FIELDS INSTITUTE MONOGRAPHS

THE FIELDS INSTITUTE FOR RESEARCH IN MATHEMATICAL SCIENCES

Efficient Graph Representations

Jeremy P. Spinrad

American Mathematical Society

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
511′.–dc21

2003045106
CIP

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

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 $2^{(1+\epsilon)n \log n}$ 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