members of the O sets) are removed from S_{ABCD}.

The number of subproblems created is governed by a recurrence $\mathrm{P}(n) \leq 4\mathrm{P}(\frac{9n}{10})$, which has a polynomial solution. Note that for transforming this into a polynomial number of subproblems, we only recursively look at the graph induced by vertices which remain in S_{ABCD}. At the end of this recursion, we have a polynomial number of subproblems; for each subproblem, every vertex has a maximum of 3 sets which may contain the vertex.

I have described only one part of the algorithm of [192] here, the elimination of the set S_{ABCD} by creating a polynomial number of subproblems. That paper then uses similar techniques to create a polynomial number of subproblems of a special form.

One special form involves elimination of sets corresponding to a vertex having the possibility of belonging to three different subsets, creating a polynomial number of subproblems in which every vertex is restricted to one of at most two sets from A, B, C, D. This is shown to be solvable using an algorithm for 2-satisfiability. To see that it is polynomially solvable, one can choose any vertex restricted to two sets, choose to try it in one of the sets, and find all implications (i.e. placing in set A would force all nonneighbors out of set B and into their other candidate set, which would have further implications). If no contradiction is reached, all implications can safely be finalized, while if a contradiction is reached then the vertex must be placed in its other candidate set.

The other special form of solvable subproblem is when there is a pair of sets IJ, where I is either A or B and J is either C or D, and no vertex has the possibility of being in both set I and set J. Assume without loss of generality that no vertex has the possibility of being in both set A and set C. If this case is reached, A and C are chosen to be exactly those vertices which have only this set as a possibility. Since all other vertices have B or D as a possibility and there are no adjacency constraints between vertices chosen to be in B and vertices chosen to be in D, set B can be chosen to consist of all vertices which still have set B as a possibility, and D can be chosen to contain all other vertices. Since we maintain the property that nonneighbors of the vertices in A are excluded from set B, and neighbors of C are excluded from D, this will always be a valid skew partition.

Therefore, a skew partition can be found in polynomial time; incidentally, the exponent of this polynomial as described in [192] is unusually high (using a very simple but natural time analysis, this even goes over n^{100}), and can undoubtedly be reduced. However, using skew partitions to test for perfect graphs would seem to involve a form of recursive decomposition. While it is true that no hole or antihole can contain vertices from A, B, C, and D (exercise 11.39), testing for skew partitions recursively on the graphs induced by ABC, ABD, ACD, BCD could potentially take exponential time; see exercise 11.42. Nevertheless, extending this polynomial algorithm from a single skew decomposition step to a general test of perfection given the new theorems seems to be a very promising approach towards recognizing perfect graphs in polynomial time. It is quite possible that duplication of work in the various recursive calls might be avoidable, and the fact that various sets must obey adjacency patterns of the original skew partition could be useful.

The major open problem, which seems to be the major obstacle to recognizing perfect graphs in polynomial time, is to look for odd holes or antiholes efficiently as a graph is decomposed by repeated skew partitions.

Recursive decomposition using special types of star cutsets has been used to solve some algorithmic problems, such as recognition of a class of graphs called quasi-Meyniel graphs [193]. Special skew partitions have not yet been studied for special classes of perfect graphs. Exercises 11.40 and 11.41 deal with decompositions for weakly chordal and comparability graphs.

We mentioned briefly that the proof of the strong perfect graph theorem actually shows that nonbasic perfect graphs can be decomposed using either 2-joins, M-joins, or a special form of skew partition in the graph or its complement. A skew partition A, B, C, D is balanced if there is no odd induced path between nonadjacent vertices u, v of $A \cup B$ (similarly $C \cup D$) which uses only vertices of $C \cup D$ ($A \cup B$), and also no odd induced path of this form in the complement; note that this would correspond to an odd path using vertices of $C \cup D$ in the complement between vertices from $A \cup B$ which are adjacent in G, or a similar path between vertices of $C \cup D$. At this time, no polynomial algorithm is known for finding balanced skew partitions; this might be a promising approach to take.

Open Problem 11.7: Is there a polynomial time algorithm for finding a balanced skew partition in a graph?

The problem above is also interesting if we restrict our attention to Berge graphs. However, some interesting side issues might arise if you have an algorithm which finds balanced skew partitions in Berge graphs, but which might not be balanced if G is not Berge. As far as I know, even the problem of determining whether a given skew partition is balanced seems to be an open problem, and this could pose a problem if the assumption that the partition was balanced was used as part of a recognition algorithm.

11.7 2-Join

The proof of the strong perfect graph conjecture works by decomposing perfect graphs which are not contained in certain basic perfect graph classes using either a skew partition, as discussed above, a 2-join, or an M-join. In general, the 2-join decomposition, described more precisely below, generalizes the join decomposition in much the same way that k-modules generalize modules. The join decomposition requires edges which go between the subsets of the partition to form a complete bipartite graph; in the 2-join, these edges must form two complete bipartite graphs. The M-join is a type of homogeneous pair (or 2-module), and will be discussed at the end of this section.

The definition of a 2-join used in the proof of the strong perfect graph conjecture is somewhat different than the definition I had encountered earlier, which is described in exercise 13.34 as a generalization of the join decomposition and as a special case of a k-join, and includes both the 2-join and M-join as special cases. Consider a partition of the vertices of a graph into sets X, Y, such that X has nonempty disjoint subsets A_1,B_1 and Y has nonempty disjoint subsets A_2,B_2, there is a path in X from A_1 to B_1 and a path in Y from A_2 to B_2. X, Y is a 2-join in the new sense if all edges exist between A_1 and A_2, all edges exist between B_1 and B_2, there are no other edges between X and Y, and if $|A_i| = |B_i| = 1$ and the

set (X or Y) containing A_i and B_i induces a simple path, this path contains at least 3 edges.

It is not clear to me whether there is a polynomial algorithm for finding 2-joins under this definition. There is an algorithm for finding 2-joins using the alternative definition, as asked in exercise 13.34; this algorithm involves selecting initial vertices for the various sets, and adding vertices if they are forced into a set. The key difference in definitions that could make it harder in the new sense is the requirement of a path from A_i to B_i. However, if there is no path between A_i and B_i then the graph can be decomposed using the join decomposition rather than 2-join, and this would be at least as useful both for algorithms and proofs of perfection. Therefore, the question of finding 2-joins in the new sense in polynomial time does not seem particularly crucial, though finding an algorithm which is significantly faster than the natural algorithm based on forcing might be interesting.

Although the skew partition presents fundamental problems with regard to polynomial recognition algorithms, 2-joins do not present a problem in and of themselves. First, in contrast with skew partition, if a graph is decomposed using 2-joins only, the number of subproblems created is polynomial. [350] shows that there is a polynomial time algorithm which guarantees that G has an odd hole if and only if one of the blocks created using 2-joins has an odd hole. In addition, 2-joins can be chosen in such a way that if a graph has no skew partition, no new skew partitions are created as the graph is decomposed using 2-joins [130]. Therefore, if you have a graph which is indecomposable with respect to skew partitions, there are algorithms which can test whether the graph is perfect. Viewed another way, if your recursive skew partitioning creates a polynomial number of subproblems, the graph can be tested for perfection in polynomial time, so that skew partitions seem to be the key bottleneck for recognizing perfect graphs.

An M-join is defined by a pair of disjoint vertex sets A, B such that allother vertices can be partitioned into nonempty sets of vertices of the following types; neighbors of A but not B, neighbors of B but not A, neighbors of A and B, nonneighbors of A and B. In addition, every vertex in A must have both a neighbor and a nonneighbor in B, and every vertex in B must have both a neighbor and a nonneighbor in A. Note that an M-join is a special type of homogeneous pair, or 2-module; the name comes from the fact that edges connecting A,B with the rest of the graph form the shape of the letter M. [110] showed that no minimal imperfect graph can have a homogeneous pair of any type. Algorithms for finding M-joins and testing perfection while decomposing using M-joins follow from the same algorithms for 2-joins. An algorithm specicifically designed for finding homogeneous pairs is given in [181].

11.8 Exercises

1) Give an O(n+m) algorithm which finds the smallest module containing both x and y, for any x and y.

2) We describe an algorithm to find a source of a prime comparability graph in section 11.1.4. Start with a candidate set C - v for any vertex v. Select any vertex w of V - C; if w splits C, remove nonneighbors of w from C. We stop when C contains a single vertex c. Show that if G is a prime comparability graph, c can be a source of a transitive orientation. Hint: consider the first vertex removed from C

which has an edge to c.

3) Is the following property maintained during the transitive orientation algorithm? Between every pair of sets, all edges between the sets are oriented in the same direction.

4) Can there be graphs (obviously not comparability graphs) for which the orientation algorithm described in this chapter can produce a cycle?

5) Show that the maximal proper submodules of a prime module are unique.

6) A partial order is represented by a set of k linear extensions if $x \rightarrow y$ if and only if x comes before y in all the linear extensions; this is essentially the parameter dimension. Show how to verify that a directed acyclic graph given as input is represented properly by a set of $k = 2$ lists represent G in $O(n+m)$ time. This problem is open for larger values of k.

7) Give an $O(m\log n)$ algorithm which takes a substitution decomposition tree and a set of m edges from a graph on n vertices, and assigns each edge (x,y) to the least common ancestor of x and y in the tree.

8) It is well known that a two dimensional partial order which is prime with respect to the substitution decomposition has a single two dimensional realizer. Show that in general, a prime partial order can have exponentially many realizers of minimum size.

9) Suppose that you are given a substitution decomposition tree, but are not told whether internal nodes correspond to prime nodes, series nodes, or parallel nodes. Show how to determine the type of module for each internal node in $O(n+m)$ time. You might find it useful to show that the total number of edges in all quotient graphs for modules is $O(m)$.

10) Suppose that you are given 1 list L of vertices in a partial order, and you want to determine whether this is part of a realizer of size 2. Design a linear time algorithm to solve this problem. It is an open question to determine whether L is part of a realizer of size 3 in polynomial time.

11) Show that there are $O(m^{1/2})$ vertices in a graph which have degree larger than $m^{1/2}$.

12*) Give an algorithm which finds the transitive reduction of a two dimensional partial order in $O(n^2)$ time.

13) Given 2 lists, show how to construct the adjacency list for the corresponding 2 dimensional partial order in $O(m+n)$ time. Note: this is open for dimension $d > 2$.

14) The vertex/edge inclusion poset of G is a height 1 partial order, with an element for each vertex of G, an element for each edge of G, and the relationship $x < e$ if and only if vertex x is an endpoint of edge e in G. A fascinating theorem

due to Schnyder [428] shows that a graph G is planar iff the vertex/edge inclusion poset has dimension at most 3. Show that graphs with vertex/edge inclusion posets having dimension 2 are forests.

15) Show that the partitioning algorithm B of section 11.1.2 which chooses vertices from a smaller set to split other sets can have $\Omega(m\log n)$ time complexity, i.e. that there are graphs for which this actually takes time proportional to $m\log n$.

16) Show that algorithm B of section 11.1.2 is also an $O(n^2)$ algorithm.

17) Show that if G is not a comparability graph, it must be possible for the partitioning algorithm to create two orientations of G which are not the reverse of each other.

18) Describe a simple linear time method for obtaining the substitution decomposition tree for the complement of G from the substitution decomposition tree for G.

19a) A module M is split by an orientation if there are vertices m_1, m_2 in M and v not in M such that v has an edge to m_1, and m_2 has an edge to v. Show that no module of the decomposition tree is split by a transitive orientation.

19b) Describe an efficient algorithm for counting the number of transitive orientations of a comparability graph.

20) Show that in every transitive orientation of a comparability graph which is prime with respect to substitution decomposition, there are at least two source vertices.

21) In a weak order, vertices are assigned weights so that $x < y$ if and only if $w(x) < w(y)$. Weak order may be thought of as representing a grading scheme, which is not a total order of students since students with the same grade are incomparable. The weak order dimension is the minimum number of weak orders which give P as their intersection. Show that for $k > 1$, a partial order P has weak order dimension k if and only if it has dimension k.

22) There have been a number of interesting ways to form a mathematical model of human notions of ranking preferences for objects. If humans ranked objects using a weak order, indifference between objects would be transitive, and it has long been realized that this is not the case. One explanation for this fact could be that people rank objects along several scales; for instance, you might say a student is better than another student if and only if they were both more creative and harder working. This explanation is modeled nicely by the parameter dimension; each scale is still a total order.

A fundamentally different explanation arises if we assume that there is a single scale, but there is a level of uncertainty; we are indifferent between objects if they rank sufficiently close on the scale. This is modeled by a semiorder; vertices are assigned weights as in a weak order above, but there is a threshold t such that

$x < y$ if and only if $w(x) + t < w(y)$.

22a) Show that a bipartite semiorder has dimension at most 3. In fact, the dimension of an arbitrary semiorder is at most 3, but this is a bit tricky to prove [401].

22b) Show that a partial order P is a semiorder if and only if the complement of P is a unit interval graph.

22c) Show that there are semiorders which are not the containment graphs of paths in a tree. You may use the fact that there are semiorders which have dimension 3.

23) Show that every permutation graph is completely decomposable with respect to the generalized join.

24) Let Bits(G) be the minimum number of bits used to represent a graph G. Show that if G can be split into G_1, G_2 by the generalized join decomposition, Bits(G) is at most Bits(G_1) + Bits(G_2) + O($n$log$n$).

25) Consider a graph decomposition formed by successively removing vertices of degree at most k. Show that graphs which are completely decomposable using this decomposition neither contain nor are contained in the class of graphs with substitution decomposition diameter at most k.

26) Prove or disprove: the class of graphs with substitution decomposition diameter k is hereditary.

27) Show that for any i, you can determine whether there is a decomposition of G into A, B, $|A| > 0$, $|B| > i$ such that each vertex in A has at most i neighbors in B in polynomial time.

28) Show that trees are a recursively decomposable class.

29a) Show that there are classes of graphs which are recursively decomposable, and do not have bounded substitution decomposition diameter

29b) Show that there are classes of graphs which have bounded substitution decomposition diameter, but are not recursively decomposable.

30) Show that for any set of rules with respect to k terminal decomposition, there is a polynomial algorithm to decide whether an input graph G is indecomposable.

31a) Show that if G has NLC-width k, G has clique-width at most $2k$.

31b) Show that if G has clique-width k, G has NLC-width at most k.

32) Show that graphs with NLC-width 1 are exactly equal to cographs. Note that cographs are also exactly equal to graphs with clique-width 2.

33) Show that every graph with clique-width k has a balanced k-module.

34*) Show that we can find a balanced k-module (if one exists) in a graph in polynomial time for any fixed k. Hint: try all possible choices for assigning a single vertex to each subset, and decide whether these choices can be completed to get a balanced k-module. It may help to construct a forcing graph, to describe the fact that placing vertex x in the k-module may imply that y must also be placed in the k-module.

35a) Show that $O(n^2)$ clique separators are found if you recursively decompose a graph using an arbitrary clique separator decomposition.

35b) Show that $O(n)$ clique separators are found if you recursively decompose a graph using an arbitrary clique separator decomposition. Hint: show that if G is not a clique, then the number of separators found is at most $2n$ - 2ω - 1, where ω is the size of the largest clique in G. This observation seems to have been made first by Hoàng and Maffray.

36) This problem gives the fundamental idea of Whitesides' algorithm for finding a clique separator in a graph.

Let S be a chordless cycle of length at least 4. Repeatedly find a connected component C of V - S; let R be the neighbors of C in S. If R is a clique, R is a clique cutset, since it separates S from C. Otherwise, take any nonadjacent x, y in R; let P be a chordless path from x to y which uses only vertices in C. Add vertices of P to S. Repeat until S contains all vertices.

36a) Show that if K is a clique cutset, then S is always in a single connected component of the graph induced by V - K.

36b) Show that this gives an $O(nm)$ algorithm for finding a clique cutset in G.

37) Show that finding a clique cutset in a graph is at least as hard as determining whether a graph has a simplicial vertex.

38) Show that a nonadjacent pair v_i,v_j is part of the fill caused by an ordering if and only if there is a path from v_i to v_j which uses only vertices which precede v_i and v_j in the ordering.

39) Show that no odd hole or antihole can contain vertices from A, B, C, and D from a skew partition.

40) Given that every weakly chordal graph has a two-pair, explain why every weakly chordal graph has a 2-join or skew partition.

41) Show that every comparability graph which is not bipartite has a skew partition. I am indebted to Guenter Sterten for pointing out the simple proof of this fact.

42) The following example is due to Ross McConnell. Consider the graph G formed by taking a path $= x_1..x_{2k+1}$, and adding an edge each odd vertex to the next odd vertex.

42a) Show that G is a comparability graph, and therefore perfect.

42b) Show that in every skew partition of G, $|A| + |B| \leq 3$.

42c) Recursive skew partitioning would then consider the graphs G-A, G-B, G-C, G-D. Since A and B are small, at least 2 of these subproblems are of size at least n-2. The recurrence $\mathrm{T}(n) \geq 2\mathrm{T}(n\text{-}2)$ has an exponential solution. Does this imply that the number of subproblems encountered during recursive skew partitioning is exponential? Explain your answer.

42d) Chinh Hoàng gave the following example, which met certain objections to the example above. Take a clique, and replace every vertex by an independent pair of vertices. Show that recursive decomposition using skew partition as described above creates an exponential number of different labeled graphs on this example.

CHAPTER 12

Elimination Schemes

We have already seen one important elimination scheme which characterized a graph class, in the correspondence between chordal graphs and graphs with perfect elimination schemes in chapter 3. Elimination schemes and characterizations are often desirable. Typically, we have some local property (for perfect elimination schemes, N(v) is a clique) which allows us to remove a vertex simply, and this can sometimes be sufficient to recognize a graph class defined by a global property (G has no induced cycles of length greater than 3). Elimination schemes also are useful in optimization problems.

Here, we deal with classes which are defined by elimination schemes. In several of these cases, the scheme itself gives rise to a representation which uses o(n^2) space, but is not generalized implicit.

12.1 Distance Hereditary Graphs

Distance hereditary graphs are usually defined in terms of a global property; G is distance hereditary if and only if for every pair of vertices x, y, all induced paths between x and y have the same length. If we think in terms of elimination schemes, we want to find vertices which can be removed via local testing which will not affect whether the graph is distance hereditary.

Vertices u, v are called twins if N(v) - u = N(u) - v; twins correspond to modules of size 2. It is not hard to see that removing one vertex from a pair of twins cannot affect whether a graph is distance hereditary. It is also not difficult to see that if w is a degree 1 vertex (also called a pendant vertex), G is distance hereditary if and only if G - w is distance hereditary (exercise 12.1). These simple observations give rise to an elimination scheme characterization of distance hereditary graphs. A sequence v_1 .. v_n is a pruning sequence if each v_i is either a degree 1 vertex in the graph induced by $v_{i+1}..v_n$, or a twin of some vertex in the graph induced by $v_{i+1}..v_n$.

Theorem 12.1 [27]*: G is distance hereditary if and only if G has a pruning sequence.*

The characterization above shows that the number of distance hereditary graphs is $2^{O(nlogn)}$, since we can reconstruct G from the pruning sequence if we know the neighbor of each degree 1 vertex in the sequence, the twin of each twinned vertex, and whether the twins are adjacent. This does not give us an immediate implicit representation, however; if we know that u was created as the twin of x and v was created as the twin of y, we do not know whether u and v are adjacent. However, it is not hard to see from this characterization that every distance hereditary graph is a circle graph, since twins and degree 1 vertices can easily be added to a circle model (exercise 12.2). Pruning sequences are also useful for optimization problems

on distance hereditary graphs. We will present a linear time recognition algorithm for distance hereditary graphs below. Proofs of the correctness of the algorithm can be found in [246].

First, let us note that pruning sequences for cographs can be found in linear time; in fact, cographs are exactly equal to graphs which have pruning sequences without the pendant vertex operation (exercise 12.8). As discussed in chapter 8, we can construct a cotree T for any cograph in linear time. Any pair of leaves of the cotree with a common parent p in T are twins; all leaf siblings can be compressed to a single vertex, which replaces p in T; this process is repeated until a single vertex remains.

Perform a breadth first search from any vertex v; vertices are now divided into levels. The pruning sequence is composed by working backwards from the last level; let j be the current last level. Each connected component of level j is a P_4-free module [246], and is compressed to a single vertex as in the previous paragraph. Order the remaining vertices w_1, w_2,...,w_k of level j by increasing degree in the current graph, and remove any degree 1 vertices. The neighbors of w_1 form a P_4-free module M; M is shrunk to a single vertex, after which w is removed as a degree 1 vertex. After this, the neighbors of w_2 are a P_4-free module, and the same process is repeated for w_2,...,w_k.

Every vertex is in at most one call to a cotree construction algorithm on an induced subgraph of G, so the total time taken on cotree construction is $O(m+n)$. Other steps clearly can be performed in linear time.

The relationship between join decomposition and distance hereditary graphs (distance hereditary graphs are exactly those graphs which are completely decomposable by the join decomposition) is mentioned in chapter 11. This led Dahlhaus [143] to modify the algorithm above to find the join decomposition of an undirected graph in $O(n+m)$ time, using a modular decomposition algorithm as a subroutine.

12.2 Strongly Chordal Graphs

For our purposes, strongly chordal graphs, which were first studied in [184], are defined by the following elimination scheme. A vertex v is simple if $N(v)$ is a clique, and the neighborhoods of vertices in $N(v)$ are ordered by set containment.

A simple elimination ordering is an ordering of the vertices $v_1..v_n$ so that each v_i is simple in the subgraph induced by $v_{i+1}..v_n$. G is strongly chordal if and only if G has a simple elimination ordering.

It is easy to see that the graph of Figure 12.1 is not strongly chordal, since it has no simple vertex. It can be shown that G is strongly chordal if and only if G is chordal and has no induced 'sun' of size at least 3, where a sun is a clique of size k connected cyclically to an independent set of size k as in figure 12.1 [183]. Note that I use the term sun to refer to what is called a complete sun in the original paper; see exercise 12.4. Some papers on the subject use the term trampoline rather than sun.

First, let us note that the definition of strongly chordal graphs immediately gives a representation with $o(n^2)$ bits. Let v be a simple vertex. If $|N(v)|$ is at most $n^{1/2}$, we can certainly eliminate v, using at most $n^{1/2}\log n$ bits. Suppose that $|N(v)|$ is at least $n^{1/2}$. Then we can eliminate v and all neighbors of v by ordering $N(v)$ by degree, and maintaining for each vertex w in V - $N(v)$ the number of neighbors of w which are adjacent to w; this information can be stored with $O(n\log n)$ bits.

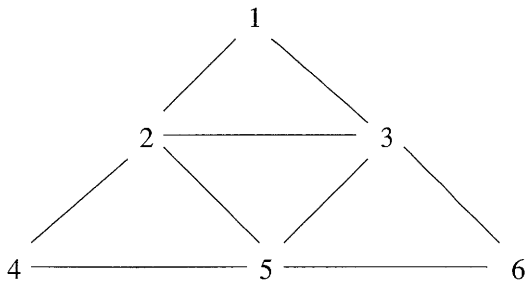


Figure 12.1 The 3-sun: Chordal, but not Strongly Chordal

If we repeatedly remove low degree simple vertices or high degree simple vertices and their neighborhoods, we obtain a set of $O(n^{1.5} \log n)$ bits from which G can be reconstructed. Thus, the problem of finding a generalized implicit representation for the class is potentially interesting.

In fact, the number of graphs in the class is much smaller than $2^{n^{1.5} log n}$, as can be seen from the close correspondence between strongly chordal graphs and chordal bipartite graphs studied in chapter 9. This correspondence can be shown via direct reduction, or by the following characterization of strongly chordal graphs. The augmented adjacency matrix of the graph is the matrix formed by replacing 0s along diagonal entries with 1s.

Theorem 12.2 [183] *G is strongly chordal if and only if the vertices can be ordered so that the augmented adjacency matrix is Γ-free.*

The theorem above shows that the number of strongly chordal graphs is $2^{O(nlog^2 n)}$, since we showed that the number of Γ-free matrices is $2^{\Theta(nlog^2 n)}$ in chapter 9. However, it may not be clear that this bound is tight, since not every Γ-free matrix is the augmented adjacency matrix of a strongly chordal graph; for example, augmented adjacency matrices are always symmetric. Often, however, chordal bipartite graphs and strongly chordal graphs can be viewed as different ways of looking at an underlying Γ-free matrix. To show that this bound is tight, we will show how to transform these classes into each other. The two lemmas below give a proof that the number of strongly chordal graphs is $2^{\Theta(nlog^2 n)}$.

Let $B^+(G)$ be the the graph B(G) formed by making two copies v_1, v_2 of each vertex v, and making x_1 and y_2 adjacent if $x = y$ or x and y are adjacent in G. Note that this differs from a construction of a graph $B(G)$ elsewhere in this book, in that the two copies of v are adjacent in B^+ but nonadjacent in B.

Lemma 12.3 [66] *G is strongly chordal if and only if $B^+(G)$ is chordal bipartite.*

Proof If G is strongly chordal, the Γ-free ordering of the augmented adjacency matrix of G immediately gives a Γ-free ordering of $B^+(G)$ (the matrices are the same), so $B^+(G)$ is chordal bipartite. If G is not strongly chordal, G is either not chordal or G has a k-sun. In exercise 12.5, it is shown that either case implies a chordless cycle of length at least 6 in $B^+(G)$, so G is strongly chordal if and only if $B^+(G)$ is chordal bipartite. □

Lemma 12.4 [141] *Consider a bipartite graph $G = (X,Y,E)$. Let G' be the graph formed by making Y into a clique. Then G is chordal bipartite if and only if G' is strongly chordal.*

Proof Suppose that G is chordal bipartite, and assume that we have a Γ-free ordering of the bipartite adjacency matrix of G. The ordering of Y as in the Γ-free ordering, followed by any ordering of X, must be a simple elimination order for G'. Otherwise, let x_1 be the first vertex of X which is not simple in this scheme. There must be neighbors $y_1 < y_2$ of x_1 such that $N(y_1)$ is not a subset of $N(y_2)$ in the graph induced by the remaining vertices; let x_2 be in $N(y_1)$ - $N(y_2)$. Then rows x_1, x_2 and columns y_1, y_2 would induce a Γ in the ordering of G. If G is not chordal bipartite, then G' is not strongly chordal, since no vertex from a chordless cycle of length at least 6 in G can be simple. □

I should note that the simplicity of the reductions between the two classes of graphs led me to believe that it was not useful to consider the recognition problem separately on the two classes, but there is one important issue which this viewpoint made me overlook. I have looked only at recognition of chordal bipartite graphs; I feel that there should be a linear time algorithm for this, since (as stated in chapter 9) current algorithms do more than is necessary, finding a doubly lexical ordering of an arbitrary matrix without using any special properties which must hold if the matrix has a Γ-free ordering. Therefore, I was very interested when a paper appeared which claimed a linear time algorithm for recognizing strongly chordal graphs. Although the paper had serious, unfixable flaws (see exercise 12.14), it made me realize that the reductions above are not symmetric with respect to cost. The reduction from strongly chordal recognition to chordal bipartite recognition is linear; this implies that any linear algorithm for recognizing chordal bipartite graphs would give a linear algorithm for recognizing strongly chordal graphs. However, the reduction from chordal bipartite recognition to strongly chordal recognition involves adding $\Theta(n^2)$ edges. Thus, it is possible to have a linear time algorithm for strongly chordal graph recognition which does not imply a linear time algorithm for chordal bipartite graph recognition, and I might advise the reader interested in solving an open problem to try the former problem first.

There are optimization problems which behave very differently on the classes of chordal bipartite and strongly chordal graphs. Dominating set and Steiner tree are both NP-complete for chordal bipartite graphs [379], but have simple polynomial algorithms for strongly chordal graphs.

12.3 k-Simplicial Elimination Schemes

I find the following elimination scheme to be a natural extension of perfect elimination schemes for chordal graphs. The class has not been studied previously, and I have decided to include it for the reader despite the fact that it is not closely related to notions of representability of graph classes.

A perfect elimination scheme is a very satisfying characterization of chordal graphs, since it allows testing by local properties (are the neighbors of v a clique) whether the graph has a global property (does G contain a hole of size > 3). When studying graphs without holes of size $> k$ for other values of k, it is natural to extend the definition of a simplicial vertex. Clearly, a vertex which is not in the center of an induced path on k vertices cannot be part of a chordless cycle of

length greater than k. A vertex v is k-simplicial if v is not in the center of a path of length k; simplicial vertices correspond to 3-simplicial vertices. A graph has a k-simplicial elimination scheme if we can successively remove k-simplicial vertices until no vertex remains.

In general, k-simplicially reducible graphs are not equivalent to graphs without holes of size $> k$ (exercise 12.7). Chordal bipartite graphs are equivalent to bipartite graphs with 5-simplicial elimination schemes (exercise 9.3) It is not clear whether this equivalence holds for other classes of interest. Given that the property is local, it may be easier to test for k-simplicially reducible graphs than for graphs without holes of size $> k$, though I know of no work on these classes.

In general, there are too many k-simplicially reducible graphs to have an interesting form of representation, since every split graph is 3-simplicially reducible. For bipartite graphs, these classes may become smaller; this is related to Open Problem 9.4.

12.4 Doubly and Dually Chordal Graphs

Two other classes which come up in chapter 14 are defined by elimination orderings. In order to see why these classes are defined, let us consider a use of a simple elimination ordering. A dominating set is a set D of vertices such that every vertex of G is either a member of D, or adjacent to a member of D. The dominating set problem asks for the smallest dominating set of vertices in the graph. Suppose that you want to solve the dominating set problem, and you are given a simple elimination ordering for G. It is clearly best to choose the maximum degree neighbor of the first vertex to cover this vertex; we continue in a similar fashion stepping through the ordering, and for each undominated vertex we encounter we choose its maximum neighbor in the remaining graph.

For this problem, it is not important that all neighborhoods of neighbors of v are ordered by set containment, as long as we know there is some neighbor w of v such that $N[w]$ contains the neighborhood of all other neighbors of v; such a neighbor is called a maximum neighbor of v. A graph is doubly chordal if there is an elimination scheme such that each vertex removed is simplicial and has a maximum neighbor in the remaining subgraph. A graph is dually chordal if there is an elimination scheme such that each vertex has a maximum neighbor in the remaining subgraph.

Although there are too many graphs in each class to have an efficient representation (exercise 12.12), there are linear time algorithms to construct each of the elimination schemes on the class which allows optimization algorithms to be solved in linear time. In fact, these algorithms are considerably simpler than finding the doubly lexical ordering of the adjacency matrix, which is used to find the elimination ordering of a strongly chordal graph.

The algorithms for constructing maximum neighborhood orderings for dually chordal graphs and perfect maximum neighborhood orderings for doubly chordal graphs are due to Brandstädt, Chepoi, and Dragan [67]; that paper presents correctness proofs for the algorithms as well.

Vertices are placed in 3 categories; unnumbered, numbered, and marked; note that marked vertices are also considered to be numbered. Initially, all vertices are unnumbered. Repeatedly select and mark a numbered (but not marked) vertex u adjacent to the maximum possible number of numbered vertices; at the first

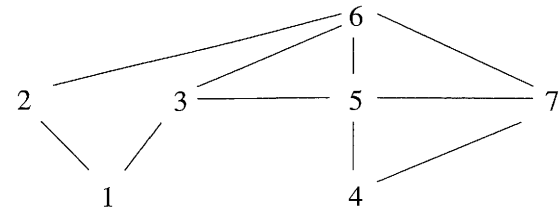


Figure 12.2 Sample Graph for Maximum Neighborhood Ordering

step, any vertex is chosen as u. Assign all unnumbered neighbors of u maximum remaining numbers (in any order); u will be the maximum neighbor of these vertices (this is denoted by $\text{mn}(x) = u$).

Consider the graph of Figure 12.2; note that this graph is not dually chordal. Initially, we may choose vertex 7 to go last; at this point we place 4,6,5 in any order before 7, and make 7 the maximum neighbor of these vertices. Vertex 5 must be the next vertex marked, since it is adjacent to three numbered vertices while 4 and 6 are adjacent to only two. Vertex 3 is numbered and placed before 4,6,5; $\text{mn}(3) = 5$. Vertex 6 is marked next; vertex 2 is numbered at this point, with $\text{mn}(2) = 6$. Vertex 3 or 4 can be chosen at the next step; 4 numbers no new vertices while 3 numbers 2. The final ordering may be as follows: 1(3) 2(6) 3(5) 5(7) 6(7) 4(7) 7, with maximum neighbors in parentheses. 3 is not a maximum neighbor of 1, since 3 is nonadjacent to 2; thus, this graph is not dually chordal.

One can use standard data structures to implement this algorithm to run in linear time (exercise 12.9). Of course, this algorithm will construct an ordering even if the graph is not dually chordal, as shown above, so we must verify that the result is a maximum neighborhood ordering. The paper [67] tests this indirectly, verifying that the input is dually chordal by testing a corresponding hypergraph property.

For our verification algorithm, let us modify the ordering found above slightly; vertices which are numbered at the same time are reordered on the basis of when they were marked in the algorithm; vertices marked earlier are placed later in the ordering. Thus, in the example above, we change the ordering from 1 2 3 5 6 4 7 to 1 2 3 4 5 6 7. This reordering can be done in linear time (exercise 12.10). Suppose that we want to test whether the modified algorithm above produces a maximum neighborhood ordering. We say a vertex x has been verified if we have checked that $\text{N}(x)$ was a subset of the neighborhood of some other vertex. Step through the elimination scheme in order; let v be the next vertex encountered. If v has a neighbor which comes after $\text{mn}(v)$, the ordering is not a maximum neighborhood ordering. For each neighbor w of v in the remaining subgraph, if w is unverified and w is not $\text{mn}(v)$, verify that $\text{N}(w) \subseteq \text{N}(\text{mn}(v))$ in the remaining subgraph. After all neighbors are verified, we verify that $\text{N}(v) \subseteq \text{N}(\text{mn}(v))$ and remove v from the graph. Testing for neighborhood containment can be done in $\text{O}(\text{N}(w))$ time, either by using the technique from exercise 1.2 or by adding neighbors of w to a list which is checked by $\text{mn}(v)$ immediately before $\text{mn}(v)$ is removed. Since each w is checked once for neighborhood containment, the total time taken is $\text{O}(n+m)$.

If the elimination scheme is correct, we will never find a violation in the verification phase. If v has a neighbor z which comes after $\text{mn}(v)$, then $\text{mn}(z)$ must

be adjacent to z but not $\text{mn}(v)$; this was the point of the reordering step, since if $\text{mn}(v)$ and z share the same mn value, z will be chosen as the maximum neighbor of v. Thus, the algorithm will answer that this is a maximum neighborhood ordering.

Suppose that our elimination scheme is not correct, i.e. for some v there is a neighbor w after v such that $\text{N}(w)$ is not a subset of $\text{N}(\text{mn}(v))$ in the subgraph induced on the remaining vertices. Let v be the first vertex of the scheme for which $\text{mn}(v)$ is not a maximum neighbor, and let w be the last vertex of $\text{N}(v)$ in the scheme such that $\text{N}(w)$ is not a subset of $\text{N}(\text{mn}(v))$. If w comes after $\text{mn}(v)$, or w is unverified, we find the violation at this step. Suppose that $\text{N}(w)$ was verified to be a subset of $\text{N}(v_i)$; since $\text{mn}(x)$ is always the last vertex of the scheme adjacent to x, v_i comes after w in the ordering. The neighborhood of w in the remaining subgraph is a subset of $\text{N}(v_i)$, so v_i must also be a neighbor of v and not satisfy $\text{N}(v_i) \subseteq \text{N}(\text{mn}(v))$, contradicting the assumption that w was the last neighbor of v in the ordering causing a violation.

If we want to create an ordering for a doubly chordal graph which is both a perfect elimination scheme and a maximum neighborhood ordering, we make a simple modification to the maximum neighborhood ordering above. When u is chosen as a maximum neighbor, instead of placing neighbors of u in any order before u, we repeatedly choose neighbors of u which are adjacent to the maximum number of vertices already in the output list, placing each in the output list after all remaining vertices, until all neighbors of u are exhausted. For example, in the graph of Figure 12.2 above with 7 chosen to go last, we could not choose 4 and then 6; if 4 is chosen, 5 must be the next vertex chosen in the output list. G is doubly chordal if and only if G is both chordal and dually chordal [67], so the linear algorithms for recognizing these classes allow us to determine whether the above ordering is correct in linear time.

Since neither dually chordal nor doubly chordal graphs are a hereditary class (the addition of a universal vertex makes every graph dually chordal), it is natural to consider what class we get if we want to require that every induced subgraph has a maximum neighborhood ordering. This class of hereditary dually chordal graphs was shown to be exactly equal to the strongly chordal graphs [69].

12.5 Exercises

1a) Show that G is distance hereditary iff G - w is distance hereditary, for any degree 1 vertex w.

1b) Show that for any vertex u which has a twin, G is distance hereditary if and only if G - u is distance hereditary.

2) Show that every distance hereditary graph is a circle graph.

3) Bandelt and Mulder [27] show that G is distance hereditary if and only if G does not contain the house (the complement of P_5), the gem (P_4 plus a universal vertex), the domino (C_6 plus a chord creating two cycles of length 4), and a chordless cycle of length greater than 4 as an induced subgraph. Show that each of these graphs is a minimal non-distance hereditary graph.

4) In the original definition of a sun, which we will call an incomplete sun, a more general structure was allowed. The original definition specified a partition of the vertex set into two sets S_1, S_2 of the same cardinality $k, k > 2$. S_1 was still an independent set, and S_1 must be cyclically connected to S_2 (i.e. $s_{1,i}$ is adjacent to $s_{2,i}$ and $s_{2,i+1}$), as in our definition, but S_2 was not required to be a clique. Show that every chordal graph which contains an incomplete sun also contains a sun as an induced subgraph.

5a) Show that chordless cycles in G map to chordless cycles of length at least 6 in $B^+(G)$.

5b) Show that k-suns map to chordless cycles of length $2k$ in $B^+(G)$.

6) G is completely separable by the join decomposition if every induced subgraph with at least four vertices is decomposable. Show that every distance hereditary graph is completely separable with respect to join decomposition.

7*) Show that there are graphs without any holes of size $> k$ which are not k-simplicially reducible.

8) Show that G is a cograph if and only if there is a pruning sequence which only removes twins from G; i.e., we do not ever use degree 1 removal in our elimination step.

9) Show that the elimination schemes for dually and doubly chordal graphs can be found in $O(n+m)$ time. Note that the key point is that one can find the vertex adjacent to the maximum number of previously chosen vertices efficiently.

10) The maximum neighborhood ordering algorithms discussed in section 12.4 number a set of vertices arbitrarily when the first neighbor of these vertices is marked. For the purposes of verification, it is desirable to have an ordering such that if two vertices x, y were numbered at the same time, x is given a higher number than y if x is marked before y is marked. Show that this reordering can be done in linear time.

11) Maximum cardinality search (MCS) and lexicographic breadth first search (LBFS) are two standard methods for constructing a perfect elimination scheme. LBFS is defined in chapter 13. Each step of MCS chooses a vertex to go last in the elimination scheme; any vertex adjacent to the maximum number of already selected vertices can be chosen at a step. Show that neither LBFS nor MCS necessarily gives a maximum neighborhood ordering of a doubly chordal or dually chordal graph.

12) Show that there are $2^{\Theta(n^2)}$ doubly chordal graphs. Since every doubly chordal graph is dually chordal, the same result clearly holds for dually chordal graphs.

13) A graph is brittle if there is an elimination scheme such that each vertex removed is either not in the middle of a P_4 in the remaining graph, or not at the end of a P_4 in the remaining graph. Show that the number of brittle graphs is $2^{\Omega(n^2)}$, and that this remains true if we are only allowed to eliminate vertices which are

not at the end of a P_4.

14) A supposed linear time algorithm which appeared in draft form for recognizing strongly chordal graphs worked as follows. First, verify that G is chordal. If G is chordal, construct the intersection graph H of its maximal cliques; two maximal cliques are adjacent in H if they share some common vertex. The algorithm then tested a condition on H. Question a indicates that the time bound of such an algorithm is unlikely to be linear, since finding H is as hard as determining whether a split graph has diameter 2, and no linear time algorithm is known for this. Question b makes this all rather academic (but isn't everyone reading this academic anyway?), since no condition on H can determine whether or not G is strongly chordal.

14a) Show that computing H is at least as hard as computing the diameter of a split graph.

14b) Show that there are chordal graphs G_1, G_2 which have the same clique intersection graph H, such that G_1 is strongly chordal and G_2 is not strongly chordal.

15) At one point, I thought I had a linear time algorithm for recognizing strongly chordal graphs, by means of a clever implementation of a different algorithm which I received in draft form. Unfortunately, this algorithm proved to be wrong as well. The algorithm works as follows. A minimal separator is a set S of vertices such that for some u and v there is no path from u to v in the graph induced by V - S, but there is a path from u to v in every V - S + s for every $s \in S$.

For every pair of vertices x, y of a chordal graph G, let commonsep(x, y) be the number of minimal separators which contain both x and y. Perform lexicographic breadth first search on G as in chapter 13, with the following modification. Let X be the vertices which have already been placed in the output list. Let sepweight(y) be the sum of commonsep(x, y) over all $x \in X$. Instead of choosing a vertex from the last group arbitrarily, select a vertex from the last group which has the largest value sepweight(y). The draft claimed that G is strongly chordal if and only if the ordering produced is a simple elimination order.

Show that the algorithm is incorrect; specifically, you should show that the algorithm can produce an ordering of a strongly chordal graph which is not a simple elimination order. Hint: it is possible to take a split graph, and add vertices which do not affect whether the input is strongly chordal, but force the algorithm to reorder vertices to form a violation of the simple order property.

16a) Design a linear time algorithm for finding a maximum matching in a strongly chordal graph, if a simple elimination scheme is given as part of the input. To the best of my knowledge, there is no known linear time matching algorithm for strongly chordal graphs if the input is given in adjacency list form.

16b) Show that finding a perfect matching in a chordal graph is at least as hard as finding a perfect matching in a bipartite graph.

CHAPTER 13

Recognition Algorithms

This chapter covers efficient recognition algorithms for a number of classes of intersection graphs. Particular attention will be given to algorithms which have been considered too complex to cover in other texts. We will try to present the ideas of the algorithms, without covering details which might be necessary to implement the recognition algorithms completely within the best known time bounds. We also present a collection of relevant classes which are NP-complete to recognize, although the NP-completeness proofs are not given here.

For a variety of reasons, recognition of some classes of graphs are covered in other chapters. Specifically, recognition algorithms for permutation graphs, comparability graphs, and two dimensional partial orders are presented in chapter 11, recognition of chordal comparability graphs and an alternative recognition algorithm for weakly chordal graphs appear in chapter 5, algorithms for claw-free graphs, triangle-free graphs, cographs, and graphs with fixed Dilworth number are treated in chapter 8, chordal bipartite graphs in chapter 9, distance hereditary graphs, strongly chordal graphs, doubly and dually chordal graphs in chapter 12.

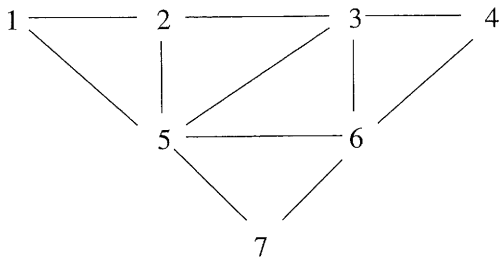
13.1 Chordal Graphs

Recognition of chordal graphs is covered nicely in Golumbic [223]. We review this recognition algorithm briefly, since it is simple, and variants will be used throughout the book. The definition of chordal graphs implies a polynomial time recognition algorithm, since for any k we can determine whether G has a chordless cycle of length at least k in $O(n^k)$ time. This would give an $O(n^4)$ algorithm for chordal graph recognition; we will review algorithms which reduce the time bound to $O(n+m)$.

Efficient chordal graph recognition algorithms rely on the following characterization of chordal graphs. A vertex v is simplicial if the neighbors of v form a clique. Note that a simplicial vertex can clearly not be part of any chordless cycle of length greater than 3. Therefore, if we want to determine whether a graph is chordal, we can remove any simplicial vertex we find; removal of some simplicial vertices may cause other vertices to become simplicial. It can be shown that if the graph is chordal, then this process of successively removing simplicial vertices will remove all vertices from the graph. This simple idea for testing chordality is formalized as follows.

A perfect elimination scheme $v_1 v_2 v_3 ... v_n$ is an ordering of the vertex set if and only if for all i, v_i is simplicial in the graph induced by v_{i+1} through v_n.

Theorem 13.1 [160] *G is chordal if and only if there is a perfect elimination scheme for G.*



(1234567) choose 7
(1234)(56)(7) choose 6
(12)(34)(5)(6)(7) choose 5
(12)(4)(3)(5)(6)(7) choose 3
(1)(2)(4)(3)(5)(6)(7)

Figure 13.1 Lexicographic Breadth First Search

See [223] for a proof of this theorem. It is not difficult (though not completely trivial) to use this theorem directly reduce the time needed for chordal graph recognition to $O(n^3)$, but we can do much better by making more sophisticated use of the theorem. The algorithm is like the comparability graph recognition algorithm discussed in chapter 11 in being usefully divided into two phases; a construction phase which creates an ordering which is a perfect elimination scheme if and only if G is chordal, and a verification phase which checks whether the ordering which was constructed is in fact a perfect elimination scheme.

The construction phase is relatively simple, and quite a number of algorithms have been designed to construct a perfect elimination scheme for a chordal graph in linear time. We choose to describe lexicographic breadth first search (LBFS) for constructing the ordering, since this is also used in other algorithms of the book. The perfect elimination scheme is generated in reverse order.

Vertices are partitioned into sets; initially, all vertices are in the same set. At each step, an arbitrary vertex x from the last set is chosen. We remove x from the graph, and place it in the output list. Note that x comes before all previously selected vertices in the elimination scheme, and after all vertices which remain in the graph. All sets are subdivided into neighbors and nonneighbors of x, with neighbors of x in S placed immediately after the set of nonneighbors of x in S. An example is shown in Figure 13.1. Standard data structures can be used to make this algorithm run in $O(n+m)$ time; for details, see [223, 414].

We now need to verify that the list generated is a perfect elimination scheme for G, since the algorithm clearly produces an ordering whether or not G is chordal. Step through the perfect elimination scheme from first vertex to last; let v be the next vertex in the order. Find the first neighbor of v which comes after v in the order, and call this vertex w. Verify that w has edges to all neighbors of v which come after w in the ordering; if not, answer that this is not a perfect elimination scheme, since v is not simplicial. If the verification step works for all vertices, G is chordal and this is a perfect elimination scheme. Since the verification takes time proportional to $|N(v)|$ for each vertex v, this runs in $O(n+m)$ time. The correctness proof for this algorithm is left for exercise 13.1.

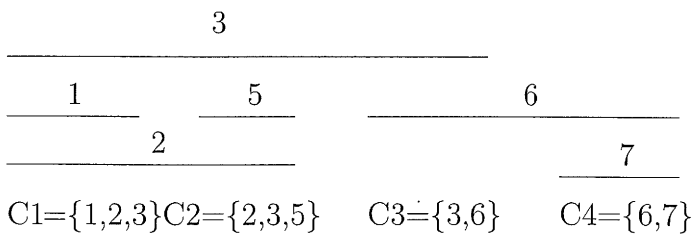


Figure 13.2 Building an Interval Model From a Consecutive Clique Arrangement

13.2 Interval Graphs

We will describe the standard algorithm for interval graph recognition due to Booth and Lueker [56] using PQ-trees. We note that this algorithm has been simplified in several ways, by using modified PQ-trees [318] and by dispensing with PQ-trees and complex data structures altogether. However, we will need to introduce PQ-trees in any case, since they will be used below as part of an efficient recognition algorithm for circular-arc graphs. We then describe an algorithm due to Corneil, Olariu, and Stewart [127], which recognizes interval graphs without using PQ-trees.

Booth and Lueker (as I understand it originally independently, though it was published as a joint paper) designed an algorithm for recognizing interval graphs in $O(n+m)$ time as follows. As a first step, verify that G is chordal; if not, halt and answer that G is not an interval graph. If G is chordal, enumerate all maximal cliques of G; a discussion of this step is deferred to chapter 15, where maximal cliques are used as part of a chordal graph representation.

The number of maximal cliques in a chordal graph is at most n; this number can be exponential in general graphs. The total maximal clique size of G, defined by adding k to the sum for a maximal clique of size k, is $O(n+m)$. This allows processing vertex/clique incidence relationships in linear time. The interval representation is constructed by creating an ordering of the maximal cliques such that maximal cliques containing each vertex appear consecutively; the interval for v is created by simply spanning these cliques, as shown in Figure 13.2.

For this reason, Booth and Lueker created a data structure called a PQ-tree, which is designed to enforce constraints of the type 'this set of objects must occur consecutively'. We will not give details of PQ-tree construction here; they are discussed in a number of texts such as [223] and [179]; the original paper [56] is also quite well written. A PQ-tree has two types of nodes, called P and Q nodes. Children of a P node are allowed to rotate arbitrarily, and children of a Q node can only appear in a fixed order or in the reverse of the order. Booth and Lueker's algorithm adds one constraint at a time, and makes modifications to the tree forced by the constraint; an arrangement obeys all constraints if and only if it can be obtained from the tree by rotating children of P nodes and/or reversing children of Q nodes. Thus, the PQ-tree of Figure 13.3 corresponds to such arrangements as 3 5 4 1 9 8 7 6 2, 2 6 7 8 9 1 3 4 5, 5 3 4 1 6 7 8 9 2.

I should note that PQ-trees used for interval graph recognition can also be viewed as a special case of modular decomposition trees. Prime graphs have unique

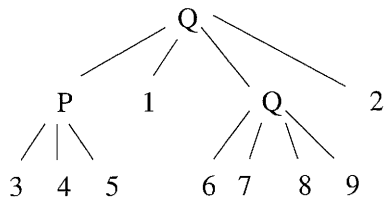


Figure 13.3 A PQ-tree

consecutive maximal clique arrangements [265], and correspond to PQ-trees consisting of a root Q node and all maximal cliques as children of the root. The ordering of children cliques can be obtained from the unique transitive orientation of the graph; see [268] for details. Children of a prime module in the modular decomposition tree also correspond to children in the PQ-tree. Series modules in an interval graph are essentially trivial, with only one child submodule containing more than one vertex. Neighbors of a parallel module must form a clique, making these correspond naturally to P nodes in the decomposition tree. The paper [268] shows that there are simpler modular decomposition and transitive orientation algorithms for chordal graphs than for general graphs; these can be used to recognize interval graphs as well.

In general, we can assume that we are given a collection of sets S_1 ... S_i; each S_i constrains a particular set of items to occur consecutively. Let $m = |S_1| + |S_2| + ... + |S_i|$. Let n be the number of elements. The overall bound for enforcing a set of these constraints using PQ-trees is $O(n+m)$. The standard notion involves forcing all sets to be consecutive in a linear order, and from the discussion above gives a linear time algorithm for interval graph recognition. However, the data structure can also be used to force sets to be consecutive within a circular order; this will be used in the next section as part of a recognition algorithm for circular-arc graphs.

We now describe an interval graph recognition algorithm from [127]. The word 'simple' has many different meanings when applied to an algorithm, some of which are illustrated nicely in this case. I believe that the algorithm below would be quite simple to implement, and I include it for that reason. However, the correctness proof is extraordinarily long (much longer than the proof of the algorithm based on PQ-trees), and will certainly not be presented here; the interested reader can consult the paper. Erdös imagined a divine book of proofs, containing the most beautiful proof of each theorem. Examples like this show us that the God of graph theory may be different from the God of graph algorithms; which would you include in a justification of the statement that interval graphs can be recognized in linear time?

I should mention that I use the terms earlier and later in the search differently from [127]; I say vertices occur later in the lexicographic search ordering if they were chosen for removal earlier in the algorithm; this usage comes from the idea that lexicographic breadth first search also produces a perfect elimination scheme, and vertices selected earlier occur later in the scheme.

The algorithm makes five passes of lexicographic breadth first search using certain tie-breaking rules to choose the appropriate vertex from the last set; the result is a valid ordering of the right endpoints in an interval model if G is an interval graph. To construct the intervals, we simply create an interval from each

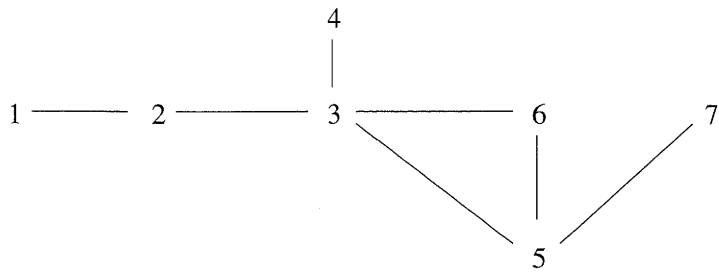


Figure 13.4 Sample Graph for Interval Graph Recognition

vertex v to the first neighbor of v in the ordering, and verify that each intermediate endpoint is a neighbor of v.

The algorithm uses different tie-breaking rules in the five passes of lexicographic breadth first search. Although most of the rules are simple to understand, the tie-breaking rule for the final pass is somewhat tricky. Therefore, we will describe the tie-breaking rule for the fifth pass at two levels. The first description is easy to understand, but hard to see how to implement in linear time, while the second description gives a linear time implementation of the search procedure.

Pass 1: Perform standard LBFS.

Pass 2: Perform LBFS, breaking ties by always choosing the vertex which occurs earliest in the ordering produced by Pass 1.

Pass 3: Perform LBFS, breaking ties by always choosing the vertex which occurs earliest in the ordering produced by pass 2.

Pass 4: Perform LBFS, breaking ties by always choosing the vertex which occurs earliest in the ordering produced by pass 3.

Pass 5: Perform LBFS, breaking ties as follows. We always choose either the vertex a which occurs earliest in Pass 3 from the set, or the vertex b which occurs earliest in Pass 4 from the set. We choose b unless one of the following two conditions hold:

1) b is adjacent to some vertex which is in an earlier set of Pass 5 while a is adjacent to no such vertex.
2) Neither a nor b is adjacent to a vertex from an earlier set of Pass 5, b has a neighbor in its own set which is adjacent to a vertex in an earlier set of Pass 5, while a has no such neighbor.

Consider the graph of Figure 13.4. The first pass, a standard LBFS, may produce the order 1 7 2 4 5 6 3. The second pass produces 7 6 5 4 3 2 1, and the third produces 1 2 4 3 6 5 7. Pass 4 produces 7 5 6 4 3 2 1. Pass 5 works as follows; our initial set is (1,2,3,4,5,6,7) our two candidate are 7 and 1; the tie-breaking rule chooses 7 since there are no other sets. This gives us (1,2,3,4,6)(5) 7, and 5 must be chosen. We now have (1,2,4)(3,6) 5 7. The two candidates are $b = 6$, $a = 3$. In this case, a has a neighbor in an earlier set (2 and 4) while b does not, so 6 is chosen next. Our next step involving a choice comes after selecting 6 and 3, when

the configuration is (1)(2,4) 3 6 5 7. We set $a = 2$, $b = 4$; again, a has a neighbor in an earlier set while b does not, so our order is 1 2 4 3 6 5 7.

It is easy to see how to implement the tie-breaking rules for passes 2, 3, and 4 so that lexicographic breadth first search still runs in linear time. Sort all adjacency lists on the basis of order of vertices in the previous pass (see exercise 1.1), and the standard method of dividing sets into neighbors and nonneighbors will guarantee that the first vertex in the set will always be the earliest vertex from the previous pass. The same method is used to identify the vertices a and b in Pass 5; we can keep two copies of the list, one in which vertices of a set are ordered according to position in Pass 3, and in the other vertices are ordered by their position in Pass 4. It is harder to see how to choose between a and b in Pass 5 in linear time. Therefore, we present an alternative description of Pass 5, which is harder to visualize but simpler to implement in linear time.

Pass 5: Create lists A(v) (B(v)), where these are linked lists of neighbors of v which come after v in the orderings of pass 3 (pass 4), are adjacent to some vertex which comes before v in the ordering of pass 3 (pass 4), and have not yet been chosen in Pass 5. Perform LBFS, breaking ties as follows. Let a be the earliest vertex from the set in Pass 3, while b is the earliest vertex from the set in Pass 4. To choose between a and b, use the following rules in order until one succeeds.

1) If a comes after some neighbor of a in pass 3, choose b
2) If b comes after some neighbor of b in pass 4, choose a
3) If B(b) is empty or A(a) is nonempty, choose b
4) Let y be an arbitrary element of B(b). If the earliest neighbor of y in Pass 3 is a, then choose b; otherwise, choose a.

It is not hard to see that this version of Pass 5 can be implemented to run in linear time. Calculating initial sets A and B for each vertex is easy; first calculate for every vertex w the earliest neighbor of w in the pass, and it is easy to check whether a neighbor w of v belongs on A(v) in constant time. When a vertex x is chosen in Pass 5, standard methods allow us to delete x from all A and B lists in O(N(x)) time; other steps are also easy to implement.

Therefore, interval graphs can be recognized in linear time, using standard data structures.

A number of graph classes are defined by whether the graph or its complement is a chordal and/or comparability graph; for example, interval graphs are equal to chordal co-comparability graphs. For some of these classes, it is useful to know whether algorithms are efficient in the complement as well. It is relatively easy to recognize co-chordal graphs in linear time; the construction and verification of a perfect elimination scheme in the complement can be done using edges of G (exercise 13.2). Co-interval graphs can also be recognized in linear time, though this result is less obvious.

An interval order is a partial order such that each element corresponds to an interval, with $x < y$ if the interval for x is completely before the interval for y. It is not hard to see that G is a co-interval graph if and only if G has a transitive orientation which is an interval order; in fact, G is a co-interval graph if and only if every transitive orientation of G is an interval order. A partial order is an interval order if and only if the neighborhoods of vertices are totally ordered by set containment [392] (exercise 13.31).

In chapter 11, we mention that a transitive orientation can be found in linear time; we can thus test whether G is a co-interval graph in linear time by whether neighborhoods in the transitive orientation are ordered by set containment. Parts of the transitive orientation algorithm can be simplified for the special case of co-interval graphs using the techniques of [268], which give a simpler linear time algorithm for finding the substitution decomposition and transitive orientation of a chordal graph. In particular, it is easy to derive the substitution decomposition tree for the complement of G from the substitution decomposition tree of G; see exercise 11.18.

13.3 Circular-Arc Graph Recognition

Recognition of circular-arc graphs was a longstanding open problem, first posed in 1964 [245]. The first polynomial algorithm for circular-arc graph recognition is due to Tucker in 1980 [465]. This algorithm has a reputation for being difficult to understand. In part, this is because the algorithm is complex, but the reputation of the algorithm has been exaggerated by a number of issues. Most importantly, Tucker himself talks about the need for simplification in the paper.[1] Some of the proofs are quite complex, even in portions where the algorithm is not, but it is hard for a reader to separate these issues since they are presented together in the paper. Since the time it was written, efficient algorithms for a number of other problems have led to modifications of Tucker's algorithm which are conceptually simpler, and improve the running time. The algorithm described here is due to Eschen and Spinrad [177] though large parts of it are due directly to Tucker. In the interest of simplicity, an $O(n^3)$ algorithm will be presented here, with some indications of what is necessary to reduce the time bounds to $O(n^2)$ as presented in [177]. Recently, a linear time algorithm for recognizing circular-arc graphs has been developed; see [365].

Before going into circular-arc graph recognition, let us briefly discuss why we cannot use PQ-trees to solve the recognition problem for circular-arc graphs simply. For interval graphs, we enforced constraints for each maximal clique of G, using the fact that the total size of all constraints is $O(n+m)$. The clique structure of a circular-arc graph is much more complex than that of an interval graph; in fact, circular-arc graphs may have an exponential number of maximal cliques (exercise 13.10). Fundamentally, this is because cliques in circular-arc graphs do not necessarily share a common point on the circle (formally, arcs on the circle do not have the 'Helly property'); the reader can see this easily by laying out a triangle so that there is no point in common. The tempting assumption that cliques of a circular-arc graph meet at a point has led to quite a number of incorrect submissions to technical journals; if it were true, many algorithms on the class would be much simpler.

The recognition algorithm has much of the structure of a partitioning algorithm, as discussed in chapter 11. The algorithm first finds an approximate layout of arcs

[1] Tucker's statements from the paper include 'The real difficulty in our algorithm is not with speed but its length' and 'As mentioned earlier, simplification rather than greater speed is what is really needed, since the constant in $O(n^3)$ is likely to be horrendous'. The author disagrees slightly with the latter statement; I do not believe the constant in the running time is such a problem, but it is very difficult for someone who has not studied the paper very carefully to produce a correct implementation of the algorithm.

Figure 13.5 Overlap vs. Circle Covering

on the circle, and uses partitioning to get a better approximation of the final layout. Finally, a reduction is made to permutation graph recognition.

A pair of intersecting arcs in a circular-arc graph can intersect in a variety of ways. We use comparisons between $N[x]$ and $N[y]$ to decide which type of arc intersection to enforce in our model as follows. Assume that G is preprocessed to eliminate neighbors with the same closed neighborhood; this preprocessing step can be performed in $O(n+m)$ time. We will force the arc corresponding to x to contain the arc corresponding to y if $N([y])$ is a subset of $N([x])$. If neither neighborhood contains the other, we have two possibilities; the arcs may have an overlap relationship, as in Figure 13.5-a, or may cover the circle, as in Figure 13.5-b. We will force a circle-covering relationship between x and y if and only if every nonneighbor of x is contained in y. In exercise 13.11, it is shown that if G is a circular-arc graph, there is a model where all relationships are of the type we require. These relationships can easily be computed in $O(n^3)$ time. A statement in Tucker's paper that this might be the key bottleneck step of the recognition algorithm inspired the design of our $O(n^2)$ recognition algorithm.

If the graph has clique cover number 2 (case I as presented by Tucker [465]) we will solve the problem completely by reduction to permutation graph recognition, as we will see later. We note that the clique cover 2 case has been called the difficult portion of Tucker's original algorithm [489], and is one of the major reasons Tucker's algorithm is viewed as being very complex. By contrast, we will solve the clique cover 2 case very easily. To test whether G has clique cover 2, we see whether the complement is bipartite. The two cases will be discussed in separate subsections.

13.3.1 Circular-Arc Graphs with Clique Cover Number Greater than 2. We now discuss Tucker's case II, in which G has clique cover number greater than 2. Since the complement of G is nonbipartite, we either find a triangle in the complement or a longer odd hole; one of these can easily be found in linear time. The recognition algorithm is then divided into three stages. In stage 1, an initial set of vertices is placed on the circle, dividing the circle into regions. In stage 2, other vertices have their endpoints placed in the appropriate pair of regions (for certain special cases, both endpoints will be in a single region). Partitioning is used in stage 3 to get the final refinement of the layout.

We will not present the case in which we find a longer odd hole in this book. It is not difficult to show that there is a unique method for placing the odd hole itself

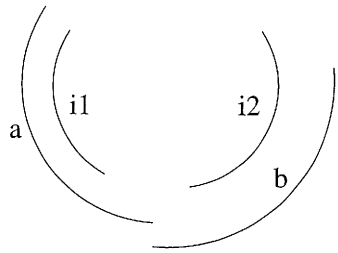


Figure 13.6 Constraints Imposed by Single Arcs are Insufficient

on the circle (stage 1). We then need to place other vertices with respect to the odd hole (stage 2). Placing other vertices with respect to the odd hole is similar to placement in the case when we find a triangle; for a complete description of the placement procedure in this case, see [177, 175]. The procedure for stage 3, finding the exact placement of endpoints, is the same whether we find a triangle or an odd hole.

Stage 1: Positioning Vertices from the Independent Set

Suppose that we find a triangle in the complement of G; this corresponds to an independent set of size 3 in G. We extend this to a maximal independent set I_1. We then slightly modify the independent set to get an independent set I which has the following properties: 1) there is no vertex v in I such that N[w] is a proper subset of N[v] for some w 2) We cannot get a bigger independent set by replacing some v in I with a pair of neighbors of v. Details of this modification step are the subject of exercise 13.12.

The technique for finding a layout for the maximal independent set I was given by Tucker, though the description here is substantially different. The only data structure needed for solving this efficiently is a PQ-tree. Recall that a PQ-tree takes a set of items, and enforces constraints (either on the line or the circle) of the type 'these objects must occur consecutively'; each new constraint takes O(n) time. For details of an amortized linear time analysis in the case of interval graph recognition, see [56].

The idea for this step comes from finding sets of constraints which obviously must be enforced in a layout, and showing that this set of constraints are sufficient. Showing the sufficiency of these constraints is a difficult proof, and is given in Tucker [465]. However, we should not confuse the difficulty of the proof with the difficulty of the algorithm; the algorithm for this step is easy to understand.

One obvious set of constraints is that for each vertex v, the neighbors of v in I must occur consecutively on the circle. This is not quite enough to force a correct ordering on I, as shown in Figure 13.6. Clearly, i1 and i2 must be consecutive in the example, but there is no vertex adjacent to both of these. Thus, we add the following constraints, which it is easy to see are necessary. For every pair of adjacent vertices v and w, the neighbors of v and w in I must occur consecutively on the circle. Tucker shows that these simple necessary conditions are sufficient to enforce a correct ordering on the set I.

There are clearly at most n^2 such orderings to enforce; since PQ-trees enforce a constraint in O(n) time, stage 1 can be performed simply in O(n^3) time. In fact, it is possible to select a set of O(n) constraints to enforce (selecting the overlapping

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

Figure 13.7 Adjacency Matrix for Circular-Arc Recognition Example

arcs which extend as little as possible on each side of an arc is sufficient); finding these constraints in $O(n^2)$ time is discussed in [175].

In the graph of Figure 13.7, we might find an initial maximal independent set 3,4,10; this is modified by replacing 10 with 1,2 in the independent set I. We then use relationships between vertices and pairs of vertices to I, to restrict the set of orderings of I. For example, vertex 6 forces 3 and 4 to be consecutive in the circular ordering of I, and the adjacent pair 5,9 force their joint neighborhood (1 and 4) to be consecutive in the ordering. This results in a circular ordering 1,2,3,4 for the independent set I.

Stage 2: Placing Endpoints in Regions

We now enter stage 2, in which vertices are placed with respect to the placement of arcs in I. For many types of vertices, this placement is simple. Each v must be adjacent to a consecutive sequence of vertices in I; v's arc will properly contain all arcs that are not at the outer ends of this sequence. Suppose that v is adjacent to at least two vertices of I, and is nonadjacent to at least one vertex of I. We can determine whether v has an overlap or containment relationship with the two vertices i_c and i_{cc} on the clockwise and counterclockwise endpoints of the range by determining whether $N(v)$ contains $N(i_c)$ and $N(i_{cc})$. These placements clearly take $O(n)$ time per vertex.

The more complex cases involve placement of vertices which are adjacent to only one vertex of I, or are adjacent to every vertex of I. Consider a vertex v which is adjacent to a single vertex i of I; note that v must be adjacent to some vertex of I since I is a maximal independent set. We chose our independent set such that v's arc is not contained in i's arc. If arc v contains arc i, placement of v is easy. Suppose that v and i have an overlap relationship. One endpoint of v will be in the region inside of arc i. We cannot tell just by looking at relationships between v and vertices of I whether the other endpoint of v should be in the region between arc i-1 and i, or in the region between i and i+1. A similar difficult case arises for a vertex v which is adjacent to all vertices in I, and has an overlap relationship with exactly one vertex i; one endpoint is within i, and one either between i and i-1 or between i and i+1.

In the example graph, vertex 6 must start inside interval 4, contain interval 3, and stop before interval 2; vertex 7 must start inside interval 1, and contain intervals 2 and 3, stopping before interval 4; vertex 10 contains intervals 1 and 2,

stopping before intervals 3 and 4. The intervals which are harder to place are 5 and 11 (adjacent only to 4 from I), 9 (adjacent only to 1), and 8 (adjacent to every vertex of I, and contains all except for 4).

Let i be a member of I. We will give a procedure for placing the endpoints of vertices adjacent only to i from I, and endpoints of vertices which are adjacent to all of I and have an overlap relationship only with i. The only relevant vertices are those which can have endpoints within i and/or in one of the regions next to i. Our goal is to turn the placement problem on these vertices into a clique cover two problem.

One clique is formed by the set of vertices S adjacent only to i; there cannot be two nonadjacent vertices in this set, since we found a maximal independent set which could not be augmented by replacing a single vertex with two. However, the remaining vertices do not necessarily form a clique; vertices which approach i from the counterclockwise direction may be nonadjacent to vertices which approach i from the clockwise direction. For the placement process, we treat all vertices of $V - S$ as adjacent, but we add extra vertices to make sure that vertices approach S from the correct direction. Think of i on the bottom of the circle; we add vertices b,c,d, which can be thought of as clockwise, above, and counterclockwise from i respectively. Make a 4-cycle i, b, c, d; add adjacencies from S to b and d, from c to all vertices not in $S \cup i$ and from b and d to all vertices which approach i from the counterclockwise and clockwise direction respectively. The new graph has clique cover number two, and any placement of this graph can be used as a placement of endpoints in G. We solve the clique cover two problem using a reduction to two dimensional partial order recognition which will be given later.

Every vertex appears in at most four of the graphs used in the placement of endpoints near i, and each such problem will be solved in $O(n^2)$ time, yielding an $O(n^2)$ algorithm for the placement phase.

To continue with our example graph, consider placement of endpoints around interval 4, i.e. the vertices 5,8,11. We create a graph G_4 including vertices 5, 8, 11, the vertices with possible endpoints inside interval 4 or in an adjoining region (vertices 6,7,9,10), and a 4-cycle 4,b, c, d. We make 6,7,9,10 a clique, 9 and 10 adjacent to b and c, 6 and 7 adjacent to c and d. Vertices 5, 8, and 11 are made adjacent to b and d; 8 is also made adjacent to c. We find a realization of G_4 as a circular-arc graph; this can be found using the algorithm of the next section. Although we discuss how to solve this problem in general in the next section, the reader can see that vertex 5 must extend past 4 on the b side (due to adjacency to 9 and 10), while 11 must extend past 4 on the d side (due to adjacency to 7); vertex 8, which is nonadjacent only to 11, must extend past 4 on the b side, wrapping around the circle with its second endpoint stopping inside interval d.

Therefore, we place both vertices 5 and 8 as passing through the clockwise endpoint of interval 4, and 11 as passing through the counterclockwise endpoint.

Stage 3: Endpoint Placement within a Region

Stage 3 of the recognition algorithm involves taking our initial placement of arcs, and performing a partitioning procedure as described in chapter 11 until we get no further refinements. Vertices are placed in the same initial set if their endpoints are placed in the same pair of regions at the end of stage 1. Note that this will place endpoints in the same set if they span the circle in opposite directions;

a vertex which goes clockwise from region 4 to region 6 will be placed in the same initial set as a vertex which goes clockwise from region 6 to region 4 of the circle.

When vertices are placed or split into different sets, the relationship between at least one endpoint of each corresponding arc is fixed, and the necessary relationship between the other pair of endpoints can be determined. For example, if the clockwise endpoint of x is placed counterclockwise with respect to the counterclockwise endpoint of y on one region of the circle, then the other endpoint of x must be placed clockwise with respect to the other endpoint of y if and only if x and y are nonadjacent. Thus, whenever sets are split, we can determine and store in a matrix the necessary conditions on the other endpoints so that these can be accessed in constant time.

Suppose that we have more than one set of endpoints in a region. Take each vertex s from some set S; each one divides the region into a subregion clockwise of this endpoint of s, and counterclockwise of this endpoint of s. Vertices from other sets can be placed in the appropriate subregion in constant time, since we have stored relationships to this endpoint of s; each other set is split into a subset which has one endpoint in the region clockwise of S, and a set which has one endpoint in the region counterclockwise of s. Vertices of S are then grouped into subsets, depending on how many endpoints of the vertex lie clockwise in the region.

This procedure continues until all regions have exactly one set. Using the partitioning algorithm from chapter 11, we can use the fact that the relationship between each pair of endpoints is examined at most once to perform this partitioning in $O(n^2)$ time. Any set S with greater than one vertex after this step must have clique cover number two, and can be treated as a separate graph, since relationships between these endpoints and the rest of the graph has been completely determined.

Taking one region of our example, the region between 1 and 4 has 3 sets of endpoints, each set containing 1 endpoint. Set 1 is the clockwise endpoint of vertex 5, set 2 is the counterclockwise endpoint of vertex 9, and set 3 is the counterclockwise endpoint of vertex 10. Vertex 5 is adjacent to both 9 and 10, and thus the endpoint of is placed clockwise from the other endpoints. Since N[10] contains N(9), the endpoints in this region must be placed (reading from 4 to 1) as 10 9 4. In general, this procedure can iterate as sets are split into subsets, which poses no theoretical problems but would require a very large example to demonstrate all possible steps.

13.3.2 Circular-Arc Graphs with Clique Cover Number 2. We are now left with the possibility that the graph has clique cover number 2. We may assume that there is no pair of neighbors with identical closed neighborhoods, since these can use equivalent arcs and can be found easily by a preprocessing step. We reduce this problem to a two dimensional partial order recognition problem as follows.

Let X, Y be any pair of cliques which covers the vertex set of G. Create a directed graph G' as follows. Vertices of G' correspond to vertices of G. For 2 vertices x_1, x_2 in X, add an edge from x_1 to x_2 if and only if the neighborhood of x_1 is contained in the neighborhood of x_2. For a pair of vertices x in X, y in Y, add an edge from x to y if and only if x and y are nonadjacent in G, and an edge from y to x if and only if x and y cover the circle. For a pair of vertices y_1, y_2 in Y, add an edge from y_1 to y_2 if and only if the neighborhood of y_1 contains the neighborhood of y_2. G is a circular-arc graph if and only if G' is a two dimensional partial order.

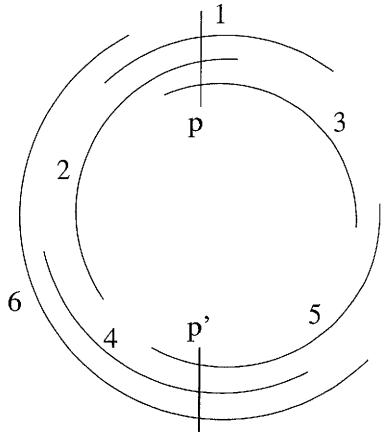


Figure 13.8 Sample Model for Reduction to Dimension 2 Posets

The reduction above is not difficult to understand if viewed in the following way. Tucker [465] showed that for any clique cover 2 circular-arc graph, there is a model in which some pair of points on the circle such that every arc passes through one of these points. The theorem is necessary because in general, cliques of a circular-arc graph do not have to meet at a point. Tucker's theorem can be modified easily to show that for any pair of cliques X, Y which partition the vertex set, there is a model in which all arcs corresponding to vertices in X pass through a point p of the circle, and all arcs corresponding to vertices in Y pass through a point p'.

If we split the circle at points p and p', we get two orderings of the vertex set, since each arc has an endpoint on both semicircles. We interpret this as a representation of a two dimensional partial order by 'reading' each semicircle from p to p'. Thus, for example, if two vertices from X have an overlap relationship, one endpoint will come first in each list, and the vertices are nonadjacent in the two dimensional partial order. If arc x contains arc x', then x' will be encountered before x in both lists; on the other hand, if y in Y contains y' then the endpoints of y' will be encountered first on both semicircles. In the graph of Figure 13.8, reading clockwise from p to p' we get the order 2 1 5 3 6 4, and reading counterclockwise from p to p' gives 3 6 1 4 2 5.

The reduction can easily be performed in $O(n^3)$ time. In order to reduce the time complexity, we need to be able to find the arc relationships (independent, overlap, containment, or circle-cover) between all pairs of neighbors in $O(n^2)$ time. The key step involves turning the problem into neighborhood containment on an associated chordal bipartite graph, and using techniques developed in chapter 9; details are contained in [177, 175], with the key step being the subject of exercise 13.18.

Together, these techniques give an algorithm for recognizing circular-arc graphs. As far as I know, Tucker's algorithm has never been written up as a program, and was generally viewed as extraordinarily complex. Hopefully, this section has made the algorithm seem less intimidating rather than more so. I feel that the algorithm is not so difficult to understand conceptually, and would not be difficult to program if you are satisfied with an $O(n^3)$ implementation, but I may be too close to the problem to view this objectively.

I will close with some remarks on the linear time circular-arc recognition algorithm of [365]. The algorithm first computes types of arc overlaps using a reduction to chordal bipartite graphs as discussed in exercise 13.18, using a closer analysis to argue that this actually can be done in linear time. Once types of overlaps are decided, the algorithm "flips" vertices which cover a part of the circle (flipping all neighbors of a minimal arc would do this, though the paper chooses a slightly different set of arcs to flip), by letting these have the same endpoints but changing the clockwise/counterclockwise orientation of the arc, turning the circular-arc graph into an interval graph. Note that it is necessary to know types of overlap in order to do this; if two adjacent arcs overlap, they will still overlap after a flip, while if arc x contains arc y then x and y will not overlap after x is flipped. This would seem to turn circular-arc graph recognition into interval graph recognition, but there is an extra level of complexity. In order to be able to reverse the process and obtain a circular-arc model from the interval model on the flipped graph, the intervals must have the correct containment and overlap relationships; one cannot always allow interval x to contain adjacent interval y if $N(x)$ contains $N(y)$, since this containment might lead to an error in the circular-arc model after intervals are flipped back to their original position. Therefore, an algorithm is developed which constructs interval models with pre-specified types of adjacency between neighbors. The algorithm is based more on the linear time transitive orientation algorithm of [367], using the relationship between interval models and transitive orientation of the complement, than on the simpler problem of interval graph recognition.

13.4 Restrictions on Intervals and Arcs

A number of natural restrictions of interval and circular-arc graphs have received attention. The most widely studied are requiring intervals or arcs to have the same length, and requiring that no interval or arc properly contains the other. The first restriction defines the unit interval and unit circular-arc graphs, while the second gives rise to proper interval and proper circular-arc graphs. We also deal briefly with Helly circular-arc graphs; in these, cliques are required to share a common point on the circle.

13.4.1 Unit and Proper Interval Graphs. The classes of unit and proper interval graphs are identical [410]. We will give a constructive proof of this equivalence below. Another useful characterization is that G is a proper/unit interval graph if and only if G is an interval graph, and G contains no $K_{1,3}$ [480, 410]; exercise 13.21 uses this characterization to design a linear time recognition algorithm from an arbitrary interval model for G. In chapter 15, we present an algorithm which recognizes unit interval graphs and is simpler than general interval graph recognition.

Suppose that we are given a proper model for an interval graph. It is easy to take the model, and construct a consecutive clique arrangement for G; see exercise 13.9. We use this to construct a unit interval model. Vertices from the leftmost maximal clique are placed with left endpoints in the same order as in the proper model, very close together. Other vertices are added from left to right. Each vertex v is placed just after the rightmost nonneighbor x of v, and any previously placed left endpoints to the right of x. Since left and right endpoints occur in the same order, the neighbors of v are exactly the previously placed vertices which occur

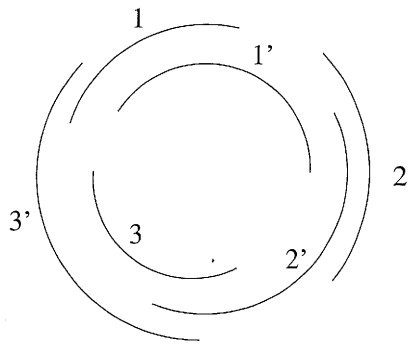


Figure 13.9 A Proper Circular Arc Graph which is not Unit Circular Arc

after x, and if all intervals have unit length then v extends past all these vertices. Some other issues arising from this construction process are raised in chapter 15.

13.4.2 Proper Circular-Arc Graphs. The equivalence between unit and proper models breaks down for circular-arc graphs; it is not hard to see that the graph of Figure 13.9, shown with a proper circular-arc model, is not a unit circular-arc graph (exercise 13.15). We will give an algorithm for constructing a proper circular-arc model for G, if such a model exists, and show how to test whether this can be transformed into a unit circular-arc graph.

Proper circular-arc graphs can be recognized in $O(n+m)$ time, as we will show below. The history of this problem is quite confusing, coming from the fact that the algorithms rely on a sequence of papers which spread over more than 30 years. The net result is that while no authors make a mathematical error, I believe that the published linear time bounds do not stand up, as authors misinterpret statements which appear in previous papers.

The first polynomial time algorithm to solve this problem is due to Tucker in 1971 [466]. The algorithm comes from the following mathematical characterization of proper circular-arc graphs which forms the core of the paper.

Tucker's algorithm divides the problem into two cases, depending on whether the input graph has clique cover number two or not. If G does not have clique cover number two, Tucker shows that G is a proper circular-arc graph if and only if the augmented adjacency matrix of G has the circular 1s property for columns. He invokes an algorithm due to Fulkerson and Gross from 1965 [203] for solving the circular 1s problem. The clique cover 2 case is not covered in full detail in the paper; we will describe an efficient algorithm for handling the clique cover two case below.

Tucker does not give an explicit time bound for the algorithm. A later paper quotes a time bound of $O(n^2)$ for Tucker's algorithm.

[151] gives an algorithm which solves the problem in linear time for sparse graphs, and uses Tucker's algorithm for the case when the graph is sufficiently dense, thus getting a linear time bound, assuming that Tucker's algorithm is $O(n^2)$.

Unfortunately, there is a slight problem with the claim of $O(n^2)$ running time for Tucker's algorithm. This comes from a statement in Fulkerson and Gross which is mathematically correct, but which is quite misleading for more recent readers. They give their time bound as $O(n^2)$, if taking the dot product of 2 n-vectors is

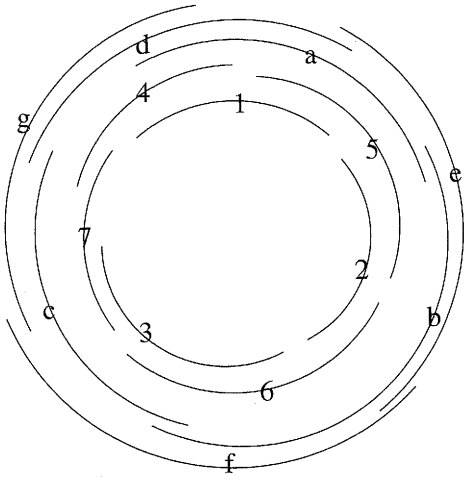


Figure 13.10 More Complex Proper but not Unit Circular-Arc Graph

viewed as a constant time operation. If the running time comes from this statement, we must give the time complexity as $O(n^3)$ rather than $O(n^2)$.

However, the linear time bound can be derived in another way. We can use PQ-trees, which were developed after Tucker's paper was written, to solve the circular 1s property for a matrix in linear time. Thus we only need to find an $O(n^2)$ algorithm for the clique cover two case to get a linear algorithm for proper circular-arc graph recognition.

If G has clique cover number two, we use an easier form of the reduction from general clique cover two circular-arc graph recognition to permutation graph recognition. If G is a proper circular-arc graph, we can assume that all adjacent vertices are overlapping; containment relationships are not allowed in the model, and covering the circle is not necessary (exercise 13.22). This implies that the target permutation graph can be constructed very easily; it is simply the complement of the input graph. It also means that the resulting graph is bipartite. Linear time algorithms for recognizing bipartite permutation graphs are much easier than those for recognizing general permutation graphs; see chapter 9 for a simple algorithm in this case.

13.4.3 Unit and Helly Circular-Arc Graphs. The material from this section will be treated in greater depth in the forthcoming paper [165]. Although no previous paper has explicitly given a polynomial time bound for recognizing unit circular-arc graphs, an efficient algorithm is implicit in a paper of Tucker's [468].

If you are asked to provide an example of a graph which is a proper circular-arc graph but not a unit circular-arc graph, you will almost certainly come up with an example such as Figure 13.9, in which some cycle has the same size as an independent set. It is tempting, but not correct, to conjecture that every proper circular-arc graph is a unit circular-arc graph unless this occurs. A counterexample is given in Figure 13.10.

We can see that this proper circular-arc arrangement cannot be modified to give a unit circular-arc graph. The circularly ordered set $1, 2, 3, 4, 5, 6, 7$ are consecutively nonoverlapping, and each arc in the sequence is the next arc after the

counterclockwise endpoint of the previous arc. If we assume all arc lengths are 1, this sequence of 7 arcs implies that the circle must have circumference at least 7/2. The circular set of vertices a, b, c, d, e, f, g are consecutively adjacent, and wrap around the circle 2 times. This implies that the circumference of the circle is less than 7/2 if every arc has unit length.

More generally, an (n, k) circuit of a proper circular-arc graph is a set of n circularly ordered vertices such that vertices i and $i+1$ are adjacent, arc $i+1$ is clockwise of arc i-1, and the set of arcs wraps around the circle k times. An (m, l) independent set is a set of m circularly ordered vertices such that vertices i and $i+1$ are nonadjacent, arc $i+1$ is the first arc of the set which starts clockwise of the counterclockwise endpoint of arc i, and the arcs wrap around the circle l times.

Tucker [468] was able to show that G is a unit circular-arc graph if and only if an arbitrary proper circular-arc model (without a circle-covering pair) does not contain an (i, j) circuit and a (k, l) independent set such that $i/j \leq k/l$. In fact, Tucker [468] proved stronger results; for example, if a proper circular-arc graph G is not unit circular-arc, there must be both an (i,j)-circuit and an (i,j) independent set. Thus, we will describe a method for searching for an obstruction of this form; this method comes from Tucker's proof of the corresponding theorem.

We look for upper and lower bounds on the circumference in greedy fashion, starting with every arc. Each search can be implemented to run in $O(n\text{log}\log n)$ time, so that the entire algorithm runs in $O(n^2\text{log}\log n)$ time if we start with the proper interval model. Let x be any vertex; we show how to find the lower bound starting with x.

Maintain a circularly ordered list of unused arcs, the number of arcs used thus far, and the number of times you have traversed the circle. Let c be your current arc. Find the next unused arc y which starts after c; replace c with y, and continue as long as some arc is nonadjacent to c. If y is adjacent to x, add 1 to the number of times you have traversed the circle. At each point, calculate the current ratio of number of arcs used to the number of times you have traversed the circle; keep the maximum value of this ratio over all starting arcs x as your lower bound on the circumference. To calculate an upper bound, choose the unused neighbor which extends furthest at each step rather than the next unused nonneighbor.

The time complexity of the search from x is $O(\text{log}\log n)$, if we use the data structure of van Emde Boas [171] which allows for searching for predecessors and successors of elements in the range 1..$O(n)$ time per operation, with insertions and deletions allowed. Therefore, we can recognize unit circular-arc graphs in $O(n^2\text{log}\log n)$ time; I imagine that this bound can be improved to linear time. For issues which arise if we want to construct a unit circular-arc model, see chapter 15.

For some optimization problems, Helly circular-arc graphs are simpler than general circular-arc graphs [212]. Again, no paper has been written on the recognition problem for Helly circular-arc graphs, but these can certainly be recognized in polynomial time. For Helly circular-arc graphs, all cliques meet at a point (by definition), which means that there are $O(n)$ maximal cliques. We can therefore use PQ-trees to enforce a correct circular-ordering on the maximal cliques; cliques containing each vertex v must appear consecutively on the circle. The bottleneck step is enumerating the maximal cliques. This can be done in $O(n^3)$ time for Helly circular-arc graphs, since there are $O(n)$ maximal cliques and all maximal cliques can be generated in $O(n^2C)$ in an arbitrary graph with C maximal cliques [394]. I feel certain that some enterprising reader will improve this algorithm, either by an

improved algorithm for enumerating maximal cliques for this special case or by a direct solution of the problem.

13.5 Trapezoid Graphs and Related Classes

In this section, we present an algorithm which solves a number of closely related recognition problems. The problems solved include trapezoid graph recognition, Ferrers dimension 2 digraph recognition, biorder dimension 2 and interval dimension 2 recognition, 2-chain subgraph cover, and threshold dimension 2 for split graphs; definitions of these are given along with the reductions below. This section is called trapezoid graph recognition, since trapezoid graphs are most directly linked to the other classes treated in this chapter. The problems are all solved by reduction to a problem called 2-chain cover; we then present an algorithm for the 2-chain cover problem.

The history of this problem is remarkably confusing. This comes from the different names and applications of essentially the same problem; authors would know relations between some pairs of graph classes, but not know about relationships to others. Another source of confusion is that two crucial papers by the same author were written on recognizing this class in polynomial time, and since the papers were presented without explicit time bounds other authors cited the simpler (but slower) algorithm rather than the faster algorithm. Improvements were then made on the time bounds in the slower algorithm, and at least three interesting new algorithms were developed which were faster than the algorithm of one paper, but were not in fact as fast as an algorithm implied by the other existing paper. Finally, the algorithm of the first paper was modified to give an algorithm which was both faster and newer than previous algorithms; we will present the modified algorithm here.

13.5.1 Relationships Between Problems. We outline the close relationships between a number of problems which have at times been studied independently. Some of these problems can be thought of as different views of the same underlying problem, and reductions in both directions can be found in [357]. For our purposes, it will be sufficient to reduce these problems to a single problem, called the 2 chain subgraph cover problem, which we will solve efficiently in a later section.

A chain graph is a bipartite graph in which neighborhoods of all vertices from a single color class can be ordered by set containment; in other words, the bipartite graph contains no induced $2K_2$. Chain graphs were used by Yannakakis to show that the partial order dimension problem and the minimum fill-in problem were NP-complete [492, 491]; these were two of the original open problems posed in Garey and Johnson's book [206].

The bipartite complement of a graph is the bipartite graph on the same vertex set such that two vertices from different color classes are adjacent if and only if these vertices are nonadjacent in G. Clearly, the bipartite complement of a chain graph is itself a chain graph.

The 2 chain subgraph cover problem asks whether there is a pair G_1, G_2 of chain subgraphs of a bipartite graph G such that every edge of G is contained in at least one of G_1 or G_2.

The chain dimension of a bipartite graph G is the minimum number of chain graphs which give G as their intersection.

It is easy to see that G can be covered by 2 chain graphs if and only if the bipartite complement of G has chain dimension 2; any graph used in one problem can be replaced by the bipartite complement in the other problem. Some of the problems in this chapter reduce naturally to chain cover 2, while others are more easily viewed as reductions to chain dimension 2.

A Ferrers digraph is a digraph such that for any x, y, either $\mathrm{N}(x)$ is a subset of $\mathrm{N}(y)$ or $\mathrm{N}(y)$ is a subset of $\mathrm{N}(x)$. The Ferrers dimension of a digraph G is the smallest number of Ferrers digraphs on the same vertex set such that G is exactly equal to the intersection of this set of Ferrers digraphs.

There is a natural correspondence between Ferrers graphs and chain graphs, since both correspond to neighborhoods being ordered by set containment. To convert a Ferrers dimension k problem to a chain dimension problem, make a bipartite graph $\mathrm{B}(G)$ by making two copies V and V' of the vertices, and add an edge from v to w' if and only if $v \rightarrow w$ in G. The chain dimension of $\mathrm{B}(G)$ is equal to the Ferrers dimension of G.

The interval dimension of a poset P is the minimum number of interval orders which intersect to give P. The connection to chain graphs comes from the characterization that interval orders are exactly those partial orders such that the neighbors of each vertex are ordered by set containment. Thus, interval dimension on P can be reduced to chain dimension on $\mathrm{B}(P)$ as defined above, since there is a direct correspondence between chain graphs which contain $\mathrm{B}(P)$ and interval orders which contain P.

Trapezoid graphs, also called II graphs, are the intersection graphs of trapezoids defined by intervals defined along two parallel lines; for variants of these, see chapter 3. It is easy to see that the complement of a trapezoid graph must be orientable to be an interval dimension 2 poset. Since interval dimension is a comparability graph invariant [243], G is a trapezoid graph if and only if the complement of G has interval dimension 2. Thus, we can take a transitive orientation algorithm for the complement of G as described in chapter 11, and turn the trapezoid graph recognition problem into a chain cover problem (by way of interval dimension 2) in $\mathrm{O}(n^2)$ time.

We note that for recognizing partial orders with interval dimension 2 and trapezoid graphs, we do not want to have to check that the input is a comparability or co-comparability graph; recall that the transitive orientation problem will find some orientation for a graph even if the graph has no transitive orientation. However, our algorithm for the chain cover two problem also allows us to construct the chain graphs which cover G, which are easily translated into a pair of interval orders which represent P or a trapezoidal representation of G. After these are constructed, we verify in linear time whether the representations are correct; this is not a bottleneck step (exercise 13.25). Thus, our reductions allow recognition of interval dimension 2 posets and trapezoid graphs in $\mathrm{O}(n^2)$ time, and can construct the corresponding models in the same time bound.

Reductions from the bidimension problem and threshold dimension 2 on split graphs to chain dimension can be found in [357].

13.5.2 Cogis' Reduction. Cogis was interested in the Ferrers dimension k problem, both for general values and for the special case $k = 2$. One paper [114] provided a simple reduction from Ferrers dimension k problems to poset dimension k recognition as follows.

For any digraph G, create a poset $P(G)$ as follows; elements of P correspond to vertices of G. Add an edge from x to y whenever $x \rightarrow y$ in G, or $N(x)$ is a subset of $N(y)$ in G.

Theorem 13.2 [114] *The Ferrers dimension of G is equal to the dimension of P(G).*

For the special case of Ferrer's dimension 2, this gives a polynomial time recognition algorithm. The time complexity is dominated by the cost of the reduction, though this was not true when Cogis wrote his paper, since the most efficient poset dimension 2 recognition algorithms had not been developed. The dominant step is computing neighborhood containment, which takes time proportional to matrix multiplication. If a simple algorithm is preferred, there are easy graph theoretic algorithms for all steps which take $O(n^3)$ time.

Unfortunately for the tangled history of the problem, the above reduction was only part of a more general paper proving the polynomial equivalence of Ferrers dimension and partial order dimension; thus, the fact that the special case of dimension 2 is solvable in this way was not emphasized. Cogis also was the author of an earlier paper which recognized digraphs with Ferrers dimension 2 in polynomial time [113], which became widely known as Cogis' algorithm. This algorithm, variants of which also appear in [274, 162], involves setting up an auxiliary graph with vertices corresponding to 0s in the adjacency matrix, with two vertices of the auxiliary graph adjacent if and only if the four corresponding rows and columns induce a permutation of the 2 by 2 identity matrix. The auxiliary graph is bipartite if and only if the original graph has Ferrers dimension 2, leading to a simple $O(n^4)$ recognition algorithm for the class.

13.5.3 Reducing the Time Complexity. We present a different reduction to the partial order dimension problem which leads to an $O(n^2)$ algorithm for the chain cover two problem. Proofs which are not given here can be found in [357].

The first idea behind the new reduction is that we do not need to concern ourselves with the time for computing a general reduction from k-chain cover to partial order dimension k recognition; time complexities are only important for the special case $k = 2$. We then tailor our reduction so that the bottleneck step of neighborhood containment can be performed on a chordal bipartite graph rather than a general bipartite graph; using results from chapter 9, neighborhood containment on a chordal bipartite graph can be performed in $O(n^2)$ time.

Our reduction from chain cover two to two dimensional poset recognition works as follows. Let G be a bipartite graph, and let X and Y be the color classes of G. We first remove all vertices with identical neighborhoods; we then add edges to $P(G)$ as follows. Add an edge from x_1 to x_2, where x_1 and x_2 are in X, if $N(x_1)$ contains $N(x_2)$. Add an edge from y_1 in Y to y_2 in Y if $N(y_1)$ is contained in $N(y_2)$. Add an edge from y in Y to x in X if x and y are nonadjacent. After these are done, we add an edge from x in X to y in Y if all neighbors of y in X are also neighbors of x.

An example of this reduction is given in Figure 13.11. The partial order created in this example is two dimensional; one realizer is 2 4 3 6 1 5, 2 6 1 4 3 5. The corresponding chain graphs are created by selecting only those edges $x \in X, y \in Y$ of G such that x comes before y in the ordering. In the example, the edge (1,4) is

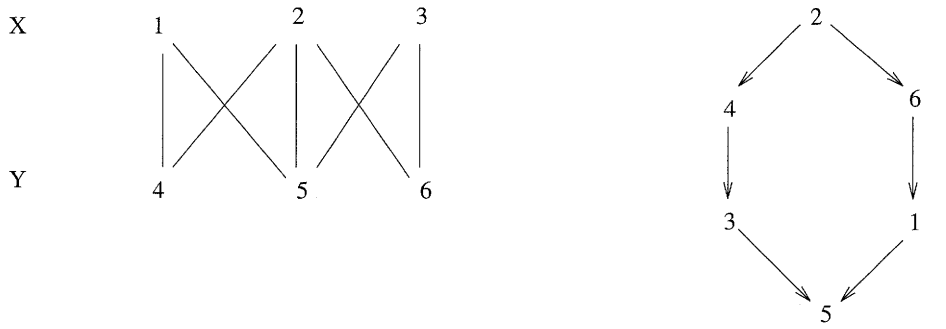


Figure 13.11 Graph and Transitive Reduction of Poset for Chain Cover

deleted to get the first chain graph of the cover; (3,6) is deleted to get the second chain graph.

Theorem 13.3 [357] *The dimension of $P(G)$ is equal to the number of chain subgraphs needed to cover G.*

Although the reduction above holds for arbitrary chain cover number, we are only interested in time complexity of the chain cover two problem. Since two dimensional partial orders can be recognized in linear time, the only step which can take more than $\Theta(n^2)$ time is the reduction itself.

Suppose that the bipartite complement of the input graph G is not chordal bipartite. Then G must have chain cover number at least 3, since putting the bipartite complement of a cycle C_k of length at least 6 through the reduction above produces C_k, and two dimensional partial orders cannot contain C_k for $k > 4$.

Neighborhood containment in G can be derived trivially from neighborhood containment in the bipartite complement. Suppose that we want to test whether G has chain cover number two. We first test in $O(n^2)$ time whether the bipartite complement is chordal bipartite; if it not, we answer that G does not have chain cover number two. Otherwise, we perform the reduction above. Steps involving computing neighborhood containment in G take $O(n^2)$ time, since they can be computed on the bipartite complement. The only other part of the reduction which could be a problem is the last step, in which edges are added from x to y if all neighbors of y in X are also neighbors of X. However, the bipartite graph formed by making a copy of X and Y into one color class, a second copy X' of X as another color class, and adding an edge from v to x' if and only if an edge has been added in the poset from v to x' is also chordal bipartite [357]; this allows the final step of the reduction to be computed in $O(n^2)$ time as well.

13.5.4 Historical Note. The reader may note a similarity between the fast algorithm for clique cover two circular-arc recognition and the algorithm for trapezoid graph recognition. Both use chordal bipartite graphs to avoid neighborhood containment calculation in a reduction to the poset dimension two problem. At one point, I thought that this was a coincidence, illustrating the usefulness of chordal bipartite graphs rather than showing a direct relationship between these problems. However, Trotter and Moore [463] showed that a graph with clique cover number two is a circular-arc graph if and only if the complement has interval dimension

two; this shows a direct relationship between the problems. For this book, I decided not to use the reduction to interval dimension 2 recognition as part of the circular-arc recognition algorithm, so that a reader would be following the same algorithm in this book and in the original paper which provides the details of a faster implementation. The reduction to interval dimension can certainly be used as well.

The paper showing the relationship between circular-arc graphs and interval dimension was written in 1976. Given my longstanding interest in circular-arc graphs and general admiration of Trotter's work, I read that paper at some time; I even cite the paper in [439]. At that time, I would have considered interval dimension two to be a less tractable problem than circular-arc recognition. By the time I became involved in developing recognition algorithms for circular-arc graphs and interval dimension 2 partial orders, I had completely forgotten Trotter and Moore's paper. Thus, I am in no position to criticize other authors who have been unaware of the complete history of this topic!

13.6 Circle Graphs

This section provides an algorithm for recognizing circle graphs, which are the intersection graphs of chords in the circle. Circle graphs have also been called alternance graphs [63] and overlap graphs; the latter comes from a characterization studied in exercise 13.28.

Circle graph recognition received a good deal of attention after being raised as an open problem by Golumbic [223]. The first polynomial algorithm, due to Naji, runs in $O(n^9)$ time, as described by Bouchet [63]; this was successively improved to $O(n^5)$ by Bouchet [63], Gabor, Hsu, and Supowit to $O(n^3)$ [204] and Ma and Spinrad [359, 442] to $O(n^2)$. We will not present the fastest known algorithm here, since the algorithm is somewhat complex. We present a conceptually simpler algorithm due to Bouchet [63], which runs in $O(n^4)$ time when one takes into account improvements in certain algorithms which were used as subroutines by Bouchet.

13.6.1 Join Decomposition. Recognition of circle graphs is closely tied to a form of graph decomposition known as join decomposition. The correspondence between the decomposition and recognition of circle graphs is very similar to the correspondence between substitution decomposition and comparability graph recognition or permutation graph recognition discussed in chapter 11. Indecomposable graphs have a unique representation, and a graph is a circle graph if and only if each component of the decomposition is a circle graph. The relationship between join decomposition and circle graph recognition was discovered independently in [204] and [63].

The join decomposition was originally intended to be a tool for dealing with directed graphs [138, 139], but the most important applications have proved to be on undirected graphs. A split of a graph is a partition of the vertex set into V_1, V_2 with the following properties. Let V_{1io} be the set of vertices in V_1 which are adjacent to at least one vertex in V_2, and let V_{2io} be the set of vertices of V_2 which are adjacent to at least one vertex in V_1.

1) $|V_1|$ and $|V_2|$ are at least 2.
2) Every vertex of V_{1io} must be adjacent to every vertex of V_{2io}.

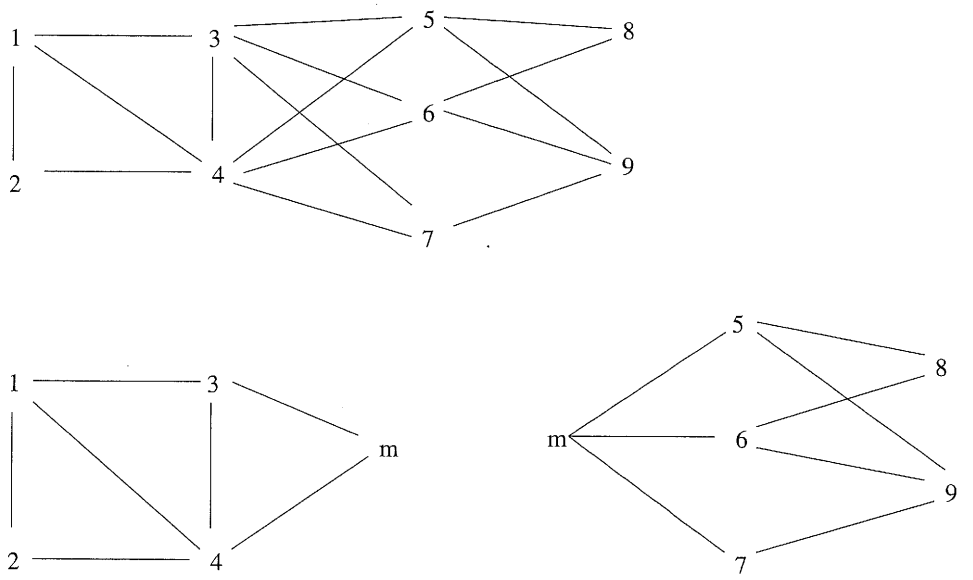


Figure 13.12 A Step of the Join Decomposition

In other words, the set of edges which go between V_1 and V_2 must form a complete bipartite graph. The join decomposition takes any split of G, replaces G by the subgraphs induced by $V_1 \cup m$ and $V_2 \cup m$, where m is a marker vertex adjacent exactly to vertices in V_{1io} and V_{2io}, and decomposes these graphs recursively. Using this marker vertex, it is easy to reconstruct G from the components after any split. An example of a step of this decomposition is given in Figure 13.12. [139] gives uniqueness theorems for the final components of the join decomposition.

It is not hard to see that G is a circle graph if and only if each component of the join decomposition is a circle graph. Each component is an induced subgraph of the original graph, so the representation of the entire circle graph gives a representation of each component. Suppose that there are circle graph representations of two components formed by a split with marker m. These representations will be called A,m,B,m and X,m,Y,m, where A, B, X, and Y are orderings of vertices on the semicircles formed by removing m in the representations. Then the entire graph can be represented by A,X,m,B,Y,m, since orderings of pairs of vertices from a single component are preserved, and chords from different components cross if and only if both chords cross m. Circle graph representations of the two subgraphs of Figure 13.12 and the corresponding representation of the entire graph are shown in Figure 13.13.

[204] and [63] show that if there is no split of a circle graph G, then G has a unique representation (ignoring reversal) on the circle. This will be crucial to the algorithm described below, but the proof will be omitted here in order to focus attention on the recognition algorithm. We will now look at a simple $O(nm)$ algorithm for finding the join decomposition, due to Cunningham [138] as modified in [204]. The fastest algorithm for finding the join decomposition runs in $O(n+m)$ time [143].

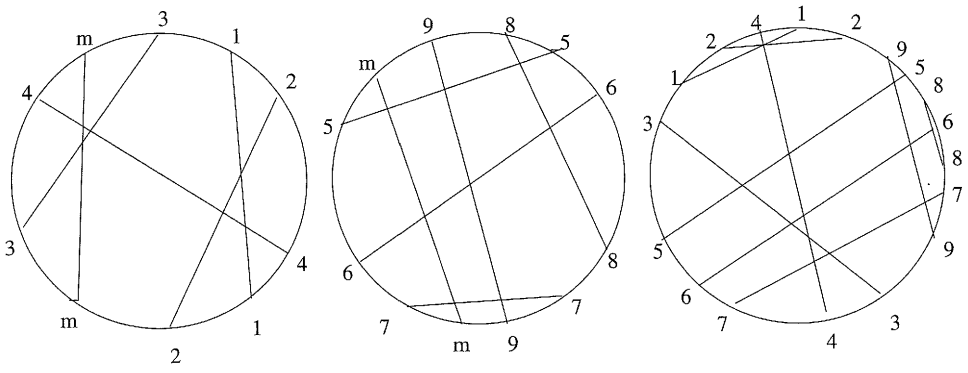


Figure 13.13 Combining Circle Representations for Join Decomposable Graphs

Let (x,y) be any edge of G. We test whether there is a split of G such that this edge crosses the split as follows. Let z be any other vertex of G. We test whether there is a split with x and z on one side, and y on the other, and we test whether there is a split with y and z on one side, and x on the other. Since the procedures are equivalent, we assume below that we are testing for a split in which x and z are in V_1 and y is in V_2.

Place vertices on a queue when they are known to be in V_1; originally, this queue contains vertex z. Vertices of V - x, z are placed in the set $V_{2io?}$ if they are adjacent to x, and in the set $V_{2?}$ if they are nonadjacent to x. While the queue is nonempty, we remove a vertex from the queue; call this vertex v. If v is nonadjacent to y (so that v must be in V_1 - V_{1io}), we remove all neighbors of v from $V_{2io?}$ and $V_{2?}$, and place them on the queue. If v is adjacent to y (v is in V_{1io}), remove neighbors of v from $V_{2?}$ and place them on the queue, and remove nonneighbors of v from $V_{2io?}$ and place them on the queue.

If this procedure stops with at least 2 vertices in $V_{2io?} + V_{2?}$, we have found a split; otherwise, there is no split with x and z in V_1, y in V_2. For each vertex v, the time taken to partition $V_{2?}$ and $V_{2io?}$ is $O(N(v))$, so the total time spent partitioning sets is $O(m)$. There is a possibility of $O(n)$ work in adding vertices to V_1, meaning that we test for a split which is crossed by (x,y) in $O(n+m)$ time.

If implemented as is, this gives quite a high time complexity for finding the join decomposition, but slight changes make for large improvements, as was noted in the original analysis by Cunningham of this algorithm [138]. Our initial analysis could involve testing m edges at a decomposition step, giving $O(m^2)$ time to find a single split. Since there are at most n splits found by the join decomposition (exercise 13.29), this would give an $O(m^2n)$ time bound for computing the entire decomposition.

Let T be any spanning tree for G. If there is a split, then some edge of T crosses the split. This allows us to test whether G has a split in $O(nm)$ time, and gives an $O(n^2m)$ algorithm for join decomposition. Another factor of n can be saved by the fact that if an edge does not cross any split of G, then that edge cannot cross any split of any components of G formed by the join decomposition [138]. This implies that any edge which is tested, and found not to cause a split can be reused in the spanning tree of one of the components, and does not need to be tested again for causing a split.

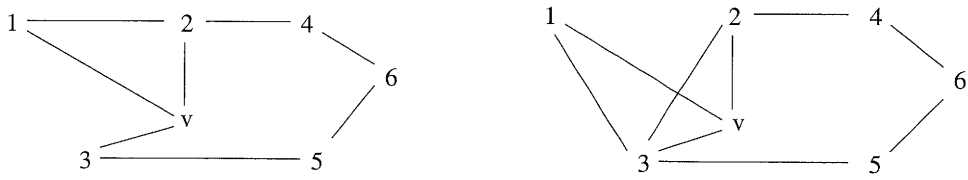


Figure 13.14 Local Complementation at v

Suppose that we reuse all edges which are known not to cause splits in the spanning trees of each component. The total number of edges tested which cause splits is $O(n)$. The final set of spanning trees is a forest on $O(n)$ vertices, and includes all edges which were tested and did not cause a split; this implies that the number of edges tested which do not cause splits is $O(n)$. Since $O(n)$ edges are tested, and each test takes $O(m)$ time, the time to find a join decomposition is $O(nm)$.

13.6.2 Recognizing Circle Graphs. Using the results of the preceding section, we may assume that we are testing a graph which is indecomposable with respect to the join decomposition. The simplest approach we could hope for in recognizing circle graphs is to find a sequence of vertices to add to the model so that each subgraph encountered is prime; the unique representation of prime graphs means that we would only have to consider adding one chord to the existing model. To find this sequence, we might hope that each prime circle graph on k vertices would contain an induced prime circle graph on k-1 vertices; if this was true, our join decomposition algorithm would give a simple method for finding the sequence of vertices.

Unfortunately, there are prime circle graphs on k vertices which contain no prime circle graphs on k-1 vertices (exercise 13.31). Bouchet manages to use this basic approach to get a simple recognition algorithm, but needs to modify the graphs using an operation called local complementation, which is described below.

The local complementation of G at v, written as $G * v$, is the graph formed by making two neighbors of v adjacent if and only if these vertices are nonadjacent in G.

See Figure 13.14 for an example of local complementation at v. Local complementation corresponds naturally to a simple operation on a circle graph representation. If we have a model for G, a model for $G * v$ can be formed by reversing one of the semicircles defined by the two endpoints of v. Thus, G is a circle graph if and only if $G * v$ is a circle graph. In fact, we will use a stronger property: G is a prime circle graph if and only if $G * v$ is a prime circle graph (exercise 13.30).

Bouchet finds a sequence of vertices to add using the following theorem, which we present without proof.

Theorem 13.4 [63] *Suppose that G is a prime circle graph with more than 5 vertices. Then there is a vertex v such that one of the following statements is true.*

1) G - v is prime.

2) $G * v$ - v is prime

3) $G * v * w * v$ - v is prime for every neighbor w of v

Given the theorem above, it is easy to recognize circle graphs in polynomial time. First, G is decomposed with respect to the join decomposition; G is a circle graph if and only if each component is a circle graph, and a representation for the entire graph can be built easily from circle representations of the subgraphs. We then repeatedly find a vertex v using the theorem above, using $O(n)$ calls to a primetesting algorithm for the join decomposition at each step, getting a smaller prime subgraph each time. Since we are interested in finding a smaller prime subgraph, it is not necessary for us to check every pair of vertices v,w for condition 3 on v; instead, we test each v with an arbitrary neighbor w of v, and either find a smaller prime subgraph or reject v as a choice.

We are left with a 5 vertex prime subgraph, and we find a circle graph representation of this subgraph. In general, we will go from a prime i vertex subgraph G_i to a prime i vertex subgraph G_{i+1}. If G_i was derived from G_{i+1} by removing v, we simply find appropriate places to add each endpoint of v. If G_i was derived from G_{i+1} via $G_i = G_{i+1} * v$ - v, a model could come by reversing a semicircle of G_{i+1} defined by the endpoints of v, and removing v. We can thus try placing v at each possible position, reversing the appropriate semicircle, and seeing whether this represents G_{i+1}. If $G_i = G_{i+1} * v * w$ - v, the transformation is a little more complex, but can be deduced using elementary observations; if the ordering on the circle in G_{i+1} is v,A,w,B,v,C,w,D then $G_{i+1} * v * w * v$ produces v,C,w,B,v,A,w,D. Thus, we can choose two endpoints for v, switch the positions of the corresponding circular regions A and C, and see if this represents G_{i+1}.

Finding the vertex sequence involves $O(n^2)$ calls to a primetesting algorithm. This takes $O(n^3m)$ time if we test each edge of a spanning tree using the algorithm described above. Primetesting can be done in $O(n^2)$ time, either by the algorithms in [143, 359] for finding the entire decomposition, or the simpler algorithm in [440] for doing only primetesting; thus, the vertex sequence can be found in $O(n^4)$ time.

As we go from G_i to G_{i+1}, we have $O(n^2)$ possible placements of v to try; these placements may also require the rotation of semicircles if $G_i = G_{i+1} * v$ - v or $G_i = G_{i+1} *vwv$ - v. If we perform completely new computations each time, we could take $O(n^4)$ time to go from G_i to G_{i+1}, getting an $O(n^5)$ time bound for circle graph recognition. However, it is easy to keep track of the changes as we go from placing v at position i,j to position i, j+1 efficiently. We leave the details of finding a representation of G_{i+1} from a representation of G_i in $O(n^3)$ time to exercise 13.32; this produces an $O(n^4)$ algorithm for recognizing circle graphs.

13.6.3 Circle Graph Obstructions. The local complement operation was used again by Bouchet to characterize circle graphs in another way. Even before we had a recognition algorithm, circle graph recognition was obviously in NP, since you can prove that you have a circle graph by giving the representation on the circle. Although we can recognize circle graphs in polynomial time, it is not completely satisfying to show that a graph is not a circle graph by running this algorithm, and saying that it is not a circle graph because the algorithm fails to find a representation. Bouchet was able to find a simpler description, which shows that circle graph recognition is in co-NP.

Theorem 13.5 [62] *G is a circle graph if and only if there is no series of local complementations and vertex deletions which yield one of the subgraphs shown in Figure 13.15.*

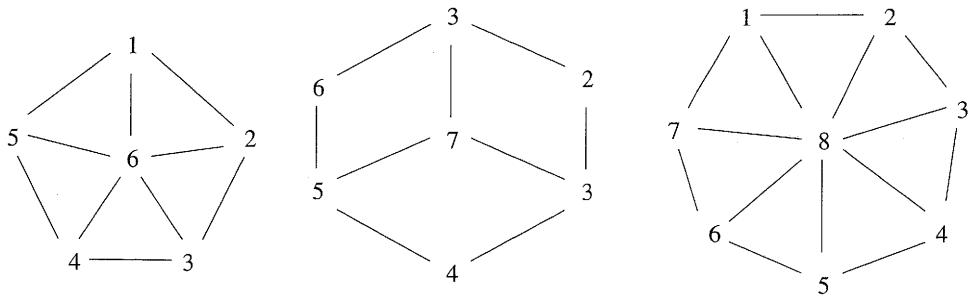


Figure 13.15 Circle Graph Obstructions

The proof of this theorem is surprisingly algebraic rather than graph theoretic, and will not be presented here.

13.7 Circular Permutation Graphs

Permutation graphs are the intersection graphs of segments which connect endpoints lying along two parallel lines. Circular permutation graphs, first introduced in [415], generalize this by letting vertices have endpoints on two concentric circles rather than two parallel lines. These vertices now correspond to chords, and two vertices are defined to be adjacent if and only if the corresponding chords intersect; chords are only allowed to intersect exactly once. Although they are not defined in this way, I find an alternate representation which was known to the authors of [415] but was not written down to be more appealing; circular permutation graphs are exactly equal to the containment graphs of arcs on the circle (exercise 13.35). This representation also makes it clear that circular permutation graphs are comparability graphs, which is important to a nice characterization theorem for the class.

It is natural to try to reduce the recognition problem for circular permutation graphs to recognition of permutation graphs. This leads to the notion of cutting the circles at the endpoints of a vertex v, and 'unrolling' the representation into a permutation graph. A chord w which crosses v will have to be sent around in the other direction of the circle; for a single w, this will have the effect of making w adjacent to exactly those vertices which are nonadjacent to w.

Thus, we have a natural operation switch(G,x), which makes x adjacent to V - x - N(x). We say switch(G,S) is the graph formed from G by successively switching all vertices in S; the net effect is to leave adjacencies within S unchanged, and complement adjacencies between S and V - S.

It is not hard to see that if G is a circular permutation graph, then for any v, switch(N(v)) is a permutation graph. This is not a sufficient condition, but becomes one when combined with the fact that this class is contained in the class of comparability graphs.

Theorem 13.6 [415] *Let v be an arbitrary vertex of G. Then G is a circular permutation graph if and only if G is a comparability graph, and switch(G,N(v)) is a permutation graph.*

Theorem 13.6 gives an O(MM) algorithm for recognizing circular permutation graphs, where the bottleneck step is verifying that G is a comparability graph.

Sritharan [450] modifies the steps of the algorithm to avoid the bottleneck, and produces a linear time algorithm as described below. He uses the results of chapter 11, which shows that a comparability graph can be transitively oriented in linear time, even if we cannot verify that the result is transitive in this time bound.

In general, switching vertices does not preserve possible transitive orientations. However, it is not hard to see that switching a source or sink of a transitively oriented graph gives a transitively oriented graph; the orientation simply turns a source s in G into a sink in switch(G,s). This gives rise to a notion of directed switching for a source vertex; given a source (sink) s in a directed graph G, switch(G,s) is the graph formed by removing all edges out of s, and adding edges from all nonneighbors of s into s for sources, and to nonneighbors from s for sinks. If s is involved in a transitivity violation $s \rightarrow x \rightarrow y$, switch ($G$,$s$) contains the transitivity violation $x \rightarrow y \rightarrow s$; thus, a digraph is transitive if and only if switching a source or sink is transitive.

Thus, Sritharan's algorithm works as follows. Let G be an input graph, and let v be a vertex of minimum degree in G. Find the graph G' produced by the linear time transitive orientation algorithm in [367]. While some neighbor s of v in G is a source or a sink in G', let $G' = \text{switch}(G',s)$.

If the algorithm switches all neighbors of v, and the final graph G' is a two dimensional partial order, then G is a circular permutation graph; otherwise, G is not a circular permutation graph. If all neighbors are switched, the fact that switching sources preserves transitive orientations means that the original graph must have been a comparability graph, and the underlying graph of G' is a permutation graph equal to switch(G,N(v)). If the algorithm stops before all neighbors of v are switched, then G' is not transitive (exercise 13.37). Together, this shows that Sritharan's algorithm produces the same output as the algorithm of [415].

We now show that the time complexity of this algorithm is linear. The time complexity is dominated (in the asymptotic sense) by the cost of computing G'. Each neighbor of v is switched once. Since a vertex can be switched in O(n) time, the total cost is O($n|N(v)|$). Since v is a minimum degree vertex, this is less than or equal to the sum of all vertex degrees = $2m$, so the total cost of finding G' (as well as the final size of G') is O(n+m).

13.8 Weakly Chordal Graphs

We discussed one algorithm for recognizing weakly chordal graphs in chapter 5; in this section, we describe a new algorithm developed in [36].

A minimal separator is a set S of vertices such that for some vertex pair a, b, there is no path from a to b in G - S, and there is no proper subset S' of S such that a and b are in different connected components of G - S'. We note that, counterintuitively, one minimal separator can contain another, since a set which minimally disconnects a from b can contain a set which minimally disconnects c from d. Although this can cause confusion, we want to avoid even greater confusion which would be caused if we used terms differently from the way these are used in the papers this section is based on.

An edge (x, y) of a graph is called LB-simplicial if either the neighborhood of the edge includes V, or for every minimal separator S contained in N(x, y) and every connected component C of the complement of G[S], one of x, y is adjacent to every vertex of C. The key to the algorithm is that G is weakly chordal if and

only if every edge is LB-simplicial; the proof is contained in [36] and is based on the earlier paper [253].

Minimal separators contained in $N(x, y)$ can be found using the following algorithm. For each connected component C of G - $N[x, y]$, neighbors of C are minimal separators contained in $N(x, y)$, and no other minimal separators are contained in $N(x, y)$. This implies that the sum of $|S_i|$ over all minimal separators contained in $N(x, y)$ is at most m, since each member of a separator corresponds to some edge between a vertex of G - $N[x, y]$ and a vertex in $N(x, y)$. We store each separator generated, along with the edge (x, y) whose neighborhood contained this separator.

It can be shown that the number of distinct minimal separators of a weakly chordal graph is at most $m+n$ [36]. However, the algorithm above may generate a single minimal separator many times. We want to avoid computing complement-connected components of a single separator multiple times; we use the following technique to avoid this problem.

Create a vertex/separator incidence graph $S(G)$ in the natural way as follows. One color class of $S(G)$ consists of vertices for each vertex of G. The other color class has vertices corresponding to each separator S generated by the algorithm, with an edge from x to S if and only if x is in S. $S(G)$ can be constructed in $O(m^2)$ time, since each edge addition corresponds to membership in a separator, and each edge contains separators with total size at most m.

Create a partition list with an element for each separator; initially, all separators are place in a single block. Standard data structures can be used to partition each block (a block will correspond to a set of separators) into subblocks of separators which include x and separators which do not include x in time proportional to the degree of x in $S(G)$; this is similar to the partitioning of sets into neighbors and nonneighbors of a vertex during lexicographic breadth first search. After partitioning on every vertex x, separators are in the same block if and only if they contain the same vertices.

If there are more than $m+n$ blocks, we stop and answer that the input is not weakly chordal. Otherwise, we take one separator S from each block, and find all complement connected components of this separator. Since the number of distinct separators is $O(m)$, and complement-connected components can be found in $O(m)$ time, this step takes $O(m^2)$ time.

For each (x, y) such that S is contained in $N(x, y)$ (note that there is one such edge (x, y) stored with each copy of S in the block of the partition list) and each complement-connected component C of S, we verify that either x or y is adjacent to every vertex of C. Since the total size of all separators generated is $O(m^2)$, we can verify that every edge is LB-simplicial (and thus that a graph is weakly chordal) in $O(m^2)$ time.

13.9 Paths in Trees

There are many classes of graphs defined as intersection graphs of paths in trees. We will follow the classification scheme from [374], which gives the following variants, as well as a number of other classes we will not cover here.

1) A path may be viewed as a sequence of vertices, or as a sequence of edges.
2) A tree may be either undirected or directed. Directed trees may or may not be rooted.

This allows us to define 6 classes of graphs. A class defined by vertex intersection will end with a V, while a class defined by edge intersection ends with E. Classes defined by undirected trees start with U, directed (not necessarily rooted) trees with D, and rooted directed trees with RD. Since the terminology used for directed trees is not standardized, let us clarify the distinction between D and RD; a directed tree is a digraph such that removing directions from edges gives a tree, and a directed tree is rooted if there is one vertex r which has a path to all other vertices. Thus, the class UV is the intersection graphs of paths in undirected trees, where two paths intersect if they share a common vertex, while the class RDE is the intersection graphs of paths in a rooted directed tree, where vertices are adjacent if the corresponding paths share an edge. The classes RDV and RDE are equivalent (exercise 13.40), so we are left with 5 classes to study. The UE graphs, more commonly known as EPT graphs, are NP-complete to recognize.

We will present a polynomial time recognition algorithm for the class DE here, and explain how to trivially modify the recognition algorithm for DE to recognize DV as well. The first algorithm for recognizing the class RDV in polynomial time is presented in [215]; a linear time algorithm appears in [158]. For the class UV, or path graphs, the first polynomial recognition algorithm due to Gavril runs in $O(n^4)$ time; the fastest current runs in $O(n + m)$ time [144]. Gavril's algorithm places all cliques containing a single vertex in a valid path in a phase; he constructs a necessary and sufficient set of conditions for constructing such a path, and gives an algorithm for constructing the path. Unfortunately, the set of constraints is difficult to describe concisely. Schäffer gives an $O(nm)$ algorithm [422] which is similar to the algorithm for DE recognition described below, constructing a tree from the subtrees formed by a clique separator, but one step of the construction is much more difficult.

13.9.1 Recognizing Edge Intersection Graphs of Directed Paths in Trees. We present an algorithm due to Monma and Wei [374] to recognize graphs in these classes.

The algorithms for recognizing DV and DE can be summarized as follows. We first find a maximal clique separator decomposition tree for G. This can be obtained by modifying Tarjan's algorithm which is presented in chapter 11 in $O(nm)$ time; we do not give the modifications which produce a maximal clique separator decomposition tree rather than a clique separator decomposition tree.

We will test each indecomposable graph corresponding to a leaf of the maximal clique separator decomposition tree to see that it is in the class using an algorithm described below; a tree model can be constructed for these if it is in the class. We then look at each separator of the maximal clique separator decomposition tree in postorder, and see whether the trees on the children can be combined to derive the appropriate tree on the entire subgraph.

Consider any connected graph H which has no clique separator. It is not hard to see that if H has a representation as paths in a directed/undirected tree T, H can be represented as paths in some directed/undirected star. If the underlying undirected graph of T contains a path u, v, w, x, the set of vertices which use the edge (v, w) form a clique C; if C is not a separator, then at least one of the vertices u or x can be removed from T.

For the class DE, the indecomposable graphs with respect to the clique separator decomposition, corresponding to edge intersection graphs of a directed star, are precisely the line graphs of bipartite multigraphs (exercise 13.41).

Line graphs of bipartite multigraphs can be recognized in linear time. Let G be an arbitrary graph. If G is a line graph, every vertex is in at most two maximal cliques of G. If this condition is satisfied, build a graph G_c with a vertex for each maximal clique, and edges between two vertices if the cliques share a common node. G is the line graph of a bipartite multigraph if and only if G_c is bipartite; the coloring of G_c can be used to derive the appropriate star representation for G by making one color class the inedges of the star and the other color class outedges; a vertex of G then corresponds to a path in this star going between its two maximal cliques.

We now show how to take a clique separator C for a graph G in which both subgraphs created by decomposing G using clique separator decomposition are known to be DE, and test whether the entire graph is DE. The algorithm comes from the following ideas. Every clique C of a DE graph shares a common edge (exercise 13.39). The paths for each vertex of C then branch off from this common edge e or set of edges in two directions; one direction for edges which come after e in all paths, and one for edges which come before e in all paths. If two connected components G_i,G_j of G - C are connected to different subsets C_1, C_2 of C, where neither C_1 nor C_2 contain the other and both contain a common vertex, then one of G_i,G_j must be attached to C before the edge e, while the other must be attached to C after the edge e. This idea is refined slightly to get a complete condition of when two components can be attached on the same side of e, and attaching the models to C is transformed to a two coloring problem on a conflict graph for the components.

Create a vertex G_i for each connected subgraph of G - C. Create an edge between G_i and G_j if and only if at least one of the following conditions holds for some set of cliques defined as follows. Let C_i, C_i', C_i'' be maximal cliques of $G_i \cup C$, and C_j, C_j' and C_j'' be maximal cliques of $G_j \cup C$, where these cliques need not be distinct, and all cliques considered include at least one vertex of C. We say $C_1 \geq (>)\ C_2$ if $C_1 \cap C$ contains (properly contains) $C_2 \cap C$, and $C_1 = C_2$ if these have the same neighbors in C, and $C_1 \mid C_2$ if the cliques have no common neighbors in C. The conditions imply that one subgraph cannot be attached 'further along a common path' from e than the other subgraph. Components with an edge between them are called antipodal components.

Conditions for adding an edge between G_i and G_j:

1) C_i, C_j share a common vertex, and neither contains the other.
2) $C_i > C_j$, $C_j' > C_i'$.
3) $C_i > C_j$, $C_i' \geq C_j'$, $C_i' \mid C_j''$.
4) $C_i = C_j$, $C_i \mid C_j'$, $C_j'' = C_1'$, $C_j'' \mid C_1''$.

If the graph is not two colorable, G is not DE. If the graph is two colorable, there is a natural ordering on each color class which allows us to produce both a valid branching of C and a gluing of the trees on the subgraphs to produce a representation of G. Details are presented in [374]; the reader should understand the general construction, in which G_i precedes G_j if some $C_i > C_j$, which implies that G_i is attached to C closer to e than G_j. Note that for the classes of graphs in question, the number of maximal cliques is $O(n)$ (for DV this comes from the fact

that these are chordal, while edge paths in directed trees satisfy the Helly property as shown in exercise 13.42), so all cliques in each subgraph can be enumerated and these conditions can be tested in polynomial time.

The algorithm for recognizing the class DV is very similar to the one given above. In fact, the only difference in recognition algorithms is the test as to whether an indecomposable graph is in the class. Since any internal vertex of a tree representation is a clique, the atoms of a DV graph are either a single clique, or two cliques sharing some common vertices; these conditions can easily be tested in linear time.

13.10 NP-complete Classes

In this section, we list related class of graphs for which the recognition problem is NP-complete. We do not want to imply that the NP-completeness proofs are all easy or uninteresting; authors such as Kratochvíl and Yannakakis found nice constructions to show longstanding open recognition problems for classes with important applications were NP-complete. However, the conscious effort in this book to deal less with proofs than with algorithms means that the proofs themselves will not be presented here.

I would like to mention at this point that each NP-complete class which has an implicit representation poses a new problem which has not been studied until now. Can we find an implicit representation of members of the class in polynomial time? We discuss this question in chapter 15.

Partial Order Dimension: Is there a set of k linear extensions of P such that for every nonadjacent pair x, y, x comes before y in at least 1 list while y comes before x in some other?

This was one of the original open problems in Garey and Johnson, and was found to be NP-complete for all $k > 2$ by Yannakakis [492]. Variants of the proof used showed that interval dimension k and boxicity k (as discussed below) are NP-complete for all $k > 3$. Some open problems related to this are given below; more open problems on dimension can be found in [460, 443].

Open Problem 13.1: Can height 1, dimension 3 posets be recognized in polynomial time?

Yannakakis showed that height 1, dimension k poset recognition is NP-complete for all $k > 3$. Interest in height 3, dimension 1 posets was heightened by an interesting result due to Schnyder [428]. The vertex/edge inclusion poset of a graph G is a height 1 poset with an element for each vertex, and an element for each edge, with an edge from v to e if and only if v is an endpoint of e. G is planar if and only if the vertex/edge inclusion poset has dimension at most 3.

Open Problem 13.2: Suppose that you are given a partial order P of dimension k. Can you find a set of f(n,k) linear extensions which realize P in polynomial time?

Even for 3 dimension partial orders, no algorithm for producing a set of o(n) linear extensions which realize P seems to be known.

EPT graphs: Given a graph G, is there a tree T such that G is the intersection graph of paths in T, where paths are defined to be intersecting if they share a common edge?

Surprisingly, the seemingly small difference in definition from path graphs, where paths intersect if they share a common vertex, makes the recognition problem NP-complete. EPT graphs are not perfect; in fact, the coloring and clique cover problems are NP-complete on EPT graphs [226]. The clique and independent set problems on EPT graphs are discussed in chapter 14.

Boxicity: Given a graph G and an integer k, is G the intersection graph of isoriented (i.e. aligned with the axes) boxes in k dimensional space?

This was proved to be NP-complete for $k > 3$ in [492]. The boxicity 2 problem, which seems to have applications in such distant fields as ecology [411] and others, was open for many years before being shown to be NP-complete in [320].

Covering Graph: Given a graph G, can you direct G to be the transitive reduction of a partial order?

This was shown to be NP-complete in [79]; note that if transitive reduction is changed to transitive closure, we get the comparability graphs. The problem has not been examined for special classes. Perhaps it would be possible to determine whether it is possible to direct the edges to be the transitive reduction of a two dimensional partial order, giving the covering graph equivalent of permutation graphs, but since dimension is not a covering graph invariant I am less inclined to believe that this is possible.

Planar Posets: A poset is planar if one can embed the transitive reduction in the plane so that every edge goes upwards, and no edges cross. This was shown to be NP-complete in [209].

Disk Intersection Graphs: Is G the intersection graph of disks in the plane?

We discuss optimization on unit disk intersection graphs (which are also NP-hard to recognize) in chapter 14. Disk intersection graphs were shown to be NP-hard to recognize in [322].

Interval Number: Given a graph G and an integer k, is G the intersection graph of k intervals per vertex?

This was shown to be NP-complete in [481]; among variants, one of interest to us is given below, since it is not an implicit representation.

Total Interval Number: Given a graph G and an integer k, is there a set of at most k intervals such that each interval is associated with some vertex, and G is the intersection graph of these intervals?

Lines in the Plane: There are several relevant models here. Hardness results were shown for a variety of such classes in [320]. NP-complete classes to recognize include intersection graphs of horizontal and vertical lines on the plane, and lines in general position. If we allow intersection graphs of not simply lines but nonintersecting curves, in the plane, we get the string graphs, which are NP-complete to recognize [320, 421].

k-polygon graphs: Given a graph G and an integer k, is G the intersection graph of chords of a k-gon, where a chord must go between two different sides? If k is part of the input, this is NP-complete, but is polynomial for any fixed k [170].

13.11 Open Classes

We collect several of the most closely related problems for which recognition is not known to be polynomial or NP-complete. Visibility graph recognition is dealt with in chapter 6.

Open Problem 13.3: Triangle Graph Recognition. There are two related classes of graphs, which contain the permutation graphs and are contained in the trapezoid graphs, and for which the recognition problem remains open. These are the classes of PI (point-interval) and PI* graphs, in which vertices map to triangles with endpoints on two parallel lines. The difference between the classes is that in one class all triangles have a single endpoint on the top line, while in the other class the single endpoint may be on either the top or bottom line.

Open Problem 13.4: Triangles and other polygons inscribed in circle. This problem refers to intersection graphs of triangles inscribed in the circle. The problem is also open for k-gons in the circle for fixed k. The status of the problem for intersection graphs of arbitrary polygons inscribed is less clear. As stated earlier in this book, I now consider this problem open, since there were errors in the proposed polynomial time algorithm of [315], and no final version of the paper has appeared.

Open Problem 13.5: Containment graphs of paths in a tree. For some reason, although there have been many papers on variants of intersection graphs of paths in trees, this class has been overlooked. Perhaps it is because applications of the intersection model are more apparent, but this class seems sufficiently natural to deserve some attention.

Open Problem 13.6: Circle orders and angle orders. These are two well known classes of posets, defined as the containment graphs of circles and of angles. There seem to be no results on recognizing these classes; the techniques used for proving disk intersection graphs are NP-complete may be useful here.

Open Problem 13.7: Interval filament graphs. These graphs were mentioned in Chapter 3, and are the intersection graphs of curves connecting pairs of endpoints on a fixed line, where the curves are required to stay above the line in the plane. They are important because various NP-complete problems can be solved if we are given an interval-filament model, but are open if the input is given in adjacency list form; this will be discussed in more detail in the next chapter.

13.12 Exercises

1) Let after(v) be the set of neighbors of v which come after v in a vertex ordering. Show that a vertex ordering is a perfect elimination scheme if and only if for every vertex v, after(v) is a subset of N[w], where w is the first vertex of after(v) in the ordering.

2) Design a linear time algorithm for recognizing co-chordal graphs.

3) Let M be a module in a chordal graph. Show that if G is chordal, then either M is a clique or N(M) - M is a clique.

4) Show that if v is in k maximal cliques of a chordal graph, then |N(v)| is at least k.

5a) Show that there is a diametral pair of simplicial vertices in a chordal graph; that is, if the maximum distance between pairs of points in G is k, there is some pair of simplicial vertices at distance k apart.

5b) Show that the first vertex of a LBFS on a chordal graph is not necessarily part of a diametral pair.

5c*) A k-sweep LBFS repeats LBFS k times, each time placing as the last vertex that vertex which appears first in the previous LBFS. Show that there is no k such that k-sweep LBFS on a chordal graph produces a vertex in a diametral pair.

6) Design an algorithm which takes a chordal graph G and finds a value d such that the diameter of G is either d or $d+1$ in $O(n^2)$ time.

7) An alternative method for creating a perfect elimination scheme in a chordal graph is called maximum cardinality search. Any vertex is chosen to go last; at each step, you choose an unselected vertex adjacent to the maximum possible number of previously selected vertices as the next vertex in the scheme, with ties broken arbitrarily. Show that it is possible to perform maximum cardinality search of a graph in $O(n + m)$ time. A proof that this gives a perfect elimination scheme for a chordal graph can be found in [456].

8) Design an $O(n+m)$ algorithm to determine whether an interval representation correctly represents G, assuming endpoints are integers in the range $1..2n$.

9) Show that we can construct a consecutive clique arrangement from an interval model in $O(n+m)$ time.

10) Show that a circular-arc graph can have an exponential number of maximal cliques.

11) Show that if G is a circular-arc graph, there is a model of G such that for all edges (x, y) arc x is contained in arc y if and only if $N(x) \subseteq N[y]$, and x and y cover the circle if and only if for every nonneighbor z of x, $N[z] \subseteq N[y]$.

12) Design an $O(n^2)$ algorithm which finds a maximal independent set S such that there is no $x \in S$, y, $z \notin S$ such that $S - x \cup y, z$ is an independent set; remember that a maximal set does not have to be of maximum size.

13) Tucker showed that in every model of a circular-arc graph with clique cover number two, there is a pair p_1, p_2 of points on the circle such that every arc passes through at least one of p_1,p_2. Show that this can be extended to show that in every model of a circular-arc graph with clique cover number $k > 1$, there is a set of k points on the circle such that every arc passes through at least one of these k points.

14) Design an $O(n+m)$ algorithm which finds an odd cycle in the complement of G, or determines that G has clique cover number 2.

15) Prove that the proper circular-arc graph of Figure 13.9 is not a unit circular-arc graph.

16) Show that an interval graph can be preprocessed in linear time so that neighborhood containment queries can be answered in constant time.

17) Show that for any minimal arc v in a circular-arc model (i.e. an arc which properly contains no other arc), the subgraph induced by $N(v)$ has clique cover number at most 2.

18) The $O(n^2)$ algorithm for recognizing circular-arc graphs is based on computing neighborhood containment effectively for the class. Let v be a minimum degree vertex in a circular-arc graph. V - $N(v)$ induces an interval graph I; neighborhood containment when restricted to I can be computed efficiently as in exercise 13.16. Other containments are computed with the help of the following graph. Create a bipartite graph D with one color class consisting of V, and the second class copies w' of each vertex w in $N(v)$. Add an edge between v and w' if and only if v is not equal to w, and v and w are nonadjacent in G.

18a) Show that D is chordal bipartite.

18b) Use the results of part a to show that all neighborhood containments in a circular-arc graph can be computed in $O(n^2)$ time.

19) As part of the $O(n^2)$ circular-arc recognition algorithm, we need to distinguish overlapping relationships from circle-covering relationships on circular-arc graphs with clique cover number 2. Let A, B be a two clique cover of a circular-arc graph. Create a bipartite graph H on color classes $A \cup B$ and B', where B' is a copy of B. Add an edge (a,b') for a in A, b' in B' if and only if a and b are nonadjacent in G. Add an edge $(b_1,b_{2'})$ for b_1, b_2 in B if and only if b is not equal to b' and $N(b_2)$ is contained in $N(b_1) \cup b_1$; H can be constructed in $O(n^2)$ time if we know neighborhood containment relationships between neighbors in G.

19a) Show that H is a chordal bipartite graph.

19b) Show that H can be used to recognize circular-arc graphs with clique cover number 2 in $O(n^2)$ time.

20) Show how to find an optimal coloring of a circular-arc graph with clique cover number two in linear time.

21) We can also test whether G is a proper interval graph by determining from the interval model whether G contains $K_{1,3}$.

21a) Suppose that you are given an interval model for G. Show how to find for all vertices v the leftmost nonneighbor of v which occurs after v in the model in total time $O(n)$.

21b) Show how to find for all vertices v from an interval model the neighbor of v with leftmost endpoint in total time $O(n)$.

21c) Show how to determine whether G is a unit interval graph from the above information in $O(n)$ time.

22) Show that if G is a proper circular-arc graph, there is a circular-arc model in which all pairs of adjacent vertices have an overlap relationship; that is, you must eliminate all circle-covering pairs.

23) Proper circular-arc graphs can also be recognized as follows. We assume that G is not an interval graph. If G does not have clique cover 2, find an odd cycle C in the complement. If C is an independent set a, b, c, find paths between each pair which avoid all neighbors of the third vertex, and place these on the circle. Otherwise, lay out C on the circle. Place each other vertex according to which regions of the circle an endpoint is in; vertices with the same adjacencies to C as a vertex c in the cycle are placed together with c . Use partitioning to get a proper circular-arc representation.

23a) Show that if C has length > 3, C has a unique placement on the circle.

23b) Show that in the partitioning process, no groups meet on both side of the circle. This makes it easier to see that there is only one choice of placements whenever a group is split.

24) Design a linear time algorithm for recognizing bipartite circular-arc graphs.

25) A partial order is an interval order if and only if elements can be mapped to intervals, with $x < y$ if and only if interval x is to the left of interval y. Show that a partial order is an interval order if and only the outneighborhoods of the vertices are ordered by set inclusion.

26) A partial order P has interval dimension k if there is a set of k interval orders which give P as their common intersection. Show that P has interval dimension two if and only if the complement of P is a trapezoid graph.

27) The algorithm for recognizing interval dimension 2 partial orders constructs a trapezoid model, which will be a trapezoid model of the complement of G if and only if G has interval dimension 2. Design a linear time algorithm to determine whether a partial order is equal to the complement of a trapezoid model.

28) Show that G is a circle graph if and only if G is the overlap graph of a set of intervals on the line; in the overlap model, two vertices are adjacent if and only if the corresponding intervals have a nonempty intersection, and neither interval contains the other.

29) The join decomposition works on a graph by finding a split of the graph, and decomposing the two subgraphs formed by this split (including the marker vertex on each side) recursively until all remaining graphs are indecomposable. Show that

after $O(n)$ splits using the join decomposition, every remaining subgraph is indecomposable.

30) Show that G is prime with respect to the join decomposition if and only if $G * v$ is prime.

31) Find a prime (with respect to join decomposition) circle graph G on at least six vertices such that there is no v such that G - v is prime.

32) The circle graph recognition algorithm creates a series of join indecomposable graphs starting with the input graph $G = G_n$. Each graph G_i is either equal to G_{i+1} - v, $G_{i+1} * v$ - v, or $G_{i+1} * v * w * v$ - v for some v and w; the last graph is a prime circle graph on five vertices. It constructs a model for the five vertex graph, and then successively builds the model for each G_i. Show that the model for G_{i+1} can be constructed from the model for G_i in $O(n^2)$ time.

33) Prove or disprove: G is decomposable with respect to join decomposition and only if the complement of G is decomposable.

34) G has a k-join iff the vertices of V can be partitioned into V_1, V_2, each containing at least $k+1$ vertices, and edges which go between V_1 and V_2 can be partitioned into k complete bipartite subgraphs. Show that one can find a k-join of G, if it exists, in polynomial time.

Note that the definition above has some ambiguity, as does the literature, since it is not clear whether we want to allow the complete bipartite graph to sheare common vertices. These variants will not affect the fundamental structure of the algorithm. I will note that in the definition of 2-join used in the proof of the strong perfect graph conjecture the bipartite graphs are not allowed to share vertices. An extra condition is also required which make it unclear whether these 2-joins can be found in polynomial time; there must be a path within the subgraph induced by V_1 between some pair of endpoints from the two different bipartite graphs, and a similar path within V_2. See section 11.7 for more discussion of algorithms for 2-joins and perfect graph recognition.

35) Show that circular permutation graphs are exactly equal to containment graphs of arcs on the circle.

36) Find a graph G and vertex v such that switch(N(v)) is a permutation graph, but G is not a circular permutation graph.

37) In circular permutation graph recognition, the operation switch(x) makes x adjacent to y if and only if x is currently nonadjacent to y. Let v be any vertex in a comparability graph G. Consider repeatedly switching any neighbor of v which is a source or sink in an arbitrary transitive orientation, until the resulting graph has no transitive orientation or there are no more source or sink neighbors of v. Show that this procedure will switch all neighbors of v before stopping.

38) This question explores the containment relationships between graph classes defined by different notions of intersection in trees. Recall that E and V mean that intersection is defined by edge intersection or vertex intersection. Trees are classified either as U = undirected, D = directed (i.e. there are arbitrary directions assigned to an underlying undirected tree), and RD = rooted and directed.

38a) Show that RDV = RDE.

38b) Show that RDE is a proper subclass of DV.

38c) Show that DV is a proper subclass of UV.

38d) Show that UV is a proper subclass of chordal graphs

38e) Show that DV is a proper subclass of DE.

38f) Show that DE is a proper subclass of UE.

39) Show that every clique of a DE graph shares common edge in the tree, and that this is not true for cliques in UE graphs.

40) Show that there are 3 dimensional posets which are not containment graphs of paths in a tree, and that there are posets which are not 2 dimensional but are containment graphs of paths in a tree.

41) Show that G is a DE graph which is indecomposable with respect to clique separator decomposition if and only if G is a line graph of a bipartite multigraph.

42) Show that if we define a pair of paths in a directed tree to intersect if they share a common edge, then every set of mutually intersecting paths must share a single common edge.

43) Show that dimension is not a covering graph invariant, i.e. that there is a graph G which can be directed to be the transitive reduction of two partial orders P_1, P_2 such that the dimension of P_1 is different from the dimension of P_2.

CHAPTER 14

Robust Algorithms for Optimization Problems

In this chapter, we look at the relationship between optimization problems and representations of graphs.

Before we look at solutions to optimization problems on specific classes of graphs, we want to emphasize some issues that are rarely addressed in the literature. This is especially important when we deal with classes of graphs which have nice representations, but there is no known polynomial time algorithm for constructing these representations; indeed, recognizing the class of graphs and constructing the representation may be NP-hard. It is often important when dealing with these classes to distinguish between solving the optimization problem on the class if the input is given in the form of a good representation, and if the input is given in the form of adjacency lists or matrices. When the recognition problem is NP-complete, this separates out the question of whether a problem is easy on a class because the graph class is tractable from whether the problem is easy on the class because you are given the answer to an NP-hard problem.

A further distinction is important if the input is given in adjacency list or adjacency matrix form. Suppose that we know a fast algorithm for solving an optimization problem if it is in a special class C of graphs. Is it sufficient to run the algorithm, and say that we have solved the problem on the class C? The answer depends on our assumptions. If we are given a true promise that the input graph is in the class, then we are satisfied. However, we may only suspect, or perhaps merely even hope, that the graph is in C. If we are required to run a recognition algorithm to determine whether G is in the class, we cannot solve any problems on NP-complete classes efficiently. There is another way to solve optimization problems on special classes that should, in my opinion, be entirely satisfactory, even if the recognition problem is NP-complete for the class.

An algorithm which solves a problem (either a decision problem or an optimization problem) on class C is called robust if it satisfies the following conditions.

1) Whenever the input is in class C, the algorithm finds the correct solution.
2) If the input is not in the class C, then the algorithm either finds the correct solution, or answers that the input is not in the class.

An algorithm which satisfies condition 1 above but not condition 2 will be called a nonrobust algorithm for the class C.

Robust algorithms might seem rare, but they are actually quite common. Often, they work by solving the problem on a larger class of graphs. For example, it is NP-complete to recognize planar Hamiltonian graphs, but any algorithm which solves a problem on planar graphs is a robust optimization algorithm for planar Hamiltonian graphs. In other cases, the algorithm solves the problem exactly on

the class C, but if the input is in some larger class it may (depending on choices made by the algorithm) either solve the problem or answer that it is not in the class C. We will make this issue more concrete by discussing a problem which has a very simple nonrobust solution, but is difficult in the robust sense.

A graph is well-covered if every maximal independent set is a maximum independent set. For a survey of well-covered graphs, see [398]. Well-covered graphs are interesting in this context, because finding a maximum independent set in a graph which is known to be well-covered is particularly simple (any greedy method works), but recognizing well-covered graphs is co-NP complete [419]. Thus, if we are given a promise that the graph is well-covered, independent set is in P; it is not clear what happens in the model that we present here. In our terminology, there is a nonrobust algorithm for independent set in well-covered graphs, but this does not necessarily imply the existence of a robust polynomial algorithm. We note that the representation problem for well-covered graphs is not interesting, since the number of well-covered graphs on n vertices is $2^{\Theta(n^2)}$. Later in this chapter, we will show that there is no robust polynomial algorithm for the independent set problem in well-covered graphs unless P = NP.

In the opinion of this author, it is important to design robust algorithms for a problem before you declare the problem to be solved on the class of graphs. Although situations exist in which you may have a true promise that a graph is in a class, one cannot completely trust the result in general unless it produces some form of valid output if the graph happens to be outside the class. As an example, it has been conjectured that certain graphs arising in the study of ecosystems always have boxicity 2. Thus, algorithms for solving problems on such graphs would then want to use boxicity 2 algorithms, if the problem is solved efficiently on boxicity 2 graphs. However, if the conjecture is wrong, we do not want to get incorrect answers to our problems!

For some difficult problems, we might want to create a library of routines that work for certain restricted classes. The first thought is to have a corresponding recognition algorithm for each class, so that if a new graph is given to the library, we test whether it is in each class, and if it is in a tractable class we run the appropriate algorithm. However, this assumes that the recognition algorithm is at least as fast as the optimization algorithm. We will see examples of problems for which the recognition problem is harder than the optimization problems for the class. For these classes, a robust algorithm lets us avoid the recognition problem entirely.

In the next section, we will examine classes of graphs and problems for which a robust algorithm exists which is faster than any recognition algorithm. In the following section, we look at problems where having a representation leads to algorithms which are much more efficient than any algorithm which is known if the input is given in adjacency list form. Some of these problems will be candidates for problems in which the representation provably helps, an open problem mentioned in chapter 2. We note that independent set in well-covered graphs does not quite fit either model; this is a problem for which a promise, rather than a representation, seems to help. The independent set problem on well-covered graphs is studied in detail in the complexity section of this chapter.

14.1 Robust Algorithms which are Faster Than Recognition

We start with an example of a problem on a natural class of graphs, such that there is a robust algorithm which is much faster than the recognition problem. Consider the maximum clique problem on boxicity 2 graphs; the solution described can be generalized to boxicity k for any fixed value of k. The recognition problem for boxicity k graphs is NP-complete for any fixed k greater than or equal to two.

It is easy to find a maximum clique in a boxicity 2 graph if we are given a representation of the vertices as rectangles in the plane. The key observation is that rectangles have the Helly property; this says that any clique must meet at some common point. It is easy to see that a boxicity 2 graph can be represented on a $2n$ by $2n$ grid. Thus, we can simply check how many boxes cover each of the grid points; the maximum value will be the size of the maximum clique.

It is not difficult to modify the above approach to produce a robust algorithm for the maximum clique problem on boxicity 2 graphs. The argument above shows that if G is a boxicity 2 graph, then The number of maximal cliques of G is at most $4n^2$, since each maximal clique corresponds to some grid point in a $2n$ by $2n$ grid. Maximal cliques of a graph can be enumerated in $O(n^3M)$ time, where M is the number of maximal cliques [285], using an algorithm which takes $O(n^3)$ time to generate the next maximal clique. Given an arbitrary graph as input, we can enumerate maximal cliques until either the number of maximal cliques is greater than $4n^2$, in which case we answer that G does not have boxicity 2, or we have enumerated all of the (at most $4n^2$) maximal cliques, in which case we solve the maximum clique problem correctly whether or not the input graph has boxicity 2.

A similar, but slightly more complex example, deals with EPT graphs. These are the intersection graphs of paths in a tree, if two paths are said to intersect when they share a common edge rather than a common vertex as in path graphs. This apparently small variant makes the class NP-complete to recognize; the coloring and clique cover problems also remain NP-complete on the class [226]. We will look at polynomial algorithms for the clique and independent set problems on EPT graphs; the algorithms are based on properties which come from the model but do not need to be given the model as input.

For the clique problem, although all cliques must go through a common tree vertex, it is not true that all sets which go through a common tree vertex form a clique; thus, we might think that the number of maximal cliques could be exponential. It is not necessary for all cliques to share a common edge. However, Golumbic and Jamison [226] were able to show that each maximal clique either corresponds to a set of paths which share a common edge of T, or a set of paths which go through two edges of some claw ($K_{1,3}$) in T. Since the size of T can be shown to be $O(n)$ (exercise 14.7), the number of maximal cliques is polynomial, and we can solve the clique problem by enumerating all maximal cliques in polynomial time without being given the tree model.

The algorithm for solving the independent set problem on EPT graphs in polynomial time is due to Tarjan [454]. We use the clique separator decomposition discussed in chapter 11. As the name implies, this involves finding a clique C such that the graph induced by G - C is disconnected; we then decompose $G_i \cup C$ for each connected component G_i of G - C. Clique separators can be used to decompose an EPT graph into subsets of vertices which correspond to paths in stars; if the tree is not currently a star, then the set of vertices which pass through any edge

e which disconnects the tree form a clique cutset. An algorithm for computing this decomposition is given in chapter 11.

Theorem 14.1 [454] *G is an edge intersection graph of paths in a star if and only if G is a line graph.*

Proof Let G be the edge intersection graph of the paths $p_1, ..., p_n$ in a star S with center c; we may assume that all paths have length 2 (exercise 14.7). G is the line graph of G', where vertices of G' correspond to the vertices of S other than c. For each p_i x, c, y, create an edge from x to y in G'; it should be clear that two paths in G share a common edge if and only if the corresponding edges of G' share a common endpoint.

Similarly, for any graph H, the line graph of H is the edge intersection graph of paths in a star. Create a star with one outer vertex for each vertex in H, and a path from x to y in the star for each edge from x to y in H; two paths on the star share an edge if and only if the corresponding edges of H share a common endpoint. □

Line graphs can be recognized in linear time [340, 416]; an associated graph G such that the input is the line graph of G is also found by these algorithms. Independent sets in a line graph correspond to matchings in the associated graph; thus, we can use any weighted matching algorithm to solve the weighted independent set problem in polynomial time for each indecomposable graph found by the clique separator decomposition. Let G_i be one of the indecomposable graphs found by the clique separator decomposition, and let C be the clique separator which separates G_i - C from the remaining graph. Find the weight w_i of the maximum independent set in G_i - C. For each c in C, compute the weight w_c of the maximum independent set on G_i - $C \cup c$. Remove G_i - C from the graph, and give each vertex c in C a weight w_c - w_i. The weight of the maximum independent set in the new graph is equal to the weight of the maximum independent set in G - w_i.

This gives a polynomial time algorithm for finding the maximum independent set problem on an an EPT graph, which does not use the tree model. If the input graph G is not an EPT graph, we will either find that some graph found by the clique separator decomposition is not a line graph, or we will find the correct value for the maximum independent set in G.

Another example of solving a problem robustly and more efficiently by moving to a larger class of graphs deals with optimization for strongly chordal graphs. We noted in chapter 12 that the fastest known algorithms for recognizing strongly chordal graphs and finding a simple elimination scheme take $\Theta(n^2)$ time. However, strongly chordal graphs are a subset of the doubly chordal and dually chordal graphs defined in chapter 12, and there are linear time algorithms for the dominating set problem on doubly and dually chordal graphs; thus, we can solve the optimization problem on strongly chordal graphs faster than they can be recognized. This is of more interest than, for example, solving an optimization algorithm on 3-colorable planar graphs by using an algorithm designed for general planar graphs, since using the simple elimination ordering would seem to be the natural method for solving the dominating set problem on strongly chordal graphs.

The above problems are examples of solving a problem robustly by solving the problem on a larger class of graphs. Thus, boxicity 2 graphs are a subclass of graphs with at most $4n^2$ maximal cliques, while EPT graphs are a subclass of graphs which

decompose via clique separators into line graphs. Nevertheless, it is important to realize that the reason that the problem is easy on the class is not because we are given a nice representation, but because the class has interesting properties which allow fast optimization.

Many robust algorithms will be of the form above. However, it is not always easy to determine what the larger class is that can be solved efficiently, and finding the larger class can lead to solutions of other optimization problems on the class.

Finding a robust algorithm does not always mean solving the problem on a larger class of graphs. We give examples in which the class of graphs where the problem is solved by the robust algorithm is not well defined. This occurs when the algorithm on a single input graph G might either solve the problem correctly, or answer that G is not in the class; the output might depend, for example, on the numbers given to vertices of G.

Our first example is the clique problem on comparability graphs. As we saw in chapter 11, there is a linear time algorithm for producing a transitive orientation of a comparability graph, but there is no known recognition algorithm for comparability graphs which is faster than matrix multiplication. The orientation algorithm will direct the edges of a comparability graph correctly in linear time, but on other graphs it can produce some other (nontransitive) orientation in linear time; checking the transitivity of the resulting digraph is the bottleneck step of the recognition problem.

Suppose that we are given a linear extension L of a transitive orientation of a comparability graph G. The following algorithm finds the size of the maximum clique in G. We keep track of the size of the largest clique $\mathrm{C}(i)$ found involving each vertex i; initially $\mathrm{C}(i) = 1$ for all i. Step through L in order; let x be the next vertex of L. For each edge $x \rightarrow y$ of the transitive orientation of G, let $\mathrm{C}(y)$ be the maximum of $\mathrm{C}(y)$ and $1+\mathrm{C}(x)$. When we reach the end of L, the maximum clique in G is the largest value of $\mathrm{C}(i)$.

We first show that this algorithm works to find a maximum clique if G is a comparability graph, but does not necessarily work if G is not a comparability graph. We then give a simple modification which makes it a robust algorithm, and show that this robust algorithm may give different answers on the same input graph.

Let G be a comparability graph. Suppose that G has a clique $C = c_1, c_2,\ldots,c_k$, where vertices of C are numbered in increasing order of position in L. When each c_i is reached, the edge $c_i \rightarrow c_{i+1}$ causes c_{i+1} to have a final value which is at least one larger than c_i, so we will answer that G has a maximum clique of size at least k. Suppose that the algorithm answers that G has a clique of size j. We can find a sequence $c_1,c_2,\ldots,c_j$ such that c_i caused c_{i+1} to be labeled with the value $i+1$. This implies that there are edges from c_i to $c_i + 1$ for $i = 1,2,\ldots,j\text{-}1$; since the orientation is transitive, this implies that $c_1,c_2,\ldots,c_j$ form a clique of size j in G. Therefore, if G is a comparability graph, we can find the maximum clique in G in linear time.

Of course, this algorithm may produce an incorrect value if the input graph G is not a comparability graph. Consider a graph consisting of $C_7 = 1,2,3,4,5,6,7,1$ and a vertex 8 adjacent only to 1, 3, and 5, and suppose that the orientation algorithm produces the list $L = 8\ 7\ 6\ 4\ 5\ 2\ 1\ 3$ as a linear extension of its incorrect transitive orientation. Then $\mathrm{C}(5) = 3$ at the end of the algorithm, although G has no clique of size 3. However, this problem can be remedied easily, using the observation that

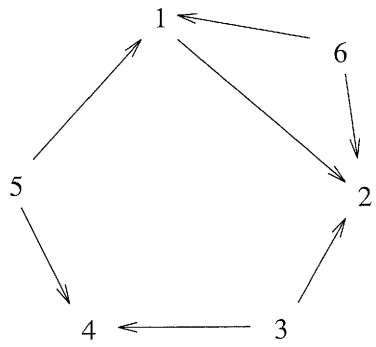


Figure 14.1 Possible Orientation for a Noncomparability Graph

the value found by the algorithm is always an upper bound on the value of the maximum clique.

Our modification of the algorithm above to make it robust is straightforward. For each vertex v, maintain a parent pointer $p(v)$, with all parent pointers initially nil. We run the same algorithm, but whenever vertex x changes the value of $C(y)$, we change the parent value of y to point to x. When the maximum value $C(i)$ is found at the end of the previous algorithm, we use the parent pointers to construct the set S which should be a clique of maximum size if G is a comparability graph. If S is a clique, then we have found a maximum clique in G, and if S is not a clique we answer that G is not a comparability graph.

If G is a comparability graph, this algorithm works for the reasons proved earlier. Suppose that G is not a comparability graph. If the algorithm answers that G is not a comparability graph, we have found a set S such that the orientation algorithm produced edges $s_1 \rightarrow s_2 \ldots \rightarrow s_i$, but S was not a clique of G. Since the orientation produced was not transitive, G is not a comparability graph. Suppose that G has a clique C of size j. If the vertices of C are ordered $c_1 \ldots c_j$, it is clear that the value associated with vertex c_i is less than the value $C(c_{i+1})$. Thus, the maximum C value over all vertices is at least as large as the maximum clique, and if the algorithm returns a value it actually finds a clique of this size. Therefore, if the input is not a comparability graph, the algorithm will either answer that G is not a comparability graph, or find a maximum clique in G. All steps of the algorithm can be implemented easily (once we are given the linear extension L) in linear time, despite the fact that we cannot recognize comparability graphs in linear time.

It is interesting to note that we cannot say that this algorithm is solving the problem on a specific larger class of graphs. Consider a 5-cycle 1,2,3,4,5,1 plus vertex 6 adjacent to exactly 1 and 2. This graph has no transitive orientation, and the algorithm may come up with the orientation of figure 14.1. The linear extensions 5 6 3 4 1 2 and 6 5 3 4 1 2 are both consistent with this orientation, and each could be produced using the same algorithm by renaming vertices. If 5 3 6 4 1 2 is used, the algorithm would find the candidate clique 5,1,2, determine that this is not a clique, and answer that G is not a comparability graph. If 6 5 3 4 1 2 is used as a linear extension, the algorithm answers that 6,1,2 is a largest clique in G. Thus, depending on choices made by the algorithm, the algorithm may or may not find the maximum clique on this graph.

We next outline an algorithm from [270] for finding the maximum independent set in a circular-arc graph in linear time. Note that circular-arc graphs can now be recognized in linear time [365], but this algorithm will solve the independent set problem robustly on circular-arc graphs in linear time without resorting to the much more complex recognition algorithm. We present only part of the algorithm here; the key point is that we eliminate vertices which clearly do not change the size of the maximum independent set, but may change a graph from being outside the class to being in the graph class.

The algorithm selects any minimum degree vertex v from the graph. We remove any neighbor w of v such that N(v) is a subset of N(w), since w could be replaced by v in any maximum independent set, whether or not G is a circular-arc graph. It then checks a number of conditions which must hold for G if G is a circular-arc graph. These include testing whether the graph induced by V - N(v) is an interval graph. If the set of conditions, all of which are testable in linear time, are satisfied, the algorithm takes O(n+m) time, and either finds the maximum independent set on the remaining graph whether or determines that the remaining graph is not a circular-arc graph. Details of the algorithm can be found in [270].

Consider how this algorithm would behave if the input is the cycle 1,2,3,4,5,6,7,1 plus vertices x adjacent to 1,2,4,7 and y adjacent to 2,3,4,5,6,7. If 1 is chosen as a minimum degree vertex, G - N(1) is an interval graph; in fact, x will be removed and the remaining graph is a circular-arc graph, so the algorithm will find the maximum independent set in G. If 6 is chosen as the minimum degree vertex, G - N(6) is not an interval graph (it contains the chordless cycle 4,x,2,3), so the algorithm will answer that G is not a circular-arc graph. Both answers are acceptable, but the algorithm may or may not find the maximum independent set when this graph is given as input.

We now present an example of a problem which is well known in the literature, although it is not usually recognized that the optimization problem is solved much more quickly than the recognition problem on this class of graphs. We will discuss the matching problem on claw-free graphs. Recognition of claw-free graphs is discussed in chapter 8; no algorithm seems to be known other than those which can be derived easily from the definition. The best recognition algorithm seems to take n matrix multiplications.

For general graphs, no linear time matching algorithm is known. There is a very simple algorithm for finding a maximum matching on claw-free graphs, however. Construct a breadth first search tree T for G; we will process T in postorder. Consider the set of children of any node p in T which is not the root. The children must form a clique, or p, parent(p), and any two nonadjacent children of p are a claw. Thus, we match consecutive children at such a level; if 1 child remains, we match the child with p. If any of these consecutive children are nonadjacent, we declare that G is not claw-free.

We are left with a set of children of the root r. Take the first three children; there must be some edge between a pair, or they form a claw with r; match any which have an edge. Repeat this on any 3 remaining children, until we have a root and at most 2 children. If any children remain, match the root with 1 child. Thus, we match all but at most 1 vertex. Since each new matching edge is found in constant time, this algorithm takes O(n) time once the BFS tree has been constructed. If we find a matching with at most 1 vertex unmatched, the matching is clearly optimal, whether or not the original graph is claw-free; otherwise, we will declare

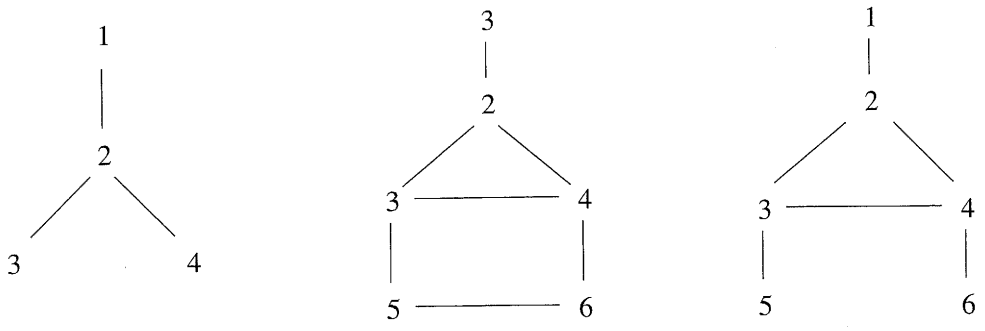


Figure 14.2 Claw, Antenna, and Net

that the input is not claw-free. Note that if G is not claw-free, our result may depend on the particular breadth first tree which is constructed for G.

Thus, the frequently cited theorem that every claw-free graph on an even number of vertices has a perfect matching is also an example of a robust linear time optimization problem on a class which cannot be recognized in linear time.

An interesting robust algorithm known as struction was developed by Hammer, Mahadev, and DeWerra [247] for solving independent set on graphs without any of the forbidden subgraphs shown in figure 14.2; such graphs are called CAN-free (for claw, antenna, net free).

Create a graph G' from an arbitrary graph G with vertices numbered from 1 to n and any vertex v of G as follows. Add a vertex wx for every nonadjacent pair of neighbors $w < x$ of v. Add edges between a vertex y of G and a new vertex wx of G' if y is adjacent to at least one of w or x in G. Add an edge between new vertices wx and yz unless $w = y$ and x and z are nonadjacent in G. Delete v and $N(v)$ from G'.

For any graph G, the size of the maximum independent set in G is equal to 1 + the size of the maximum independent set in G'. Therefore, this is called a stability reduction step; the name struction is derived from this. We can find the maximum independent set in an arbitrary graph G by applying this rule at most n-1 times.

This does not imply that P = NP, since in general the size of G' can be larger than the size of G. The authors were able to show that if G does not contain any of the forbidden subgraphs, then G' is no larger than G and is CAN-free. Since each transformation takes $O(n^2)$ time, where n is the number of vertices in the output graph, this implies an $O(n^3)$ algorithm for the independent set problem on CAN-free graphs. If we do not know whether G is CAN-free, we can simply run this algorithm; if the graph ever grows during a transformation, we know it is not CAN-free, and otherwise we solve the independent set problem correctly on G. In fact, a graph with a forbidden subgraph may be transformed into a graph with no forbidden subgraphs by the algorithm, so it sometimes optimizes other graphs as well; there seems to be no obvious method for testing whether G is in the class besides trying all k vertex subgraphs. The algorithm above is interesting in that a graph class is defined on the basis of an algorithm, as well as the natural interest in a general algorithm for solving the independent set problem. The technique was later refined to solve the independent set problem in claw-free net-free (CN-free) graphs [248].

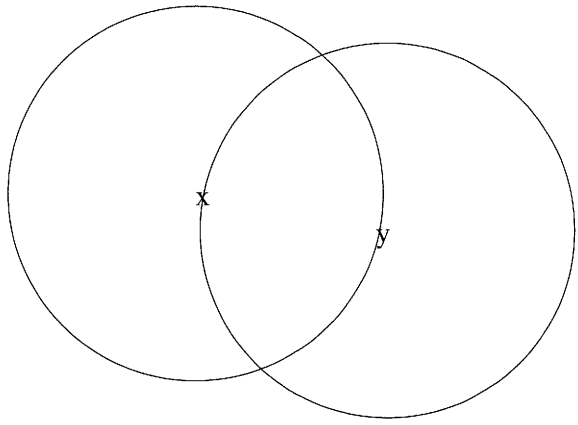


Figure 14.3 Example Graph for Clique in Unit Disk Graph

Eventually, structural properties of the class of CN-free were discovered which allowed problems to be solved more efficiently; these led to linear time algorithms for Hamilton cycle on CN-free graphs [71] and an $O(nm)$ algorithm for independent set, which is faster than the algorithm based on struction [70]; nevertheless, it was the general technique of struction, and the robust algorithm it produces, that motivated study of this class.

14.1.1 Cliques in Unit Disk Graphs. We now present an algorithm from [402] which takes a problem which was known to be solved in polynomial time if a model was given as input, and modifies the algorithm to be a robust algorithm. The author first became aware of this problem through the web-based survey on intersection graphs written by Prisner [399].

G is a unit disk graph if G is the intersection graph of a set of circles with same radius. The first paper dealing with cliques on unit disk graphs was by Clark, Colbourn, and Johnson [112]; they give a method for solving the problem if the centers and radii of each circle was known. However, it has been shown since that time that it is NP-hard to recognize unit disk graphs [77] (membership in NP is open), so one might think that an algorithm which uses the model would not help in designing a robust algorithm which takes the adjacency matrix as input. We will see that by focusing on certain steps of the algorithm which lead to efficient computation of the clique problem for this class, we are able to remove the dependence on the model.

Clark, Colbourn, and Johnson's algorithm takes the model as input, and examines every pair of adjacent vertices x,y. Let $d(x, y)$ be the distance between the centers of circles x and y. For each of x,y, draw a circle with this center and radius $d(x, y)$ as in figure 14.3. Let $S(x, y)$ be the set of vertices with centers in both of these circles; we note that this will not necessarily include all common neighbors of x and y.

It is not hard to see that $S(x, y)$ induces a graph which can be covered by two cliques, i.e. a co-bipartite graph. If we divide the area of common intersection by a line between x and y, the distance between any of the centers in one section can be no larger than the distance between the center of x and the center of y, implying that the set forms a clique. Thus, finding the maximum clique in $S(x, y)$ corresponds to

finding the maximum clique in a co-bipartite graph. This is equivalent to finding the maximum independent set in a bipartite graph; it is well known that this can be done as quickly as bipartite matching, which can be solved in $O(n^{2.5})$ time.

However, since $S(x, y)$ does not include all common neighbors of x and y, it is not immediately apparent that we can use this to find the maximum clique in G. Let C be a maximum clique, and let u,v be a pair of centers of maximum distance in C. It is easy to see that C is the maximum clique in $S(u, v)$; thus the size of the maximum clique found over all possible $S(x, y)$ will be the size of the maximum clique in G.

We now modify the solution to be a robust algorithm for the maximum clique problem on unit disk graphs even if the model is not given as input. We focus on what makes the above solution polynomial; we want to search for maximum cliques in co-bipartite subgraphs in such a way that we are guaranteed that one of these contains the maximum clique in G.

Let G_i be the graph formed by including only edges which occur after edge i in an ordering. An ordering of the edges of G is called a co-bipartite edge elimination ordering if the following property is fulfilled.
For all $e_i = (x, y)$, the common neighbors of x and y in G_i induce a co-bipartite graph in G.

We will show that every unit disk graph has a co-bipartite edge elimination order, and that we can find such an ordering (if it exists) in an arbitrary graph in polynomial time. We then show that we can find a maximum clique in any graph with a co-bipartite edge elimination order efficiently; together, this gives a robust polynomial algorithm for the clique problem on unit disk graphs.

First, we show that every unit disk graph has a co-bipartite edge elimination order. Given a unit disk model, order the edges by decreasing distance between the corresponding centers. For any edge $e_i = (x, y)$, the neighbors of x and y in G_i are those vertices closer to x than y is, and closer to y than x is. These are exactly the vertices $S(x, y)$ in the Clark, Colbourn, Johnson algorithm, and are co-bipartite in G for this reason.

Next, we show that it is possible to find a co-bipartite edge elimination order in polynomial time, if such an ordering exists. We use a greedy algorithm; simply take any e such that the common neighbors of the endpoints of e induce a co-bipartite graph, and place it next on the list, and delete it from the set of edges in G_i. Suppose that this fails (i.e. this halts before putting all edges in the list), but there is a valid ordering O for the edges of G. Let e be the first edge in O which is not in the output list. Then the common neighbors of e in G_i with respect to our list are a subset of the common neighbors of e when it was placed in O (since all earlier edges are also in our list); thus, our algorithm would not have terminated at this point.

Finally, it is easy to solve the maximum clique problem in polynomial time, if a co-bipartite edge elimination order is given. We simply take the maximum clique over all the co-bipartite $N(e_i)$ with respect to the ordering. Let C be the maximum clique in G, and let (x, y) be the first edge of C in the order. Then C is a subset of the joint neighbors of x and y, and will be found at this step.

Therefore, we have modified the algorithm given the model to solve the problem robustly for unit disk graphs. If the input graph does not have a valid ordering, we can stop and declare that the input is not a unit disk graph. If the input has such an ordering, we will solve the clique problem correctly (whether or not the input is

in the class); since we solve the problem correctly on all unit disk graphs and give correct answers on all graphs not in the class, this algorithm is robust.

14.2 Problems Helped by a Representation

This section studies problems which can be solved more efficiently if one is given a good representation of a graph class than the best known algorithms for the problem if the graph is given in adjacency list form. We should mention that many problems have faster algorithms if one is given the representation. For example, the independent set problem for circular-arc graphs can be solved in $O(n)$ time if the input is in the form of a circularly ordered list of arc endpoints [271, 225, 363]. We are primarily concerned here with problems where the algorithm is polynomial if the representation is given, but no polynomial algorithm is known if the input is in the class but is given in adjacency list form. The major open problem, stated as Open Problem 2.2, is to find a problem which is not solvable in polynomial time (making standard complexity-theoretic assumptions) on a class if given in adjacency matrix form, but is polynomial on the class if given the good representation scheme.

It is surprisingly difficult to find problems which are hard if the representation is not given, but easy if it is given. We will discuss in more detail two problems which can currently be solved much more efficiently if a good representation is given. The first is the clique problem on visibility graphs, which can be solved in polynomial time if the visibility polygon is given; in this case, we do not yet know either whether the problem is difficult if given in adjacency matrix form, or whether finding the visibility polygon is NP-hard. The second problem we deal with is coloring perfectly orderable graphs; here, the problem is very easy if given a 'representation'; it is known to be polynomial, though in a somewhat unsatisfactory sense, if the input is given in adjacency matrix form. Finding the representation is known to be NP-complete for perfectly orderable graphs.

14.2.1 Cliques in Visibility Graphs. Before discussing the clique problem on visibility graphs, let us note that the Hamilton cycle problem on visibility graphs could also be said to be a candidate problem for being difficult without a model and easy with a model. However, the decision version of the Hamilton circuit problem for visibility graphs is trivial, in the sense that the answer to the Hamilton circuit problem is always yes. Nevertheless, the problem of constructing a Hamilton circuit of a visibility graph is open, if the adjacency matrix is given as input.

As we discussed in chapter 6, visibility graph recognition is a major open problem; in fact, the problem is not known to be in NP. The clique problem on visibility graphs was first solved in [161]. We will describe the algorithm quite differently, however, since we intend to focus on what is possible if the model is not given, whereas that paper was written with the assumption that the polygon was part of the input.

It is possible to compute the maximum clique which contains point p as its leftmost endpoint; since the maximum clique must have some leftmost endpoint, this gives a polynomial algorithm for the clique problem. We will now describe the basic property which allows us to compute the maximum clique containing p as leftmost endpoint in polynomial time.

Order the vertices of $N(p)$ which are to the right of p in the same order as they appear in the polygon, assuming that the polygon starts at p. Suppose that there are $p_1 < p_2 < p_3$ in this ordering, such that these three vertices form a triangle in

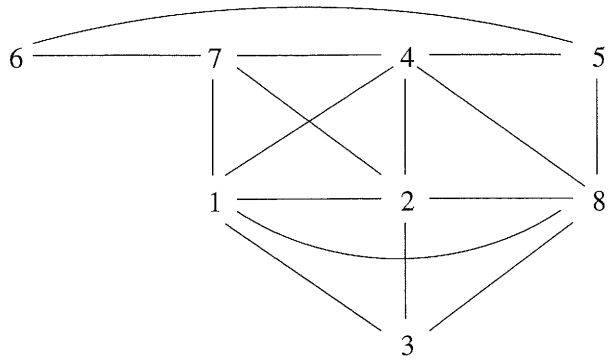


Figure 14.4 Example Graph for Clique in Visibility Graphs

the visibility graph. For any vertex of N(p) $p_0 < p_1$ which is adjacent to p_1 and p_2, simple geometric arguments can be used to show that p_0 must be visible from p_3. This implies that if we want to determine whether a clique can be extended to include a new vertex with higher number in the ordering, it is sufficient to keep the two highest numbered vertices in the clique.

Thus, we get a natural dynamic programming algorithm to find the largest clique containing p as leftmost vertex. Let C_{ij} be the size of the maximum clique containing i and j as the second to last and last vertices of the ordering. Neighbors of p are added to the set under consideration in order of their appearance on the polygon; let k be the next vertex added. For all $i < j < k$, if k is adjacent to both i and j, then one candidate for C_{jk} is $C_{ij} + 1$; if this is larger than the current value of C_{jk}, we make it the current value. It is easy to keep track of the clique itself, rather than just the size of the clique, without affecting any time bounds (exercise 14.10).

Let us present an example of this algorithm, on the graph of figure 14.4. We assume that neighbors of p are ordered 1..8 on the polygon. We start with vertex 1 by itself. Since 2 is adjacent to 1, $|C_{12}| = 2$. Since 3 be adjacent to 1 and 2; $|C_{13}| = 2$, but $|C_{23}| = |C_{12}|+1 = 3$. Adding 4 gives $|C_{14}| = 2$ and $|C_{24}| = 3$. Since 5 is adjacent only to 4, we get $|C_{45}| = 2$, and adding 6 gives $|C_{56}| = 2$. Adding 7 adjacent to 1,2,4,6, we get $C_{17} = 2$, $C_{27} = |C_{12}|+1 = 3$, $|C_{47}|$ is 1 plus the maximum of $|C_{14}|, |C_{24}| = 4$, and $|C_{67}|$ is 1 plus the maximum of $|C_{16}, C_{26}, C_{46}, C_{66}| = 2$. Adding 8 adjacent to 1,2,3,4,5 gives $|C_{18}| = 2$, $|C_{28}| = 3$, $|C_{38}| = 1 +$ the maximum of $|C_{13}, C_{23}| = 4$, $|C_{48}|$ is $1 +$ the maximum of $|C_{14}, C_{24}, C_{34}| = 4$, and $|C_{58}|$ is $1 +$ the maximum of $|C_{15}, C_{25}, C_{35}, C_{45}| = 3$. The maximum clique among these vertices therefore has size 4, the highest number found during the running of the algorithm.

Now let us think of what is possible if the input is given in adjacency matrix form. If G is really a visibility graph, there must be some vertex v such that the neighbors of v can be ordered so that if $n_1 < n_2 < n_3$ are adjacent to each other, the neighbors of n_1 and n_2 which precede n_1 in the ordering are all neighbors of n_3. Let us call such an ordering of the vertices a triangle-extendible ordering. If you are given a triangle-extendible ordering of the neighbors of v, then you can calculate the maximum clique containing v using the dynamic programming solution described above, whether or not G is a visibility graph. Thus, you can

convince me of the size of the maximum clique in a visibility graph in polynomial time; successively show me triangle-extendible orderings of the neighborhood of some vertex v, calculate the maximum clique containing v, and eliminate v from further consideration. Thus, even though we cannot be convinced that G is a visibility graph, we can be convinced of the size of the maximum clique in a visibility graph; the clique problem is not likely to be NP-complete for this class even if input is given in adjacency matrix form. Of course, if we could find a triangle-extendible ordering for a graph G in polynomial time, we could find a robust optimization problem for maximum clique in visibility graphs in polynomial time, so we pose the following open problem.

Open Problem 14.1: Can you determine whether a set of vertices has a triangle-extendible ordering in polynomial time?

Note that triangle-extendible orderings can also be viewed as related to transitive orientations. A graph is transitively orderable (i.e., a comparability graph) if and only if the there is an ordering such that for each edge $i \rightarrow j$ where $j > i$, $\mathrm{N}(i)$ contains $\mathrm{N}(j)$. To calculate the maximum clique C_j including j and vertices before j, we simply take the maximum C_i such that $i \rightarrow j$ and add 1; this can be viewed as a simplification of the triangle-extendible algorithm choosing the maximum $C_{ij}+1$ such that i, j precede k and are adjacent to k. In other words, a transitive orientation allows each edge $i \rightarrow j$ to increase the size of a previous clique, while triangle-extendible orderings allow each triangle to increase the size of a previous clique.

Note that it is only the neighborhood of some vertex in a visibility graph which has a triangle-extendible ordering, rather than the graph itself. It also interesting to study the class of graphs which have a triangle-extendible ordering for the entire vertex set; this class includes comparability graphs.

We could also generalize from triangle-extendible to k-clique extendible for any fixed k. An ordering is k-clique-extendible if for every x and every k-clique K which comes after x in the ordering, if x is adjacent to every vertex in K then then x is adjacent to all joint neighbors of K which come after K in the ordering. For any fixed k, there is a polynomial time dynamic programming algorithm which solves the maximum clique problem given a k-extendible ordering. This gives rise to many possible questions; we might consider the k-clique-extendibility of G as a graph theoretic parameter, for example. Resolving the question of whether the clique problem is solvable on triangle-extendible graphs given only the adjacency matrix would be interesting, whether the problem proves to be NP-complete or polynomial. If it is polynomial, we have a very large class for which the clique problem can be solved efficiently, and get a clique algorithm for visibility graphs even if the model is not part of the input. On the other hand, if the clique problem is provably difficult, we have found a problem which is solvable given a 'representation' but is intractable if none is given.

Open Problem 14.2: Is there a polynomial algorithm for the clique problem in triangle-extendible graphs? Is there some k such that the clique problem is provably intractable for k-clique-extendible graphs?

It might also be interesting to consider how this parameter behaves for random graphs, i.e. for what value f(n) is a random graph on n vertices likely to be f(n)-clique-extendible? This could be of interest, since if f(n) is small then it is possible to prove that a clique in a random graph is of maximum size.

It is also interesting to think of what is possible if only the ordering of endpoints in the polygon is given, rather than a complete model. In terms of size of the representation, of course, we are much more comfortable with being given the ordering than being given the coordinates of each corner of the polygon. Recall that for recognizing visibility graphs, we do not know how to use a known order of endpoints to help us with the recognition or construction problem. However, being given the order of endpoints is entirely sufficient to solve the clique problem in visibility graphs. There must be some vertex v such that the ordering on neighbors of v is a triangle-extendible ordering; we then remove this vertex and continue. Since we can easily test whether an ordering is triangle-extendible in polynomial time, this gives us a polynomial algorithm for the clique problem on visibility graphs if only the endpoint ordering is known.

A further restriction on triangle-extendible orderings is given in [2]. An ordering is inversion-free if for every pair of edges (i,j), (x,y) with $i < x < j < y$, (i,y) is also an edge. To see that this is a restriction, note that saying an ordering is triangle-extendible is equivalent to saying that for every $K_4 - e$, the nonadjacent vertices are not the pair of vertices on the outside of the ordering. Placing nonadjacent vertices from a $K_4 - e$ on the outside of the ordering is also impossible in an inversion-free ordering, but there are also restrictions on how subgraphs which do not induce $K_4 - e$ are placed. The neighborhood of the leftmost vertex of the polygon has an inversion-free ordering; this implies that finding an inversion-free ordering in polynomial time would be sufficient for solving the clique problem on visibility graphs even if the model is not given as input.

A polygon is a convex fan if there is a convex vertex which sees all other vertices of the polygon. An ordered cycle $c_1 ... c_k$ is a cycle in which each c_i comes after c_{i-1} with respect to an order. A Hamilton cycle of a graph G is the border of a visibility graph of a convex fan if and only if it is inversion-free and contains no chordless ordered cycle of length > 3 [2]. Since the neighbors of a convex vertex induce a convex fan, this implies that there is a vertex v in a visibility graph such that N(v) is inversion-free.

Inversion-free graphs are an interesting class for representation; there are $2^{O(nlog^2 n)}$ inversion-free graphs, and there is a representation using O($log^2 n$) bits per vertex, as shown below.

We give a representation of any inversion-free graph. Assume that we are given an inversion-free order, with vertices numbered from 1 to n in the order. Successively divide the order in two, and for each x, store the 'closest' vertex in the other half which is adjacent to x (that is, the earliest vertex if x is in the first half and the latest if x is in the second half). Recursive splitting will give us log$n$ numbers stored at each vertex. To determine whether x and y are adjacent, we find the first split which separates x from y (this can be determined from the value of x and y), and look at the numbers stored at x and y for this split. Assuming that x is in the first half of the split, x and y are adjacent if and only if x is stored at y, y is stored at x, or x stores a number less than y while y stores a number greater than x; the fact that the ordering is inversion-free implies an edge in the latter case.

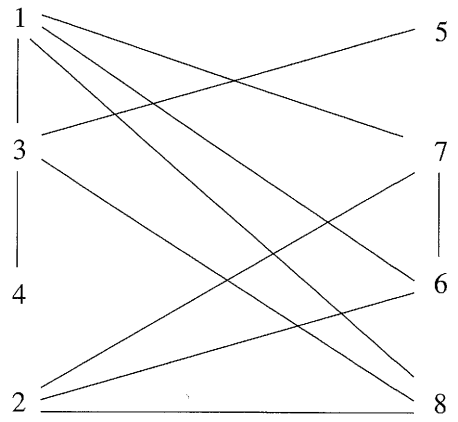


Figure 14.5 Graph with Inversion-Free Representation

An example of this storage scheme is given in figure 14.5, where the inversion-free ordering is 1,2,3,4,5,6,7,8. We use 0 to mean that there is no edge to the other set. For example, 3 is given the tuple 4,1,5; this means that 4 is the first neighbor of 3 from the set 4, 1 is the last neighbor of 3 from the set 1,2, and 5 is the first neighbor of 3 from the set 5,6,7,8. We can deduce that 2 is adjacent to 8 by the presence of the edges (2,6) and (3,8) which are given directly in the representation.

The question as to whether this is a generalized implicit representation is open, since the counting problem is not resolved for inversion-free graphs. A natural way to try to reduce the size of the representation is to use a notion of 'nonredundancy' similar to that used in the chapter dealing with chordal bipartite graphs; an edge (x, y) of an inversion-free ordered graph is redundant if there is a pair of vertices u, v such that $x < u < v < y$, with x adjacent to v and u adjacent to y. For example, in figure 14.5, the edge (1,6) appears in our representation but is redundant; it can be deduced from the existence of edges (1,3) and (2,6).

The argument above shows that there are $O(n\log n)$ nonredundant edges; together with an argument from exercise 14.17 showing that G can be reconstructed from the set of nonredundant edges, any improvement on the upper bound on number of such edges would lead to a better upper bound on the number of inversion-free graphs. Unfortunately, exercise 14.18 shows that there are inversion-free graphs with $\Omega(n\log n)$ nonredundant edges, so this approach will not close the counting problem.

We close the discussion by posing the obvious major open problem, which I feel should be answered affirmatively.

Open Problem 14.3: Is there a robust polynomial time algorithm for the clique problem on visibility graphs?

14.2.2 Coloring Perfectly Orderable Graphs. Perfectly orderable graphs arose as a result of an interesting observation of Chvátal. Papers on subclasses of perfect graphs work on finding the special technique which allow the particular class to be optimized effectively. For example, a perfect elimination ordering can be found for a chordal graph, and this allows many optimization problems such as coloring to be solved in a straightforward, greedy manner if given a perfect elimination

scheme. Similarly, one can color a comparability graph easily if one is given a linear extension of the transitive orientation. Chvátal thought of generalizing this notion of coloring greedily from a simple ordering scheme as far as possible, while keeping the graph class hereditary and perfect.

A greedy coloring is one of the simplest and most natural approaches to graph coloring. Color vertices in some order; let x be the next vertex to be colored. Assign x to have the smallest color which is not used by any previously colored neighbor of x. Given an ordering, one can perform greedy coloring in $O(n+m)$ time; in general, greedy coloring can have very bad performance, using $\Theta(n)$ colors for a 2-colorable graph (exercise 14.5).

G is perfectly orderable if there is an ordering of the vertex set such that for every induced subgraph H of G, the number of colors used by the greedy algorithm, with order of vertices chosen according to the ordering scheme, is equal to the size of the maximum clique in H. Perfectly orderable graphs form a large subclass of perfect graphs, including the chordal graphs and comparability graphs; for a survey on perfectly orderable graphs, see [261].

We will call the ordering scheme associated with a perfectly orderable graph G, along with the adjacency matrix for G, a representation of G. Given this representation, we can solve the coloring problem on perfectly orderable graphs efficiently; simply use the greedy coloring algorithm with this ordering.

It is not immediately clear that this solution can be trusted; what would happen if the representation was not correct, that is, the ordering is not a perfect ordering? Note that this sort of question does not arise with, for example, a representation of a boxicity 2 graph; the difference may have something to do with why we are less comfortable with calling this a representation of the graph. Nevertheless, we can verify that an ordering is a perfect ordering efficiently, as described below, and thus we can trust the result of a coloring algorithm if we are given a perfect ordering.

The characterization of a perfect ordering follows the pattern of making a simple observation on what must be true for a perfect ordering, and proving that this is all that can stop an ordering from being perfect. Suppose that there is a P_4 1,2,3,4 such that the ordering places 1 before 2 and 4 before 3. If we color the P_4 according to such an ordering, vertices 1 and 4 will get color 1, either 2 or 3 (whichever is earlier in the ordering) gets colored 2, and the other vertex receives color 3. This is clearly not an optimal coloring, so no P_4 1,2,3,4 can have 1 before 2 and 4 before 3 in the ordering. [104] shows that an ordering is a perfect ordering if and only if we can avoid this ordering on all P_4s; this gives us an efficient test for whether an order of the vertices is a perfect order. However, we cannot find a perfect ordering efficiently, since recognizing perfectly orderable graphs is NP-complete [369].

There is a sense in which we can color perfectly orderable graphs efficiently even if the graph is given in adjacency matrix form. There is a polynomial time algorithm for coloring perfect graphs, and perfectly orderable graphs are a subclass of perfect graphs. In fact, there is a robust algorithm for coloring perfect graphs; the algorithm will color a perfect graph optimally, and if the input is imperfect will either color it optimally or declare that the graph is not perfect; this can be done despite the fact that there is no polynomial time algorithm known for recognizing perfect graphs. The algorithm for coloring perfect graphs can be found in [237]; depending on one's perspective, it is one of the most wonderful and/or horrible algorithms ever designed in graph algorithms. This algorithm was a major achievement, using deep techniques to solve coloring on a very large class of graphs

which still cannot be recognized efficiently. However, the algorithm is unusually complex; the algorithm involves nested calls of ellipsoid type algorithms developed for solving linear programming problems. This author is also amazed by some of the calculations the algorithm performs; for example, it is hard for me to understand how eigenvalues arise when we are dealing with problems such as clique or coloring. Thus, while the polynomial algorithm is very interesting, it does not resolve the question entirely; we would also like an algorithm which is more combinatorial in nature, as well as having better time bounds. We would like such an algorithm for general perfect graphs, but that would involve a major breakthrough. Given that the problem can be solved efficiently given our representation of perfectly orderable graphs, we pose the following problem.

Open Problem 14.4: Design an efficient, combinatorial algorithm to color perfectly orderable graphs, if the input is in adjacency matrix form.

As far as the author knows, no work has been done on the question of studying whether the algorithm of [237] itself can be simplified to run more efficiently on subclasses of perfect graphs, such as perfectly orderable graphs. The analogy would be with the simplex algorithm for linear programming; no variant of simplex is known to be polynomial in the worst case, but it can be shown that simplex runs in polynomial time for various types of linear programs which correspond to natural graph theoretic questions.

14.3 Open Problems for Robust Algorithms

In this section, we collect some open optimization problems. Some of these problems are known to have a solution if the input is given with a nice representation, and are open if no such model is given; others are open even if a representation is provided.

Perhaps the most natural open question of this form is the clique problem on intersection graphs of line segments in general position on the plane. Kratochvíl and Nesetril show that this can be solved efficiently if we restrict the segments to any fixed set of k angles on the plane, but the general problem is open. This problem is open both if the segments are given as part of the input, and if the graph is given in adjacency matrix form.

More generally, one can consider the clique problem on the intersection graph of convex objects on the plane. This problem is NP-complete if the number of faces is allowed to be unbounded [323], but is open if the number of sides in the convex object is bounded.

In chapter 11, we discussed the notion of clique-width of a graph. The original papers solving optimization problems on clique-width k graphs assume that a representation is given as part of the input; for some problems such as independent set and k-colorability posed in a very restricted form of second order monadic logic, this dependence on the representation is eliminated in [57]. However, there are many problems on the class, including such problems as dominating set which can be posed in a somewhat more general form of second order monadic logic and solved using the techniques of [135], and problems such as Hamilton circuit [478] and general coloring [313] which can be solved on clique-width k graphs given the decomposition tree and are open if the input is given in adjacency list form.

Tolerance graphs are discussed in chapter 4. Vertices are associated with both an interval and a tolerance, and two vertices are adjacent if the overlap between

them is larger than the tolerance of either vertex. A tolerance representation is called bounded tolerance if the tolerance of every vertex is smaller than the length of its interval. G is a bounded tolerance graph if and only if it can be modeled as a trapezoid graph in which all trapezoids are parallelograms [186]; the question of whether such a representation can be found in polynomial time remains open. The paper [296] describes an algorithm for finding the minimum dominating set in the complement of a bounded tolerance graph in polynomial time, if the parallelograms are given as part of the input.

Cai [86] studies the problem of coloring a graph which is close to being in a tractable family for the coloring problem. A set of k edges which can be added or deleted to a graph G to obtain a graph G' in a class C is called a modulator of size k. Cai shows that if a class C can be colored in polynomial time and is closed under edge contraction, it is also possible to color any graph G optimally in polynomial time if we are given a modulator of k edges such that the resulting G' is in C. If the class C can be recognized in polynomial time, we can try all sets of k edges and test whether any of these form a modulator. However, this is not possible when the class cannot be recognized in polynomial time. If we try all possible sets of k edges to add to G, and keep the best coloring found on any of these, we cannot be sure whether the graph does have a modulator of size k and therefore the coloring is optimal, or whether the graph has no modulator of size k and the coloring found might use more colors than is necessary. It is not known whether graphs with small modulators can be colored efficiently, if the modulators are not given as part of the input.

Several algorithms have been designed to solve problems on graph classes given a model, where the recognition problem for the corresponding graph class has never been studied. We focus on a few results regarding classes of overlap representation; recall that two sets overlap if they have a nonempty intersection, and neither set contains the other. The maximum clique problem has been solved for the classes of circular-arc overlap graphs [293] and rectangle overlap graphs [408] when the models are given as input. I know of no work regarding the recognition algorithm for the corresponding classes. It might be of interest both to determine the complexity of finding these models, and if it is difficult to see whether the solutions to the clique problem really require the model as input. More generally, as discussed in Chapter 3, it is possible to solve maximum independent set on an overlap model if you can solve the weighted independent set problem on the corresponding intersection model. This result assumes that the model is given as input; it would be very interesting if the general theorem could also be proved without having access to the model.

An algorithm for solving the independent set and clique problem on interval filament graphs if the model is part of the input is given in [213]; the techniques are generalizations of the algorithm for circle graphs described in Chapter 3. There is no work on whether interval filament graphs can be recognized, and their model constructed, in polynomial time.

In addition to graph problems, there are some matrix algorithms similar to those discussed in Chapter 10 of this form which seem to be open. We discuss the connectivity of 0/1 matrices in that chapter; a matrix is k-connected if the columns can be permuted so that for any pair of rows, the difference of the two rows changes sign at most k times. There are algorithms for solving NP-complete problems on 2-connected matrices in polynomial time [10]; however, no polynomial time

recognition algorithm is known for determining whether a matrix can be permuted to have connectivity 2.

There are many optimization problems which strike me as natural to study, for which no results are known either with or without being given a representation. Selecting some of my favorite problems at random, let me mention the clique problem on disk graphs (rather than unit disk graphs studied above), the clique problem on interval number 2 graphs, and the domination problem for partial orders of dimension three.

Although the focus above has been on questions as to whether polynomial robust algorithms exist, robust algorithms are also important for polynomially solvable problems. One problem which seems to fit particularly nicely within the framework of this book deals with finding a dominating pair in an AT-free graph; a dominating pair is a pair of vertices x, y such that every path from x to y is a dominating set. The paper [125] shows that if a graph is AT-free, a dominating pair can be found by starting at any vertex, performing a lexicographic breadth first search as described in Chapter 13 ending in a vertex y, performing a second lexicographic breadth first search from y ending in a vertex z, and y, z will be a dominating pair. Since LBFS runs in linear time, this is a linear algorithm if we are given a promise that the input graph is AT-free. However, the fastest algorithm known for recognizing AT-free graphs and for determining whether u, v is a dominating pair both involve matrix multiplication [332], so there is no obvious method to eliminate dependence on the promise and create a robust linear time algorithm for the dominating pair problem on AT-free graphs.

14.4 Complexity Issues

It is known that many special representations are NP-hard to find; examples include small interval number representations, disk representations, and many others. Since the representations appear to be quite useful, it may seem surprising that we are not easily able to find problems which are polynomially solvable if given the representation, and provably intractable without the representation.

Suppose that a class of graphs has an easily verifiable representation, and a decision problem can be solved in polynomial time with the representation given. This implies that if the input is in the class of graphs, there is always a short proof of either a yes or no answer to the question; one can provide the representation, and run the algorithm on the model. Thus, given standard complexity assumptions, we will not be able to find a reduction from satisfiability which produces a graph in the class with output yes if the formula is satisfiable, and produces a graph in the class with output no if the formula is not satisfiable; such a reduction would imply that NP = co-NP. This has been the standard notion of being NP-complete on a graph class. We call problems which are difficult in this sense classically NP-complete on a class of graphs

Even with respect to classical NP-completeness, graph classes with nice models may be interesting. It is difficult to find problems which are in both NP and co-NP, but not known to have polynomial time solutions; the most famous problem of this type, testing whether a number is prime, was recently shown to be solvable in polynomial time. If you can solve a problem on a class of graphs given a model, the problem is in NP $\cap$ co-NP for the class. Thus, problems on graph classes which

have good models but constructing the model is difficult are good possible sources for difficult problems in NP ∩ co-NP.

We need a new notion of intractability to correspond to our new definition of being robustly solvable. A problem is robust-NP-complete on the class C if there is a polynomial reduction from the satisfiability problem which produces a graph in the class with answer yes if the formula is satisfiable, and produces a graph with answer no (which may or may not be in the class) if the formula is not satisfiable. Therefore, a robust algorithm would have to answer yes if the answer to the original problem was yes, and would give some other answer (either no or G is not in the class) if the answer to the original problem is no. Thus, a robust algorithm for such a problem would imply that P equals NP.

We demonstrate this new form of reduction in the following theorem.

Theorem 14.2 *There is no robust algorithm for the independent set problem on well-covered graphs unless P=NP.*

Proof We reduce from the independent set problem to the independent set problem on well-covered graphs. Let G, k be the input to the independent set problem. We create G' as follows.

Create k copies $v_1, v_2, ..., v_k$ of each vertex v. Let v_i be adjacent to w_j if $i = j$, $v = w$, or v and w are adjacent in G.

Add new vertices x_{ij} for all j not equal to i. The intent of these new vertices is to supplement any independent set to be a maximal independent set of size k-1. Add edges from x_{ij} to all x_{kl} with i not equal to k, and to all v_i and v_j. The meaning of x_{ij} is as a replacement for vertex j in an independent set which includes no v_i or replacement for v_i.

Suppose that G has an independent set of size k. Then there is an independent set of size k in G', taking the first copy of the 1st vertex of the independent set, the second copy of the next, etc. Suppose that G has no independent set of size k. Then every maximal independent set of G' has size k-1. If any vertex x_{ij} is used, then no v vertex with first subscript i can be used; however, we will always be able to supplement any other set to a set of size k-1 by using the appropriate x_{ih} whenever nothing of the form v_h was chosen. If no vertex x_{ij} is used, the vertices correspond to an independent set in G. If fewer than k-1 vertices are in this set, they can be augmented by some x_{ij} where no v_i or v_j is used.

Suppose that we had a robust algorithm for the independent set problem on well-covered graphs. We take our independent set problem on G, and produce G' as described. If our robust algorithm answers that the independent set in G' has size k or that G' is not well-covered, then we know that G has an independent set of size k. If the algorithm answers that G' has an independent set of size k-1, we answer that G has no independent set of size k.

Thus, a robust algorithm for the independent set problem on well-covered graphs is not possible unless P= NP. □

This theorem may seem surprising in light of how simple the problem is if we are given a promise that the input is well-covered. As an aside, we note that the reduction above can also be used to show that recognizing well-covered graphs is co-NP-complete, which was already proved using different reductions [419].

We also need a new complexity class for problems which are difficult if no model is given, but easy if a model is given. We will not define the class formally

here, since it is not clear which assumptions of intractability will prove useful in such a definition. As noted above, we will (presumably) need a different notion of reduction to prove any result of this form; a standard reduction mapping an NP-complete problem to a problem solvable given a model would imply that NP = co-NP. Nevertheless, being given the solution to the NP-hard problem of constructing a model should help in solving some problems; the difficulty seems to be in proving that the problem cannot be solved without the model.

14.4.1 EP Theorems. When I began research on robust algorithms in the late 1990s, I was unable to find any general treatment of the topic, although individual papers had presented robust algorithms for particular problems. I have recently learned that a closely related concept had been introduced as a complexity theoretic model by Cameron and Edmonds [88] approximately ten years previous to my work.[1]

The paper [88] introduces the notion of an EP (existentially polytime) theorem. An EP theorem has the property that every condition of the theorem can be verified in polynomial time. Many theorems which are not initially in EP form can be rewritten to be in EP form. For example, the statement that every comparability graph is a Berge graph is not an EP theorem, since Berge graph recognition is not known to be in NP and thus cannot currently be verified in polynomial time. However, the statement that every graph either is not a comparability graph or has a coloring which uses a number of colors equal to its clique size is an EP theorem, since each of the two conditions can be verified in polynomial time. Given that comparability graphs are hereditary and the strong perfect graph conjecture is true, these two theorems are equivalent. The first statement is false if and only if some comparability graph contains an odd hole or anti-hole, and the second is false if and only if some comparability graph G cannot be colored with clique size equal to chromatic number, which (by the strong perfect graph conjecture) is equivalent to saying that G contains an odd hole or anti-hole.

Although I did not recognize it originally, the reason for preferring EP theorems is tied to the notion of a robust algorithm; if there is an EP theorem, we can hope to find an algorithm which produces as output one of the (easily verifiable) conditions of the theorem. In many ways, EP theorems are to robust algorithms exactly what NP $\cap$ co-NP is to polynomial time algorithms. If there is a polynomial time algorithm for a problem, the problem is in NP $\cap$ co-NP, and showing problems are in NP $\cap$ co-NP is used as an important step in finding polynomial algorithms. Similarly, any robust algorithm corresponds to an EP theorem (we can say either the stopping condition which allows the algorithm to answer that the graph is not in the class is satisfied, or the output of the algorithm is correct, and the robust algorithm itself gives a polynomial proof of one of these two conditions), and the existence of an EP-theorem leads us to search for a robust algorithm (which has been called a constructing polytime algorithm [429] and an EP search problem [89] in the context of EP theorems) which produces one of these conditions.

[1]Actually, Kathie Cameron had provided me slightly earlier with a copy of this paper, but I did not grasp the connection with robust algorithms until a memorably enjoyable day at the 2002 SIAM Discrete Mathematics meeting. Jack Edmonds took a group of us to the San Diego beach with his family, and explained the connection to a group of us at some length. Jack and Kathie's remarkably good natured children had no problem with this type of adult discussion, but I think that some of the laid-back surf San Diego surfing community must have found the sight of a passionate mathematical discussion very unusual!

If this is done carefully, the concept of a robust algorithm can be defined entirely in terms of EP theorems, and this predates my work on robustness. As a student of algorithms rather than complexity, I find the algorithmic justifications for robustness more natural, but Edmonds has pointed out that from a complexity theory point of view the EP formulation seems to be more natural. There is a symmetry between the conditions for an EP theorem, while there is an asymmetry between the problem solved and the input domain for a robust algorithm which can be misleading in some cases. As an example, consider the problem of matching in a claw-free graph with an even number of vertices; we gave a robust algorithm for finding a perfect matching in an even vertex claw-free graph. When stated as an EP theorem, we get symmetric conditions; every graph on an even number of vertices either has a perfect matching or a claw. In fact, our algorithm can also be used as a robust algorithm for finding a claw in a graph without a perfect matching, but this is not clear from our original statement.

Whether we prefer to state results as robust algorithms or EP theorems, I agree with Cameron and Edmonds on the importance of the concept. I am happy to acknowledge the earlier work and will let the community decide which name and formulation should be used to describe these algorithms/theorems.

14.5 Exercises

1) Show that the number of well-covered graphs is $2^{\Theta(n^2)}$.

2) The fact that the independent set problem on circular-arc graphs can be solved more efficiently than recognition may not be surprising if viewed in the following light. Design a simple $O(n+m)$ algorithm to find an independent set which has cardinality at most 1 smaller than the largest independent set in G, if G is a circular-arc graph. Prove that your algorithm is a robust algorithm.

3) The following subproblem arises in the independent set problem for circular-arc graphs if G - N(v) is a clique. Design an O(n+m) algorithm to determine whether a circular-arc graph contains an independent set of size 3, without being given the representation.

4) Show that for any graph G, there is an ordering of G such that greedy coloring with this ordering produces an optimal coloring.

5) Greedy coloring can perform very badly indeed if the ordering is not good. Show that there are orderings of 2-colorable graphs for which the number of colors used is $\Omega(n)$.

6) Show that every EPT graph has a tree model on a tree with $O(n)$ nodes.

7) In an EPT representation on a star, it is possible to have paths consisting of a single edge. Prove that every EPT graph of a star has a representation in which every path has length two.

8) Independent set on comparability graphs is somewhat harder than the clique problem. The algorithm, first presented in [198] for partial orders, works as follows. Transitively orient G; call the result G'. Create a bipartite graph $B(G')$ by creating two vertices v_{in}, v_{out} for each vertex v, and add an edge from v_{out} to w_{in} if and only if $v \rightarrow w$ in G.

A maximum independent set in a graph is exactly equal to the vertices not contained in a minimum vertex cover, and the minimum vertex cover in a bipartite graph can be found in linear time from a maximum matching [198]. The following problems show that independent set in comparability graphs is no harder than bipartite matching. We note that determining the size of the maximum independent set in a comparability graph set is as hard as determining the size of a maximum matching in a bipartite graph, since every bipartite graph is a comparability graph. The construction used in this problem can be used to determine whether a comparability graph has an independent set of k for fixed k in $O(n^2)$ time; see chapter 8.

8a) Prove that if G is a comparability graph, the optimal vertex cover in $B(G')$ will never use both v_{in} and v_{out}.

8b) Prove that the optimal vertex cover in $B(G')$ is an optimal vertex cover in G.

9) A graph G is brittle if in every induced subgraph of G, either there is a vertex which is not in the center of a P_4 or there is a vertex which is not at the end of a P_4. Show that all brittle graphs are perfectly orderable.

10a) Analyze the time complexity of the algorithm for finding a maximum clique in a visibility graph given in section 14.2.1.

10b) Show that we can get the clique itself as well as the clique size in same bound given in part a.

11) Show that you can solve the clique problem on induced visibility graphs if you are given the ordering of endpoints in the induced visibility model, and which points of the model are visible to each other.

12a) Show that the following 10 vertex graph is not triangle-extendible. Consider a 4-clique and a 6-clique, where the vertices of the 6-clique are nonadjacent exactly to different pairs of vertices from the 4-clique. Richard Borie wrote a computer program which found this and other 10 vertex graphs which are not triangle-extendible, and proved that every graph with fewer than 10 vertices is triangle-extendible.

12b) For any $k > 2$, let X be a clique of size $2(k\text{-}1)$, and let Y be a clique consisting of distinct vertices nonadjacent to each possible pair of vertices from X. Show that the graph constructed in this fashion is not k-clique extendible.

13) Show that the class of co-bipartite triangle-extendible graphs has an implicit representation.

14) Show that every inversion-free ordering is triangle-extendible.

15) Give an example of a graph which is triangle-extendible, but not inversion-free.

16) A convex graph is a bipartite graph with color classes X and Y in which the vertices of X can be ordered so that neighbors of every vertex y in Y occur consecutively. Show that every convex graph is inversion-free.

17) An edge (v, y) of an inversion-free ordering is redundant if there are vertices $v < w < x < y$ such that (v, x) and (w, y) are adjacent. Show that an inversion-free graph G can be reconstructed in polynomial time from the ordering, and the set of nonredundant edges for this ordering.

18) Show that the graph formed in the following way has $\Omega(n \log n)$ nonredundant edges, in the sense of exercise 14.17. It is interesting to note that this graph is also the key to exercise 6.2, in that it can be drawn with all nonredundant edges as pseudolines which touch the outer visibility envelope.

We start with 2 nonadjacent vertices. To get a graph of size $2k$ from 2 graphs of size k, create the ordering $x_1 \ldots x_k \; y_1 \ldots y_k$ and add edges from each x_i to each y_i. This will give a graph G_1 with $\Omega(n \log n)$ edges. Let G_2 be formed by adding all edges to G_1 which are necessary in order to make the result inversion-free.[2]

Show that all edges of G_1 are nonredundant in G_2.

19) In the definition of robust NP-completeness, explain why the following type of reduction was not allowed: Given a yes instance, the transformation must produce a graph in the class for which the answer is yes, and given a no instance the transformation must produce either a graph which is not in the class or a graph in the class with answer no.

20) A tolerance graph has bounded tolerance if the tolerance of an interval is never larger than the length of an interval. Show that G is a bounded tolerance graph if and only if G can be modeled as a trapezoid graph in which all trapezoids are parallelograms.

21a) The class of curve contact graphs is discussed in [257]. In these graphs, vertices correspond to noncrossing curves in the plane, and two vertices are adjacent if and only if an end of one curve touches another curve. It can be shown that any clique corresponds either to a set of curves touching a single point, or a set of curves which touch at least two of three contact points [257]. Use this fact to show that there is a robust polynomial time algorithm for the maximum clique problem on curve contact graphs, even if the graph is given in adjacency list form.

21b) Show that there are $2^{O(n \log n)}$ curve contact graphs on n vertices.

---

[2] tournament chess players may note that G_1 corresponds to the games played in a Swiss system tournament with logn rounds and no upsets

22) Exercise 6.3c discusses a necessary condition for a visibility graphs. This is the notion of a blocking vertex; for each nonadjacent pair of vertices u and w, there must be some v such that as we traverse the polygon from u to w through v, every vertex on the path from u to v is nonadjacent to every vertex on the path from v to w. This blocking vertex can be on either of the two paths from u to w on the polygon.

Suppose that we require this condition for a linearly ordered set of vertices rather than a circularly ordered set of vertices. An ordering $x_1x_2...x_n$ of the vertices of G is a blocking ordering if for every pair of nonadjacent vertices x_i, x_k with $i < k$, there is some vertex x_j such that all vertices $x_ix_{i+1}...x_{j-1}$ are nonadjacent to the vertices $x_{j+1}...x_k$. Show that an ordering is a blocking ordering if and only if it is an inversion-free ordering.

23) Prove that the complement of a chordal bipartite graph is perfectly orderable if and only if G is chordal bipartite. Hint: the ordering follows very simply from a Γ-free ordering.

CHAPTER 15

Characterization and Construction

The primary focus of this book has been on representation of graphs. In this chapter, we look at some issues in the closely related construction and characterization problems which have been ignored in other general works on classes of graphs. To see the difference between the problems, note that we have completely ignored up to now classes of graphs which have $2^{\Omega(n^2)}$ members, since the adjacency matrix gives a space optimal representation for which adjacency testing can be done in constant time. This representation can easily be made generalized implicit, which has been our goal in most of the book. However, some of these classes have forms of representations which are well studied and can lead to efficient algorithms; prominent questions about some such classes are studied in this chapter.

15.1 Chordal Graphs

Chordal graphs, as defined in Chapter 3, are graphs in which every cycle of length > 3 has a chord. It is easy to see that the number of chordal graphs is $2^{\Omega(n^2)}$. Take any bipartite graph, and make one of the color classes a clique (graphs which can be partitioned into an independent set and a clique are called split graphs; see [223]). The result is chordal; a proof is left to the reader (exercise 15.1). Since the number of bipartite graphs is $2^{\Omega(n^2)}$, the number of split graphs and chordal graphs is $2^{\Omega(n^2)}$.

[476, 216, 84] proved that chordal graphs are exactly equal to the intersection graphs of subtrees of a tree. This is an example of a nice characterization which is not at all good as a representation, if the goal is to minimize storage space and test adjacency efficiently.

There are several ways to think of storing a graph as the intersection graph of subtrees of a tree T. For example, we might store T as a global structure, and the leaves of the subtree corresponding to vertex v at v. It can be shown that we can choose a host tree T which has at most n nodes, but adjacency testing will still take $\Omega(n)$ time in the worst case using this storage method. In addition, the storage method does not seem to be space optimal; we could potentially be forced to store $\Theta(n)$ numbers with each vertex v, where each number uses $\Theta(\log n)$ bits, thus using $\Theta(n^2 \log n)$ bits overall. It is not immediately obvious that there is any chordal graph such that this storage method actually requires $\Theta(n^2 \log n)$ bits; this does not seem to me to be a crucial open question to resolve, since there are clearly better ways of storing a chordal graph if optimizing space and time for adjacency testing are our goals.

Nevertheless, these trees are important to construct for solving certain optimization problems and for dealing with subclasses of chordal graphs. We have seen that they are useful in finding an implicit representation of chordal comparability graphs in chapter 5. The tree constructed is sometimes called the clique tree, since

nodes of the tree correspond to maximal cliques in G. Note that the algorithm also enumerates all maximal cliques in the graph in linear time.

We now give an algorithm for constructing the clique tree of a connected chordal graph in linear time; we believe this is simpler than standard algorithms for this problem such as the one described in [223]. In particular, it is much simpler to see how this algorithm constructs the tree in linear time. The first algorithm for constructing an intersection model which clearly takes linear time is given by Hsu and Ma in [268]. Their algorithm is tied closely to lexicographic breadth first search, while the algorithm here creates a clique tree from any perfect elimination scheme. Assume that vertices are labeled from 1 to n according to their position in a perfect elimination scheme.

We construct the clique tree for the graph induced on vertices i through n for all vertices, starting with $i = n$ and ending with $i = 1$. Let C(v) be the clique consisting of v and all neighbors of v which appear after v in the elimination scheme. After each vertex v is processed, v is given a pointer to the clique C(v) in the tree. We note that vertices may be added to this clique later in the algorithm, but v will always point to a clique which contains C(v).

Let i be the next vertex considered, and assume we know the clique tree on the graph induced by vertices i+1..n. We need to add C(i) to the clique tree. Let j be the first vertex of C(i) in the elimination ordering, other than i itself. If $|C(i)| = 1 + |C(j)|$, and the clique pointed to by j is equal to C(j), we add i to this clique; in other words, C(i) replaces C(j) in the tree. Otherwise, add C(i) as a new node of the tree. Connect C(i) to the tree by adding an edge from C(i) to the clique pointed to by j.

It is easy to make the algorithm above work in linear time, since we make one decision at each vertex, and we can decide whether the clique pointed to by j is equal to C(j) by simply checking the number of elements in this clique.

To see that the algorithm is correct, it is sufficient to look at two cases. Either C(j) is a maximal clique on the elements induced by vertices i+1..n, or it is not. If C(j) is a maximal clique, it clearly must be replaced by C(i) if C(j) is contained in C(i), which occurs if C(i) equals C(j) $\cup$ i, and the algorithm does this correctly. If C(j) is not a maximal clique, or C(i) does not contain C(j), then C(i) cannot contain any maximal clique on the elements i+1 through n, and must be added as a new node. All elements of C(i) - i are in the clique pointed to by j, so the subtrees induced by all elements remain connected.

We illustrate the algorithm on the graph of figure 15.1, and the perfect elimination scheme 1..8. We begin with a single clique 8, which 8 points to. Vertex 7 finds 8 as its first neighbor; since 7 has 1 later neighbor and 8 points to a clique of size 1, we add 7 to the clique. Similarly, 6 has 2 later neighbors and 7 points to a clique of size 2, so we have a clique 6,7,8 which is pointed to by all these vertices. Vertex 5 has 2 later neighbors, but 6 points to a clique of size 3, so we add a clique 5,6,8 pendant to the clique 6,7,8. Vertex 4 has 2 neighbors and 5 points to a clique of size 3, so 4,5,6 is added with an edge to 5,6,8. Since 3 has 2 later neighbors and 4 points to a clique of size 3, we add a clique 3,4,5 pendant to 4,5,6. Vertex 2 has 2 later neighbors and 7 points to a clique of size 3 (note that this clique had size 2 when 7 was added, but 6 is now part of the clique pointed to by 7), we add a clique 2,7,8 pendant to 6,7,8. Vertex 1 has 2 later neighbors and 7 points to a clique of size 3, so we add the clique 1,7,8 pendant to 6,7,8, giving the clique tree of figure 15.1.

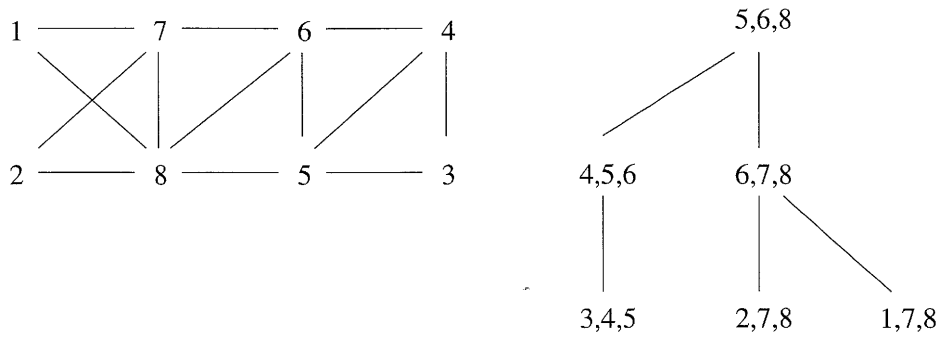


Figure 15.1 Constructing a Clique Tree

It is easy to incorporate the standard algorithm for verification of a perfect elimination scheme into this framework. Suppose that the input is not a perfect elimination scheme. Let i be the last vertex such that neighbors of i are not a clique, and let j be the first neighbor of i which appears after i. We will find that the neighbors of i are not contained in the set of neighbors of j. Therefore, when each vertex i is processed, we check that $N(i)$ is a subset of $j \cup N(j)$; if it is, we add the clique to the clique tree, and if not we declare that the input is not a perfect elimination ordering. Since this test takes $O(N(i))$ time, we can either construct a clique tree for G or answer that the input is not a perfect elimination ordering in $O(n+m)$ time.

15.2 Unit Interval Graphs

In Chapter 13, we saw that G is a unit interval graph if and only if G is a proper interval graph, and that these graphs are exactly equal to interval graphs with no induced $K_{1,3}$. We then gave two linear time recognition algorithms for the class, based on determining whether an interval model contains $K_{1,3}$, and finding proper interval models. Finding unit interval models for unit interval graphs is not always stated clearly, so we provide a construction algorithm here.

In chapter 13, we described a method for turning a proper interval representation into a unit interval representation. We process intervals in the natural left to right order as given in the proper interval model, and place each new interval i just to the right of the rightmost interval which is nonadjacent to i, and to the right of any intervals which have already been placed. Although this algorithm is conceptually simple, it neglects several crucial points if we actually want to construct a representation.

Suppose, as is usually the case, that we want to place all endpoints of intervals at integers on the line. Suppose that we place the first interval a starting at position 1, and the second interval i_2 starting at position 2. If all endpoints are integers, we will have a problem if there are two nonequivalent vertices which are adjacent to i_2 but not to i_1; both will have to be placed between the right endpoint of i_1 and the right endpoint of i_2. The most natural way to solve this problem is to place any new endpoint halfway between the bracketing endpoints. However, to make all endpoints integers, this would require the length of an interval to be exponential. This destroys the notion of an efficient representation. To get a space efficient

representation, we need to find a representation in which intervals have polynomial size, so that the number of bits used for each interval is $O(logn)$. If integer sizes are exponential, we cannot truly call the running time of the algorithm linear, since manipulating numbers in the range 1 to 2^n requires $\Theta(n)$ time.

We now present an algorithm which constructs a unit interval model from a proper interval model which has the property that all interval endpoints are integers in the range 0 to $O(n^2)$. The running time of an algorithm constructing this representation is $O(m+n)$. The proof of the correctness of the algorithm can be found in [124]. It is also possible to construct a unit interval representation which uses intervals of the minimum possible length (subject to the constraint of placing endpoints on integers); see [371] for a description of such an algorithm.

In the first step of the algorithm, we find the leftmost interval. Choose an arbitrary vertex u, and perform breadth first search from u. Let z be a vertex from the last level of the breadth first search which has minimum possible degree. Make z the leftmost interval.

We now do a breadth first search from z. Order vertices from $1..n$ such that all vertices at level i have lower number than vertices at level $i+1$, and vertices in a single level are ordered in increasing order of (neighbors in next level - neighbors in previous level). This step gives a proper interval ordering, if such an ordering exists, by extending each endpoint to the right until we reach a nonneighbor.

Construct a directed tree on the vertices by adding an edge to each vertex v other than z from the earliest neighbor of v in the ordering. Construct a second ordering of the vertices called postorder by performing a postorder traversal where smaller numbered children are visited first; i.e., if w has children x and y with $x < y$, we go down the path from w to x before we go from w to y, and consequently $\text{postorder}(x) < \text{postorder}(y)$.

The unit interval model is now constructed as follows. If v is a vertex from level k of the breadth first search, make the left endpoint of $v = nk + \text{postorder}(v)$ and the right endpoint of $v = nk + \text{postorder}(v) + n$. Clearly, all intervals have length n.

In the graph of figure 15.2, our initial breadth first search may choose 1 as the leftmost vertex z. The proper interval ordering is 1,2,5,3,4; note that 5 has more neighbors at level two than 2, and 3 has more neighbors at level 1 than 4. The tree produced is shown in figure 15.2. The postorder traversal creates the order 3,2,4,5,1. The left endpoints in the model are placed as follows

$$1 : 52 : 73 : 114 : 135 : 9$$

and all intervals have length 5.

The paper [124] shows that if G is a unit interval graph, this is a valid unit interval model. It is easy to determine whether the model is valid in $O(n+m)$ time, so this gives an efficient method for constructing a unit interval model which is an implicit representation.

15.3 Unit Circular-Arc Graphs

In chapter 13, we looked at the recognition problem for unit circular-arc graphs. These graphs were recognized by constructing a proper circular-arc model, and checking whether the model had an obstruction which Tucker [468] proved exists if and only if G is not a unit circular-arc graph. Tucker's proof actually constructs

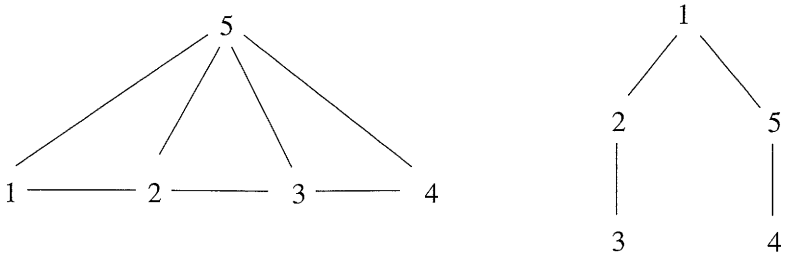


Figure 15.2 Unit Interval Model Construction

a unit circular-arc model, and thus gives an alternative unit circular-arc graph recognition algorithm which we will outline very briefly.

The circumference of the circle is set to a length between the lower and upper bounds dictated by a minimal (with respect to n/k) (n,k)-circuit and maximal (with respect to r/l) (r,l) independent set; these terms are discussed in the recognition chapter for this class.

The algorithm works by successively fixing one more arc A to be unit length, while maintaining a proper circular-arc model. The process involves shrinking or lengthening A; to maintain the proper model, other arcs are lengthened or shortened together with A. However, arcs which have already been set to be unit length must maintain unit length; if such an arc must be modified to maintain the proper circular-arc model, it is rotated rather than shrunk or expanded. The rotation of a unit arc may cause other unit arcs to be rotated; fundamentally, we end up rotating (i,j)-circuits and/or (g,h) independent sets, and the algorithm works because the circumference can handle the bounds forced by any such circuit or independent set.

Tucker does not worry about the size of integers in his construction, since it was intended to be a tool for showing that any proper circular-arc graph which did not contain an obstruction as in chapter 13 is a unit circular-arc graph. This makes analyzing the time complexity of Tucker's algorithm tricky, since manipulation of large integers cannot be assumed to take constant time; sometimes arcs can be handled simply as an ordered list of endpoints, but at other times we must perform tests for length conditions. Therefore, we pose the following open problem.

Open Problem 15.1: Is it always possible to construct a unit (in the sense that all sizes are equal) circular-model with endpoints corresponding to integers of polynomial size? If so, how efficiently can such a model be constructed?

15.4 Construction Problem for NP-complete Classes

In my opinion, there is a type of problem which has been unjustly ignored in the literature. Suppose that a class of graphs with a nice form of representation is studied, and the recognition problem proves to be NP-complete for the class. Typically, this does not stop research on purely graph theoretical questions on the class, as the large body of work on such classes as partial order dimension or interval number shows. However, there is very little work on algorithmic problems on such classes. In particular, the problem of finding a nice representation of graphs in these classes has very rarely been studied; how can one represent them nicely without being given the model if they are NP-complete to recognize?

As an example, let us look at graphs with boxicity 2. The implicit graph question is not interesting for this class, since the definition gives an implicit representation. However, if a graph is given in adjacency list form, we will not be able to find the rectangular representation efficiently. This does not necessarily mean that we cannot find an alternate implicit representation efficiently, however.

The most obvious attempt is to find an approximation algorithm; perhaps we can find an algorithm such that if G has boxicity 2, we can construct a representation of G as boxes in k dimensional space for some fixed k. Since these problems have not been studied carefully, our level of knowledge is very low. Not only do we not know whether we can find a k dimensional representation of a boxicity 2 graph in polynomial time, we do not even know whether we can find a representation as $n^{1/2}$ dimensional boxes in polynomial time; even achieving a bound of $o(n)$ is not known, though it is possible that this can be obtained easily. The analogy here is with graph coloring; when Johnson first examined the problem of finding a coloring for a 3-colorable graph, he found that all known algorithms could use $\Theta(n)$ colors [284] This bound has since been reduced to $O(n^{3/8}\text{polylog}(n))$ [41], and the first improvements used very basic graph theory [485].

However, approximation algorithms are not the only direction of research. If we want an implicit representation, we should not be restricted to a representation which is a variant on boxicity. The situation is similar to the robust optimization problems studied in chapter 14; we might find that boxicity 2 graphs are a subclass of a graph class for which an implicit representation can be found efficiently, or it is possible that we find one of a number of types of implicit representations depending on a choice made by a representation algorithm. Certainly the boxicity 2 graphs have many interesting properties, such as having $O(n^2)$ maximal cliques, which could give rise to an entirely different form of efficient representation.

I do not know of this problem being studied on any class of graphs. In my opinion, finding an implicit representation in polynomial time would be interesting for many classes which are NP-complete to recognize, including boxicity 2 graphs, interval number 2 graphs, partial orders of dimension 3, disk intersection graphs, EPT graphs, and many more. The problem is also interesting for classes which are open with respect to recognition, such as visibility graphs and circle orders. I feel this should be possible for at least some of the classes described above, and is sufficiently important to be posed separately as an open problem.

Open Problem 15.2: Can you construct an implicit representation for graphs in any of the classes given above in polynomial time?

15.5 Exercises

1) G is a split graph if the vertices can be partitioned into a clique and an independent set. Show that every split graph is chordal.

2) Show that every chordal graph is the intersection graph of a tree with all degrees no larger than 3.

3) Show that if we are given a perfect elimination scheme for a connected chordal graph, every vertex v except the last has an edge to a vertex which appears after v in the perfect elimination scheme.

4) In the unit interval model construction algorithm, we find a leftmost vertex z, perform a lexicographic breadth first search from z, and assign an order O_1 at level i by increasing value of number of neighbors at level i+1 - number of neighbors in level i-1. We then create a tree T by making each node a child of its earliest neighbor in the ordering, and assign an order O_2 to vertices based on a postorder traversal of T. Show that if x and y are at the same level of the LBFS, x precedes y in O_1 if and only if x precedes y in O_2.

5) Show that if G is a unit interval graph, the algorithm finds a valid unit interval model. You may assume that problem 4 has been solved.

6) The following approach due to Yiannis Papoutsakis also constructs an implicit unit interval representation efficiently. Start with any proper interval representation. The length of an interval is the sum of the weights of all points between its endpoints. Initially, the points are equal to the endpoints; all endpoints are given weight 1. Consider endpoints in left to right order; we will assume that all intervals to the left of the current interval i have the same length l. If length(i) $<$ length (i-1), add a point with weight length(i-1) - length(i) immediately before the right endpoint of i. If length(i) $>$ length(i-1), add points with weight length(i) - length(i-1) in such a way that every interval to the left of i contains exactly one new point.

6a) Show that the algorithm halts with all lengths of size O(n).

6b) Show that we can make this construction algorithm run in O(n) time if the proper interval representation is given.

7) It is tempting to believe that the maximum length of intervals with all endpoints on integers in a unit interval model can be made no larger than the maximum degree of a vertex plus 1. Show that there are unit interval graphs such that the maximum length - maximum degree is arbitrarily large for every unit interval model of the graph.

CHAPTER 16

Applications

Before dealing with applications of classes of graphs, I must confess that my own interests are not based on applications at all. I enjoy seeing applications of the classes, but have not worked directly on any of them, and I feel that I am more likely to make erroneous statements here than anywhere else in the book. I also feel uneasy talking about applications, after seeing too many papers which continue to describe an application of a graph class long after the relevant problem has become obsolete. Finally, I cannot hope to compete in this chapter with the writing or the scope of applications in Roberts' book [411]; Golumbic [223] also covers applications of many graph classes very nicely. Nevertheless, quite a number of these classes have applications in areas which did not exist when the books mentioned above were written, so I will give some coverage of these new applications.

16.1 Computational Biology

One of the most important historical applications in the field is the relationship between interval graphs and arrangement of genes on the chromosome. Early work, due to Benzer [32], was important both in biology and for stimulating mathematical research on a variety of graph classes. Enormous recent advances and interest in the field of computational biology have made the actual problem Benzer worked on less important, but the connection between gene placement and interval graphs is behind a number of important new problems connecting the fields. Since graph theorists and biologists started talking, they have also discovered other applications of graph theory, some of which will be discussed here.

16.1.1 Genome Reconstruction. This is the area which is the most direct descendant of Benzer's work. When Benzer did his research, the only way to get information about the arrangement of genes on the chromosome was to look at traits which were observable on the individual, usually in terms of mutations; this is clearly not going to get very far if we want to map the human genome, which is a popular research topic among current biologists. Since that time, many tests have been developed which allow examination of pieces of the genetic structure itself.

An important technique at the current time is to cut relatively small, random pieces of the chromosome, and run chemical experiments on the small pieces to get accurate information about these. This allows researchers to get information about fragments and the frequency of types of fragments, but it loses information about how the fragments are ordered on the chromosome. Thus, we want to reconstruct the chromosome from the fragments. Since each fragment should form an interval on the chromosome, it is natural to model this problem as an interval graph problem, with fragments corresponding to vertices.

For this model to be useful, there must be a way to determine which fragments should overlap in the chromosome, corresponding to which vertices should be adjacent in the interval model. Such tests have been developed, which makes it seem simple to use interval graph techniques to solve the gene reconstruction problem.

There is an important problem with the tests for whether fragments overlap, however; the tests may produce errors. Errors come in two forms; the tests may say that fragments overlap when they do not (a false positive test), or they may say that fragments do not overlap when they really do (a false negative). Depending on which test is used and what thresholds are set, false positives may be much more common than false negatives, much less common, or both may occur with relatively equal probability.

This gives rise naturally to a number of graph theoretic problems. If we assume that all errors are false negatives, we want to know the minimum number of edges we can add to make the graph an interval graph. This problem is called the interval graph completion problem. If we assume that all errors are false positives, we want to know the minimum number of edges we can delete from G to get an interval graph; this is called the maximum interval subgraph problem. If errors can be false negatives or false positives, we want to minimize the number of edge additions and deletions so that the result is an interval graph; this is called the interval graph modification problem.

All of the problems above are NP-complete. However, there are additional constraints which may make the problem easier. For example, we know that all of the intervals are of approximately (how close this approximation is depends on the exact test) the same length; thus, it is natural to study the same problems on unit interval graphs as well.

As well as new problems for study, applications can sometimes lead to the introduction of new graph classes. Even if we assume that there are no errors in the tests described above, there is a completely different way to reconstruct the genome which is arguably more efficient. If we have n fragments, and we want to reconstruct the entire interval graph, the method described above calls for $\Theta(n^2)$ chemical tests to determine whether fragments overlap. If we want to minimize the number of chemical tests, it is not hard to see that much of the work is unnecessary; if we know that fragment x is adjacent to fragment y and nonadjacent to fragment z with y to the left of z, we can deduce that x is nonadjacent to any fragment to the right of z.

One method for reducing the number of chemical tests to reconstruct the genome works as follows. A subset of the fragments are selected to be 'probes', and the remaining fragments are called nonprobes. Instead of testing overlap relationships between pairs of fragments, tests are only made between a pair x, y of fragments if at least one of x and y is a probe. The corresponding graph model is called a probe interval model; vertices correspond to intervals on the line, with x adjacent to y if and only if the corresponding intervals overlap, and at least one of x and y is a probe. A recognition algorithm for probe interval graphs assuming vertices are prelabeled as probes and nonprobes is presented in [287]. Although certain parts of the algorithm become more complex than standard interval graph recognition, our understanding of interval graphs is crucial in the design of an algorithm for this problem.

We note that the problem above is sometimes called tagged probe interval graph recognition, since vertices are 'tagged' as being probes or nonprobes. It is

not known whether there is a polynomial time algorithm to determine whether there is a partition of the vertices into probes and nonprobes so that the resulting assignment has a probe interval model.

Of course, it is important to remember that biologists are different from those of us who design algorithms; they do not really want the optimal solution to the graph theoretic problem if this does not look like the actual chromosome arrangement. Thus, solutions to the problems described above should be viewed as candidate arrangements rather than final results, and this can affect our choice of algorithms. There are many attempts to find practical solutions to the special cases of NP-complete problems as well, both through imposing further constraints coming from by biology and by designing good approximation algorithms.

16.1.2 Phylogeny. Another link between the field of graph representations and computational biology comes from the phylogeny problem. Informally, the phylogeny problem is an attempt to find the most probable evolutionary development of a set of species. For example, it is assumed that chimpanzees, gorillas, orangutans, baboons, and humans all had a common ancestor at some point in the past. At some point, some of the animals evolved in one direction while others evolved in another. How do we determine whether there was a split into, for example, a common ancestor of chimpanzees, gorillas, and humans and a common ancestor of baboons and orangutans, as opposed to a split in which humans evolved on one path and the other apes on another?

The evolutionary sequence can be described by some tree. What would make one evolutionary tree a better candidate for the actual sequence than another? In general, this is not a trivial question to answer, but some rules seem reasonable. Suppose that we have a set of characteristics for each species; each characteristic can have some number of different values. The perfect phylogeny question asks whether there is a single tree with the species as leaves, all internal nodes assigned some value for each characteristic, such that nodes with the same value on a characteristic form a connected subtree. Note that internal nodes correspond to ancestor species.

For example, suppose that we have four species evaluated by size and behavior. Species A is big and mean, B is big and timid, C is small and timid, D is small and friendly. A possible perfect phylogeny is shown in Figure 16.1.

Since this is a model involving subtrees of a tree, it is natural to look for a relationship between perfect phylogeny and chordal graphs. The connection was discovered by Buneman [84] in 1974, and will be described below; he turns the perfect phylogeny problem into a colored chordal graph completion problem, in which one is allowed to add edges between vertices of different colors in order to produce a chordal graph.

Construct a graph G with a vertex for every value of every characteristic. Add edges between vertices if there is a species which has both of the corresponding characteristic values. Assign a color to each characteristic; thus, vertices with different values for this characteristic will get the same color, which will correspond to not allowing edges between these vertices.

In the example above, there are 5 vertices; big and small get color 1 while mean, timid, and friendly get color 2. There would be edges between big and mean, big and timid, small and timid, small and friendly. Since this graph has a colored chordal completion (actually, it is already chordal), there is a perfect phylogeny.

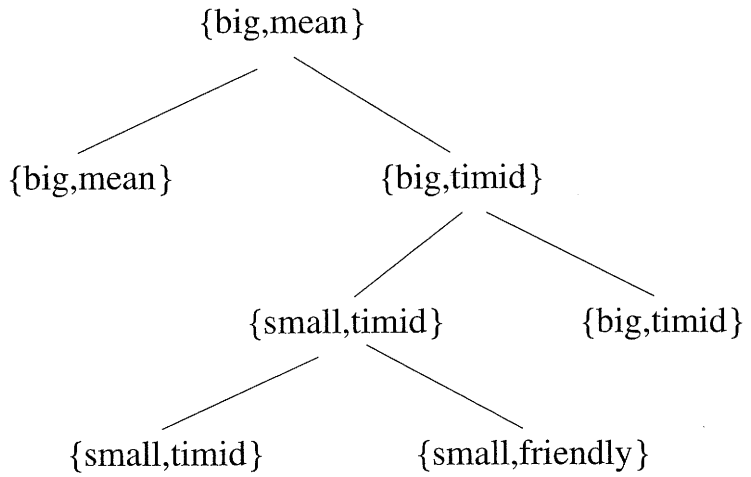


Figure 16.1 A Possible Perfect Phylogeny Tree

Suppose that there is a valid solution to the perfect phylogeny problem. It is easy to construct the corresponding chordal graph G', in which two vertices are adjacent iff the corresponding characteristic value trees have a nonempty intersection; it should be clear that G' contains G and includes no edges between vertices of the same color.

Suppose that we can create a graph G' as the solution of the colored chordal graph completion problem. We can easily modify the clique tree for G' to be a solution to the perfect phylogeny problem. The set of characteristics associated with a species correspond to a maximal clique in G'; if this clique is an internal node i rather than a leaf of the clique tree, we can add a child of i with the same characteristic values as i. Thus, chordal colored graph completion is equivalent to the perfect phylogeny problem.

The perfect phylogeny/colored chordal graph completion was open for two decades before being shown to be NP-complete in [46]; various restrictions can be solved in polynomial time. Other phylogeny problems are also studied, and chordal graph models may play a role in this area.

16.2 Networks

There are many connections between intersection graphs and networks, some of which will be discussed here. Since advances make a particular physical model outdated very quickly, I will try to describe applications which are not closely tied to a specific technology.

Fundamentally, networks are arranged in a particular topology, which corresponds naturally to a graph. Messages of various types are passed between nodes in the network. Each message corresponds to a path in the graph. In some models, there is a choice of which path should correspond to a message, while in other models the path is completely specified. Various notions of conflict between messages, such as requiring that messages are node disjoint or share no common links, correspond directly to questions about an underlying intersection graph.

Thus, if your network is a simple ring, messages will correspond to arcs on the circle. If we assume that the routing is specified for each message, we may want to solve the coloring problem on a circular-arc graph in order to minimize the number of phases, allowing an independent set of messages to be transmitted in a single phase. Although this problem is NP-complete on circular-arc graphs, good approximation algorithms exist. Other common simple networks correspond to interval graphs, and either path graphs or EPT graphs. We focus on coloring circular-arc graphs, which has applications in register allocation as well. We note that permutation graphs, and their generalizations such as circle graphs and circular permutation graph graphs, are often mentioned in the context of network routing problems; see, for example, [351].

16.2.1 Coloring Circular-Arc Graphs. It is interesting to note that coloring is NP-complete on circular-arc graphs, but the clique problem can be solved in polynomial time.

For a long time, no algorithm was known in the western world for finding a coloring of circular-arc graphs which guaranteed a coloring of cOPT, where $c < 2$. This was improved to 5/3 OPT by Hsu and Shih [269]. However, a simple algorithm using 3/2 OPT colors was discovered by Karapetjan in 1980; this algorithm is still largely unknown in the west, since it appeared only in Russian. I am indebted to David Gamarnik and Nathan Edwards for making a translation available.

This problem has been posed to me more than any other problem by researchers who claimed applications, so I will take the opportunity to describe both algorithms. This may seem to be an odd choice, since the Karapetjan algorithm uses fewer colors, appeared earlier, and is simpler. I choose to present both algorithms because although I have checked the correctness of Karapetjan's algorithm by going through his proof on a line by line basis, I cannot present the proof at a level which I feel is appropriate for this text, while Shih and Hsu's proof follows more directly from the algorithm itself.

We first present Karapetjan's remarkably simple algorithm, completely without proof. Let p be a point of the circle with maximum 'load', i.e. there is no point on the circle which has more arcs passing through the point than p. Let L be the load at p. Order the arcs which pass through p as a_1,a_2,...,a_L, with $i < j$ meaning arc j extends further in the clockwise direction from p than arc i.

In phase 1, the algorithm finds $L/2$ independent sets in G - a_1,...a_L as follows. For $i = 1$ to $L/2$, start with the first arc clockwise from a_i, and continue around the circle, always choosing the next clockwise arc from the current vertex, until no other arc can be added.

In phase 2, the algorithm finds L independent sets in G - vertices chosen in phase 1 as follows. For $i = 1$ to L, start with arc a_i, and continue around the circle counterclockwise, always choosing the next arc counterclockwise which is independent of all previous arcs in the family.

This simple algorithm always uses at most 3/2 L colors; the proof can be found in [292]. Since L is clearly no more than the chromatic number of G, this algorithm uses at most 3/2 OPT colors.

We now present the algorithm of Hsu and Shih, which finds a 5/3 OPT coloring of a circular-arc graph. One of the problems in working with circular-arc graphs is that it is not always true that cliques meet at a point. Hsu and Shih show that it is easy to modify an arbitrary circular-arc graph in polynomial time so that the

size of the chromatic number and maximum clique size do not change, and there is a maximum clique C such that all arcs of C go through a common point in the circle. To do this, we find the maximum clique C' in G and the maximum load on the circle, and add a set of tiny arcs on the interval with maximum load to increase the clique size to the size of C'. Exercise 16.9 asks the reader to show that this does not alter the size of the maximum clique or the chromatic number. For the rest of the discussion, we will assume that C is a maximum clique such that all arcs of C pass through a common point on the circle.

To simplify the description, it is useful to describe a natural greedy notion for coloring a circular-arc graph. If we start at any arc u, we traverse the circle in the clockwise direction, adding the next arc which is independent of u, until u is encountered again, giving all these arcs a single color. We can extend this single independent set to a longer sequence by starting with the first arc v in our sequence which overlaps with u, and finding our next independent set starting from v. This method of coloring will be called greedy coloring; it is uniquely defined if we are given a starting arc, a finishing arc, and how many times we wish to go around the circle.

A singular pair (singular trio) is a pair (trio) of arcs which cover the circle. Note that any singular pair on a graph with at least three vertices can be extended to be a singular trio. Tucker [469] showed that a simple algorithm will use $3/2\ |C|$ colors on a circular-arc graph if there are no singular trios. Hsu and Shih extended this algorithm to find a $3/2\ |C|$ coloring if there is a clique C which contains exactly one vertex from each singular trio. Thus, the initial goal is to eliminate singular trios which do not involve exactly one vertex of C without using too many colors.

We first describe the steps used for eliminating bad singular trios. At each removal step, the algorithm removes at least two vertices of C, reduces the load on the circle by 1, and uses at most 2 colors. If we have made $|C|/3$ removal steps, we stop. In this case we have used at most $2|C|/3$ colors, the load on the circle is at most $2|C|/3$, and the number of arcs passing through the common point of C is at most $C/3$; this gives us a $5/3|C|$ coloring algorithm by cutting the circle at the common point of C. If we stop after some number t of iterations where $t < |C|/3$, the total number of colors used is at most $2t + 3/2\ (|C| - t) = 3|C|/2 + t/2 \leq 5|C|/3$. After describing the removal steps, we will describe the coloring algorithm for the case in which no bad singular trios remain.

We begin by successively listing and removing singular pairs from C until none remain. Let the sequence of pairs at this point be $(u_1,v_1),...,(u_k,v_k)$. If this list includes all vertices of C, it is easy to see that $V - C$ is an interval graph which can be colored with at most $|C|/2$ colors, giving a 3/2 OPT algorithm. Let us assume there are some remaining vertices C' of C after this step. There must be a point d on the circle which is not covered by any arc in C'; let D be the arcs of $V - C$ which cover d.

Construct a bipartite graph with vertices of D on one side, vertices of C on the other, with c adjacent to d if and only if the vertices are nonadjacent in G. Create a maximum matching M on this subgraph; we note that this can be done in $O(n^2)$ time using the techniques presented in chapter 9. Every vertex of D will be matched in M, since C is a maximum clique (exercise 16.4).

We now eliminate the singular pairs $(u_1,v_1)...(u_k,v_k)$. For each (u_i,v_i), we give one color to u_i and $M(u_i)$ (if it exists), and another color to v_i and $M(v_i)$; $M(u_i)$ and $M(v_i)$ are also removed in this step.

We next eliminate singular pairs which have one vertex in C'; the other vertex of the pair must be in D, and cannot be matched in M with a vertex which has already been colored. Let $(c_1,d_1)...(c_j,d_j)$ be the sequence of singular pairs found at this stage. Once again, we use one color for c_i and M(c_i) and another color for d_i and M(d_i), using two colors and eliminating two vertices of C (this condition is where the matching is useful) and reducing the load by at least 1. At the end of this stage, we may assume that the model has no singular pairs, and that any singular trios contain a vertex which is not in C.

We next eliminate singular trios which contain exactly two vertices of C (and thus exactly one vertex of D). All arcs of C meet at a point p; let c_1 be the arc of C which extends furthest counterclockwise from p. If there is a singular trio containing two members of C, then there is a singular trio which contains c_1, some $d' \in D$, and some other $c' \in C$.

Traverse the circuit in clockwise fashion starting with the clockwise endpoint of c_1, maintaining a sequence c_1 through c_i of elements. Add the next independent arc to the set, until some arc c_k passes the counterclockwise endpoint of d'. Give 1 color to c_1 ... c_{k-1}. If c_k is nonadjacent to c_1, give c_k this color as well, and give a second color to d' and M(d'). If c_k is adjacent to c_1, then c_k is in D by exercise 16.5; assign a second color to c_k and M(c_k).

Note that the reason that the above procedure reduces the load is that for each clockwise endpoint, at least one arc covering this endpoint is colored by the procedure. This guarantees reduction of the load; see exercise 16.10.

We now describe an algorithm for finding a $3s/2$ coloring of a circular-arc graph with load s in which every singular trio contains exactly one vertex from maximal clique Q, where all arcs of Q pass through a common point p. The algorithm works by successively using three colors, and reducing the load by 2.

Start with the arc u_1 of Q which extends the shortest distance counterclockwise from p. Find a greedy coloring from u_1 until p is reached a second time; let u_k be the first arc be the first arc which reaches p or starts after p. Note that since u_1 extends least in the counterclockwise direction from p, u_{k-1} cannot be adjacent to u_1, or u_1 would be adjacent to all of Q, contradicting the assumption that Q is a maximal clique. Therefore, we can give $u_1..u_{k-1}$ a single color. If u_k does not cover p, we are in a good situation; the arcs $u_1..u_{k-1}$ are given a single color, and removal of these vertices will reduce the load by 1.

Otherwise, we find a greedy coloring from u_k, continuing until some arc u_l passes the counterclockwise endpoint of u_k, giving the arcs $u_k..u_{l-1}$ a single color. We then find a greedy coloring from u_l, stopping when p is reached for a third time by arc u_m; if u_m does not cover p it is not included in the set of arcs.

We have steeped greedily around the circle twice, so removing the arcs encountered will reduce the load by 2. Suppose that the arcs chosen from $u_l..u_m$ cannot be given a single color; this can be a problem only if u_l and u_m are adjacent, and u_m covers p. However, this would mean that u_k, u_l, u_m form a singular trio, with both u_k and u_m covering p and therefore being members of Q. Since we are working with a graph in which no singular trio contains two arcs from Q, we can color the arcs from u_l through u_m with a single color, and removing the arcs colored by the 3 colors reduces the load by at least 2.

16.3 Programming Languages

There has been a good deal of research dealing with special classes of graphs and the flow graphs of various programming languages. To see a simplified form of this connection, think of solving register allocation problem on a program which has no branching statements whatsoever. A variable remains active over a fixed interval of the program. Two variables must be assigned to different registers if their intervals overlap; thus the register allocation problem on such a program can be viewed as coloring an interval graph.

Some generalizations of the example above are given in Roberts [411]. Much of the work on series-parallel graphs, such as [475], was motivated by their usefulness in modeling control-flow graphs of programs. A control-flow graph has computer instructions as vertices, and an edge from a to b if it is possible for the program to jump from instruction a to instruction b. Variables may not be active merely over a connected interval, as in the previous paragraph, but instead over a connected region of the flow control graph. Thus, register allocation corresponds to coloring the intersection graph of connected subgraphs of the flow control graph.

The paper [291] shows that if the control-flow graph is series-parallel, the register allocation problem corresponds to a coloring problem which can be approximated within a factor of 2. Note that since a cycle is a series-parallel graph, this result generalizes the fact that circular-arc graph coloring to within a constant factor to a significantly larger class of graphs.

A more recent and surprising result due to Thorup relates real programming language flow graphs to the parameter treewidth. Treewidth has been defined in a number of ways; we give the following two characterizations as most in keeping with the ideas in this book.

A triangulation of a graph G is obtained by adding edges to obtain a graph G' which is chordal. The treewidth corresponds to the size of the smallest maximum clique size in any possible triangulation of G - 1. A k-tree is formed by starting with a clique of size k, and repeatedly adding a new vertex adjacent to a k-clique in the existing graph. A partial k-tree is a subgraph of a k-tree. The treewidth of G is also equal to the smallest value k such that G is a partial k-tree.

Specifically, Thorup [459] has shown the following results relating treewidth to control-flow graphs. These results imply that one can find an optimal solution to the register allocation problem in all of these types of programs.

Goto-free Algol and Pascal programs have control-flow graphs with treewidth at most 3.
All Modula-2 programs have control-flow graphs with treewidth at most 5.
Goto-free C programs have control-flow graphs with treewidth at most 6.

I do not know of any work dealing with intersection graphs of restricted subgraphs (such as paths) of a 2-tree or other fixed k-tree, rather than subgraphs of a tree; this might lead to new classes of potential interest.

16.4 A Cautionary Tale

There is an application of interval graphs to the seriation problem in archeology which is well known to theorists in the field; in fact, I have seen this listed as an application of other classes of graphs as well, even though the problem really deals

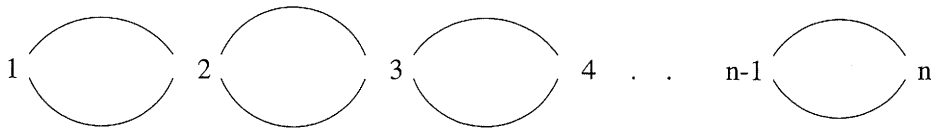


Figure 16.2 Chain of Edge Pairs for Problem 16.1

with intervals. In this problem, we are given information about which types of pottery are found in common graves. We assume that a type of pottery is made over a time interval, and we want to find relative dating as to when a pottery type was made.

This maps nicely to an interval graph problem, if we place an edge between vertices which were found in a common grave. An interval model for the corresponding graph maps to a possible ordering in time of the pottery types.

The majority of this book was written while I lived in the unusually pleasant confines of the American School of Classical Studies in Athens, due to my wife's work in the field of archeology. Thus, I was living among one of the densest concentrations of archeologists on the planet. At some point, I was asked what this book was about, and decided to include a description of the archeological application to help introduce graph theory.

Introducing the application proved to be a serious mistake, since the application seems to bear no relationship to ways in which pottery is actually dated. At this point, I looked up the original paper on the subject. The author notes the relationship between interval graphs and dating pottery, but in later papers by the author this is viewed as an unrealistic case, in which all information about time overlap is known. Later papers deal with a much more difficult problem, in which each pair of pottery types is assigned a probability, and one attempts to come up with a model which is a best fit in a probabilistic sense. In fact, all these approaches are theoretical; archeologists do not normally date pottery using any mathematical tools at all.

The moral of this story is to beware of applications passed on in theoretical texts like this one, until you can validate them with people who really work in the field!

16.5 Exercises

1) Consider the intersection graphs of simple paths (no repeated vertices allowed) on a chain of edge pairs as shown in Figure 16.2, where intersection is defined to be sharing a common edge.

1a) Show that there are $2^{\Theta(n^2)}$ graphs in the class.

1b) Show that the class contains the circular-arc graphs, so coloring is NP-complete. Note: you cannot quite interpret the model as a circle by separating the bottom and top of each edge pair, since paths must be simple.

1c) Find a 2 times optimal approximate coloring algorithm for this class.

2) Suppose that you are given a circularly ordered set of nodes, and instead of being given paths between vertices (corresponding to circular arcs), you are given endpoints of a proposed message; this allows you to choose an arc which goes in either direction around the circle. Many problems may be proposed in such a scheme; suppose that we want to minimize the maximum 'load' of any node; the load of a node is the number of arcs which pass through the node. Design a 2 * optimal algorithm for this problem. The problem can be solved optimally in polynomial time; see [202] for a description of an optimal algorithm.

3) Design a polynomial time algorithm to determine whether a circular-arc graph is 3-colorable. Note: this can be generalized; k-coloring of circular-arc graphs can be solved in $O(n+m)$ time for any fixed k.

4) This problem concerns one step of Hsu and Shih's circular-arc coloring algorithm. Let C be a maximum clique in G such that all vertices of C meet at a point. Once singular pairs have been removed from C to get a subset C', there is a point d on the circle such that no arc of C' covers d; let D be the arcs of V-C which cover d. The algorithm creates a bipartite matching problem with C as one color class, D as the other, and vertices from different classes adjacent in the matching problem if and only if they are nonadjacent in G. Show that every vertex of D is matched in a maximum matching on this graph.

5) We use the same sets C and D as in the previous problem. A singular trio is a set of three arcs which cover the circle; we now discuss the algorithm after singular pairs and singular trios containing three members of C have been removed. Let c_1 be the arc of C which extends furthest counterclockwise from the common intersection point for C, and is part of a circular trio c_1, d', c', with $d' \in D, c' \in C$. Create a greedy independent set by taking the next clockwise arc from c_1, until we find an arc x which passes the counterclockwise endpoint of d'. Show that if x is adjacent to c_1, x is in D.

6) Design a simple algorithm (and proof) which colors circular-arc graphs with clique cover number 3 using at most 5/3 OPT colors in polynomial time.

7)(This problem solution due to Vijay Raghavan) Show that there are circular-arc graphs with clique size k and chromatic number $3k/2$ for arbitrarily large values of k. Therefore, there is no constant c such that the chromatic number of a circular-arc graph is no larger than c plus the size of the maximum clique. Note: making copies of vertices on a 5-cycle will not work here; cloning each vertex of a 5-cycle gives a graph with maximum clique size 4 and chromatic number 5.

8) Show that the maximum clique size in a circular-arc graph can be arbitrarily close to twice the load on the circle.

9) Show that it is possible to add arcs to any circular-arc graph G so as to get a circular-arc graph with the same chromatic number and clique size, such that the load becomes the same as the clique size in the new graph.

10) Show that the load of a circular-arc graph is equal to the maximum number of arcs which overlap some clockwise endpoint. This fact is used in the proof of the bound on Hsu and Shih's circular-arc coloring algorithm implicitly, in arguing that removing certain sets of arcs decreases the load.

11) Suppose that we are given a set of colors, and we want to color the maximum possible number of vertices. Note that the number of colors may be larger than a constant.

11a) Show that greedy coloring on a set of intervals (i.e. color the earliest ending vertex with a legal color, if possible) does not necessarily color the maximum number of vertices.

11b) Design an O(nlogn) algorithm to solve this problem. It is possible to solve this problem in $O(n)$ time, but one must be very clever in the implementation of the algorithm [90].

11c) Design an $O(n+m)$ algorithm to solve this problem.

12) Show that every k-tree is chordal.

Glossary

I should note that these definitions sometimes may be in conflict with general use of a term. If a term is used in a specific manner in this book, and this specific usage is simple to explain, I choose to include the definition which applies to this book rather than a more general definition which might be harder to understand. For example, this book does not discuss multigraphs, and in some cases the definition I choose to include would not correspond to standard definitions on multigraphs. I also do not worry about the fact that my definition of order notation symbols is nonstandard if negative functions are allowed, since these are used only for time analyses in this book.

Adjacent: Vertex x is adjacent to vertex y if there is an edge from x to y.

Adjacency lists: A form of graph representation in which neighbors of vertex i are stored in a linked list. In general, this list is assumed to be unordered, in the sense that vertex j may precede vertex k on an adjacency list even though $j < k$. However, it is possible to sort all adjacency lists in $O(n+m)$ time, if this is desired. Technically, an adjacency list is a matrix of linked lists, thus allowing the list of neighbors of vertex i to be accessed in constant time.

Adjacency matrix: A form of graph representation in which a graph is stored as an $n \times n$ matrix. The entry (i, j) of the adjacency matrix is 1 if i has an edge to j, and 0 otherwise.

Anti-hole: A hole (chordless induced cycle of length at least 4) in the complement graph.

Arboricity: The arboricity of G is the minimum number of forests such that every edge of G is contained in one of these forests.

Arc: A connected subset of a circle. Also used as an alternate name for edge, especially in a tree.

Asteroidal triple: A trio of mutually nonadjacent vertices x, y, z with the property that for each pair of vertices in the trio, there is a path between these vertices which avoids all neighbors of the third vertex in the trio. Graphs without asteroidal triples are also called AT-free.

AT-free: A graph is AT-free if it contains no asteroidal triples.

Augmented adjacency matrix: Entries are the same as in the adjacency matrix, except with 1s along the main diagonal rather than 0s.

Autonomous set: Alternative name for module.

Balanced k-module: A subset S of vertices induces a balanced k-module if there is a partition S_1, S_2 of S such that $|S|/3 \leq |S_1| \leq 2|S|/3$, and S_1 can be partitioned into at most k subsets $S_{1,i}$ in such a way that every vertex of S_2 is either adjacent to every vertex of $S_{1,i}$, or to no vertex in $S_{1,i}$.

Bandwidth: The bandwidth of an ordering of the vertices of G is the maximum difference between endpoint positions of any edge. The NP-complete bandwidth problem asks whether there is any ordering of the vertices which has bandwidth at most k.

Berge graph: A graph without any odd holes or odd anti-holes of length greater than three; these have recently been shown to be equivalent to perfect graphs, resolving a famous open question.

Biconvex graph: Bipartite graph with the property that there are orderings of each color class satisfying for all x, the neighbors of x occur consecutively in the ordering.

Bipartite adjacency matrix: Matrix representation of a bipartite graph. Rows correspond to vertices from one color class, columns to vertices from the other color class, with a 1 in row i column j if vertex i is adjacent to vertex j, and 0 otherwise.

Bipartite graph: A graph with the property that vertices can be partitioned into sets X, Y such that every edge goes between a vertex in X and a vertex in Y. The sets X and Y are called color classes of G.

Bisimplicial edge: An edge (x, y) of a bipartite graph such that $\mathrm{N}(x) \cup \mathrm{N}(y)$ induces a complete bipartite graph.

Bit: A single digit in a binary (0/1) representation of an object.

Block: In a partition of a set, each subset is called a block.

Bounded substitution diameter decomposition: A graph has substitution decomposition diameter k if every prime node in the decomposition tree has at most k children. If a class of graphs has substitution decomposition diameter at most k for k fixed, then the class has bounded substitution decomposition diameter.

Bounded tolerance graph: Tolerance graph such that no tolerance value is larger than the length of the interval. This implies that if an interval corresponding to vertex x contains the interval corresponding to y, the vertices must be adjacent.

Boxicity: The minimum dimension d such that a graph can be represented as an intersection graph of d-dimensional rectangles, with all rectangles oriented along the axes. Boxicity 1 graphs correspond to interval graphs.

Brittle graph: A graph G is brittle if there is an elimination scheme for G which successively removes either a vertex which is not a midpoint of any P_4, or not an endpoint of any P_4.

C_i: A chordless cycle on i vertices.

Certificate: A certificate of a property is a proof that the property holds. Certificates are important in the theory of NP-completeness; NP can be defined in terms of existence of polynomial size certificates. Certificates are used in a number of ways in this book; in algorithms, the existence of a certificate of size $f(n)$) can be a separate problem from finding an $O(f(n)$ algorithm to solve the problem, and our discussion of robust algorithms deals with issues which arise if we are given a certificate that a graph is in a class as opposed to when such a certificate is not given.

Chain graph: A bipartite graph with the property that for each pair of vertices u, v from the same color class, either $N(u) \subseteq N(v)$ or $N(v) \subseteq N(u)$.

Child: In a rooted tree, children of v are neighbors of v other than the parent of v.

Chord: In a cycle, an edge which goes between vertices which are not consecutive on the cycle. On a circle, a straight line connecting two points on the circle.

Chordal graph: A graph with no induced (i.e. chordless) cycle of length > 3.

Chordal bipartite graph: A bipartite graph with no chordless cycles of length greater than 4. An important characterization is that G is chordal bipartite if and only if the bipartite adjacency matrix has a Γ-free ordering.

Chromatic number: The minimum number of colors necessary to color a graph.

Circle graph: Intersection graph of chords of a circle.

Circle order: Containment graph of disks in the plane.

Circuit: Alternative name for cycle.

Circular-arc graph: Intersection graph of arcs of a circle.

Circular permutation graph: Intersection graphs of curves connecting points on two concentric circles of different diameter, such that no pair of curves intersects at more than one point.

Circular 1s property: A circular ordering of a set, which obeys a set of restrictions requiring particular subsets to appear consecutively in the circular order.

Classically NP-complete: A problem P is classically NP-complete on a restricted domain D of inputs (such as a class of graphs) if every problem in NP can be mapped in polynomial time and space to an instance of P from D, such that the answer to the original problem is the same as the answer to the instance mapped to.

Claw-free graph: A graph with no induced subgraph equal to $K_{1,3}$.

Clique: A set of mutually adjacent vertices.

Clique cutset: A cutset which is also a clique.

Clique separator: Alternate name for clique cutset.

Clique separator decomposition: A form of graph decomposition which takes an arbitrary clique cutset C, decomposes the graph into $S \cup C$ for each connected component S of $G - C$, and decomposes each subgraph recursively.

Clique problem: In decision form, the problem of whether an input graph G has a clique of size equal to input number k. Also used in optimization form to denote the problem of determining the cardinality of the maximum clique in a graph. The decision problem is NP-complete.

Clique tree representation: An intersection model for a chordal graph G. There is a tree T, such each node of T corresponds to a maximal clique of G, and cliques containing each individual vertex v correspond to a connected subtree of T. G has a clique tree representation if and only if G is chordal, and this representation can be found in linear time.

Clique-width: The minimum number of labels needed to construct a graph from one vertex graphs using the operations union, addition of all edges between vertices with label i and vertices with label j, and the relabeling operation which gives label j to all vertices with current label i.

Clone: A clone c of vertex v with respect to a set of vertices S is a vertex such that $N(c) \cap S - v = N(v) \cap S - v$.

Closure: The k-closure of a graph is formed by repeatedly adding edges between nonadjacent pairs x, y of vertices such that degree(x) + degree(y) is at least k, until no further edges can be added. In a variant called k'-closure, edges are added between nonadjacent x, y such that $|N(x) \cup N(y)|$ is at least k.

Co-: For a graph class C, a graph is a co-C graph if the complement is in C.

Co-NP: A decision problem P is in co-NP if the problem of deciding whether the answer to P is no is in NP; i.e., no answers have a polynomial size certificate.

Cograph: In the original definition, a graph which can be built from single vertex graphs using the operations of complement and union. An important characterization is that these are exactly the graphs without any P_4.

Coloring: An assignment of numbers (called colors) to vertices of a graph, such that no edge connects vertices with the same color. The question of whether it is possible to color a graph with k colors is NP-complete even for $k = 3$. In the optimization version of the coloring problem, the goal is to color a graph with the minimum number of different colors.

Color class: In this book, this refers to one of the subsets of vertices used to partition a bipartite graph. More generally, a color class is a set of vertices given the same color by a coloring assignment.

Comparability graph: An undirected graph such that all edges can be assigned directions, such that in the directed graph whenever $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

Complement: The complement of G, written as $\overline{G}$, has the same vertex set as G, and an edge from x to y in $\overline{G}$ if and only if there is no edge from x to y in G.

Complete graph: Alternative name for clique.

Complete bipartite graph. A bipartite graph which has an edge between each pair of vertices from different color classes.

Completely k-decomposable: A graph is completely k-decomposable with respect to a decomposition if every induced subgraph containing more than k vertices k is decomposable.

Completion problem: The completion problem for a class of graphs C asks for the minimum number of edges necessary to add to an input graph G, so that the augmented graph is in C.

Composition sequence: A class of graphs has a composition sequence if there is a countable sequence $G_1, .., G_i, ...$ of graphs in the class such that each G_i is an induced subgraph of G_{i+1}, and every graph in the class is a subgraph of some G_i in the sequence.

Connected component: Maximal connected subgraph.

Connected graph: A graph such that there is a path between each pair of vertices.

Connected matrix: A matrix such that no pair of rows and columns induces I_2.

Consecutive 1s property: An ordering of a set which obeys restrictions requiring elements in various subsets to occur consecutively.

Construction problem: The problem of constructing a form of graph representation.

Containment graph: The containment graph of a set of objects has a vertex corresponding to each objects, and a directed edge from x to y if object x contains object y.

Containment representation: A representation of a graph as a containment graph.

Contraction: The graph formed by contraction of an edge (x, y) replaces the pair of vertices x, y by a new vertex z, with $\mathrm{N}(z) = N(x) \cup N(y)$.

Convex graph: Bipartite graph such that vertices of one color class can be ordered so that for every vertex v from the other color class, $\mathrm{N}(v)$ occurs consecutively in the ordering.

Convex fan: A polygon P in which there is a single endpoint v with the property that every other endpoint w of P is either a neighbor of v on P, or the line segment connecting v and w is entirely inside of P.

Convex vertex: Given a model for a visibility graph, a vertex is convex if it maps to an endpoint corresponding to a polygon angle of less than 180 degrees.

Cook's theorem: The theorem proving the NP-completeness of CNF-satisfiability. Almost all NP-completeness proofs eventually derive from Cook's theorem.

Cotree: A representation of a cograph. Vertices correspond to leaves of a tree, and internal nodes of the tree are labeled with 0 and 1. Two vertices u, v of G are adjacent if and only if the least common ancestor of u and v in the cotree has label 1. G can be represented by a cotree if and only if G is a cograph.

Counting problem: The problem of determining the number of graphs in a class.

Covering graph: The undirected graph formed by removing directions from the transitive reduction of a partial order.

Cowly perfect: Literal translation of one of my favorite names for a graph class, vachement parfait, which unfortunately did not find a place in this book. Those interested in correct names for graph classes could debate whether or not such a class should be bull-free.

Cutset: A set S of vertices such that G - S is disconnected.

Cycle: A sequence of vertices $v_1, v_2, ..., v_k$ such that for i in the range 1..k-1, v_i is adjacent to v_{i+1}, and v_k is adjacent to v_1.

Cycle-free partial order: A partial order such that the underlying graph is chordal.

Decision problem: A problem such that the answer is either yes or no. The term NP-complete applies by definition to decision problems.

Degree: For the purposes of this book (in which self-loops are never used), the number of edges out of a vertex.

Deletion problem: The deletion problem for a class C of graphs asks for the minimum number of edges necessary to delete from an input graph G so that the resulting graph is in C.

Dense: Having many edges. Often used in an informal sense, but sometimes used to mean number of edges is $\Theta(n^2)$.

Descendant: A vertex d in a rooted tree is a descendant of v if the path from the root to d goes through v.

Deterministically k-decomposable: A graph is deterministically decomposable with respect to a decomposition if for every possible choice of recursive decomposition steps, every prime component has size at most k.

Diameter: The diameter of a graph is the maximum distance between any pair of vertices.

Digraph: Directed graph.

Dilworth Number: The maximum cardinality of a set S of vertices in a graph such that for all pairs of vertices x, y in S, some neighbor of x is not in $\mathrm{N}[y]$ and some neighbor of y is not in $\mathrm{N}[x]$.

Dimension: The minimum number of linear extensions of a partial order which give the partial order as their intersection. In other words, the minimum number of lists of vertices of a partial order P such that $x < y$ if and only if x precedes y in all lists.

Disk intersection graph: Intersection graph of disks in the plane, where a disk denotes a circle plus the interior of a circle; thus, if one disk contains the other, the corresponding vertices are adjacent.

Distance: The distance between x and y is the length of the shortest bath between x and y.

Distance hereditary graph: A graph such that for every pair of vertices x, y, all chordless paths from x to y have the same length. Relevance to this book comes in part from an elimination scheme characterization; a connected graph G is distance hereditary if and only if it can be reduced to a single vertex by repeated elimination of twins or pendant vertices.

Distance labeling: Generalization of implicit representation to the distance function. Each vertex is assigned a label, and the distance between x and y can be computed using only the label information stored at x and y.

Dominating pair: A pair of vertices x, y is a dominating pair if for all vertices z, every path from x to y contains at least one neighbor of z.

Dominating set: A set S of vertices such that every vertex of V - S is adjacent to at least one vertex in S. Deciding whether a graph has a dominating set of size input number k is NP-complete. In the optimization version, it is necessary to find the smallest cardinality dominating set.

Domination graph: A graph G with the property that for every induced subgraph H of G, there is a pair of vertices x, y such that $\mathrm{N}(x) \subseteq \mathrm{N}[y]$ in H.

Dot-product representation: Representation of a graph in which each vertex is associated with a vector, and vertices are adjacent if and only if the dot-product of the corresponding vectors is at least 1. The dot-product dimension is the minimum number of elements in vectors which can be used to represent G. Representing graphs with constant dot-product dimension implicitly is open.

Doubly chordal graph: A graph is doubly chordal if there is an ordering of the vertices $v_1 v_2 .. v_n$ such that each v_i is simplicial in the graph induced by $v_i .. v_n$, and v_i has a maximal neighbor in the graph induced by $v_{i+1} .. v_n$.

Doubly convex graph: Alternate name for biconvex graph.

Dually chordal graph: Graph which can be ordered $v_1 .. v_n$ such that each v_i has a maximal neighbor in the graph induced by $v_i .. v_n$.

Edge: A connection between two vertices. This can be written as an unordered pair or as u - v for an undirected graph, and as an ordered pair or $u \rightarrow v$ for a directed graph.

Efficient representation: This is an informal term used throughout the book. In general, it means that this form of representation has important advantages over standard methods for representing a class of graphs, even though the representation might not be space optimal, and adjacency testing might not take constant time or be distributed to bits stored at vertices.

Elimination scheme: An ordering $v_1 v_2 .. v_n$ of vertices of a graph, such that each vertex v_i satisfies some particular property P in the graph induced by $v_i .. v_n$.

Envelope: The envelope of a set of line segments consists of the union of line subsegments on the infinite face of the union of the segments.

EPT graph: Intersection graph of paths in a tree, where we say two paths intersect if they share a common edge (i.e. two paths which intersect only at a vertex must correspond to nonadjacent vertices).

External visibility graph: A graph in which vertices correspond to endpoints of a polygon, and two vertices are adjacent if the line segment connecting them is entirely outside the polygon.

f-diagram: Structure used in a representation of co-comparability graphs. Vertices correspond to curves connecting points on two parallel lines, with curves intersecting (at least once) if and only if the corresponding vertices are adjacent. This representation is possible if and only if G is a co-comparability graph.

Feedback vertex set: A set S is a feedback vertex set if G - S is acyclic. The feedback vertex set problem asks for the minimum cardinality feedback vertex set; the decision version of feedback vertex set is NP-complete.

Ferrers digraph: A directed graph such that for all pairs x, y, either $\mathrm{N}[x]$ contains $\mathrm{N}(y)$ or $\mathrm{N}[y]$ contains $\mathrm{N}(x)$.

Ferrers dimension: The Ferrers dimension of a directed graph G is the minimum number of Ferrers digraphs which yield G as their intersection.

Fill-in scheme: A method for constructing a graph by repeatedly adding edges to the current graph using some set of rules.

Generalized implicit graph question: The open question as to whether every hereditary class of graphs with $2^{f(n)}$ members has a generalized implicit representation.

Generalized implicit representation: For a graph class with $2^{f(n)}$ graphs on n vertices, a representation which stores $\mathrm{O}(f(n)/n)$ bits at each vertex, such that adjacency between x and y can be tested using only the bits stored at x and y.

Global information: Information representing a graph is global if it can be accessed from more than one specific vertex of the graph.

Graph: An ordered pair (V, E), where V is a set of vertices and E is a set of edges. Edges are unordered pairs of vertices.

Graph class: A (usually infinite) set of graphs.

Graph isomorphism: The problem of determining whether two graphs are isomorphic. No polynomial algorithm for graph isomorphism is known, and graph isomorphism is not known to be NP-complete.

Greedy algorithm: An algorithm which adds to its current solution sequentially according to some criterion, and never backtracks by removing an element from its current solution.

Greedy coloring: The assignment of colors to vertices according to an order, which assigns the next vertex v the smallest color number which has not already been assigned to a neighbor of v. In the context of circular-arc graph coloring, the order

corresponds to next arc in the circular-arc model.

Grid: Grid is used in two difference senses in this book. If a geometrically defined class of graphs is defined, placing the model on a grid means assigning integer coordinates to all objects used in the definition. The grid is also a special graph in which endpoints have labels (b, c) for b, c in some ranges $1..i$,$1..j$ and two vertices (b, c) and (d, e) are adjacent if and only if $b = d$ and c differs from e by 1, or $c = e$ and b differs from d by 1. A graph is a grid graph if it is an induced subgraph of the grid.

Grid intersection graph: Intersection graph of a set of line segments in the plane, where each line segment is either horizontal or vertical.

Hamilton cycle: A Hamilton cycle in a graph is a cycle which goes through every vertex exactly once. Also called Hamilton circut. The problem of deciding whether a graph has a Hamilton cycle is NP-complete, as is deciding whether a graph has a similarly defined Hamilton path.

Height: In this book, the height of a poset is the length of a longest path (or size of a maximum chain) in the poset. Thus, we would say a bipartite poset has height 1, while this is called height 2 in some texts.

Helly property: A set has the Helly property if for all subsets S such that each pair of objects in S has a common intersection, then the entire set S has at least one common intersection point. For intersection representations of graphs, this means that all cliques share a common point in the representation.

Hereditary: A class of graphs is hereditary if for every G in the class and every vertex v of G, G - v is also in the class.

Hole: Chordless cycle of length at least 4. This book allows holes to be of even length, while some papers reserve the name for odd length chordless cycles.

Homogeneous Pair: Vertex sets A, B such that $|A| + |B| > 2$, A is a module of $G - B$, and B is a module of $G - A$. Also known as a 2-module.

Homogeneous set: Alternative name for module.

House: The specific graph on 5 vertices consisting of a cycle on 4 vertices, and a 5th vertex adjacent to 2 neighbors in the cycle.

Hypergraph: A hypergraph differs from a graph in that each edge (called a hyperedge) is a subset of any number of vertices, rather than being a subset of exactly 2 vertices.

Identity matrix: A square matrix with 1s on the main diagonal, and 0s everywhere else.

Immediate predecessor/successor: In a partial order, these are vertices which have edges to/from v in the transitive reduction.

Implicit graph conjecture: The conjecture that every hereditary class of graphs which has $2^{O(n \log n)}$ graphs on n vertices must have an implicit representation.

Implicit representation: Graph representation which stores O(log$n$) bits per vertex, such that adjacency between x and y can be tested using only the bits stored at x and y.

Indegree: The number of edges into a vertex of a directed graph.

Independent set: A set of mutually nonadjacent vertices. Determining whether a graph has an independent set of size input parameter k is NP-complete. In the optimization problem, you ask for the maximum cardinality independent set.

Induced subgraph: The subgraph induced by a set S of vertices in a graph G has S as its vertex set, with two vertices adjacent in the induced subgraph if and only if these vertices are adjacent in G.

Induced visibility graph: A graph which is an induced subgraph of some visibility graph.

Intersection class: A graph class which corresponds exactly to intersection graphs of a particular type of object.

Intersection graph: The intersection graph of a set of objects has a vertex corresponding to each object, with x and y adjacent if and only if the corresponding objects have a nonempty intersection.

Interval dimension: The interval dimension of a partial order P is the minimum number of interval orders which yield P as their intersection.

Interval filament graph: The intersection graph of a set of curves connecting endpoints on a fixed line L, where all curves are required to stay above L in the plane, and between the two endpoints.

Interval graph: Intersection graph of intervals on the line.

Interval number: The interval number of G is the smallest i such that G can be represented using i intervals for each vertex, with vertices x and y adjacent if and only if at least one of x's intervals has a nonempty intersection with at least one of y's intervals.

Interval order: A partial order is an interval order if elements can be associated with intervals on the line, such that $x < y$ if and only if the interval associated with x lies entirely to the left of the interval associated with y.

Inversion-free: An inversion-free ordering of a graph is an ordering $v_1 v_2 .. v_n$ of the vertices, such that for all $i < j < k < l$, if there is an edge from v_i to v_k and v_j to v_l, there must be an edge from v_i to v_l. A graph is inversion-free if it admits an inversion-free ordering.

Isomorphic: Two graphs G_1, G_2 are isomorphic if there is a one-to-one, onto mapping f from vertices of G_1 to vertices of G_2, such that x is adjacent to y in G_1 if and only if $f(x)$ is adjacent to $f(y)$ in G_2. Graph isomorphism is a famous open problem, in the sense that it is not known to be either polynomially solvable or NP-complete. A class of graphs is isomorphism-complete if solving the isomorphism problem on the class with a polynomial time algorithm would imply a polynomial time algorithm for general graph isomorphism.

Isooriented: A set of objects whose principal axes are either mutually parallel or perpendicular.

Join decomposition: A form of graph decomposition based on splits; a split is a partition of the vertex set into S_1, S_2 such that both S_1 and S_2 have at least 2 vertices, and edges which go from S_1 to S_2 form a complete bipartite graph. If G has a split S_1, S_2, G is decomposed into $S_1 \cup m$, $S_2 \cup m$, where m is a marker vertex which has edges to exactly those vertices which have neighbors in the other partition class. The two sets $S_1 \cup m$ and $S_2 \cup m$ are then decomposed recursively.

k-tree: A graph which can be constructed from a k-clique by repeatedly adding a vertex adjacent exactly to some k-clique in the current tree.

k-module: A set M of vertices which can be partitioned into at most k subsets $M_1 .. M_k$ such that each M_i is a module in $G - M \cup M_i$.

K_i: A clique on i vertices.

$K_{i,j}$: A complete bipartite graph with i vertices in one color class, and j in the other.

Labeled graph: A labeled graph has names attached to vertices. For example, if we have two graphs, one of which corresponds to the path 1,2,3 and the other to the path 1,3,2, these are different labeled graphs (they have different answers to the question of whether vertex 1 is adjacent to 2), although the two graphs are isomorphic.

Leaf: In an undirected tree, a vertex of degree 1. In a rooted tree, a vertex with no children.

Lexicographic breadth first search: An important restriction of breadth first search, designed originally for recognizing chordal graphs. Initially, all vertices are placed in a single set. The algorithm repeatedly removes a vertex v from the last current set, and divides all subsets into neighbors of v and nonneighbors of v, placing the subset of nonneighbors immediately before the subset of neighbors.

Line graph: The line graph of G has a vertex for each edge of G, and an edge between two vertices of the line graph if and only if the corresponding edges have a common endpoint. H is a line graph if and only if H is the line graph of some graph G.

Line-of-sight graph: Given a set of objects, a line-of-sight graph has a vertex for each object and an edge between vertices if some line segment between these objects avoids all other objects. In some cases, restrictions are placed on the segments, such as requiring the line segment to be horizontal.

Linear extension: For a partial order P, a total ordering T of elements of P such that whenever $x < y$ in P, $x < y$ in T.

Linear matrix: A 0/1 matrix which does not contain any induced 2×2 induced submatrix with all values equal to 1.

Linear time: Time proportional to the size of the input plus the size of the output.

Linked list: A fundamental data structure in computer science. If items are stored in a linked list, reaching the ith item in the list requires stepping through each of the i-1 preceding items in the list.

Literal: In a satisfiability problem, a literal is either x or $\overline{x}$ for some variable x.

Local complementation: Local complementation of G at v changes a graph by replacing the subgraph induced by neighbors of v with its complement.

Local information: Information associated with specific vertices of a graph.

Local Structure: Alternative name for implicit representation.

log: $\log x$ is defined (in the tradition of computer science) to be the base 2 logarithm of x.

Matching: A set of edges is a matching if each vertex is an endpoint of at most one edge in the set. Computing a maximum cardinality matching can be done in polynomial time, but finding an $O(n^2)$ algorithm for matching would be considered an extraordinary achievement, so that if you can show a problem is as hard as maximum matching you have given some evidence of the difficulty of finding a linear time algorithm.

Maximal: A set S is maximal with respect to a property if there is no set S' which properly contains S and has the property.

Maximal neighbor: A maximal neighbor of v is a neighbor x of v such that for all neighbors w of v, $N(w)$ is a subset of $N[x]$.

Maximum: A set S is maximum with respect to a property if there is no set with larger cardinality which has the property.

Maximum cut: An NP-complete graph problem, asking whether there is a partition V_1, V_2 of the vertices such that at least k edges have one endpoint in V_1 and the other endpoint in V_2. Also known as max cut and simple max cut.

Minimal: A set S is minimal with respect to a property if there is no proper subset of S which has the property.

Minimal fill problem: The problem of finding a set S of edges to add to G so that $G \cup S$ is chordal, and there is no proper subset S' of S such that $G \cup S'$ is chordal. A set S of edges is a minimal fill set if and only if every edge of S is the unique chord in some 4-cycle of $G \cup S$.

Minimal separator: Unfortunately, this term has several different meanings in the field. In this book, a set S is a minimal separator if there is some pair of vertices a, b such that a and b are in different components of $G - S$, and for every proper subset S' of S, a and b are in the same component of $G - S$. Note that this is not identical to a cutset S such that no proper subset of S is a cutset, which is an alternative usage of the term minimal separator.

Minimum: A set S is minimum with respect to a property if there is no set which has smaller cardinality with the property. Every minimum set is minimal for a property, but a set can be minimal and not minimum.

Minimum fill problem: The problem of minimizing the number of edges which can be added to G so that the graph is chordal. The decision version is NP-complete.

Model: Many of the graph classes discussed in this book are described in terms of relations (intersection, containment, visibility, and others) between a set of objects. In such cases, we say that we are given a model if we are given the objects, rather than being given the graph in a standard format such as adjacency lists or an adjacency matrix.

Modular decomposition: Alternate name for substitution decomposition.

Module: A set M of vertices such that for every vertex v of V - M, v is either adjacent to every vertex of M, or v is adjacent to no vertex of M.

N(v), N[v]: $\mathrm{N}(v)$ is the set of vertices adjacent to v, and $\mathrm{N}[v]$ includes both v and $\mathrm{N}(v)$.

N-free poset: A partial order such that the transitive reduction contains no induced N, where an N is a set of four vertices u, v, w, x with edges (w, u), (x, u), (x, v), forming a shape roughly equal to the letter N if u and v are placed on top with w and x on bottom.

Neighborhood: The set of vertices which can be reach from a vertex using a single edge.

NLC-width: The NLC-width of a graph G is the minimum number k of labels which can be used to construct G from single vertex labeled graphs using the union and relabel operations. The relabel operation assigns all vertices with current label i the label j. The union operation takes two disjoint graphs G_1,G_2 and a bipartite graph where each color class contains a vertex for each label, creating a single graph by adding an edge between vertex i_1 from G_1 and vertex j_2 from G_2 if there is an edge of the bipartite graph between the copy of the label of i_1 from color class 1 and the label of j_2 from color class 2.

Node: Alternative name for vertex, used especially in trees.

Nondetermistically k-decomposable: A graph is nondeterministically k-decomposable with respect to a decomposition if there exists a sequence of decomposition steps which decomposes the graph into prime components of size at most k.

NP: A decision problem is in NP if there is a polynomial size certificate for every instance of the problem for which the answer is yes.

NP-complete: A problem X is NP-complete if it is in NP, and every problem in NP can be polynomially transformed to X. This is a large topic which cannot be treated fully in this book; readers unfamiliar with the term are advised to consult [206] for more information. In general, NP-completeness of a problem is used as strong evidence of the intractability of the problem; at the least, a polynomial algorithm to solve an NP-complete problem would require an enormous breakthrough.

o: f(n) is o(g(n)) if for every constant c there exists a constant n_0 such that for all $n > n_0$, $f(n) < cg(n)$. Intuitively, this means that f(n) grows more slowly than g(n) even if constants are ignored. For the purposes of this book, the simpler definition that f(n) is o(g(n)) whenever f(n) is O(g(n)) but g(n) is not O(f(n)) will be equivalent for all functions studied here, though the definitions are not equivalent in general. A typical use is posing the question of finding an o(f(n)) algorithm to solve a problem, where the fastest known algorithm has running time Θ(f(n)).

O: f(n) is O(g(n)) if there exist constants c, n_0 such that for all $n \geq n_0$, $f(n) \leq cg(n)$. Intuitively, g(n) grows at least as quickly as f(n), if constant factors are ignored.

Optimization problem: A problem in which the goal is to find a maximum or minimum set. Many problems have a decision (yes/no) or optimization version of the problem.

Order notation: Mathematical notation designed to compare growth rates of functions, while ignoring constant factors. In this book, we will use the symbols o, O, Ω, and Θ, which correspond to $<, \leq, \geq, =$ if we ignore constant factors.

Ordered chordal graph: Given a graph G and an ordering O of the vertices, G is ordered chordal if there is no chordless cycle C of length at least 4 such that vertices

occur in the same order in C as in O.

Orientation: Assignment of directions to edges of an undirected graph.

Outdegree: The number of edges out of a vertex in a directed graph.

Outerplanar graph: A graph with a planar embedding such that every vertex is on the outer face.

Overlap graph: The overlap graph of a set of objects has a vertex for each object, and an edge between vertices if and only if the objects have a nonempty intersection, and neither object contains the other.

P_i: A chordless path on i vertices.

Parallel module: A module which induces a disconnected graph.

Parent: In a rooted tree, the next vertex on the path to the root is called the parent.

Partial k-tree: A (not necessarily induced) subgraph of a k-tree.

Partially complemented representation: A form of representation in which for each vertex v, you are given either a list of neighbors of v, or a list of nonneighbors of v.

Partial order: Normally, a partial order on a set of elements is a relation which is reflexive, transitive, and antisymmetric. Since this book deals graphs without self-loops, the terms partial order and transitive graph are sometimes viewed as equivalent.

Partition: Division of a set into subsets, such that each element is in exactly one subset. The subsets are often referred to as blocks of the partition.

Partitioning: In general, the process of subdividing blocks of a partition based on a particular rule. In this book, we focus on partitions which correspond to subsets of the vertices of a graph, and a subdivision rule which divides blocks into neighbors and nonneighbors of some vertex x which is not in the block.

Path: A sequence of vertices $v_1, v_2, .., v_n$ such that each v_i is adjacent to v_{i+1}.

Path graph: Intersection graphs of undirected paths in undirected trees. Differs from EPT graphs in the paths which share a common tree node but no common tree edge correspond to adjacent vertices in a path graph, and nonadjacent vertices in an EPT graph.

Path orderable graph: A graph which admits an ordering $v_1 v_2 .. v_n$ such that for all sets of 3 independent vertices with $v_i < v_j < v_k$ in the ordering, every path from v_i to v_k goes through at least one neighbor of v_j.

Pathwidth: The pathwidth problem tries to minimize the size of the maximum clique over all interval completions of a graph. Determining whether a G has an interval completion with maximum clique size k for input variable k is NP-complete.

Pendant vertex: A vertex of degree 1.

Perfect elimination ordering/scheme: These terms, (perfect elimination scheme and perfect elimination ordering are used interchangeably) refer to an elimination scheme defined by successive removal of a simplicial vertex. A graph is chordal if and only if it has a perfect elimination scheme.

Perfect elimination bipartite graph: A graph in which all edges can be removed by repeatedly removing both endpoints of a bisimplicial edge.

Perfect graph: A graph G in which for every induced subgraph H of G, the chromatic number of H is equal to the size of the largest clique in H.

Perfect matching: A matching is perfect if every vertex is an endpoint of some edge in the matching.

Perfectly orderable graph: A graph G which has an ordering O with the property that for every induced subgraph H of G, greedy coloring of H which colors vertices in the order which the vertices appear in O produces an optimal coloring.

Permutation diagram: Intersection model (as described below) of a permutation graph.

Permutation graph: Intersection graph of line segments, where each line segment connects a point on line L_1 with a point on the parallel line L_2.

Phylogeny problem: The problem of finding the tree describing the evolution of distinct species from a common ancestor species. In the perfect phylogeny problem, distinct species are assigned sets of attributes, and you want an evolutionary tree such that each internal node is given a set of attributes, and the set of nodes with a given attribute form a connected subtree of the evolutionary tree.

PI graph: Intersection graphs of triangles such that each triangle has one endpoint on line L_1, and two endpoints on the parallel line L_2.

PI graph*: Intersection graph of triangles such that all endpoints are on two parallel lines L_1, L_2. Differs from PI graphs in that some triangles can have two endpoints on L_1, while others have two endpoints on L_2.

Planar graph: A graph which can be drawn in the plane such that no pair of edges crosses each other.

Polygon-circle graph: Intersection graph of a set of polygons embedded in a circle.

Poset: Alternate name for partial order.

Postorder: A postorder traversal of a tree visits all children of a node before visiting the node itself.

PQ-tree: A tree which is used to represent possible orderings of a set; elements of the set are stored as leaves of the tree. The set of leaf descendants of a particular internal node i must occur consecutively in the ordering. Internal nodes are labeled either P or Q; children of a P node can be permuted in any order with respect to the final output list, while children of a Q node are assigned an ordering, and must appear in either this order or in reverse order in the output list.

Prime: Indecomposable with respect to a particular form of graph decomposition.

Probe interval graph: A graph in which each vertex corresponds to an interval, and each vertex is labeled either as a probe or nonprobe. Two vertices are adjacent if the corresponding intervals have a nonempty intersection, and at least one of the vertices is a probe. These arose from an application in computational biology.

Projective plane: Although the theory of projective planes is quite a broad subject, they are used in only one way in this book. We use the fact that for infinitely many values n, it is possible to create a 0/1 matrix M with n rows and n columns, such that each row has $\Omega(n^{1/2})$ 1 entries, and each pair of rows r_1, r_2 has exactly 1 common column c such that $M[r_1, c] = M[r_2, c] = 1$.

Proper circular-arc graph: Intersection graph of a set of arcs on a circle, such that no arc contains another arc in the set.

Proper interval graph: Intersection graph of a set of intervals on the line, such that no interval contains any other interval. The class is equivalent to unit interval graphs.

Pruning sequence: An elimination scheme defined by successive removal of pendant vertices or one vertex from a pair of twins. G has a pruning sequence if and only if G is distance hereditary.

Pseudoline: Pseudolines connect points in the plane via curves, and have the property that each pair of pseudolines cross each other at most once.

Queue: A data structure which allows the operations insertion and removal of the vertex which has been in the queue for the longest amount of time.

Quotient graph: The quotient graph of a prime module M with respect to substitution decomposition has one vertex for each maximal proper submodule of M, with two vertices adjacent in the quotient graph if there are edges between the corresponding maximal submodules in the original graph.

Random graph: For this book, a random graph is the result of choosing each edge to be in the graph or not in the graph with equal probability.

Realizer: A set of linear extensions of a partial order P which produce P as their common intersection.

Recognition problem: The problem of determining whether a graph is a member of a class of graphs.

Reconstructible vertex: A vertex is reconstructible from a set of information if adjacencies to all other vertices can be deduced from this information. This is used in a more specific manner when dealing with AT-free co-AT-free graphs, where we specify a set of rules which allow the determination of adjacencies to a vertex.

Rectangle number: The rectangle number of G is the minimum number k such that every vertex can be associated with k isooriented rectangles, and x is adjacent to y if and only if some rectangle associated with x has a nonempty intersection with some rectangle associate with y.

Rectangle overlap graph: Intersection graph of isooriented rectangular regions in the plane. We note that a rectangle is considered as a region of space, so that if one rectangle contains another they are considered to have a nonempty intersection.

Reduced matrix: The reduced M' matrix of a Γ-free matrix m is a 0/1 matrix such that 1 entries of M' correspond to entries of M which are 1, and are not redundant 1 entries.

Redundant 1: A redundant 1 in a Γ-free matrix is a 1 entry which would create a Γ if the entry was changed to 0. These 1s are called redundant, since it is unnecessary to store the positions of redundant 1s in our representation of Γ-free matrices.

Refinement: The process of taking a partition of a set, and getting a new partition by dividing sets into subsets.

Representation problem: The problem of finding a form of representation for a class of graphs which satisfies some condition (e.g finding an implicit representation, or finding a space-optimal representation which allows constant time adjacency testing). This is separate from the construction problem, in that it deals only with the existence of such a representation rather than construction of representations for individual graphs.

Representative graph: Alternate name for quotient graph.

Robust algorithm: A robust algorithm for solving a problem P on an input class C is required to always answer P correctly when the input is in class C, and when the input is not in C may either answer P correctly or answer that the input is not in C.

Rooted tree: A rooted tree is a directed graph, which has an underlying tree and a special vertex r called the root. Directions are assigned so that every vertex of the

tree is reachable via a directed path from r.

Sandwich problem: Given a set of edges E_1 and a set of edges E_2 the sandwich problem for a class of graphs asks whether there is any graph G in the class such that G contains every edge from E_1, and every edge in G comes from E_2.

Satisfiability: The problem of determining whether a Boolean formula given in conjunctive normal form has a truth assignment such that the entire formula is evaluated as true. This was the first problem shown to be NP-complete.

Semiorder: A partial order such that each element e can be mapped to a number w(e), and there is a fixed threshold t such that $x < y$ if and only if $w(y) - w(x) > t$.

Separator: Alternate name for cutset.

Series module: A module which induces a subgraph M such that the complement of M is disconnected.

Series-parallel graph: Although there are slightly different definitions co-existing in the literature, in this book a graph is series-parallel if all edges can be removed by a sequence of removal of degree 1 vertices, and removal of degree 2 vertices v with neighbors w, x followed by addition of an edge w, x if such an edge does not exist already.

Series-parallel poset: A partial order which can be constructed from single element partial orders using the operations disjoint union and join, where join makes every element of $P_1 <$ every element of P_2. Equivalent to partial orders with underlying cographs.

Sibling: Vertices in a rooted tree which have a common parent.

Sign pattern: A sign pattern for a sequence $p_1, , .., p_k$ of polynomial functions is a sequence $s_1, .., s_k$ of 1s and -1s such that there is some assignment to variables which makes every p_i positive if and only if $s_i = 1$.

Simple matrix: A matrix such that no pair of columns is identical.

Simple vertex: A vertex v such that neighbors of v induce a clique, and for each pair of neighbors w, x of v either N$(v) \subseteq$ N$[w]$ or N$(w) \subseteq$ N$[v]$.

Simplicial vertex: A vertex v such that N$[v]$ induces a clique.

Sink: A vertex in a directed graph or partial order which has no outedges.

Skew partition: A skew partition of a graph is a partition of the vertex set into four nonempty subset A, B, C, D such that there is an edge from every vertex of A to every vertex of B, and there is no edge from any vertex of C to any vertex of D. Recurrsive decomposition using skew partition finds a skew partition A, B, C, D, and recursively decomposes the subgraphs induced by A, B, C, by A, B, D, by A, C, D,

and by B, C, D.

Source: A vertex in a directed graph or partial order which has no inedges.

Space optimal: A representation of a graph class which has f(n) graphs on n vertices is space optimal if it uses O(log(f(n)) bits to store each graph in the class.

Sparse graph: A graph with a smaller number of edges. The term is informal; in some cases sparse may mean $o(n^2)$, in other cases it could be as restrictive as $O(n)$.

Sphericity: The sphericity of G is the smallest dimension such that G can be represented as the intersection graph of d-dimensional spheres. Every graph is the intersection graph of spheres in some dimension [360]; this contrasts with the fact that there are partial orders which are not containment graphs of spheres in any dimension. [187]

Spider graph: Alternate name for polygon-circle graph.

Split: A partition of vertices of a graph into S_1, S_2 is a split if it meets the following conditions. 1) S_1 and S_2 have at least two vertices. 2) Let S_{1in} of vertices in S_1 with at least one neighbor in S_2, and let S_{2in} be the set of vertices in S_2 with at least one neighbor in S_1. Every vertex of S_{1in} must be adjacent to every vertex of S_{2in}. Note that this is a special case of the original definition of split in [138, 139], which was defined for directed graphs as well as undirected graphs.

Split decomposition: Alternate name for join decomposition.

Split graph: G is a split graph if the vertices of G can be partitioned into S, K, where S induces an independent set in G and K induces a clique in G; i.e., G can be obtained from a bipartite graph by making one color class a clique. Split graphs are exactly those graphs which are both chordal and co-chordal.

Stable set: Alternate name for independent set.

Stack: A data structure which allows the operations insert, and remove the object which has been in the set for the shortest period of time.

Star: $K_{1,i}$ for any value of $i > 1$.

Steiner tree: The Steiner tree problem takes a graph G and a subset S of vertices as input, and asks for the minimum subtree T of G which connects all vertices in S. The decision version of the Steiner tree problem is NP-complete, even in the unweighted case.

Star cutset: A star cutset of a graph is a set C such that G - C is disconnected, and some vertex c in C is adjacent to every other vertex of C.

String graph: Intersection graph of curves in the plane.

Strong perfect graph conjecture: Famous problem in graph theory, stating that G is perfect if and only if G is a Berge graph. A proof of the conjecture has just been announced. [102]

Strongly chordal graph: Although properly speaking this is a characterization rather than the original definition of the class, we will say a graph is strongly chordal if it has an elimination scheme defined by successive removal of simple vertices.

Strongly connected graph: A directed graph in which there is a path from every vertex to every other vertex.

Substitution decomposition: A form of graph decomposition which recursively decomposes a graph into connected components if G is disconnected, connected components of $\overline{G}$ if $\overline{G}$ is disconnected, and maximal proper submodules of G if both G and $\overline{G}$ are connected.

Threshold graph: A graph in which each vertex v van be assigned a weight w(v) and two vertices x, y are adjacent if $w(x) + w(y) > t$ for some fixed threshold t.

Tolerance graph: A graph in which every vertex can be assigned both an interval and a tolerance, such that x and y are adjacent if and only if the intersection of the intervals exceeds the minimum of the two tolerances. Interval graphs correspond to tolerance graphs with tolerance 0. No implicit representation is known for this class.

Topological sort: An ordering of vertices of a directed acyclic graph, such that for every directed edge (x, y), x precedes y in the ordering.

Total interval number: The total interval number of a graph G is the smallest number of intervals such that each interval is associated with exactly one vertex (a single vertex can be associated with multiple intervals), and x is adjacent to y if and only if some interval associated with x has an intersection with some interval associated with y.

Totally balanced: A hypergraph is totally balanced if every cycle of length greater than 2 has an edge containing at least 3 vertices of the cycle. A well-known theorem is that a hypergraph is totally balanced if and only if every subhypergraph is a hypertree.

Transitive closure: Given a directed graph G, the transitive closure of G is a graph on the same vertex set, with an edge from x to y in the transitive closure if and only if there is a path from x to y in G.

Transitive graph: A directed graph G such that whenever there are edges (x, y) and (y, z), (x, z) is also an edge.

Transitive orientation: An assignment of directions to edges of a comparability graph, such that the resulting direction of edges is transitive.

Transitive reduction: Given a directed acyclic graph G, the transitive reduction of G is a graph on the same vertex set, with an edge from x to y if x has an edge to y in G, and there is no path of length greater than 1 from x to y in G.

Trapezoid graph: Intersection graph of trapezoids, where each trapezoid has two endpoints on a line L_1 and two endpoints on a parallel line L_2. Note that L_1 and L_2 are the same for all trapezoidal objects.

Tree: A connected acyclic graph.

Treewidth: The original definition of treewidth deals with a form of tree decomposition. For this book, it is easier to understand a characterization; G has treewidth k if and only if G is a partial k-tree. The fact that minor closed classes which do not contain all planar graphs must have bounded treewidth has deep consequences; see [412]. Determining the treewidth of a graph is NP-complete, although determining whether a graph has treewidth k is polynomial for fixed k. The treewidth problem can also be viewed as minimizing the maximum clique size over all chordal completions of a graph.

Triangle-extendible: A graph is triangle-extendible if the vertices can be ordered so that for every triangle $u < v < w$, every vertex which comes after w and is adjacent to both v and w is also adjacent to u; in other words, for every K_4 - a single edge, the nonadjacent vertices are not both first and last in the ordering.

Twins: A pair of vertices x, y such that $\mathrm{N}(x) = \mathrm{N}(y)$ if x and y are nonadjacent, and $\mathrm{N}[x] = \mathrm{N}[y]$ if x and y are adjacent.

Two-pair: A pair of vertices x, y such that every chordless path between x and y has length 2. Equivalently for a connected graph, removing $\mathrm{N}(x) \cap \mathrm{N}(y)$ disconnects x and y.

Underlying graph: The underlying graph of a directed graph or poset is the graph formed by removing directions from edges.

Uniform cost assumption: The assumption that arithmetic operations involving numbers of polynomial size takes constant time. This is a standard simplifying assumption made in analysis of algorithms.

Unit circular-arc graph: Intersection graphs of arcs with identical lengths in the circle.
Unit disk graph: Intersection graph of disks with equal diameters in the plane.

Unit interval graph: Intersection graphs of intervals of identical size in the line. These are equivalent to proper interval graphs.

Universal graph: A graph G is (vertex-induced) universal for a class C of graphs if every graph in C is an induced subgraph of G. The concept of a universal graph in the non-induced sense also exists in the literature, but does not apply to this book.

Universal vertex: A vertex which is adjacent to every other vertex in the graph.

Vertex cover: A set S of vertices such that every edge of the graph has at least one endpoint in S. Deciding whether G has a vertex cover of input size k is NP-complete. In the optimization version, the problem asks for the size of the smallest vertex cover.

Vertex expansion: The operation of replacing vertex v by a clique C, each member of which has the same adjacency to other vertices of the graph as v.

Vertex multiplication: The operation of replacing vertex v by an independent set I, each member of which has the same adjacency to other vertices of the graph as v.

Visibility graph: A graph in which vertices can be mapped to corners of a polygon, such that vertices are adjacent if they are either neighbors on the polygon, or the line segment connecting the two vertices is entirely inside the polygon.

von Emde Boas trees: A data structure which allows insertion, deletion, and search for predecessor in the current set over a set with elements in the range $1..n$. The trees use $O(\log\log n)$ time per operation rather than the $O(\log n)$ time per operation of such structures as 2-3-trees designed to handle these operations on sets with no upper bound on range.

Warren's theorem: This important theorem, discussed in chapter 4, was designed to give an upper bound on the number of sign patterns of a set of polynomials. In this book, it is used to give an upper bound on number of graphs in a class of graphs. For various classes of graphs, vertices can be made to correspond with sets of variables, and adjacency between vertices x and y is determined by the sign of a polynomial equation on the variables corresponding to x, y, and possibly other vertex variables.

Weak order: A partial order in which every element v can be assigned a number $\mathrm{w}(v)$, and $x < y$ if $\mathrm{w}(x) < \mathrm{w}(y)$. In other words, these are orders which can be derived from total orders by substituting independent sets for elements.

Weakly chordal graph: A graph G with no induced cycles of length greater than 4 in either G or $\overline{G}$.

Weakly triangulated graph: Alternate name for weakly chordal graph.

Well-covered graph: A graph in which all maximal independent sets have the same cardinality.

Width: The width of a partial order is the maximum independent set (also known as antichain in this context) in the underlying graph.

0/1 matrix multiplication: The result of performing matrix multiplication $A \times B$ for matrices A and B with all entries equal to 0 or 1, and replacing all nonzero entries of the result by 1s.

α: If A and B are n by n matrices, the time of the fastest matrix multiplication algorithm for computing $A \times B$ known is written as $O(n^\alpha)$. When this book was written, α was approximately 2.376.

Γ: A pair of rows $r_1 < r_2$ and columns $c_1 < c_2$ in a 0/1 matrix, such that the submatrix induced by these columns has 1s in every entry except position r_2, c_2.

Ω: Intuitively, f(n) is Ω(g(n)) means that f(n) grows at least as quickly as g(n), if constants are ignored. There are several differing precise definitions, which are not equivalent in general but are equivalent for all functions in this book. My preferred definition is that f(n) is Ω(g(n)) if there is a constant c such that f(n) $\geq$ g(n) for infinitely many values of n, but for this book it is sufficient to say that f(n) is Ω(g(n)) whenever g(n) is O(f(n)).

Θ: f(n) is Θ(g(n)) if f(n) is O(g(n)) and g(n) is O(f(n)). This is our notion of equality in order notation, i.e. if constants are ignored.

2-join: Fundamentally, a partition of the vertex set such that edges between the set can be divided into two complete bipartite graphs. There are several variants, depending on whether these complete bipartite graphs are allowed to share vertices, and on conditions for ruling out trivial partitions. Thus, there are definitions which require only that each set contains at least 3 vertices, while others do not allow one side of the partition to be an induced path. Some definitions also require a path within each side of the partition connecting the different complete bipartite graphs.

Survey of Results on Graph Classes

This chapter was suggested by one of the referees, who noted that many algorithms for dealing with specific classes of graphs are discussed in the book, but that these are difficult for a reader interested in a single class of graphs to discover. The number of graph classes and results on these classes are truly overwhelming, so rather than try to organize these in the form of a table with graph classes on one side and all possible optimization problems on the other, I have chosen to give a very brief sketch of what I believe to be some of the most important results and open problems in algorithms and representation for a number of classes of graphs.

An algorithmic survey on variations of the dominating set problem is given in [329]; I will restrict my treatment here to the basic dominating set problem unless there is a particular reason to cover other domination problems. Let me note that proving NP-completeness of minimum fill was a longstanding open problem before being shown in [491]. There is far less work showing that minimum fill is NP-complete for a class than is the case for other problems, given the difficulty of showing NP-completeness at all; if there is no comment on the fill problem, this generally means that there is no polynomial time algorithm to find the minimum fill on this class of graphs, and the class does not seem to be near the border of what we know to be polynomially solvable.

There is an extensive body of literature dealing with parallel algorithms on these graph classes; since my area of research is sequential algorithms, I will leave a survey on parallel algorithms to some other author. As opposed to algorithms discussed in the rest of the text, I did not try to verify that all the algorithms listed here are correct.

There are many important resources available for finding algorithms on graph classes. The book [74] has a table dealing with best time complexities for recognizing a much larger set of graph classes than is discussed here; corrections to this table are currently kept at

`www.informatik.uni-rostock.de/~ab/survey/errata.html`.

At the same site, a very nice program for questions of inclusions between classes of graphs can be found at

`www.informatik.uni-rostock.de/~gdb/isgci/isgci.html`.

A table of graph class inclusions and counterexamples specializing in subclasses of perfect graphs is available on-line at

`www.informatik.hu-berlin.de/~hougardy/paper/classes.html`.

A survey dealing with polynomial algorithms for some fundamental NP-complete optimization problems on various classes of graphs appeared in [282], and some updates appeared in [283]. An ambitious attempt to maintain an on-line reference

list of best known algorithms for problems on graph classes was attempted and some partial results can be seen at

`http://web.cs.ualberta.ca/~stewart/GRAPH/index.html`.

This project became much too large for a single person to maintain; the current plan is to distribute the work to individuals with good knowledge of a particular class of graphs. If you are interested in helping with this project, it would be a great help to all of us!

AT-FREE GRAPHS: There are $2^{\Theta(n^2)}$ graphs without asteroidal triples, so representation is not an issue. A straightforward algorithm can recognize AT-free graphs in $O(n^3)$ time. Using fast matrix multiplication as a subroutine, this can be reduced to $O(n^{2.79})$ time [332]. It will be difficult to reduce the cost to less than the cost of a single matrix multiplication, since the problem is as hard as recognizing triangle-free graphs, as discussed in the text. Polynomial algorithms on asteroidal triple-free graphs include an $O(n^3)$ algorithm for weighted independent set [316], based on the algorithm of [81], an $O(n^3)$ Steiner tree algorithm [26], and an $O(n^6)$ unweighted dominating set algorithm [330]; weighted dominating set is NP-complete on the subclass of co-comparability graphs. Clique and clique cover are NP-complete on this class [81]; these problems separate AT-free graphs in complexity from co-comparability graphs. Treewidth and minimum fill are also NP-complete on AT-free graphs, as a consequence of the original NP-completeness reduction for these problems [24, 491], but are polynomially solvable for the subclass of AT-free co-AT-free graphs [316]. Bandwidth is NP-complete for the subclass of co-bipartite graphs; an approximation algorithm for AT-free graphs graphs with performance ratio 2 is given in [308]. Max cut is also NP-complete for co-bipartite graphs [47], and co-bipartite graphs are isomorphism-complete. Chromatic number and Hamilton cycle are open for AT-free graphs at this time. Feedback vertex set, which is polynomial for co-comparability graphs, does not seem to have been studied for AT-free graphs. A very simple linear time algorithm for finding a dominating pair in an AT-free graph is given in [125]; finding a robust linear time algorithm for this problem is open. Finding a simple intersection representation which exactly characterizes the class is an open problem.

BOUNDED TOLERANCE GRAPHS: The problem of recognizing bounded tolerance graphs in polynomial time is open. An implicit representation can be constructed in polynomial time, thanks to the fact that every bounded tolerance graph is a trapezoid graph. Although most algorithms on these graphs are based upon their containment in larger graph classes, several papers have been written solving problems on bounded tolerance graphs if the model is given as input. There are papers for solving the dominating set and Hamilton cycle problems on bounded tolerance graphs more efficiently ($O(n^2)$ for dominating set and $O(n\log n)$ for Hamilton cycle) than on the larger class of co-comparability graphs if a model is given [3, 4], and a paper giving an $O(n^3 log^2 n)$ algorithm for dominating set in the complement of bounded tolerance graphs [296]. The last paper mentioned is an example of a problem which can be solved efficiently if a model is given as input, but is open if the input is given in adjacency matrix form. I know of no NP-completeness results specifically for bounded tolerance graphs; thus, we know only results implied from

containment of such classes as interval graphs and permutation graphs. Isomorphism is open for bounded tolerance graphs, both with the model given and when it is not.

BOXICITY: Recognition of boxicity k graphs for $k > 1$ is NP-complete. Boxicity k graphs have an implicit representation by definition, but finding an implicit representation in polynomial time is open. Clique can be solved in polynomial time ($O(nlogn)$ time given the model for boxicity 2 [276]) on this class even if the model is not given, since the number of maximal cliques is polynomially bounded. Clique cover [200], 3-coloring [337] and independent set [408] have been shown to be NP-complete on boxicity 2 graphs. Containment of grid graphs and trees shows that Hamilton cycle, dominating set, Steiner tree, and bandwidth are NP-complete for boxicity 2 graphs. An $O(logn)$ approximation algorithm for independent set on boxicity graphs is given in [6]; [174] notes that this achieves the same approximation ratio for weighted independent set. Feedback vertex set is NP-complete on graphs with bounded boxicity, since every planar graph has boxicity at most 3 [458]; this may be open for boxicity 2 graphs. As far as I could determine, the complexity of such problems as max cut, isomorphism, and treewidth is unknown for graphs of bounded boxicity. However, it is possible that some classes for which these problems are known to be difficult have bounded boxicity; for example, I am not sure whether such classes as degree 3 graphs and line graphs can have arbitrarily large boxicity.

CHORDAL GRAPHS: Chordal graphs can be recognized in linear time, as discussed in the text. An intersection representation as subtrees of a tree can also be found in linear time. Many problems are solvable efficiently in polynomial time from the perfect elimination scheme or clique tree model. Weighted clique, weighted independent set, coloring, and clique cover can all be solved in linear time in this fashion. Many problems which are NP-complete on chordal graphs, such as dominating set, Steiner tree, bandwidth [306] max cut [47] and Hamilton cycle [115] are shown to be NP-complete on the smaller class of split graphs; graph isomorphism is also isomorphism-complete on both classes [54]. Treewidth and minimum fill are trivial on chordal graphs. Treewidth can be solved in polynomial time if there are no chordless cycles of length larger than k for any fixed k [52]; maximum weight induced paths can also be found in polynomial time on this superclass of chordal graphs [214]. The weighted feedback vertex problem is solvable in polynomial time for chordal graphs; this comes from the fact that feedback vertex set on a chordal graph corresponds to finding a minimum set of vertices containing at least one vertex from each triangle, and this is a special case of a covering problem solved for chordal graphs in [121].

CHORDAL BIPARTITE GRAPHS: As discussed in the text, chordal bipartite graphs can be recognized in $O(\min\{n^2, mlogn\})$ time. Existence and construction of an implicit representation are open for this class. NP-complete problems on chordal bipartite graphs include dominating set, Hamilton cycle (these two problems can be solved in polynomial time on the subclass of convex graphs [150, 378]) Steiner tree [379, 378], and bandwidth; chordal bipartite graphs are also isomorphism-complete [380]. The feedback vertex problem, which is polynomial for convex graphs [347], does not seem to have been studied for chordal bipartite graphs. Treewidth can be solved in $O(m^\alpha)$ time, where α is the exponent for matrix multiplication [305]; the related problem of pathwidth (corresponding to minimizing the maximum clique

size in an interval completion, rather than a chordal completion, of the input graph) is NP-complete for chordal bipartite graphs [303]. Minimum fill can be solved in $O(n^4)$ time [94]. Many problems, such as matching, computing neighborhood containments, independent set, shortest paths can be solved more simply and efficiently on chordal bipartite graphs than on general bipartite graphs using the Γ-free matrix characterization, as discussed in the text.

CIRCLE GRAPHS: Circle graphs can be recognized in $O(n^2)$ time [442]. This algorithm also constructs the model, which is an implicit representation. Since neighbors of a vertex form a permutation graph, clique is clearly solvable in polynomial time; the best known complexity is $O(n\log n+\min\{m, n\omega\})$ for the unweighted case [364], and $O(n\log n+\min\{n^2, m\log\log n\})$ for the weighted case [20], assuming the model is given as part of the input. It is also not difficult to devise polynomial time independent set algorithms for this class, with $O(n^2)$ the best known time bound. For the unweighted case this is attributed to Buckingham in a 1981 manuscript at Stewart's website listed in the introduction, but I could not find a published version achieving the same time bound earlier than [20], which achieves this bound for the weighted case as well. Dominating set is NP-complete for this class [295]; a $2+\epsilon$ approximation algorithm is given in [277]. The k-coloring problem is NP-complete for $k > 3$ [208, 471]; an $O(n\log n)$ algorithm for the 3-coloring problem on circle graphs is given in [472]. Note that there is some disagreement in the literature on relationship between clique size and chromatic number in circle graphs; the paper [471] claims that chromatic number is at most twice the clique size for circle graphs, but more recent papers by experts in the field give 2^ω as the best known upper bound on chromatic number in terms of clique size [319]. The Hamilton cycle problem is NP-complete on circle graphs [147]; note that this is given as solvable in polynomial time in [282], but this is a case of a misplaced column entry in the table. Bandwidth is NP-complete, since every tree is a circle graph. Schäffer gave Johnson an outline of a Steiner tree algorithm for circle and circular-arc graphs (and thus the problem is listed as polynomially solvable in [282]); [40] will be the first generally available polynomial algorithm for Steiner tree on circle graphs. Other polynomial time solvable problems on circle graphs include minimum fill and treewidth, which can be solved in $O(n^3)$ time [302, 311], and isomorphism, which can be solved in $O(nm)$ time [265]. I was unable to find any work on clique cover for circle graphs; since both clique cover and circle graphs are heavily studied, this may be an interesting open problem.

CIRCULAR-ARC GRAPHS: This is a well-studied class, and many more algorithms have been designed specifically for circular-arc graphs than I will list here. Circular-arc graphs can be recognized by an algorithm which also constructs the model in linear time [365]. Maximum cardinality independent set can be solved in $O(n)$ time, if the input is given as a circularly ordered list of endpoints [225, 363], as can minimum cardinality dominating set [271]. The best known time bound for the weighted dominating set problem is $O(n+m)$ [92]. I know of no paper which specifically solves the weighted independent set problem for circular-arc graphs ([188] solves the problem in $O(n^2)$ time for the superclass of circle trapezoid graphs), though it is obviously solvable from the model in $O(ln)$ time where l is the minimum load on the circle, i.e. the number of arcs passing through the point on the circle which is covered by the fewest arcs. Maximum clique is solvable given the model in $O(n)$ time [14]. Weighted clique can be solved in $O(n\log n+m\log\log n)$ time [433].

Hamilton cycle can be solved in $O(n^2 logn)$ time on circular-arc graphs [432] (note that the linear and $O(n^2)$ time Hamilton circuit algorithms cited as forthcoming in [346] were flawed) and clique cover in $O(n)$ time [271]. Isomorphism can be solved in $O(nm)$ time [265]; at one point, I was part of a paper claiming an $O(n^2)$ bound for this, but our algorithm relied on a result from another paper which may be in error. Coloring is NP-complete on circular-arc graphs, as discussed in the text. It is possible to find a 3/2 optimal coloring [292], and to optimally solve the k-coloring problem for constant k in $O(n)$ time [208]; there is also a randomized polynomial algorithm with an approximation ratio of 1+1/e if the number of colors is not $O(\log n)$ [335]. Bandwidth is NP-complete for circular-arc graphs; an approximation algorithm with (optimal) performance ratio 2 is given in [334]. Other problems solved for circular-arc graphs include an $O(n \log n)$ algorithm for matching when the model is given [348], and $O(n^3)$ algorithms for minimum fill [311] and treewidth [451]. Although I have generally limited myself to coverage of sequential algorithms, note that unweighted independent set, clique cover, and dominating set can be solved optimally given the model in parallel, using $O(\log n)$ time on $O(n/\log n)$ processors [404]. Maximum cut seems to be open for circular-arc graphs, as well as the subclass of interval graphs. The status of Steiner tree is slightly unclear; Schäffer sketched a proof that this is polynomially solvable (for both circle and circular-arc graphs), and thus it appeared as polynomial in the table of [282], though no algorithm solving the problem appears in the general literature. I found no papers dealing with feedback vertex set in circular-arc graphs, but the problem can be solved in polynomial time by reduction to feedback vertex set on interval graphs. Pathwidth is open for circular-arc graphs.

CLIQUE-WIDTH K GRAPHS: The recognition problem can be solved in polynomial time for clique-width 1 (trivial), 2 (equals cograph recognition), and 3 [122], and is open in general and for other fixed values of k. A polynomial time algorithm which constructs an $O(\log n)$ clique-width decomposition tree for any graph with clique-width at most k for fixed k is given in [281]. For any fixed k, an implicit representation can be found in polynomial time using balanced k-homogeneous sets, as discussed in the text. The key result on this class is that any problems posed in monadic second order logic with quantification over vertex sets, but no quantification over edge sets, can be solved in linear time on clique-width k graphs if the decomposition tree is given as part of the input [135]. Problems solvable in this way include independent set, dominating set, i-colorability for fixed i, and various other NP-complete and polynomially solvable problems for general graphs. Certain problems which are not expressible in this form of logic can also be solved on clique-width k graphs; such problems include Hamilton cycle and max cut [478], and chromatic number when the number of colors may be arbitrarily large [313]. If the graph is given in adjacency matrix form, independent set and i-colorability can still be solved in polynomial time [57], but the question of whether all problems posed in the restricted second order monadic logic by robust algorithms remains open. Most other problems have not been investigated for graphs with bounded clique-width. Thus, for example, the complexity of isomorphism on graphs of bounded clique-width would seem to be an open problem, both with the decomposition tree given and for robust algorithms.

COGRAPHS: Cographs are a particularly tractable class of graphs. They can be recognized in linear time, and a cotree can be constructed in the same time bound

[128]. Since this cotree is unique, cograph isomorphism is reduced in $O(n+m)$ time to rooted tree isomorphism, which can be solved in $O(n)$ time. It is also possible to find an implicit representation in linear time. Many results on cographs are a consequence of containments in larger classes of graphs, such as permutation graphs, distance hereditary graphs, and clique-width k graphs, for which many problems are known to be linear time solvable. Nevertheless, many papers have been written about complexity of problems on cographs, since so many problems which are difficult on larger classes of graphs can be solved efficiently on cographs. Examples of tractable problems on cographs include an $O(n^2)$ max cut algorithm [47], and linear time algorithms for minimum fill [129], bandwidth [490], and pathwidth (which equals treewidth for cographs) [51]. The sandwich problem for cographs is polynomially solvable [228], contrasting with the NP-completion of the completion and deletion problems for cographs [168]. Other NP-complete problems on cographs include list coloring [279] and subgraph isomorphism [148].

COMPARABILITY AND CO-COMPARABILITY GRAPHS: Although a transitive orientation can be found in linear time, recognizing comparability graphs is as hard as recognizing triangle-free graphs, currently requiring time proportional to matrix multiplication. The maximum weighted clique and vertex coloring problems can be solved by a robust algorithm in linear time [367], and independent set can be transformed in linear time to bipartite matching once the transitive orientation is given. Many problems on comparability graphs are NP-complete since these contain the bipartite graphs; for example, Hamilton cycle, dominating set, bandwidth, feedback vertex, treewidth [24], minimum fill [301] and Steiner tree are already NP-complete on bipartite graphs, and bipartite graphs are isomorphism-complete. As far as I can determine, the max cut problem, which is trivial on bipartite graphs, is open for comparability graphs.

Since [367] produces a transitive orientation of the complement in linear time, maximum independent set and clique cover can be solved robustly on co-comparability graphs in linear time. Hamilton cycle can be solved in $O(n^3)$ time for co-comparability graphs [154]. More precisely, the algorithm takes $O(in^2)$ time, where i is the size of a maximum independent set in G; this is based on $O(i)$ calls to a Hamilton path algorithm for co-comparability graphs from [149]. The authors of [154] also explain why the algorithm for Hamilton cycle can probably be implemented to run in $O(n^2)$ time, though the full details are not worked out in the paper. Minimum weight Steiner trees in co-comparability graphs can be found in $O(n\log n+m)$ time [78]. The cardinality Steiner tree can be solved in $O(n+m)$ time, as noted independently by Colbourn and Lubiw; the algorithm is described in [333]. Although weighted dominating set is NP-complete on co-comparability graphs [93], the cardinality dominating set problem can be solved in $O(nm^2)$ time [78], and the problem remains solvable in polynomial time for integer weights up to any fixed k [93]. Note that there was a report claiming a running time for dominating set proportional to matrix multiplication, but the algorithm was apparently not correct. There is an $O(n^2m)$ algorithm for the feedback vertex set problem on co-comparability graphs [347]. Bandwidth, minimum fill, max cut and treewidth [308, 491, 47, 24] are NP-complete on co-bipartite graphs, and co-bipartite graphs are isomorphism-complete.

DISK INTERSECTION GRAPHS: Recognition of disk intersection graphs is NP-hard [258]. Existence and construction of an implicit representation are natural open

problems for this class of graphs. Another natural question is whether the clique problem, which is polynomially solvable for unit disk graphs, can be solved for the more general class. Disk graphs contain both unit disk graphs and planar graphs as subclasses, implying NP-completeness of such problems as 3-coloring, independent set, clique cover, Steiner tree, feedback vertex, dominating set, Hamilton cycle, and bandwidth. If the model is given as input, polynomial time approximation schemes for weighted independent set and weighted vertex cover are given in [174]; these also apply to intersection graphs of regular polygons, in any fixed dimension. I know of no exact algorithmic results designed specifically for disk intersection graphs; thus problems which are solvable on planar graphs, such as max cut, minimum fill, and isomorphism, seem to be open for disk intersection graphs. Treewidth also does not seem to have been studied for disk graphs, and is singled out as an interesting open problem for the subclass of planar graphs in [42].

DISTANCE HEREDITARY GRAPHS: Distance hereditary graphs can be recognized in linear time, and an implicit representation can be constructed in the same time bound. However, as noted in [399], no natural intersection model is known for the class, despite the fact that this is an intersection class as discussed in the text. Many problems are solvable efficiently thanks to the pruning sequence, and the fact that these have clique-width at most 3. For example, maximum weighted clique, maximum weight independent set, coloring, and clique cover can be solved in $O(n)$ time if the pruning sequence is given as input, and thus in linear time if the graph is given in adjacency list form [246]. Linear time algorithms for the dominating set problem are given in [386, 97]. Weighted Steiner tree is solvable in linear time for nonnegative edge weights [493], as are the treewidth and minimum fill problems [82], and feedback vertex set [288]. Hamilton cycle can be solved in $O(n^2)$ time on distance hereditary graphs [385]. Isomorphism can be tested in polynomial time for distance hereditary graphs, since every distance hereditary graph is a circle graph. In fact, I believe that isomorphism can be tested in linear time on this class, by reduction to tree isomorphism, but the algorithm only appeared in a technical report, and the report does not give an explicit time bound [29], see also [28]. Since every tree is distance hereditary, bandwidth is NP-complete for the class. Max cut is polynomial on the class since every distance hereditary graph has clique-width at most 3, but I know of no specific algorithm for distance hereditary graphs.

DOMINATION GRAPHS: The most interesting problem for domination graphs is recognition, which remains open. Algorithms on domination graphs generally come either from the definition of the class, or from the fact that every domination graph is weakly chordal, while NP-completeness results follow from containment of classes such as chordal graphs, trapezoid graphs and tolerance graphs.

DOT PRODUCT k GRAPHS: The recognition problem is open for every $k > 1$, as is the problem of existence and construction of an implicit representation. No algorithmic work has been done for $k > 1$, i.e. threshold graphs.

DOUBLY CHORDAL GRAPHS: Doubly chordal graphs can be recognized in linear time, since they are exactly equal to chordal and dually chordal graphs. Although several algorithms have been designed specifically for doubly chordal graphs, the best current bounds for problems discussed in this survey come from results on the superclasses of chordal graphs or dually chordal graphs.

DUALLY CHORDAL GRAPHS: Dually chordal graphs can be recognized in linear time [67]. Many problems are NP-complete on dually chordal graphs, since adding a universal vertex makes any graph dually chordal. Thus, for example, clique, independent set, coloring, clique cover, Hamilton cycle, treewidth, and many other problems can easily be seen to be NP-complete on this class. For some problems, adding a universal vertex would make the problem trivial; the paper [67] shows that a variety of domination and location problems, including unweighted dominating set and Steiner tree (the weighted problems are obviously NP-complete, by adding a universal vertex with appropriate weight) can be solved in linear time on dually chordal graphs. The all pairs shortest path problem can be solved in $O(n^2)$ time on dually chordal graphs [68].

EPT GRAPHS: Recognition is NP-complete for EPT graphs [227]. Although EPT graphs have an implicit representation, constructing an implicit representation in polynomial time from the adjacency matrix of an EPT graph is an open problem. Every line graph is an EPT graph, which implies that such problems as coloring, clique cover, Hamilton cycle, dominating set, and Steiner tree are NP-complete, and that EPT graph isomorphism is isomorphism-complete. The coloring problem on EPT graphs is equivalent to path coloring on undirected tree networks, and has received some attention; Tarjan [454] gave a 3/2 optimal coloring algorithm, and a 4/3 optimal algorithm with asymptotic approximation ratio 11/10 is given in [173]. Weighted clique and weighted independent set are solvable by robust algorithms, as discussed in the text. Other problems not known to be NP-complete for line graphs (these include max cut, which is polynomially solvable for line graphs [239], and bandwidth, which is open for line graphs [280]), seem to be open by way of not having been studied. As noted in [454], every EPT graph is decomposable using clique separators into line graphs; thus, if a problem solution can be constructed from the solutions on clique-separated components, the complexity on EPT graphs will be the same as the complexity on line graphs. For example, this implies that the complexity of minimum fill or treewidth on EPT graphs is equal to the complexity of these problems on line graphs.

INTERVAL FILAMENT GRAPHS: Recognizing interval filament graphs, and creating an interval filament model, are open problems. If the model is given, clique and independent set are solvable on interval filament graphs in polynomial time [211]; these problems are open for robust algorithms.

INTERVAL GRAPHS: The first linear time algorithm for recognizing interval graphs appeared in [56]; many others have appeared, and one is described in the text. The original recognition algorithm can also be used to test isomorphism of interval graphs in linear time [355]. Weighted independent set, weighted clique, coloring, and clique cover can be solved in $O(n)$ time if the intervals are given in sorted order. Hamilton cycle can also be solved in $O(n)$ time given the interval model, though this is more difficult [96]. An $O(n)$ weighted dominating set algorithm given in [92]. Bandwidth can be solved on interval graphs in $O(n\log n)$ time (for the decision problem; finding the optimal bandwidth takes $O(nlog^2n)$ time using this algorithm) [449], and weighted feedback vertex set in $O(n+m)$ time [353]. As far as I can determine, max cut is open for interval graphs, though it can be solved in linear time for unit interval graphs [50].

INTERVAL NUMBER: Recognizing graphs with interval number k is NP-complete for any fixed $k > 1$ [481]. No algorithm is known which constructs an implicit representation of a graph with interval number 2 in polynomial time. Every circular-arc graph has interval number at most 2, so bandwidth and coloring are NP-complete on the class. Every degree 3 graph has interval number at most 2, implying NP-completeness for such problems as independent set and Hamilton cycle, and containment of line graphs implies NP-completeness for clique cover, dominating set, and Steiner tree, as well as isomorphism-completeness. Max cut [47] is also NP-complete for interval number 2. Every planar graph has interval number at most 3 [426], implying results such as NP-completeness of feedback vertex set when interval number is bounded. The real challenge for this class is to use the representation to solve problems effectively; only a few papers, such as [494], which gives a $2k$ times optimal weighted independent set algorithm for graphs with interval number k, attempt to do so. The clique problem seems to be open for graphs with interval number 2, and perhaps for graphs with bounded interval number as well.

INVERSION-FREE GRAPHS: Although these are not a well known class of graphs, I include them in this survey due to the interesting open problems, as described in the text section on robust algorithms. Recognition of the class is open. The clique problem can be solved if a model is given as input, but is open if one is given only a promise that the input is inversion-free. Solving the clique problem with a robust algorithm would also give a robust algorithm for finding a maximum clique in a visibility graph.

K-POLYGON GRAPHS: It is possible to test whether a graph is a k-polygon graph in $O(4^k n^2)$ time, making this polynomial when k is fixed, but recognition is NP-complete if k is allowed to vary [170]. Since k-polygon graphs are a subset of circle graphs, it is possible to construct an implicit representation in polynomial time. In contrast with circle graphs, dominating set (and a variety of other domination problems) are polynomial for any fixed k [169]. Natural problems to study include those which are polynomial for permutation graphs, but open or NP-complete for circle graphs; these include the Hamilton cycle, coloring, and clique cover problems. An approximation algorithm for bandwidth with performance ratio $2k^2$ (the authors note that an improvement to $4/3k^2$ has been communicated to them) for k-polygon graphs is given in [334]; I believe it is open whether bandwidth is NP-complete for every fixed k.

PATH GRAPHS: Path graphs can be recognized in $O(n + m)$ time [144]; the natural intersection model is also produced. This result is not well known, since the paper was published in a proceedings which is relatively difficult to find; the paper [422] gives an $O(nm)$ time algorithm in a more widely distributed journal. Every path graph is chordal, so problems such as clique, coloring, independent set, and clique cover can be solved in linear time. Dominating set [55], max cut [47], Hamilton cycle [38], and bandwidth are NP-complete on path graphs. Hamilton cycle is also NP-complete on the subclass of directed path graphs [381] and open for rooted directed path graphs; dominating set is polynomial on rooted directed path graphs [55]. Note that the math review article of the paper [163] is a bit ambiguous, and might lead the reader (i.e., I read it this way) to believe dominating set is polynomial on path graphs; the class of graphs in the paper is actually the dually chordal graphs. Path graphs are isomorphism-complete [54]; the subclass of directed path graphs is also isomorphism-complete, while isomorphism of rooted directed path graphs

can be tested in polynomial time [25]. Path graphs are separated from split and chordal graphs by the minimum dominating clique problem, which can be solved in polynomial time on path graphs but is NP-complete on split graphs [331]. I was unable to find any work on the Steiner tree problem restricted to path graphs, but I am told that this should be NP-complete as a simple xtension of a proof that connected dominating set is NP-complete for the class [328].

PERFECT GRAPHS: Recognition of perfect graphs is a famous open problem. The difficult, but polynomial, algorithms for the clique, coloring, clique cover, and independent set [237] are robust algorithms for perfect graphs. Perfect graphs include many subclasses, including classes for which such problems as Hamilton cycle, dominating set, Steiner tree, feedback vertex, bandwidth, treewidth, and minimum fill are NP-complete. With the exception of the important paper cited above, there are few results solving NP-complete problems on perfect graphs in general; an $O(n^3)$ combinatorial algorithm for 3-coloring on perfect graphs is given in [470]. One area of algorithmic research is recognition of perfect graphs when restricted to various subclasses. Examples include recognition of perfect graphs without certain induced subgraphs, such as perfect claw-free graphs [111], perfect K_4-e-free graphs [197], perfect bull-free graphs [405], paw-free perfect graphs [389], and dart-free perfect graphs [108]. Other classes for which perfect graph recognition is polynomial include planar graphs [266] and 2-split graphs, i.e. graphs which can be partitioned into two split graphs [262]. Open problems in this area are collected at

`www.cs.rutgers.edu/~chvatal/perfect/problems.html`.

PERFECTLY ORDERABLE GRAPHS: Recognizing perfectly orderable graphs is NP-complete [369]. There are linear time ($O(nm)$ in the weighted case) [259] algorithms for solving chromatic number and maximum clique if a perfect ordering is given as part of the input; if the order is not given, it is open to solve these problems without resorting to more difficult and general algorithms for perfect graphs. Many NP-completeness results follow from the fact that such classes as chordal graphs, comparability graphs, and other classes are perfectly orderable. See [261] for more information on perfectly orderable graphs.

PERMUTATION GRAPHS: Permutation graphs can be recognized in linear time [367], by an algorithm which constructs the natural implicit representation. Independent set and clique can be solved in $O(n\log\log n)$ time if the model is given as input; these are equivalent to finding the longest increasing subsequence of a permutation. The well known algorithm $O(n\log n)$ for finding a longest increasing subsequence can be modified to run in $O(n\log\log n)$ time if input numbers are in the range $1..n$ by using von Emde Boas trees, and can also be easily modified to handle the weighted case. Since permutation graphs are perfect, coloring and clique cover can be solved in the same time bound; a maximum cardinality matching algorithm with the same running time is given in [406]. An $O(n)$ algorithm for finding the minimum cardinality dominating set problem when given the permutation diagram is contained in [98]; an $O(n+m)$ algorithm for the weighted problem is given in [407]. Isomorphism of permutation graphs can be tested in $O(n^2)$ time [447]. Steiner tree can be solved in $O(n)$ time, since every permutation graph is a trapezoid graph, and in $O(n\log n)$ time in the weighted case [275], if the permutation diagram is given. A linear time algorithm for finding a Hamilton cycle in a permutation graph is given in [153]. There is an $O(nt)$ algorithm to determine whether a permutation

graph has treewidth (which equals pathwidth for permutation graphs) at most t [48]. The weighted feedback vertex set problem can be solved in $O(mn)$ time for permutation graphs [343]. The complexity of bandwidth on permutation graphs is an open problem; it is also open for the subclass of bipartite permutation graphs, but can be solved in polynomial time for chain graphs [309]. Max cut also seems to be open for this class of graphs; the the reduction given in [16] intended to show that the problem is NP-complete for permutation graphs is not polynomial.

PI GRAPHS: Recognition of PI graphs in polynomial time is open. Nevertherless, an implicit representation can be constructed in polynomial time, since every PI graph is a trapezoid graph. Although there have been papers written solving algorithmic problems on PI graphs, I know of no problem for which the best algorithm on PI graphs is faster than the time of the best current known algorithm for trapezoid graphs. I also know of no results which explicitly separate the complexity on triangle graphs from trapezoid graphs, and NP-completeness results come only as a consequence of the fact that these contain such classes as permutation graphs and interval graphs.

PROBE INTERVAL GRAPHS: Probe interval graphs can be recognized in linear time if the division of the vertices into probes and nonprobes is given [286, 287], but is open if the partition is not given. No algorithms have been specifically designed for probe interval graphs, so any known algorithms and NP-completeness results come only from containments between graph classes.

SPLIT GRAPHS: Split graphs can be recognized in linear time, and in $O(n)$ time if the degree sequence is give as input [223]. Split graphs have the same complexities as chordal graphs on all problems described under chordal graphs, and often seem to be at the core of algorithms and proofs of difficulty for chordal graphs. The two classes are separated in complexity for a number of problems such as triangle packing, i.e. the problem of finding the maximum number of vertex disjoint triangles. The triangle packing problem is polynomial for split graphs, but NP-complete for chordal graphs [239]; pathwidth is also NP-complete on chordal graphs, and polynomially solvable on split graphs [240]. The sandwich problem (given a set of required and optional edges, is there a set of optional edges which can be included to give a graph in the class) is polynomial for split graphs, in contrast to most of the other classes discussed here; specifically, the sandwich problem is NP-complete for comparability graphs, permutation graphs, co-comparability graphs, circle graphs, interval graphs, circular-arc graphs, path graphs, chordal graphs, and co-chordal graphs, and was left as an open problem for chordal bipartite graphs and strongly chordal graphs in [228]. However, the split graph completion problem and split graph deletion problems are NP-complete [383], while the number of edges which need to be added or deleted to a given graph to obtain a split graph can be computed in polynomial time [249]. Although bandwidth is NP-complete, there is a 2-optimal approximation algorithm for split graphs [306].

STRONGLY CHORDAL GRAPHS: Strongly chordal graphs can be recognized in $O(n^2)$ or $O(m \log n)$ time. No implicit representation is known for this class. Strongly chordal graphs are both chordal and dually chordal, so such problems as independent set, clique, coloring, dominating set, and Steiner tree can be solved in linear time. Unlike both of these classes, weighted dominating set is polynomial

on strongly chordal graphs; the time bound is linear if the strong elimination ordering is given as part of the input [182], and thus is $O(n^2)$ or $O(m\log n)$ if given in adjacency list form. The same time bounds (with elimination order given or not) apply to the feedback vertex set problem, as a consequence of [95] and the correspondence of feedback vertex set and a covering problem as mentioned under chordal graphs. Hamilton cycle is NP-complete on strongly chordal graphs [378].

I believe that strongly chordal graphs are isomorphism-complete by a simple reduction from the isomorphism completeness of chordal bipartite graphs shown in [380]; note that this does not follow directly from the result of [25] showing that directed path graph isomorphism is isomorphism-complete. This paper distinguishes between rooted directed path graphs, in which all edge directions emanate from the root, and directed path graphs, in which edges can be directed both towards the root and away from the root. The first class is a subclass of strongly chordal graphs, while the second class is not (it contains the 4-sun). Unfortunately, both classes are sometimes referred to as directed path graphs, and the paper [282] gives directed path graphs in the first sense as a subclass of strongly chordal graphs, leading to potential confusion. [25] shows that isomorphism is polynomial for the rooted case. Bandwidth is NP-complete for this class, since every tree is strongly chordal. As far as I could determine, max cut is open for strongly chordal graphs.

TOLERANCE GRAPHS: Recognition of tolerance graphs is an open problem, as is existence of an implicit representation. Although it is not obvious, recognition of tolerance graphs is in NP [255]. Most algorithms on tolerance graphs are a consequence of containment in weakly chordal graphs, but a few algorithms have been designed to work on tolerance graphs when a representation is given as part of the input. For example, the clique and coloring problems can be solved in $O(n^2)$ time on tolerance graphs, if the tolerance representation is given [232]. Independent set and clique cover can be solved in linear time given a tolerance representation, using the fact that a co-perfect ordering comes directly from the tolerance representation [229], and that independent set and clique cover can be solved in linear time given a co-perfect ordering [109]. An $O(n^5)$ algorithm for minimum fill on the class of multitolerance graphs, which includes both tolerance and trapezoid graphs, is given in [393]. Bandwidth is NP-complete for tolerance graphs [272]; this is the only example I know of a problem proved to be NP-complete on tolerance graphs which is not known to be NP-complete on a subclass. Many algorithmic problems seem to be open for tolerance graphs, whether the model is given or not. I might choose dominating set and Hamilton cycle as interesting examples, both with the model given and for robust algorithms; these problems are solvable in polynomial time on bounded tolerance graphs and NP-complete on weakly chordal graphs.

TRAPEZOID GRAPHS: Trapezoid graphs can be recognized in $O(n^2)$ time [357], by an algorithm which constructs the natural implicit intersection model. Trapezoid graphs are co-comparability graphs, but a number of algorithms have been designed to solve problems more efficiently than on the larger class. Given the model, the chromatic number, clique cover, weighted independent set, and weighted clique problems can be solved in $O(n\log\log n)$ time [188]; k-coloring can also be solved in $O(kn)$ time [140]. It is also possible to solve the Steiner tree problem in $O(n)$ time [344] given the trapezoidal representation. Minimum weight dominating sets can be found in $O(nm)$ time [345], and the treewidth and minimum fill problems can be solved in $O(n^2)$ time [49]. In general, algorithms for trapezoid graphs

received less attention than algorithms for many of the other graph classes. I found no specific papers dealing with Hamilton cycle, unweighted domination, or isomorphism when restricted to trapezoid graphs; in all these cases, I feel that behavior of trapezoid graphs should be closer to the subclass of permutation graphs than the superclass of co-comparability graphs. In particular, having worked on the related recognition problem for trapezoid graphs, I feel that there is a polynomial time isomorphism algorithm, though none has yet been developed. Along these lines, [190] remarks that the $O(nm)$ algorithm for solving the weighted feedback vertex set problem on permutation graphs [343] can easily be extended to trapezoid graphs. Although [308] gives a 2-approximation algorithm for trapezoid graphs which is simpler and faster than the algorithm for AT-free graphs, the complexity of bandwidth on trapezoid graphs seems to be an open problem; max cut is also apparently open.

TREEWIDTH k: Although computing the treewidth is NP-complete, it is possible to determine whether a graph has treewidth at most k for fixed k in linear time [43], and to find the corresponding tree decomposition. Every problem which can be posed in second order monadic logic can be solved for graphs of bounded treewidth [132]; the class of problems solvable is extended in [59], and algorithms have also been designed for many problems which do not fit in a general framework of solvable problems. There are too many problems which have been solved on graphs of fixed treewidth to deal with here; [44] is a survey devoted to this issue. Problems such as clique, independent set, traveling salesman, and dominating set can be solved in $O(n)$ time on graphs of bounded treewidth. Max cut is $O(n)$ on graphs of bounded treewidth [47]; note that even the weighted case is solvable in linear time, while weighted max cut is NP-complete on any class which contains arbitrarily large cliques. Isomorpism can be tested in polynomial time for graphs of fixed treewidth k, though k appears in the exponent of the running time [45]. Bandwidth is one of the better known problems which is NP-complete on the class, since it remains NP-complete for trees.

UNIT DISK GRAPHS: Unit disk graph recognition is NP-hard [77]; membership in NP is open. Existence and construction of an implicit representation is an open problem. k-coloring for every $k > 2$ is NP-complete for unit disk graphs [234], as is independent set [112]. Dominating set, Hamilton cycle, bandwidth, and Steiner tree are NP-complete for the subclass of grid graphs [112, 278, 157, 207]. Clique is solvable by a robust algorithm in polynomial time, as discussed in the text. Polynomial time approximation schemes (these achieve approximation ratios of $1+\epsilon$ for any ϵ, with running time polynomial in the size of the input and $1/\epsilon$) for independent set, vertex cover and dominating set on unit disk graphs are presented in [273]; extensions for independent set and vertex cover are discussed under disk graphs. Good approximation algorithms for bandwidth and various other vertex ordering problems on intersection graphs of randomly generated unit disks are presented in [156]. [362] gives a coloring algorithm with approximation ratio 3 for unit disk graphs; this can be improved to a ratio of 2 if the clique size is fixed and a model is given as part of the input [233]. There is a considerable body of work dealing with unit-disk graphs as an idealized model for radio transmission, solving broadcasting problems of various types; there is also a body of work using unit disk graphs for map labeling. Since I am not expert in these areas, I will not try to pick out the important results on these subjects. Although there is a great

deal of work dealing with algorithms for specialized problems on unit disk graphs, the complexity of many of the classical graph problems on unit disk graphs seems to be open. Thus, I was not able to determine the complexity of such problems as clique cover, isomorphism, treewidth, minimum fill, feedback vertex set, or max cut when restricted to unit disk graphs, whether the model was given as part of the input or not. Interested readers should note that unit disk graphs are studied under many different names (one common name is geometric graphs) in the literature.

VISIBILITY GRAPHS: Visibility graph recognition is a well known open problem, both in the case when an ordering of vertices on the outside of the polygon is specified, and when any ordering is permissible. No implicit representation is known for this class of graphs. If the model is given, the clique problem can be solved in polynomial time for visibility graphs; this is open for robust algorithms, and is discussed in the text. Independent set and dominating set are NP-complete for visibility graphs, and isomorphism is as hard as for general graphs [349]. Clique cover is also NP-complete for visibility graphs [137]; a logarithmic approximation algorithm is given in [167]. The complexity of coloring visibility graphs is an open problem. Although every visibility graph has a Hamilton cycle, it is open whether you can find a Hamilton cycle in a visibility graph when the input is given in adjacency matrix form.

WEAKLY CHORDAL GRAPHS: Weakly chordal graphs can be recognized in $O(m^2)$ time using two fundamentally different algorithms [36, 256], both of which are discussed in the text. The best known algorithms for independent set, clique cover, coloring, and clique run in $O(nm)$ time for the unweighted case [256] and $O(n^4)$ time for the weighted case [446]. NP-complete problems include dominating set, max cut, Steiner tree, bandwidth, and Hamilton cycle, which are NP-complete on chordal graphs. Treewidth and minimum fill-in can be solved in $O(n^6)$ time on weakly chordal graphs [64]. Feedback vertex does not seem to have been studied for this class. Weakly chordal graphs seem much more general than chordal graphs, but there are relatively few problems known to be tractable on chordal graphs but NP-hard on weakly chordal graphs; the most natural problem I could find of this form is testing whether an input graph is perfectly orderable [260], which is trivial on chordal graphs.

WELL COVERED GRAPHS: Well-covered graphs are co-NP-complete to recognize. As is shown in the text, the independent set problem, which is trivial on well-covered graphs given a promise that the input is in the class, is intractable for robust algorithms. Many problems, such as clique, coloring, Steiner tree, minimum fill, max cut, Hamilton cycle, clique cover, and dominating set are NP-complete on this class, and well-covered graphs are isomorphism-complete, all of which are shown in [419]. It is easy to show that other problems are NP-complete, since adding pendant vertices adjacent to each vertex of G makes any graph well-covered; for example, it is easy to see that treewidth and minimum fill are NP-complete in this way. Although I would not classify it as a significant open problem, it is interesting to note that this transformation cannot be used to show that bandwidth is NP-complete. Therefore, despite the fact that bandwidth is NP-complete on most of the classes above and most optimization problems are hard for well-covered graphs, bandwidth has not been studied for this class and cannot be trivially shown to be NP-complete. Distinctions between complexity of problems on well-covered graphs and problems on subclasses such as very well-covered graphs (in which all maximal

independent sets have size $n/2$, and for which recognition, clique cover, dominating set, and Hamilton cycle are polynomial) are studied in [418].

Bibliography

[1] N. ABBAS, L.K. STEWART, Biconvex Graphs: Ordering and Algorithms, Discrete Applied Mathematics 103, 1-19, 2000

[2] J. ABELLO, H. LIN, S. PISUPATI, On Visibility Graphs of Simple Polygons, Congressus Numerantium 90, 119-128, 1992

[3] G.S. ADHAR, Domination in Bounded Interval-Tolerance Graphs, 7th International Conference on Parallel and Distributed Systems, 29-34, 2000

[4] G.S. ADHAR, Optimal Parallel Algorithms for Cut-Vertices, Bridges and Hamilton Path in Bounded Interval-Tolerance Graphs, 8th International Conference on Parallel and Distributed Systems, 91-98, 2001

[5] P.K. AGARWAL, N. ALON, B. ARONOV, S. SURI, Can Visibility Graphs be Represented Compactly?, Discrete Computational Geometry 12, 347-365, 1994

[6] P.K. AGARWAL, M. VAN KREVELD, S. SURI, Label Placement by Maximum Independent Set in Rectangles, Computational Geometry: Theory and Applications 11, 209-218, 1998

[7] P. AGARWAL, M. SHARIR, S. TOLEDO, Applications of Parametric Searching in Geometric Optimization, Journal of Algorithms 17, 292-318, 1994

[8] A. AGARWAL, S. SURI, The Biggest Diagonal in a Simple Polygon, Information Processing Letters 35, 13-18, 1990

[9] P.K. AGARWAL, M. SHARIR, P. SHOR, Sharp Upper and Lower Bounds for the Length of General Davenport-Schinzel Sequences, Journal of Combinatorial Theory Series A 52, 1989 228-274

[10] A.A. AGEEV, V.L. BERESNEV, Polynomially Solvable Special Cases of Simple Plant Location Problem, Proceedings of the First IPCO Conference, Waterloo University Press, 1-6, 1990

[11] A. V. AHO, J. E. HOPCROFT, J. D. ULLMAN, The Design and Analysis of Computer Algorithms Addison-Wesley Reading, MA 1974

[12] V.E. ALEKSEEV, On the Entropy Values of Hereditary Classes of Graphs, Discrete Mathematical Applications 3, 191-199, 1993

[13] V.E. ALEKSEEV, D.V. KOROBITSYN, Complexity of Some Problems on Hereditary Classes of Graphs, Diskret. Mat. 4, 34-40, 1992

[14] S.G. AKL, B.K. BHATTACHARYA, Computing Maximum Cliques of Circular Arcs in Parallel, Parallel Algorithms and Applications 12, 305-320, 1997

[15] O. ALEVIZOS, J.-D. BOISSONNANT, F.P. PREPARATA, An Optimal Algorithm for the Boundary of a Cell in a Union of Rays, Algorithmica 5, 573-590, 1990

[16] H.H. ALI, N.A. SHERWANI, A. BOALS, On the Max Cut Problem in Permutation Graphs, University of Nebraska Omaha Report UNO-CS-TR-90-6, 1990

[17] N. ALON, E.R. SCHEINERMAN, Degrees of Freedom Versus Dimension for Containment Orders, Order 5, 1988 11-16

[18] R.P. ANSTEE, M. FARBER, Characterizations of Totally Balanced Matrices, Journal of Algorithms 5, 215-230, 1984

[19] R.P. ANSTEE, Z. FÜREDI, Forbidden Submatrices, Discrete Mathematics 62, 225-243, 1986

[20] A. APOSTOLICO, M.J. ATALLAH, S. E. HAMBRUSCH, New Clique and Independent Set Algorithms for Circle Graphs, Discrete Applied Mathematics 36, 1-24, 1992 Erratum: Discrete Applied Mathematics 41, 179-180, 1993

[21] J.C. ARDITTI, Partially Ordered Sets and their Comparability Graphs, their Dimension and their Adjacency, *Problemes Combinatoires et Theorie des Graphes*, 1976, *Proc. Coll. Int. CNRS*, Orsay, France

[22] S.R. ARIKATI, C. PANDU RANGAN, An Efficient Algorithm for Finding a Two-Pair, and its Applications, Discrete Applied Mathematics 31, 71-74, 1991

[23] E.M. ARKIN, D. HALPERIN, K. KEDEM, J.S.B. MITCHELL, N. NAOR, Arrangement of Segments that Share Endpoints: Single Face Results, Discrete Computational Geometry 13, 257-270, 1995

[24] S. ARNBORG, D.G. CORNEIL, A. PROSKUROWSKI, Complexity of Finding Embeddings in a k-tree, SIAM Journal on Algebraic Discrete Methods 8, 277-284, 1987

[25] L. BABEL, I.N. PONOMARENKO, G. TINHOFER, The Isomorphism Problem for Directed Path Graphs and for Rooted Directed Path Graphs, Journal of Algorithms 21, 542-564, 1996

[26] H. BALAKRISHNAN, R. RAJARAMAN, C. PANDU RANGAN, Connected Domination and Steiner Set on Asteroidal Triple-Free Graphs, Lecture Notes in Computer Science 709, WADS 93, 131-141, 1993

[27] H. BANDELT, H.M. MULDER, Distance-Hereditary Graphs, Journal of Combinatorial Theory Series B 41, 182-208, 1986

[28] H.J. BANDELT, A. D'ATRI, M. MOSCARINI, H.M. MULDER, A. SCHULTZE, Operations on Distance-Hereditary Graphs, Instituto di Analisi di Sistimi ed Informatica, Research Report 209, 1988

[29] A. D'ATRI, M. MOSCARINI, H.M. MULDER, On the Isomorphism Problem for Distance-Hereditary Graphs, Econometric Institute Technical Report EI9241/A, Rotterdam School of Economics, 1992

[30] G. DI BATISTA, P. EADES, R. TAMASSIA, I.G. TOLLIS, Graph Drawing, Prentice Hall, Upper Saddle River, New Jersey, 1998

[31] C. BENZAKEN, P.L. HAMMER, D. DE WERRA, Threshold Characterization of Graphs with Dilworth Number Two, Journal of Graph Theory 9, 245-267, 1985

[32] S. BENZER, On the Topology of the Genetic Fine Structure, Proc. Nat. Acad. Sci. USA 45, 1607-1620, 1959

[33] V.L. BERESNEV, A.I. DAVYDOV, On Matrices with Connectedness Properties, Upravlyaemye Sistemy 19, 3-13, 1979

[34] C. BERGE, Farbung von Graphen, deren Samtliche bzw. deren Ungerade Kreise Starr sind, Wiss. Zeitschrift Martin-Luther-Univ. Halle-Wittenburg, 1961, 114

[35] C. BERGE, Hypergraphs, North Holland, Amsterdam, 1989

[36] A. BERRY, J.-P. BORDAT, P. HEGGERNES, Recognizing Weakly Triangulated Graphs by Edge Separability, Nordic Journal of Computing 7, 164-177, 2000

[37] A. BERRY, A. SIGAYRET, C. SINOQUET, Improving Phylogenetic Data by Correcting the 2-Pair Edge-Addition Ordering, manuscript, 2001

[38] A.A. BERTOSSI, M.A. BONUCCELLI, Hamiltonian Circuits in Interval Graph Generalizations, Information Processing Letters 23, 195-200, 1986

[39] B.K. BHATTACHARYA, D. KALLER, An $O(m+n\log n)$ Algorithm for the Maximum Clique Problem in Circular-Arc Graphs, Journal of Algorithms 25, 336-358, 1997

[40] B. BHORE, S. PEMMARAJU, Minimum Cardinality Steiner Forests in Circle Graphs, manuscript, 2002

[41] A. BLUM, New Approximation Algorithms for Graph Coloring, Journal of the ACM 41, 470-516, 1994

[42] H.L. BODLAENDER, Treewidth: Algorithmic Results and Techniques, Technical Report UU-CS-1997-31, Universiteit Utrecht, 1997

[43] H.L. BODLAENDER, A Linear-Time Algorithm for Finding Tree-Decompositions of Small Treewidth, SIAM Journal on Computing 25, 1305-1317, 1996

[44] H.L. BODLAENDER, Treewidth: Algorithmic Techniques and Results, MFOCS 97, Lecture Notes in Computer Science 1295, 19-36, 1997

[45] H. BODLAENDER, Polynomial Algorithms for Graph Isomorphism and Chromatic Index on Partial k-Trees, Journal of Algorithms 11, 631-643, 1990

[46] H.L. BODLAENDER, M.R. FELLOWS, T. WARNOW, Two Strikes Against Perfect Phylogeny, ICALP 92, Lecture Notes in Computer Science 623 273-283, 1992

[47] H.L. BODLAENDER, K. JANSEN, On the Complexity of the Maximum Cut Problem, Nordic Journal of Computing 7, 14-31, 2001

[48] H.L. BODLAENDER, T. KLOKS, D. KRATSCH, Treewidth and Pathwidth of Permutation Graphs, SIAM Journal on Discrete Mathematics 8, 606-616, 1995

[49] H.L. BODLAENDER, T. KLOKS, D. KRATSCH, H. MÜLLER, Treewidth and Minimum Fill-In on d-Trapezoid Graphs, Journal of Graph Algorithms and Applications 2, 1-23, 1998

[50] H.L. BODLAENDER, T. KLOKS, R. NIEDERMEIER, Simple Max-Cut for Unit Interval Graphs and Graphs with few P_4s, manuscript, 2002

[51] H.L. BODLAENDER, R.H. MÖHRING, The Pathwidth and Treewidth of Cographs, SIAM Journal on Discrete Mathematics 6, 181-188, 1993

[52] H.L. BODLAENDER, D.M. THILIKOS, Treewidth for Graphs with Small Chordality, Discrete Applied Mathematics 79, 45-61, 1997

[53] J.A. BONDY, V. CHVÁTAL, A Method in Graph Theory, Discrete Mathematics 15, 111-135, 1976

[54] K.S. BOOTH, C.J. COLBOURN, Problems Polynomially Equivalent to Graph Isomorphism, Report CS-77-04, Computer Science Department, University of Waterloo, 1980

[55] K.S. BOOTH, J.H. JOHNSON, Dominating Sets in Chordal Graphs, SIAM Journal on Computing 11, 191-199, 1982

[56] K.S. BOOTH, G.S. LUEKER Testing for the Consecutive Ones Property, Interval Graphs, and Planarity Using PQ-Tree Algorithms, Journal of Computer and System Science 13, 335-379, 1976

[57] R. BORIE, J. JOHNSON, V. RAGHAVAN, J. SPINRAD, Robust Algorithms for Optimization Problems on Graphs of Bounded Clique-Width, in preparation, 2002

[58] R. BORIE, R.G. PARKER, C.A. TOVEY, A Deterministic Decomposition of Recursive Graph Families, SIAM Journal on Discrete Mathematics 4, 481-501, 1991

[59] R.B. BORIE, R.G. PARKER, C.A. TOVEY, Automatic Generation of Linear-Time Algorithms from Predicate Calculus Descriptions on Recursively Constructed Graph Families, Algorithmica 7, 555-582, 1992

[60] R.B. BORIE, J.P. SPINRAD, Construction of a Simple Elimination Scheme for a Chordal Comparability Graph in Linear Time, Discrete Applied Mathematics 91, 287-292, 1999

[61] P. BOSE, H. EVERETT, S. FEKETE, M.E. HOULE, A. LUBIW, H. MEIJER, K. ROMANIK, G. ROTE, T. SHERMER, S. WHITESIDES, C. ZELLE, A Visibility Representation for Graphs in Three Dimensions, Journal of Graph Algorithms and Applications 2, 1-16, 1998

[62] A. BOUCHET, Circle Graph Obstructions, Journal of Combinatorial Theory Series B 60, 107-144, 1994

[63] A. BOUCHET, Reducing Prime Graphs and Recognizing Circle Graphs, Combinatorica 7, 243-254, 1987

[64] V. BOUCHITTÉ, I. TODINCA, Treewidth and Minimum Fill-In: Grouping the Minimal Separators, SIAM Journal on Computing 31, 212-232, 2001

[65] V. BOUCHITTÉ, I TODINCA, Listing all Potential Maximal Cliques of a Graph, STACS 2000, Lecture Notes in Computer Science 1770, 503-515, 2000

[66] A. BRANDSTÄDT, Classes of Bipartite Graphs Related to Chordal Graphs, Discrete Applied Mathematics 32, 51-60, 1991

[67] A. BRANDSTÄDT, V.D. CHEPOI, F.F. DRAGAN, The Algorithmic Use of Hypertree Structure and Maximum Neighborhood Orderings, Discrete Applied Mathematics 82, 43-77, 1998

[68] A. BRANDSTÄDT, V.D. CHEPOI, F.F. DRAGAN, Clique r-Domination and Clique r-Packing Problems on Dually Chordal Graphs, SIAM Journal on Discrete Mathematics 10, 109-127, 1997

[69] A. BRANDSTÄDT, V.D. CHEPOI, F.F. DRAGAN, V. VOLOSHIN, Dually Chordal Graphs, SIAM Journal on Discrete Mathematics 11, 437-455, 1998

[70] A. BRANDSTÄDT, F.F. DRAGAN, On the Linear and Circular Structure of (claw,net)-free Graphs, manuscript, 1999

[71] A. BRANDSTÄDT, F.F. DRAGAN, E. KÖHLER, Linear Time Algorithms for Hamiltonian Problems on (claw,net)-free Graphs, SIAM Journal on Computing 30, 1662-1677, 2000

[72] A. BRANDSTÄDT, T. KLEMBT, V.B. LE, S. MAHFUD, T. SZYMCZAK, On the Structure and Clique Width of Graphs Defined by Two Forbidden 4-Vertex Graphs, manuscript in preparation, 2002

[73] A. BRANDSTÄDT, V.V. LOZIN, On the Linear Structure and Clique-Width of Bipartite Permutation Graphs, Rutgers Research Report RRR 29-2001, 2001

[74] A. BRANDSTÄDT, V.B. LE, J. SPINRAD Graph Classes: A Survey, SIAM, Philadelphia, 1999

[75] A. BRANDSTÄDT, S. MAHFUD, Maximum Weight Stable Set on Graphs without Claw and Co-Claw (and Similar Graphs Classes) can be Solved in Linear Time, Information Processing Letters, in press, 2002

[76] A. BRANDSTÄDT, J. SPINRAD, L. STEWART, Bipartite Permutation Graphs are Bipartite Tolerance Graphs, Congressus Numerantium 58, 165-174, 1987

[77] H. BREU, D.G. KIRKPATRICK, Unit Disk Graph Recognition is NP-hard, Computational Geometry 9, 3-24, 1998

[78] H. BREU, Algorithmic Aspects of Unit Disk Graphs, PhD Thesis, Computer science, University of British Columbia, 1996, Technical Report UBC CS TR-96-15

[79] G. BRIGHTWELL, On the Complexity of Diagram Testing, Order 10, 297-303, 1993

[80] G. BRIGHTWELL, D.A. GRABLE, H.J. PRÖMEL, Forbidden Induced Partial Orders, Discrete Mathematics 201, 53-80, 1999

[81] H. BROERSMA, T. KLOKS, D. KRATSCH, H. MÜLLER, Independent Sets in Asteroidal Triple-Free Graphs, SIAM Journal on Discrete Mathematics 12, 276-287, 1999

[82] H. BROERSMA, E. DAHLHAUS, T. KLOKS, A Linear Time Algorithm for Minimum Fill-In and Treewidth for Distance Hereditary Graphs, Discrete Applied Mathematics 99, 367-400, 2000

[83] H. BUER, R.H. MOHRING, A Fast Algorithm for the Decomposition of Graphs and Posets, Journal of Mathematics and Operations Research 8, 170-184, 1983

[84] P. BUNEMAN, A Characterization of Rigid Circuit Graphs, Discrete Mathematics 9, 205-212, 1974

[85] R.E. BURKARD, On the Role of Bottleneck Monge Matrices in Combinatorial Optimization, Operations Research Letters 17, 53-56, 1995

[86] L. CAI, The Complexity of Colouring Parameterized Graphs, University of Hong Kong Technical Report CS-TR-1999-07, 1999

[87] K.B. CAMERON, Polyhedral and Algorithmic Ramifications of Antichains, PhD Thesis, University of Waterloo, 1982

[88] K. CAMERON, J. EDMONDS, Existentially Polytime Theorems, Polyhedral Combinatorics, DIMACS Series on Discrete Mathematics and Theoretical Computer Science 1, 83-100, 1990

[89] K. CAMERON, J. EDMONDS, Some Graphic Uses of an Even Number of Odd Holes, Annals Inst. Fourier (Grenoble) 49, 815-827, 1999

[90] M.C. CARLISLE, E.L. LLOYD, On the k-coloring of Intervals, Discrete Applied Mathematics 59, 225-235, 1995

[91] E. CENEK, L. STEWART, Overlap Graphs of Trees, in preparation

[92] M.S. CHANG, Efficient Algorithms for the Domination Problems on Interval and Circular-Arc Graphs, SIAM Journal on Computing 27, 1671-1694, 1998

[93] M.S. CHANG, Weighted Domination on Cocomparability Graphs, Discrete Applied Mathematics 80, 135-148, 1997

[94] M.S, CHANG, Algorithms for Maximum Matching and Minimum Fill-In on Chordal Bipartite Graphs, Lecture Notes in Computer Science 1178, Algorithms and Computation 96, 146-155, 1996

[95] M.S. CHANG, Y.H. CHEN, G.J. CHANG, J.H. YAN, Algorithmic Aspects of the Generalized Clique-Transversal Problem on Chordal Graphs, Discrete Applied Mathematics 66, 189-203, 1996

[96] M.S. CHANG, S.L. PENG, J.L. LIAW, Deferred-Query: An Efficient Approach for Some Problems on Interval Graphs, Networks 34, 1-10, 1999

[97] M.S. CHANG, S.C. WU, G.J. CHANG, H.G. YEH, Domination in Distance-Hereditary Graphs, Discrete Applied Mathematics 116, 103-113, 2002

[98] H.S. CHAO, F.R. HSU, R.C.T. LEE, An Optimal Algorithm for Finding the Minimum Cardinality Dominating Set on Permutation Graphs, Discrete Applied Mathematics 102, 159-173, 2002

[99] B. CHAZELLE, A Polygon Cutting Theorem, Proceedings of the 23d Symposium on Foundations of Computer Science 339-349, 1982

[100] L. CHEN, Y. YESHA, Efficient Parallel Algorithms for Bipartite Permutation Graphs. Networks 22, 29-39, 1993

[101] N. CHIBA, T. NISHIZEKI, Arboricity and Subgraph Listing Algorithms, SIAM Journal on Computing 14, 210-223, 1985

[102] M. CHUDNOVSKY, N. ROBERTSON, P. SEYMOUR, R. THOMAS, The Strong Perfect Graph Theorem, manuscript. 2002
[103] V. CHVÁTAL, A Combinatorial Theorem in Plane Geometry, Journal of Combinatorial Theory Series B, 39-41, 1975
[104] V. CHVÁTAL, Perfectly Ordered Graphs, Annals of Discrete Mathematics 21, 63-65, 1984
[105] V. CHVÁTAL, Recognizing Decomposable Graphs, Journal of Graph Theory 8, 51-53, 1984
[106] V. CHVÁTAL, Star-Cutsets and Perfect Graphs, Journal of Combinatorial Theory Series B 39, 189-199, 1985
[107] V. CHVÁTAL, Which Claw-Free Graphs are Perfectly Orderable?, Discrete Applied Mathematics 44, 39-63, 1993
[108] V. CHVÁTAL, J. FONLUPT, L. SUN, A. ZEMIRLINE, Recognizing Dart-Free Perfect Graphs, SIAM Journal on Computing 31, 1315-1338, 2002
[109] V. CHVÁTAL, C.T. HOÀNG, N.V.R. MAHADEV, D. DE WERRA, Four Classes of Perfectly Orderable Graphs, Journal of Graph Theory 11, 481-485, 1987
[110] V. CHVÁTAL, N. SBIHI, Bull-free Berge Graphs are Perfect, Graphs and Combinatorics 3, 127-139, 1987
[111] V. CHVÁTAL, N. SBIHI, Recognizing Claw-Free Perfect Graphs, Journal of Combinatorial Theory Series B 44, 154-176, 1988
[112] B.N. CLARK, C.J. COLBOURN, D.S. JOHNSON, Unit Disk Graphs, Discrete Mathematics 86, 165-177, 1990
[113] O. COGIS, A Characterization of Digraphs with Ferrers Dimension 2, Res. Report 19, CNRS Paris, 1979
[114] O. COGIS, On the Ferrers Dimension of a Digraph, Discrete Mathematics 38, 47-52, 1982
[115] C.J. COLBOURN, L.K. STEWART, Dominating Cycles in Series Parallel Graphs, Ars Combinatorica 19A, 107-112, 1985
[116] P. COLLEY, A. LUBIW, J. SPINRAD, Visibility Graphs of Towers, Computational Geometry 7, 161-172, 1997
[117] M. CONFORTI, G. CORNUEJOLS, A. KAPOOR, K. VUSKOVIC, Even-Hole-Free Graphs, Part II: Recognition Algorithm, Journal of Graph Theory 40, 238-266, 2002
[118] D. COPPERSMITH, S. WINOGRAD, Matrix Multiplication via Arithmetic Progressions, Journal of Symbolic Computation 9, 1-6, 1990
[119] T. CORMEN, C. LEISERSON, R. RIVEST, Introduction to Algorithms, MIT Press, Cambridge MA, 1989
[120] D.G. CORNEIL, personal communication, 1999
[121] D.G. CORNEIL, J. FONLUPT, The Complexity of Generalized Clique Covering, Discrete Applied Mathematics 22, 109-118, 1988/89
[122] D.G. CORNEIL, M. HABIB, J.M. LANLIGNEL, B. REED, U. ROTICS, Polynomial Time Algorithm for the 3-Clique-Width Problem, Lecture Notes in Computer Science 1776, LATIN 2000, 126-134, 2000
[123] D. G. CORNEIL, P. K. KAMULA, Extensions of Permutation and Interval Graphs, Congressus Numerantium 58, 267-276, 1987
[124] D.G. CORNEIL, H. KIM, S. NATARAJAN, S. OLARIU, A.P. SPRAGUE, Simple Linear Time Recognition of Unit Interval Graphs, Information Processing Letters 55, 99-104, 1995
[125] D. CORNEIL, S. OLARIU, L. STEWART, Linear Time Algorithms for Dominating Pairs in Asteroidal Triple-Free Graphs, SIAM Journal on Computing 28, 1284-1297, 1999
[126] D. CORNEIL, S. OLARIU, L. STEWART, Asteroidal Triple-Free Graphs, SIAM Journal on Discrete Mathematics 10, 399-430, 1997
[127] D. CORNEIL, S. OLARIU, L. STEWART, The Ultimate Interval Graph Recognition Algorithm?, SODA, 1998, corrected manuscript 2000
[128] D.G. CORNEIL, Y. PERL, L.K. STEWART, A Linear Recognition Algorithm for Cographs, SIAM Journal on Computing 14, 926-934, 1985
[129] D.G. CORNEIL, Y. PERL, L.K. STEWART, Cographs: Recognition, Applications, and Algorithms, Congressus Numerantium 43, 249-258, 1984
[130] G. CORNUEJOLS, X. LIU, K. VUSKOVIC, Recognizing Three Classes of Perfect Graphs, in preparation, 2002
[131] C. COULLARD, A. LUBIW, Distance Visibility Graphs, International Journal of Computational Geometry Applications 2, 349-362, 1992

[132] B. COURCELLE, The Monadic Second-Order Logic of Graphs I: Recognizable Sets of Finite Graphs, Information and Computation 85, 12-75, 1990
[133] B. COURCELLE, The Monadic Second-Order Logic of Graphs XIV: Uniformly Sparse Graphs and Edge-Set Quantifications, Theoretical Computer Science, to appear
[134] B. COURCELLE, J. ENGELFRIET, G. ROZENBERG, Handle Rewriting Graph Grammars, Journal of Computer and System Science 46, 218-270, 1993
[135] B. COURCELLE, J.A. MAKOWSKY, U. ROTICS, Linear Time Solvable Optimization Problems on Graphs of Bounded Clique-Width, Theory of Computing Systems 33, 125-150, 2000
[136] A. COURNIER, M. HABIB, A New Linear Algorithm for Modular Decomposition, Lecture Notes in Computer Science 787, 68-84, 1994
[137] J.C. CULBERSON, R.A. RECKHOW, Covering Polygons is Hard, Journal of Algorithms 17, 2-44, 1991
[138] W.H. CUNNINGHAM, Decomposition of Directed Graphs, SIAM Journal on Algebraic Discrete Methods 3, 214-228, 1982
[139] W.H. CUNNINGHAM, J. EDMONDS, A Combinatorial Decomposition Theory, Canadian Journal of Mathematics 22, 734-765, 1980
[140] I. DAGAN, M.C. GOLUMBIC R.Y. PINTER, Trapezoid Graphs and their Coloring, Discrete Applied Mathematics 21, 35-46, 1988
[141] E. DAHLHAUS, Chordale Graphen im besonderen Hinblick auf parallele Algorithmen, Habilitation Thesis, Universität Bonn, 1991
[142] E. DAHLHAUS, Sequential and Parallel Algorithms on Compactly Represented Chordal and Strongly Chordal Graphs, Lecture Notes in Computer Science 1200, 487-498, 1997
[143] E. DAHLHAUS, Parallel Algorithms for Hierarchical Clustering and Applications to Split Decomposition and Parity Graph Recognition, Journal of Algorithms 36, 205-240, 2000
[144] E. DAHLHAUS, G. BAILEY, Recognition of Path Graphs in Linear Time, Theoretical Computer Science (Ravello 1995), 201-210, World Sci. Publishing, River Edge, NJ, 1996
[145] E. DAHLHAUS, J. GUSTEDT, R.M. MCCONNELL, Efficient and Practical Algorithms for Sequential Modular Decomposition, Journal of Algorithms 41, 360-387, 2001
[146] E. DAHLHAUS, J. GUSTEDT, R.M. MCCONNELL, Partially Complemented Representations of Digraphs, Discrete Mathematics and Theoretical Computer Science, to appear
[147] P. DAMASCHKE, The Hamilton Circuit Problem for Circle Graphs is NP-complete, Information Processing Letters 32, 1-2, 1989
[148] P. DAMASCHKE, Induced Subgraph Isomorphism for Cographs is NP-complete, Lecture Notes in Computer Science 484, WG 90, 72-78, 1991
[149] P. DAMASCHKE, J. DEOGUN, D. KRATSCH, G. STEINER, Finding Hamiltonian Paths in Cocomparability Graphs using the Bump Number, Order 8, 383-391, 1991/92
[150] P. DAMASCHKE, H. MÜLLER, D. KRATSCH, Domination in Convex and Chordal Bipartite Graphs, Information Processing Letters 36, 231-236, 1990
[151] X. DENG, P. HELL, J. HUANG, Linear-Time Representation Algorithms for Proper Circular-Arc Graphs and Proper Interval Graphs, SIAM Journal on Computing 25, 390-403, 1996
[152] J.S. DEOGUN, T. KLOKS, D. KRATSCH, H. MÜLLER, On the Vertex Ranking Problem for Trapezoid, Circular-Arc and Other Graphs, Discrete Applied Mathematics 98, 39-63, 1999
[153] J.S. DEOGUN, C. RIEDESEL, Hamiltonian Cycles in Permutation Graphs, Journal of Combinatorial Mathematics and Combinatorial Computing 27, 161-200, 1998
[154] J.S. DEOGUN, G. STEINER, Polynomial Algorithms for Hamilton Cycle in Cocomparability Graphs, SIAM Journal on Computing 23, 520-552, 1994
[155] U. DERIGS, O. GOECKE, R. SCHRADER, Bisimplicial Edges, Gaussian Elimination, and Matchings in Bipartite Graphs, Report 84332-OR, Institut für Operations Research, Universität Bonn
[156] J. DÍAZ, M.D. PENROSE, Approximating Layout Problems on Random Geometric Graphs, Journal of Algorithms 39, 78-116, 2001
[157] J. DÍAZ, M.D. PENROSE, J. PETIT, M.J. SERNA, Linear Orderings of Random Geometric Graphs, Lecture Notes in Computer Science 1665, WG 99, 291-302, 1999
[158] P.F. DIETZ, Intersection Graph Algorithms, Ph.D. Thesis, Comp. Sci. Dept., Cornell University, 1984
[159] R.P. DILWORTH, A Decomposition Theorem for Partially Ordered Sets, Annals of Mathematics Series 2, 161-166, 1950
[160] G.A. DIRAC, On Rigid Circuit Graphs, Abh. Math. Sem. Univ. Hamburg 25, 71-76, 1961.

[161] D. DOBKIN, H. EDELSBRUNNER, Searching for Empty Convex Polygons, Algorithmica 5, 561-571, 1990
[162] J.P. DOIGNON, A. DUCAMP, J. C. FALMAGNE, On Realizable Biorders and the Biorder Dimension of a Relation, Journal of Mathematical Psychology 28, 77-109, 1984
[163] F.F. DRAGAN, K.F. PRISAKAR, V.D. CHEPOI, The Location Problem on Graphs and the Helly Problem, Diskret. Mat. 4, 67-73, 1992
[164] F.F. DRAGAN, V.I. VOLOSHIN, Incidence Graphs of Biacyclic Hypergraphs, Discrete Applied Mathematics 68, 259-266, 1996
[165] G. DURAN, A. GRAVANO, J. SPINRAD, A.C. TUCKER, Polynomial Time Recognition of Unit Circular-Arc Graphs, manuscript, 2002
[166] B. DUSHNICK, E. W. MILLER, Partially Ordered Sets, American Journal of Mathematics 63, 1941, 600-610
[167] S. EIDENBENZ, C. STAMM, Maximum Clique and Minimum Clique Partition in Visibility Graphs, Lecture Notes in Computer Science 1872, IFIP TCS 00, 200-21, 2000
[168] E.S. ELMALLAH, C.J. COLBOURN, The Complexity of Some Edge Deletion Problems, IEEE Transactions on Circuits and Systems 35, 354-362, 1988
[169] E.S. ELMALLAH, L.K. STEWART, Independence and Domination in Polygon Graphs, Discrete Applied Mathematics 44, 65-77, 1993
[170] E.S. ELMALLAH, L.K. STEWART, Polygon Graph Recognition, Journal of Algorithms 26, 101-140, 1998
[171] P. VAN EMDE BOAS, Preserving Order in a Forest in Less than Logarithmic Time and Linear Space, Information Processing Letters 6, 80-82, 1977
[172] P. ERDÖS, D.B. WEST, A Note on the Interval Number of a Graph, Discrete Mathematics 55, 129-133, 1985
[173] T. ERLEBACH, K. JANSEN, The Complexity of Path Coloring and Call Scheduling, Theoretical Computer Science 255, 33-50, 2001
[174] T. ERLEBACH, K. JANSEN, E. SEIDEL, Polynomial-Time Approximation Schemes for General Graphs, Proceedings of 12th ACM-SIAM Symposium on Discrete Algorithms, 671-679, 2001
[175] E. ESCHEN, Circular-Arc Graph Recognition and Related Problems, Ph.D. Thesis, Dept. of Computer Science, Vanderbilt University, 1996
[176] E. ESCHEN, R. HAYWARD, J. SPINRAD, R. SRITHARAN, Weakly Triangulated Comparability Graphs, SIAM Journal on Computing 29, 378-386, 1999
[177] E. ESCHEN, J. SPINRAD, An $O(n^2)$ Algorithm for Circular-Arc Graph Recognition Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms 128-137, 1993
[178] E. ESCHEN, J. SPINRAD, Finding Triangles in Restricted Classes of Graphs, in preparation, 2002
[179] S. EVEN, Graph Algorithms, Computer Science Press, 1979
[180] H. EVERETT, D. CORNEIL, Negative Results on Visibility Graphs, Computational Geometry Theory and Applications 5, 51-63, 1995
[181] H. EVERETT, S. KLEIN, B. REED, An Algorithm for Finding Homogeneous Pairs, Discrete Applied Mathematics 72, 209-218, 1997
[182] M. FARBER, Domination, Independent Domination, and Duality in Strongly Chordal Graphs, Discrete Applied Mathematics 7, 115-130, 1984
[183] M. FARBER, Characterizations of Strongly Chordal Graphs, Discrete Mathematics 43, 173-189, 1983
[184] M. FARBER, Applications of Linear Programming Duality to Problems Involving Independence and Domination, Ph.D. Thesis, Simon Fraser University, 1981
[185] T. FEDER, P. HELL, S. KLEIN, R. MOTWANI, Complexity of Graph Partition Problems, Proceedings of the 31st Symposium on the Theory of Computing 464-472, 1999
[186] S. FELSNER, Tolerance Graphs and Orders, Journal of Graph Theory 28, 129-140, 1998
[187] S. FELSNER, P.C. FISHBURN, W.T. TROTTER, Finite Three-Dimensional Partial Orders which are not Sphere Orders, Discrete Mathematics 201, 101-132, 1999
[188] S. FELSNER, R. MULLER, L. WERNISCH, Trapezoid Graphs and Generalizations, Geometry and Algorithms, Discrete Applied Mathematics, 13-32, 1997
[189] S. FELSNER, V. RAGHAVAN, J. SPINRAD, Recognition Algorithms for Orders of Small Width and Graphs of Small Dilworth Number, manuscript, 2000
[190] P. FESTA, P.M. PARDALOS, M.G.C. RESENDE, Feedback Set Problems, Handbook of Combinatorial Optimization, Supplement Volume A, 209-258, Kluwer, 1999

[191] C.M. FIDUCCIA, E.R. SCHEINERMAN, A. TRENK, J.S. ZITO, Dot Product Representations of Graphs, Discrete Mathematics 181, 113-138, 1998
[192] C.M.H. DE FIGUEIREDO, S. KLEIN, Y. KOHAYAKAWA, B.A. REED, Finding Skew Partitions Efficiently, Journal of Algorithms 37, 505-521, 2000
[193] C.M.H. DE FIGUEIREDO, K. VUSKOVIC, Recognition of Quasi-Meyniel Graphs, Discrete Applied Mathematics 13, 255-260, 2001
[194] P. C. FISHBURN, Interval Orders and Interval Graphs, Wiley, New York, 1985
[195] P.C. FISHBURN, W.T. TROTTER, Angle Orders, Order 1, 333-343, 1985
[196] S. FOLDES, P.L. HAMMER, Split Graphs, Congressus Numerantium 19, 311-315, 1977
[197] J. FONLUPT, A. ZEMIRLINE, A Polynomial Recognition Algorithm for Perfect $K_4 - e$-Free Graphs, Rev. Maghrébine Math 2, 1-26, 1993
[198] L. FORD, D. FULKERSON, Flows in Networks, Princeton University Press, Princeton NJ, 1962
[199] J.L. FOUQUET, J.L. JOLIVET, M. RIVIERE, An $O(n^3)$ Algorithm to Recognize Bipartite P_6-free Graphs, listed as to appear Journal of Algorithms 1994!!
[200] R.J. FOWLER, M.S. PATERSON, S.L. TANIMOTO, Optimal Packing and Covering in the Plane are NP-complete, Information Processing Letters 12, 133-137, 1981
[201] R. FREIVALDS, Fast Probabilistic Algorithms, Lecture Notes in Computer Science 74, MFOCS 79, 57-69, 1979
[202] A. FRANK, T. NISHIZEKI, N. SAITO, H. SUZUKI, E. TARDOS, Algorithms for Routing Around a Rectangle, Discrete Applied Mathematics 40, 363-378, 1992
[203] D.R. FULKERSON, O.A. GROSS, Incidence Matrices and Interval Graphs, Pacific Journal of Mathematics 15, 835-855, 1965
[204] C.P. GABOR, W.L. HSU, K.J. SUPOWIT, Recognizing Circle Graphs in Polynomial Time, Journal of the ACM 36, 435-473, 1989
[205] T. GALLAI, Transitiv Orientbare Graphen, Acta Math. Acad. Sci. Hung. Tom. 18, 25-66, 1967
[206] M.R. GAREY, D.S. JOHNSON, Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman, San Francisco, CA, 1978
[207] M.R. GAREY, D.S. JOHNSON, The Rectilinear Steiner Tree Problem is NP-complete, SIAM Journal on Applied Mathematics 32, 826-834, 1977
[208] M.R. GAREY, D.S. JOHNSON, G.L. MILLER, C.H. PAPADIMITRIOU, The Complexity of Coloring Circular Arcs and Chords, SIAM Journal on Algebraic Discrete Methods 1, 216-227, 1980
[209] A. GARG, R. TAMASSIA, Upward Planarity Testing, Order 12, 109-133, 1995
[210] C. GAVOILLE, D. PELEG, S. PÉRENNES, R. RAZ, Distance Labeling in Graphs, 12th ACM-SIAM Symposium on Discrete Algorithms, 210-219, 2001
[211] F. GAVRIL, Algorithms for a Maximum Clique and a Maximum Independent Set of a Circle Graph, Networks 3, 261-273, 1973
[212] F. GAVRIL, Algorithms on Circular-Arc Graphs, Networks 4, 357-369, 1974
[213] F. GAVRIL, Maximum Weight Independent Sets and Cliques in Intersection Graphs of Filaments, Information Processing Letters 73, 181-188, 2000
[214] F. GAVRIL, Algorithms for Maximum Weight Induced Paths, Information Processing Letters 81, 203-208, 2002
[215] F. GAVRIL, A Recognition Algorithm for the Intersection Graphs of Directed Paths in Directed Trees, Discrete Mathematics 13, 237-249, 1975
[216] F. GAVRIL, The Intersection Graphs of Subtrees of Trees are Exactly the Chordal Graphs, Journal of Combinatorial Theory Series B 16, 47-56, 1974
[217] F. GAVRIL, Algorithms for Maximum k-colorings and k-coverings of Transitive Graphs, Networks 17, 465-470, 1987
[218] S. GHOSH, On Recognizing and Characterizing Visibility Graphs of Simple Polygons, Lecture Notes in Computer Science 318, SWAT 88, 96-104, 1988
[219] V. GIAKOUMAKIS, I. RUSU, Weighted Parameters in P_5,$\overline{P_5}$-free Graphs, Discrete Applied Mathematics 80, 255-261, 1997
[220] P.C. GILMORE, A.J. HOFFMAN, A Characterization of Comparability Graphs and Interval Graphs, Canadian Journal of Mathematics 16, 539-548, 1964
[221] F. GLOVER, Maximum Matching in a Convex Bipartite Graph, Naval Research Logistics Quarterly 14, 313-316, 1967

[222] L. GOH, D. ROTEM, Recognition of Perfect Elimination Bipartite Graphs, Information Processing Letters 15, 179-182, 1982
[223] M.C. GOLUMBIC, Algorithmic Graph Theory and Perfect Graphs, Academic Press, New York, 1980
[224] M.C. GOLUMBIC, C.F. GOSS, Perfect Elimination and Chordal Bipartite Graphs, Journal of Graph Theory 2, 155-163, 1978
[225] M.C. GOLUMBIC, P.L. HAMMER, Stability in Circular-Arc Graphs, Journal of Algorithms 9, 314-320, 1988
[226] M.C. GOLUMBIC, R. E. JAMISON, The Edge Intersection Graphs of Paths in Trees, Journal of Combinatorial Theory Series B 38, 8-22, 1985
[227] M.C. GOLUMBIC, R.E. JAMISON, Edge and Vertex Intersection of Paths in a Tree, Discrete Mathematics 55, 151-159, 1985
[228] M.C. GOLUMBIC, H. KAPLAN, R. SHAMIR, Graph Sandwich Problems, Journal of Algorithms 19, 449-473, 1995
[229] M.C. GOLUMBIC, C.L. MONMA, W.T. TROTTER, Tolerance Graphs, Discrete Applied Mathematics 9, 157-170, 1984
[230] M.C. GOLUMBIC, U. ROTICS, On the Clique-Width of Some Perfect Graph Classes, International Journal on Foundations of Computer Science 11, 423-443, 2000
[231] M.C. GOLUMBIC, E.R. SCHEINERMAN, Containment Graphs, Posets, and Related Classes of Graphs, Annals New York Academy of Sciences 555, 192-204, 1989
[232] M.C. GOLUMBIC, A. SIANI, Coloring Algorithms for Tolerance Graphs: Reasoning and Scheduling with Interval Constraints, proceedings of AISC 2002, to appear
[233] A. GRÄF, Coloring and Recognizing Special Graph Classes, PhD Thesis, Johannes Gutenberg Universität, Mainz, 1994, and report 20/95 in Musikinformatik und MedienTechnik
[234] A. GRÄF, M. STUMPF, G. WEISSENFELS, On Coloring Unit Disk Graphs, Algorithmica 20, 1998
[235] R.L. GRAHAM, B.L. ROTHSCHILD, J.H. SPENCER, Ramsey Theory, Wiley, New York, 1990
[236] J.R. GRIGGS, Extremal Values of the Interval Number of a Graph, II, Discrete Mathematics 28, 37-47, 1989
[237] M. GRÖTSCHEL, L. LOVÁSZ, A. SCHRIJVER, The Ellipsoid Method and its Consequences in Combinatorial Optimization, Combinatorica 1, 169-197, 1981
[238] V. GURUSWAMI, Maximum Cut on Line and Total Graphs, Discrete Applied Mathematics 92, 217-221, 1999
[239] V. GURUSWAMI, C.PANDU RANGAN, M.S. CHANG, G.J. CHANG, C.K. WONG, The K_r-Packing Problem, Computing 66, 79-89, 2001
[240] J. GUSTEDT, On the Pathwidth of Chordal Graphs, Discrete Applied Mathematics 45, 233-248, 1993
[241] W. GUTJAHR, E. WELZL, G. WOEGINGER, Polynomial Graph Colorings, Discrete Applied Mathematics 35, 29-45, 1992
[242] M. HABIB, R. JEGOU, N-free Posets as Generalizations of Series-Parallel Posets, Discrete Applied Mathematics 3, 279-291, 1985
[243] M. HABIB, D. KELLY, R.H. MÖHRING, Interval Dimension is a Comparability Invariant, Discrete Mathematics 88, 211-229, 1991
[244] M. HABIB, M.C. MAURER, On the X-join Decomposition for Undirected Graphs, Discrete Applied Mathematics 3, 198-207, 1979
[245] H. HADWIGER, H. DEBRUNNER, V. KLEE, Combinatorial Geometry in the Plane, Holt Rinehardt and Winston, New York, 1964, p. 54
[246] P.L. HAMMER, F. MAFFRAY, Completely Separable Graphs, Discrete Applied Mathematics 27, 85-100, 1990
[247] P.L. HAMMER, N.V.R. MAHADEV, D. DE WERRA, Stability in CAN-free Graphs, Journal of Combinatorial Theory Series B 38, 23-30, 1985
[248] P.L. HAMMER, N.V.R. MAHADEV, D. DE WERRA, The Struction of a Graph: Application to CN-free Graphs, Combinatorica 5, 141-147, 1985
[249] P.L. HAMMER, B. SIMEONE, The Splittance of a Graph, Combinatorica 1, 275-284, 1981
[250] K. HAN, C.N. SEKHARAN, R. SRIDHAR, Unified All-Pairs Shortest Path Algorithms in the Chordal Hierarchy, Discrete Applied Mathematics 77, 59-71, 1997
[251] D. HAREL, R.E. TARJAN, Fast Algorithms for Finding Nearest Common Ancestors, SIAM Journal on Computing 13, 338-355, 1984

[252] I.B.-A. HARTMAN, I. NEWMAN, R. ZIV, On Grid Intersection Graphs, Discrete Mathematics 87, 41-52, 1991
[253] R. HAYWARD, Weakly Triangulated Graphs, Journal of Combinatorial Theory Series B 39, 200-209, 1985
[254] R. HAYWARD, C. HOÁNG, F. MAFFRAY, Optimizing Weakly Triangulated Graphs, Graphs and Combinatorics 5, 339-349, 1989
[255] R. HAYWARD, R. SHAMIR, Tolerance Graphs are in NP, in preparation
[256] R. HAYWARD, J. SPINRAD, R. SRITHARAN, Weakly Chordal Graph Algorithms via Handles, Proceeding of the 11th Symposium on Discrete Algorithms, 42-49, 2000
[257] P. HLINENY, The Maximal Clique and Colourability of Curve Contact Graphs, Discrete Applied Mathematics 81, 59-68, 1998
[258] P. HLINENY, J. KRATOCHVIL, Representing Graphs by Disks and Balls (A Survey of Recognition-Complexity Results), Discrete Mathematics 229, 101-124, 2001
[259] C.T. HOÀNG, Efficient Algorithms for Minimum Weighted Colouring of Some Classes of Perfect Graphs, Discrete Applied Mathematics 55, 133-143, 1994
[260] C.T. HOÀNG, On the Complexity of Recognizing a Class of Perfectly Orderable Graphs, Discrete Applied Mathematics 66, 219-226, 1996
[261] C.T. HOÁNG, Perfectly Orderable Graphs: A Survey, in Perfect Graphs, Ed. J.L. Ramirez and B.A. Reed, Wiley, Chichester, 139-166, 2001
[262] C.T. HOÀNG, V.B. LE, Recognizing Perfect 2-Split Graphs, SIAM Journal on Discrete Mathematics 13, 48-55, 2000
[263] C.T. HOÁNG, N.V.R. MAHADEV, A Note on Perfect Orders, Discrete Mathematics 74, 77-84, 1989
[264] A.J. HOFFMAN, A.W.J. KOLEN, M. SAKAROVITCH, Totally-Balanced and Greedy Matrices, SIAM Journal on Algebraic Discrete Methods 6, 721-730, 1985
[265] W.L. HSU, O(mn) Algorithms for the Recognition and Isomorphism Problems on Circular-Arc Graphs, SIAM Journal on Computing 24, 411-439, 1995
[266] W.L. HSU, Recognizing Planar Perfect Graphs, Journal of the ACM 34, 255-288, 1987
[267] W.L. HSU, Decomposition of Perfect Graphs, Journal of Combinatorial Theory, Series B 43, 70-94, 1987
[268] W.L. HSU, T.H. MA, Fast and Simple Algorithms for Recognizing Chordal Comparability Graphs and Interval Graphs, SIAM Journal on Computing 28, 1004-1020, 1999
[269] W.L. HSU, W.K. SHIH, An Approximation Algorithm for Coloring Circular-Arc Graphs, manuscript, 1996
[270] W.L. HSU, J. SPINRAD, Independent Sets in Circular-Arc Graphs, Journal of Algorithms 19, 145-160, 1995
[271] W.L. HSU, K.H. TSAI, Linear Time Algorithms on Circular-Arc Graphs, Information Processing Letters 40, 123-129, 1991
[272] L.T.Q. HUNG, M.M. SYSLO,M.L. WEAVER,D.B. WEST, Bandwidth and Density for Block Graphs, Discrete Mathematics 189, 163-176, 1998
[273] H.B. HUNT, M.V. MARATHE, V. RADHAKRISHNAN, S.S. RAVI, D.J. ROSENKRANTZ, NC-Approximation Schemes for NP- and PSPACE-Hard Problems on Geometric Graphs, Journal of Algorithms 26, 238-274, 1998
[274] T. IBARAKI, U. N. PELED, Sufficient Conditions for Graphs to have Threshold Dimension 2, Studies on Graphs and Discrete Programming, Annals of Discrete Mathematics 11, 241-268, 1981
[275] O.H. IBARRA, Q. ZHENG, Some Efficient Algorithms for Permutation Graphs, Journal of Algorithms 16, 453-469, 1994
[276] H. IMAI, T. ASANO, Finding the Connected Components and Maximum Clique of an Intersection Graph of Rectangles in the Plane, Journal of Algorithms 4, 310-323, 1983
[277] M.D. IORDACHE, A 2+ϵ-Approximation Scheme for Minimum Domination on Circle Graphs, Journal of Algorithms 42, 255-276, 2002
[278] A. ITAI, C.H. PAPADIMITRIOU, J.L. SZWARCFITER, Hamilton Paths in Grid Graphs, SIAM Journal on Computing 11, 676-686, 1982
[279] K. JANSEN, P. SCHEFFLER, Generalized Coloring for Tree-like Graphs, Discrete Applied Mathematics 75, 135-155, 1997
[280] T, JIANG, D. MUBAYI, A. SASTRI, D.B. WEST, Edge-Bandwidth of Graphs, SIAM Journal on Discrete Mathematics 12, 307-316, 1999

[281] O. JOHANSSON, log$n$-Approximative NLC_k-Decomposition in $O(n^{2k+1})$ Time, Lecture Notes in Computer Science 2204, WG 01, 229-240, 2001
[282] D.S. JOHNSON, The NP-Completeness Column: An Ongoing Guide, Journal of Algorithms 6, 434-451, 1985
[283] D.S. JOHNSON, The NP-Completeness Column: An Ongoing Guide: Announcements, Updates, and Greatest Hits, Journal of Algorithms 8, 438-448, 1987
[284] D.S. JOHNSON, Worst Case Behavior of Graph Coloring Algorithms, Proceedings of the 5th Southeastern Conference on Combinatorics, Graph Theory and Computing, Congressus Numerantium 10, 513-528, 1974
[285] D.S. JOHNSON, M. YANNAKAKIS, C.H. PAPADIMITRIOU, On Generating All Maximal Independent Sets, Information Processing Letters 27, 119-123, 1988
[286] J. JOHNSON, R. MCCONNELL, J. SPINRAD, Linear Time Recognition of Probe Interval Graphs, in preparation, 2002
[287] J. JOHNSON, J. SPINRAD, A Polynomial Time Recognition Algorithm for Probe Interval Graphs, Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms, 477-486, 2001
[288] C.M.L. JONES, The Weighted Feedback Vertex Set Problem on Distance Hereditary Graphs, MS Thesis, Cung Cheng University, 2000
[289] D. JOSEPH, J. MEIDANIS, P. TIWARI, Determining DNA Sequence Similarity Using Maximum Independent Set Algorithms for Interval Graphs, Lecture Notes in Computer Science 621, SWAT 92, 326-337, 1992
[290] S. KANNAN, M. NAOR, S. RUDICH, Implicit Representation of Graphs, SIAM Journal on Discrete Mathematics 5, 596-603, 1992
[291] S. KANNAN, T. PROEBSTING, Register Allocation in Structured Programs, Journal of Algorithms 29, 223-237, 1998
[292] I.A. KARAPETJAN, Coloring of Arc Graphs, Akad. Nauk Armyan SSR Dokl 70, 306-311, 1980
[293] T. KASHIWABARA, S. MASUDA, K. NAKAJIMA, T. FUJISAWA, Polynomial Time Algorithms on Circular-Arc Overlap Graphs, Networks 21, 195-203, 1991
[294] M. KATZ, N. KATZ, D. PELEG, Distance Labeling Schemes for Well-Separated Graph Classes, Lecture Notes in Computer Science 1770, STACS 00, 516-528, 2000
[295] J.M. KEIL, The Complexity of Domination Problems in Circle Graphs, Discrete Applied Mathematics 42, 51-63, 1993
[296] M. KEIL, P. BELLEVILLE, Dominating the Complements of Bounded Tolerance Graphs and Trapezoid Graphs, manuscript, 2000
[297] H.A. KIERSTEAD, W.T. TROTTER, J. QIN, The Dimension of Cycle-Free Orders, Order 9, 103-110, 1992
[298] D.J. KLEITMAN, K.J. WINSTON, The Asymptotic Number of Lattices, Annals of Discrete Mathematics 6, 243-249, 1980
[299] B. KLINZ, R. RUDOLF, G. WOEGINGER, Permuting Matrices to Avoid Forbidden Submatrices Discrete Applied Mathematics 60, 223-248, 1995
[300] B. KLINZ, R. RUDOLF, G.J. WOEGINGER, On the Recognition of Permuted Bottleneck Monge Matrices, Discrete Applied Mathematics 63, 43-74, 1995
[301] T. KLOKS, Treewidth – Computations and Approximations, Springer Verlag, Lecture Notes in Computer Science 842, 1994
[302] T. KLOKS, Treewidth of Circle Graphs, Lecture Notes in Computer Science 762, Algorithms and Computation 93, 108-117, 1993
[303] T. KLOKS, H. BODLAENDER, H. MÜLLER, D. KRATSCH, Computing Treewidth and Minimum Fill-in: all you Need are the Minimal Separators, Lecture Notes in Computer Science 726, ESA 93, 260-271, 1993
[304] T. KLOKS, D. KRATSCH, Listing all Minimal Separators of a Graph, SIAM Journal on Computing 27, 605-613, 1998
[305] T. KLOKS, D. KRATSCH, Treewidth of Chordal Bipartite Graphs, Journal of Algorithms 19, 266-281, 1995
[306] T. KLOKS, D. KRATSCH, Y. LEBORGNE, H. MÜLLER, Bandwidth of Split and Circular Permutation Graphs, Lecture Notes in Computer Science 1928, WG 00, 243-254, 2000
[307] T. KLOKS, D. KRATSCH, H. MÜLLER, Finding and Counting Small Induced Subgraphs Efficiently, Lecture Notes in Computer Science 1017, WG 95, 14-23, 1995

[308] T. KLOKS, D. KRATSCH, H. MÜLLER, Approximating the Bandwidth for Asteroidal Triple-Free Graphs, Journal of Algorithms 32, 41-57, 1999
[309] T. KLOKS, D. KRATSCH, H. MÜLLER, Bandwidth of Chain Graphs, Information Processing Letters 68, 313-315, 1998
[310] T. KLOKS, D. KRATSCH, J. SPINRAD, On Treewidth and Minimum Fill-in of Asteroidal Triple-free Graphs, Theoretical Computer Science 175, 309-335, 1997
[311] T. KLOKS, D. KRATSCH, C.K WONG, Minimum Fill-In on Circular and Circular-Arc Graphs, Journal of Algorithms 28, 272-289, 1998
[312] T. KLOKS, H. MÜLLER, C.K. WONG, Vertex Ranking of Asteroidal Triple-free Graphs, Information Processing Letters 68, 201-206, 1998
[313] D. KOBLER, U. ROTICS, Edge Dominating Sets and Colorings on Graphs with Fixed Clique-Width, Discrete Applied Mathematics, in press, 2002
[314] M. KOEBE, Colouring of Spider Graphs, Topics in Combinatorics and Graph Theory, Physica Verlag, Heidelberg, 1990, 435-441
[315] M. KOEBE, On a New Class of Intersection Graphs, unpublished manuscript, includes results published under the same title in Fourth Czechoslovakian Symposium on Combinatorics, Graphs, and Complexity, Elsevier, 141-143, 1992, as well as a proposed recognition algorithm
[316] E. KÖHLER, Graphs without Asteroidal Triples, Ph.D. Thesis, TU-Berlin, 1999
[317] E. KÖHLER, D.G. CORNEIL, S. OLARIU, L. STEWART, On Subfamilies of AT-free Graphs, Lecture Notes in Computer Science 2204, WG 01, 241-253, 2001
[318] N. KORTE, R.H. MÖHRING, Transitive Orientation of Graphs with Side Constraints, Lecture Notes on Computer Science 246, WG 85, 1-16, 1987
[319] A. KOSTOCHKA, J. KRATOCHVÍL, Covering and Coloring Polygon-Circle Graphs, Discrete Mathematics 163, 299-305, 1997
[320] J. KRATOCHVÍL, A Special Planar Satisfiability Problem and a Consequence of its NP-completeness, Discrete Applied Mathematics 52, 233-252, 1994
[321] J. KRATOCHVÍL, k-Ramsey Classes and Dimension of Graphs, Commentationes Mathematicae Universitatis Carolinae 36, 263-269, 1995
[322] J. KRATOCHVÍL, Intersection Graphs of Noncrossing Arc-Connected Sets in the Plane, Lecture Notes in Computer Science 1190, Graph Drawing 96, 257-270, 1996
[323] J. KRATOCHVÍL, A. KUBENA, On Intersection Representations of Co-Planar Graphs, Discrete Mathematics 178, 251-255, 1998
[324] J. KRATOCHVÍL, J. MATOUSEK, NP-hardness Results for Intersection Graphs, Commentationes Mathematicae Universitatis Carolinae 30, 761-773, 1989
[325] J. KRATOCHVÍL, J. MATOUSEK, Intersection Graphs of Segments, Journal of Combinatorial Theory Series B 62, 289-315, 1994
[326] J. KRATOCHVÍL, J. NESETRIL, INDEPENDENT SET and CLIQUE Problems in Intersection-Defined Classes of Graphs, Commentationes Mathematicae Universitatis Carolinae 31, 85-93, 1990
[327] J. KRATOCHVÍL, Z. TUZA, Intersection Dimensions of Graph Classes, Graphs and Combinatorics 10, 159-168, 1994
[328] D. KRATSCH, personal communication, 2002
[329] D. KRATSCH, Algorithms, from Domination in Graphs, Monograph Textbooks in Pure and Applied Mathematics 209, 191-231, 1998
[330] D. KRATSCH, Domination and Total Domination on Asteroidal Triple-Free Graphs, Discrete Applied Mathematics 99, 111-123, 2000
[331] D. KRATSCH, Finding Dominating Cliques Efficiently in Strongly Chordal Graphs and Undirected Path Graphs, Discrete Mathematics 86, 225-238, 1990
[332] D. KRATSCH, J. SPINRAD, Between $O(nm)$ and $O(n^\alpha)$, to appear, SODA 2003
[333] D. KRATSCH, L. STEWART, Domination of Cocomparability Graphs, SIAM Journal on Discrete Mathematics 6, 400-417, 1993
[334] D. KRATSCH, L. STEWART, Approximating Bandwidth by Mixing Layouts of Interval Graphs, SIAM Journal on Discrete Mathematics 15, 435-449, 2002
[335] V. KUMAR, An Approximation Algorithm for Circular-Arc Colouring, Algorithmica 30, 406-417, 2001
[336] L.J. LANGLEY, A Note on Bipartite Interval Tolerance Graphs, Congressus Numerantium 102, 191-192, 1994

[337] D.T. LEE, J.Y.-T. LEUNG, On the 2-Dimensional Channel Assignment Problem, IEE Transactions on Computing, C-33, 2-6, 1984
[338] J. VAN LEEUWEN, Graph Algorithms, in Handbook of Theoretical Computer Science Volume A, ed. J. van Leeuwen, 525-631, Elsevier Science Publishers, Amsterdam, MIT Press, Cambridge, MA, 1990
[339] J. LEHEL, A Characterization of Totally Balanced Hypergraphs, Discrete Mathematics 57, 59-65, 1985
[340] P.G.H. LEHOT, An Optimal Algorithm to Detect a Line Graph and Output its Root Graph, Journal of the ACM 21, 569-575, 1974
[341] C.G. LEKKERKERKER, J. BOLAND, Representation of a Finite Graph by a Set of Intervals on the Line, Fund Math. 51, 45-64, 1962
[342] H. Lerchs, On Cliques and Kernels, Technical Report, Dept. of Compueter Science, University of Toronto, 1971
[343] Y.D. LIANG, On the Feedback Vertex Set Problem in Permutation Graphs, Information Processing Letters 52, 123-129, 1994
[344] Y.D. LIANG, Steiner Set and Connected Domination in Trapezoid Graphs, Information Processing Letters 56, 101-108, 1995
[345] Y.D. LIANG, Dominations in Trapezoid Graphs, Information Processing Letters 52, 309-315, 1994
[346] Y.D. LIANG, N. BLUM, Circular Convex Graphs: Maximum Matching and Hamiltonian Circuits, Information Processing Letters 56, 215-219, 1995
[347] Y.D. LIANG, M.S. CHANG, Minimum Feedback Vertex Sets in Cocomparability Graphs and Convex Bipartite Graphs, Acta Informatica 34, 337-346, 1997
[348] Y.D. LIANG, C. RHEE, Finding a Maximum Matching in a Circular-Arc Graph, Information Processing Letters 45, 185-190, 1993
[349] Y-L LIN, S. SKIENA, Complexity Aspects of Visibility Graphs, International Journal of Computational Geometry 5, 289-312, 1995
[350] X. LIU. K. VUSKOVIC, Perfection-Preserving Decompositions, in preparation, 2002
[351] R.-D. LOU, M. SARRAFZADEH, Circular Permutation Graph Family with Applications, Discrete Applied Mathematics 40, 433-457, 1992
[352] L. LOVASZ, Normal Hypergraphs and the Perfect Graph Conjecture, Discrete Mathematics 2, 253-267, 1972
[353] C.L. LU, C.Y. TANG, A Linear-Time Algorithm for the Weighted Feedback Vertex Problem on Interval Graphs, Information Processing Letters 61, 107-111, 1997
[354] A. LUBIW, Doubly Lexical Orderings of Matrices, SIAM Journal on Computing 16, 854-879, 1987
[355] G.S. LUEKER, K.S. BOOTH, A Linear Time Algorithm for Deciding Interval Graph Isomorphism, Journal of the ACM 26, 183-195, 1979
[356] T.-H. MA, J. SPINRAD, Cycle-free Partial Orders and Chordal Comparability Graphs, Order 8, 49-61, 1991
[357] T.-H. MA, J. SPINRAD, On the 2-Chain Subgraph Cover and Related Problems, Journal of Algorithms 1994, 251-268
[358] T.-H. MA, J. SPINRAD, Transitive Closure for Restricted Classes of Partial Orders, Order 8, 175-183, 1991
[359] T.-H. MA, J. SPINRAD, An $O(n^2)$ Algorithm for the Undirected Split Decomposition, Journal of Algorithms 16, 264-282, 1994
[360] H. MAEHARA, Space Graphs and Sphericity, Discrete Applied Mathematics 7, 55-64, 1984
[361] N.V.R. MAHADEV, U.N. PELED, Threshold Graphs and Related Topics, Annals of Discrete Mathematics 56, North Holland, 1995
[362] M.V. MARATHE, H. BREU, H.B. HUNT, S.S. RAVI, D.J. ROSENKRANTZ, Simple Heuristics for Unit Disk Graphs, Networks 25, 59-68, 1995
[363] S. MASUDA, K. NAKAJIMA, An Optimal Algorithm for Finding a Maximum Independent Set of a Circular-Arc Graph, SIAM Journal on Computing 17, 41-52, 1988
[364] S. MASUDA, K. NAKAJIMA, T. KASHIWABARA, T. FUJISAWA, Efficient Algorithms for Finding Maximum Cliques of an Overlap Graph, Networks 20, 157-171, 1990
[365] R.M. MCCONNELL, Linear-Time Recognition of Circular-Arc Graphs, Proceedings of the 42d IEEE Symposium on Foundations of Computer Science, 386-394, 2001

[366] R.M. McConnell, J. Spinrad, Linear-time Modular Decomposition and Efficient Transitive Orientation of Comparability Graphs, Proceedings of the 5th ACM-SIAM Symposium on Discrete Algorithms, 536-545, 1994
[367] R.M. McConnell, J. Spinrad, Modular Decomposition and Transitive Orientation, Discrete Mathematics 201, 189-241, 1999
[368] T.A. McKee, F.R. McMorris, Topics in Intersection Graph Theory, SIAM, Philadelphia, 1999
[369] M. Middendorf, F. Pfeiffer, On the Complexity of Recognizing Perfectly Orderable Graphs, Discrete Mathematics 80, 1990, 327-333
[370] G.J. Minty, On Maximal Independent Sets of Vertices in Claw-free Graphs, Journal of Combinatorial Theory Series B 28, 284-304, 1980
[371] J. Mitas, Minimal Representation of Semiorders with Intervals of Same Length, Lecture Notes in Computer Science 831, Ordal 94, 162-175, 1994
[372] R.H. Möhring, Almost all Comparability Graphs are UPO, Discrete Mathematics 50, 63-70, 1984
[373] R.H. Möhring, F.J. Radermacher, Substitution Decomposition for Discrete Structures and Connections with Combinatorial Optimization, Annals of Discrete Mathematics 19, 257-356, 1984
[374] C.L. Monma, V.K. Wei, Intersection Graphs of Paths in a Tree, Journal of Combinatorial Theory Series B 41, 141-181, 1986
[375] M. Moscarini, Doubly Chordal Graphs, Steiner Trees, and Connected Domination, Networks 23, 59-69, 1993
[376] J.H. Muller, Local Structure in Graphs Ph.D. Thesis, School of Information and Computer Science, Georgia Institute of Technology Atlanta, GA, 1988
[377] J.H. Muller, J. Spinrad, Incremental Modular Decomposition, Journal of the ACM 36, 1-19, 1989
[378] H. Müller, Hamilton Circuits in Chordal Bipartite Graphs, Discrete Mathematics 156, 291-298, 1996
[379] H. Müller, A. Brandstädt, The NP-completeness of Steiner Tree and Dominating Set for Chordal Bipartite Graphs, Theoretical Computer Science 53, 257-265, 1987
[380] T. Nagoya, R. Uehara, S. Toda, Completeness of Graph Isomorphism Problem for Bipartite Graph Classes, IEICE Technical Report COMP2001-93, 2001
[381] G. Narasimhan, A Note on the Hamilon Circuit Problem on Directed Path Graphs, Information Processing Letters 32, 167-170, 1989
[382] G. Narasimhan, R. Manber, Stability and Chromatic Number of Tolerance Graphs, Discrete Applied Mathematics 36, 47-56, 1992
[383] A. Natanzon, R. Shamir, R. Sharan, Complexity Classification of Some Edge Modification Problems, Discrete Applied Mathematics 113, 109-128, 2001
[384] J. Nesetril, S. Poljak, On the Complexity of the Subgraph Problem, Commentationes Mathematicae Universitatis Carolinae 26, 415-419, 1985
[385] F. Nicolai, Hamiltonian Problems on Distance-Hereditary Graphs, Technical Report, Gerhard-Mercator-Universitat Gesamthochschule Duisburg SM-DU-255, 1994
[386] F. Nicolai, T. Szymczak, Homogeneous Sets and Domination: A Linear Time Algorithm for Distance Hereditary Graphs, Networks 37, 117-128, 2001
[387] C. Nomikos, Path Coloring Problems in Graphs are Non-Approximable, manuscript, 2000
[388] T. Ohtsuki, A Fast Algorithm for Finding an Optimal Ordering for Vertex Elimination on a Graph, SIAM Journal on Computing 5, 133-145, 1976
[389] S. Olariu, Paw-Free Graphs, Information Processing Letters 28, 53-54, 1998
[390] J. O'Rourke, Art Gallery Theorems and Algorithms, Oxford University Press, 1987
[391] R. Paige, R.E. Tarjan, Three Partition Refinement Algorithms, SIAM Journal on Computing 16, 973-989, 1987
[392] C.H. Papadimitriou, M. Yannakakis, Scheduling Interval-Ordered Tasks, SIAM Journal on Computing 8, 405-409, 1979
[393] A. Parra, Triangulating Multitolerance Graphs, Discrete Applied Mathematics 84, 183-197, 1998
[394] M.C. Paull, S.H. Unger, Minimizing the Number of States in Incompletely Specified Sequential Switching Functions, IRE Trans. Electronic Computers EC-8, 356-367, 1959
[395] C. Payan, Perfectness and Dilworth Number, Discrete Mathematics 44, 229-230, 1983

[396] D. PELEG, Proximity Preserving Labeling Schemes and their Applications, Journal of Graph Theory 33, 167-176, 2000
[397] U. PFERSCHY, R. RUDOLF, G.J. WOEGINGER, Monge Matrices Make Maximization Manageable, Operations Research Letters 16, 245-254, 1994
[398] M.D. PLUMMER, Well-Covered Graphs: A Survey, Quaestiones Mathematicae 16, 253-287, 1993
[399] E. PRISNER, A Journey Through Intersection Graph County, http:// www.math.uni-hamburg.de /spag/gd/mitarbeiter /prisner/ Pris/Rahmen.html
[400] H.J. PRÖMEL, A. STEGER, Excluding Induced Subgraphs III. A General Asymptotic, Random Structures and Algorithms 3, 19-31, 1992
[401] I. RABINOVITCH, The Dimension of Semiorders, Journal of Combinatorial Theory Series A 25, 50-61, 1978
[402] V. RAGHAVAN, J. SPINRAD, Robust Algorithms for Restricted Domains, Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms, 460-467, 2001
[403] J.L. RAMÍREZ ALFONSÍN, B.A. REED, Perfect Graphs, Wiley, Chichester, 2001
[404] A.S. RAO, C. PANDU RANGAN, Optimal Parallel Algorithms on Circular-Arc Graphs, Information Processing Letters 33, 147-156, 1989
[405] B. REED, N. SBIHI, Recognizing Bull-Free Perfect Graphs, Graphs and Combinatorics 11, 171-178, 1995
[406] C. RHEE, Y.D. LIANG, Finding a Maximum Matching in a Permutation Graph, Acta Informatica 32, 779-792, 1995
[407] C. RHEE, Y.D. LIANG, S.K. DHALL, S. LAKSHIVARAHAN, An $O(m+n)$ Algorithm for Finding Minimum Weight Dominating Set in Permutation Graphs, SIAM Journal on Computing 25, 404-419, 1996
[408] C.S. RIM, K. NAKAJIMA, On Rectangle Intersection and Overlap Graphs, IEEE Trans. Circuits Systems/ Fund. Theory Appl. 42, 549-553, 1995
[409] R.L. RIVEST, J. VUILLEMIN, On Recognizing Graph Properties from Adjacency Matrices, Theoretical Computer Science 3, 371-384, 1976/77
[410] F.S. ROBERTS, Indifference Graphs, Proof Techniques in Graph Theory, Ed. F. Harary, Academic Press, New York, 1969, 139-146
[411] F.S. ROBERTS, Graph Theory and Its Application to Problems of Society, SIAM, 1978
[412] N. ROBERTSON, P.D. SEYMOUR, Graph Minors - A Survey, in Surveys in Combinatorics, I. Anderson, ed., Cambridge University Press, Cambridge, 153-171, 1985
[413] D.J. ROSE, Triangulated Graphs and the Elimination Process, Journal of Mathematical Analysis and Applications 32, 597-609, 1970
[414] D.J. ROSE, R.E. TARJAN, G.S. LUEKER, Algorithmic Aspects of Vertex Elimination on Graphs, SIAM Journal on Computing 5, 266-283, 1976
[415] D. ROTEM, J. URRUTIA, Circular Permutation Graphs, Networks 12, 428-437, 1982
[416] N.D. ROUSSOPOLOUS, A O(max(m,n)) Algorithm for Determining the Graph H from its Line Graph, Information Processing Letters 2, 108-112, 1973
[417] I. RUSU, J. SPINRAD, Forbidden Subgraph Decomposition, Discrete Mathematics 247, 159-168, 2002
[418] R.S. SANKARANARAYANA, Tracking the P-NP Boundary for Well-Covered Graphs, Networks 29, 29-37, 1997
[419] R.S. SANKARANARAYANA, L.K. STEWART, Complexity Results for Well Covered Graphs, Networks 22, 247-262, 1992
[420] N. SBIHI, Algorithme de Recherche d'un Stable de Cardinalité Maximum dans un Graphe sans Étoile, Discrete Mathematics 29, 53-76, 1980
[421] M. SCHAEFER, E. SEDGWICK, D. STEFANKOVIC, Recognizing String Graphs is in NP, 34th Symposium on the Theory of Computing, 1-6, 2002
[422] A.A. SCHÄFFER, A Faster Algorithm to Recognize Undirected Path Graphs, Discrete Applied Mathematics 43, 261-295, 1993
[423] E. SCHEINERMAN, Local Representation Using Very Short Labels, Discrete Mathematics 203, 287-290, 1999
[424] E.R. SCHEINERMAN, The Many Faces of Circle Orders, Order 9, 343-348, 1992
[425] E.R. SCHEINERMAN, Characterizing Intersection Classes of Graphs, Discrete Mathematics 55, 185-193, 1985

[426] E.R. SCHEINERMAN, D.B. WEST, The Interval Number of a Planar Graph: Three Intervals Suffice, Journal of Combinatorial Theory Series B 35, 224-239, 1983
[427] E. SCHEINERMAN, J. ZITO, On the Size of Hereditary Classes of Graphs, Journal of Combinatorial Theory Series B 61, 16-39, 1994
[428] W. SCHNYDER, Planar Graphs and Poset Dimension, Order 5, 323-343, 1989
[429] A. SEBÖ, Integer Polyhedra: Combinatorial Properties and Complexity, working paper, Leibniz-IMAG, France, 2001
[430] F.B. SHEPHERD, The Theta Body and Imperfection, Perfect Graphs, Wiley, Chichester, 261-291, 2001
[431] T. SHERMER, Recent Results in Art Galleries, Proceedings of the IEEE 80, 1992
[432] W.K. SHIH, T.C. CHERN, W.L. HSU, An $O(n^2 \log n)$ Algorithm for the Hamiltonian Cycle Problem on Circular-Arc Graphs, SIAM Journal on Computing 21, 1026-1046, 1992
[433] W,K. SHIH, W.L. HSU, An $O(n \log n + m \log\log n)$ Maximum Weight Clique Algorithm for Circular-Arc Graphs, Information Processing Letters 31, 129-134, 1989
[434] D.J. SKRIEN, A Relationship between Triangulated Graphs, Comparability Graphs, Proper Interval Graphs, Proper Circular-Arc Graphs, and Nested Interval Graphs, Journal of Graph Theory 6, 309-316, 1982
[435] J. SPINRAD, Nonredundant Ones and Chordal Bipartite Graphs SIAM Journal on Discrete Mathematics 8, 251-257, 1995
[436] J.P. SPINRAD, On Comparability and Permutation Graphs, SIAM Journal on Computing 14, 658-670, 1987
[437] J.P. Spinrad, Two Dimensional Partial Orders, Ph.D. Thesis, Dept. of EECS, Princeton Univ., 1982
[438] J.P. Spinrad, Doubly Lexical Ordering of Dense 0-1 Matrices, Information Processing Letters 45, 229-235, 1993
[439] J.P. SPINRAD, Circular-Arc Graphs with Clique Cover Number Two, Journal of Combinatorial Theory Series B 44, 300-306, 1988
[440] J.P. SPINRAD, Prime Testing for the Split Decomposition of an Undirected Graph, SIAM Journal on Discrete Mathematics 2, 590-599, 1989
[441] J.P. SPINRAD, Finding Large Holes, Information Processing Letters 39, 227-229, 1991
[442] J. SPINRAD, Recognition of Circle Graphs, Journal of Algorithms 16, 264-282, 1994
[443] J. SPINRAD, Dimension and Algorithms, Lecture Notes in Computer Science 831, Ordal 94, 33-52, 1994
[444] J. SPINRAD, Recognizing Quasi-Triangulated Graphs, Discrete Applied Mathematics, to appear
[445] J.P. SPINRAD, A. BRANDSTÄDT, L. STEWART, Bipartite Permutation Graphs, Discrete Applied Mathematics 18, 279-292, 1987
[446] J. SPINRAD, R. SRITHARAN, Algorithms for Weakly Triangulated Graphs, Discrete Applied Mathematics 19, 181-191, 1995
[447] J. SPINRAD, J. VALDES, Recognition and Isomorphism of Two-Dimensional Partial Orders, Lecture Notes in Computer Science 154, ICALP 1983, 676-686, 1983
[448] A.P. SPRAGUE, Recognition of Bipartite Permutation Graphs, Congressus Numerantium 112, 151-161, 1995
[449] A.P. SPRAGUE, An $O(n \log n)$ Algorithm for Bandwidth of Interval Graphs, SIAM Journal on Discrete Mathematics 7, 213-220, 1994
[450] R. SRITHARAN, A Linear Time Algorithm to Recognize Circular Permutation Graphs, Networks 27, 171-174, 1996
[451] R. SUNDARAM, K.S. SINGH, C. PANDU RANGAN, Treewidth of Circular-Arc Graphs, SIAM Journal on Discrete Methods 7, 647-655, 1994
[452] M. TALAMO, P. VOCCA, Representing Graphs Implicitly Using Almost Optimal Space, Discrete Applied Mathematics 108, 193-210, 2001
[453] R.E. TARJAN, Efficiency of a Good but not Linear Set Union Algorithm, Journal of the ACM 22, 215-225, 1975
[454] R.E. TARJAN Decomposition by Clique Separators, Discrete Mathematics 55, 221-232, 1985
[455] R.E. TARJAN, Complexity of Combinatorial Algorithms, SIAM Review 20, 457-489, 1978
[456] R.E. TARJAN, M. YANNAKAKIS, Simple Linear Time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs, SIAM Journal on Computing 13, 566-579, 1984

[457] A. TARSKI, A Decision Method for Elementary Algebra, RAND Corporation, 1948
[458] C. THOMASSEN, Interval Representations of Planar Graphs, Journal of Combinatorial Theory Series B, 9-20, 1986
[459] M. THORUP, All Structured Programs have Small Tree Width and Good Register Allocation, Information and Computation 142, 159-181, 1998
[460] W.T. TROTTER, Problems and Conjectures in the Combinatorial Theory of Ordered Sets, Annals of Discrete Mathematics 41, 401-416, 1989
[461] W.T. TROTTER, Combinatorics and Partially Ordered Sets: Dimension Theory, Johns Hopkins University Press, Baltimore MD, 1992
[462] W.T. TROTTER, F. HARARY, On Double and Multiple Interval Graphs, Journal of Graph Theory 3, 205-211, 1979
[463] W. T. TROTTER JR., J. I. MOORE, Characterization Problems for Graphs, Partially Ordered Sets, Lattices, and Families of Sets, Discrete Mathematics 16, 361-381, 1976
[464] S. TSUKIYAMA, M. IDE, H. ARIYOSHI, I. SHIRAKAWA, A New Algorithm for Generating all the Maximal Independent Sets, SIAM Journal on Computing 6, 505-517, 1977
[465] A.C. TUCKER, An Efficient Test For Circular-Arc Graphs, SIAM Journal on Computing 9, 1-24, 1980
[466] A.C. TUCKER, Matrix Characterizations of Circular-Arc Graphs, Pacific Journal of Mathematics 19, 535-545, 1971
[467] A.C. TUCKER, A Structure Theorem for the Consecutive 1's Property, Journal of Combinatorial Theory Series B 12, 153-162, 1972
[468] A.C. TUCKER Structure Theorems for Some Circular-Arc Graphs, Discrete Mathematics 7, 167-195, 1974
[469] A.C. TUCKER Coloring a Family of Circular-Arc Graphs, SIAM Journal on Applied Mathematics 29, 493-502, 1975
[470] A.C. TUCKER, Coloring Perfect $(K_4 - e)$-free Graphs, Journal of Combinatorial Theory Series B 43, 151-172, 1987
[471] W. UNGER, On the k-Colouring of Circle Graphs, Lecture Notes in Computer Science 294, STACS 88, 61-72, 1988
[472] W. UNGER, The Complexity of Colouring Circle Graphs, extended abstract available at citeseer.nj.nec.com/195897.html.
[473] J. URRUTIA, Partial Orders and Euclidean Geometry, Algorithms and Order, I. Rival ed., Kluwer, Dordrecht, 387-434, 1989
[474] J. URRUTIA, F. GAVRIL, An Algorithm for Fraternal Orientation of Graphs, Information Processing Letters 41, 271-274, 1992
[475] J. VALDES, R.E. TARJAN, E.L. LAWLER The Recognition of Series-Parallel Digraphs, SIAM Journal on Computing 11, 298-314, 1978
[476] J.R. WALTER, Representations of Chordal Graphs as Subtrees of a Tree, Journal of Graph Theory 2, 265-267, 1978
[477] D.W. WANG, Y.S. KUO, A Study on Two Geometric Location Problems, Information Processing Letters 28, 281-286, 1988
[478] E. WANKE, k-NLC Graphs and Polynomial Algorithms, Discrete Applied Mathematics 54, 251-266, 1994
[479] H. WARREN, Lower Bounds for Approximation by Nonlinear Manifolds, Transactions of the AMS 133, 167-178, 1968
[480] G. WEGNER, Eigenschaften der Nerven Homologisch-Einfacher Familien in R^n, Dissertation, University of Göttingen, 1967
[481] D.B. WEST, D.B. SHMOYS, Recognizing Graphs with Fixed Interval Number is NP-complete, Discrete Applied Mathematics 8, 295-305, 1984
[482] D.B. WEST, Introduction to Graph Theory, Prentice Hall, 1996
[483] S.H. WHITESIDES, An Algorithm for Finding Clique Cut-Sets, Information Processing Letters 12 , 31-32, 1981
[484] S.H. WHITESIDES, A Method for Solving Certain Graph Recognition and Optimization Problems, with Applications to Perfect Graphs, Annals of Discrete Mathematics 21, 281-297, 1984
[485] A. WIGDERSON, Improving the Performance Guarantee for Approximate Graph Coloring, Journal of the ACM 30, 729-735, 1983

[486] T.V. WIMER, Linear Algorithms on k-terminal Graphs, PhD Thesis, Dept. of Computer Science, URI-030, Clemson University, 1987
[487] P.M. WINKLER, Proof of the Squashed Cube Conjecture, Combinatorica 3, 135-139, 1983
[488] S.K, WISMATH, Characterizing Bar Line-of-Sight Graphs, Proceedings 1st ACM Symposium on Computational Geometry, 147-152, 1985
[489] T.-H. WU, An $O(n^3)$ Isomorphism Test for Circular-Arc Graphs, Ph.D. Thesis, Applied Mathematics and Statistics, SUNY-Stonybrook, 1983
[490] J.-H. YAN, The Bandwidth Problem in Cographs, Tamsui Oxford Journal of Mathematical Sciences 13, 31-36, 1997
[491] M. YANNAKAKIS, Computing the Minimum Fill-in is NP-complete, SIAM Journal on Algebraic Discrete Methods 2, 77-79, 1981
[492] M. YANNAKAKIS, The Complexity of the Partial Order Dimension Problem, SIAM Journal on Algebraic Discrete Methods 3, 351-358, 1982
[493] H.G. YEH, G.J. CHANG, Weighted Connected Domination and Steiner Trees in Distance-Hereditary Graphs, Discrete Applied Mathematics 87, 245-253, 1998
[494] R. BAR-YEHUDA, M.M. HALLDORSSON, J. NAOR, H. SHACHNAI, I. SHAPIRA, Scheduling Split Intervals, 13th ACM-SIAM Symposium on Discrete Algorithms, 2002
[495] C.W. YU, G.H. CHEN, Efficient Parallel Algorithms for Doubly-Convex Bipartite Graphs, Theoretical Computer Science 147, 249-265, 1995

Index

M-join, 174, 175
S-ordering, 129
Γ-free, 96, 183
Γ-free matrix, 111, 112, 120, 124, 132, 145
Ω, 6
Θ, 6
k-clique extendible ordering, 243
k-closure, 66, 70
k-module, 166
k-polygon graph, 130, 223, 311
k-simplicial vertex, 185
k-tree, 50
2-join, 171, 174, 175

Abbas, 129
adjacency list, 9
adjacency matrix, 8, 9
adjacent, 6
Agarwal, 74
Alekseev, 106
Alon, 54, 55, 74
alternance graph, 212
angle order, 58, 224
Anstee, 143
arboricity, 146
Aronov, 74
art gallery problem, 83
asteroidal triple, 85, 129, 134
asteroidal triple-free graph, 85, 86, 88, 90, 93–95
AT-free comparability graph, 95
AT-free graph, 85, 88, 91, 93, 94, 96, 102, 249, 304
autograph, 28

B1-orientable, 47
balanced k-module, 166, 179
balanced skew partition, 174
Bandelt, 20, 187
bandwidth, 305–316
bar-representable graph, 80
Benzer, 11, 12, 265
Berge, 136
Berge graph, 50, 251
biconvex graph, 96, 128, 134
bipartite complement, 131, 208
bipartite graph, 88, 106, 126, 170, 308
bipartite permutation graph, 88, 96, 126, 133
bisimplicial edge, 130, 133
bitonic path, 115
blocking vertex, 255
Boland, 86
Bondy, 66
Booth, 145, 193
Borie, 51, 163, 164, 253
bottleneck Monge matrix, 126
Bouchet, 212, 215
Bouchitté, 34
bounded substitution decomposition diameter, 160, 178
bounded tolerance graph, 58, 248, 254, 305
boxicity, 36, 47, 53, 222, 223, 233, 262, 305
Brandstädt, 20, 185
brittle graph, 189, 253
Buckingham, 306
Buneman, 267

Cai, 248
Cameron, 251
Cenek, 42
certificate, 87
chain cover, 210
chain cover problem, 208
chain dimension, 208
chain graph, 125, 134, 208
characterization problem, 14
Chazelle, 75, 79
Chen, 129
Chepoi, 185
chord, 31
chordal bipartite graph, 88–90, 94, 95, 111, 112, 114, 118–126, 130, 132, 135, 136, 159, 183, 203, 211, 255, 306, 313
chordal comparability graph, 33, 60, 62, 64, 69, 70, 90
chordal graph, 29, 31, 33, 38, 47, 49, 50, 60, 61, 63–65, 70, 71, 85–88, 90, 95, 111, 120–122, 124, 133, 187, 189, 191, 196, 224, 225, 246, 257, 262, 267, 272, 305, 312, 313

chromatic number, 8, 36, 40, 49, 80
Chvátal, 46, 66, 83, 245
circle graph, 20, 39, 42, 47, 49–51, 130, 134, 159, 187, 212, 228, 306, 311, 313
circle intersection graph, 36
circle order, 42, 54, 58, 224, 262
circle trapezoid graph, 306
circular permutation graph, 39, 42, 217, 218, 228
circular-arc filament graph, 44
circular-arc graph, 35, 38, 47, 48, 123, 124, 149, 159, 194, 197, 202, 211, 225, 237, 241, 252, 260, 269, 272, 273, 306, 307, 311, 313
circular-arc overlap graph, 248
Clark, 239, 240
claw-free graph, 103, 104, 109, 110, 191, 237, 312
clique, 8, 32–34, 36–38, 40–42, 44, 46–49, 80, 169, 233, 235, 239, 241, 242, 247, 248, 253, 304–317
clique cover, 8, 46, 305–307, 309–312, 314, 316, 317
clique cutset, 167, 234
clique separator, 167, 169, 170, 179, 233
clique separator decomposition, 221
clique tree, 31, 64, 258
clique-width, 164–166, 178, 247, 307, 309
clone, 92, 109
co-AT-free graph, 91, 93
co-bipartite edge elimination order, 240
co-bipartite graph, 106, 170
co-chordal graph, 87, 90, 94, 95, 196, 224, 313
co-comparability graph, 33, 38, 39, 47, 49, 51, 60, 86, 88, 90, 95, 305, 308, 313, 314
co-interval graph, 196
co-planar graph, 41
Cogis, 124, 209
cograph, 28, 97, 109, 160, 165, 178, 188, 191, 307
Colbourn, 239, 240
coloring, 8, 32, 35, 36, 38, 46, 87, 169, 245, 248, 304–317
comparability graph, 28, 29, 31, 33, 38, 50, 60, 64, 70, 88, 90, 91, 94, 96, 102, 107, 110, 123, 124, 157, 159, 176, 177, 179, 191, 196, 209, 212, 217, 235, 243, 246, 251, 253, 308, 313
complement, 6
completely decomposable, 161
composition sequence, 44, 50
connected matrix, 138, 249
consecutive 1s property, 145
construction problem, 13
containment model, 51
convex, 133
convex fan, 76, 244
convex graph, 96, 126, 128, 133, 145, 254, 306
convex ordering, 128
Corneil, 39, 51, 82, 98, 193
cotree, 98, 109
Coullard, 82
counting problem, 13
covering graph, 223
Cunningham, 213, 214
curve contact graph, 254
cycle-free partial order, 60

Dahlhaus, 10, 117
decision problem, 7
deterministically decomposable, 161
diameter, 189, 225
diametral pair, 225
Dilworth, 107
Dilworth number, 107
dimension, 41, 49, 50, 61, 94, 107, 118, 133, 176, 177, 222, 229, 249, 262
Dirac, 191
directed path graph, 311, 314
disk graph, 249
disk intersection graph, 36, 48, 53, 55, 57, 223, 262, 308
distance hereditary graph, 20, 161, 165, 181, 182, 187, 191, 308, 309
distance labeling, 25, 29
dominating pair, 249, 304
dominating set, 36, 48, 70, 80, 185, 248, 249, 304–317
domination graph, 50, 58, 133, 309
dot product graph, 56, 58, 309
doubly chordal graph, 185, 187, 188, 191, 234, 309
doubly convex graph, 128, 145
doubly lexical ordering, 112, 120, 121, 132, 133, 146
Dragan, 185
dually chordal graph, 124, 185, 186, 188, 191, 234, 310, 311, 313
Dushnick, 41, 87
dynamic programming, 34

edge, 6
Edmonds, 251
Edwards, 269
elimination, 181
enclosure property, 133
envelope, 77
EPT graph, 38, 59, 69, 102, 167, 220, 222, 233, 234, 252, 262, 269, 310
Erdös, 194
Eschen, 197
Euler, 85
Everett, 82
external visibility graph, 74, 83

f-diagram, 51
Földes, 87
Füredi, 143
feedback vertex, 305–315
Fellows, 40
Felsner, 58
Ferrers digraph, 209
Ferrers dimension, 124, 209
fill, 167, 179
Ford, 89
fraternally orientable, 47
Fulkerson, 89, 205

Gabor, 212
Gallai, 88
Gamarnik, 269
Garey, 65, 167, 222
Gavril, 40, 42, 108, 220
generalized implicit representation, 22, 23, 79, 117, 132
generalized join, 178
Ghosh, 81
Gilmore, 86
Golumbic, 5, 20, 51, 58, 60, 65, 85, 87, 90, 130, 191, 212, 233, 265
Grötschel, 14, 46
graph, 6
graph drawing, 5
greedy coloring, 47, 51, 246, 252
grid, 38, 48
grid graph, 38, 305, 315
gridicity, 129
Gross, 205
Gustedt, 10, 159

Habib, 28
Halin graph, 163, 164
Hamilton cycle, 8, 40, 87, 241, 304–317
Hammer, 87
Hayward, 58
Helly circular-arc graph, 207
Helly property, 33, 197, 233
hereditary, 19, 48, 76, 106
Hiraguchi poset, 49
Hoàng, 180
Hoaàng, 179
Hoffman, 86, 121
hole, 46, 50
homogeneous set, 150
Hsu, 212, 269, 275

implicit graph conjecture, 19
implicit graph question, 85
implicit representation, 17, 19–21, 25, 28, 33, 37, 48, 53, 73, 98, 110, 134, 160, 161, 166, 253
independent set, 8, 33, 35–37, 40, 43, 44, 46, 49, 50, 80, 110, 163, 169, 233, 237, 238, 249, 250, 252, 304–317
induced subgraph, 6
induced visibility graph, 76, 79, 83, 253
intersection characterization, 50
intersection class, 96
intersection graph, 5
intersection model, 44
interval dimension, 94, 124, 209, 212, 222, 227
interval filament graph, 40, 44, 49, 224, 248
interval graph, 11, 16, 17, 27, 33–35, 42, 47–50, 56, 60, 85, 86, 88, 90, 129, 145, 193, 196, 197, 204, 259, 265, 269, 273, 275, 307, 310, 313
interval number, 37, 48, 223, 249, 262, 311
interval order, 196, 209, 227
inversion-free, 244
inversion-free graph, 255, 311
inversion-free ordering, 254
isomorphism, 80, 305–312, 314–317
isomorphism complete, 59

Jamison, 233
Jegou, 28
Johnson, 65, 167, 222, 239, 240
join decomposition, 160, 182, 188, 212, 213, 228

k-join, 228
k-polygon graph, 40
k-sparse graph, 19
k-tree, 27, 272
Kamula, 39
Kannan, 17, 19, 21, 22
Karapetjan, 269
Kierstead, 63
Klee, 83
Kloks, 103
Kolen, 121
Kratochvíl, 40, 41, 222, 247
Kratsch, 103
Kubena, 41

labeled graph, 6
LB-simplicial, 218
LBFS, 70, 168, 188, 192, 225, 263
Le, 20
Lehel, 119
Lekkerkerker, 86
lexicographic breadth first search, 94, 188, 189, 192, 194, 249
Lin, 74, 80
line graph, 27, 170, 221, 234, 305, 310, 311
line of sight graph, 80
linear extension, 32, 41, 157
linear matrix, 136, 146
local complement, 215, 228
local structure, 18
Lovász, 14, 46
Lubiw, 82, 120, 121

Lueker, 145, 168, 193

Müller, 103
Ma, 169, 212
Maffray, 179
matching, 70, 189, 237, 253, 306, 307, 312
Matousek, 40
matrix multiplication, 86, 101, 103, 109
max cut, 305, 307–311, 313–317
maximal, 8
maximal cardinality search, 188
McConnell, 10, 115, 149, 155, 158, 180
MCS, 188
Miller, 41, 87
minimal fill, 167, 169
minimal separator, 34, 48, 49, 111, 189, 218
minimal separators, 36
minimum fill, 91, 167, 306, 308–310, 312, 314, 316, 317
Minty, 103
modular decomposition, 92, 160, 194
modulator, 248
module, 92, 109, 150, 152, 175, 176
Monge matrix, 142, 146
Monma, 220
monotone property, 9
monotonic path, 115
Moore, 211
Mulder, 20, 187
Muller, 17, 18, 21, 22
multitolerance graph, 314

N-free poset, 27
Naji, 212
Nakajima, 42
Naor, 17, 19, 21, 22
Nastos, 95
neighbor, 6
neighborhood containment, 123, 133
Nesetril, 247
NLC-width, 164, 178
nondeterministically decomposable, 161
nonredundant edge, 254
NP-complete, 7

O, 6
Olariu, 193
optimization problem, 14
order notation, 6
ordered chordal graph, 82, 83
overlap, 42, 47, 49, 51
overlap graph, 212
overlap model, 51

Paige, 121
Papoutsakis, 263
parallel module, 153
Parker, 163, 164
partial k-tree, 27, 163
partial order, 96
partially complemented, 10
partitionable graph, 49
partitioning, 149–151, 157, 159, 177
path graph, 21, 28, 38, 59, 68, 220, 269, 311, 313
path ordering, 95
pathwidth, 308, 313
pendant vertex, 20, 181
perfect elimination, 64
perfect elimination bipartite graph, 65, 130, 133
perfect elimination order, 64
perfect elimination scheme, 31, 111, 121, 124, 191
perfect graph, 32, 46, 49, 50, 69, 87, 107, 170, 173, 228, 246, 312
perfectly orderable graph, 120, 245, 253, 255, 312, 316
Perl, 98
permutation graph, 38, 47–49, 51, 58, 60, 87, 88, 90, 95, 96, 98, 109, 126, 149, 158, 160, 178, 191, 198, 217, 306, 308, 311–313
PI graph, 39, 224, 313
PI* graph, 39, 224
planar graph, 20, 38, 69, 177, 309, 311, 312
planar poset, 223
polygon-circle graph, 41, 51
PQ-tree, 129, 145, 193, 197, 199, 206
PQ-trees, 145
prime module, 153
probe interval graph, 58, 266, 313
projective plane, 136
proper circular-arc graph, 205, 227
proper interval graph, 204
proximity graph, 36
pruning sequence, 181
pseudoline, 81

Qin, 63
quotient graph, 92, 153

Raghavan, 66, 274
Ramsey theory, 24
realizer, 158
recognition problem, 13
reconstructible vertex, 92
rectangle number, 118, 133
rectangle overlap graph, 248
recursively decomposable graph, 163, 178
reduced matrix, 113
redundant 1, 113
representation problem, 13
Rim, 42
Roberts, 265
robust algorithm, 231, 250, 316, 317
robust NP-complete, 254
robust-NP-complete, 250

rooted directed path graph, 59, 312, 314
Rose, 168
Rudich, 17, 19, 21, 22

Sakarovich, 121
sandwich problem, 308, 313
Sbihi, 103
Schäffer, 307
Scheinerman, 23, 44, 51, 55, 105
Schnyder, 222
Schrijver, 14, 46
semiorder, 178
series module, 153
series-parallel graph, 15, 38, 164, 272
Shamir, 58
Shepherd, 49
Shih, 269, 275
shortest path, 51
sign pattern, 54
simple elimination order, 70, 182, 185, 189
simple matrix, 143
simple vertex, 169, 182
simplicial vertex, 31, 102, 107, 109, 179, 191
skew partition, 170, 171, 173, 179
Skiena, 74, 80
source, 156, 176
sphericity, 37
spider graph, 41, 49
Spinrad, 20, 197, 212
split, 212
split graph, 21, 29, 60, 87, 90, 104, 106, 189, 262, 305, 312, 313
Sritharan, 218
stable set, 8
star cutset, 171, 174
Stathopolous, 16
Steiner tree, 305–317
Steinhaus, 66
Steinhaus graph, 65, 69
Sterten, 179
Stewart, 42, 98, 129, 193, 306
Stobert, 15, 69
string graph, 40, 223
strong ordering, 127
strong perfect graph conjecture, 170
strongly chordal graph, 112, 118, 119, 124, 169, 182, 185, 189, 191, 234, 313
subgraph isomorphism, 308
substitution decomposition, 98, 149, 152, 156, 176–178, 212
sun, 182
Supowit, 212
Suri, 74

Tarjan, 121, 149, 167–169, 220, 233
Tarski, 57
Thorup, 272
threshold graph, 56, 58, 107, 309
threshold signed graph, 107
Todinca, 34
tolerance graph, 57, 58, 134, 247, 314
topological sort, 32
total graph, 28
total interval number, 37, 48, 223
totally balanced hypergraph, 119
Tovey, 163, 164
trampoline, 182
transitive closure, 60
transitive graph, 158
transitive orientation, 31, 51, 150, 156–158, 176, 197
transitive reduction, 60, 176
trapezoid graph, 38, 50, 58, 94, 124, 130, 134, 209, 211, 227, 248, 254, 305, 309, 312–314
traveling salesman, 315
tree, 51, 178
treewidth, 91, 272, 305, 306, 308–310, 312–317
Trenk, 58
triangle, 158
triangle-extendible graph, 244, 253
triangle-extendible ordering, 242
triangle-free graph, 49, 100, 101, 103, 109, 110, 191
triangulation, 272
Trotter, 41, 63, 211
Tucker, 197–199, 203, 205, 225
Turàn, 109
twin, 20, 24, 181
two dimensional partial order, 41, 48, 60, 87, 158, 176, 191, 202, 210, 218
two-pair, 67, 69, 70, 179

uniformly sparse, 19
unique neighbor set, 92
unit circular-arc graph, 206, 226, 260
unit disk graph, 36, 239, 309, 315
unit interval graph, 178, 204, 259, 263
unit-cost assumption, 8
universal graph, 21, 22

vertex, 6
vertex cover, 253, 315
vertex expansion, 44, 50
vertex multiplication, 45
visibility graph, 55, 73, 74, 76, 80, 81, 83, 126, 241, 243–245, 253, 255, 262, 311, 316

Warren, 54
Warren representable, 55
Warren's theorem, 54–56, 73, 74, 77
weak order, 177
weakly chordal graph, 32, 67, 69, 71, 83, 87–91, 93–96, 110, 123, 124, 179, 218, 219, 306, 309, 314, 316
weakly triangulated graph, 32, 191

Wei, 220
well-covered graph, 232, 250, 252, 317
Whitesides, 167, 169, 179
width, 107

Yannakakis, 65, 167, 222
Yu, 129

Zito, 23, 105

Titles in This Series

19 **Jeremy P. Spinrad,** Efficient graph representations, 2003

18 **Olavi Nevanlinna,** Meromorphic functions and linear algebra, 2003

17 **Vitaly I. Voloshin,** Coloring mixed hypergraphs: theory, algorithms and applications, 2002

16 **Neal Madras,** Lectures on Monte Carlo Methods, 2002

15 **Bradd Hart and Matthew Valeriote, Editors,** Lectures on algebraic model theory, 2002

14 **Frank den Hollander,** Large deviations, 2000

13 **B. V. Rajarama Bhat, George A. Elliott, and Peter A. Fillmore, Editors,** Lectures in operator theory, 2000

12 **Salma Kuhlmann,** Ordered exponential fields, 2000

11 **Tibor Krisztin, Hans-Otto Walther, and Jianhong Wu,** Shape, smoothness and invariant stratification of an attracting set for delayed monotone positive feedback, 1999

10 **Jiří Patera, Editor,** Quasicrystals and discrete geometry, 1998

9 **Paul Selick,** Introduction to homotopy theory, 1997

8 **Terry A. Loring,** Lifting solutions to perturbing problems in C^*-algebras, 1997

7 **S. O. Kochman,** Bordism, stable homotopy and Adams spectral sequences, 1996

6 **Kenneth R. Davidson,** C*-Algebras by example, 1996

5 **A. Weiss,** Multiplicative Galois module structure, 1996

4 **Gérard Besson, Joachim Lohkamp, Pierre Pansu, and Peter Petersen Miroslav Lovric, Maung Min-Oo, and McKenzie Y.-K. Wang, Editors,** Riemannian geometry, 1996

3 **Albrecht Böttcher, Aad Dijksma and Heinz Langer, Michael A. Dritschel and James Rovnyak, and M. A. Kaashoek Peter Lancaster, Editor,** Lectures on operator theory and its applications, 1996

2 **Victor P. Snaith,** Galois module structure, 1994

1 **Stephen Wiggins,** Global dynamics, phase space transport, orbits homoclinic to resonances, and applications, 1993