

# Computer Solution of Sparse Linear Systems

Alan George  
Department of Computer Science  
University of Waterloo

Joseph Liu  
Department of Computer Science  
York University

Esmond Ng  
Mathematical Sciences Section  
Oak Ridge National Laboratory

January 5, 1994

# Contents

Preface vii

---

<b>1 Introduction</b>	<b>1</b>
1.1 About the Book . . . . .	1
1.2 Cholesky's Method and the Ordering Problem . . . . .	2
1.3 Positive Definite vs. Indefinite Matrix Problems . . . . .	5
1.4 Iterative Versus Direct Methods . . . . .	10

---

<b>2 Fundamentals</b>	<b>13</b>
2.1 Introduction . . . . .	13
2.1.1 Notations . . . . .	13
2.2 The Factorization Algorithm . . . . .	16
2.2.1 Existence and Uniqueness of the Factorization . . . . .	16
2.2.2 Computing the Factorization . . . . .	18
2.2.3 Sparse Matrix Factorization . . . . .	21
2.3 Solving Triangular Systems . . . . .	26
2.3.1 Computing the Solution . . . . .	26
2.3.2 Operation Counts . . . . .	28
2.4 Some Practical Considerations . . . . .	32
2.4.1 Storage Requirements . . . . .	33
2.4.2 Execution Time . . . . .	37

---

<b>3 Graph Theory Notation</b>	<b>41</b>
3.1 Introduction . . . . .	41
3.2 Basic Terminology and Some Definitions . . . . .	42

3.3	Computer Representation of Graphs . . . . .	46
3.4	General Information on the Graph Subroutines . . . . .	49
<hr/>		
<b>4</b>	<b>Band and Envelope Methods</b>	<b>55</b>
4.1	Introduction . . . . .	55
4.2	The Band Method . . . . .	55
4.3	The Envelope Method . . . . .	59
4.3.1	Matrix Formulation . . . . .	59
4.3.2	Graph Interpretation . . . . .	63
4.4	Envelope Orderings . . . . .	65
4.4.1	The Reverse Cuthill-McKee Algorithm . . . . .	65
4.4.2	Finding a Starting Node . . . . .	70
4.4.3	Subroutines for Finding a Starting Node . . . . .	74
4.4.4	Subroutines for the Reverse Cuthill-McKee Algorithm . . . . .	78
4.5	Implementation of the Envelope Method . . . . .	90
4.5.1	An Envelope Storage Scheme . . . . .	90
4.5.2	The Storage Allocation Subroutine <b>FNENV</b> (FiNd EN- VeloPe) . . . . .	92
4.6	The Numerical Subroutines <b>ESFCT</b> , <b>ELSLV</b> and <b>EUSLV</b> . . . . .	94
4.6.1	The Triangular Solution Subroutines <b>ELSLV</b> and <b>EUSLV</b> . . . . .	94
4.6.2	The Factorization Subroutine <b>ESFCT</b> . . . . .	100
4.7	Additional Notes . . . . .	104
<hr/>		
<b>5</b>	<b>General Sparse Methods</b>	<b>107</b>
5.1	Introduction . . . . .	107
5.2	Symmetric Factorization . . . . .	107
5.2.1	Elimination Graph Model . . . . .	108
5.2.2	Modelling Elimination By Reachable Sets . . . . .	110
5.3	Computer Representation of Elimination Graphs . . . . .	117
5.3.1	Explicit and Implicit Representations . . . . .	118
5.3.2	Quotient Graph Model . . . . .	119
5.3.3	Implementation of the Quotient Graph Model . . . . .	126
5.4	The Minimum Degree Ordering Algorithm . . . . .	132
5.4.1	The Basic Algorithm . . . . .	132
5.4.2	Description of the Minimum Degree Algorithm Using Reachable Sets . . . . .	133
5.4.3	An Enhancement . . . . .	136

5.4.4	Implementation of the Minimum Degree Algorithm . . .	142
5.5	Sparse Storage Schemes . . . . .	160
5.5.1	The Uncompressed Scheme . . . . .	160
5.5.2	Compressed Scheme . . . . .	161
5.5.3	On Symbolic Factorization . . . . .	165
5.5.4	Storage Allocation for the Compressed Scheme and the Subroutine <b>SMBFCT</b> . . . . .	171
5.6	Numerical Subroutines for Factorization and Solution . . . .	178
5.6.1	The Subroutine <b>GSSLV</b> (General sparse Symmetric SoLve) . . . . .	183
5.7	Additional Notes . . . . .	185
<hr/>		
<b>6</b>	<b>Quotient Tree Methods</b>	<b>187</b>
6.1	Introduction . . . . .	187
6.2	Solution of Partitioned Systems . . . . .	189
6.2.1	Factorization of a Block Two by Two Matrix . . . . .	189
6.2.2	Triangular Solution of a Block Two by Two System .	193
6.3	Quotient Graphs, Trees, and Tree Partitionings . . . . .	196
6.3.1	Partitioned Matrices and Quotient Graphs . . . . .	197
6.3.2	Trees, Quotient Trees, and Tree Partitionings . . . . .	199
6.3.3	Asymmetric Block Factorization and Implicit Block Solution of Tree-partitioned Systems . . . . .	202
6.4	A Quotient Tree Partitioning Algorithm . . . . .	208
6.4.1	A Heuristic Algorithm . . . . .	208
6.4.2	Subroutines for Finding a Quotient Tree Partitioning .	211
6.5	A Storage Scheme and Allocation Procedure . . . . .	227
6.5.1	The Storage Scheme . . . . .	228
6.5.2	Internal Renumbering of the Blocks . . . . .	231
6.5.3	Storage Allocation and the Subroutines <b>FNTENV</b> , <b>FNOFNZ</b> , and <b>FNTADJ</b> . . . . .	237
6.6	The Numerical Subroutines <b>TSFCT</b> and <b>TSSLV</b> . . . . .	246
6.6.1	Computing the Block Modification Matrix . . . . .	247
6.6.2	The Subroutine <b>TSFCT</b> (Tree Symmetric FaCTorization)	250
6.6.3	The Subroutine <b>TSSLV</b> (Tree Symmetric SoLve) . . . .	256
6.7	Additional Notes . . . . .	264
<hr/>		

<b>7</b>	<b>One-Way Dissection Methods</b>	<b>267</b>
7.1	Introduction . . . . .	267
7.2	An Example – The $s \times t$ Grid Problem . . . . .	268
7.2.1	A One-Way Dissection Ordering . . . . .	268
7.2.2	Storage Requirements . . . . .	272
7.2.3	Operation Count for the Factorization . . . . .	273
7.2.4	Operation Count for the Solution . . . . .	276
7.3	A One-Way Dissection Ordering Algorithm . . . . .	279
7.3.1	The Algorithm . . . . .	279
7.3.2	Subroutines for Finding a One-Way Dissection Partitioning . . . . .	283
7.4	The Envelope Structure of Diagonal Blocks . . . . .	289
7.4.1	Statement of the Problem . . . . .	289
7.4.2	Characterization of the Block Diagonal Envelope via Reachable Sets . . . . .	290
7.4.3	An Algorithm and Subroutines for Finding Diagonal Block Envelopes . . . . .	292
7.4.4	Execution Time Analysis of the Algorithm . . . . .	300
7.5	Additional Notes . . . . .	301
<hr/>		
<b>8</b>	<b>Nested Dissection Methods</b>	<b>303</b>
8.1	Introduction . . . . .	303
8.2	Nested Dissection of a Regular Grid . . . . .	304
8.2.1	The Ordering . . . . .	304
8.2.2	Storage Requirements . . . . .	307
8.2.3	Operation Counts . . . . .	313
8.2.4	Optimality of the Ordering . . . . .	315
8.3	Nested Dissection of General Problems . . . . .	319
8.3.1	A Heuristic Algorithm . . . . .	319
8.3.2	Computer Implementation . . . . .	319
8.4	Additional Notes . . . . .	325
<hr/>		
<b>9</b>	<b>Numerical Experiments</b>	<b>327</b>
9.1	Introduction . . . . .	327
9.2	Description of the Test Problems . . . . .	328
9.3	The Numbers Reported and What They Mean . . . . .	329
9.4	Comparison of the Methods . . . . .	340

9.4.1	Criteria for Comparing Methods . . . . .	340
9.4.2	Comparison of the Methods Applied to Test Set #1 .	341
9.4.3	Comparison of the Methods Applied to Test Set #2 .	343
9.5	The Influence of Data Structures . . . . .	343
9.5.1	Storage Requirements . . . . .	345
9.5.2	Execution Time . . . . .	347
<hr/>		
<b>A</b>	<b>Hints on Using the Subroutines</b>	<b>351</b>
1.1	Sample Skeleton Drivers . . . . .	351
1.2	A Sample Numerical Value Input Subroutine . . . . .	355
1.3	Overlaying Storage in Fortran . . . . .	358
<hr/>		
<b>B</b>	<b>SPARSPAK: A Sparse Matrix Package</b>	<b>361</b>
2.1	Motivation . . . . .	361
2.2	Basic Structure of SPARSPAK . . . . .	363
2.3	User Mainline Program and an Example . . . . .	363
2.4	Description of the Interface Subroutines . . . . .	367
2.4.1	Modules for Input of the Matrix Structure . . . . .	367
2.4.2	Modules for Ordering and Storage Allocation . . . . .	369
2.4.3	Modules for Inputting Numerical Values of $\mathbf{A}$ and $\mathbf{b}$ .	369
2.4.4	Modules for Factorization and Solution . . . . .	371
2.5	Save and Restart Facilities . . . . .	372
2.6	Solving Problems with the Same Structure . . . . .	373
2.7	Output From the Package . . . . .	374
<hr/>		
	<b>References</b>	<b>376</b>
	<b>Index</b>	<b>382</b>



# List of Figures

1.2.1 . . . . .	6
1.2.2 . . . . .	6
1.2.3 . . . . .	7
1.2.4 . . . . .	7
1.2.5 . . . . .	8
1.2.6 . . . . .	8
1.3.1 . . . . .	9
2.2.1 . . . . .	19
2.2.2 . . . . .	21
2.2.3 . . . . .	22
2.3.1 . . . . .	27
2.3.2 . . . . .	28
2.3.3 . . . . .	31
2.4.1 . . . . .	34
2.4.2 . . . . .	35
2.4.3 . . . . .	36
3.2.1 . . . . .	42
3.2.2 . . . . .	43
3.2.3 . . . . .	44
3.2.4 . . . . .	45
3.2.5 . . . . .	46
3.2.6 . . . . .	47
3.3.1 . . . . .	48
3.3.2 . . . . .	49
3.3.3 . . . . .	50
3.4.1 . . . . .	51
3.4.2 . . . . .	52



3.4.3 . . . . .	53
4.2.1 . . . . .	56
4.2.2 . . . . .	57
4.3.1 . . . . .	59
4.3.2 . . . . .	61
4.3.3 . . . . .	62
4.3.4 . . . . .	62
4.3.5 . . . . .	64
4.4.1 . . . . .	66
4.4.2 . . . . .	67
4.4.3 . . . . .	67
4.4.4 . . . . .	68
4.4.5 . . . . .	69
4.4.6 . . . . .	69
4.4.7 . . . . .	71
4.4.8 . . . . .	72
4.4.9 . . . . .	73
4.4.10. . . . .	79
4.4.11. . . . .	87
4.4.12. . . . .	89
4.5.1 . . . . .	91
4.6.1 . . . . .	95
4.6.2 . . . . .	96
4.6.3 . . . . .	101
4.6.4 . . . . .	104
5.2.1 . . . . .	108
5.2.2 . . . . .	109
5.2.3 . . . . .	111
5.2.4 . . . . .	112
5.2.5 . . . . .	113
5.2.6 . . . . .	114
5.2.7 . . . . .	115
5.2.8 . . . . .	115
5.3.1 . . . . .	119
5.3.2 . . . . .	120
5.3.3 . . . . .	123
5.3.4 . . . . .	125

*LIST OF FIGURES*

5.3.5 . . . . .	127
5.3.6 . . . . .	129
5.3.7 . . . . .	130
5.4.1 . . . . .	133
5.4.2 . . . . .	134
5.4.3 . . . . .	135
5.4.4 . . . . .	136
5.4.5 . . . . .	138
5.4.6 . . . . .	140
5.4.7 . . . . .	141
5.4.8 . . . . .	144
5.4.9 . . . . .	145
5.4.10. . . . .	146
5.5.1 . . . . .	161
5.5.2 . . . . .	162
5.5.3 . . . . .	163
5.5.4 . . . . .	164
5.5.5 . . . . .	167
5.5.6 . . . . .	168
5.5.7 . . . . .	170
5.5.8 . . . . .	170
5.6.1 . . . . .	179
6.1.1 . . . . .	188
6.2.1 . . . . .	191
6.2.2 . . . . .	192
6.2.3 . . . . .	193
6.3.1 . . . . .	197
6.3.2 . . . . .	198
6.3.3 . . . . .	199
6.3.4 . . . . .	200
6.3.5 . . . . .	201
6.3.6 . . . . .	203
6.3.7 . . . . .	204
6.3.8 . . . . .	206
6.4.1 . . . . .	209
6.4.2 . . . . .	210
6.4.3 . . . . .	212
6.4.4 . . . . .	213

6.4.5 . . . . .	214
6.4.6 . . . . .	215
6.4.7 . . . . .	215
6.4.8 . . . . .	220
6.5.1 . . . . .	228
6.5.2 . . . . .	229
6.5.3 . . . . .	230
6.5.4 . . . . .	232
6.5.5 . . . . .	233
6.5.6 . . . . .	244
6.6.1 . . . . .	248
6.6.2 . . . . .	249
6.6.3 . . . . .	250
6.6.4 . . . . .	251
6.6.5 . . . . .	261
6.7.1 . . . . .	264
7.2.1 . . . . .	268
7.2.2 . . . . .	269
7.2.3 . . . . .	270
7.2.4 . . . . .	271
7.2.5 . . . . .	274
7.2.6 . . . . .	275
7.2.7 . . . . .	279
7.3.1 . . . . .	282
7.3.2 . . . . .	283
7.4.1 . . . . .	293
7.4.2 . . . . .	298
7.5.1 . . . . .	302
8.1.1 . . . . .	304
8.2.1 . . . . .	305
8.2.2 . . . . .	306
8.2.3 . . . . .	307
8.2.4 . . . . .	308
8.2.5 . . . . .	310
8.2.6 . . . . .	310
8.2.7 . . . . .	311
8.2.8 . . . . .	312

*LIST OF FIGURES*

8.2.9 . . . . .	313
8.2.10. . . . .	316
8.2.11. . . . .	318
8.3.1 . . . . .	320
9.2.1 . . . . .	330
9.2.2 . . . . .	331
1.1.1 . . . . .	352
1.1.2 . . . . .	353
1.1.3 . . . . .	354
1.3.1 . . . . .	359
2.1.1 . . . . .	362
2.2.1 . . . . .	364
2.3.1 . . . . .	365
2.6.1 . . . . .	373
2.6.2 . . . . .	375



# List of Tables

5.3.1 . . . . .	125
5.5.1 . . . . .	166
9.2.1 . . . . .	329
9.2.2 . . . . .	329
9.3.1 . . . . .	334
9.3.2 . . . . .	335
9.3.3 . . . . .	335
9.3.4 . . . . .	336
9.3.5 . . . . .	336
9.3.6 . . . . .	337
9.3.7 . . . . .	337
9.3.8 . . . . .	338
9.3.9 . . . . .	338
9.3.10. . . . .	339
9.3.11. . . . .	339
9.4.1 . . . . .	342
9.4.2 . . . . .	342
9.4.3 . . . . .	344
9.4.4 . . . . .	344
9.5.1 . . . . .	345
9.5.2 . . . . .	348



# Preface

This book is intended to introduce the reader to the important practical problem of solving large systems of sparse linear equations on a computer. The problem has many facets, from fundamental questions about the inherent complexity of certain problems, to less precisely specified questions about the design of efficient data structures and computer programs. In order to limit the size of the book, and yet consider the problems in detail, we have restricted our attention to symmetric positive definite systems of equations. Such problems are very common, arising in numerous fields of science and engineering. For similar reasons, we have limited our treatment to one specific method for each general approach to solving large sparse positive definite systems. For example, among the numerous methods for approximately minimizing the bandwidth of a matrix, we have selected only one, which through our experience has appeared to perform well. Our objective is to expose the reader to the important ideas, rather than the method which is necessarily best for his particular problem. Our hope is that someone familiar with the contents of the book could make sound judgements about the applicability and appropriateness of proposed ideas and methods for solving sparse systems.

The quality of the computer implementation of sparse matrix algorithms can have a profound effect on their performance, and the difficulty of implementation varies a great deal from one algorithm to another. Thus, while “paper and pencil” analyses of sparse matrix algorithms are useful and important, they are not enough. Our view is that studying and using subroutines which implement these algorithms is an essential component in a good introduction to this important area of scientific computation. To this end, we provide listings of Fortran subroutines, and discuss them in detail. The procedure for obtaining machine readable copies of these is provided in Appendix A.

We are grateful to Mary Wang for doing a superb job of typing the orig-



inal manuscript, and to Anne Trip de Roche and Heather Pente for coping with our numerous revisions. We are also grateful to the many students who debugged early versions of the manuscript, and in particular to Mr. Hamza Rashwan and Mr. Esmond Ng for a careful reading of the final manuscript.

Writing a book consumes time that might otherwise be spent with ones wife and children. We are grateful to our wives for their patience and understanding, and we dedicate this book to them.

Alan George  
Joseph W-H Liu

# Chapter 1

## Introduction

### 1.1 About the Book

This book deals with efficient computer methods for solving large sparse systems of linear algebraic equations. The reader is assumed to have a basic knowledge of linear algebra, and should be familiar with standard matrix notation and manipulation. Some basic knowledge of graph theory notation would be helpful, but it is not required since all the relevant notions and notations are introduced as they are needed.

This is a book about *computing*, and it contains numerous Fortran subroutines which are to be studied and used. Thus, the reader should have at least a basic understanding of Fortran, and ideally one should have access to a computer to execute programs using the subroutines in the book. The success of algorithms for sparse matrix computations, perhaps more than in any other area of numerical computation, depends on the quality of their *computer implementation*; i.e., the computer program which executes the algorithm. Implementations of these algorithms characteristically involve fairly complicated storage schemes, and the degree of complication varies substantially for different algorithms. Some algorithms which appear extremely attractive “on paper” may be much less so in practice because their implementation is complicated and inefficient. Other less theoretically attractive algorithms may be more desirable in practical terms because their implementation is simple and incurs very little “overhead.”

For these and other reasons which will be apparent later, we have included Fortran subroutines which implement many of the important algorithms discussed in the book. We have also included some numerical exper-

iments which illustrate the implementation issues noted above, and which provide the reader with some information about the absolute time and storage that sparse matrix computations require on a typical computer. The subroutines have been carefully tested, and are written in a portable subset of Fortran (Ryder [46]). Thus, they should execute correctly on most computer systems without any changes. They would be a useful addition to the library of any computer center which does scientific computing. Machine readable copies of the subroutines, along with the test problems described and used in Chapter 9, are available from the authors.

Our hope is that this book will be valuable in at least two capacities. First, it can serve as a text for senior or graduate students in computer science or engineering. The exercises at the end of each chapter are designed to test the reader's understanding of the material, to provide avenues for further investigation, and to suggest some important research problems. Some of the exercises involve using and/or changing the programs we provide, so it is desirable to have access to a computer which supports the Fortran language, and to have the programs available in a computer library.

This book should also serve as a useful reference for all scientists and engineers involved in solving large sparse positive definite matrix problems. Although this class of problems is special, a substantial fraction (perhaps the majority) of linear equation problems arising in science and engineering have this property. It is a large enough class to warrant separate treatment. In addition, as we shall see later, the solution of sparse problems with this property is fundamentally different from that for the general case.

## 1.2 Cholesky's Method and the Ordering Problem

All the methods we discuss in this book are based on a single numerical algorithm known as *Cholesky's method*, a symmetric variant of Gaussian elimination tailored to symmetric positive definite matrices. We shall define this class of matrices and describe the method in detail in Section 2.2. Suppose the given system of equations to be solved is

$$\mathbf{Ax} = \mathbf{b}, \tag{1.2.1}$$

where  $\mathbf{A}$  is an  $n \times n$ , symmetric, positive definite *coefficient matrix*,  $\mathbf{b}$  is a vector of length  $n$ , called the *right hand side*, and  $\mathbf{x}$  is the *solution vector* of length  $n$ , whose components are to be computed. Applying Cholesky's

method to  $\mathbf{A}$  yields the *triangular factorization*

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T, \quad (1.2.2)$$

where  $\mathbf{L}$  is *lower triangular* with positive diagonal elements. A matrix  $\mathbf{M}$  is lower upper triangular if  $m_{ij} = 0$  for  $i < j$   $\{i > j\}$ . The superscript  $T$  indicates the *transpose* operation. In Section 2.2 we show that such a factorization always exists when  $\mathbf{A}$  is symmetric and positive definite.

Using (1.2.2) in (1.2.1) we have

$$\mathbf{L}\mathbf{L}^T \mathbf{x} = \mathbf{b}, \quad (1.2.3)$$

and by substituting  $\mathbf{y} = \mathbf{L}^T \mathbf{x}$ , it is clear we can obtain  $\mathbf{x}$  by solving the triangular systems

$$\mathbf{L}\mathbf{y} = \mathbf{b}, \quad (1.2.4)$$

and

$$\mathbf{L}^T \mathbf{x} = \mathbf{y}. \quad (1.2.5)$$

As an example, consider the problem

$$\begin{pmatrix} 4 & 1 & 2 & \frac{1}{2} & 2 \\ 1 & \frac{1}{2} & 0 & 0 & 0 \\ 2 & 0 & 3 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & \frac{5}{8} & 0 \\ 2 & 0 & 0 & 0 & 16 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 7 \\ 3 \\ 7 \\ -4 \\ -4 \end{pmatrix}. \quad (1.2.6)$$

The Cholesky factor of the coefficient matrix of (1.2.6) is given by

$$\mathbf{L} = \begin{pmatrix} 2 & & & & \\ 0.5 & 0.5 & & & \\ 1 & -1 & 1 & & \\ .25 & -.25 & -.5 & 0.5 & \\ 1 & -1 & -2 & -3 & 1 \end{pmatrix}. \quad (1.2.7)$$

Solving  $\mathbf{L}\mathbf{y} = \mathbf{b}$ , we obtain

$$\mathbf{y} = \begin{pmatrix} 3.5 \\ 2.5 \\ 6 \\ -2.5 \\ -0.50 \end{pmatrix},$$

and then solving  $\mathbf{L}^T \mathbf{x} = \mathbf{y}$  yields

$$\mathbf{x} = \begin{pmatrix} 2 \\ 2 \\ 1 \\ -8 \\ -0.50 \end{pmatrix}.$$

The example above illustrates the most important fact about applying Cholesky's method to a sparse matrix  $\mathbf{A}$ : the matrix usually suffers *fill-in*. That is,  $\mathbf{L}$  has nonzeros in positions which are zero in the lower triangular part of  $\mathbf{A}$ .

Now suppose we relabel the variables  $x_i$  according to the recipe  $x_i \rightarrow \tilde{x}_{5-i+1}$ ,  $i = 1, 2, \dots, 5$ , and rearrange the equations so that the last one becomes the first, the second last becomes the second, and so on, with the first equation finally becoming the last one. We then obtain the equivalent system of equations (1.2.8).

$$\begin{pmatrix} 16 & 0 & 0 & 0 & 2 \\ 0 & \frac{5}{8} & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 3 & 0 & 2 \\ 0 & 0 & 0 & \frac{1}{2} & 1 \\ 2 & \frac{1}{2} & 2 & 1 & 4 \end{pmatrix} \begin{pmatrix} \tilde{x}_1 \\ \tilde{x}_2 \\ \tilde{x}_3 \\ \tilde{x}_4 \\ \tilde{x}_5 \end{pmatrix} = \begin{pmatrix} -4 \\ -4 \\ 7 \\ 3 \\ 7 \end{pmatrix}. \quad (1.2.8)$$

It should be clear that this relabelling of the variables and reordering of the equations amounts to a symmetric permutation of the rows and columns of  $\mathbf{A}$ , with the same permutation applied to  $\mathbf{b}$ . We refer to this new system as  $\tilde{\mathbf{A}}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$ . Using Cholesky's method on this new system as before, we factor  $\tilde{\mathbf{A}}$  into  $\tilde{\mathbf{L}}\tilde{\mathbf{L}}^T$ , obtaining (to three significant figures)

$$\tilde{\mathbf{L}} = \begin{pmatrix} 4 & & & & \\ 0 & 0.791 & & & \\ 0 & 0 & 1.73 & & \\ 0 & 0 & & 0.707 & \\ 0.500 & 0.632 & 1.15 & 1.41 & 1.29 \end{pmatrix}.$$

Solving  $\tilde{\mathbf{L}}\tilde{\mathbf{y}} = \tilde{\mathbf{b}}$  and  $\tilde{\mathbf{L}}^T \tilde{\mathbf{x}} = \tilde{\mathbf{y}}$  yields the solution  $\tilde{\mathbf{x}}$ , which is simply a rearranged form of  $\mathbf{x}$ . The crucial point is that our reordering of the equations and variables provided a triangular factor  $\tilde{\mathbf{L}}$  which is now *just as sparse as the lower triangle of  $\mathbf{A}$* . Although it is rarely possible in practice to achieve

this, for most sparse matrix problems a judicious reordering of the rows and columns of the coefficient matrix can lead to enormous reductions in fill-in, and hence savings in computer execution time and storage (assuming of course that sparsity is exploited.) The study of algorithms which automatically perform this reordering process is one of the major topics of this book, along with a study of effective computational and storage schemes for the sparse factors  $\tilde{L}$  that these reorderings provide.

The 5 by 5 matrix example above illustrates the basic characteristics of sparse elimination and the effect of reordering. To emphasize these points, we consider a somewhat larger example, the zero-nonzero pattern of which is given in Figure 1.2.1. On factoring this matrix into  $LL^T$ , we obtain the structure shown in Figure 1.2.2. Evidently the matrix in its present ordering is not good for sparse elimination, since it has suffered a lot of fill.

Figures 1.2.3 and 1.2.5 display the structure of two symmetric permutations  $A'$  and  $A''$  of the matrix  $A$  whose structure is shown in Figure 1.2.1. The structure of their Cholesky factors  $L'$  and  $L''$  is shown in Figures 1.2.4 and 1.2.6 respectively. The matrix  $A'$  has been permuted into so-called *band form*, to be discussed in Chapter 4. The matrix  $A''$  has been ordered to reduce fill-in; a method for obtaining this type of ordering is the topic of Chapter 5. The number of nonzeros in  $L$ ,  $L'$  and  $L''$  is 369, 189, and 177 respectively.

As our example shows, some orderings can lead to dramatic reductions in the amount of fill, or confine it to certain specific parts of  $L$  which can be easily stored. This task of finding a “good” ordering, which we refer to as the “ordering problem,” is central to the study of the solution of sparse positive definite systems.

### 1.3 Positive Definite Versus Indefinite Matrix Problems

In this book we deal exclusively with the case when  $A$  is symmetric and positive definite. As we noted earlier, a substantial portion of linear equation problems arising in science and engineering have this property, and the ordering problem is both different from and easier than that for general sparse  $A$ . For a general indefinite sparse matrix  $A$ , some form of *pivoting* (row and/or column interchanges) is necessary to ensure numerical stability. Thus given  $A$ , one normally obtains a factorization of  $PA$  or  $PAQ$ , where  $P$  and  $Q$  are permutation matrices of the appropriate size. (note that the

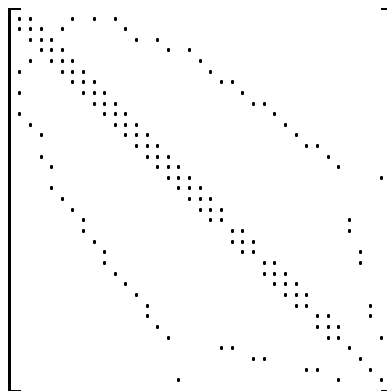


Figure 1.2.1: The structure of a 35 by 35 matrix  $A$ .

---

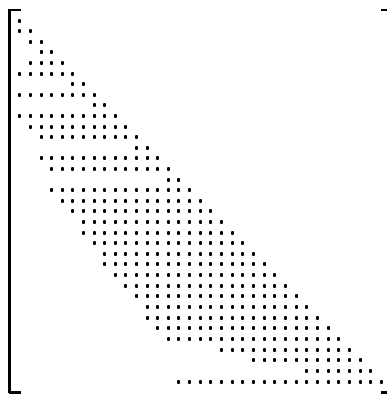


Figure 1.2.2: Nonzero pattern of the Cholesky factor  $L$  for the matrix whose structure is shown in Figure 1.2.1.

---

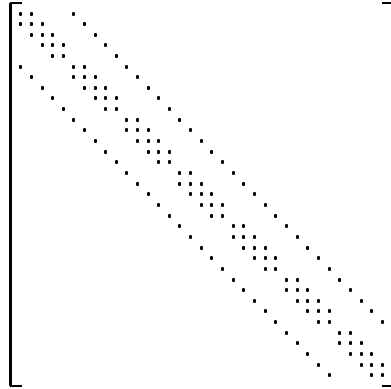


Figure 1.2.3: The structure of  $\mathbf{A}'$ , a symmetric permutation of the matrix  $\mathbf{A}$ , whose structure is shown in Figure 1.2.1.

---

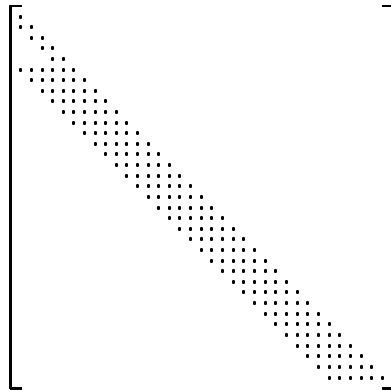


Figure 1.2.4: The structure of  $\mathbf{L}'$ , the Cholesky factor of  $\mathbf{A}'$ , whose structure is shown in Figure 1.2.3.

---



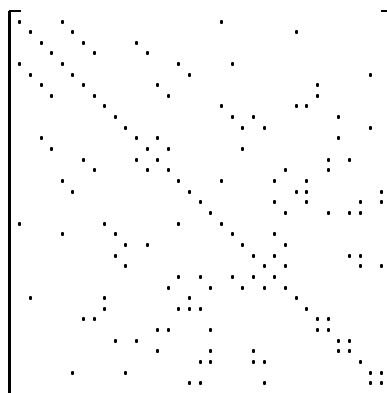


Figure 1.2.5: The structure of  $\mathbf{A}''$ , a symmetric permutation of the matrix  $\mathbf{A}$ , whose structure is shown in Figure 1.2.1.

---

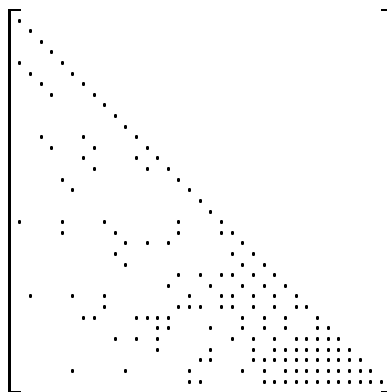


Figure 1.2.6: The structure of  $\mathbf{L}''$ , the Cholesky factor  $\mathbf{A}''$ , whose structure is shown in Figure 1.2.5.

---

application of  $\mathbf{P}$  on the left permutes the rows of  $\mathbf{A}$ , and the application of  $\mathbf{Q}$  on the right permutes the columns of  $\mathbf{A}$ .) These permutations are determined *during the factorization* by a combination of (usually competing) numerical stability and sparsity requirements (Duff [12]). Different matrices, even though they may have the same zero/nonzero pattern, will normally yield different  $\mathbf{P}$  and  $\mathbf{Q}$ , and therefore have factors with different sparsity patterns. In other words, it is in general not possible to predict where fill-in will occur for general sparse matrices before the computation begins. Thus, we are obliged to use some form of *dynamic* storage scheme which allocates storage for fill-in as the computation proceeds.

On the other hand, symmetric Gaussian elimination (Cholesky's method, or one of its variants, described in Chapter 2) applied to a symmetric positive definite matrix does not require interchanges (pivoting) to maintain numerical stability. Since  $\mathbf{PAP}^T$  is also symmetric and positive definite for any permutation matrix  $\mathbf{P}$ , this means we can choose to reorder  $\mathbf{A}$  symmetrically i) without regard to numerical stability and ii) before the actual numerical factorization begins.

These options, *which are normally not available to us when  $\mathbf{A}$  is a general indefinite matrix*, have enormous practical implications. Since the ordering can be determined before the factorization begins, the locations of the fill-in suffered during the factorization can also be determined. Thus, the data structure used to store  $\mathbf{L}$  can be constructed before the actual numerical factorization, and spaces for fill components can be reserved. The computation then proceeds with the storage structure remaining *static* (unaltered). Thus, the three problems of i) finding a suitable ordering, ii) setting up the appropriate storage scheme, and iii) the actual numerical computation, can be isolated as separate objects of study, as well as separate computer software modules, as depicted in Figure 1.3.1.

This independence of tasks has a number of distinct advantages. It encourages software modularity, and, in particular, allows us to tailor storage methods to the given task at hand. For example, the use of lists to store matrix subscripts may be quite appropriate for an implementation of an ordering algorithm, but decidedly inappropriate in connection with actually storing the matrix or its factors. In the same vein, knowing that we can use a storage scheme in a static manner during the factorization sometimes allows us to select a method which is very efficient in terms of storage requirements, but would be a disaster in terms of bookkeeping overhead if it had to be altered during the factorization. Finally, in many engineering design applications, numerous *different* positive definite matrix problems having the

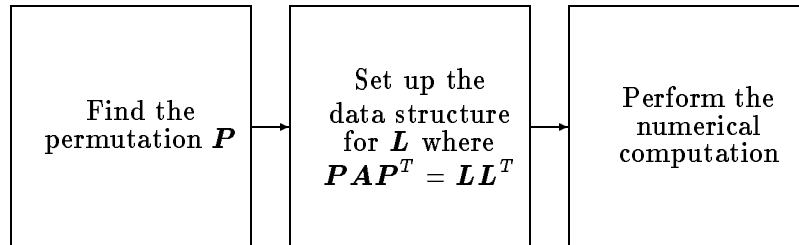


Figure 1.3.1: Sequence of tasks for sparse Cholesky factorization.

---

*same* structure must be solved. Obviously, the ordering and storage scheme set-up only needs to be performed once, so it is desirable to have these tasks isolated from the actual numerical computation.

In numerous practical situations matrix problems arise which are unsymmetric but have symmetric structure, and for which it can be shown that pivoting for numerical stability is not required when Gaussian elimination is applied. Almost all the ideas and algorithms described in this book extend immediately to this class of problems. Some hints on how this can be done are provided in Exercise 4.6.1 on page 103, Chapter 4.

## 1.4 Iterative Versus Direct Methods

Numerical methods for solving systems of linear equations fall into two general classes, *iterative* and *direct*. A typical iterative method involves the initial selection of an approximation  $\mathbf{x}^{(1)}$  to  $\mathbf{x}$ , and the determination of a sequence  $\mathbf{x}^{(2)}, \mathbf{x}^{(3)}, \dots$  such that  $\lim_{i \rightarrow \infty} \mathbf{x}^{(i)} = \mathbf{x}$ . Usually the calculation of  $\mathbf{x}^{(i+1)}$  involves only  $\mathbf{A}$ ,  $\mathbf{b}$ , and one or two of the previous iterates. In theory, when we use an iterative method we must perform an infinite number of arithmetic operations in order to obtain  $\mathbf{x}$ , but in practice we stop the iteration when we believe our current approximation is acceptably close to  $\mathbf{x}$ . On the other hand, in the absence of rounding errors, direct methods provide the solution after a finite number of arithmetic operations have been performed.

Which class of method is better? The question cannot be answered in general since it depends upon how we define “better,” and also upon the particular problem or class of problems to be solved. Iterative methods are

attractive in terms of computer storage requirements since their implementations typically require only  $\mathbf{A}$ ,  $\mathbf{b}$ ,  $\mathbf{x}^{(i)}$  and perhaps one or two other vectors to be stored. On the other hand, when  $\mathbf{A}$  is factored, it typically suffers some fill-in, so that the *filled matrix*  $\mathbf{F} = \mathbf{L} + \mathbf{L}^T$  has nonzeros in positions which are zero in  $\mathbf{A}$ . Thus, it is often true that direct methods for sparse systems require more storage than implementations of iterative methods. The actual ratio depends very much on the problem being solved, and also on the ordering used.

A comparison of iterative and direct methods in terms of computational requirements is even more complicated. As we have seen, the ordering used can dramatically affect the amount of arithmetic performed using Gaussian elimination. The number of iterations performed by an iterative scheme depends very much on the characteristics of  $\mathbf{A}$ , and on the sometimes delicate problem of determining, on the basis of computable quantities, when  $\mathbf{x}^{(i)}$  is “close enough” to  $\mathbf{x}$ .

In some situations, such as in the design of some mechanical devices, or the simulation of some time-dependent phenomena, many systems of equations having the same coefficient matrix must be solved. In this case, the cost of the direct scheme may be essentially that of solving the triangular system *given the factorization*, since the factorization cost amortized over all solutions may be negligible. In these situations it is also often the case that the number of iterations required by an iterative scheme is quite small, since a good starting vector  $\mathbf{x}^{(1)}$  is often available.

The above remarks should make it clear that unless the question of which class of method should be used is posed in a quite narrow and well defined context, it is either very complicated or impossible to answer. Our justification for considering only direct methods in this book is that several excellent references dealing with iterative methods are already available (Varga [4], Young [57]), while there is no such comparable reference known to the authors for direct methods for large sparse systems. In addition, there are situations where it can be shown quite convincingly that direct methods are far more desirable than any conceivable iterative scheme.



## Chapter 2

# Fundamentals

### 2.1 Introduction

In this chapter we examine the basic numerical algorithm used throughout the book to solve symmetric positive definite matrix problems. The method, known as Cholesky's method, was discussed briefly in Section 1.2. In what follows we prove that the factorization always exists for positive definite matrices, and examine several ways in which the computation can be performed. Although these are mathematically and (usually) numerically equivalent, they differ in the order in which the numbers are computed and used. These differences are important with respect to computer implementation of the method. We also derive expressions for the amount of arithmetic performed by the method.

As we indicated in Section 1.2, when Cholesky's method is applied to a sparse matrix  $\mathbf{A}$ , it generally suffers some fill-in, so that its Cholesky factor  $\mathbf{L}$  has nonzeros in positions which are zero in  $\mathbf{A}$ . For some permutation matrix  $\mathbf{P}$ , we can instead factor  $\mathbf{PAP}^T$  into  $\tilde{\mathbf{L}}\tilde{\mathbf{L}}^T$ , and  $\tilde{\mathbf{L}}$  may be much more attractive than  $\mathbf{L}$ , according to some criterion. In Section 2.4 we discuss some of these criteria, and indicate how practical implementation factors complicate the comparison of different methods.

#### 2.1.1 Notations

The reader is assumed to be familiar with the elementary theory and properties of matrices as presented in (Stewart [50]). In this section, we shall describe the matrix notations used throughout the book.

We shall use bold face capital italic letters for matrices. The entries of a matrix will be represented by lower case italic letters with two subscripts. For example, let  $\mathbf{A}$  be an  $n$  by  $n$  matrix. Its  $(i, j)$ -th element is denoted by  $a_{ij}$ . The number  $n$  is called the *order* of the matrix  $\mathbf{A}$ .

A vector will be denoted by a lower case bold italic letter and its elements by lower case letters with a single subscript. Thus, we have

$$\mathbf{v} = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix},$$

a vector of length  $n$ .

For a given matrix  $\mathbf{A}$ , its  $i$ -th row and  $i$ -th column are denoted by  $\mathbf{A}_{i*}$  and  $\mathbf{A}_{*i}$  respectively. When  $\mathbf{A}$  is symmetric, we have  $\mathbf{A}_{i*} = \mathbf{A}_{*i}^T$  for  $i = 1, \dots, n$ .

We shall use  $\mathbf{I}_n$  to represent the *identity matrix* of order  $n$ ; that is, the matrix with all entries zero except for ones on the diagonal.

In sparse matrix analysis, we often need to count the number of nonzeros in a vector or matrix. We use  $\eta(\square)$  to denote the number of nonzero components in  $\square$ , where  $\square$  stands for a vector or a matrix. Obviously,

$$\eta(\mathbf{I}_n) = n.$$

We also often need to refer to the number of members in a *set*  $S$ ; we denote this number by  $|S|$ .

Let  $f(n)$  and  $g(n)$  be functions of the independent variable  $n$ . We use the notation

$$f(n) = O(g(n))$$

if for some constant  $K$  and all sufficiently large  $n$ ,

$$\left| \frac{f(n)}{g(n)} \right| \leq K.$$

We say that  $f(n)$  has at most the *order of magnitude* of  $g(n)$ . This is a useful notation in the analysis of sparse matrix algorithms, since often we are only interested in the dominant term in arithmetic and nonzero counts.

For example, if  $f(n) = \frac{1}{6}n^3 + \frac{1}{2}n^2 - \frac{2}{3}n$ , we can write

$$f(n) = O(n^3).$$

For large enough  $n$ , the relative contribution from the terms  $\frac{1}{2}n^2$  and  $-\frac{2}{3}n$  is negligible.

Expressions such as  $f(n)$  above arise in counting arithmetic operations or numbers of nonzeros, and are often the result of some fairly complicated summations. Since we are usually only concerned with the dominant term, a very common device used to simplify the computation is to replace the summation by an integral sign. For example, for large  $n$ ,

$$\sum_{k=1}^n (2n+k)(n-k) \approx \int_0^n (2n+k)(n-k)dk.$$

### Exercises

2.1.1) Compute directly the sum  $\sum_{i=1}^n i^2(n-i)$ , and also approximate it using an integral, as described at the end of this section.

2.1.2) Compute directly the sum  $\sum_{i=1}^n \sum_{j=1}^{n-i+1} (i+j)$ , and also approximate it using a double integral.

2.1.3) Let  $\mathbf{A}$  and  $\mathbf{B}$  be two  $n$  by  $n$  sparse matrices. Show that the number of multiplications required to compute  $\mathbf{C} = \mathbf{AB}$  is given by

$$\sum_{i=1}^n \eta(\mathbf{A}_{*i})\eta(\mathbf{B}_{i*}).$$

2.1.4) Let  $\mathbf{B}$  be a given  $m$  by  $n$  sparse matrix. Show that the product  $\mathbf{B}^T \mathbf{B}$  can be computed using

$$\frac{1}{2} \sum_{i=1}^m \eta(\mathbf{B}_{i*})(\eta(\mathbf{B}_{i*}) + 1)$$

multiplications.

2.1.5) A common scheme to store a sparse vector has a main storage array which contains all the nonzero entries in the vector, and an accompanying vector which gives the subscripts of the nonzeros. Let  $\mathbf{u}$  and  $\mathbf{v}$  be two sparse vectors of size  $n$  stored in this format. Consider the computation of the inner product  $\mathbf{w} = \mathbf{u}^T \mathbf{v}$ .

a) If the subscript vectors are in ascending (or descending) order, show that the inner product can be done using only  $O(\eta(\mathbf{u}) + \eta(\mathbf{v}))$  comparisons.



- b) What if the subscripts are in random order?
- c) How would you perform the computation if the subscripts are in random order and a temporary real array of size  $n$  with all zero entries is provided?

## 2.2 The Factorization Algorithm

### 2.2.1 Existence and Uniqueness of the Factorization

A symmetric matrix  $\mathbf{A}$  is *positive definite* if  $\mathbf{x}^T \mathbf{A} \mathbf{x} > 0$  for all nonzero vectors  $\mathbf{x}$ . Such matrices arise in many applications; typically  $\mathbf{x}^T \mathbf{A} \mathbf{x}$  represents the energy of some physical system which is positive for any configuration  $\mathbf{x}$ . In a positive definite matrix  $\mathbf{A}$  the diagonal entries are always positive since

$$\mathbf{e}_i^T \mathbf{A} \mathbf{e}_j = a_{ij},$$

where  $\mathbf{e}_i$  is the  $i$ -th *characteristic vector*, the components of which are all zeros except for a one in the  $i$ -th position. This observation will be used in proving the following factorization theorem due to Cholesky (Stewart [50]).

**Theorem 2.2.1** *If  $\mathbf{A}$  is an  $n$  by  $n$  symmetric positive definite matrix, it has a unique triangular factorization  $\mathbf{L}\mathbf{L}^T$ , where  $\mathbf{L}$  is a lower triangular matrix with positive diagonal entries.*

**Proof:** The proof is by induction on the order of the matrix  $\mathbf{A}$ . The result is certainly true for one by one matrices since  $a_{11}$  is positive.

Suppose the assertion is true for matrices of order  $n - 1$ . Let  $\mathbf{A}$  be a symmetric positive definite matrix of order  $n$ . It can be partitioned into the form

$$\mathbf{A} = \begin{pmatrix} d & \mathbf{v}^T \\ \mathbf{v} & \bar{\mathbf{H}} \end{pmatrix},$$

where  $d$  is a positive scalar and  $\bar{\mathbf{H}}$  is an  $n - 1$  by  $n - 1$  submatrix. The partitioned matrix can be written as the product

$$\begin{pmatrix} \sqrt{d} & \mathbf{0} \\ \frac{\mathbf{v}}{\sqrt{d}} & \mathbf{I}_{n-1} \end{pmatrix} \begin{pmatrix} 1 & \mathbf{0} \\ \mathbf{0} & \mathbf{H} \end{pmatrix} \begin{pmatrix} \sqrt{d} & \frac{\mathbf{v}^T}{\sqrt{d}} \\ \mathbf{0} & \mathbf{I}_{n-1} \end{pmatrix},$$

where  $\mathbf{H} = \bar{\mathbf{H}} - \frac{\mathbf{v}\mathbf{v}^T}{d}$ . Clearly the matrix  $\mathbf{H}$  is symmetric. It is also positive definite since for any nonzero vector  $\mathbf{x}$  of length  $n - 1$ ,

$$\begin{pmatrix} -\frac{\mathbf{x}^T \mathbf{v}}{d} & \mathbf{x}^T \end{pmatrix} \begin{pmatrix} d & \mathbf{v}^T \\ \mathbf{v} & \bar{\mathbf{H}} \end{pmatrix} \begin{pmatrix} -\frac{\mathbf{x}^T \mathbf{v}}{d} \\ \mathbf{x} \end{pmatrix} = \mathbf{x}^T \left( \bar{\mathbf{H}} - \frac{\mathbf{v}\mathbf{v}^T}{d} \right) \mathbf{x}$$

$$= \mathbf{x}^T \mathbf{H} \mathbf{x},$$

which implies  $\mathbf{x}^T \mathbf{H} \mathbf{x} > 0$ . By the induction assumption,  $\mathbf{H}$  has a triangular factorization  $\mathbf{L}_H \mathbf{L}_H^T$  with positive diagonals. Thus,  $\mathbf{A}$  can be expressed as

$$\begin{aligned} & \begin{pmatrix} \sqrt{d} & \mathbf{0} \\ \frac{\mathbf{v}}{\sqrt{d}} & \mathbf{I}_{n-1} \end{pmatrix} \begin{pmatrix} 1 & \mathbf{0} \\ \mathbf{0} & \mathbf{L}_H \end{pmatrix} \begin{pmatrix} 1 & \mathbf{0} \\ \mathbf{0} & \mathbf{L}_H^T \end{pmatrix} \begin{pmatrix} \sqrt{d} & \frac{\mathbf{v}^T}{\sqrt{d}} \\ \mathbf{0} & \mathbf{I}_{n-1} \end{pmatrix} \\ &= \begin{pmatrix} \sqrt{d} & \mathbf{0} \\ \frac{\mathbf{v}}{\sqrt{d}} & \mathbf{L}_H \end{pmatrix} \begin{pmatrix} \sqrt{d} & \frac{\mathbf{v}^T}{\sqrt{d}} \\ \mathbf{0} & \mathbf{L}_H^T \end{pmatrix} \\ &= \mathbf{L} \mathbf{L}^T. \end{aligned}$$

It is left to the reader to show that the factor  $\mathbf{L}$  is unique. □

If we apply the result to the matrix example

$$\begin{pmatrix} 4 & 8 \\ 8 & 25 \end{pmatrix},$$

we obtain the factors

$$\begin{pmatrix} 2 & 0 \\ 4 & 3 \end{pmatrix} \begin{pmatrix} 2 & 4 \\ 0 & 3 \end{pmatrix}.$$

It is appropriate here to point out that there is a closely related factorization of a symmetric positive definite matrix (Martin [40]). Since the Cholesky factor  $\mathbf{L}$  has positive diagonal elements, one can factor out a diagonal matrix  $\mathbf{D}^{1/2}$  from  $\mathbf{L}$ , yielding  $\mathbf{L} = \tilde{\mathbf{L}} \mathbf{D}^{1/2}$  whence we have

$$\mathbf{A} = \tilde{\mathbf{L}} \mathbf{D} \tilde{\mathbf{L}}^T. \tag{2.2.1}$$

In the above matrix example, this alternative factorization is

$$\begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix} \begin{pmatrix} 4 & 0 \\ 0 & 9 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}.$$

This factorization is as easy to compute as the original, and can be obtained without square root calculation (see Exercise 2.2.4 on page 25). We do not use it in our book because in some circumstances it leads to certain disagreeable asymmetries in calculations involving partitioned matrices.

### 2.2.2 Computing the Factorization

Theorem 2.2.1 guarantees the existence and uniqueness of the Cholesky factor for a symmetric positive definite matrix, but the order and the way in which the components of the factor  $\mathbf{L}$  are actually computed can vary. In this section, we examine some different ways in which  $\mathbf{L}$  can be computed; these options are important because they provide us with flexibility in the design of storage schemes for the sparse matrix factor  $\mathbf{L}$ .

The constructive proof of Theorem 2.2.1 suggests a computational scheme to determine the factor  $\mathbf{L}$ . It is the so-called *outer product form* of the algorithm. The scheme can be described step by step in matrix terms as follows.

$$\begin{aligned}
 \mathbf{A} = \mathbf{A}_0 = \mathbf{H}_0 &= \begin{pmatrix} d_1 & \mathbf{v}_1^T \\ \mathbf{v}_1 & \bar{\mathbf{H}}_1 \end{pmatrix} & (2.2.2) \\
 &= \begin{pmatrix} \sqrt{d_1} & \mathbf{0} \\ \frac{\mathbf{v}_1}{\sqrt{d_1}} & \mathbf{I}_{n-1} \end{pmatrix} \begin{pmatrix} 1 & \mathbf{0} \\ \mathbf{0} & \bar{\mathbf{H}}_1 - \frac{\mathbf{v}_1 \mathbf{v}_1^T}{d_1} \end{pmatrix} \begin{pmatrix} \sqrt{d_1} & \frac{\mathbf{v}_1^T}{\sqrt{d_1}} \\ \mathbf{0} & \mathbf{I}_{n-1} \end{pmatrix} \\
 &= \mathbf{L}_1 \begin{pmatrix} 1 & \mathbf{0} \\ \mathbf{0} & \mathbf{H}_1 \end{pmatrix} \mathbf{L}_1^T \\
 &= \mathbf{L}_1 \mathbf{A}_1 \mathbf{L}_1^T,
 \end{aligned}$$

$$\begin{aligned}
 \mathbf{A}_1 &= \begin{pmatrix} 1 & \mathbf{0} \\ \mathbf{0} & \mathbf{H}_1 \end{pmatrix} = \begin{pmatrix} 1 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & d_2 & \mathbf{v}_2^T \\ \mathbf{0} & \mathbf{v}_2 & \bar{\mathbf{H}}_2 \end{pmatrix} \\
 &= \begin{pmatrix} 1 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \sqrt{d_2} & \mathbf{0} \\ \mathbf{0} & \frac{\mathbf{v}_2}{\sqrt{d_2}} & \mathbf{I}_{n-2} \end{pmatrix} \begin{pmatrix} 1 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & 1 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \bar{\mathbf{H}}_2 - \frac{\mathbf{v}_2 \mathbf{v}_2^T}{d_2} \end{pmatrix} \begin{pmatrix} 1 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \sqrt{d_2} & \frac{\mathbf{v}_2^T}{\sqrt{d_2}} \\ \mathbf{0} & \mathbf{0} & \mathbf{I}_{n-2} \end{pmatrix} \\
 &= \mathbf{L}_2 \mathbf{A}_2 \mathbf{L}_2^T,
 \end{aligned}$$

⋮

$$\mathbf{A}_{n-1} = \mathbf{L}_n \mathbf{I}_n \mathbf{L}_n^T.$$

Here, for  $1 \leq i \leq n$ ,  $d_i$  is a positive scalar,  $\mathbf{v}_i$  is a vector of length  $n - i$ , and  $\mathbf{H}_i$  is an  $n - i$  by  $n - i$  positive definite symmetric matrix.

After  $n$  steps of the algorithm, we have

$$\mathbf{A} = \mathbf{L}_1 \mathbf{L}_2 \cdots \mathbf{L}_n \mathbf{L}_n^T \cdots \mathbf{L}_2^T \mathbf{L}_1^T = \mathbf{L} \mathbf{L}^T,$$

where it can be shown (see Exercise 2.2.6 on page 26 ) that

$$\mathbf{L} = \mathbf{L}_1 + \mathbf{L}_2 + \cdots + \mathbf{L}_n - (n - 1)\mathbf{I}_n. \quad (2.2.3)$$

Thus, the  $i$ -th column of  $\mathbf{L}$  is precisely the  $i$ -th column of  $\mathbf{L}_i$ .

In this scheme, the columns of  $\mathbf{L}$  are computed one by one. At the same time, each step involves the modification of the submatrix  $\bar{\mathbf{H}}_i$  by the outer product  $\mathbf{v}_i \mathbf{v}_i^T / d_i$  to give  $\mathbf{H}_i$ , which is simply the submatrix remaining to be factored. The access to the components of  $\mathbf{A}$  during the factorization is depicted as follows.

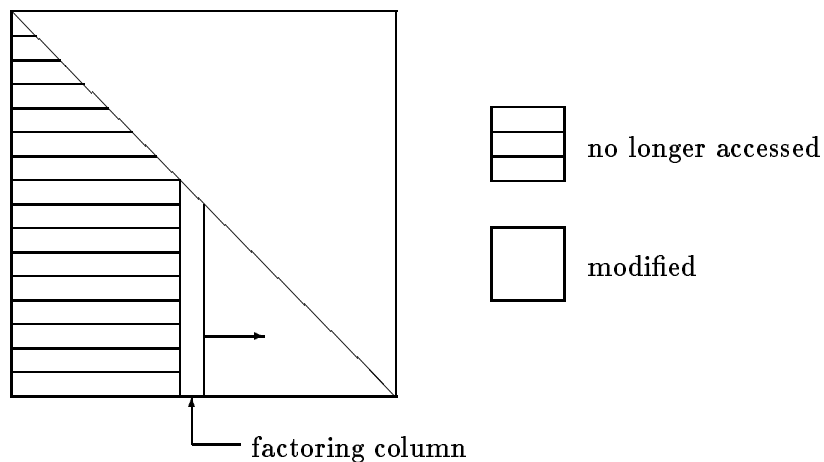


Figure 2.2.1: Access pattern in the outer product formulation of the Cholesky factorization algorithm.

---

An alternative formulation of the factorization process is the *bordering method*. Suppose the matrix  $\mathbf{A}$  is partitioned as

$$\mathbf{A} = \begin{pmatrix} \mathbf{M} & \mathbf{u} \\ \mathbf{u}^T & s \end{pmatrix},$$

where the symmetric factorization  $\mathbf{L}_M \mathbf{L}_M^T$  of the  $n - 1$  by  $n - 1$  leading principal submatrix  $\mathbf{M}$  has already been obtained. (Why is  $\mathbf{M}$  positive definite?) Then the factorization of  $\mathbf{A}$  is given by

$$\mathbf{A} = \begin{pmatrix} \mathbf{L}_M & \mathbf{0} \\ \mathbf{w}^T & t \end{pmatrix} \begin{pmatrix} \mathbf{L}_M^T & \mathbf{w} \\ \mathbf{0} & t \end{pmatrix}, \quad (2.2.4)$$

where

$$\mathbf{w} = \mathbf{L}^{-1} \mathbf{M} \mathbf{u} \quad (2.2.5)$$

and

$$t = (s - \mathbf{w}^T \mathbf{w})^{1/2}.$$

(Why is  $s - \mathbf{w}^T \mathbf{w}$  positive?)

Note that the factorization  $\mathbf{L} \mathbf{M} \mathbf{L}^T$  of the submatrix  $\mathbf{M}$  is also obtained by the bordering technique. So, the scheme can be described as follows.

For  $i = 1, 2, \dots, n$ ,

$$\text{Solve } \begin{pmatrix} l_{1,1} & & \mathbf{O} \\ \vdots & \ddots & \\ l_{i-1,1} & \cdots & l_{i-1,i-1} \end{pmatrix} \begin{pmatrix} l_{i,1} \\ \vdots \\ l_{i,i-1} \end{pmatrix} = \begin{pmatrix} a_{i,1} \\ \vdots \\ a_{i,i-1} \end{pmatrix}.$$

$$\text{Compute } l_{i,i} = \left( a_{i,i} - \sum_{k=1}^{i-1} l_{i,k}^2 \right)^{1/2}.$$

In this scheme, the rows of  $\mathbf{L}$  are computed one at a time; the part of the matrix remaining to be factored is not accessed until the corresponding part of  $\mathbf{L}$  is to be computed. The sequence of computations can be depicted as follows.

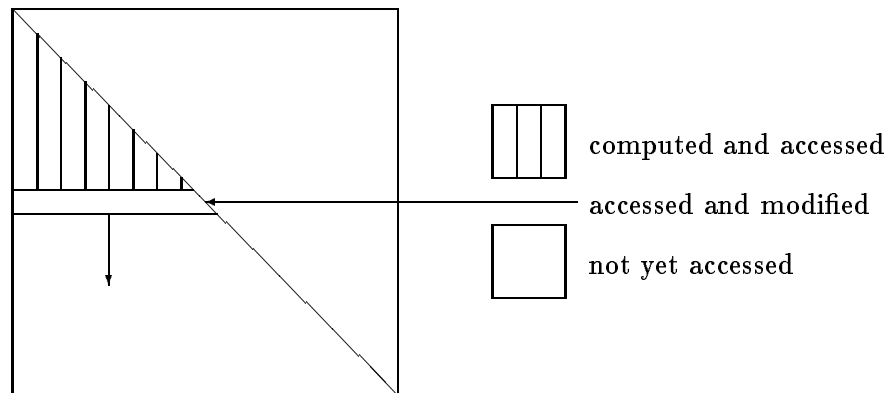


Figure 2.2.2: Access pattern in the bordering method.

---

The final scheme for computing the components of  $\mathbf{L}$  is the *inner product form* of the algorithm. It can be described as follows.

For  $j = 1, 2, \dots, n$

$$\text{Compute } l_{j,j} = \left( a_{j,j} - \sum_{k=1}^{j-1} l_{j,k}^2 \right)^{1/2}.$$

For  $i = j + 1, j + 2, \dots, n$

$$\text{Compute } l_{i,j} = \left( a_{i,j} - \sum_{k=1}^{j-1} l_{i,k} l_{j,k} \right) / l_{j,j}.$$

These formulae can be derived directly by equating the elements of  $\mathbf{A}$  to the corresponding elements of the product  $\mathbf{L}\mathbf{L}^T$ .

Like the outer product version of the algorithm, the columns of  $\mathbf{L}$  are computed one by one, but the part of the matrix remaining to be factored is not accessed during the scheme. The sequence of computations and the relevant access to the components of  $\mathbf{A}$  (or  $\mathbf{L}$ ) is depicted as follows.

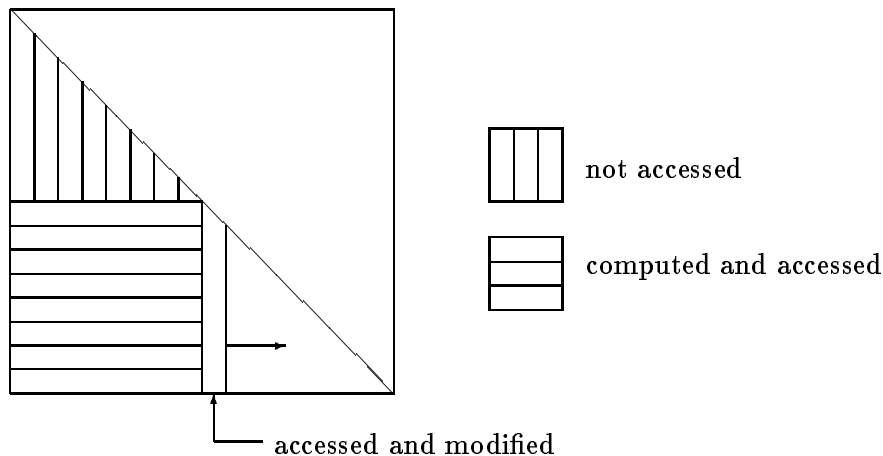


Figure 2.2.3: Access pattern in the inner product formulation of the Cholesky factorization algorithm.

---

The latter two formulations can be organized so that only inner products are involved. This can be used to improve the accuracy of the numerical factorization by accumulating the inner products in double precision. On some computers, this can be done at little extra cost.

### 2.2.3 Sparse Matrix Factorization

As we have seen in Chapter 1, when a sparse matrix is factored, it usually suffers some fill-in; that is, the lower triangular factor  $\mathbf{L}$  has nonzero components in positions which are zero in the original matrix. Recall in Section 1.2 the factorization of the matrix example

$$\mathbf{A} = \begin{pmatrix} 4 & 1 & 2 & \frac{1}{2} & 2 \\ 1 & \frac{1}{2} & 0 & 0 & 0 \\ 2 & 0 & 3 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & \frac{5}{8} & 0 \\ 2 & 0 & 0 & 0 & 16 \end{pmatrix}.$$

Its triangular factor  $\mathbf{L}$  is given by

$$\mathbf{L} = \begin{pmatrix} 2 & 0 & 0 & 0 & 0 \\ 0.5 & 0.5 & 0 & 0 & 0 \\ 1 & -1 & 1 & 0 & 0 \\ 0.25 & -0.25 & -0.5 & 0.5 & 0 \\ 1 & -1 & -2 & -3 & 1 \end{pmatrix}$$

so that the matrix  $\mathbf{A}$  suffers fill at  $a_{32}$ ,  $a_{42}$ ,  $a_{43}$ ,  $a_{52}$ ,  $a_{53}$  and  $a_{54}$ . This phenomenon of *fill-in*, which is usually ignored in solving dense systems, plays a crucial role in sparse elimination.

The creation of nonzero entries can be best understood using the outer-product formulation of the factorization process. At the  $i$ -th step, the submatrix  $\bar{\mathbf{H}}_i$  is modified by the matrix  $\mathbf{v}_i \mathbf{v}_i^T / d_i$  to give  $\mathbf{H}_i$ . As a result, the submatrix  $\mathbf{H}_i$  may have nonzeros in locations which are zero in  $\bar{\mathbf{H}}_i$ . In the example above,

$$\bar{\mathbf{H}}_1 = \begin{pmatrix} \frac{1}{2} & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & \frac{5}{8} & 0 \\ 0 & 0 & 0 & 16 \end{pmatrix}$$

and it is modified at step 1 to give (to three significant figures)

$$\mathbf{H}_1 = \bar{\mathbf{H}}_1 - \frac{1}{4} \begin{pmatrix} 1 \\ 2 \\ \frac{1}{2} \\ 2 \end{pmatrix} \begin{pmatrix} 1 & 2 & \frac{1}{2} & 2 \end{pmatrix}$$

$$= \begin{pmatrix} .25 & -.5 & -.125 & -.5 \\ -.5 & 2 & -.25 & -1 \\ -.125 & -.25 & .563 & -.25 \\ -.5 & -1 & -.25 & 15 \end{pmatrix}.$$

If zeros are exploited in solving a sparse system, fill-in affects both the storage and computation requirements. Recall that  $\eta(\square)$  is the number of nonzero components in  $\square$ , where  $\square$  stands for a vector or a matrix. Clearly, from (2.2.2) and (2.2.3), the number of nonzeros in  $\mathbf{L}$  is given by

$$\eta(\mathbf{L}) = n + \sum_{i=1}^{n-1} \eta(\mathbf{v}_i). \quad (2.2.6)$$

In the following theorem, and throughout the book, we measure arithmetic requirements by the number of multiplicative operations (multiplications and divisions), which we simply refer to as “operations.” The majority of the arithmetic performed in matrix operations involves sequences of arithmetic operations which occur in multiply-add pairs, so the number of additive operations is about equal to the number of multiplicative operations.

**Theorem 2.2.2** *The number of operations required to compute the triangular factor  $\mathbf{L}$  of the matrix  $\mathbf{A}$  is given by*

$$\frac{1}{2} \sum_{i=1}^{n-1} \eta(\mathbf{v}_i)(\eta(\mathbf{v}_i) + 3) = \frac{1}{2} \sum_{i=1}^{n-1} (\eta(\mathbf{L}_{*i}) - 1)(\eta(\mathbf{L}_{*i}) + 2). \quad (2.2.7)$$

**Proof:** The three formulations of the factorization differ only in the order in which operations are performed. For the purpose of counting operations, the outer-product formulation (2.2.2) is used. At the  $i$ -th step,  $\eta(\mathbf{v}_i)$  operations are required to compute  $\mathbf{v}_i/\sqrt{d_i}$ , and  $\frac{1}{2}\eta(\mathbf{v}_i)(\eta(\mathbf{v}_i) + 1)$  operations are needed to form the symmetric matrix

$$\frac{\mathbf{v}_i \mathbf{v}_i^T}{d_i} = \left( \frac{\mathbf{v}_i}{\sqrt{d_i}} \right) \left( \frac{\mathbf{v}_i}{\sqrt{d_i}} \right)^T.$$

The result follows from summing over all the steps. □

For the dense case, the number of nonzeros in  $\mathbf{L}$  is

$$\frac{1}{2}n(n + 1) \quad (2.2.8)$$



and the arithmetic cost is

$$\frac{1}{2} \sum_{i=1}^{n-1} i(i+3) = \frac{1}{6}n^3 + \frac{1}{2}n^2 - \frac{2}{3}n. \quad (2.2.9)$$

Consider also the Cholesky factorization of a symmetric positive definite tridiagonal matrix, an example of a sparse matrix. It can be shown (see Chapter 5) that if  $\mathbf{L}$  is its factor,

$$\eta(\mathbf{L}_{*i}) = 2, \text{ for } i = 1, \dots, n-1.$$

In this case, the number of nonzeros in  $\mathbf{L}$  is

$$\eta(\mathbf{L}) = 2n - 1,$$

and the arithmetic cost of computing  $\mathbf{L}$  is

$$\frac{1}{2} \sum_{i=1}^{n-1} 1(4) = 2(n-1).$$

Comparing these results with the counts for the dense case, we see a dramatic difference in storage and computational costs.

The costs for solving equivalent sparse systems with different orderings can also be very different. As illustrated in Section 1.2, the matrix example  $\mathbf{A}$  at the beginning of this section can be ordered so that it does not suffer any fill-in at all! The permutation matrix used is

$$\mathbf{P} = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

which reverses the ordering of  $\mathbf{A}$  when applied. We obtain the permuted matrix

$$\mathbf{PAP}^T = \begin{pmatrix} 16 & 0 & 0 & 0 & 2 \\ 0 & \frac{5}{8} & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 3 & 0 & 2 \\ 0 & 0 & 0 & \frac{1}{2} & 1 \\ 2 & \frac{1}{2} & 2 & 1 & 4 \end{pmatrix}$$

This simple example illustrates that a judicious choice of  $\mathbf{P}$  can result in dramatic reductions in fill-in and arithmetic requirements. Therefore, in solving a given linear equation problem

$$\mathbf{Ax} = \mathbf{b},$$

the *general procedure* involves first finding a permutation or ordering  $\mathbf{P}$  of the given problem. Then the system is expressed as

$$(\mathbf{PAP}^T)(\mathbf{Px}) = \mathbf{Pb}$$

and Cholesky's method is applied to the symmetric positive definite matrix  $\mathbf{PAP}^T$  yielding the triangular factorization  $\mathbf{LL}^T$ . By solving the equivalent permuted system, we can often achieve a reduction in the computer storage and execution time requirements.

### Exercises

2.2.1) Show that the Cholesky factorization for a symmetric positive definite matrix is unique.

2.2.2) Let  $\mathbf{A}$  be an  $n$  by  $n$  symmetric positive definite matrix. Show that

- a) any principal submatrix of  $\mathbf{A}$  is positive definite,
- b)  $\mathbf{A}$  is nonsingular and  $\mathbf{A}^{-1}$  is also positive definite,
- c)  $\max_{1 \leq i \leq n} a_{ii} = \max_{1 \leq i, j \leq n} |a_{ij}|$ .

2.2.3) Let  $\mathbf{A}$  be a symmetric positive definite matrix. Show that

- a)  $\mathbf{B}^T \mathbf{A} \mathbf{B}$  is positive definite if and only if  $\mathbf{B}$  is non-singular,
- b) the augmented matrix

$$\begin{pmatrix} \mathbf{A} & \mathbf{u} \\ \mathbf{u}^T & s \end{pmatrix}$$

is positive definite if and only if  $s > \mathbf{u}^T \mathbf{A}^{-1} \mathbf{u}$ .

2.2.4) Write out equations similar to those in (2.2.2) and (2.2.3) which yield the factorization  $\mathbf{LDL}^T$ , where  $\mathbf{L}$  is now lower triangular with ones on the diagonal, and  $\mathbf{D}$  is a diagonal matrix with positive diagonal elements.

- 2.2.5) Let  $\mathbf{E}$  and  $\mathbf{F}$  be two  $n$  by  $n$  lower triangular matrices which for some  $k$  ( $1 \leq k \leq n$ ) satisfy

$$\begin{aligned} e_{jj} &= 1 \text{ for } j > k \\ e_{ij} &= 0 \text{ for } i > j \text{ and } j > k \\ f_{jj} &= 1 \text{ for } j \leq k \\ f_{ij} &= 0 \text{ for } i > j \text{ and } j \leq k. \end{aligned}$$

The case when  $n = 6$  and  $k = 3$  is depicted below.

$$\mathbf{E} = \begin{pmatrix} \times & & & & & \\ \times & \times & & & & \mathbf{O} \\ \times & \times & \times & & & \\ \times & \times & \times & 1 & & \\ \times & \times & \times & 0 & 1 & \\ \times & \times & \times & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{F} = \begin{pmatrix} 1 & & & & & \\ 0 & 1 & & & & \mathbf{O} \\ 0 & 0 & 1 & & & \\ 0 & 0 & 0 & \times & & \\ 0 & 0 & 0 & \times & \times & \\ 0 & 0 & 0 & \times & \times & \times \end{pmatrix}$$

Show that  $\mathbf{EF} = \mathbf{E} + \mathbf{F} - \mathbf{I}$ , and hence prove that (2.2.3) holds.

- 2.2.6) Give an example of a symmetric matrix which does not have a triangular factorization  $\mathbf{LL}^T$  and one which has more than one factorization.

## 2.3 Solving Triangular Systems

### 2.3.1 Computing the Solution

Once we have computed the factorization, we must solve the triangular systems  $\mathbf{Ly} = \mathbf{b}$  and  $\mathbf{L}^T \mathbf{x} = \mathbf{y}$ . In this section, we consider the numerical solution of triangular systems.

Consider the  $n$  by  $n$  linear system

$$\mathbf{T}\mathbf{x} = \mathbf{b}$$

where  $\mathbf{T}$  is nonsingular and triangular. Without loss of generality, we assume that  $\mathbf{T}$  is lower triangular. There are two common ways of solving the system, which differ only in the order in which the operations are performed.

The first one involves the use of inner-products and the defining equations are given by:

For  $i = 1, 2, \dots, n$ ,

$$x_i = \left( b_i - \sum_{k=1}^{i-1} t_{i,k} x_k \right) / t_{i,i}. \quad (2.3.1)$$

The sequence of computations is depicted by the diagram in Figure 2.3.1.

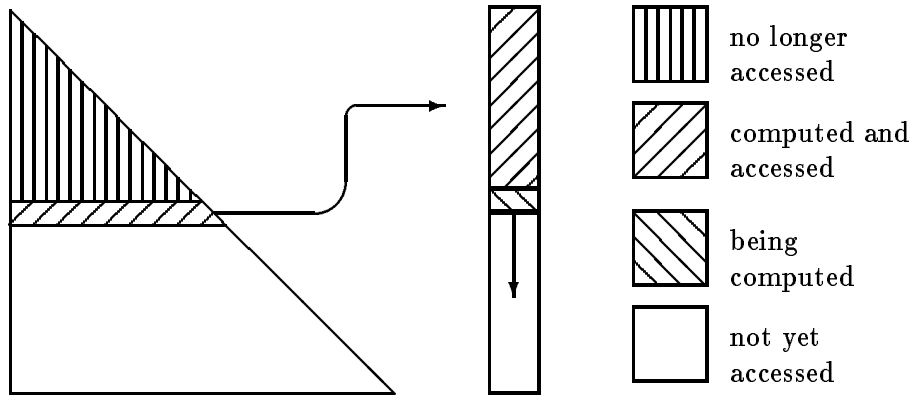


Figure 2.3.1: Access pattern in the inner product formulation of the triangular solution algorithm.

The second method uses the matrix components of  $T$  in the same way as the outer-product version of the factorization. The defining equations are as follows.

For  $i = 1, 2, \dots, n$ ,

$$x_i = b_i / t_{i,i}$$

$$\begin{pmatrix} b_{i+1} \\ \vdots \\ b_n \end{pmatrix} \leftarrow \begin{pmatrix} b_{i+1} \\ \vdots \\ b_n \end{pmatrix} - x_i \begin{pmatrix} t_{i+1,i} \\ \vdots \\ t_{n,i} \end{pmatrix} \quad (2.3.2)$$

Note that this scheme lends itself to exploiting sparsity in the solution  $\mathbf{x}$ . If  $b_i$  turns out to be zero at the beginning of the  $i$ -th step,  $x_i$  is zero and the entire step can be skipped. The access to components of the system is shown as follows.

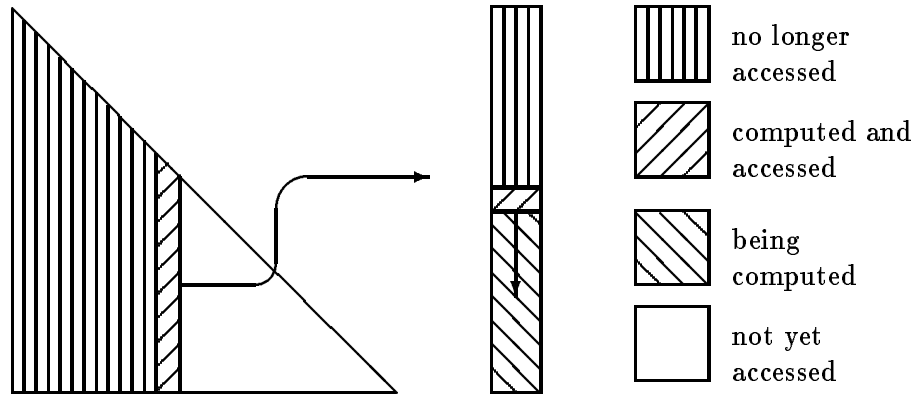


Figure 2.3.2: Access pattern in the outer product formulation of the triangular solution algorithm.

The former solution method accesses the components of the lower triangular matrix row by row and therefore lends itself to row-wise storage schemes. If the matrix is stored column by column, the latter method is more appropriate. It is interesting to note that this column oriented method is often used to solve the *upper* triangular system

$$L^T \mathbf{x} = \mathbf{y},$$

where  $L$  is a lower triangular matrix stored using a row-wise scheme.

### 2.3.2 Operation Counts

We now establish some simple results about solving triangular systems. They will be helpful later in obtaining operation counts.

Consider the solution of

$$T\mathbf{x} = \mathbf{b},$$

where  $T$  is nonsingular and lower triangular.

**Lemma 2.3.1** *The number of operations required to solve for  $\mathbf{x}$  is*

$$\sum_i \{\eta(T_{*i}) \mid x_i \neq 0\}$$

**Proof:** It follows from (2.3.2) that if  $x_i \neq 0$ , the  $i$ -th step requires  $\eta(\mathbf{T}_{*i})$  operations.  $\square$

**Corollary 2.3.2** *If the sparsity of the solution vector  $\mathbf{x}$  is not exploited (that is,  $\mathbf{x}$  is assumed to be full), then the number of operations required to compute  $\mathbf{x}$  is  $\eta(\mathbf{T})$ .*

Thus, it follows that the operation count for solving  $\mathbf{T}\mathbf{x} = \mathbf{b}$ , when  $\mathbf{T}$  and  $\mathbf{x}$  are full, is

$$\frac{1}{2}n(n+1). \quad (2.3.3)$$

The following results give some relationships between the structure of the right hand side  $\mathbf{b}$  and the solution  $\mathbf{x}$  of a lower triangular system. Lemma 2.3.3 and Corollary 2.3.5 appeal to a *no-cancellation assumption*; that is, whenever two nonzero quantities are added or subtracted, the result is nonzero. This means that in the analysis we ignore any zeros which might be created through exact cancellation. Such cancellation rarely occurs, and in order to predict such cancellation we would have to know the numerical values of  $\mathbf{T}$  and  $\mathbf{b}$ . Such a prediction would be difficult in general, particularly in floating point arithmetic which is subject to rounding error.

**Lemma 2.3.3** *With the no-cancellation assumption, if  $b_i \neq 0$  then  $x_i \neq 0$ .*

**Proof:** Since  $\mathbf{T}$  is non-singular,  $t_{ii} \neq 0$  for  $1 \leq i \leq n$ . The result then follows from the no-cancellation assumption and the defining equation (2.3.1) for  $x_i$ .  $\square$

**Lemma 2.3.4** *Let  $\mathbf{x}$  be the solution to  $\mathbf{T}\mathbf{x} = \mathbf{b}$ . If  $b_i = 0$  for  $1 \leq i \leq k$ , then  $x_i = 0$  for  $1 \leq i \leq k$ .*

**Corollary 2.3.5** *With the no-cancellation assumption,  $\eta(\mathbf{b}) \leq \eta(\mathbf{x})$ .*

### Exercises

2.3.1) Use Lemma 2.3.1 to show that factorization by the bordering scheme requires

$$\frac{1}{2} \sum_{i=1}^{n-1} (\eta(\mathbf{L}_{*i}) - 1)(\eta(\mathbf{L}_{*i}) + 2)$$

operations.

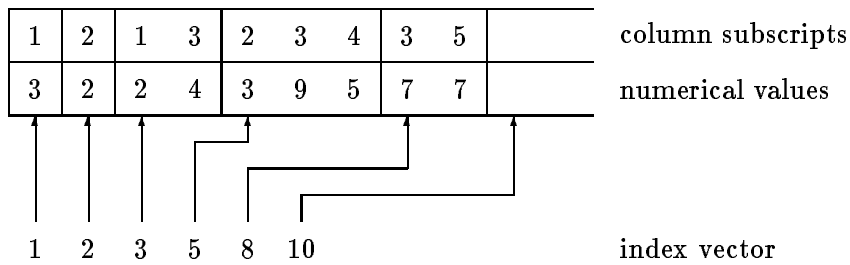
2.3.2) Show that the inverse of a nonsingular lower triangular matrix is lower triangular (use Lemma 2.3.4).

- 2.3.3) Let  $\mathbf{T}$  be a nonsingular lower triangular matrix with the *propagation property*, that is,  $t_{i,i-1} \neq 0$  for  $2 \leq i \leq n$ .
- Show that in solving  $\mathbf{T}\mathbf{x} = \mathbf{b}$ , if  $b_i \neq 0$  then  $x_j \neq 0$  for  $i \leq j \leq n$ .
  - Show that  $\mathbf{T}^{-1}$  is a full lower triangular matrix.
- 2.3.4) Does Lemma 2.3.1 depend upon the no-cancellation assumption? Explain. What about Theorem 2.2.2 and Lemma 2.3.4?
- 2.3.5) Prove a result analogous to Lemma 2.3.4 for upper triangular matrices.
- 2.3.6) Suppose you have numerous  $n$  by  $n$  lower triangular systems of the form  $\mathbf{L}\mathbf{y} = \mathbf{b}$  to solve, where  $\mathbf{L}$  and  $\mathbf{b}$  are both sparse. It is known that the solution  $\mathbf{y}$  is also sparse for these problems. You have a choice of two storage schemes for  $\mathbf{L}$ , as illustrated by the 5 by 5 example in Figure 2.3.3; one is column oriented and one is row oriented. Which one would you choose, and why would you choose it? If you wrote a Fortran program to solve such systems using your choice of data structures, would the execution time be proportional to the number of operations performed? Explain. (Assume that the number of operations performed is at least  $O(n)$ .)
- 2.3.7) Let  $\mathbf{L}$  and  $\mathbf{W}$  be  $n$  by  $n$  non-sparse lower triangular matrices, with nonzero diagonal elements. Approximately how many operations are required to compute  $\mathbf{L}^{-1}\mathbf{W}$ ? How many operations are required to compute  $\mathbf{W}^T\mathbf{W}$ ?
- 2.3.8) Suppose that  $n = 1 + k(p - 1)$  for some positive integer  $k$ , and that  $\mathbf{W}$  is an  $n$  by  $p$  full (pseudo) lower triangular matrix. That is,  $\mathbf{W}$  has zeros above position  $1 + (i - 1)k$  in column  $i$  of  $\mathbf{W}$ , and is nonzero otherwise. An example with  $n = 7$  and  $p = 4$  appears below. Roughly how many operations are required to compute  $\mathbf{W}^T\mathbf{W}$ , in terms of  $n$  and  $p$ ?

$$\begin{pmatrix} \times & & & & & & \\ & \times & & & & & \\ & & \times & & & & \\ & & & \times & & & \\ & & & & \times & & \\ & & & & & \times & \\ & & & & & & \times \end{pmatrix}$$

$$L = \begin{pmatrix} 3 & & & & \\ 0 & 2 & & & O \\ 2 & 0 & 4 & & \\ 0 & 3 & 9 & 5 & \\ 0 & 0 & 7 & 0 & 7 \end{pmatrix}$$

Scheme 1



Scheme 2

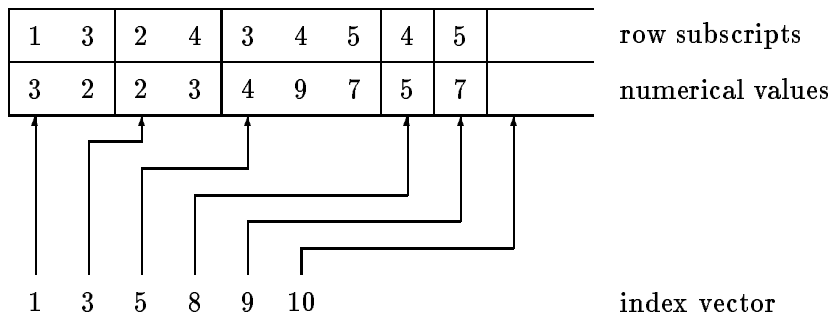


Figure 2.3.3: Two storage schemes for a 5 by 5 lower triangular matrix  $L$ .



- 2.3.9) Suppose  $\mathbf{L}$  is a nonsingular  $n$  by  $n$  lower triangular matrix, and  $\mathbf{W}$  is as described in Exercise 2.3.8. Approximately how many operations are required to compute  $\mathbf{L}^{-1}\mathbf{W}$ , as a function of  $n$  and  $p$ ?
- 2.3.10) a) Suppose  $\mathbf{A} = \mathbf{L}\mathbf{L}^T$ , where  $\mathbf{L}$  is as in Exercise 2.3.9 and  $\eta(\mathbf{L}_{*i}) \geq 2$ ,  $1 \leq i \leq n$ . Assuming the no-cancellation assumption, show that computing  $\mathbf{A}^{-1}$  by solving  $\mathbf{L}\mathbf{W} = \mathbf{I}$  and  $\mathbf{L}^T\mathbf{Z} = \mathbf{W}$  yields a full matrix.
- b) Suppose  $\mathbf{A}$  is unsymmetric with triangular factorization  $\mathbf{L}\mathbf{U}$ , where  $\mathbf{L}$  is unit lower triangular and  $\mathbf{U}$  is upper triangular. State the conditions and results analogous to those in a) above.

## 2.4 Some Practical Considerations

The objective of studying sparse matrix techniques for solving linear systems is to reduce *cost* by exploiting sparsity of the given system. We have seen in Section 2.2.3 that it is possible to achieve drastic reductions in storage and arithmetic requirements, when the solutions of dense and tridiagonal systems are compared.

There are various kinds of sparse storage schemes, which differ in the way zeros are exploited. Some might store some zeros in exchange for a simpler storage scheme; others exploit all the zeros in the system. In Chapters 4 to 8, we discuss the commonly used sparse schemes for solving linear systems.

The choice of a storage method naturally affects the storage requirement, and the use of ordering strategies (choice of permutation matrix  $\mathbf{P}$ ). Moreover, it has significant impact on the implementation of the factorization and solution, and hence on the complexity of the programs and the execution time.

However, irrespective of what sparse storage scheme is used, there are four distinct phases that can be identified in the entire computational process.

**Step 1 (Ordering)** Find a “good” ordering (permutation  $\mathbf{P}$ ) for the given matrix  $\mathbf{A}$ , with respect to the chosen storage method.

**Step 2 (Storage allocation)** Determine the necessary information about the Cholesky factor  $\mathbf{L}$  of  $\mathbf{P}\mathbf{A}\mathbf{P}^T$  to set up the storage scheme.

**Step 3 (Factorization)** Factor the permuted matrix  $\mathbf{P}\mathbf{A}\mathbf{P}^T$  into  $\mathbf{L}\mathbf{L}^T$ .

**Step 4** (*Triangular solution*) Solve  $\mathbf{L}\mathbf{y} = \mathbf{b}$  and  $\mathbf{L}^T\mathbf{z} = \mathbf{y}$ . Then set  $\mathbf{x} = \mathbf{P}^T\mathbf{z}$ .

Even with a prescribed storage method, there are many ways for finding orderings, determining the corresponding storage structure of  $\mathbf{L}$ , and performing the actual numerical computation. We shall refer to a sparse storage scheme and an associated ordering-allocation-factorization-solution combination collectively as a *solution method*.

The most commonly cited objectives for choosing a solution method are to a) reduce computer storage, b) reduce computer execution time or c) reduce some combination of storage and execution which reflects the way charges are assessed to the user of the computer system. Although there are other criteria which sometimes govern the choice of method, these are the main ones and serve to illustrate the complications involved in evaluating a strategy.

In order to be able to declare that one method is better than another with respect to one of the measures cited above, we must be able to evaluate precisely that measure for each method, and this evaluation is substantially more complicated than one would expect. We deal first with the computer storage criterion.

### 2.4.1 Storage Requirements

Computer storage used for sparse matrices typically consists of two parts, *primary storage* used to hold the numerical values, and *overhead storage*, used for pointers, subscripts and other information needed to record the structure of the matrix and to facilitate access to the numerical values. Since we must pay for computer storage regardless of how it is used, any evaluation of storage requirements for a solution method must include a description of the way the matrix or matrices involved are to be stored, so that the storage overhead can be included along with the primary storage in the storage requirement. The comparison of two different strategies with respect to the storage criterion may involve basically different data structures, having very different storage overheads. Thus, a method which is superior in terms of reducing primary storage may be inferior when overhead storage is included in the comparison. This point is illustrated pictorially in Figure 2.4.1.

As a simple example, consider the two orderings of a matrix problem in Figure 2.4.2, along with their corresponding factors  $\mathbf{L}$  and  $\tilde{\mathbf{L}}$ . The elements of the lower triangle of  $\mathbf{L}$  (excluding the diagonal) are stored row by row in

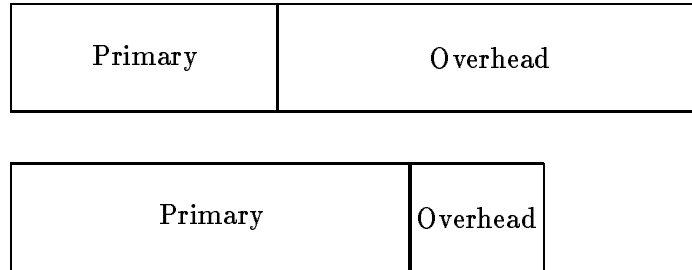


Figure 2.4.1: Primary/Overhead storage for two different methods.

---

a single array, with a parallel array holding their column subscripts. A third array indicates the position of each row, and a fourth array contains the diagonal elements of  $\mathbf{L}$ . The matrix  $\tilde{\mathbf{L}}$  is stored using the so-called envelope storage scheme, described in Chapter 4. Nonzeros in  $\mathbf{A}$  are denoted by  $\times$ , with  $*$  denoting fill-in components in  $\mathbf{L}$  or  $\tilde{\mathbf{L}}$ .

The examples in Figures 2.4.2 and 2.4.3 illustrate some important points about orderings and storage schemes. On the surface, ordering 1, corresponding to  $\mathbf{A}$  appears to be better than ordering 2 since it yields no fill-in at all, whereas the latter ordering causes two fill components. Moreover, the storage scheme used for  $\tilde{\mathbf{L}}$  appears to be inferior to that used for  $\mathbf{L}$ , since the latter actually ignores some sparsity, while all the sparsity in  $\mathbf{L}$  is exploited. However, because of differences in overhead, the second ordering/storage combination yields the lower *total* storage requirement. Of course the differences here are trivial, but the point is valid. As we increase the sophistication of our storage scheme, exploiting more and more zeros, the primary storage decreases, but the overhead usually increases. There is usually a point where it pays to ignore some zeros, because the overhead storage required to exploit them is more than the decrease in primary storage.

To summarize, the main points in this section are:

1. Storage schemes for sparse matrices involve two components: *primary* storage and *overhead* storage.
2. Comparisons of ordering strategies must take into account the storage scheme to be used, if the comparison is to be practically relevant.



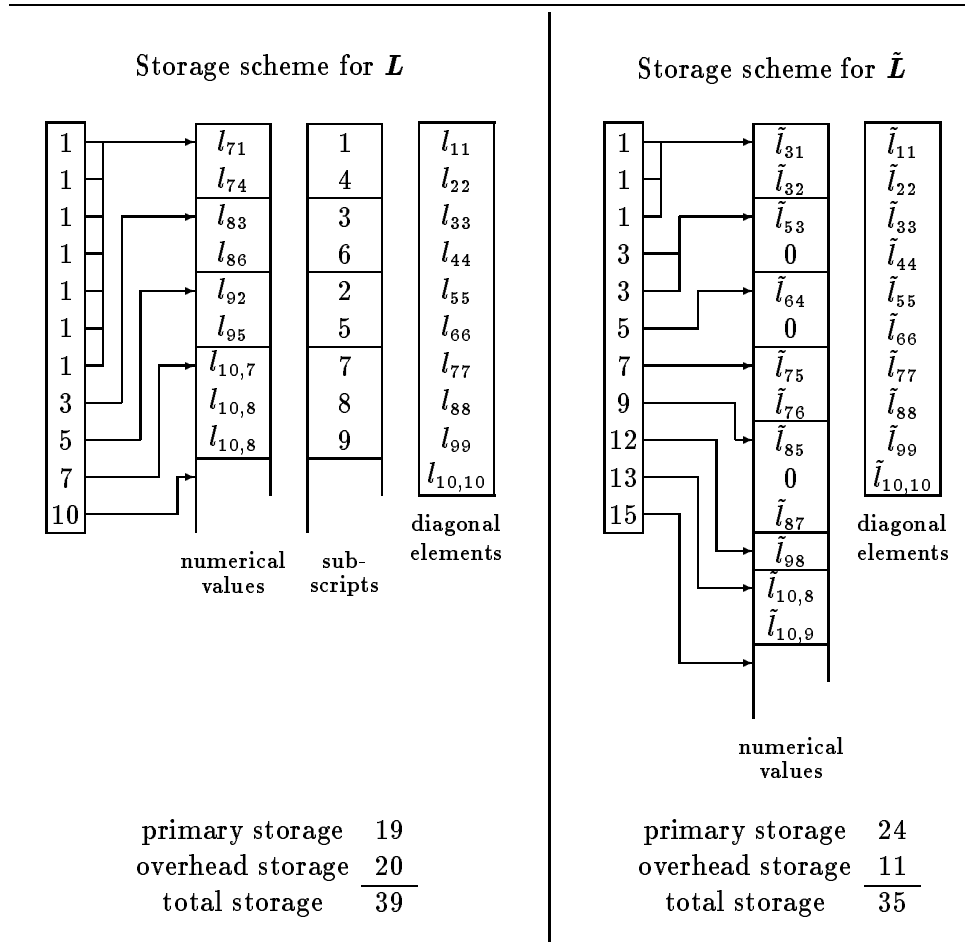


Figure 2.4.3: Storage schemes for the matrices  $L$  and  $\tilde{L}$  of Figure 2.4.2.

### 2.4.2 Execution Time

We now turn to computer execution time as a criterion. It is helpful in the discussion to consider the four steps in the entire computation process: ordering, allocation, factorization and solution.

As we shall see in Chapter 9, the execution times required to find different orderings can vary dramatically. But even after we have found the ordering, there is much left to do before we can actually begin the numerical computation. We must set up the appropriate storage scheme for  $L$ , and in order to do this we must determine its structure. This allocation step also varies in cost, depending on the ordering and storage scheme used. Finally, as we shall see through the numerous experiments supplied in Chapter 9, differences in storage schemes can lead to substantial differences in the arithmetic operations-per-second output of the factorization and triangular solution subroutines. Normally, the execution of a sparse matrix program will be (or should be) roughly *proportional* to the amount of arithmetic performed. However, differences in orderings and data structures can lead to large differences in the constant of proportionality. Thus, arithmetic operation counts may not be a very reliable measure for comparing solution methods, or at best must be used with care. The constant of proportionality is affected not only by the data structure used, but also by the computer architecture, compiler, and operating system.

In addition to the variation in the respective costs of executing each of the steps above, comparisons of different strategies often depend on the particular context in which a problem is being solved. If the given matrix problem is to be solved only once, a comparison of strategies should surely include the execution time required to produce the ordering and set up the storage scheme.

However, sometimes *many different* problems having the *same structure* must be solved, and it may be reasonable to ignore this initialization cost in comparing methods, since the bulk of the execution time involves the factorization and triangular solutions. In still other circumstances, many systems differing only in their right hand sides must be solved. In this case, it may be reasonable to compare strategies simply on the basis of their respective triangular solution times.

To summarize, the main points of the section are:

1. The overall solution of  $A\mathbf{x} = \mathbf{b}$  involves four basic steps. Their relative execution times in general vary substantially over different orderings and storage schemes.

2. Depending on the problem context, the execution times of some of the steps mentioned above may be practically irrelevant when comparing methods.

### Exercises

- 2.4.1) Suppose you have a choice of two methods (method 1 and method 2) for solving a sparse system of equations  $\mathbf{Ax} = \mathbf{b}$  and the criterion for the choice of method is execution time. The ordering and allocation steps for method 1 require a total of 20 seconds, while the corresponding time for method 2 is only 2 seconds. The factorization time for method 1 is 6 seconds and the solve time is .5 seconds, while for method 2 the corresponding execution times are 10 seconds and 1.5 seconds.
- What method would you choose if the system is to be solved only once?
  - What method would you choose if twelve systems  $\mathbf{Ax} = \mathbf{b}$ , having the *same* sparsity structure but different numerical values in  $\mathbf{A}$  and  $\mathbf{b}$  are to be solved?
  - What is your answer to b) if only the numerical values of the right side  $\mathbf{b}$  differ among the different systems?
- 2.4.2) Suppose for a given class of sparse positive definite matrix problems you have a choice between two orderings, “turtle” and “hare.” Your friend P.C.P. (Pure Complexity Pete, Esq.), shows that the turtle ordering yields triangular factors having  $\eta_t(n) \approx n^{3/2} + n - \sqrt{n}$  nonzeros, where  $n$  is the size of the problem. He also shows that the corresponding function for the hare ordering is  $\eta_h(n) \approx 7.75n \log_2(\sqrt{n} + 1) - 24n + 11.5\sqrt{n} \log_2(\sqrt{n} + 1) + 11\sqrt{n} + .75 \log_2(\sqrt{n} + 1)$ . Another friend, C.H.H. (Computer Hack Harold), implements linear equation solvers which use storage schemes appropriate for each ordering. Harold finds that for the hare implementation he needs one integer data item (a subscript) for each nonzero element of  $\mathbf{L}$ , together with 3 pointer arrays of length  $n$ . For the turtle implementation, the overhead storage is only  $n$  pointers.
- Suppose your choice of methods is based strictly on the total computer storage used to hold  $\mathbf{L}$ , and that integers and floating

point numbers each require one computer word. For what values of  $n$  would you use the hare implementation?

- b) What is your answer if Harold changes his programs so that integers are packed three to a computer word?





## Chapter 3

# Some Graph Theory Notation and Its Use in the Study of Sparse Symmetric Matrices

### 3.1 Introduction

In this chapter we introduce a few basic graph theory notions, and establish their correspondence to matrix concepts. Although rather few results from graph theory have found direct application to the analysis of sparse matrix computations, the notation and concepts are convenient and helpful in describing algorithms and identifying or characterizing matrix structure. Nevertheless, it is easy to become over-committed to the use of graph theory in such analyses, and the result is often to obscure some basically simple ideas in exchange for notational elegance. Thus, although we may sacrifice uniformity, where it is appropriate and aids the presentation, we will give definitions and results in both graph theory and matrix terms. In the same spirit, our intention is to introduce most graph theory notions only as they are required, rather than introducing them all in this section and then referring to them later.

### 3.2 Basic Terminology and Some Definitions

For our purposes, a *graph*  $\mathcal{G} = (X, E)$  consists of a finite set of *nodes* or *vertices* together with a set  $E$  of *edges*, which are unordered pairs of vertices. An *ordering* {*labelling*}  $\alpha$  of  $\mathcal{G} = (X, E)$  is simply a mapping of  $\{1, 2, \dots, n\}$  onto  $X$ , where  $n$  denotes the number of nodes of  $\mathcal{G}$ . Unless we specifically state otherwise, a graph will be unordered; the graph  $\mathcal{G}$  labelled by  $\alpha$  will be denoted by  $\mathcal{G}^\alpha = (X^\alpha, E)$ .

Since our objective in introducing graphs is to facilitate the study of sparse matrices, we now establish the relationship between graphs and matrices. Let  $\mathbf{A}$  be an  $n$  by  $n$  symmetric matrix. The *ordered graph of  $\mathbf{A}$* , denoted by  $\mathcal{G}^{\mathbf{A}} = (X^{\mathbf{A}}, E^{\mathbf{A}})$  is one for which the  $n$  vertices of  $\mathcal{G}^{\mathbf{A}}$  are numbered from 1 to  $n$ , and  $\{x_i, x_j\} \in E^{\mathbf{A}}$  if and only if  $a_{ij} = a_{ji} \neq 0, i \neq j$ . Here  $x_i$  denotes the node of  $X^{\mathbf{A}}$  with label  $i$ . Figure 3.2.1 illustrates the structure of a matrix and its labelled graph. We denote the  $i$ -th diagonal element of a matrix by circle  $i$  to emphasize its correspondence with node  $i$  of the corresponding graph. Off-diagonal nonzeros are depicted by  $\times$ .

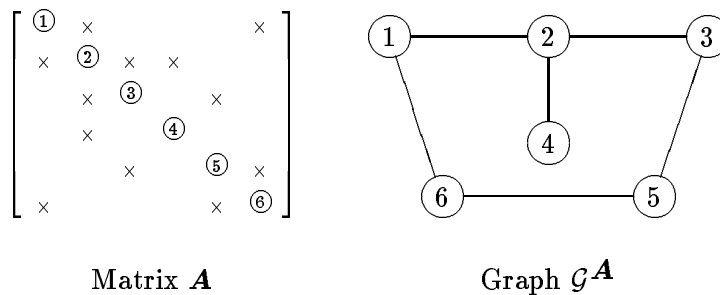


Figure 3.2.1: A matrix and its labelled graph, with  $\times$  denoting a nonzero entry of  $\mathbf{A}$ .

---

For any  $n$  by  $n$  *permutation matrix*  $\mathbf{P} \neq \mathbf{I}$ , the unlabelled graphs of  $\mathbf{A}$  and  $\mathbf{PAP}^T$  are the same but the associated labellings are different. Thus, the unlabelled graph of  $\mathbf{A}$  represents the structure of  $\mathbf{A}$  without suggesting any particular ordering. It represents the equivalence class of matrices  $\mathbf{PAP}^T$ , where  $\mathbf{P}$  is any  $n$  by  $n$  permutation matrix. Finding a “good” permutation for  $\mathbf{A}$  can be regarded as finding a good labelling for its graph.

Figure 3.2.2 illustrates these points.

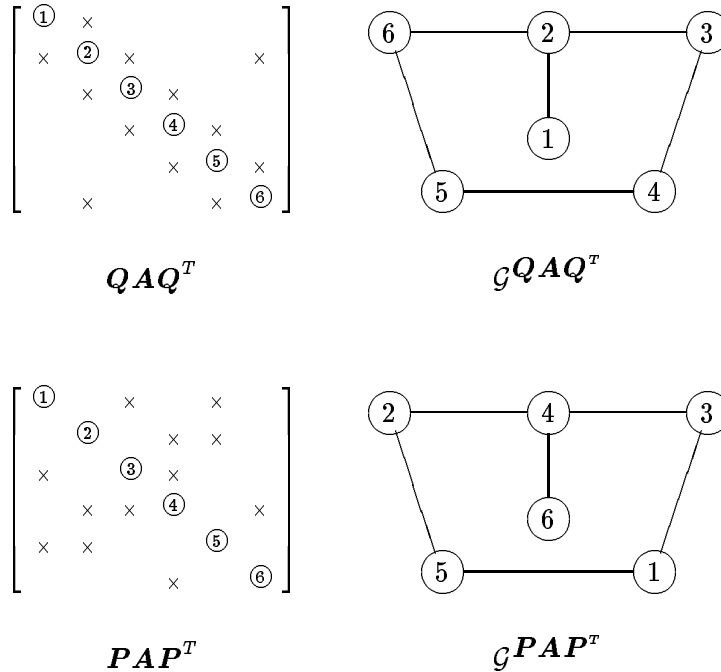


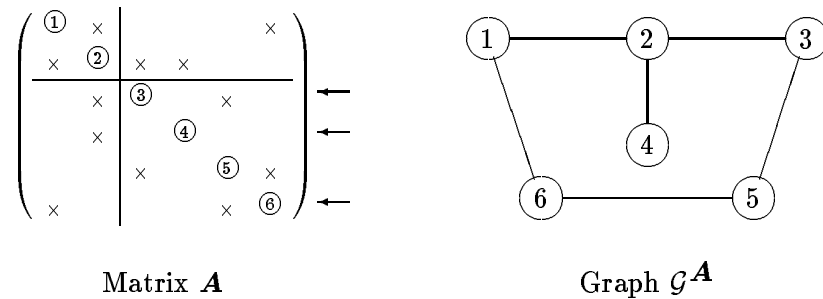
Figure 3.2.2: Graph of Figure 3.2.1 with different labellings, and the corresponding matrix structures. Here  $P$  and  $Q$  denote permutation matrices.

Some graph theory definitions involve *unlabelled* graphs. In order to interpret these definitions in matrix terms, we must have a matrix to refer to, and this immediately implies an ordering on the graph. Although this should not cause confusion, the reader should be careful not to attach any significance to the particular ordering chosen in our matrix examples and interpretations. When we refer to “the matrix corresponding to  $\mathcal{G}$ ,” we must either specify some ordering  $\alpha$  of  $\mathcal{G}$ , or understand that some arbitrary ordering is assumed.

Two nodes  $x$  and  $y$  in  $\mathcal{G}$  are *adjacent* if  $\{x, y\} \in E$ . For  $Y \subset X$ , the *adjacent set* of  $Y$ , denoted by  $Adj(Y)$ , is

$$Adj(Y) = \{x \in X - Y \mid \{x, y\} \in E, y \in Y\}. \tag{3.2.1}$$

Here and elsewhere in this book, the notation  $Y \subset X$  means that  $Y$  may be equal to  $X$ . When  $Y$  is intended to be a proper subset of  $X$ , we will explicitly indicate so. In words,  $Adj(Y)$  is simply the set of nodes in  $\mathcal{G}$  which are not in  $Y$  but are adjacent to at least one node in  $Y$ . Figure 3.2.3 illustrates the matrix interpretation of  $Adj(Y)$ . For convenience, the set  $Y$  has been labelled consecutively. When  $Y$  is the single node  $y$ , we will write  $Adj(y)$  rather than the formally correct  $Adj(\{y\})$ .



$$Y = \{x_1, x_2\}, \quad Adj(Y) = \{x_3, x_4, x_6\}$$

Figure 3.2.3: An illustration of the adjacent set of a set  $Y \subset X$ .

For  $Y \subset X$ , the *degree* of  $Y$ , denoted by  $Deg(Y)$ , is simply the number  $|Adj(Y)|$ , where  $|S|$  denotes the number of members in the set  $S$ . Again, when  $Y$  is a single node  $y$  we write  $Deg(y)$  rather than  $Deg(\{y\})$ . For example, in Figure 3.2.3,  $Deg(x_2) = 3$ .

A *subgraph*  $\mathcal{G}' = (X', E')$  of  $\mathcal{G}$  is a graph for which  $X' \subset X$  and  $E' \subset E$ . For  $Y \subset X$ , the *section graph*  $\mathcal{G}(Y)$  is the subgraph  $(Y, E(Y))$ , where

$$E(Y) = \{\{x, y\} \in E \mid x \in Y, y \in Y\}. \quad (3.2.2)$$

In matrix terms, the section graph  $\mathcal{G}(Y)$  is the graph of the matrix obtained by deleting all rows and columns from the matrix of  $\mathcal{G}$  except those corresponding to  $Y$ . This is illustrated in Figure 3.2.4.

A section graph is said to be a *clique* if the nodes in the subgraph are pairwise adjacent. In matrix terms, a clique corresponds to a full submatrix. For example  $\mathcal{G}(\{x_2, x_4\})$  is a clique.

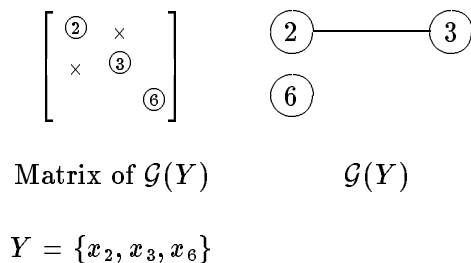


Figure 3.2.4: Example of a section graph  $\mathcal{G}(Y)$  and the matrix correspondence. The original graph  $\mathcal{G}$  is that of Figure 3.2.1.

---

The example in Figure 3.2.4 illustrates a concept we now explore, namely that of the *connectedness* of a graph. For distinct nodes  $x$  and  $y$  in  $\mathcal{G}$ , a *path* from  $x$  to  $y$  of length  $l \geq 1$  is an ordered set of  $l + 1$  distinct nodes  $(v_1, v_2, \dots, v_{l+1})$  such that  $v_{i+1} \in \text{Adj}(v_i)$ ,  $i = 1, 2, \dots, l$  with  $v_1 = x$  and  $v_{l+1} = y$ . A graph is *connected* if every pair of distinct nodes is joined by at least one path. Otherwise  $\mathcal{G}$  is *disconnected*, and consists of two or more *connected components*. In matrix terms, it should be clear that if  $\mathcal{G}$  is disconnected and consists of  $k$  connected components and each component is labelled consecutively, the corresponding matrix will be *block diagonal*, with each diagonal block corresponding to a connected component. The graph  $\mathcal{G}(Y)$  in Figure 3.2.4 is so ordered, and the corresponding matrix is block diagonal. Figure 3.2.5 shows a path in a graph and its interpretation in matrix terms.

Finally, the set  $Y \subset X$  is a *separator* of the connected graph  $\mathcal{G}$  if the section graph  $\mathcal{G}(X - Y)$  is disconnected. Thus, for example,  $Y = \{x_3, x_4, x_5\}$  is a separator of the graph of Figure 3.2.5, since  $\mathcal{G}(X - Y)$  has three components having node sets  $\{x_1\}$ ,  $\{x_2\}$ , and  $\{x_6, x_7\}$ .

### Exercises

3.2.1) A symmetric matrix  $A$  is said to be *reducible* if there exists a permutation matrix  $P$  such that

$$P^T A P = \begin{pmatrix} A_{11} & O \\ O & A_{22} \end{pmatrix}.$$

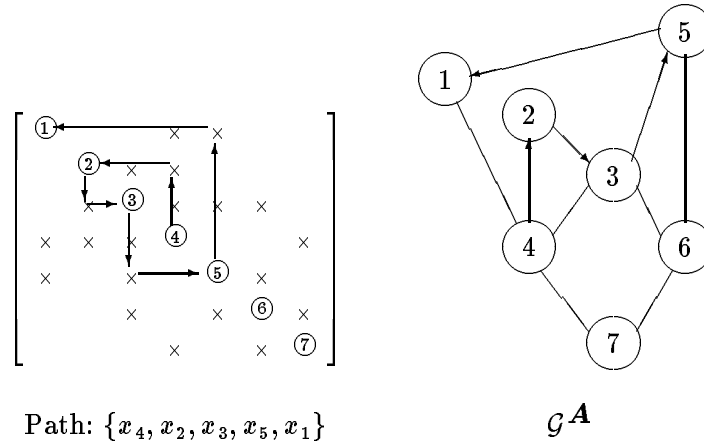


Figure 3.2.5: A path in a graph and the corresponding matrix interpretation.

Otherwise,  $\mathbf{A}$  is said to be *irreducible*. Show that a symmetric matrix  $\mathbf{A}$  is irreducible if and only if its associated graph  $\mathcal{G}^{\mathbf{A}}$  is connected.

- 3.2.2) Let  $\mathbf{A}$  be a symmetric matrix. Show that the matrix  $\mathbf{A}$  has the *propagation property* (see Exercise 2.3.3 on page 30 ) if and only if there exists the path  $(x_1, x_2, \dots, x_n)$  in the associated graph  $\mathcal{G}^{\mathbf{A}}$ .
- 3.2.3) Characterize the graphs associated with the matrices in Figure 3.2.6.

### 3.3 Computer Representation of Graphs

In general, the performances of graph algorithms are quite sensitive to the way the graphs are represented. For our purposes, the basic operation used is that of retrieving adjacency relations between nodes. So, we need a representation which provides the adjacency properties of the graph and which is economical in storage.

Let  $\mathcal{G} = (X, E)$  be a graph with  $n$  nodes. An *adjacency list* for  $x \in X$  is a list containing all the nodes in  $Adj(x)$ . An *adjacency structure* for  $\mathcal{G}$  is simply the set of adjacency lists for all  $x \in X$ . Such a structure can be implemented quite simply and economically by storing the adjacency lists sequentially in

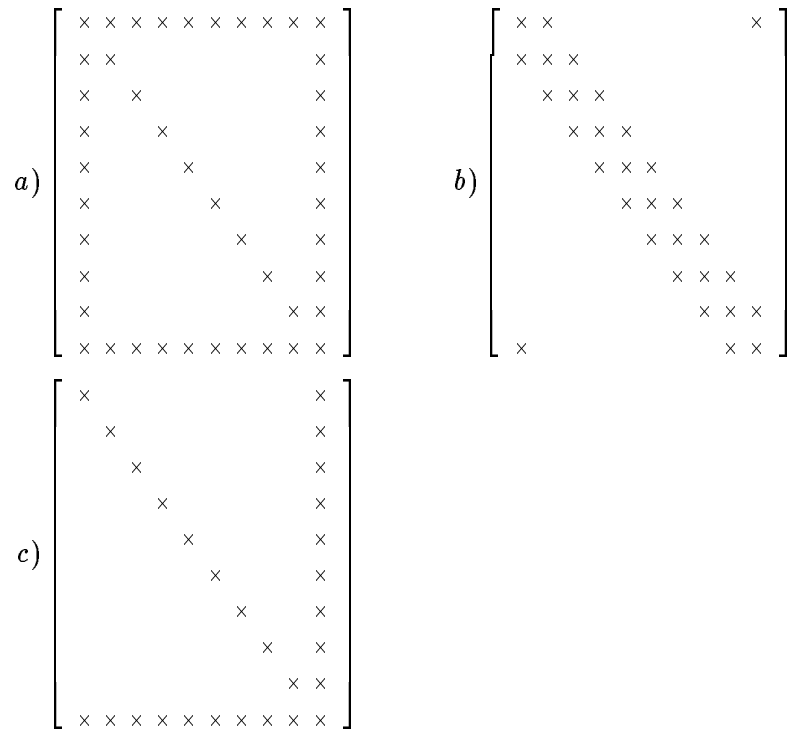


Figure 3.2.6: Examples of matrices with very different graphs.



a one-dimensional array `ADJNCY` along with an index array `XADJ` of length  $n + 1$  containing pointers to the beginning of each adjacency list in `ADJNCY`. An example is shown in Figure 3.3.1. It is often convenient for programming purposes to have an extra entry in `XADJ` such that `XADJ( $n + 1$ )` points to the next available storage location in `ADJNCY`, as shown in Figure 3.3.1. Clearly the total storage requirement for this storage scheme is then  $|X| + 2|E| + 1$ .

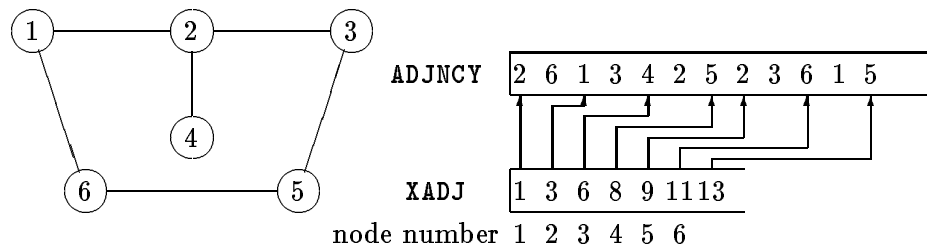


Figure 3.3.1: Example of an adjacency structure.

To examine all the neighbors of a node, the following program segment can be used.

```

NBRBEG = XADJ(NODE)
NBREND = XADJ(NODE + 1) - 1
IF (NBREND .LT. NBRBEG) GO TO 200

DO 100 I = NBRBEG, NBREND
  NABOR = ADJNCY(I)
100 CONTINUE
200

```

Although our implementations involving graphs use the storage scheme described above, several others are often used. A common storage scheme is a simple *connection table*, having  $n$  rows and  $m$  columns, where  $m = \max\{Deg(x) \mid x \in X\}$ . The adjacency list for node  $i$  is stored in row  $i$ . This storage scheme may be quite inefficient if a substantial number of the nodes have degrees less than  $m$ . An example of a connection table for the graph of Figure 3.3.1 is given in Figure 3.3.2.

The first two schemes described have a distinct disadvantage. Unless the degrees of the nodes are known *a priori*, it is difficult to construct the storage scheme when the graph is provided as a list of edges because we do not know

---

Node	Neighbours		
1	2	6	—
2	1	3	4
3	2	5	—
4	2	—	—
5	3	6	—
6	1	5	—

---

Figure 3.3.2: Connection table for the graph of Figure 3.3.1. Unused positions in the table are indicated by —.

---

the ultimate size of the adjacency lists. We can overcome this difficulty by introducing a *link field*. Figure 3.3.3 illustrates an example of such a scheme for the graph of Figure 3.3.1. The pointer  $\text{HEAD}(i)$  starts the adjacency list for node  $i$ , with  $\text{NBRS}$  containing a neighbor of node  $i$  and  $\text{LINK}$  containing the pointer to the location of the next neighbor of node  $i$ . For example, to retrieve the neighbors of node 5, we retrieve  $\text{HEAD}(5)$  which is 8. We then examine  $\text{NBRS}(8)$  which yields 3, one of the neighbors of node 5. We then retrieve  $\text{LINK}(8)$ , which is 2, implying that the next neighbor of node 5 is  $\text{NBRS}(2)$ , which is 6. Finally, we discover that  $\text{LINK}(2) = -5$ , which indicates the end of the adjacency list for node 5. (In general, a negative link of  $-i$  indicates the end of the adjacency list for node  $i$ .) The storage requirement for this graph representation is  $|X| + 4|E|$ , which is substantially more than the adjacency list scheme we use in our programs.

Provided there is enough space in the arrays  $\text{NBRS}$  and  $\text{LINK}$ , new edges can be added with ease. For example, to add the edge  $\{3, 6\}$  to the adjacency structure, we would adjust the adjacency list of node 3 by setting  $\text{LINK}(13)$  to 1,  $\text{NBRS}(13)$  to 6, and  $\text{HEAD}(3)$  to 13. The adjacency list of node 6 would be similarly changed by setting  $\text{LINK}(14)$  to 5,  $\text{NBRS}(14)$  to 3, and  $\text{HEAD}(6)$  to 14.

### 3.4 Some General Information on the Subroutines which Operate on Graphs

Numerous subroutines that operate on graphs are described in subsequent chapters. In all these subroutines, the graph  $\mathcal{G} = (X, E)$  is stored using the integer array pair  $(\text{XADJ}, \text{ADJNCY})$ , as described in Section 3.3. In addition,

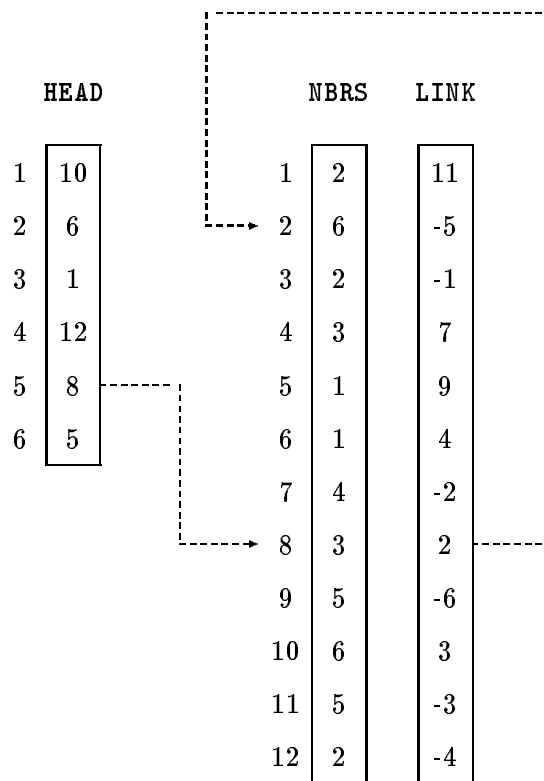


Figure 3.3.3: Adjacency linked lists for the graphs of Figure 3.3.1.

---

many of the subroutines share other common parameters. In order to avoid repeatedly describing these parameters in subsequent chapters, we discuss their role here, and refer to them later as required.

It should be clear that the mere fact that a graph is stored using the (XADJ, ADJNCY) array pair *implies* a particular labelling of the graph. This ordering will be referred to as the *original* numbering, and when we refer to “node  $i$ ,” it is this numbering we mean. When a subroutine finds a new ordering, the ordering is stored in an array PERM, where  $\text{PERM}(i) = k$  means the original node number  $k$  is the  $i$ -th node in the new ordering. We often use a related permutation vector INVP of length  $n$  (the inverse permutation) which satisfies  $\text{INVP}(\text{PERM}(i)) = i$ . That is,  $\text{INVP}(k)$  gives the position in PERM where the node originally numbered  $k$  resides.

It is necessary in many of our algorithms to perform operations only on certain section subgraphs of the graph  $\mathcal{G}$ . To implement these operations, many of our subroutines have an integer array MASK, of length  $n$ , which is used to prescribe such a subgraph. The subroutines only consider those nodes  $i$  for which  $\text{MASK}(i) \neq 0$ . Figure 3.4.1 contains an example illustrating the role of the integer array MASK.

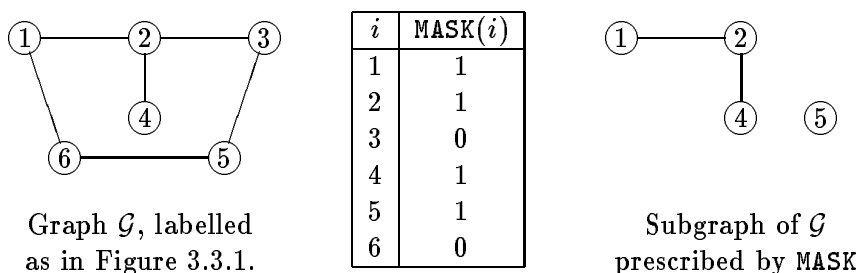


Figure 3.4.1: An example showing how the array MASK can be used to prescribe a subgraph of  $\mathcal{G}$ .

---

Finally, some of our subroutines have a single node number, usually called ROOT, as an argument, with  $\text{MASK}(\text{ROOT}) \neq 0$ . These subroutines typically operate on the *connected component* of the section subgraph prescribed by MASK which contains the node ROOT. That is, the combination of ROOT and MASK determine the connected subgraph of  $\mathcal{G}$  to be processed. We will often use the phrase “the component prescribed by ROOT and MASK” to refer to this connected subgraph. For example, the combination of  $\text{ROOT} = 2$  along with the array MASK and graph  $\mathcal{G}$  in Figure 3.4.1 would specify the graph shown

in Figure 3.4.2.

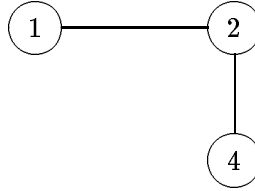


Figure 3.4.2: The subgraph of the graph in Figure 3.4.1 prescribed by  $\text{ROOT} = 2$  and  $\text{MASK}$ .

---

To summarize, some frequently used parameters in our subroutines, along with their contents are listed as follows:

$(\mathbf{XADJ}, \mathbf{ADJNCY})$  the array pair which stores the graph in its original ordering. The original labels of the nodes adjacent to node  $i$  are found in  $\mathbf{ADJNCY}(k)$ ,  $\mathbf{XADJ}(i) \leq k < \mathbf{XADJ}(i + 1)$ , with  $\mathbf{XADJ}(n + 1) = 2|E| + 1$ .

$\mathbf{PERM}$  an integer array of length  $n$  containing a new ordering.

$\mathbf{INVP}$  an integer array of length  $n$  containing the inverse of the permutation.

$\mathbf{MASK}$  an integer array of length  $n$  used to prescribe a section subgraph of  $\mathcal{G}$ . Subroutines ignore nodes for which  $\mathbf{MASK}(i) = 0$ .

$\mathbf{ROOT}$  a node number for which  $\mathbf{MASK}(\mathbf{ROOT}) \neq 0$ . The subroutine usually operates on the component of the subgraph specified by  $\mathbf{MASK}$  which contains the node  $\mathbf{ROOT}$ .

### Exercises

3.4.1) Suppose we represent a graph  $\mathcal{G} = (X, E)$  using a *lower adjacency structure*. That is, instead of storing the entire  $\text{Adj}(x)$  for each node  $x$ , we only store those nodes in  $\text{Adj}(x)$  with labels larger than that of  $x$ . For example, the graph of Figure 3.3.1 could be represented as shown in Figure 3.4.3, using the pair of arrays  $\mathbf{LADJ}$  and  $\mathbf{XLADJ}$ .

Design a subroutine that transforms a lower adjacency structure to the entire adjacency structure. Assume you have an array  $\mathbf{LADJ}$  of

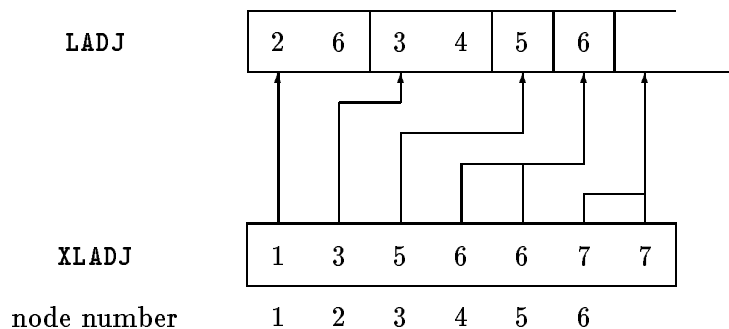


Figure 3.4.3: The lower adjacency structure of the graph of Figure 3.3.1.

- length  $2|E|$  containing the lower adjacency structure in its first  $|E|$  positions, and the array **XLADJ**. In addition, you have a temporary array of length  $|X|$ . When the subroutine completes execution, the arrays **XLADJ** and **LADJ** should contain the elements of **XADJ** and **ADJNCY** as described in Section 3.3.
- 3.4.2) Suppose a disconnected graph  $\mathcal{G} = (X, E)$  is stored in the pair of arrays **XADJ** and **ADJNCY**, as described in Section 3.3. Design a subroutine which accepts as input a node  $x \in X$ , and returns the nodes in the connected component of  $\mathcal{G}$  which contains  $x$ . Be sure to describe the parameters of the subroutine, and any auxiliary storage you require.
- 3.4.3) Suppose a (possibly disconnected) graph  $\mathcal{G} = (X, E)$  is stored in the pair of arrays **XADJ** and **ADJNCY** as described in Section 3.2. Suppose a subset  $Y \subset X$  is specified by an integer array **MASK** of length  $n$  as described in Section 3.4. Design and implement a subroutine which accepts as input the number  $n$ , the arrays **XADJ**, **ADJNCY**, and **MASK**, and returns the number of connected components in the section subgraph  $\mathcal{G}(Y)$ . You may need a temporary array of length  $n$  in order to make your implementation simple and easy to understand.
- 3.4.4) Suppose the graph of the matrix **A** is stored in the array pair (**XADJ**,

- ADJNCY), as described in Section 3.3, and suppose the arrays `PERM` and `INVP` correspond to the permutation matrices  $P$  and  $P^T$ , as described in Section 3.4. Write a subroutine to list the column subscript of the first nonzero element in each row of the matrix  $PAP^T$ . Your subroutine should also print the *number* of nonzeros to the left of the diagonal in each row of  $PAP^T$ .
- 3.4.5) Design a subroutine as described in Exercise 3.4.4 on page 53, with the additional feature that it only operates on the submatrix of  $PAP^T$  specified by the array `MASK`.
- 3.4.6) Suppose a graph is to be input as a sequence of edges (pairs of node numbers), and the size of the adjacency lists is not known beforehand. Design and implement a subroutine called `INSERT` which could be used to construct the linked data structure as exemplified by Figure 3.3.3. Be sure to describe the parameter list carefully, and consider how the arrays are to be initialized. You should not assume that  $|X|$  and  $|E|$  are known beforehand. Be sure to handle abnormal conditions, such as when the arrays are not large enough to accommodate all the edges, repeated input of the same edge, etc.
- 3.4.7) Suppose the graph of a matrix  $A$  is stored in the array pair (`XADJ`, `ADJNCY`), as described in Section 3.3. Design and implement a subroutine which accepts as input this array pair, along with two node numbers  $i$  and  $j$ , and determines whether there is a path joining them in the graph. If there is, then the subroutine returns the length of a shortest such path; otherwise it returns zero. Describe any temporary arrays you need.
- 3.4.8) Design and implement a subroutine as described in Exercise 3.4.7 on page 54, with the additional feature that it only operates on the subgraph specified by the array `MASK`.

## Chapter 4

# Band and Envelope Methods

### 4.1 Introduction

In this chapter we consider one of the simplest methods for solving sparse systems, the band schemes and the closely related envelope or profile methods.

Loosely speaking, the objective is to order the matrix so that the nonzeros in  $\mathbf{PAP}^T$  are clustered “near” the main diagonal. Since this property is retained in the corresponding Cholesky factor  $\mathbf{L}$ , such orderings appear to be attractive in reducing fill, and are widely used in practice (Cuthill [9], Felippa [15], Melosh and Bamford [41]).

Although these orderings are often far from optimal in the least-arithmetic or least-fill senses, they are often an attractive practical compromise. In general the programs and data structures needed to exploit the sparsity that these orderings provide are relatively simple; that is, the storage and computational overhead involved in using the orderings tends to be small compared to more sophisticated orderings. (Recall our remarks in Section 2.4.) The orderings themselves also tend to be much cheaper to obtain than more (theoretically) efficient orderings. For small problems, and even moderate size problems which are to be solved only a few times, the methods described in this chapter should be seriously considered.

### 4.2 The Band Method

Let  $\mathbf{A}$  be an  $n$  by  $n$  symmetric positive definite matrix, with entries  $a_{ij}$ . For the  $i$ -th row of  $\mathbf{A}$ ,  $i = 1, 2, \dots, n$ , let

$$f_i(\mathbf{A}) = \min\{j \mid a_{ij} \neq 0\},$$



and

$$\beta_i(\mathbf{A}) = i - f_i(\mathbf{A}).$$

The number  $f_i(\mathbf{A})$  is simply the column subscript of the first nonzero component in row  $i$  of  $\mathbf{A}$ . Since the diagonal entries  $a_{ii}$  are positive, we have

$$f_i(\mathbf{A}) \leq i \text{ and } \beta_i(\mathbf{A}) \geq 0.$$

Following Cuthill and McKee, we define the *bandwidth* of  $\mathbf{A}$  by <sup>1</sup>

$$\begin{aligned} \beta(\mathbf{A}) &= \max\{\beta_i(\mathbf{A}) \mid 1 \leq i \leq n\} \\ &= \max\{|i - j| \mid a_{ij} \neq 0\}. \end{aligned}$$

The number  $\beta_i(\mathbf{A})$  is called the  *$i$ -th bandwidth* of  $\mathbf{A}$ . We define the *band* of  $\mathbf{A}$  as

$$\text{Band}(\mathbf{A}) = \{\{i, j\} \mid 0 < i - j \leq \beta(\mathbf{A})\}, \quad (4.2.1)$$

which is the region within  $\beta(\mathbf{A})$  locations of the main diagonal. Unordered pairs  $\{i, j\}$  are used in (4.2.1) instead of ordered pairs because  $\mathbf{A}$  is symmetric. The matrix example in Figure 4.2.1 has a bandwidth of 3. Matrices with a bandwidth of one are called tridiagonal matrices.

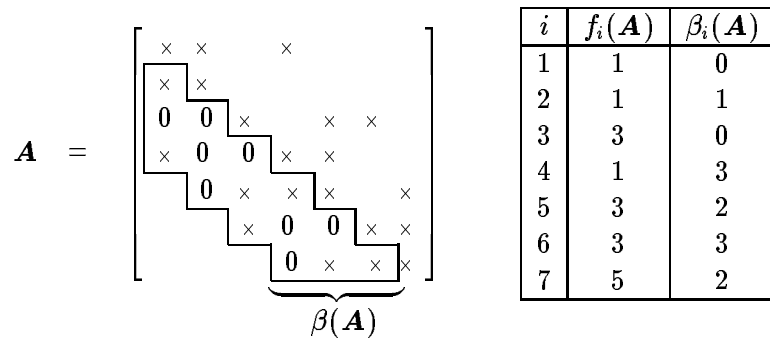


Figure 4.2.1: Example showing  $f_i(\mathbf{A})$  and  $\beta_i(\mathbf{A})$ .

Implicit in the use of the band method is that zeros outside  $\text{Band}(\mathbf{A})$  are ignored; zeros inside the band are usually stored, although often exploited

<sup>1</sup>Other authors define the bandwidth of  $\mathbf{A}$  to be  $2\beta(\mathbf{A}) + 1$ .

as far as the actual computation is concerned. This exploitation of zeros is possible in the direct solution because

$$\text{Band}(\mathbf{A}) = \text{Band}(\mathbf{L} + \mathbf{L}^T),$$

a relation that will be proved in Section 4.3 when the envelope method is considered.

A common method for storing a symmetric band matrix  $\mathbf{A}$  is the so-called *diagonal storage scheme* (Martin [40]). The  $\beta(\mathbf{A})$  sub-diagonals of the lower triangle of  $\mathbf{A}$  which comprise  $\text{Band}(\mathbf{A})$  and the main diagonal of  $\mathbf{A}$  are stored as the columns of an  $n$  by  $(\beta(\mathbf{A}) + 1)$  rectangular array, as shown in Figure 4.2.2. This storage scheme is very simple, and is quite efficient as long as  $\beta_i(\mathbf{A})$  does not vary too much with  $i$ .

---

$$\text{Matrix } \mathbf{A} = \begin{pmatrix} a_{11} & & & & & & & & \\ a_{21} & a_{22} & & & & & & & \\ 0 & 0 & a_{33} & & & & & & \\ a_{41} & 0 & 0 & a_{44} & & & & & \\ & 0 & a_{53} & a_{54} & a_{55} & & & & \\ & & a_{63} & 0 & 0 & a_{66} & & & \\ & & & 0 & a_{75} & a_{76} & a_{77} & & \end{pmatrix}$$

$$\text{Storage Array} \begin{bmatrix} - & - & - & a_{11} \\ - & - & a_{21} & a_{22} \\ - & 0 & 0 & a_{33} \\ a_{41} & 0 & 0 & a_{44} \\ 0 & a_{53} & a_{54} & a_{55} \\ a_{63} & 0 & 0 & a_{66} \\ 0 & a_{75} & a_{76} & a_{77} \end{bmatrix}$$

Figure 4.2.2: The diagonal storage scheme.

---

**Theorem 4.2.1** The number of operations required to factor the matrix  $\mathbf{A}$  having bandwidth  $\beta$ , assuming  $\text{Band}(\mathbf{L} + \mathbf{L}^T)$  is full, is

$$\frac{1}{2}\beta(\beta + 3)n - \frac{\beta^3}{3} - \beta^2 - \frac{2}{3}\beta.$$

**Proof:** The result follows from Theorem 2.2.2 and the observation that

$$\eta(\mathbf{L}_{*i}) = \begin{cases} \beta + 1 & \text{for } 1 \leq i \leq n - \beta \\ n - i + 1 & \text{for } n - \beta < i \leq n \end{cases}$$

□

**Theorem 4.2.2** *Let  $\mathbf{A}$  be as in Theorem 4.2.1. Then the number of operations required to solve the matrix problem  $\mathbf{Ax} = \mathbf{b}$ , given the Cholesky factor  $\mathbf{L}$  of  $\mathbf{A}$ , is*

$$2(\beta + 1)n - \beta(\beta + 1).$$

**Proof:** The result follows from Theorem 2.2.2 and the definition of  $\eta(\mathbf{L}_{*i})$  given in the proof of Theorem 4.2.1. □

As mentioned above, the attraction of this approach is its simplicity. However, it has some potentially serious weaknesses. First, if  $\beta_i(\mathbf{A})$  varies widely with  $i$ , the diagonal storage scheme illustrated in Figure 4.2.2 will be inefficient. Moreover, as we shall see later, there are some very sparse problems which can be solved very efficiently, but which cannot be ordered to have a small bandwidth (see Figure 4.3.3). Thus, there are problems for which band methods are simply inappropriate. Perhaps the most persuasive reason for not being very enthusiastic about band schemes is that the envelope schemes discussed in the next section share all the advantages of simplicity enjoyed by band schemes, with very few of the disadvantages.

### Exercises

- 4.2.1) Suppose  $\mathbf{A}$  is an  $n$  by  $n$  symmetric positive definite matrix with bandwidth  $\beta$ . You have two sets of numerical subroutines for solving  $\mathbf{Ax} = \mathbf{b}$ . One set stores  $\mathbf{A}$  (over-written by  $\mathbf{L}$  during the factorization) as a full lower triangular matrix by storing the rows of the lower triangular part row by row in a one dimensional array, in the sequence  $a_{11}, a_{21}, a_{22}, a_{31}, \dots, a_{n,n-1}, a_{n,n}$ . The other set of subroutines stores  $\mathbf{A}$  (again over-written by  $\mathbf{L}$  during the factorization) using the diagonal storage scheme described in this section. For a given  $\beta$  and  $n$ , which scheme would you use if you were trying to minimize storage requirements?
- 4.2.2) Consider the star graph of  $n$  nodes, as shown in Figure 4.3.3(a). Prove that *any* ordering of this graph yields a bandwidth of at least  $\lceil (n-1)/2 \rceil$ .

### 4.3 The Envelope Method

#### 4.3.1 Matrix Formulation

A slightly more sophisticated scheme for exploiting sparsity is the so-called *envelope* or *profile* method, which simply takes advantage of the variation in  $\beta_i(\mathbf{A})$  with  $i$ . The *envelope* of  $\mathbf{A}$ , denoted by  $Env(\mathbf{A})$ , is defined by

$$Env(\mathbf{A}) = \{\{i, j\} \mid 0 < i - j \leq \beta_i(\mathbf{A})\}.$$

In terms of the column subscripts  $f_i(\mathbf{A})$ , we have

$$Env(\mathbf{A}) = \{\{i, j\} \mid f_i(\mathbf{A}) \leq j < i\}.$$

The quantity  $|Env(\mathbf{A})|$  is called the *profile* or *envelope size* of  $\mathbf{A}$ , and is given by

$$|Env(\mathbf{A})| = \sum_{i=1}^n \beta_i(\mathbf{A}).$$

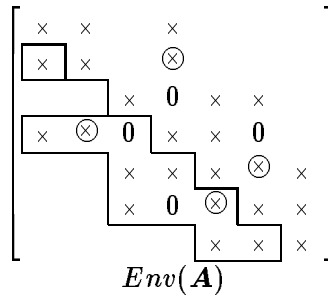


Figure 4.3.1: Illustration of the envelope of  $\mathbf{A}$ . Circled elements denote fill elements of  $\mathbf{L}$ .

#### Lemma 4.3.1

$$Env(\mathbf{A}) = Env(\mathbf{L} + \mathbf{L}^T).$$

**Proof:** We prove the lemma by induction on the dimension  $n$ . Assume that the result holds for  $n - 1$  by  $n - 1$  matrices. Let  $\mathbf{A}$  be an  $n$  by  $n$  symmetric matrix partitioned as

$$\mathbf{A} = \begin{pmatrix} \mathbf{M} & \mathbf{u} \\ \mathbf{u}^T & s \end{pmatrix},$$

where  $s$  is a scalar,  $\mathbf{u}$  is a vector of length  $n - 1$ , and  $\mathbf{M}$  is an  $n - 1$  by  $n - 1$  nonsingular matrix factored as  $\mathbf{L}\mathbf{M}\mathbf{L}^T$ . By the inductive assumption, we have  $\text{Env}(\mathbf{M}) = \text{Env}(\mathbf{L}\mathbf{M} + \mathbf{L}^T\mathbf{M})$ . If  $\mathbf{L}\mathbf{L}^T$  is the symmetric factorization of  $\mathbf{A}$ , the triangular factor  $\mathbf{L}$  can be partitioned as

$$\mathbf{L} = \begin{pmatrix} \mathbf{L}\mathbf{M} & \mathbf{0} \\ \mathbf{w}^T & t \end{pmatrix},$$

where  $t$  is a scalar, and  $\mathbf{w}$  is a vector of length  $n - 1$ . It is then sufficient to show that  $f_n(\mathbf{A}) = f_n(\mathbf{L} + \mathbf{L}^T)$ .

From (2.2.4), the vectors  $\mathbf{u}$  and  $\mathbf{w}$  are related by

$$\mathbf{L}\mathbf{M}\mathbf{w} = \mathbf{u}.$$

But  $u_i = 0$  for  $1 \leq i < f_n(\mathbf{A})$  and the entry  $u_{f_n(\mathbf{A})}$  is nonzero. By Lemmas 2.3.3 and 2.3.4, we have  $w_i = 0$  for  $1 \leq i < f_n(\mathbf{A})$  and  $w_{f_n(\mathbf{A})} \neq 0$ . Hence  $f_n(\mathbf{A}) = f_n(\mathbf{L} + \mathbf{L}^T)$ , so that

$$\text{Env}(\mathbf{A}) = \text{Env}(\mathbf{L} + \mathbf{L}^T).$$

□

### Theorem 4.3.2

$$\text{Env}(\mathbf{A}) \subset \text{Band}(\mathbf{A}).$$

**Proof:** It follows from the definitions of *Band* and *Env*. □

Lemma 4.3.1 justifies the exploitation of zeros outside the envelope or the band region. Assuming that only those zeros outside  $\text{Env}(\mathbf{A})$  are exploited, we now determine the arithmetic cost in performing the direct solution. In order to compute operation counts, it is helpful to introduce the notion of *frontwidth*. For a matrix  $\mathbf{A}$ , the  $i$ -th *frontwidth* of  $\mathbf{A}$  is defined to be

$$\omega_i(\mathbf{A}) = |\{k \mid k > i \text{ and } a_{kl} \neq 0 \text{ for some } l \leq i\}|.$$

Note that  $\omega_i(\mathbf{A})$  is simply the number of “active” rows at the  $i$ -th step in the factorization; that is, the number of rows of the envelope of  $\mathbf{A}$ , which intersect column  $i$ . The quantity

$$\omega(\mathbf{A}) = \max\{\omega_i(\mathbf{A}) \mid 1 \leq i \leq n\}$$

is usually referred to as the *frontwidth* or *wave front* of  $\mathbf{A}$  (Irons [31], Melosh [41]). Figure 4.3.2 illustrates these definitions.

The relevance of the notion of frontwidth in the analysis of the envelope method is illustrated by the following.

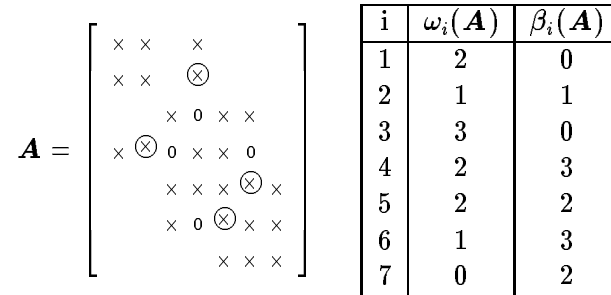


Figure 4.3.2: Illustration of the  $i$ -th bandwidth and frontwidth.

---

**Lemma 4.3.3**

$$|Env(\mathbf{A})| = \sum_{i=1}^n \omega_i(\mathbf{A}).$$

**Theorem 4.3.4** *If only those zeros outside the envelope are exploited, the number of operations required to factor  $\mathbf{A}$  into  $\mathbf{LL}^T$  is given by*

$$\frac{1}{2} \sum_{i=1}^n \omega_i(\mathbf{A})(\omega_i(\mathbf{A}) + 3),$$

*and the number of operations required to solve the system  $\mathbf{Ax} = \mathbf{b}$ , given the factorization  $\mathbf{LL}^T$  is*

$$2 \sum_{i=1}^n (\omega_i(\mathbf{A}) + 1).$$

**Proof:** If we treat the envelope of  $\mathbf{A}$  as full, the number of nonzeros in  $L_{*i}$  is simply  $\omega_i(\mathbf{A}) + 1$ . The result then follows from Theorem 2.2.2 and Lemma 2.3.1.  $\square$

Although profile schemes appear to represent a rather minor increase in sophistication over band schemes, they can sometimes lead to quite spectacular improvements. To see this, consider the example in Figure 4.3.3 showing two orderings of the same matrix.

It is not hard to verify that the number of operations required to factor the minimum profile ordered matrix, and the number of nonzeros in the corresponding factor are both  $O(n)$ , as is the bandwidth. On the other

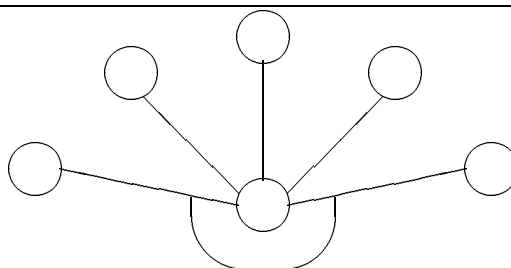
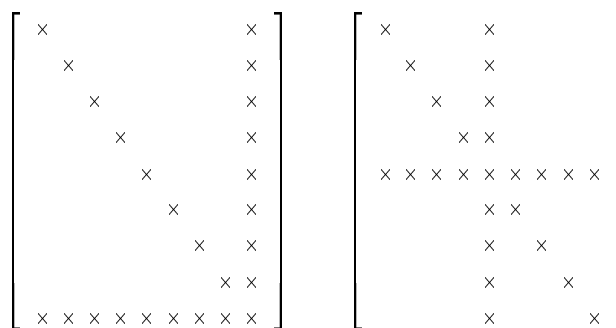


Figure 4.3.3: Star graph of  $n$  nodes.

---



Ordering corresponding  
to numbering the  
center node last

Minimum bandwidth  
ordering

Figure 4.3.4: Minimum profile ordering and minimum band ordering for the star graph on  $n$  nodes with  $n = 9$ .

---

hand, the minimum bandwidth ordering yields an  $O(n^3)$  operation count and an  $L$  having  $O(n^2)$  nonzeros.

Although this example is contrived, numerous practical examples exist where envelope schemes are much more efficient than band schemes. For some examples, see Liu and Sherman [38].

### 4.3.2 Graph Interpretation

For an  $n$  by  $n$  symmetric matrix  $\mathbf{A}$ , let its associated undirected graph be

$$\mathcal{G}^{\mathbf{A}} = (X^{\mathbf{A}}, E^{\mathbf{A}}),$$

where the node set is labelled as implied by  $\mathbf{A}$ :

$$X^{\mathbf{A}} = \{x_1, \dots, x_n\}.$$

To provide insight into the combinatorial nature of the envelope method, it is important to give graph theoretic interpretation to the matrix definitions introduced in the previous subsection.

**Theorem 4.3.5** *For  $i < j$ ,  $\{i, j\} \in Env(\mathbf{A})$  if and only if  $x_j \in Adj(\{x_1, \dots, x_i\})$ .*

**Proof:** If  $x_j \in Adj(\{x_1, \dots, x_i\})$ , then  $a_{jk} \neq 0$  for some  $k \leq i$  so that  $f_j(\mathbf{A}) \leq i$  and  $\{i, j\} \in Env(\mathbf{A})$ .

Conversely, if  $f_j(\mathbf{A}) \leq i < j$ , this means  $x_j \in Adj(x_{f_j(\mathbf{A})})$  which implies  $x_j \in Adj(\{x_1, \dots, x_i\})$ .  $\square$

**Corollary 4.3.6** *For  $i = 1, \dots, n$ ,  $\omega_i(\mathbf{A}) = |Adj(\{x_1, \dots, x_i\})|$ .*

**Proof:** From the definition of  $\omega_i(\mathbf{A})$ , we have

$$\omega_i(\mathbf{A}) = |\{j > i \mid \{i, j\} \in Env(\mathbf{A})\}|,$$

so that the result follows from Theorem 4.3.5.  $\square$

Consider the matrix example and its associated labelled graph in Figure 4.3.5. The respective adjacent sets are

$$\begin{aligned} Adj(x_1) &= \{x_2, x_4\}, \\ Adj(\{x_1, x_2\}) &= \{x_4\}, \\ Adj(\{x_1, x_2, x_3\}) &= \{x_4, x_5, x_6\}, \\ Adj(\{x_1, x_2, x_3, x_4\}) &= \{x_5, x_6\}, \\ Adj(\{x_1, \dots, x_5\}) &= \{x_6, x_7\}, \\ Adj(\{x_1, \dots, x_6\}) &= \{x_7\}, \\ Adj(\{x_1, \dots, x_7\}) &= \phi. \end{aligned}$$



Compare them with the row subscripts of the envelope entries in each column.

---

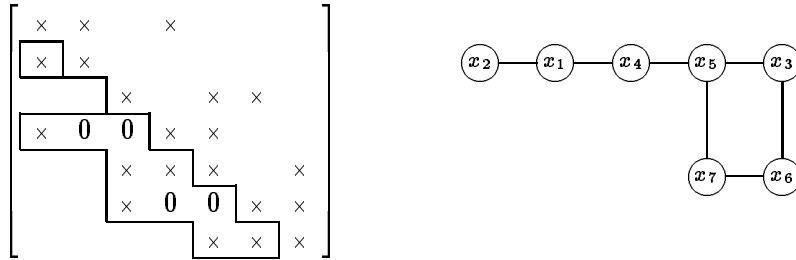


Figure 4.3.5: A matrix and its associated labelled graph.

---

The set  $Adj(\{x_1, \dots, x_i\})$  shall be referred to as the  $i$ -th front of the labelled graph, and its size the  $i$ -th frontwidth (as before).

### Exercises

4.3.1) Prove that

$$\frac{1}{2} \sum_{i=1}^n \omega_i(\mathbf{A})(\omega_i(\mathbf{A}) + 3) \leq \frac{1}{2} \sum_{i=1}^n \beta_i(\mathbf{A})(\beta_i(\mathbf{A}) + 3).$$

4.3.2) A symmetric matrix  $\mathbf{A}$  is said to have the *monotone profile property* if  $f_j(\mathbf{A}) \leq f_i(\mathbf{A})$  for  $j \leq i$ . Show that for monotone profile matrices,

$$\frac{1}{2} \sum_{i=1}^n \omega_i(\mathbf{A})(\omega_i(\mathbf{A}) + 3) = \frac{1}{2} \sum_{i=1}^n \beta_i(\mathbf{A})(\beta_i(\mathbf{A}) + 3).$$

4.3.3) Prove that the following conditions are equivalent.

- a) for  $1 \leq i \leq n$ , the section graphs  $\mathcal{G}(\{x_1, \dots, x_i\})$  are connected
- b) for  $2 \leq i \leq n$ ,  $f_i(\mathbf{A}) < i$ .

4.3.4) (*Full Envelope*) Prove that the matrix  $\mathbf{L} + \mathbf{L}^T$  has a full envelope if  $f_i(\mathbf{A}) < i$  for  $2 \leq i \leq n$ . Show that  $\mathbf{L} + \mathbf{L}^T$  has a full envelope for monotone profile matrix  $\mathbf{A}$ .

- 4.3.5) Let  $\mathbf{L}$  be an  $n$  by  $n$  lower triangular matrix with bandwidth  $\beta \ll n$ , and let  $\mathbf{V}$  be an  $n$  by  $p$  (pseudo) lower triangular matrix as defined in Exercise 2.3.8 on page 30. Approximately how many operations are required to compute  $\mathbf{L}^{-1}\mathbf{V}$ ?
- 4.3.6) Let  $\{x_1, \dots, x_n\}$  be the nodes in the graph  $\mathcal{G}^{\mathbf{A}}$  associated with a symmetric matrix  $\mathbf{A}$ . Show that the following conditions are equivalent.
- $Env(\mathbf{A})$  is full,
  - $Adj(\{x_1, \dots, x_i\}) \subset Adj(x_i)$  for  $1 \leq i \leq n$ ,
  - $Adj(\{x_1, \dots, x_i\}) \cup \{x_i\}$  is a clique for  $1 \leq i \leq n$ .
- 4.3.7) Show that if the graph  $\mathcal{G}^{\mathbf{A}}$  is connected, then  $\omega_i(\mathbf{A}) \neq 0$  for  $1 \leq i \leq n - 1$ .

## 4.4 Envelope Orderings

### 4.4.1 The Reverse Cuthill-McKee Algorithm

Perhaps the most widely used profile reduction ordering algorithm is a variant of the Cuthill-McKee ordering. In 1969, Cuthill and McKee [10] published their algorithm which was primarily designed to reduce the bandwidth of a sparse symmetric matrix.

The scheme makes use of the following observation. Let  $y$  be a labelled node, and  $z$  an unlabelled neighbor of  $y$ . To minimize the bandwidth of the row associated with  $z$ , it is apparent that the node  $z$  should be ordered as soon as possible after  $y$ . Figure 4.4.1 illustrates this point.

The Cuthill-McKee scheme may be regarded as a method that reduces the bandwidth of a matrix via a local minimization of the  $\beta_i$ 's. This suggests that the scheme can be used as a method to reduce the profile  $\sum \beta_i$  of a matrix. George [17], in his study of the profile methods, discovered that the ordering obtained by reversing the Cuthill-McKee ordering often turns out to be much superior to the original ordering in terms of profile reduction, although the bandwidth remains unchanged. He called this the *reverse Cuthill-McKee* ordering (RCM). It has since been proved that the reverse scheme is never inferior, as far as envelope storage and envelope operation counts are concerned (Liu and Sherman [38]).

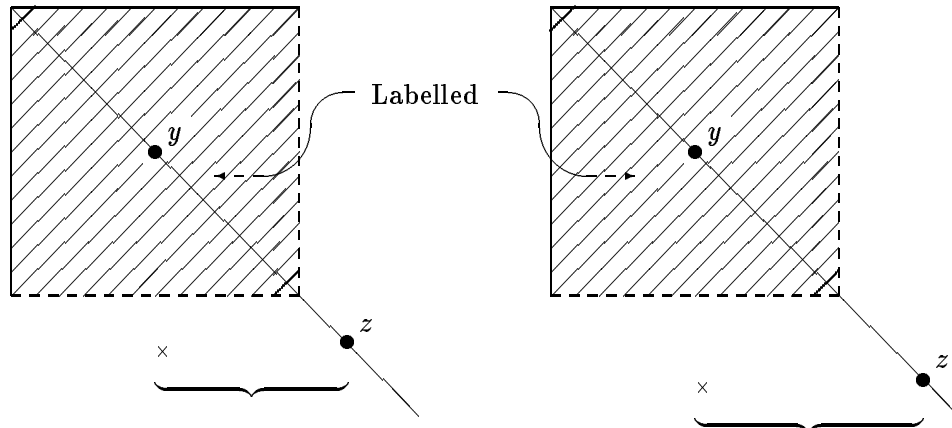


Figure 4.4.1: Effect on bandwidth of numbering node  $z$  after node  $y$ , when they are connected.

We describe the RCM algorithm for a connected graph as follows. (The task of determining the starting node in Step 1 is considered in the next section.)

**Step 1** Determine a starting node  $r$  and assign  $x_1 \leftarrow r$ .

**Step 2** (*Main loop*) For  $i = 1, \dots, n$ , find all the unnumbered neighbors of the node  $x_i$  and number them in increasing order of degree.

**Step 3** (*Reverse ordering*) The reverse Cuthill-McKee ordering is given by  $y_1, y_2, \dots, y_n$  where  $y_i = x_{n-i+1}$  for  $i = 1, \dots, n$ .

In the case when the graph  $\mathcal{G}^{\mathbf{A}}$  is disconnected, we can apply the above algorithm to each connected component of the graph. For a given starting node, the algorithm is relatively simple and we go through it in the following example.

Suppose the node “ $g$ ” in Figure 4.4.2 is picked as the starting node, that is  $x_1 = g$ . Figure 4.4.3 illustrates how nodes are numbered in Step 2 of the algorithm. The resulting reverse Cuthill-McKee ordering is given in Figure 4.4.4, and the envelope size is 22.

The effectiveness of the ordering algorithm depends quite crucially on the choice of the starting node. In the example, if we pick node “ $a$ ” instead

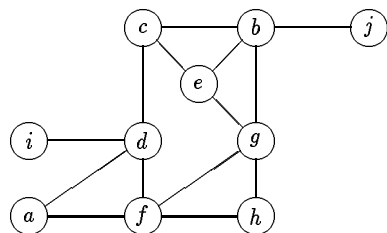


Figure 4.4.2: Graph to which the RCM algorithm is to be applied.

---



---

$i$	Node $x_i$	Unnumbered neighbors in increasing ordering of degree
1	$g$	$h, e, b, f$
2	$h$	—
3	$e$	$c$
4	$b$	$j$
5	$f$	$a, d$
6	$c$	—
7	$j$	—
8	$a$	—
9	$d$	$i$
10	$i$	—

---

Figure 4.4.3: Table showing numbering in Step 2 of the RCM algorithm.

---

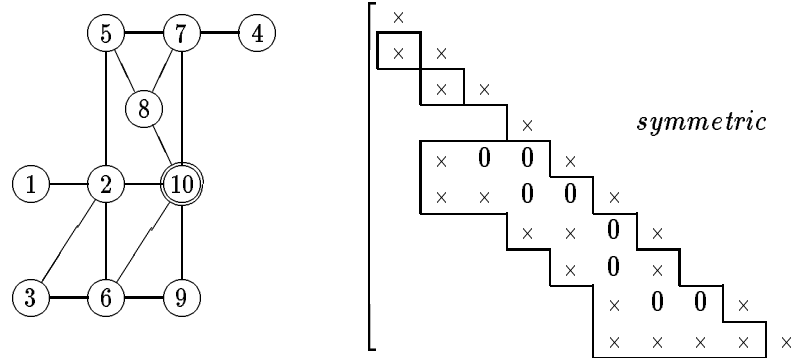


Figure 4.4.4: The final ordering and corresponding matrix structure.

as the starting node, we get a smaller profile of 18. In Section 4.4.3, we present an algorithm which experience has shown to provide a good starting node for the Cuthill-McKee algorithm.

We now establish a rough complexity bound for the execution time of the RCM algorithm, assuming that a starting node is provided. The underlying assumption here is that the execution time of the sorting algorithm used is proportional to the number of operations performed, where an operation might be a comparison, or a retrieval of a data item from the adjacency structure used to store the graph.

**Theorem 4.4.1** *If linear insertion is used for sorting, the time complexity of the RCM algorithm is bounded by  $O(m|E|)$ , where  $m$  is the maximum degree of any node.*

**Proof:** The major cost is obviously due to Step 2 of the algorithm, since Step 3 can be done in  $O(n)$  time. For some constant  $c$ , sorting  $t$  elements using linear insertion requires  $ct^2$  operations [1]. Thus, the overall time spent in sorting is less than

$$c \sum_{x \in X} |Deg(x)|^2 \leq cm \sum_{x \in X} |Deg(x)| = 2cm|E|.$$

For each index in Step 2, we have to examine the neighbors of node  $i$ , in order to retrieve the unnumbered ones for sorting by degree. This sweep through the adjacency structure requires  $2|E|$  operations. The computation

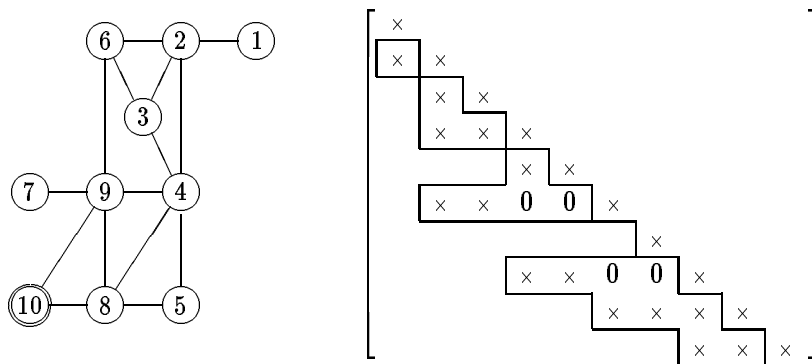


Figure 4.4.5: The RCM ordering of the example of Figure 4.4.2, using a different starting node.

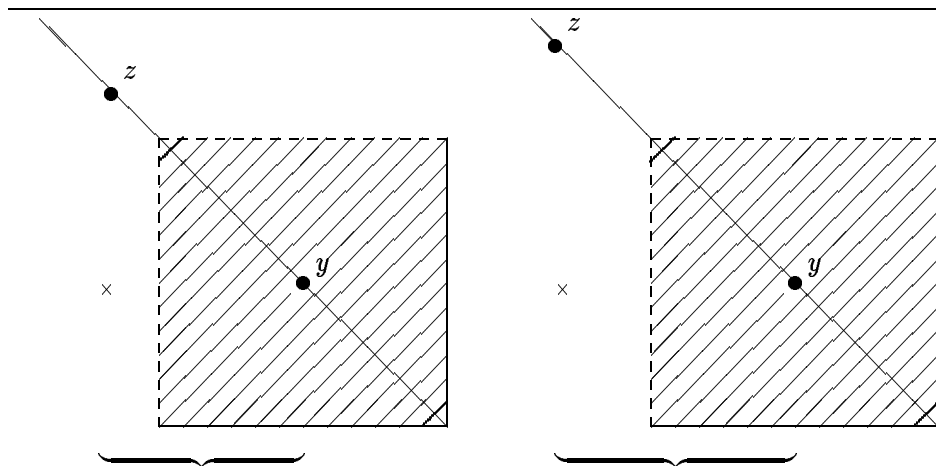


Figure 4.4.6: Diagram showing the effect of reversing the orderings indicated in Figure 4.4.1.

of the degrees of the nodes requires a further  $2|E|$  operations. Thus, the RCM algorithm requires at most

$$4|E| + 2cm|E| + n \text{ operations,}$$

where the last term represents the time required to reverse the ordering.  $\square$

#### 4.4.2 Finding a Starting Node

We now turn to the problem of finding a starting node for the RCM algorithm. We consider this problem separately because its solution is useful in connection with several other algorithms we consider in this book. In all cases the objective is to find a pair of nodes which are at maximum or near maximum “distance” apart (defined below). Substantial experience indicates that such nodes are good starting nodes for several ordering algorithms, including the RCM algorithm.

Recall from Section 3.2 that a path of length  $k$  from node  $x_0$  to  $x_k$  is an ordered set of distinct vertices  $(x_0, x_1, \dots, x_k)$ , where  $x_i \in Adj(x_{i+1})$  for  $0 \leq i \leq k - 1$ . The *distance*  $d(x, y)$  between two nodes  $x$  and  $y$  in the connected graph  $\mathcal{G} = (X, E)$  is simply the length of a shortest path joining nodes  $x$  and  $y$ . Following Berge [3], we define the *eccentricity* of a node  $x$  to be the quantity

$$\ell(x) = \max\{d(x, y) \mid y \in X\}. \quad (4.4.1)$$

The *diameter* of  $\mathcal{G}$  is then given by

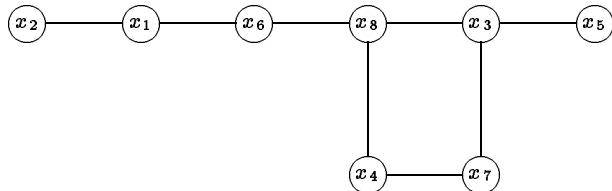
$$\delta(\mathcal{G}) = \max\{\ell(x) \mid x \in X\},$$

or equivalently

$$\delta(\mathcal{G}) = \max\{d(x, y) \mid x, y \in X\}.$$

A node  $x \in X$  is said to be a *peripheral* node if its eccentricity is equal to the diameter of the graph, that is, if  $\ell(x) = \delta(\mathcal{G})$ . Figure 4.4.7 shows a graph having 8 nodes, with a diameter of 5. The nodes  $x_2$ ,  $x_5$  and  $x_7$  are peripheral nodes.

With this terminology established, our objective in this subsection is to describe an efficient heuristic algorithm for finding nodes of high eccentricity. We emphasize that the algorithm is not *guaranteed* to find a peripheral node, or even one that is close to being peripheral. Nevertheless, the nodes found usually do have high eccentricity, and are good starting nodes for the algorithms that employ them. Furthermore, except for some fairly trivial

Figure 4.4.7: An 8-node graph  $\mathcal{G}$  with  $\delta(\mathcal{G}) = 5$ 

situations, there seems to be no reason to expect that peripheral nodes are any better as starting nodes than those found by this algorithm. Finally, in many situations it is probably too expensive to find peripheral nodes even if it were known to be desirable to use them, since the best known algorithm for finding them has a time complexity bound of  $O(|X||E|)$  (Smyth and Benzi [49]). For most sparse matrix applications this bound would be  $O(|X|^2)$ . In what follows, we will refer to nodes produced by this algorithm as *pseudo-peripheral* nodes.

We now introduce some notation and terminology which is useful in describing the algorithm. The reader may find it helpful to review the definitions of adjacent set, degree, section graph and connected component, introduced in Section 3.2. A key construct in the algorithm is the *rooted level structure* (Arany et al. [2]).<sup>2</sup> Given a node  $x \in X$ , the level structure rooted at  $x$  is the *partitioning*  $\mathcal{L}(x)$  of  $X$  satisfying

$$\mathcal{L}(x) = \{L_0(x), L_1(x), \dots, L_{\ell(x)}(x)\}, \quad (4.4.2)$$

where

$$L_0(x) = \{x\}, \quad L_1(x) = \text{Adj}(L_0(x)),$$

and

$$L_i(x) = \text{Adj}(L_{i-1}(x)) - L_{i-2}(x), \quad i = 2, 3, \dots, \ell(x). \quad (4.4.3)$$

The eccentricity  $\ell(x)$  of  $x$  is called the *length* of  $\mathcal{L}(x)$ , and the *width*  $w(x)$

<sup>2</sup>A *general* level structure is a partitioning  $\mathcal{L} = \{L_0, L_1, \dots, L_\ell\}$  where  $\text{Adj}(L_0) \subset L_1$ ,  $\text{Adj}(L_\ell) \subset L_{\ell-1}$  and  $\text{Adj}(L_i) \subset L_{i-1} \cup L_{i+1}$ ,  $i = 2, 3, \dots, \ell - 1$ .



of  $\mathcal{L}(x)$  is defined by

$$w(x) = \max\{|L_i(x)| \mid 0 \leq i \leq \ell(x)\}. \quad (4.4.4)$$

In Figure 4.4.8 we show a rooted level structure of the graph of Figure 4.4.7, rooted at the node  $x_6$ . Note that  $\ell(x_6) = 3$  and  $w(x_6) = 3$ .

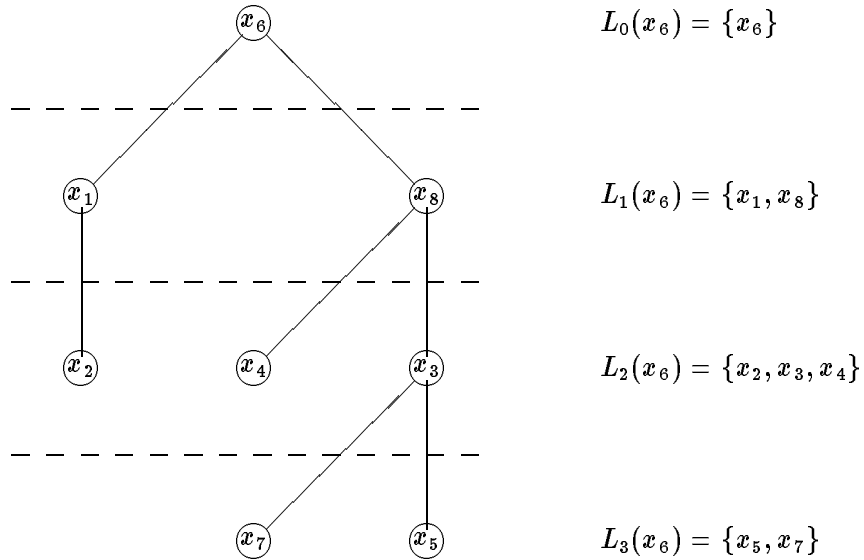


Figure 4.4.8: A level structure, rooted at  $x_6$ , of the graph of Figure 4.4.7.

We are now ready to describe the pseudo-peripheral node finding algorithm which is essentially a modification of an algorithm due to Gibbs et al. [30]. For details on why these modifications were made, see George and Liu [26]. Using our level structure notation just introduced, the algorithm is as follows.

**Step 1 (Initialization):** Choose an arbitrary node  $r$  in  $X$ .

**Step 2 (Generate a level structure):** Construct the level structure rooted at  $r$ :  $\mathcal{L}(r) = \{L_0(r), L_1(r), \dots, L_{\ell(r)}(r)\}$ .

**Step 3** (*Shrink last level*): Choose a node  $x$  in  $L_{\ell(r)}(r)$  of minimum degree.

**Step 4** (*Generate a level structure*):

- a) Construct the level structure rooted at  $x$ :
- b) If  $\ell(x) > \ell(r)$ , set  $r \leftarrow x$  and go to Step 3.

**Step 5** (*Finished*): The node  $x$  is a pseudo-peripheral node.

Computer subroutines `FNR00T` and `ROOTLS`, which implement this algorithm, are presented and discussed in the next subsection. An example showing the operation of the algorithm is given in Figure 4.4.9. Nodes in level  $i$  of the level structures are labelled with the integer  $i$ .

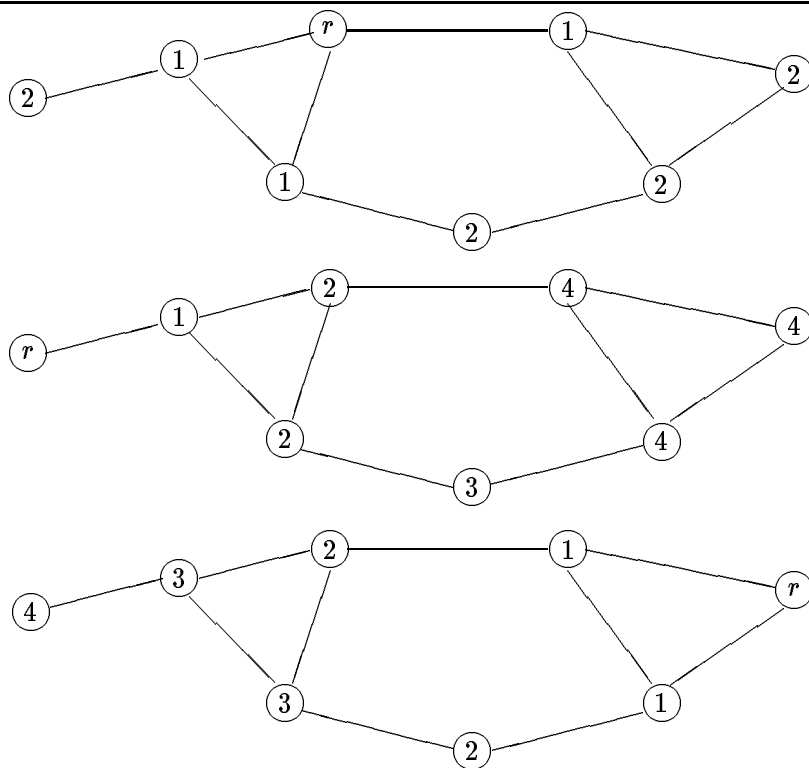


Figure 4.4.9: An example of the application of the pseudo-peripheral node finding algorithm.

---

### 4.4.3 Subroutines for Finding a Starting Node

In this subsection we present and describe a pair of subroutines which implement the algorithm of the previous section. In these subroutines, as well as those in Sections 4.4.4 and 4.5.2, several input parameters are the same, and have already been described in Section 3.4. The reader might find it useful to review that section before proceeding.

#### ROOTLS (ROOTed Level Structure)

The purpose of this subroutine is to generate a level structure of the connected component specified by the input parameters `ROOT`, `MASK`, `XADJ`, and `ADJNCY`, as described in Section 3.4. On exit from the subroutine, the level structure generated is rooted at `ROOT`, and is contained in the array pair `(XLS, LS)`, with nodes at level  $k$  given by `LS(j)`,  $XLS(k) \leq j < XLS(k+1)$ . The number of levels is provided by the variable `NLVL`. Note that since Fortran does not allow zero subscripts, we cannot have a “zero level,” so  $k$  here corresponds to level  $L_{k-1}$  in the level structure  $\mathcal{L}(\text{ROOT})$  in Section 4.4.2. Thus, `NLVL` is one greater than the eccentricity of `ROOT`.

The subroutine finds the nodes level by level; a new level is obtained for each execution of the loop `DO 400 . . .`. As each new node is found (in executing the loop `DO 300 . . .`), the node number is placed in the array `LS`, and its corresponding `MASK` value is set to zero so it will not be put in `LS` more than once. After the level structure has been generated, the values of `MASK` for the nodes in the level structure are reset to 1 (by executing the loop `DO 500 . . .`).

---

```

1. C*****
2. C*****
3. C*****   ROOTLS . . . . ROOTED LEVEL STRUCTURE   *****
4. C*****
5. C*****
6. C
7. C   PURPOSE - ROOTLS GENERATES THE LEVEL STRUCTURE ROOTED
8. C           AT THE INPUT NODE CALLED ROOT. ONLY THOSE NODES FOR
9. C           WHICH MASK IS NONZERO WILL BE CONSIDERED.
10. C
11. C   INPUT PARAMETERS -
12. C       ROOT - THE NODE AT WHICH THE LEVEL STRUCTURE IS TO
13. C            BE ROOTED.
14. C       (XADJ, ADJNCY) - ADJACENCY STRUCTURE PAIR FOR THE
15. C            GIVEN GRAPH.
16. C       MASK - IS USED TO SPECIFY A SECTION SUBGRAPH. NODES

```

```

17. C           WITH MASK(I)=0 ARE IGNORED.
18. C
19. C     OUTPUT PARAMETERS -
20. C     NLVL - IS THE NUMBER OF LEVELS IN THE LEVEL STRUCTURE.
21. C     (XLS, LS) - ARRAY PAIR FOR THE ROOTED LEVEL STRUCTURE.
22. C
23. C*****
24. C
25. C     SUBROUTINE ROOTLS ( ROOT, XADJ, ADJNCY, MASK, NLVL, XLS, LS )
26. C
27. C*****
28. C
29. C     INTEGER ADJNCY(1), LS(1), MASK(1), XLS(1)
30. C     INTEGER XADJ(1), I, J, JSTOP, JSTRT, LBEGIN,
31. C     1         CCSIZE, LVLEND, LVSIZE, NBR, NLVL,
32. C     1         NODE, ROOT
33. C
34. C*****
35. C
36. C     -----
37. C     INITIALIZATION ...
38. C     -----
39. C     MASK(ROOT) = 0
40. C     LS(1) = ROOT
41. C     NLVL = 0
42. C     LVLEND = 0
43. C     CCSIZE = 1
44. C     -----
45. C     LBEGIN IS THE POINTER TO THE BEGINNING OF THE CURRENT
46. C     LEVEL, AND LVLEND POINTS TO THE END OF THIS LEVEL.
47. C     -----
48. C     200  LBEGIN = LVLEND + 1
49. C         LVLEND = CCSIZE
50. C         NLVL = NLVL + 1
51. C         XLS(NLVL) = LBEGIN
52. C     -----
53. C     GENERATE THE NEXT LEVEL BY FINDING ALL THE MASKED
54. C     NEIGHBORS OF NODES IN THE CURRENT LEVEL.
55. C     -----
56. C     DO 400 I = LBEGIN, LVLEND
57. C         NODE = LS(I)
58. C         JSTRT = XADJ(NODE)
59. C         JSTOP = XADJ(NODE + 1) - 1
60. C         IF ( JSTOP .LT. JSTRT ) GO TO 400
61. C         DO 300 J = JSTRT, JSTOP
62. C             NBR = ADJNCY(J)
63. C             IF (MASK(NBR) .EQ. 0) GO TO 300

```

```

64.             CCSIZE = CCSIZE + 1
65.             LS(CCSIZE) = NBR
66.             MASK(NBR) = 0
67.   300       CONTINUE
68.   400       CONTINUE
69.   C         -----
70.   C         COMPUTE THE CURRENT LEVEL WIDTH.
71.   C         IF IT IS NONZERO, GENERATE THE NEXT LEVEL.
72.   C         -----
73.             LVSIZE = CCSIZE - LVLEND
74.             IF (LVSIZE .GT. 0 ) GO TO 200
75.   C         -----
76.   C         RESET MASK TO ONE FOR THE NODES IN THE LEVEL STRUCTURE.
77.   C         -----
78.             XLS(NLVL+1) = LVLEND + 1
79.             DO 500 I = 1, CCSIZE
80.               NODE = LS(I)
81.               MASK(NODE) = 1
82.   500       CONTINUE
83.             RETURN
84.             END

```

---

### FNROOT (FiNd ROOT)

This subroutine finds a pseudo-peripheral node of a connected component of a given graph, using the algorithm described in Section 4.4.2. The subroutine operates on the connected component specified by the input arguments `ROOT`, `MASK`, `XADJ`, and `ADJNCY`, as we described in Section 3.4.

The first call to `ROOTLS` corresponds to Step 2 of the algorithm. If the component consists of a single node or a chain with `ROOT` as its endpoint, then `ROOT` is a peripheral node and `LS` contains its corresponding rooted level structure, so execution terminates. Otherwise, a node of minimum degree in the last level is found (Step 3 of the algorithm; `DO 300 . . .` loop of the subroutine). The new level structure rooted at this node is generated (the call to `ROOTLS` with label 400) and the termination test (Step 4.b of the algorithm) is performed. If the test fails, control transfers to statement 100 and the procedure is repeated. On exit, `ROOT` is the node number of the pseudo-peripheral node, and the array pair `(XLS, LS)` contains the corresponding rooted level structure.

---

```

1.  C*****
2.  C*****

```

```

3. C*****      FNROOT . . . . FIND PSEUDO-PERIPHERAL NODE      *****
4. C*****
5. C*****
6. C
7. C      PURPOSE - FNROOT IMPLEMENTS A MODIFIED VERSION OF THE
8. C      SCHEME BY GIBBS, POOLE, AND STOCKMEYER TO FIND PSEUDO-
9. C      PERIPHERAL NODES. IT DETERMINES SUCH A NODE FOR THE
10. C     SECTION SUBGRAPH SPECIFIED BY MASK AND ROOT.
11. C
12. C     INPUT PARAMETERS -
13. C       (XADJ, ADJNCY) - ADJACENCY STRUCTURE PAIR FOR THE GRAPH.
14. C       MASK - SPECIFIES A SECTION SUBGRAPH. NODES FOR WHICH
15. C       MASK IS ZERO ARE IGNORED BY FNROOT.
16. C
17. C     UPDATED PARAMETER -
18. C       ROOT - ON INPUT, IT (ALONG WITH MASK) DEFINES THE
19. C       COMPONENT FOR WHICH A PSEUDO-PERIPHERAL NODE IS
20. C       TO BE FOUND. ON OUTPUT, IT IS THE NODE OBTAINED.
21. C
22. C     OUTPUT PARAMETERS -
23. C       NLVL - IS THE NUMBER OF LEVELS IN THE LEVEL STRUCTURE
24. C       ROOTED AT THE NODE ROOT.
25. C       (XLS,LS) - THE LEVEL STRUCTURE ARRAY PAIR CONTAINING
26. C       THE LEVEL STRUCTURE FOUND.
27. C
28. C     PROGRAM SUBROUTINES -
29. C       ROOTLS.
30. C
31. C*****
32. C
33. C       SUBROUTINE FNROOT ( ROOT, XADJ, ADJNCY, MASK, NLVL, XLS, LS )
34. C
35. C*****
36. C
37. C       INTEGER ADJNCY(1), LS(1), MASK(1), XLS(1)
38. C       INTEGER XADJ(1), CCSIZE, J, JSTRT, K, KSTOP, KSTRT,
39. C       1         MINDEG, NABOR, NDEG, NLVL, NODE, NUNLVL,
40. C       1         ROOT
41. C
42. C*****
43. C
44. C       -----
45. C       DETERMINE THE LEVEL STRUCTURE ROOTED AT ROOT.
46. C       -----
47. C       CALL ROOTLS ( ROOT, XADJ, ADJNCY, MASK, NLVL, XLS, LS )
48. C       CCSIZE = XLS(NLVL+1) - 1
49. C       IF ( NLVL .EQ. 1 .OR. NLVL .EQ. CCSIZE ) RETURN

```

```

50. C -----
51. C PICK A NODE WITH MINIMUM DEGREE FROM THE LAST LEVEL.
52. C -----
53. 100 JSTRT = XLS(NLVL)
54. MINDEG = CCSIZE
55. ROOT = LS(JSTRT)
56. IF ( CCSIZE .EQ. JSTRT ) GO TO 400
57. DO 300 J = JSTRT, CCSIZE
58. NODE = LS(J)
59. NDEG = 0
60. KSTRT = XADJ(NODE)
61. KSTOP = XADJ(NODE+1) - 1
62. DO 200 K = KSTRT, KSTOP
63. NABOR = ADJNCY(K)
64. IF ( MASK(NABOR) .GT. 0 ) NDEG = NDEG + 1
65. 200 CONTINUE
66. IF ( NDEG .GE. MINDEG ) GO TO 300
67. ROOT = NODE
68. MINDEG = NDEG
69. 300 CONTINUE
70. C -----
71. C AND GENERATE ITS ROOTED LEVEL STRUCTURE.
72. C -----
73. 400 CALL ROOTLS ( ROOT, XADJ, ADJNCY, MASK, NUNLVL, XLS, LS )
74. IF (NUNLVL .LE. NLVL) RETURN
75. NLVL = NUNLVL
76. IF ( NLVL .LT. CCSIZE ) GO TO 100
77. RETURN
78. END

```

---

#### 4.4.4 Subroutines for the Reverse Cuthill-McKee Algorithm

In this subsection we describe the three subroutines `DEGREE`, `RCM`, and `GENRCM`, which together with the subroutines of the previous section provide a complete implementation for the RCM algorithm described in Section 4.4.1. The roles of the input parameters `ROOT`, `MASK`, `XADJ`, `ADJNCY`, and `PERM` are as described in Section 3.4. The control relationship among the subroutines is given in Figure 4.4.10.

##### **DEGREE**

This subroutine computes the degrees of the nodes in a connected component of a graph. The subroutine operates on the connected component specified by the input parameters `ROOT`, `MASK`, `XADJ`, and `ADJNCY`.

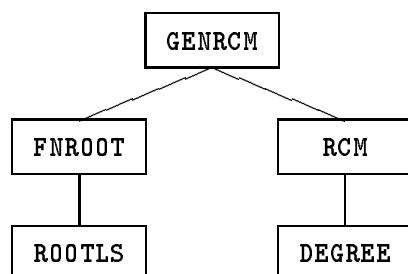


Figure 4.4.10: Control relation of subroutines for the reverse Cuthill-McKee algorithm.

---

Beginning with the first level (containing only `ROOT`), the degrees of the nodes are computed one level at a time (loop `DO 400 I = ...`). As the neighbors of these nodes are examined (loop `DO 200 J = ...`), those which are not already recorded in `LS` are put in that array, thus generating the next level of nodes. When a node is put in `LS`, its corresponding value of `XADJ` has its sign changed, so that the node will only be recorded once. (This function was performed using `MASK` in the subroutine `ROOTLS`, but here `MASK` must be maintained in its input form so that the degree will be computed correctly). The variable `CCSIZE` contains the number of nodes currently in `LS`. After all nodes have been found, and their degrees have been computed, the nodes in `LS` are used to reset the signs of the corresponding elements of `XADJ` to their original values (loop `DO 500 I = ...`).

---

```

1. C*****
2. C*****
3. C*****      DEGREE .... DEGREE IN MASKED COMPONENT      *****
4. C*****
5. C*****
6. C
7. C      PURPOSE - THIS ROUTINE COMPUTES THE DEGREES OF THE NODES
8. C      IN THE CONNECTED COMPONENT SPECIFIED BY MASK AND ROOT.
9. C      NODES FOR WHICH MASK IS ZERO ARE IGNORED.
10. C
11. C      INPUT PARAMETER -
12. C      ROOT - IS THE INPUT NODE THAT DEFINES THE COMPONENT.
13. C      (XADJ, ADJNCY) - ADJACENCY STRUCTURE PAIR.
14. C      MASK - SPECIFIES A SECTION SUBGRAPH.

```



```

15. C
16. C     OUTPUT PARAMETERS -
17. C     DEG - ARRAY CONTAINING THE DEGREES OF THE NODES IN
18. C     THE COMPONENT.
19. C     CCSIZE-SIZE OF THE COMPONENT SPECIFIED BY MASK AND ROOT
20. C
21. C     WORKING PARAMETER -
22. C     LS - A TEMPORARY VECTOR USED TO STORE THE NODES OF THE
23. C     COMPONENT LEVEL BY LEVEL.
24. C
25. C*****
26. C
27. C     SUBROUTINE DEGREE ( ROOT, XADJ, ADJNCY, MASK,
28. C     1                     DEG, CCSIZE, LS )
29. C
30. C*****
31. C
32. C     INTEGER ADJNCY(1), DEG(1), LS(1), MASK(1)
33. C     INTEGER XADJ(1), CCSIZE, I, IDEG, J, JSTOP, JSTRT,
34. C     1         LBEGIN, LVLEND, LVSIZE, NBR, NODE, ROOT
35. C
36. C*****
37. C
38. C     -----
39. C     INITIALIZATION ...
40. C     THE ARRAY XADJ IS USED AS A TEMPORARY MARKER TO
41. C     INDICATE WHICH NODES HAVE BEEN CONSIDERED SO FAR.
42. C     -----
43. C     LS(1) = ROOT
44. C     XADJ(ROOT) = -XADJ(ROOT)
45. C     LVLEND = 0
46. C     CCSIZE = 1
47. C     -----
48. C     LBEGIN IS THE POINTER TO THE BEGINNING OF THE CURRENT
49. C     LEVEL, AND LVLEND POINTS TO THE END OF THIS LEVEL.
50. C     -----
51. C     100 LBEGIN = LVLEND + 1
52. C     LVLEND = CCSIZE
53. C     -----
54. C     FIND THE DEGREES OF NODES IN THE CURRENT LEVEL,
55. C     AND AT THE SAME TIME, GENERATE THE NEXT LEVEL.
56. C     -----
57. C     DO 400 I = LBEGIN, LVLEND
58. C         NODE = LS(I)
59. C         JSTRT = -XADJ(NODE)
60. C         JSTOP = IABS(XADJ(NODE + 1)) - 1
61. C         IDEG = 0

```

```

62.             IF ( JSTOP .LT. JSTRT ) GO TO 300
63.             DO 200 J = JSTRT, JSTOP
64.                 NBR = ADJNCY(J)
65.                 IF ( MASK(NBR) .EQ. 0 ) GO TO 200
66.                 IDEG = IDEG + 1
67.                 IF ( XADJ(NBR) .LT. 0 ) GO TO 200
68.                 XADJ(NBR) = -XADJ(NBR)
69.                 CCSIZE = CCSIZE + 1
70.                 LS(CCSIZE) = NBR
71.     200         CONTINUE
72.     300         DEG(NODE) = IDEG
73.     400         CONTINUE
74.     C           -----
75.     C           COMPUTE THE CURRENT LEVEL WIDTH.
76.     C           IF IT IS NONZERO , GENERATE ANOTHER LEVEL.
77.     C           -----
78.             LVSIZE = CCSIZE - LVLEND
79.             IF ( LVSIZE .GT. 0 ) GO TO 100
80.     C           -----
81.     C           RESET XADJ TO ITS CORRECT SIGN AND RETURN.
82.     C           -----
83.             DO 500 I = 1, CCSIZE
84.                 NODE = LS(I)
85.                 XADJ(NODE) = -XADJ(NODE)
86.     500         CONTINUE
87.             RETURN
88.             END

```

---

### RCM (Reverse Cuthill-McKee)

This subroutine applies the RCM algorithm described in Section 4.4.1 to a connected component of a subgraph. It operates on a connected component specified by the input parameters `ROOT`, `MASK`, `XADJ`, and `ADJNCY`. The starting node is `ROOT`.

Since the algorithm requires the degrees of the nodes in the component, the first step is to compute those degrees by calling the subroutine `DEGREE`. The nodes are found and ordered in a level by level fashion; a new level is numbered each time the loop `DO 600 I = ...` is executed. The loop `DO 200 I = ...` finds the unnumbered neighbors of a node, and the remainder of the `DO 600` loop implements a linear insertion sort to order those neighbors in increasing order of degree. The new ordering is recorded in the array `PERM` as explained in Section 3.4. The final loop (`DO 700 I = ...`) reverses

the ordering, so that the reverse Cuthill-McKee ordering, rather than the standard Cuthill-McKee ordering is obtained.

Note that just as in the subroutine ROOTLS,  $MASK(i)$  is set to zero as node  $i$  is recorded. However, unlike ROOTLS, the subroutine RCM does not restore MASK to its original input state. The values of MASK corresponding to the nodes of the connected component that has been numbered remain set to zero on exit from the subroutine.

---

```

1. C*****
2. C*****
3. C*****      RCM . . . . REVERSE CUTHILL-MCKEE ORDERING      *****
4. C*****
5. C*****
6. C
7. C      PURPOSE - RCM NUMBERS A CONNECTED COMPONENT SPECIFIED BY
8. C      MASK AND ROOT, USING THE RCM ALGORITHM.
9. C      THE NUMBERING IS TO BE STARTED AT THE NODE ROOT.
10. C
11. C      INPUT PARAMETERS -
12. C      ROOT - IS THE NODE THAT DEFINES THE CONNECTED
13. C      COMPONENT AND IT IS USED AS THE STARTING
14. C      NODE FOR THE RCM ORDERING.
15. C      (XADJ, ADJNCY) - ADJACENCY STRUCTURE PAIR FOR
16. C      THE GRAPH.
17. C
18. C      UPDATED PARAMETERS -
19. C      MASK - ONLY THOSE NODES WITH NONZERO INPUT MASK
20. C      VALUES ARE CONSIDERED BY THE ROUTINE. THE
21. C      NODES NUMBERED BY RCM WILL HAVE THEIR
22. C      MASK VALUES SET TO ZERO.
23. C
24. C      OUTPUT PARAMETERS -
25. C      PERM - WILL CONTAIN THE RCM ORDERING.
26. C      CCSIZE - IS THE SIZE OF THE CONNECTED COMPONENT
27. C      THAT HAS BEEN NUMBERED BY RCM.
28. C
29. C      WORKING PARAMETER -
30. C      DEG - IS A TEMPORARY VECTOR USED TO HOLD THE DEGREE
31. C      OF THE NODES IN THE SECTION GRAPH SPECIFIED
32. C      BY MASK AND ROOT.
33. C
34. C      PROGRAM SUBROUTINES -
35. C      DEGREE.
36. C
37. C*****
38. C

```

```

39.          SUBROUTINE RCM ( ROOT, XADJ, ADJNCY, MASK,
40.            1              PERM, CCSIZE, DEG )
41.      C
42.      C*****
43.      C
44.          INTEGER ADJNCY(1), DEG(1), MASK(1), PERM(1)
45.          INTEGER XADJ(1), CCSIZE, FNBR, I, J, JSTOP,
46.            1          JSTRT, K, L, LBEGIN, LNBR, LPERM,
47.            1          LVLEND, NBR, NODE, ROOT
48.      C
49.      C*****
50.      C
51.      C          -----
52.      C          FIND THE DEGREES OF THE NODES IN THE
53.      C          COMPONENT SPECIFIED BY MASK AND ROOT.
54.      C          -----
55.          CALL DEGREE ( ROOT, XADJ, ADJNCY, MASK, DEG,
56.            1          CCSIZE, PERM )
57.          MASK(ROOT) = 0
58.          IF ( CCSIZE .LE. 1 ) RETURN
59.          LVLEND = 0
60.          LNBR = 1
61.      C          -----
62.      C          LBEGIN AND LVLEND POINT TO THE BEGINNING AND
63.      C          THE END OF THE CURRENT LEVEL RESPECTIVELY.
64.      C          -----
65.          100  LBEGIN = LVLEND + 1
66.              LVLEND = LNBR
67.              DO 600 I = LBEGIN, LVLEND
68.      C          -----
69.      C          FOR EACH NODE IN CURRENT LEVEL ...
70.      C          -----
71.              NODE = PERM(I)
72.              JSTRT = XADJ(NODE)
73.              JSTOP = XADJ(NODE+1) - 1
74.      C          -----
75.      C          FIND THE UNNUMBERED NEIGHBORS OF NODE.
76.      C          FNBR AND LNBR POINT TO THE FIRST AND LAST
77.      C          UNNUMBERED NEIGHBORS RESPECTIVELY OF THE CURRENT
78.      C          NODE IN PERM.
79.      C          -----
80.              FNBR = LNBR + 1
81.              DO 200 J = JSTRT, JSTOP
82.                  NBR = ADJNCY(J)
83.                  IF ( MASK(NBR) .EQ. 0 ) GO TO 200
84.                      LNBR = LNBR + 1
85.                      MASK(NBR) = 0

```

```

86 .          PERM(LNBR) = NBR
87 .   200    CONTINUE
88 .          IF ( FNBR .GE. LNBR ) GO TO 600
89 .   C     -----
90 .   C     SORT THE NEIGHBORS OF NODE IN INCREASING
91 .   C     ORDER BY DEGREE. LINEAR INSERTION IS USED.
92 .   C     -----
93 .          K = FNBR
94 .   300    L = K
95 .          K = K + 1
96 .          NBR = PERM(K)
97 .   400    IF ( L .LT. FNBR ) GO TO 500
98 .          LPERM = PERM(L)
99 .          IF ( DEG(LPERM) .LE. DEG(NBR) ) GO TO 500
100 .         PERM(L+1) = LPERM
101 .         L = L - 1
102 .         GO TO 400
103 .   500    PERM(L+1) = NBR
104 .         IF ( K .LT. LNBR ) GO TO 300
105 .   600    CONTINUE
106 .         IF (LNBR .GT. LVLEND) GO TO 100
107 .   C     -----
108 .   C     WE NOW HAVE THE CUTHILL MCKEE ORDERING.
109 .   C     REVERSE IT BELOW ...
110 .   C     -----
111 .         K = CCSIZE/2
112 .         L = CCSIZE
113 .         DO 700 I = 1, K
114 .           LPERM = PERM(L)
115 .           PERM(L) = PERM(I)
116 .           PERM(I) = LPERM
117 .           L = L - 1
118 .   700    CONTINUE
119 .         RETURN
120 .         END

```

---

### GENRCM (GENeral RCM)

This subroutine finds the RCM ordering of a general disconnected graph. It proceeds through the graph, and calls the subroutine RCM to number each connected component. The inputs to the subroutine are the number of nodes (or equations) NEQNS, and the graph in the array pair (XADJ, ADJNCY). The arrays MASK and XLS are working arrays, used by the subroutines FNROOT and RCM, which are called by GENRCM.

The subroutine begins by setting all values of MASK to 1 (loop DO 100 I = ...). It then loops through MASK until it finds an  $i$  for which MASK( $i$ ) = 1; node  $i$  along with MASK, XADJ, and ADJNCY will specify a connected subgraph of the original graph  $\mathcal{G}$ . The subroutines FNROOT and RCM are then called to order the nodes of that subgraph. (Recall that the numbered nodes will have their MASK values set to zero by RCM.) Note that NUM points to the first free position in the array PERM, and is updated after each call to RCM. The actual parameter in GENRCM corresponding to PERM in RCM is PERM(NUM); that is, PERM in RCM corresponds to the last NEQNS - NUM + 1 elements of PERM in GENRCM. Note also that these same elements of PERM are used to store the level structure in FNROOT. They correspond to the array LS in the execution of that subroutine.

After the component is ordered, the search for another  $i$  for which MASK( $i$ )  $\neq$  0 resumes, until either the loop is exhausted, or NEQNS nodes have been numbered.

---

```

1. C*****
2. C*****
3. C*****  GENRCM . . . .  GENERAL REVERSE CUTHILL MCKEE  *****
4. C*****
5. C*****
6. C
7. C    PURPOSE - GENRCM FINDS THE REVERSE CUTHILL-MCKEE
8. C    ORDERING FOR A GENERAL GRAPH. FOR EACH CONNECTED
9. C    COMPONENT IN THE GRAPH, GENRCM OBTAINS THE ORDERING
10. C    BY CALLING THE SUBROUTINE RCM.
11. C
12. C    INPUT PARAMETERS -
13. C    NEQNS - NUMBER OF EQUATIONS
14. C    (XADJ, ADJNCY) - ARRAY PAIR CONTAINING THE ADJACENCY
15. C    STRUCTURE OF THE GRAPH OF THE MATRIX.
16. C
17. C    OUTPUT PARAMETER -
18. C    PERM - VECTOR THAT CONTAINS THE RCM ORDERING.
19. C
20. C    WORKING PARAMETERS -
21. C    MASK - IS USED TO MARK VARIABLES THAT HAVE BEEN
22. C    NUMBERED DURING THE ORDERING PROCESS. IT IS
23. C    INITIALIZED TO 1, AND SET TO ZERO AS EACH NODE
24. C    IS NUMBERED.
25. C    XLS - THE INDEX VECTOR FOR A LEVEL STRUCTURE. THE
26. C    LEVEL STRUCTURE IS STORED IN THE CURRENTLY
27. C    UNUSED SPACES IN THE PERMUTATION VECTOR PERM.
28. C

```

```

29. C    PROGRAM SUBROUTINES -
30. C        FNROOT, RCM.
31. C
32. C*****
33. C
34. C        SUBROUTINE GENRCM ( NEQNS, XADJ, ADJNCY, PERM, MASK, XLS )
35. C
36. C*****
37. C
38. C        INTEGER ADJNCY(1), MASK(1), PERM(1), XLS(1)
39. C        INTEGER XADJ(1), CCSIZE, I, NEQNS, NLVL,
40. C           1      NUM, ROOT
41. C
42. C*****
43. C
44. C        DO 100 I = 1, NEQNS
45. C            MASK(I) = 1
46. C    100    CONTINUE
47. C            NUM = 1
48. C            DO 200 I = 1, NEQNS
49. C                -----
50. C                FOR EACH MASKED CONNECTED COMPONENT ...
51. C                -----
52. C                IF (MASK(I) .EQ. 0) GO TO 200
53. C                ROOT = I
54. C                -----
55. C                FIRST FIND A PSEUDO-PERIPHERAL NODE ROOT.
56. C                NOTE THAT THE LEVEL STRUCTURE FOUND BY
57. C                FNROOT IS STORED STARTING AT PERM(NUM).
58. C                THEN RCM IS CALLED TO ORDER THE COMPONENT
59. C                USING ROOT AS THE STARTING NODE.
60. C                -----
61. C                CALL FNROOT ( ROOT, XADJ, ADJNCY, MASK,
62. C           1                    NLVL, XLS, PERM(NUM) )
63. C                CALL RCM ( ROOT, XADJ, ADJNCY, MASK,
64. C           1                    PERM(NUM), CCSIZE, XLS )
65. C                NUM = NUM + CCSIZE
66. C                IF (NUM .GT. NEQNS) RETURN
67. C    200    CONTINUE
68. C        RETURN
69. C        END

```

## Exercises

- 4.4.1) Let the graph associated with a given matrix be the  $n$  by  $n$  grid graph. Here is the case when  $n = 5$ .

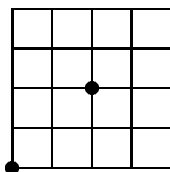


Figure 4.4.11: A 5 by 5 grid.

- a) Show that if the reverse Cuthill-McKee algorithm starts at a corner node, the profile is  $\frac{2}{3}n^3 + O(n^2)$ .
- b) What if the scheme starts at the center node?
- 4.4.2) Give an example where the algorithm of Section 4.4.2 will fail to find a peripheral node. Find a large example which is particularly bad, say some significant fraction of  $|X|$  from the diameter. The authors do not know of a large example where the execution time will be greater than  $O(|E|)$ . Can you find one?
- 4.4.3) The original pseudo-peripheral node finding algorithm of Gibbs et. al (1976b) did not have a “shrinking step;” Steps 3 and 4 were as follows:

**Step 3:** (*Sort the last level*): Sort the nodes in  $L_{\ell(r)}(r)$  in order of increasing degree.

**Step 4:** (*Test for termination*): For  $x \in L_{\ell(r)}(r)$  in order of increasing degree, generate

$$\mathcal{L}(x) = \{L_0(x), L_1(x), \dots, L_{\ell(x)}(x)\}.$$

If  $\ell(x) > \ell(r)$ , set  $r \leftarrow x$  and go to Step 3.

Give an example to show that the execution time of this algorithm can be greater than  $O(|E|)$ . Answer the first two questions in Exercise 4.4.2 on page 87 for this algorithm.

- 4.4.4) Suppose we delete Step 3 of the algorithm of Section 4.4.1. The ordering given by  $x_1, x_2, \dots, x_n$  is called the *Cuthill-McKee ordering*. Let  $\mathbf{A}_c$  be the matrix ordered by this algorithm. Show that



- a) the matrix  $\mathbf{A}_c$  has the monotone profile property (see Exercise 4.3.2 on page 64 ),
- b) in the graph  $\mathcal{G}^{\mathbf{A}_c}$ , for  $1 < i \leq n$

$$Adj(\{x_i, \dots, x_n\}) \subset \{x_{f_i(\mathbf{A}_c)}, \dots, x_{i-1}\}.$$

- 4.4.5) Show that  $Env(\mathbf{A}_r) = Env(\mathbf{A}_c)$  if and only if the matrix  $\mathbf{A}_r$  has the monotone profile property. Here  $\mathbf{A}_r$  is the matrix ordered by the algorithm of Section 4.4.1, and  $\mathbf{A}_c$  is as described in Exercise 4.4.4 on page 87.
- 4.4.6) What ensures that the pseudo-peripheral node finding algorithm described in Section 4.4.2 terminates?
- 4.4.7) Consider the  $n$  by  $n$  symmetric positive definite system of equations  $\mathbf{A}\mathbf{x} = \mathbf{b}$  derived from an  $s$  by  $s$  finite element mesh as follows. The mesh consists of  $(s - 1)^2$  small squares, as shown in Figure 4.4.10 for  $s = 5$ , each mesh square has a node at its vertices and midsides, and there is one variable  $x_i$  associated with each node. For some labelling of the  $n = 3s^2 - 2s$  nodes, the matrix  $\mathbf{A}$  has the property that  $a_{ij} \neq 0$  if and only if  $x_i$  and  $x_j$  are associated with the same mesh square.

We have a choice of two orderings,  $\alpha_1$  and  $\alpha_2$ , as shown in Figure 4.4.12. The orderings are similar in that they both number the nodes mesh line by mesh line. Their difference is essentially that  $\alpha_1$  numbers nodes on each horizontal mesh line and on the vertical lines immediately *above* it at the same time, while  $\alpha_2$  numbers nodes on a horizontal line along with nodes on the vertical lines immediately *below* it at the same time, as depicted by the dashed lines in the diagrams.

- a) What is the bandwidth of  $\mathbf{A}$ , for orderings  $\alpha_1$  and  $\alpha_2$ ?
- b) Suppose the envelope method is used to solve  $\mathbf{A}\mathbf{x} = \mathbf{b}$ , using orderings  $\alpha_1$  and  $\alpha_2$ . Let  $\theta_1$  and  $\theta_2$  be the corresponding arithmetic operation counts, and let  $\eta_1$  and  $\eta_2$  be the corresponding storage requirements. Show that for large  $s$ ,

$$\begin{aligned}\theta_1 &= 6s^4 + O(s^3) \\ \theta_2 &= 13.5s^4 + O(s^3)\end{aligned}$$

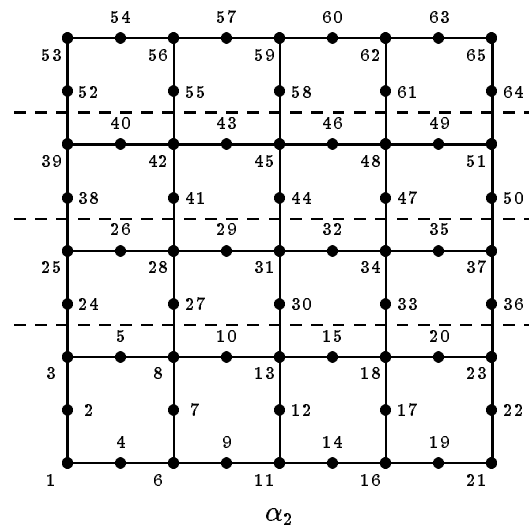
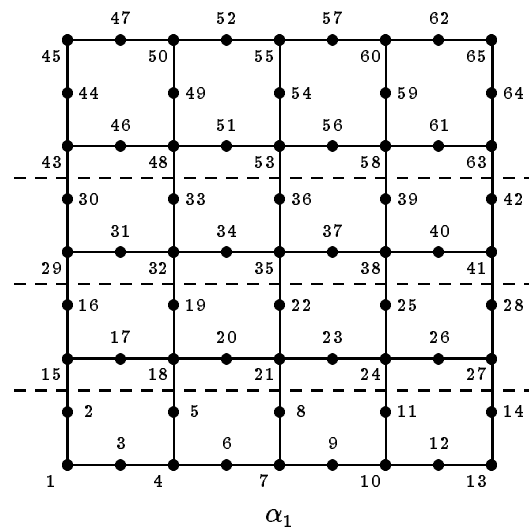


Figure 4.4.12: Two orderings  $\alpha_1$  and  $\alpha_2$  of a 5 by 5 finite element mesh.

$$\begin{aligned}\eta_1 &= 6s^3 + O(s^2) \\ \eta_2 &= 9s^3 + O(s^2).\end{aligned}$$

Orderings  $\alpha_1$  and  $\alpha_2$  resemble the type of ordering produced by the RCM and standard Cuthill-McKee ordering algorithms respectively; the results above illustrate the substantial differences in storage and operation counts the two orderings can produce. For more details see Liu and Sherman [38]

4.4.8) (*King Ordering*) King [33] has proposed an algorithm for reducing the profile of a symmetric matrix. His algorithm for a connected graph can be described as follows.

**Step 1** (*Initialization*) Determine a pseudo-peripheral node  $r$  and assign  $x_1 \leftarrow r$ .

**Step 2** (*Main loop*) For  $i = 1, \dots, n-1$ , find a node  $y \in Adj(\{x_1, \dots, x_i\})$  with minimum

$$|Adj(\{x_1, \dots, x_i, y\})|.$$

Number the node  $y$  as  $x_{i+1}$ .

**Step 3** (*Exit*) The King ordering is given by  $x_1, x_2, \dots, x_n$ .

This algorithm reduces the profile by a local minimization of the frontwidth. Implement this algorithm for general disconnected graphs. Run your program on the matrices in test set #1 of Chapter 9. Compare the performance of this algorithm with that of RCM.

## 4.5 Implementation of the Envelope Method

### 4.5.1 An Envelope Storage Scheme

The most commonly used storage scheme for the envelope method is the one proposed by Jennings [32]. For each row in the matrix, all the entries from the first nonzero to the diagonal are stored. These row portions are stored in contiguous locations in a one dimensional array. However, we use a modification of this scheme, in which the diagonal entries are stored in a separate vector. An advantage of this variant scheme is that it lends itself readily to the case when  $\mathbf{A}$  is unsymmetric; this point is pursued in an exercise at the end of this chapter.

The scheme has a main storage array **ENV** which contains the envelope entries of each row in the matrix. An auxiliary index vector **XENV** of length  $n$  is used to point to the start of each row portion. For uniformity in indexing, we set  $\mathbf{XENV}(n+1)$  to  $|\mathit{Env}(\mathbf{A})| + 1$ . In this way, the index vector **XENV** allows us to access any nonzero component conveniently. The mapping from  $\mathit{Env}(\mathbf{A})$  to  $\{1, 2, \dots, |\mathit{Env}(\mathbf{A})|\}$  is given by:

$$\{i, j\} \rightarrow \mathbf{XENV}(i+1) - (i-j).$$

In other words, a component  $a_{ij}$  within the envelope region of  $\mathbf{A}$  is found in  $\mathbf{ENV}(\mathbf{XENV}(i+1) - (i-j))$ . Figure 4.5.1 illustrates the storage scheme. For example, to retrieve  $a_{64}$ , we have

$$\mathbf{XENV}(7) - (6-4) = 8$$

so that  $a_{64}$  is stored in the 8-th element of the vector **ENV**.

A more frequently used operation is to retrieve the envelope portion of a row. This can be done conveniently as follows.

```

      .
      .
      .
      JSTRT = XENV(IROW)
      JSTOP = XENV(IROW+1) - 1
      IF (JSTOP.LT.JSTRT) GO TO 200
      DO 100 J = JSTRT, JSTOP
          ELEMNT = ENV(J)
      .
      .
      .
100 CONTINUE
200 .
      .
      .

```

The primary storage of the scheme is  $|\mathit{Env}(\mathbf{A})| + n$  and the overhead storage is  $n + 1$ . The data structure for the storage scheme can be set up in  $O(|E|)$  time and the subroutine **FNENV**, discussed in the next subsection, performs this function.

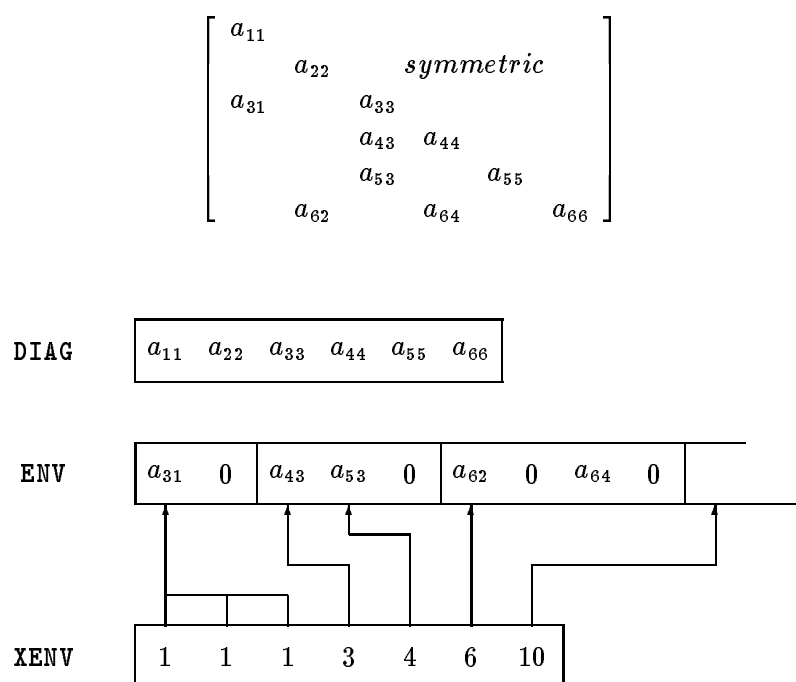


Figure 4.5.1: Example of the envelope storage scheme.

---

### 4.5.2 The Storage Allocation Subroutine FNENV (FiNd ENVe-lope)

In this section we describe the subroutine FNENV. This subroutine accepts as input the graph of the matrix  $A$ , stored in the array pair (XADJ, ADJNCY), along with the permutation vector PERM and its inverse INVP (discussed in Section 3.4). The objective of the subroutine is to compute the components of the array XENV discussed in Section 4.5.1, which is used in connection with storing the factor  $L$  of  $PAP^T$ . Also returned is the value ENVSZE, which is the envelope size of  $L$  and equals  $XENV(NEQNS + 1) - 1$ . Here as before, NEQNS is the number of equations or nodes.

The subroutine is straightforward and needs little explanation. The loop DO 200 I = ... processes each row; the index of the first nonzero in the  $i$ -th row (IFIRST) of  $PAP^T$  is determined by the loop DO 100 J = ... At the end of each execution of the loop DO 100 J = ..., ENVSZE is suitably updated. Note that PERM and INVP are used since the array pair (XADJ, ADJNCY) stores the structure of  $A$ , but the structure of  $L$  we are finding corresponds to  $PAP^T$ .

---

```

1. C*****
2. C*****
3. C*****          FNENV . . . . FIND ENVELOPE          *****
4. C*****
5. C*****
6. C
7. C      PURPOSE - FINDS THE ENVELOPE STRUCTURE OF A PERMUTED
8. C                  MATRIX.
9. C
10. C      INPUT PARAMETERS -
11. C          NEQNS - NUMBER OF EQUATIONS
12. C          (XADJ, ADJNCY) - ARRAY PAIR CONTAINING THE ADJACENCY
13. C                        STRUCTURE OF THE GRAPH OF THE MATRIX.
14. C          PERM,INVP - ARRAYS CONTAINING PERMUTATION DATA ABOUT
15. C                    THE REORDERED MATRIX.
16. C
17. C      OUTPUT PARAMETERS -
18. C          XENV - INDEX VECTOR FOR THE LEVEL STRUCTURE
19. C                TO BE USED TO STORE THE LOWER (OR UPPER)
20. C                ENVELOPE OF THE REORDERED MATRIX.
21. C          ENVSZE - IS EQUAL TO XENV(NEQNS+1) - 1.
22. C          BANDW - BANDWIDTH OF THE REORDERED MATRIX.
23. C
24. C*****
25. C

```

```

26.          SUBROUTINE FNENV ( NEQNS, XADJ, ADJNCY, PERM, INVP,
27.            1              XENV, ENVSZE, BANDW )
28.      C
29.      C*****
30.      C
31.          INTEGER ADJNCY(1), INVP(1), PERM(1)
32.          INTEGER XADJ(1), XENV(1), BANDW, I, IBAND,
33.            1          IFIRST, IPERM, J, JSTOP, JSTRT, ENVSZE,
34.            1          NABOR, NEQNS
35.      C
36.      C*****
37.      C
38.          BANDW = 0
39.          ENVSZE = 1
40.          DO 200 I = 1, NEQNS
41.              XENV(I) = ENVSZE
42.              IPERM = PERM(I)
43.              JSTRT = XADJ(IPERM)
44.              JSTOP = XADJ(IPERM + 1) - 1
45.              IF ( JSTOP .LT. JSTRT ) GO TO 200
46.      C          -----
47.      C          FIND THE FIRST NONZERO IN ROW I.
48.      C          -----
49.              IFIRST = I
50.              DO 100 J = JSTRT, JSTOP
51.                  NABOR = ADJNCY(J)
52.                  NABOR = INVP(NABOR)
53.                  IF ( NABOR .LT. IFIRST ) IFIRST = NABOR
54.            100          CONTINUE
55.              IBAND = I - IFIRST
56.              ENVSZE = ENVSZE + IBAND
57.              IF ( BANDW .LT. IBAND ) BANDW = IBAND
58.            200          CONTINUE
59.              XENV(NEQNS+1) = ENVSZE
60.              ENVSZE = ENVSZE - 1
61.              RETURN
62.          END

```

---

## 4.6 The Numerical Subroutines ESFCT, ELSLV and EUSLV

In this section we describe the subroutines which perform the numerical factorization and solution, using the envelope storage scheme described in Section 4.5.1. We describe the triangular solution subroutines ELSLV (Envelope-

Lower-SoLve) and EUSLV (Envelope-Upper-SoLve) before the factorization subroutine ESFCT (Envelope-Symmetric-FaCTorization) because ELSLV is used by ESFCT.

#### 4.6.1 The Triangular Solution Subroutines ELSLV and EUSLV.

These subroutines carry out the numerical solutions of the lower and upper triangular systems

$$L\mathbf{y} = \mathbf{b}$$

and

$$L^T \mathbf{x} = \mathbf{y},$$

respectively, where  $L$  is a lower triangular matrix stored as described in Section 4.5.1.

There are several important features of ELSLV which deserve explanation. To begin, the position (IFIRST) of the first nonzero in the right hand side (RHS) is determined. With this initialization, the program then loops (DO 500 I = ...) over the rows IFIRST, IFIRST+1, ..., NEQNS of  $L$ , using the inner product scheme described in Section 2.3.1. However, the program attempts to exploit strings of zeros in the solution; the variable LAST is used to store the index of the most recently computed *nonzero* component of the solution. (The solution overwrites the input right hand side array RHS.)

The reader is urged to simulate the subroutine's action on the problem described by Figure 4.6.1 to verify that only the nonzeros denoted by  $\otimes$  are actually used by the subroutine ELSLV.

Note that LAST simply allows us to skip certain rows; we still perform some multiplications with zero operands in the DO 300 K ... loop, but on most machines a test to avoid such a multiplication is more costly than going ahead and doing it.

The test and adjustment of IBAND just preceding the DO 300 ... loop also requires some explanation. In some circumstances ELSLV is used to solve a lower triangular system where the coefficient matrix to be used is only a *submatrix* of the matrix passed to ELSLV in the array pair (XENV, ENV), as depicted in Figure 4.6.2. Some of the rows of the envelope protrude outside the coefficient matrix to be used, and IBAND is appropriately adjusted to account for them. In the example in Figure 4.6.2,  $L$  is actually 16 by 16, and if the system we wish to solve is the submatrix indicated by the 11 by 11 system with right hand side RHS, we would solve it by executing the statement



---


$$\begin{array}{ccc}
 \left[ \begin{array}{cccccccc}
 \times & & & & & & & \\
 \times & \times & & & & & & \\
 & \times & \otimes & & & & & \\
 \otimes & \otimes & \otimes & \otimes & & & & \\
 & & & \times & & & & \\
 & & & & \otimes & & & \\
 & & & & \otimes & \otimes & & \\
 & & & & & \times & & \\
 & & & & & \times & \times & \\
 & & & & & \times & \times & \otimes \\
 & & & & & & \otimes & \otimes & \otimes \\
 \otimes & \otimes & \otimes & \otimes & & & \otimes & \otimes & \otimes
 \end{array} \right] & \left[ \begin{array}{c} \times \\ \times \\ \\ \times \\ \times \\ \\ \times \\ \times \\ \\ \times \\ \times \\ \\ \times \end{array} \right] & = & \left[ \begin{array}{c} \times \\ \\ \times \\ \times \\ \\ \times \\ \\ \times \end{array} \right] \\
 \mathbf{L} & \text{Solution} & & \text{Right hand} \\
 & & & \text{side} \\
 (\mathbf{XENV}, \mathbf{ENV}) & (\mathbf{RHS}) & & (\mathbf{RHS})
 \end{array}$$

Figure 4.6.1: Elements of  $\mathbf{L}$  actually used by ELSLV are denoted by  $\otimes$ .

---

4.6. THE NUMERICAL SUBROUTINES ESFCT, ELSLV AND EUSLV 97

```
CALL ELSLV ( 11, XENV(5), ENV, DIAG(5), RHS ) .
```

In the subroutine, XENV(5) is interpreted as XENV(1), XENV(6) becomes XENV(2), etc. This “trick” is used heavily by the subroutine ESFCT, which calls ELSLV.

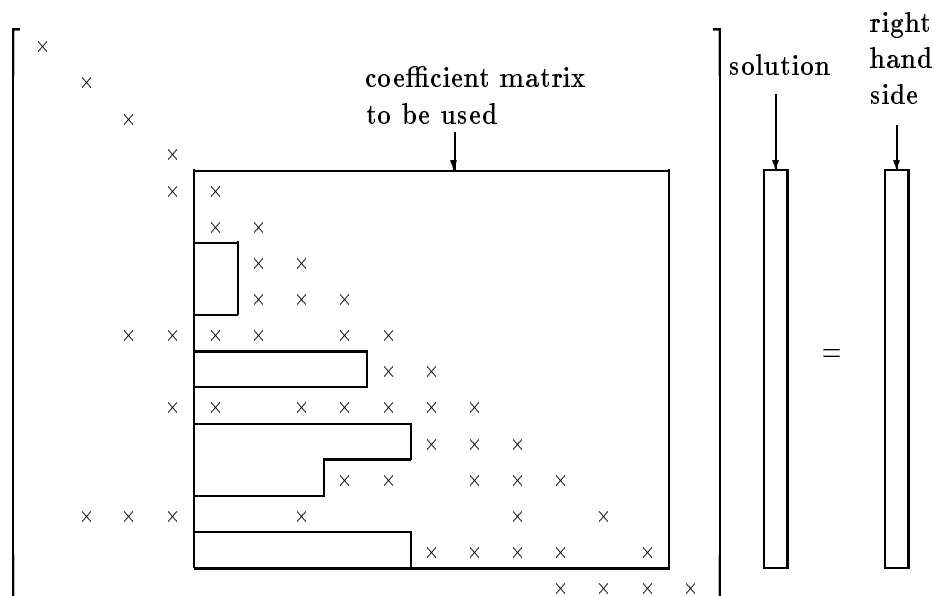


Figure 4.6.2: An example illustrating the use of ELSLV.

```

1. C*****
2. C*****
3. C*****      ELSLV . . . . ENVELOPE LOWER SOLVE      *****
4. C*****
5. C*****
6. C
7. C   PURPOSE - THIS SUBROUTINE SOLVES A LOWER TRIANGULAR
8. C   SYSTEM L X = RHS. THE FACTOR L IS STORED IN THE
9. C   ENVELOPE FORMAT.
10. C
11. C   INPUT PARAMETERS -

```

```

12. C      NEQNS - NUMBER OF EQUATIONS.
13. C      (XENV, ENV) - ARRAY PAIR FOR THE ENVELOPE OF L.
14. C      DIAG - ARRAY FOR THE DIAGONAL OF L.
15. C
16. C      UPDATED PARAMETER -
17. C      RHS - ON INPUT, IT CONTAINS THE RIGHT HAND VECTOR.
18. C      ON RETURN, IT CONTAINS THE SOLUTION VECTOR.
19. C      OPS - DOUBLE PRECISION VARIABLE CONTAINED IN THE
20. C      LABELLED COMMON BLOCK OPNS. ITS VALUE IS
21. C      INCREASED BY THE NUMBER OF OPERATIONS
22. C      PERFORMED BY THIS SUBROUTINE.
23. C
24. C*****
25. C
26. C      SUBROUTINE ELSLV ( NEQNS, XENV, ENV, DIAG, RHS )
27. C
28. C*****
29. C
30. C      DOUBLE PRECISION COUNT, OPS
31. C      COMMON /SPKOPS/ OPS
32. C      REAL DIAG(1), ENV(1), RHS(1), S
33. C      INTEGER XENV(1), I, IBAND, IFIRST, K, KSTOP,
34. C      1      KSTRT, L, LAST, NEQNS
35. C
36. C*****
37. C
38. C      -----
39. C      FIND THE POSITION OF THE FIRST NONZERO IN RHS AND
40. C      PUT IT IN IFIRST.
41. C      -----
42. C      IFIRST = 0
43. C      100  IFIRST = IFIRST + 1
44. C          IF ( RHS(IFIRST) .NE. 0.0E0 ) GO TO 200
45. C          IF ( IFIRST .LT. NEQNS ) GO TO 100
46. C          RETURN
47. C      200  LAST = 0
48. C      -----
49. C      LAST CONTAINS THE POSITION OF THE MOST RECENTLY
50. C      COMPUTED NONZERO COMPONENT OF THE SOLUTION.
51. C      -----
52. C      DO 500 I = IFIRST, NEQNS
53. C          IBAND = XENV(I+1) - XENV(I)
54. C          IF ( IBAND .GE. I ) IBAND = I - 1
55. C          S = RHS(I)
56. C          L = I - IBAND
57. C          RHS(I) = 0.0E0
58. C      -----

```

4.6. THE NUMERICAL SUBROUTINES ESFCT, ELSLV AND EUSLV 99

```

59. C          ROW OF THE ENVELOPE IS EMPTY, OR CORRESPONDING
60. C          COMPONENTS OF THE SOLUTION ARE ALL ZEROS.
61. C          -----
62.          IF ( IBAND .EQ. 0 .OR. LAST .LT. L ) GO TO 400
63.          KSTRT = XENV(I+1) - IBAND
64.          KSTOP = XENV(I+1) - 1
65.          DO 300 K = KSTRT, KSTOP
66.             S = S - ENV(K)*RHS(L)
67.             L = L + 1
68. 300        CONTINUE
69.          COUNT = IBAND
70.          OPS = OPS + COUNT
71. 400        IF ( S .EQ. 0.0E0 ) GO TO 500
72.          RHS(I) = S/DIAG(I)
73.          OPS = OPS + 1.0D0
74.          LAST = I
75. 500        CONTINUE
76.          RETURN
77.          END

```

We now turn to a description of the subroutine EUSLV, which solves the problem  $L^T x = y$ , with  $L$  stored using the same storage scheme as that used by ELSLV. This means that we have convenient access to the *columns* of  $L^T$ , and sparsity can be exploited completely, as discussed in Section 2.3.1, using an outer product form of the computation. The  $i$ -th column of  $L^T$  is used in the computation only if the  $i$ -th element of the solution is nonzero. Just as in ELSLV, the subroutine EUSLV can be used to solve upper triangular systems involving only a submatrix of  $L$  contained in the array pair (XENV, ENV), using techniques analogous to those we described above. The value of IBAND is appropriately adjusted for those columns of  $L^T$  that protrude outside the part of  $L$  actually being used.

All the subroutines which perform numerical computation contain a labelled COMMON block SPKOPS, which has a single variable OPS. Each subroutine counts the number of operations (multiplications and divisions) it performs, and increments the value of OPS accordingly. Thus, if the user of the subroutines wishes to monitor the number of operations performed, he can make the same common block declaration in his calling program and examine the value of OPS.

The variable OPS has been declared to be double precision to avoid the possibility of serious rounding error in the computation of operation counts. Our subroutines may be used to solve very large systems, so OPS may easily assume values as large as  $10^8$  or  $10^9$ , even though OPS may be incremented in

each subroutine by relatively small numbers. On many computers, if single precision is used, the floating point addition of a small number (say less than 10) to  $10^8$  will again yield  $10^8$ . (Try it, simulating 6 digit floating point arithmetic!) Using double precision for OPS makes serious rounding error in the operation count very unlikely.

---

```

1. C*****
2. C*****
3. C*****      EUSLV . . . . ENVELOPE UPPER SOLVE      *****
4. C*****
5. C*****
6. C
7. C      PURPOSE - THIS SUBROUTINE SOLVES AN UPPER TRIANGULAR
8. C      SYSTEM U X = RHS.  THE FACTOR U IS STORED IN THE
9. C      ENVELOPE FORMAT.
10. C
11. C      INPUT PARAMETERS -
12. C      NEQNS - NUMBER OF EQUATIONS.
13. C      (XENV, ENV) - ARRAY PAIR FOR THE ENVELOPE OF U.
14. C      DIAG - ARRAY FOR THE DIAGONAL OF U.
15. C
16. C      UPDATED PARAMETER -
17. C      RHS - ON INPUT, IT CONTAINS THE RIGHT HAND SIDE.
18. C      ON OUTPUT, IT CONTAINS THE SOLUTION VECTOR.
19. C      OPS - DOUBLE PRECISION VARIABLE CONTAINED IN THE
20. C      LABELLED COMMON BLOCK OPNS.  ITS VALUE IS
21. C      INCREASED BY THE NUMBER OF OPERATIONS
22. C      PERFORMED BY THIS SUBROUTINE.
23. C
24. C*****
25. C
26. C      SUBROUTINE EUSLV ( NEQNS, XENV, ENV, DIAG, RHS )
27. C
28. C*****
29. C
30. C      DOUBLE PRECISION COUNT, OPS
31. C      COMMON /SPKOPS/ OPS
32. C      REAL DIAG(1), ENV(1), RHS(1), S
33. C      INTEGER XENV(1), I, IBAND, K, KSTOP, KSTRT, L,
34. C      1          NEQNS
35. C
36. C*****
37. C
38. C      I = NEQNS + 1
39. C      100    I = I - 1
40. C      IF ( I .EQ. 0 ) RETURN
41. C      IF ( RHS(I) .EQ. 0.0E0 ) GO TO 100

```

```

42.          S = RHS(I)/DIAG(I)
43.          RHS(I) = S
44.          OPS = OPS + 1.0D0
45.          IBAND = XENV(I+1) - XENV(I)
46.          IF ( IBAND .GE. I ) IBAND = I - 1
47.          IF ( IBAND .EQ. 0 ) GO TO 100
48.          KSTRT = I - IBAND
49.          KSTOP = I - 1
50.          L = XENV(I+1) - IBAND
51.          DO 200 K = KSTRT, KSTOP
52.             RHS(K) = RHS(K) - S*XENV(L)
53.             L = L + 1
54. 200      CONTINUE
55.          COUNT = IBAND
56.          OPS = OPS + COUNT
57.          GO TO 100
58.          END

```

---

#### 4.6.2 The Factorization Subroutine ESFCT

In this section we describe some details about the numerical factorization subroutine **ESFCT**, which computes the Cholesky factorization  $\mathbf{L}\mathbf{L}^T$  of a given matrix  $\mathbf{A}$ , stored using the envelope storage scheme described in Section 4.5.1. The variant of Cholesky's method used is the bordering method (see Section 2.2.2).

Recall that if  $\mathbf{A}$  is partitioned as

$$\mathbf{A} = \begin{pmatrix} \mathbf{M} & \mathbf{u} \\ \mathbf{u}^T & s \end{pmatrix}$$

where  $\mathbf{M}$  is the leading principal submatrix of  $\mathbf{A}$  and  $\mathbf{L}_M\mathbf{L}_M^T$  is its Cholesky factorization, then the factor of  $\mathbf{A}$  is given by

$$\mathbf{L} = \begin{pmatrix} \mathbf{L}_M & \mathbf{0} \\ \mathbf{w}^T & t \end{pmatrix},$$

where  $\mathbf{L}_M\mathbf{w} = \mathbf{u}$  and  $t = (s - \mathbf{w}^T\mathbf{w})^{1/2}$ . Thus, the Cholesky factor of  $\mathbf{A}$  can be computed row by row, working with successively larger matrices  $\mathbf{M}$ , beginning with the one by one matrix  $a_{11}$ . The main point of interest in **ESFCT** concerns the exploitation of the fact that the vectors  $\mathbf{u}$  are “short” because we are dealing with an envelope matrix.

Referring to Figure 4.6.3, suppose the first  $i - 1$  steps of the factorization have been completed, so that the leading  $(i - 1) \times (i - 1)$  principal submatrix of  $\mathbf{A}$  has been factored. (Thus, the statements preceding the loop `D0 300 I = 2, . . .` have been executed, and the loop `D0 300 I = 2, . . .` has been executed  $i - 2$  times.) In order to compute row  $i$  of  $\mathbf{L}$ , we must solve the system of equations  $\mathbf{L}_M \mathbf{w} = \mathbf{u}$ .

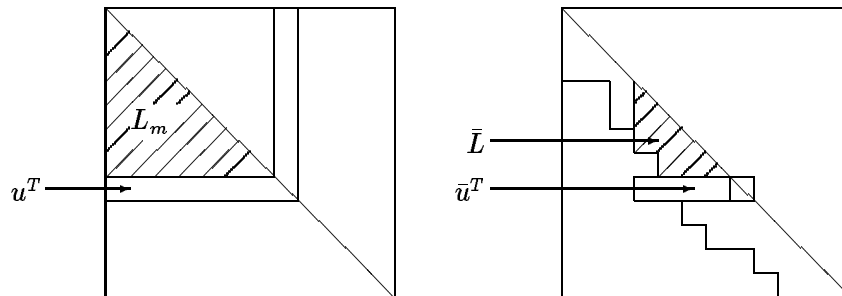


Figure 4.6.3: Sketch showing the way sparsity is exploited in ESFCT; only  $\bar{\mathbf{L}}$  enters into the computation of  $\bar{\mathbf{w}}$  from  $\bar{\mathbf{u}}$ .

However, it is clear from the picture (and from Lemmas 2.3.1 and 2.3.4) that only part of  $\mathbf{L}_M$  is involved in the computation, namely that part of  $\mathbf{L}_M$  labelled  $\bar{\mathbf{L}}$ . Thus ELSLV is called with the size of the triangular system specified as IBAND, the size of  $\bar{\mathbf{u}}$  (and  $\bar{\mathbf{L}}$ ), and IFIRST is the index in  $\mathbf{L}$  of the first row of  $\bar{\mathbf{L}}$ .

```

1. C*****
2. C*****
3. C*****   ESFCT . . . . ENVELOPE SYMMETRIC FACTORIZATION   ****
4. C*****
5. C*****
6. C
7. C   PURPOSE - THIS SUBROUTINE FACTORS A POSITIVE DEFINITE
8. C   MATRIX A INTO L*L(TRANPOSE). THE MATRIX A IS STORED
9. C   IN THE ENVELOPE FORMAT. THE ALGORITHM USED IN THE
10. C   STANDARD BORDERING METHOD.
11. C
12. C   INPUT PARAMETERS -
13. C   NEQNS - NUMBER OF EQUATIONS.
14. C   XENV - THE ENVELOPE INDEX VECTOR.
15. C

```

4.6. THE NUMERICAL SUBROUTINES ESFCT, ELSLV AND EUSLV 103

```

16. C      UPDATED PARAMETERS -
17. C          ENV - THE ENVELOPE OF L OVERWRITES THAT OF A.
18. C          DIAG - THE DIAGONAL OF L OVERWRITES THAT OF A.
19. C          IFLAG - THE ERROR FLAG. IT IS SET TO 1 IF A ZERO OR
20. C              NEGATIVE SQUARE ROOT IS DETECTED DURING THE
21. C              FACTORIZATION.
22. C
23. C      PROGRAM SUBROUTINES -
24. C          ELSLV.
25. C
26. C*****
27. C
28. C          SUBROUTINE ESFCT ( NEQNS, XENV, ENV, DIAG, IFLAG )
29. C
30. C*****
31. C
32. C          DOUBLE PRECISION COUNT, OPS
33. C          COMMON /SPKOPS/ OPS
34. C          REAL DIAG(1), ENV(1), S, TEMP
35. C          INTEGER XENV(1), I, IBAND, IFIRST, IFLAG, IXENV,
36. C              1      J, JSTOP, NEQNS
37. C
38. C*****
39. C
40. C          IF ( DIAG(1) .LE. 0.0E0 ) GO TO 400
41. C          DIAG(1) = SQRT(DIAG(1))
42. C          IF ( NEQNS .EQ. 1 ) RETURN
43. C          -----
44. C          LOOP OVER ROWS 2,3,..., NEQNS OF THE MATRIX ....
45. C          -----
46. C          DO 300 I = 2, NEQNS
47. C              IXENV = XENV(I)
48. C              IBAND = XENV(I+1) - IXENV
49. C              TEMP = DIAG(I)
50. C              IF ( IBAND .EQ. 0 ) GO TO 200
51. C              IFIRST = I - IBAND
52. C              -----
53. C              COMPUTE ROW I OF THE TRIANGULAR FACTOR.
54. C              -----
55. C              CALL ELSLV ( IBAND, XENV(IFIRST), ENV,
56. C              1      DIAG(IFIRST), ENV(IXENV) )
57. C              JSTOP = XENV(I+1) - 1
58. C              DO 100 J = IXENV, JSTOP
59. C                  S = ENV(J)
60. C                  TEMP = TEMP - S*S
61. C          100      CONTINUE
62. C          200      IF ( TEMP .LE. 0.0E0 ) GO TO 400

```



```

63.          DIAG(I) = SQRT(TEMP)
64.          COUNT = IBAND
65.          OPS = OPS + COUNT
66.    300    CONTINUE
67.          RETURN
68.  C      -----
69.  C      SET ERROR FLAG - NON POSITIVE DEFINITE MATRIX.
70.  C      -----
71.    400    IFLAG = 1
72.          RETURN
73.          END

```

---

### Exercises

- 4.6.1) Suppose  $\mathbf{A}$  has symmetric structure but  $\mathbf{A} \neq \mathbf{A}^T$ , and assume that Gaussian elimination applied to  $\mathbf{A}$  is numerically stable without pivoting. The bordering equations for factoring  $\mathbf{A}$ , analogous to those used by ESFCT in Section 4.6.2, are as follows.

$$\mathbf{A} = \begin{pmatrix} \mathbf{M} & \mathbf{v} \\ \mathbf{u}^T & s \end{pmatrix}$$

$$\mathbf{L} = \begin{pmatrix} \mathbf{L}\mathbf{M} & \mathbf{0} \\ \mathbf{w}^T & 1 \end{pmatrix} \quad \mathbf{U} = \begin{pmatrix} \mathbf{U}\mathbf{M} & \mathbf{g} \\ \mathbf{0} & t \end{pmatrix}$$

$$\mathbf{L}\mathbf{M}\mathbf{g} = \mathbf{v}, \quad \mathbf{U}^T\mathbf{M}\mathbf{w} = \mathbf{u}, \quad t = s - \mathbf{w}^T\mathbf{g}.$$

Here  $\mathbf{L}$  is now *unit* lower triangular (ones on the diagonal), and of course  $\mathbf{L} \neq \mathbf{U}^T$ .

- a) Using ELSLV as a base, implement a Fortran subroutine EL1SLV that solves unit lower triangular systems stored using the envelope storage scheme.
- b) Using ESFCT as a base, implement a Fortran subroutine EFCT that factors  $\mathbf{A}$  into  $\mathbf{LU}$ , where  $\mathbf{L}$  and  $\mathbf{U}^T$  are stored using the envelope scheme.
- c) What subroutines do you need to solve  $\mathbf{Ax} = \mathbf{b}$ , where  $\mathbf{A}$  is as described in this question? Hints:
  - i) Very few changes in ELSLV and ESFCT are required.
  - ii) Your implementation of EFCT should use EL1SLV and ELSLV.

- 4.6.2) Suppose  $L$  and  $b$  have the structure shown in Figure 4.6.4, where  $L$  is stored in the arrays XENV, ENV, and DIAG, as described in Section 4.5.1. How many arithmetic operations will ELSLV perform in solving  $Lx = b$ ? How many will EUSLV perform in solving  $L^T x = b$ ?

---

$$\begin{array}{c} \left[ \begin{array}{cccccccc} \times & & & & & & & \\ \times & \times & & & & & & \\ \times & \times & \times & & & & & \\ & & & \times & & & & \\ & & & \times & \times & & & \\ & & & & \times & \times & & \\ & & & & & \times & & \\ & & & & & & \times & \\ & & & & & & & \times \\ & & & & & & & & \times \\ & & \times & \times & & & & & \\ & & & \times & \times & \times & \times & & \\ \end{array} \right] \end{array} \quad \begin{array}{c} \left[ \begin{array}{c} \times \\ \\ \\ \times \\ \\ \times \\ \\ \times \\ \end{array} \right] \\ \\ \\ \\ \\ \\ \\ \\ \\ \end{array}$$

$L \qquad b$

Figure 4.6.4: An example of a sparse triangular system.

---

## 4.7 Additional Notes

Our lack of enthusiasm for band orderings is due in part to the fact that we only consider “in core” methods in our book. Band orderings are attractive if auxiliary storage is to be used, since it is quite easy to implement factorization and solution subroutines which utilize auxiliary storage, provided about  $\beta(\beta + 1)/2$  main storage locations are available (Felippa [16]). Wilson et al. [55] describe an out-of-core band-oriented scheme which requires even less storage; their program can execute even if there is only enough storage to hold two columns of the band of  $L$ . Another context in which band orderings are important is in the use of so-called minimal storage band methods (Sherman [47]). The basic computational scheme is similar to those which use auxiliary storage, except that the columns of  $L$  are computed, used, and then “thrown away,” instead of being written on auxiliary storage. The parts of  $L$  needed later are recomputed. Several other algorithms for producing low profile orderings have been pro-

posed. Levy [35] describes an algorithm which picks nodes to number on the basis of minimum increase in the envelope size. King [33] has proposed a similar scheme, except that the candidates for numbering are restricted to those having at least one numbered neighbor, and therefore requires a starting node. More recently, several algorithms more closely related to the one described in this chapter have been proposed (Gibbs et al. [30], Gibbs [29]).

Several researchers have described “frontal” or “wavefront” techniques to exploit the variation in the bandwidth when using auxiliary storage (Melosh and Bamford [41], Irons [31]). These schemes require only about  $\omega(\omega + 1)/2$  main storage locations rather than  $\beta(\beta + 1)/2$  for the band schemes, although the programs tend to be substantially more complicated as a result. These ideas have been proposed in the context of solving finite element equations, and a second novel feature the methods have is that the equations are generated and solved *in tandem*.

It has been shown that given a starting node, the RCM algorithm can be implemented to run in  $O(|E|)$  time (Chan and George [8]). Since each edge of the graph must be examined at least once, this new method is apparently optimal.

A set of subroutines which are similar to **ELSLV**, **EUSLV**, and **ESFCT** is provided in Eisenstat et al. [13].

## Chapter 5

# General Sparse Methods

### 5.1 Introduction

In this chapter we consider methods which, unlike those of Chapter 4, attempt to exploit all the zero elements in the triangular factor  $\mathbf{L}$  of  $\mathbf{A}$ . The ordering algorithm we study in this chapter is called the *minimum degree algorithm* (Rose [44]). It is a heuristic algorithm for finding an ordering for  $\mathbf{A}$  which suffers low fill when it is factored. This algorithm has been used widely in industrial applications, and enjoys a good reputation. The computer implementations of the allocation and numerical subroutines are adapted from those of the Yale Sparse Matrix Package (Eisenstat [13]).

### 5.2 Symmetric Factorization

Let  $\mathbf{A}$  be a symmetric sparse matrix. The *nonzero structure* of  $\mathbf{A}$  is defined by

$$\text{Nonz}(\mathbf{A}) = \{\{i, j\} \mid a_{ij} \neq 0 \text{ and } i \neq j\}.$$

Suppose the matrix is factored into  $\mathbf{L}\mathbf{L}^T$  using the Cholesky factorization algorithm. The *filled matrix*  $\mathbf{F}(\mathbf{A})$  of  $\mathbf{A}$  is the matrix sum  $\mathbf{L} + \mathbf{L}^T$ . When the matrix under study is clear from context, we use  $\mathbf{F}$  rather than  $\mathbf{F}(\mathbf{A})$ . Its corresponding structure is then

$$\text{Nonz}(\mathbf{F}) = \{\{i, j\} \mid l_{ij} \neq 0 \text{ and } i \neq j\}.$$

Recall that throughout our book, we assume that exact numerical cancellation does not occur, so for a given nonzero structure  $\text{Nonz}(\mathbf{A})$ , the corre-

sponding  $Nonz(\mathbf{F})$  is completely determined. That is,  $Nonz(\mathbf{F})$  is independent of the numerical quantities in  $\mathbf{A}$ .

This no-cancellation assumption immediately implies that

$$Nonz(\mathbf{A}) \subset Nonz(\mathbf{F}),$$

and the *fill* of the matrix  $\mathbf{A}$  can then be defined as

$$Fill(\mathbf{A}) = Nonz(\mathbf{F}) - Nonz(\mathbf{A}).$$

For example, consider the matrix in Figure 5.2.1, where fill-in entries are indicated by  $+$ . The corresponding sets are given by

$$\begin{aligned} Nonz(\mathbf{A}) &= \{\{1, 5\}, \{1, 8\}, \{2, 4\}, \{2, 5\}, \{3, 8\}, \{4, 7\}, \{5, 6\}, \{6, 8\}, \{8, 9\}\} \\ Fill(\mathbf{A}) &= \{\{4, 5\}, \{5, 7\}, \{5, 8\}, \{6, 7\}, \{7, 8\}\}. \end{aligned}$$

In the next section, we shall consider how  $Fill(\mathbf{A})$  can be obtained from  $Nonz(\mathbf{A})$ .

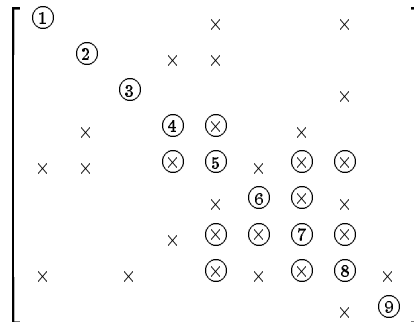


Figure 5.2.1: A matrix example of  $Nonz$  and  $Fill$ .

### 5.2.1 Elimination Graph Model

We now relate the application of symmetric Gaussian elimination to  $\mathbf{A}$ , to corresponding changes in its graph  $\mathcal{G}^{\mathbf{A}}$ . Recall from Chapter 2 that the first step of the outer product version of the algorithm applied to an  $n \times n$

symmetric positive definite matrix  $\mathbf{A} = \mathbf{A}_0$  can be described by the equation:

$$\begin{aligned} \mathbf{A} &= \mathbf{A}_0 = \mathbf{H}_0 = \begin{pmatrix} d_1 & \mathbf{v}_1^T \\ \mathbf{v}_1 & \bar{\mathbf{H}}_1 \end{pmatrix} \\ &= \begin{pmatrix} \sqrt{d_1} & \mathbf{0} \\ \frac{\mathbf{v}_1}{\sqrt{d_1}} & \mathbf{I}_{n-1} \end{pmatrix} \begin{pmatrix} 1 & \mathbf{0} \\ \mathbf{0} & \mathbf{H}_1 \end{pmatrix} \begin{pmatrix} \sqrt{d_1} & \frac{\mathbf{v}_1^T}{\sqrt{d_1}} \\ \mathbf{0} & \mathbf{I}_{n-1} \end{pmatrix} \\ &= \mathbf{L}_1 \mathbf{A}_1 \mathbf{L}_1^T, \end{aligned} \quad (5.2.1)$$

where

$$\mathbf{H}_1 = \bar{\mathbf{H}}_1 - \frac{\mathbf{v}_1 \mathbf{v}_1^T}{d_1}. \quad (5.2.2)$$

The basic step is then recursively applied to  $\mathbf{H}_1$ ,  $\mathbf{H}_2$ , and so on. Making the usual assumption that exact cancellation does not occur, equation (5.2.2) implies that the  $jk$ -th entry of  $\mathbf{H}_1$  is nonzero if the corresponding entry in  $\bar{\mathbf{H}}_1$  is already nonzero, or if *both*  $(\mathbf{v}_1)_j \neq 0$  and  $(\mathbf{v}_1)_k \neq 0$ . Of course both situations may prevail, but when only the latter one does, some *fill-in* occurs. This phenomenon is illustrated pictorially in Figure 5.2.2. After the first step of the factorization is completed, we are left with the matrix  $\mathbf{H}_1$  to factor.

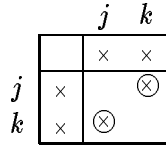


Figure 5.2.2: Pictorial illustration of fill-in in the outer-product formulation.

---

Following Parter [43] and Rose [44], we now establish a correspondence between the transformation of  $\mathbf{H}_0$  to  $\mathbf{H}_1$  and the corresponding changes to their respective graphs. As usual, we denote the graphs of  $\mathbf{H}_0 (= \mathbf{A})$  and  $\mathbf{H}_1$  by  $\mathcal{G}^{\mathbf{H}_0}$  and  $\mathcal{G}^{\mathbf{H}_1}$  respectively, and for convenience we denote the node  $\alpha(i)$  by  $x_i$ , where  $\alpha$  is the labelling of  $\mathcal{G}^{\mathbf{A}}$  implied by  $\mathbf{A}$ . Now as shown in the example of Figure 5.2.3, the graph of  $\mathbf{H}_1$  is obtained from that of  $\mathbf{H}_0$  by:

- 1) deleting node  $x_1$  and its incident edges

- 2) adding edges to the graph so that nodes in  $Adj(x_1)$  are pairwise adjacent in  $\mathcal{G}^{\mathbf{H}_1}$ .

The recipe is due to Parter [43].

Thus, as observed by Rose, symmetric Gaussian elimination can be interpreted as generating a sequence of *elimination graphs*

$$\mathcal{G}_i^\alpha = \mathcal{G}^{\mathbf{H}_i} = (X_i^\alpha, E_i^\alpha), \quad i = 1, 2, \dots, n-1,$$

where  $\mathcal{G}_i^\alpha$  is obtained from  $\mathcal{G}_{i-1}^\alpha$  according to the procedure described above. When  $\alpha$  is clear from context, we use  $\mathcal{G}_i$  instead of  $\mathcal{G}_i^\alpha$ . The example in Figure 5.2.3 illustrates this vertex elimination operation. The darker lines depict edges added during the factorization. For example, the elimination of the node  $x_2$  in the graph  $\mathcal{G}_1$  generates three fill-in edges  $\{x_3, x_4\}$ ,  $\{x_4, x_6\}$ ,  $\{x_3, x_6\}$  in  $\mathcal{G}_2$  since  $\{x_3, x_4, x_6\}$  is the adjacent set of  $x_2$  in  $\mathcal{G}_1$ .

Let  $\mathbf{L}$  be the triangular factor of the matrix  $\mathbf{A}$ . Define the filled graph of  $\mathcal{G}^{\mathbf{A}}$  to be the symmetric graph  $\mathcal{G}^{\mathbf{F}} = (X^{\mathbf{F}}, E^{\mathbf{F}})$ , where  $\mathbf{F} = \mathbf{L} + \mathbf{L}^T$ . Here the edge set  $E^{\mathbf{F}}$  consists of all the edges in  $E^{\mathbf{A}}$  together with all the edges added during the factorization. Obviously,  $X^{\mathbf{F}} = X^{\mathbf{A}}$ . The edge sets  $E^{\mathbf{F}}$  and  $E^{\mathbf{A}}$  are related by the following lemma due to Parter [43]. Its proof is left as an exercise.

**Lemma 5.2.1** *The unordered pair  $\{x_i, x_j\} \in E^{\mathbf{F}}$  if and only if  $\{x_i, x_j\} \in E^{\mathbf{A}}$  or  $\{x_i, x_k\} \in E^{\mathbf{F}}$  and  $\{x_k, x_j\} \in E^{\mathbf{F}}$  for some  $k < \min\{i, j\}$ .*

The notion of elimination graphs allows us to interpret the step by step elimination process as a sequence of graph transformations. Moreover, the set of edges added in the elimination graphs corresponds to the set of fill-ins. Thus, for the example in Figure 5.2.3, the structures of the corresponding matrix  $\mathbf{F} = \mathbf{L} + \mathbf{L}^T$  and the filled graph  $\mathcal{G}^{\mathbf{F}}$  are given in Figure 5.2.4.

Note that the filled graph  $\mathcal{G}^{\mathbf{F}}$  can easily be constructed from the sequence of elimination graphs. Finding  $\mathcal{G}^{\mathbf{F}}$  is important because it contains the structure of  $\mathbf{L}$ . We need to know it if we intend to use a storage scheme which exploits all the zeros in  $\mathbf{L}$ .

## 5.2.2 Modelling Elimination By Reachable Sets

Section 5.2.1 defines the sequence of elimination graphs

$$\mathcal{G}_0 \rightarrow \mathcal{G}_1 \rightarrow \dots \rightarrow \mathcal{G}_{n-1}$$

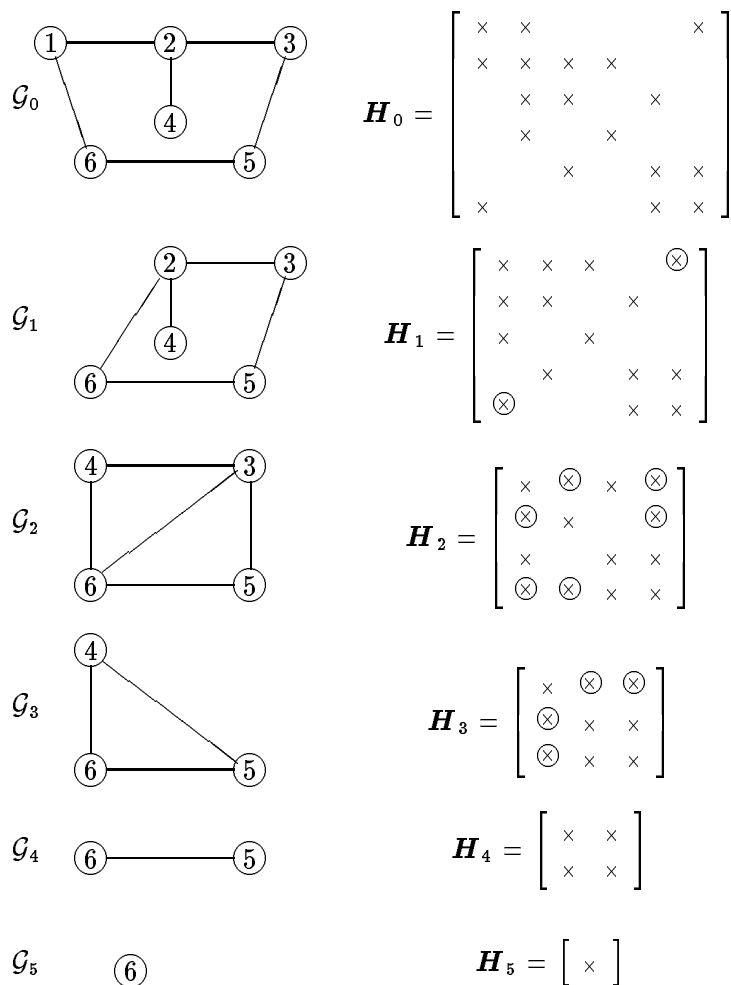


Figure 5.2.3: The sequence of elimination graphs.



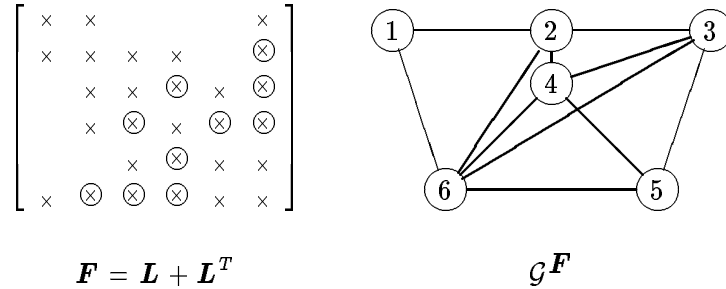


Figure 5.2.4: The filled graph and matrix of the example on Figure 5.2.3.

and provides a recursive characterization of the edge set  $E^F$ . It is often helpful, both in theoretical and computational terms, to have characterizations of  $\mathcal{G}_i$  and  $E^F$  directly in terms of the *original* graph  $\mathcal{G}^A$ . Our objective in this section is to provide such characterizations using the notion of *reachable sets*.

Let us first study the way the fill edge  $\{x_4, x_6\}$  is formed in the example of Figure 5.2.3. In  $\mathcal{G}_1$ , there is the path

$$(x_4, x_2, x_6),$$

so that when  $x_2$  is eliminated, the edge  $\{x_4, x_6\}$  is created. However, the edge  $\{x_2, x_6\}$  is not present in the original graph; it is formed from the path

$$(x_2, x_1, x_6)$$

when  $x_1$  is eliminated from  $\mathcal{G}_0$ . On combining the two, we see that the path  $(x_4, x_2, x_1, x_6)$  in the original graph is really responsible for the filled edge  $\{x_4, x_6\}$ . This motivates the use of reachable sets, which we now introduce (George [27]).

Let  $S$  be a subset of the node set with  $x \notin S$ . The node  $x$  is said to be *reachable from a node  $y$  through  $S$*  if there exists a path  $(y, v_1, \dots, v_k, x)$  from  $y$  to  $x$  such that  $v_i \in S$  for  $1 \leq i \leq k$ . Note that  $k$  can be zero, so that any adjacent node of  $y$  not in  $S$  is reachable from  $y$  through  $S$ .

The reachable set of  $y$  through  $S$ , denoted by  $Reach(y, S)$ , is then defined to be

$$Reach(y, S) = \{x \notin S \mid x \text{ is reachable from } y \text{ through } S\}. \quad (5.2.3)$$

To illustrate the notion of reachable sets, we consider the example in Figure 5.2.5. If  $S = \{s_1, s_2, s_3, s_4\}$ , we have

$$\text{Reach}(y, S) = \{a, b, c\},$$

since we can find the following paths through  $S$  :

$$(y, s_2, s_4, a),$$

$$(y, b),$$

$$(y, s_1, c).$$

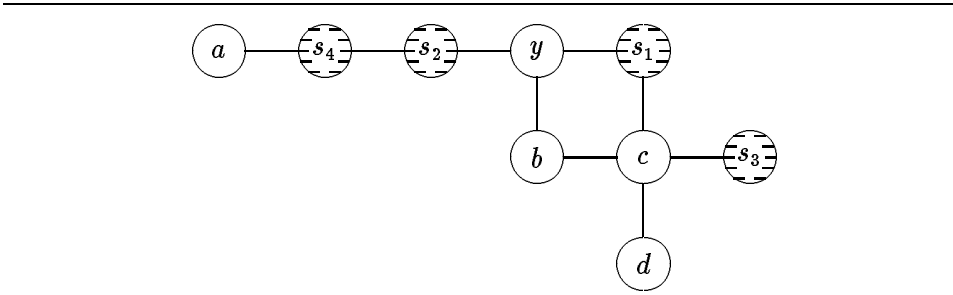


Figure 5.2.5: Example to illustrate the reachable set concept.

---

The following theorem characterizes the filled graph by reachable sets.

**Theorem 5.2.2**

$$E^{\mathbf{F}} = \{\{x_i, x_j\} \mid x_j \in \text{Reach}(x_i, \{x_1, x_2, \dots, x_{i-1}\})\}.$$

**Proof:** Assume  $x_j \in \text{Reach}(x_i, \{x_1, \dots, x_{i-1}\})$ . By definition, there exists a path  $(x_i, y_1, \dots, y_t, x_j)$  in  $\mathcal{G}^{\mathbf{A}}$  with  $y_k \in \{x_1, \dots, x_{i-1}\}$  for  $1 \leq k \leq t$ . If  $t = 0$  or  $t = 1$ , the result follows immediately from Lemma 5.2.1. If  $t > 1$ , a simple induction on  $t$ , together with Lemma 5.2.1 shows that  $\{x_i, x_j\} \in E^{\mathbf{F}}$ . Conversely, assume  $\{x_i, x_j\} \in E^{\mathbf{F}}$  and  $i \leq j$ . The proof is by induction on the subscript  $i$ . The result is true for  $i = 1$ , since  $\{x_i, x_j\} \in E^{\mathbf{F}}$  implies  $\{x_i, x_j\} \in E^{\mathbf{A}}$ . Suppose the assertion is true for subscripts less than  $i$ . If  $\{x_i, x_j\} \in E^{\mathbf{A}}$ , there is nothing to prove. Otherwise, by Lemma 5.2.1, there exists a  $k < \min\{i, j\}$  such that  $\{x_i, x_k\} \in E^{\mathbf{F}}$  and  $\{x_j, x_k\} \in E^{\mathbf{F}}$ . By the inductive assumption, a path can be found from  $x_i$  to  $x_j$  passing

through  $x_k$  in the section graph  $\mathcal{G}^{\mathbf{A}}(\{x_1, \dots, x_k\} \cup \{x_i, x_j\})$  which implies that  $x_j \in \text{Reach}(x_i, \{x_1, \dots, x_{i-1}\})$ .  $\square$

In terms of the matrix, the set  $\text{Reach}(x_i, \{x_1, \dots, x_{i-1}\})$  is simply the set of row subscripts that correspond to nonzero entries in the column vector  $\mathbf{L}_{*i}$ . For example, let the graph of Figure 5.2.5 be ordered as shown in Figure 5.2.6.

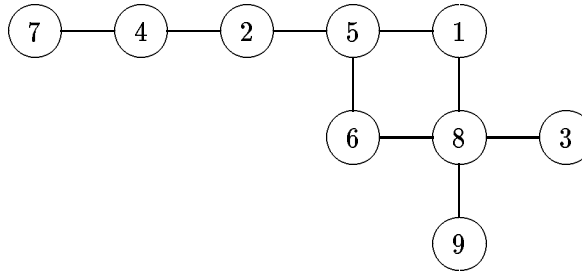


Figure 5.2.6: A labelling of the graph of Figure 5.2.5.

---

If  $S_i = \{x_1, \dots, x_i\}$ , it is not difficult to see from the definition of reachable set that

$$\begin{aligned}
 \text{Reach}(x_1, S_0) &= \{x_5, x_8\} \\
 \text{Reach}(x_2, S_1) &= \{x_4, x_5\} \\
 \text{Reach}(x_3, S_2) &= \{x_8\} \\
 \text{Reach}(x_4, S_3) &= \{x_5, x_7\} \\
 \text{Reach}(x_5, S_4) &= \{x_6, x_7, x_8\} \\
 \text{Reach}(x_6, S_5) &= \{x_7, x_8\} \\
 \text{Reach}(x_7, S_6) &= \{x_8\} \\
 \text{Reach}(x_8, S_7) &= \{x_9\} \\
 \text{Reach}(x_9, S_8) &= \phi.
 \end{aligned}$$

It then follows from Theorem 5.2.2 that the structure of the corresponding  $\mathbf{L}$  is given by the matrix in Figure 5.2.7.

We have thus characterized the structure of  $\mathbf{L}$  directly in terms of the structure of  $\mathbf{A}$ . More importantly, there is a convenient way of characterizing the elimination graphs introduced in Section 5.2.1 in terms of reachable sets. Let  $\mathcal{G}_0, \mathcal{G}_1, \dots, \mathcal{G}_{n-1}$  be the sequence of elimination graphs as defined by the nodes  $x_1, x_2, \dots, x_n$ , and consider the graph  $\mathcal{G}_i = (X_i, E_i)$ . We then have

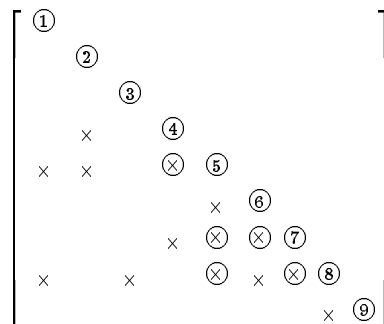


Figure 5.2.7: Structure of a Cholesky factor  $L$ .

**Theorem 5.2.3** Let  $y$  be a node in the elimination graph  $\mathcal{G}_i = (X_i, E_i)$ . The set of nodes adjacent to  $y$  in  $\mathcal{G}_i$  is given by

$$Reach(y, \{x_1, \dots, x_i\})$$

where the  $Reach$  operator is applied to the original graph  $\mathcal{G}_0$ .

**Proof:** The proof can be done by induction on  $i$ . □

Let us re-examine the example in Figure 5.2.3. Consider the graphs  $\mathcal{G}_0$  and  $\mathcal{G}_2$ .

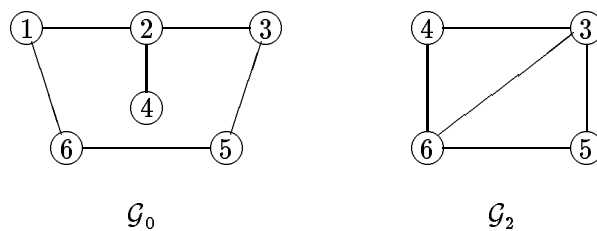


Figure 5.2.8: The graphs  $\mathcal{G}_0$  and  $\mathcal{G}_2$ .

Let  $S_2 = \{x_1, x_2\}$ . It is clear that

$$\begin{aligned} Reach(x_3, S_2) &= \{x_4, x_5, x_6\}, \\ Reach(x_4, S_2) &= \{x_3, x_6\}, \\ Reach(x_5, S_2) &= \{x_3, x_6\}, \end{aligned}$$

and

$$\text{Reach}(x_6, S_2) = \{x_3, x_4, x_5\},$$

since we have paths

$$(x_3, x_2, x_4),$$

$$(x_3, x_2, x_1, x_6),$$

and

$$(x_4, x_2, x_1, x_6)$$

in the graph  $\mathcal{G}_0$ . These reach sets are precisely the adjacent sets in the graph  $\mathcal{G}_2$ .

The importance of reachable sets in sparse elimination lies in Theorem 5.2.3. Given a graph  $\mathcal{G} = (X, E)$  and an elimination sequence  $x_1, x_2, \dots, x_n$ , the whole elimination process can be described implicitly by this sequence and the *Reach* operator. This can be regarded as an *implicit model* for elimination, as opposed to the *explicit model* using elimination graphs (Section 5.2.1).

### Exercises

5.2.1) For any nonzero structure  $\text{Nonz}(\mathbf{A})$ , can you always find a matrix  $\mathbf{A}^*$  so that its filled matrix  $\mathbf{F}^*$  has identical *logical* and *numerical* nonzero structures? Why?

5.2.2) Consider the star graph with 7 nodes (Figure 4.3.3). Assuming that the centre node is numbered first, determine the sequence of elimination graphs.

5.2.3) For a given labelled graph  $\mathcal{G}^{\mathbf{A}} = (X^{\mathbf{A}}, E^{\mathbf{A}})$ , show that

$$\text{Reach}(x_i, \{x_1, x_2, \dots, x_{i-1}\}) \subset \text{Adj}(\{x_1, x_2, \dots, x_i\}),$$

and hence conclude that  $\text{Fill}(\mathbf{A}) \subset \text{Env}(\mathbf{A})$ .

5.2.4) Show that the section graph

$$\mathcal{G}^{\mathbf{A}}(\text{Reach}(x_i, \{x_1, \dots, x_{i-1}\}) \cup \{x_i\})$$

is a clique in the filled graph  $\mathcal{G}^{\mathbf{F}}$ .

### 5.3. COMPUTER REPRESENTATION OF ELIMINATION GRAPHS 117

5.2.5) (Rose [44]) A graph is *triangulated* if for every cycle  $(x_1, x_2, \dots, x_l, x_1)$  of length  $l > 3$ , there is an edge joining two non-consecutive vertices in the cycle. (Such an edge is called a *chord* of the cycle.) Show that the following conditions are equivalent.

- a) the graph  $\mathcal{G}^{\mathbf{A}}$  is triangulated
- b) there exists a permutation matrix  $\mathbf{P}$  such that

$$\text{Fill}(\mathbf{P}\mathbf{A}\mathbf{P}^T) = \phi$$

5.2.6) Show that the graph  $\mathcal{G}^{\mathbf{F}(\mathbf{A})}$  is triangulated. Give a permutation  $\mathbf{P}$  such that  $\text{Fill}(\mathbf{P}\mathbf{F}(\mathbf{A})\mathbf{P}^T) = \phi$ . Hence, or otherwise, show that  $\text{Nonz}(\mathbf{F}(\mathbf{A})) = \text{Nonz}(\mathbf{F}(\mathbf{F}(\mathbf{A})))$ .

5.2.7) Let  $S \subset T$  and  $y \notin T$ . Show that

$$\text{Reach}(y, S) \subset \text{Reach}(y, T) \cup T.$$

5.2.8) Let  $y \notin S$ . Define the *neighborhood set* of  $y$  in  $S$  to be

$$\text{Nbrhd}(y, S) =$$

$$\{s \in S \mid s \text{ is reachable from } y \text{ through a subset of } S\}.$$

Let  $x \notin S$ . Show that, if

$$\text{Adj}(x) \subset \text{Reach}(y, S) \cup \text{Nbrhd}(y, S) \cup \{y\},$$

then

- a)  $\text{Nbrhd}(x, S) \subset \text{Nbrhd}(y, S)$
- b)  $\text{Reach}(x, S) \subset \text{Reach}(y, S) \cup \{y\}$ .

5.2.9) Prove Theorem 5.2.3.

## 5.3 Computer Representation of Elimination Graphs

As discussed in Section 5.2, Gaussian elimination on a sparse symmetric linear system can be modelled by the sequence of elimination graphs. In this section, we study the representation and transformation of elimination graphs on a computer. These issues are important in the implementation of general sparse methods.

### 5.3.1 Explicit and Implicit Representations

Elimination graphs are, after all, symmetric graphs so that they can be represented explicitly using one of the storage schemes described in Section 3.3. However, what concerns us is that the implementation should be tailored for elimination, so that the transformation from one elimination graph to the next in the sequence can be performed easily.

Let us review the transformation steps. Let  $\mathcal{G}_i$  be the elimination graph obtained from eliminating the node  $x_i$  from  $\mathcal{G}_{i-1}$ . The adjacency structure of  $\mathcal{G}_i$  can be obtained as follows.

**Step 1** Determine the adjacent set  $Adj_{\mathcal{G}_{i-1}}(x_i)$  in  $\mathcal{G}_{i-1}$ .

**Step 2** Remove the node  $x_i$  and its adjacent list from the adjacency structure.

**Step 3** For each node  $y \in Adj_{\mathcal{G}_{i-1}}(x_i)$ , the new adjacent set of  $y$  in  $\mathcal{G}_i$  is given by merging the subsets

$$Adj_{\mathcal{G}_{i-1}}(y) - \{x_i\} \text{ and } Adj_{\mathcal{G}_{i-1}}(x_i) - \{y\}.$$

The above is an algorithmic formulation of the recipe by Parter (Section 5.2.1) to effect the transformation. There are two points that should be mentioned about the implementation. First, the space used to store  $Adj_{\mathcal{G}_{i-1}}(x_i)$  in the adjacency structure can be re-used after Step 2. Secondly, the *explicit* adjacency structure of  $\mathcal{G}_i$  may require much more space than that of  $\mathcal{G}_{i-1}$ . For example, in the star graph of  $n$  nodes (Figure 4.3.3), if the centre node is to be numbered first and  $\mathcal{G}_0 = (X_0, E_0)$  and  $\mathcal{G}_1 = (X_1, E_1)$  are the corresponding elimination graphs, it is easy to show that (see Exercise 5.2.2 on page 116 )

$$|E_0| = O(n)$$

and

$$|E_1| = O(n^2).$$

In view of these observations a very flexible data structure has to be used in the explicit implementation to allow for the dynamic change in the structure of the elimination graphs. The *adjacency linked list* structure described in Section 3.3 is a good candidate.

Any explicit computer representation has two disadvantages. First, the flexibility in the data structure often requires significant overhead in storage and execution time. Secondly, the maximum amount of storage required is

*unpredictable*. Enough storage is needed for the largest elimination graph  $\mathcal{G}_i$  that occurs. (Here “largest” refers to the number of edges, rather than the number of nodes.) This may exceed greatly the storage requirement for the original  $\mathcal{G}_0$ . Furthermore, this maximum storage requirement is not known until the end of the entire elimination process.

Theorem 5.2.3 provides another way to represent elimination graphs. They can be stored *implicitly* using the original graph  $\mathcal{G}$  and the eliminated subset  $S_i$ . The set of nodes adjacent to  $y$  in  $\mathcal{G}_i$  can then be retrieved by generating the reachable set  $Reach(y, S_i)$  in the original graph. This implicit representation does not have any of the disadvantages of the explicit method. It has a small and predictable storage requirement and it preserves the adjacency structure of the given graph.

However, the amount of work required to determine reachable sets can be intolerably large, especially at the later stages of elimination when  $|S_i|$  is large. In the next section, we shall consider another model which is more suitable for computer implementation, but still retains many of the advantages of using reachable sets.

### 5.3.2 Quotient Graph Model

Let us first consider elimination on the graph given in Figure 5.2.6. After the elimination of the nodes  $x_1, x_2, x_3, x_4, x_5$  the corresponding elimination graph is given in Figure 5.3.1. Shaded nodes are those that have been eliminated.

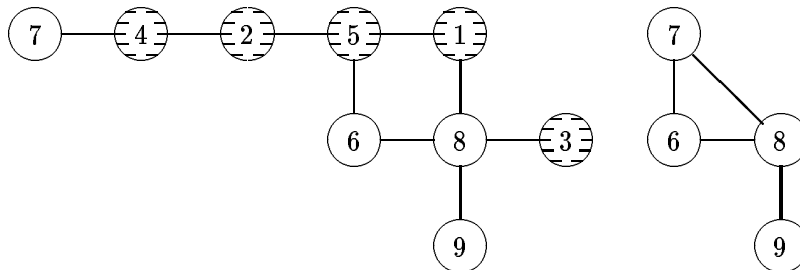


Figure 5.3.1: A graph example and its elimination graph.

---

Let  $S = \{x_1, x_2, x_3, x_4, x_5\}$ . In the implicit model, to discover that  $x_6 \in Reach(x_7, S)$ , the path

$$(x_7, x_4, x_2, x_5, x_6)$$



has to be traversed. Similarly,  $x_8 \in \text{Reach}(x_7, S)$  because of the path

$$(x_7, x_4, x_2, x_5, x_1, x_8).$$

Note that the lengths of the two paths are 4 and 5 respectively. We make two observations:

- a) the amount of work to generate reachable sets can be reduced if the lengths of paths to uneliminated nodes are shortened.
- b) if these paths are shortened to the extreme case, we get the explicit elimination graphs which have undesirable properties as mentioned in the previous section.

We look for a compromise. By coalescing connected eliminated nodes, we obtain a new graph structure that serves our purpose. For example, in Figure 5.3.1, there are two connected components in the graph  $\mathcal{G}(S)$ , whose node sets are

$$\{x_1, x_2, x_4, x_5\} \text{ and } \{x_3\}.$$

By forming two ‘‘supernodes,’’ we obtain the graph as given in Figure 5.3.2.

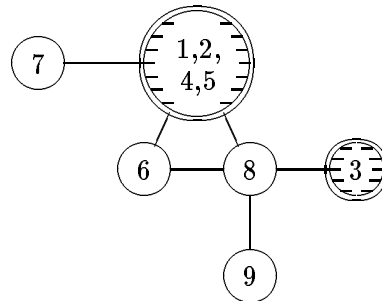


Figure 5.3.2: Graph formed by coalescing connected eliminated nodes.

---

For convenience, we set  $\bar{x}_5 = \{x_1, x_2, x_4, x_5\}$  and  $\bar{x}_3 = \{x_3\}$  to denote these connected components in  $S$ . With this new graph, we note that the paths

$$(x_7, \bar{x}_5, x_6)$$

and

$$(x_7, \bar{x}_5, x_8)$$

### 5.3. COMPUTER REPRESENTATION OF ELIMINATION GRAPHS 121

are of length *two* and they lead us from the node  $x_7$  to  $x_6$  and  $x_8$  respectively. In general, if we adopt this strategy all such paths are of length less than or equal to two. This has the obvious advantage over the reachable set approach on the original graph, where paths can be of arbitrary lengths (less than  $n$ ). What is then its advantage over the explicit elimination graph approach? In the next section we shall show that this approach can be implemented *in-place*; that is, it requires no more space than the original graph structure. In short, this new graph structure can be used to generate reachable sets (or adjacent sets in the elimination graph) quite efficiently and yet it requires a fixed amount of storage.

To formalize this model for elimination, we introduce the notion of *quotient graphs*. Let  $\mathcal{G} = (X, E)$  be a given graph and let  $\mathcal{P}$  be a partition on its node set  $X$  :

$$\mathcal{P} = \{Y_1, Y_2, \dots, Y_p\}.$$

That is,  $\bigcup_k Y_k = X$  and  $Y_i \cap Y_j = \emptyset$  for  $i \neq j$ . We define the *quotient graph* of  $\mathcal{G}$  with respect to  $\mathcal{P}$  to be the graph  $(\mathcal{P}, \mathcal{E})$ , where  $\{Y_i, Y_j\} \in \mathcal{E}$  if and only if  $Adj(Y_i) \cap Y_j \neq \emptyset$ . Often, we denote this graph by  $\mathcal{G}/\mathcal{P}$ .

For example, the graph in Figure 5.3.2 is the quotient graph of the one in Figure 5.3.1 with respect to the partitioning

$$\{x_1, x_2, x_4, x_5\}, \{x_3\}, \{x_6\}, \{x_7\}, \{x_8\}, \{x_9\}.$$

The notion of quotient graphs will be treated in more detail in Chapter 6 where partitioned matrices are considered. Here, we study its role in modelling elimination. The new model represents the elimination process as a sequence of quotient graphs.

Let  $\mathcal{G} = (X, E)$  be a given graph and consider a stage in the elimination where  $S$  is the set of eliminated nodes. We now associate a quotient graph with respect to this set  $S$  as motivated by the example in Figure 5.3.2. Define the set

$$\mathcal{C}(S) = \{C \subset S \mid \mathcal{G}(C) \text{ is a connected component in the subgraph } \mathcal{G}(S)\}, \quad (5.3.1)$$

and the partitioning on  $X$ ,

$$\bar{\mathcal{C}}(S) = \{\{y\} \mid y \in X - S\} \cup \mathcal{C}(S). \quad (5.3.2)$$

This uniquely defines the quotient graph

$$\mathcal{G}/\bar{\mathcal{C}}(S),$$

which can be viewed as the graph obtained by coalescing connected sets in  $S$ . Figure 5.3.2 is the resulting quotient graph for  $S = \{x_1, x_2, x_3, x_4, x_5\}$ . We now study the relevance of quotient graphs in elimination. Let  $x_1, x_2, \dots, x_n$  be the sequence of node elimination in the given graph  $\mathcal{G}$ . As before, let

$$S_i = \{x_1, x_2, \dots, x_i\}, \quad 1 \leq i \leq n.$$

For each  $i$ , the subset  $S_i$  induces the partitioning  $\bar{\mathcal{C}}(S_i)$  and the corresponding quotient graph

$$\mathcal{G}_i = \mathcal{G}/\bar{\mathcal{C}}(S_i) = (\bar{\mathcal{C}}(S_i), \mathcal{E}_i). \quad (5.3.3)$$

In this way, we obtain a sequence of quotient graphs

$$\mathcal{G}_1 \rightarrow \mathcal{G}_2 \rightarrow \dots \rightarrow \mathcal{G}_n$$

from the node elimination sequence. Figure 5.3.3 shows the sequence for the graph example of Figure 5.3.1. For notational convenience, we use  $y$  instead of  $\{y\}$  for such “supernodes” in  $\bar{\mathcal{C}}(S_i)$ .

The following theorem shows that quotient graphs of the form (5.3.3) are indeed representations of elimination graphs.

**Theorem 5.3.1** For  $y \in X - S_i$ ,

$$Reach_{\mathcal{G}}(y, S_i) = Reach_{\mathcal{G}_i}(y, \mathcal{C}(S_i)).$$

**Proof:** Consider  $u \in Reach_{\mathcal{G}}(y, S_i)$ . If the nodes  $y$  and  $u$  are adjacent in  $\mathcal{G}$ , so are  $y$  and  $u$  in  $\mathcal{G}_i$ . Otherwise, there exists a path

$$(y, s_1, \dots, s_t, u)$$

in  $\mathcal{G}$  where  $\{s_1, \dots, s_t\} \subset S_i$ . Let  $\mathcal{G}(C)$  be the connected component in  $\mathcal{G}(S_i)$  containing  $\{s_1\}$ . Then we have the path

$$(y, C, u)$$

in  $\mathcal{G}_i$  so that  $u \in Reach_{\mathcal{G}_i}(y, \mathcal{C}(S_i))$ .

Conversely, consider any  $u \in Reach_{\mathcal{G}_i}(y, \mathcal{C}(S_i))$ . There exists a path

$$(y, C_1, \dots, C_t, u)$$

in  $\mathcal{G}_i$  where  $\{C_1, \dots, C_t\} \subset \mathcal{C}(S_i)$ . If  $t = 0$ ,  $y$  and  $u$  are adjacent in the original graph  $\mathcal{G}$ . If  $t > 0$ , by definition of connected components,  $t$  cannot be greater than one; that is, the path must be

$$(y, C, u),$$

5.3. COMPUTER REPRESENTATION OF ELIMINATION GRAPHS<sup>123</sup>

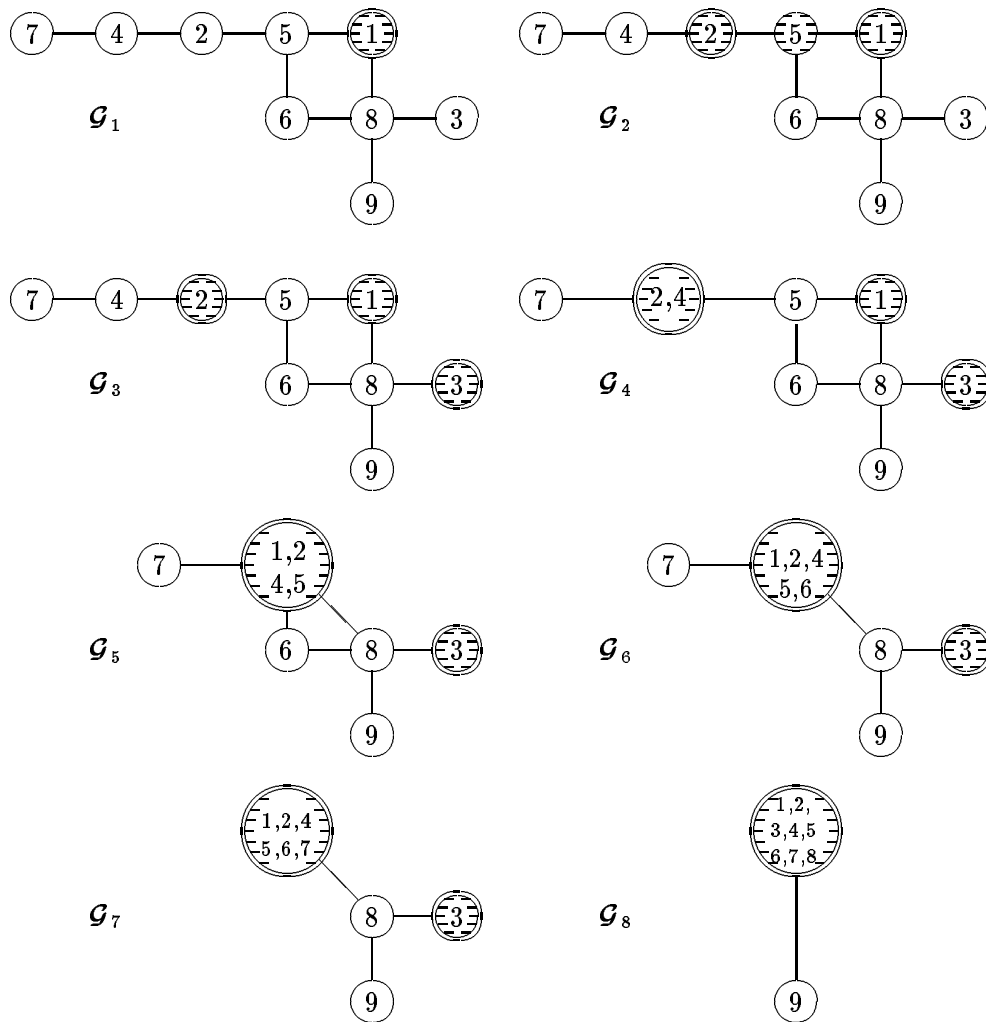


Figure 5.3.3: A sequence of quotient graphs.

so that we can obtain a path from  $y$  to  $u$  through  $C$  in the graph  $\mathcal{G}$ . Hence

$$u \in \text{Reach}_{\mathcal{G}}(y, S_i).$$

□

The determination of reachable sets in the quotient graph  $\mathcal{G}_i$  is straightforward. For a given node  $y \notin \mathcal{C}(S_i)$ , the following algorithm returns the set  $\text{Reach}_{\mathcal{G}_i}(y, \mathcal{C}(S_i))$ .

**Step 1** (*Initialization*)  $R \leftarrow \phi$ .

**Step 2** (*Find reachable nodes*)

```

for  $x \in \text{Adj}_{\mathcal{G}_i}(y)$  do
  if  $x \in \mathcal{C}(S_i)$ 
    then  $R \leftarrow R \cup \text{Adj}_{\mathcal{G}_i}(x)$ 
  else  $R \leftarrow R \cup \{x\}$ .

```

**Step 3** (*Exit*) The reachable set is given in  $R$ .

The connection between elimination graphs and quotient graphs (5.3.3) is quite obvious. Indeed, we can obtain the structure of the elimination graph  $\mathcal{G}_i$  from that of  $\mathcal{G}_i$  by the simple algorithm below.

**Step 1** Remove supernodes in  $\mathcal{C}(S_i)$  and their incident edges from the quotient graph.

**Step 2** For each  $C \in \mathcal{C}(S_i)$ , add edges to the quotient graph so that all adjacent nodes of  $C$  are pairwise adjacent in the elimination graph.

To illustrate the idea, consider the transformation of  $\mathcal{G}_4$  to  $\mathcal{G}_4$  for the example in Figure 5.3.3. The elimination graph  $\mathcal{G}_4$  is given in Figure 5.3.4.

In terms of implicitness, the quotient graph model lies in between the reachable set approach and the elimination graph model as a vehicle for representing the elimination process.

Reachable set					
on original	→	Quotient	→	Elimination	
graph		graph		graph	

The correspondence between the three models is summarized in Table 5.3.1.

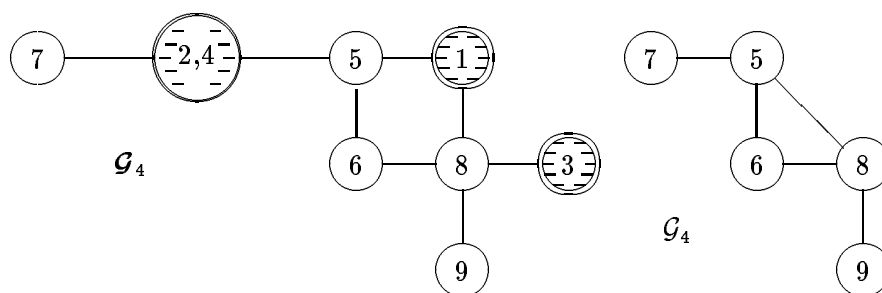


Figure 5.3.4: From quotient graph to elimination graph.

	Implicit Model	Quotient Model	Explicit Model
Representation	$S_1$	$\mathcal{G}_1$	$\mathcal{G}_1$
	$S_2$	$\mathcal{G}_2$	$\mathcal{G}_2$
	$\vdots$	$\vdots$	$\vdots$
	$S_{n-1}$	$\mathcal{G}_{n-1}$	$\mathcal{G}_{n-1}$
Adjacency	$Reach(y, S_i)$	$Reach_{\mathcal{G}_i}(y, \mathcal{C}(S_i))$	$Adj_{\mathcal{G}_i}(y)$

Table 5.3.1: Correspondence among the elimination models.

### 5.3.3 Implementation of the Quotient Graph Model

Consider the quotient graph  $\mathcal{G} = \mathcal{G}/\bar{\mathcal{C}}(S)$  induced by the eliminated set  $S$ . For notational convenience, if  $s \in S$ , we use the notation  $\bar{s}$  to denote the connected component in the subgraph  $\mathcal{G}(S)$ , containing the node  $s$ . For example, in the quotient graph of Figure 5.3.2,

$$\bar{x}_5 = \bar{x}_1 = \bar{x}_2 = \bar{x}_4 = \{x_1, x_2, x_4, x_5\}.$$

On the other hand, for a given  $C \in \mathcal{C}(S)$ , we can select any node  $x$  from  $C$  and use  $x$  as a *representative* for  $C$ , that is,  $\bar{x} = C$ . Before we discuss the choice of representative in the implementation, we establish some results that can be used to show that the model can be implemented *in-place*; that is, in the space provided by the adjacency structure of the original graph.

**Lemma 5.3.2** *Let  $\mathcal{G} = (X, E)$  and  $C \subset X$  where  $\mathcal{G}(C)$  is a connected subgraph. Then*

$$\sum_{x \in C} |Adj(x)| \geq |Adj(C)| + 2(|C| - 1).$$

**Proof:** Since  $\mathcal{G}(C)$  is connected, there are at least  $|C| - 1$  edges in the subgraph. These edges are counted twice in  $\sum_{x \in C} |Adj(x)|$  and hence the result.  $\square$

Let  $x_1, x_2, \dots, x_n$  be the node sequence and  $S_i = \{x_1, \dots, x_i\}$ ,  $1 \leq i \leq n$ . For  $1 \leq i \leq n$ , let

$$\mathcal{G}_i = \mathcal{G}/\bar{\mathcal{C}}(S_i) = (\bar{\mathcal{C}}(S_i), \mathcal{E}_i).$$

**Lemma 5.3.3** *Let  $y \in X - S_i$ . Then*

$$|Adj_{\mathcal{G}}(y)| \geq |Adj_{\mathcal{G}_i}(y)|.$$

**Proof:** This follows from the inequality

$$|Adj_{\mathcal{G}_i}(y)| \geq |Adj_{\mathcal{G}_{i+1}}(y)|$$

for  $y \in X - S_{i+1}$ . The problem of verifying this inequality is left as an exercise.  $\square$

**Theorem 5.3.4**

$$\max_{1 \leq i \leq n} |\mathcal{E}_i| \leq |E|.$$

**Proof:** Consider the quotient graphs  $\mathcal{G}_i$  and  $\mathcal{G}_{i+1}$ . If  $x_{i+1}$  is isolated in the subgraph  $\mathcal{G}(S_{i+1})$ , clearly  $|\mathcal{E}_{i+1}| = |\mathcal{E}_i|$ . Otherwise the node  $x_{i+1}$  is merged with some components in  $S_i$  to form a new component in  $S_{i+1}$ . The results of Lemmas 5.3.2 and 5.3.3 apply, so that

$$|\mathcal{E}_{i+1}| < |\mathcal{E}_i|.$$

Hence, in all cases,

$$|\mathcal{E}_{i+1}| \leq |\mathcal{E}_i|$$

and the result follows.  $\square$

Theorem 5.3.4 shows that the sequence of quotient graphs produced by elimination requires no more space than the original graph structure. On coalescing a connected set  $C$  into a supernode, we know from Lemma 5.3.2 that there are enough storage locations for  $Adj(C)$  from those of  $Adj(x)$ ,  $x \in C$ . Moreover, for  $|C| > 1$ , there is a surplus of  $2(|C| - 1)$  locations, which can be used for links or pointers.

Figure 5.3.5 is an illustration of the data structure used to represent  $Adj_{\mathcal{G}}(C)$ , in the quotient graph  $\mathcal{G}$ , where  $C = \{a, b, c\}$ . Here, zero signifies the end of the neighbor list in  $\mathcal{G}$ .

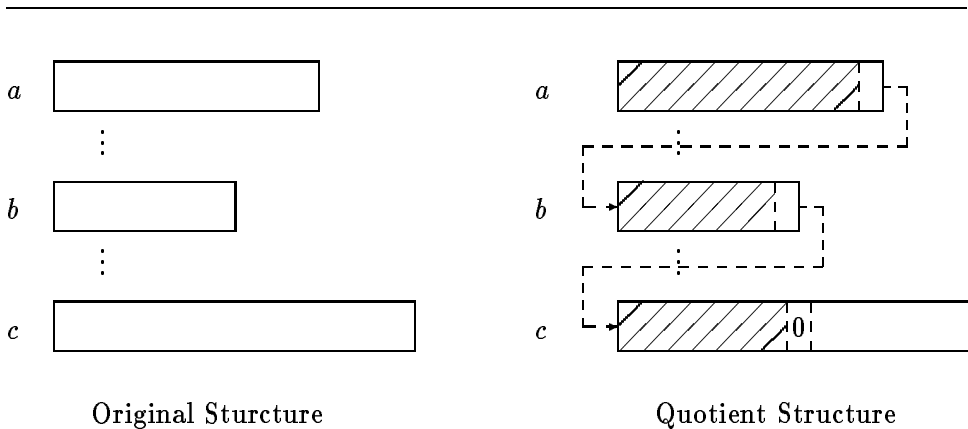


Figure 5.3.5: Data structure for quotient graphs.

Note that in the example, the node “a” is chosen to be the representative for  $C = \{a, b, c\}$ . In the computer implementation, it is important to choose a *unique* representative for each  $C \in \mathcal{C}(S)$ , so that any reference to  $C$  can be made through its representative.



Let  $x_1, x_2, \dots, x_n$  be the node sequence, and  $C \in \mathcal{C}(S)$ . We choose the node  $x_r \in C$  to be the representative of  $C$ , where

$$r = \max\{j \mid x_j \in C\}. \quad (5.3.4)$$

That is,  $x_r$  is the node in  $C$  last eliminated.

So far, we have described the data structure of the quotient graphs and how to represent supernodes. Another important aspect in the implementation of the quotient graph model for elimination is the transformation of quotient graphs due to node elimination. Let us now consider how the adjacency structure of  $\mathcal{G}_i$  can be obtained from that of  $\mathcal{G}_{i-1}$  when the node  $x_i$  is eliminated. The following algorithm performs the transformation.

**Step 1** (*Preparation*) Determine the sets

$$\begin{aligned} T &= \text{Adj}_{\mathcal{G}_{i-1}}(x_i) \cap \mathcal{C}(S_{i-1}) \\ R &= \text{Reach}_{\mathcal{G}_{i-1}}(x_i, \mathcal{C}(S_{i-1})). \end{aligned}$$

**Step 2** (*Form new supernode and partitioning*) Form

$$\begin{aligned} \bar{x}_i &= \{x_i\} \cup T \\ \mathcal{C}(S_i) &= (\mathcal{C}(S_{i-1}) - T) \cup \{\bar{x}_i\}. \end{aligned}$$

**Step 3** (*Update adjacency*)

$$\begin{aligned} \text{Adj}_{\mathcal{G}_i}(\bar{x}_i) &= R \\ \text{For } y \in R, \quad \text{Adj}_{\mathcal{G}_i}(y) &= \{\bar{x}_i\} \cup \text{Adj}_{\mathcal{G}_{i-1}}(y) - (T \cup \{x_i\}). \end{aligned}$$

Let us apply this algorithm to transform  $\mathcal{G}_4$  to  $\mathcal{G}_5$  in the example of Figure 5.3.3. In  $\mathcal{G}_4$

$$\mathcal{C}(S_4) = \{\bar{x}_1, \bar{x}_3, \bar{x}_4\}.$$

On applying Step 1 to the node  $x_5$ , we obtain

$$T = \{\bar{x}_1, \bar{x}_4\}$$

and

$$R = \{x_6, x_7, x_8\}.$$

Therefore, the new ‘‘supernode’’ is given by

$$\bar{x}_5 = \{x_5\} \cup \bar{x}_1 \cup \bar{x}_4 = \{x_1, x_2, x_4, x_5\}.$$

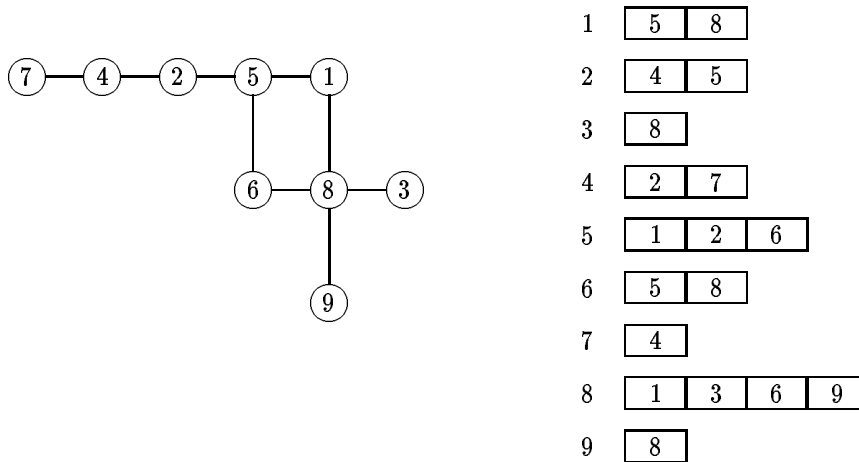


Figure 5.3.6: Adjacency representation.

and the new partitioning is

$$\mathcal{C}(S_5) = \{\bar{x}_3, \bar{x}_5\}.$$

Finally, in Step 3 the adjacency sets are updated and we get

$$Adj_{\mathcal{G}_5}(x_6) = \{\bar{x}_5, x_8\}$$

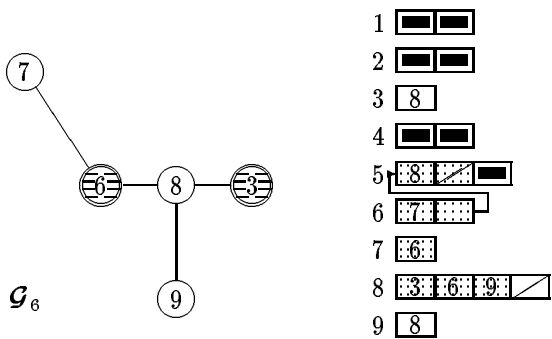
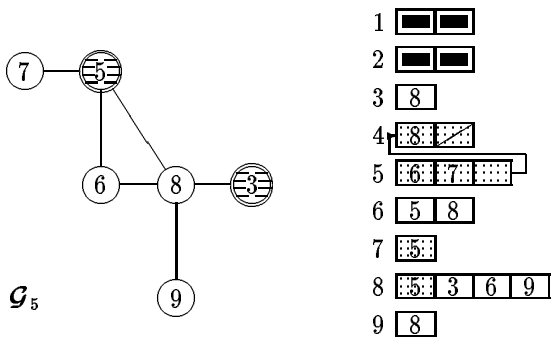
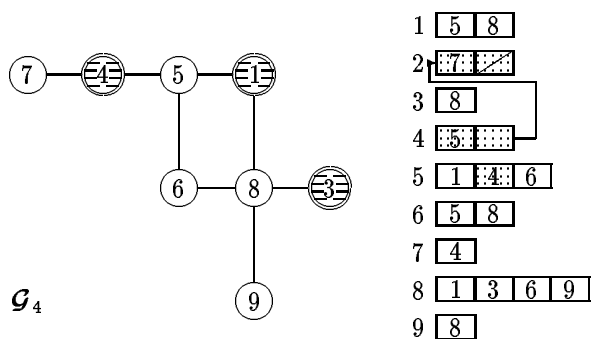
$$Adj_{\mathcal{G}_5}(x_7) = \{\bar{x}_5\}$$

$$Adj_{\mathcal{G}_5}(x_8) = \{\bar{x}_3, \bar{x}_5, x_6, x_9\},$$

and

$$Adj_{\mathcal{G}_5}(\bar{x}_5) = R = \{x_6, x_7, x_8\}.$$

The effect of the quotient graph transformation on the data structure can be illustrated by an example. Consider the example of Figure 5.3.3, where we assume that the adjacency structure is represented as shown in Figure 5.3.6. Figure 5.3.7 shows some important steps in producing quotient graphs for this example. The adjacency structure remains unchanged when the quotient graphs  $\mathcal{G}_1$ ,  $\mathcal{G}_2$  and  $\mathcal{G}_3$  are formed. To transform  $\mathcal{G}_3$  to  $\mathcal{G}_4$ , the nodes  $x_2$  and  $x_4$  are to be coalesced, so that in  $\mathcal{G}_4$ , the new adjacent set of node  $x_4$  contains



ignored   
  end of list   
  modified

Figure 5.3.7: An in-place quotient graph transformation.

that of the subset  $\{x_2, x_4\}$  in the original graph, namely  $\{x_5, x_7\}$ . Here, the last location for the adjacent set of  $x_4$  is used as a link. Note also that in the adjacent list of node  $x_5$ , the neighbor  $x_2$  has been changed to  $x_4$  in  $\mathcal{G}_4$  since node  $x_4$  becomes the representative of the component subset  $\{x_2, x_4\}$ . The representations for  $\mathcal{G}_5$  and  $\mathcal{G}_6$  in this storage mode are also included in Figure 5.3.7.

This way of representing quotient graphs for elimination will be used in the implementation of the minimum degree ordering algorithm, to be discussed in the Section 5.5

### Exercises

- 5.3.1) a) Design and implement a subroutine called **REACH** which can be used to determine the reachable set of a given node **ROOT** through a subset  $S$ . The subset is given by an array **SFLAG**, where a node  $i$  belongs to  $S$  if **SFLAG**( $i$ ) is nonzero. Describe the parameters of the subroutine and any auxiliary storage you require.
- b) Suppose a graph is stored in the array pair (**XADJ**, **ADJNCY**). For any given elimination sequence, use the subroutine **REACH** to print out the adjacency structures of the sequence of elimination graphs.
- 5.3.2) Let  $\bar{\mathcal{C}}(S_i)$  be as defined in (5.3.2) and show that  $|\bar{\mathcal{C}}(S_{i+1})| \leq |\bar{\mathcal{C}}(S_i)|$ .
- 5.3.3) Prove the inequality that appears in the proof of Lemma 5.3.3.
- 5.3.4) Let  $\mathcal{X} = \{C \mid C \in \mathcal{C}(S_i) \text{ for some } i\}$ . Show that  $|\mathcal{X}| = n$ .
- 5.3.5) Let  $C \in \mathcal{C}(S_i)$ , and  $\bar{x}_r = C$  where

$$r = \max\{j \mid x_j \in C\}.$$

Show that

- a)  $Adj_{\mathcal{G}}(C) = Reach_{\mathcal{G}}(x_r, S_i)$ .
- b)  $Reach_{\mathcal{G}}(x_r, S_i) = Reach_{\mathcal{G}}(x_r, S_{r-1})$ .
- 5.3.6) Display the sequence  $\{\mathcal{G}_i\}$  of quotient graphs for the star graph of 7 nodes, where the centre node is numbered first.

## 5.4 The Minimum Degree Ordering Algorithm

Let  $\mathbf{A}$  be a given symmetric matrix and let  $\mathbf{P}$  be a permutation matrix. Although the nonzero structures of  $\mathbf{A}$  and  $\mathbf{PAP}^T$  are different, their sizes are the same:  $|\text{Nonz}(\mathbf{A})| = |\text{Nonz}(\mathbf{PAP}^T)|$ . However, the crucial point is that there may be a dramatic difference between  $|\text{Nonz}(\mathbf{F}(\mathbf{A}))|$  and  $|\text{Nonz}(\mathbf{F}(\mathbf{PAP}^T))|$  for some permutation  $\mathbf{P}$ . The example in Figure 4.3.3 illustrates this fact.

Ideally, we want to find a permutation  $\mathbf{P}^*$  that minimizes the size of the nonzero structure of the filled matrix:

$$|\text{Nonz}(\mathbf{F}(\mathbf{P}^* \mathbf{A} \mathbf{P}^{*T}))| = \min_{\mathbf{P}} |\text{Nonz}(\mathbf{F}(\mathbf{PAP}^T))|.$$

So far, there is no efficient algorithm for getting such an optimal  $\mathbf{P}^*$  for a general symmetric matrix. Indeed, the problem has been shown to be very difficult – a so-called NP-complete problem (Yannakakis [56]). Thus, we have to rely on heuristics which will produce an ordering  $\mathbf{P}$  with an acceptably small but not necessarily minimum  $|\text{Nonz}(\mathbf{F}(\mathbf{PAP}^T))|$ .

By far the most popular fill-reducing scheme used is the *minimum degree* algorithm (Tinney [53]), which corresponds to the Markowitz scheme (Markowitz [39]) for unsymmetric matrices. The scheme is based on the following observation, which is depicted in Figure 5.4.1.

Suppose  $\{x_1, \dots, x_{i-1}\}$  have been labelled. The number of nonzeros in the filled graph for these columns is fixed. In order to reduce the number of nonzeros in the  $i$ -th column, it is apparent that in the submatrix remaining to be factored, the column with the fewest nonzeros should be moved to become column  $i$ . In other words, the scheme may be regarded as a method that reduces the fill of a matrix by a local minimization of  $\eta(\mathbf{L}_{*i})$  in the factored matrix.

### 5.4.1 The Basic Algorithm

The minimum degree algorithm can be most easily described in terms of ordering a symmetric graph. Let  $\mathcal{G}_0 = (X, E)$  be an unlabelled graph. Using the elimination graph model, the basic algorithm is as follows.

**Step 1** (*Initialization*)  $i \leftarrow 1$ .

**Step 2** (*Minimum degree selection*) In the graph  $\mathcal{G}_{i-1} = (X_{i-1}, E_{i-1})$ , choose a node  $x_i$  of minimum degree.

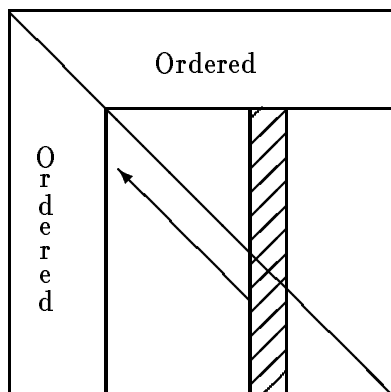


Figure 5.4.1: Motivation of the minimum degree algorithm.

**Step 3** (*Graph transformation*) Form the new elimination graph  $\mathcal{G}_i = (X_i, E_i)$  by eliminating the node  $x_i$  from  $\mathcal{G}_{i-1}$ .

**Step 4** (*Loop or stop*)  $i \leftarrow i + 1$ . If  $i > |X|$ , stop. Otherwise, go to Step 2.

As an illustration of the algorithm, we consider the graph in Figure 5.4.2. The way the minimum degree algorithm is carried out for this example is shown step by step in Figure 5.4.3. Notice that there can be more than one node with the minimum degree at a particular step. Here we break the ties arbitrarily. However, different tie-breaking strategies give different versions of the minimum degree algorithm.

#### 5.4.2 Description of the Minimum Degree Algorithm Using Reachable Sets

The use of elimination graphs in the minimum degree algorithm provides the mechanism by which we select the next node to be numbered. Each step of the algorithm involves a graph transformation, which is the most expensive part of the algorithm in terms of implementation. These transformations can be eliminated if we can provide an alternative way to compute the degrees of the nodes in the elimination graph.

Theorem 5.2.3 provides a mechanism for achieving this through the use of reachable sets. With this connection, we can restate the minimum degree algorithm as follows.

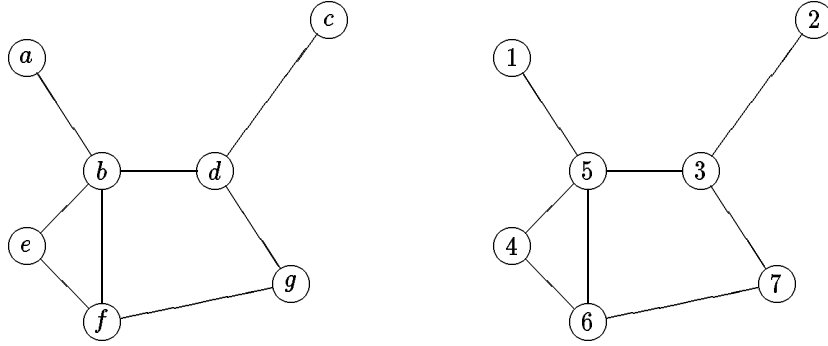


Figure 5.4.2: A minimum degree ordering for a graph.

**Step 1 (Initialization)**  $S \leftarrow \phi$ .  $Deg(x) \leftarrow |Adj(x)|$ , for  $x \in X$ .

**Step 2 (Minimum degree selection)** Pick a node  $y \in X - S$  where  $Deg(y) = \min_{x \in X - S} Deg(x)$ . Number the node  $y$  next and set  $T \leftarrow S \cup \{y\}$ .

**Step 3 (Degree update)**  $Deg(u) \leftarrow |Reach(u, T)|$  for  $u \in X - T$ .

**Step 4 (Loop or stop)** If  $T = X$ , stop. Otherwise, set  $S \leftarrow T$  and go to Step 2.

This approach uses the original graph structure throughout the entire process. Indeed, the algorithm can be carried out with only the adjacency structure

$$\mathcal{G}_0 = (X, E).$$

It is appropriate here to point out that in the degree update step of the algorithm, it is not necessary to recompute the sizes of the reachable sets for every node in  $X - T$ , since most of them remain unchanged. This observation is formalized in the following lemma. Its proof follows from the definition of reachable sets and is left as an exercise.

**Lemma 5.4.1** *Let  $y \notin S$  and  $T = S \cup \{y\}$ . Then*

$$Reach(x, T) = \begin{cases} Reach(x, S) & \text{for } x \notin Reach(y, S) \\ Reach(x, S) \cup Reach(y, S) - \{x, y\} & \text{otherwise} \end{cases}$$

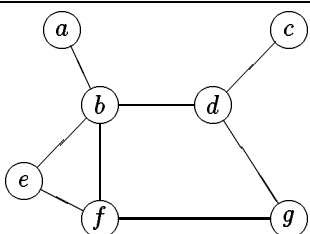
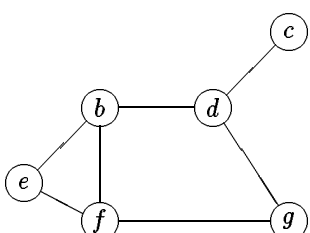
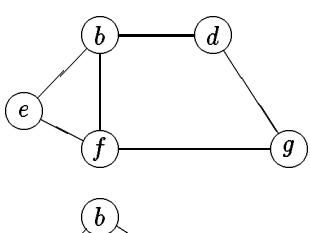
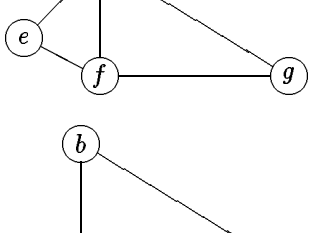
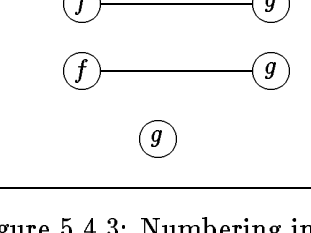


$i$	Elimination Graph $\mathcal{G}_{i-1}$	Node Selected	Minimum Degree
1		$a$	1
2		$c$	1
3		$d$	2
4		$e$	2
5		$b$	2
6		$f$	1
7		$g$	0

Figure 5.4.3: Numbering in the minimum degree algorithm.



In the example of Figure 5.4.3, consider the stage when node  $d$  is being eliminated.

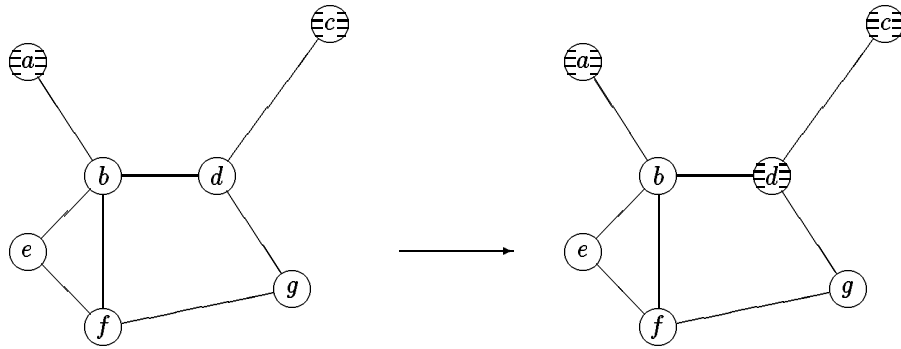


Figure 5.4.4: Elimination of node  $d$  from  $\mathcal{G}_2$  at stage 3.

We have  $S = \{a, c\}$ , so that  $Reach(d, S) = \{b, g\}$ . Therefore, the elimination of  $d$  only affects the degrees of the nodes  $b$  and  $g$ . By this observation, Step 3 in the algorithm can be restated as

**Step 3** (*Degree update*)

$$Deg(u) \leftarrow |Reach(u, T)|, \text{ for } u \in Reach(y, S).$$

**Corollary 5.4.2** *Let  $y, S, T$  be as in Lemma 5.4.1. For  $x \in X - T$ ,*

$$|Reach(x, T)| \geq |Reach(x, S)| - 1.$$

**Proof:** The result follows directly from Lemma 5.4.1. □

### 5.4.3 An Enhancement

As the algorithm stands, one node is numbered each time the loop is executed. However, when a node  $y$  of minimum degree is found at Step 2, it is often possible to detect that a subset of nodes may automatically be numbered next, without carrying out any minimum degree search.

Let us begin the study by introducing an equivalence relation. Consider a stage in the elimination process, where  $S$  is the set of eliminated nodes. Two

nodes  $x, y \in X - S$  are said to be *indistinguishable with respect to elimination* if

$$\text{Reach}(x, S) \cup \{x\} = \text{Reach}(y, S) \cup \{y\}. \quad (5.4.1)$$

(Henceforth, it should be understood that nodes referred to as “indistinguishable” are *indistinguishable with respect to elimination*.)

Consider the graph example in Figure 5.4.5. The subset  $S$  contains 36 shaded nodes. (This is an actual stage that occurs when the minimum degree algorithm is applied to this graph.) We note that the nodes  $a, b$  and  $c$  are indistinguishable with respect to elimination, since  $\text{Reach}(a, S) \cup \{a\}$ ,  $\text{Reach}(b, S) \cup \{b\}$  and  $\text{Reach}(c, S) \cup \{c\}$  are all equal to

$$\{a, b, c, d, e, f, g, h, j, k\}.$$

There are two more groups that can be identified as indistinguishable. They are

$$\{j, k\},$$

and

$$\{f, g\}.$$

We now study the implication of this equivalence relation and its role in the minimum degree algorithm. As we shall see later, this notion can be used to speed up the execution of the minimum degree algorithm.

**Theorem 5.4.3** *Let  $x, y \in X - S$ . If*

$$\text{Reach}(x, S) \cup \{x\} = \text{Reach}(y, S) \cup \{y\},$$

*then for all  $X - \{x, y\} \supset T \supset S$ ,*

$$\text{Reach}(x, T) \cup \{x\} = \text{Reach}(y, T) \cup \{y\}.$$

**Proof:** Obviously,  $x \in \text{Reach}(y, S) \subset \text{Reach}(y, T) \cup T$ , (see Exercise 5.2.7 on page 117) so that  $x \in \text{Reach}(y, T)$ . We now want to show that  $\text{Reach}(x, T) \subset \text{Reach}(y, T) \cup \{y\}$ . Consider  $z \in \text{Reach}(x, T)$ . There exists a path

$$(x, s_1, \dots, s_t, z)$$

where  $\{s_1, \dots, s_t\} \subset T$ . If all  $s_i \in S$ , there is nothing to prove. Otherwise, let  $s_i$  be the first node in  $\{s_1, \dots, s_t\}$  not in  $S$ , that is

$$s_i \in \text{Reach}(x, S) \cap T.$$

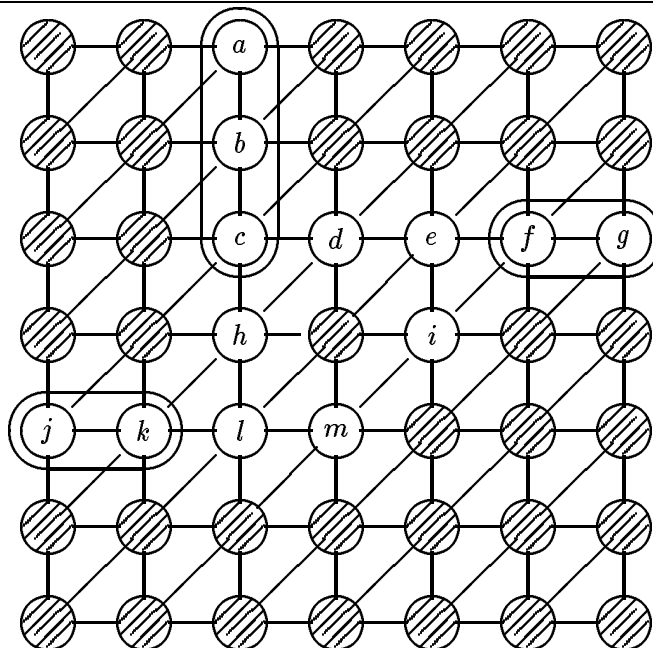


Figure 5.4.5: An example displaying indistinguishable nodes.

---

This implies  $s_i \in \text{Reach}(y, S)$  and hence  $z \in \text{Reach}(y, T)$ . Together, we have

$$\text{Reach}(x, T) \cup \{x\} \subset \text{Reach}(y, T) \cup \{y\}.$$

The inclusion in the other direction follows from symmetry, yielding the result.  $\square$

**Corollary 5.4.4** *Let  $x, y$  be indistinguishable with respect to the subset  $S$ . Then for  $T \supset S$ ,*

$$|\text{Reach}(x, T)| = |\text{Reach}(y, T)|.$$

In other words, if two nodes become indistinguishable at some stage of the elimination, they remain indistinguishable until one of them is eliminated. Moreover, the following theorem shows that they can be eliminated *together* in the minimum degree algorithm.

**Theorem 5.4.5** *If two nodes become indistinguishable at some stage in the minimum degree algorithm. then they can be eliminated together in the algorithm.*

**Proof:** Let  $x, y$  be indistinguishable after the elimination of the subset  $S$ . Assume that  $x$  becomes a node of minimum degree after the set  $T \supset S$  has been eliminated, that is,

$$|\text{Reach}(x, T)| \leq |\text{Reach}(z, T)| \text{ for all } z \in X - T.$$

Then, by Corollary 5.4.4,

$$\begin{aligned} |\text{Reach}(y, T \cup \{x\})| &= |\text{Reach}(y, T) - \{x\}| \\ &= |\text{Reach}(y, T)| - 1 \\ &= |\text{Reach}(x, T)| - 1. \end{aligned}$$

Therefore, for all  $z \in X - T \cup \{x\}$ , by Corollary 5.4.2,

$$\begin{aligned} |\text{Reach}(y, T \cup \{x\})| &\leq |\text{Reach}(z, T)| - 1 \\ &\leq |\text{Reach}(z, T \cup \{x\})|. \end{aligned}$$

In other words, after the elimination of the node  $x$ , the node  $y$  becomes a node of minimum degree.  $\square$

These observations can be exploited in the implementation of the minimum degree algorithm. After carrying out a minimum degree search to determine

the next node  $y \in X - S$  to eliminate, we can number immediately after  $y$  the set of nodes indistinguishable from  $y$ .

In addition, in the degree update step, by virtue of Corollary 5.4.4, work can be reduced since indistinguishable nodes have the same degree in the elimination graphs. Once nodes are identified as being indistinguishable, they can be “glued” together and treated as a single supernode thereafter. For example, Figure 5.4.6 shows two stages in the eliminations where supernodes are formed from indistinguishable nodes. For simplicity, the eliminated nodes are not shown. After the elimination of the indistinguishable set  $\{a, b, c\}$ , all the nodes have identical reachable sets so that they can be merged into one.

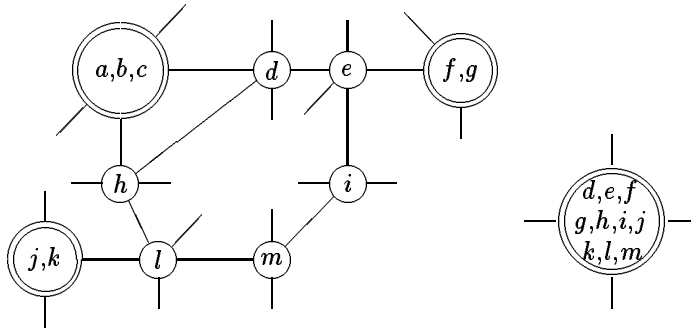


Figure 5.4.6: Indistinguishable nodes in two stages of elimination for the example in Figure 5.4.5.

In general, to identify indistinguishable nodes via the definition (5.4.1) is time consuming. Since the enhancement does not require the merging of *all possible* indistinguishable nodes, we look for some simple, easily-implemented condition. In what follows, a condition is presented which experience has shown to be very effective. In most cases, it identifies all indistinguishable nodes.

Let  $\mathcal{G} = (X, E)$  and  $S$  be the set of eliminated nodes. Let  $\mathcal{G}(C_1)$  and  $\mathcal{G}(C_2)$  be two connected components in the subgraph  $\mathcal{G}(S)$ ; that is,

$$C_1, C_2 \in \mathcal{C}(S).$$

**Lemma 5.4.6** *Let  $R_1 = Adj(C_1)$ , and  $R_2 = Adj(C_2)$ . If  $y \in R_1 \cap R_2$ , and*

$$Adj(y) \subset R_1 \cup R_2 \cup C_1 \cup C_2$$

*then  $Reach(y, S) \cup \{y\} = R_1 \cup R_2$ .*

**Proof:** Let  $x \in R_1 \cup R_2$ . Assume  $x \in R_1 = Adj(C_1)$ . Since  $\mathcal{G}(C_1)$  is a connected component in  $\mathcal{G}(S)$ , we can find a path from  $y$  to  $x$  through  $C_1 \subset S$ . Therefore,  $x \in Reach(y, S) \cup \{y\}$ .

On the other hand,  $y \in R_1 \cup R_2$  by definition. Moreover, if  $x \in Reach(y, S)$ , there exists a path from  $y$  to  $x$  through  $S$ :

$$(y, s_1, s_2, \dots, s_t, x).$$

If  $t = 0$ , then  $x \in Adj(y) - S \subset R_1 \cup R_2$ . Otherwise, if  $t > 0$ ,  $s_1 \in Adj(y) \cap S \subset C_1 \cup C_2$ . This means  $\{s_1, \dots, s_t\}$  is a subset of either  $C_1$  or  $C_2$  so that  $x \in R_1 \cup R_2$ . Hence  $Reach(y, S) \cup \{y\} \subset R_1 \cup R_2$ .  $\square$

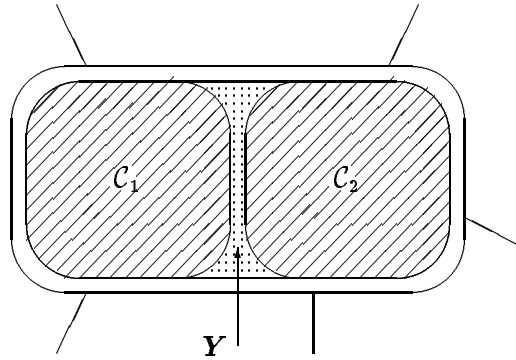


Figure 5.4.7: Finding indistinguishable nodes.

**Theorem 5.4.7** *Let  $C_1, C_2$  and  $R_1, R_2$  be as in Lemma 5.4.6. Then the nodes in*

$$Y = \{y \in R_1 \cap R_2 \mid Adj(y) \subset R_1 \cup R_2 \cup C_1 \cup C_2\} \tag{5.4.2}$$

*are indistinguishable with respect to the eliminated subset  $S$ .*

**Proof:** It follows from Lemma 5.4.6.  $\square$

**Corollary 5.4.8** For  $y \in Y$ ,

$$|\text{Reach}(y, S)| = |R_1 \cup R_2| - 1.$$

Theorem 5.4.7 can be used to merge indistinguishable nodes in the intersection of the two reachable sets  $R_1$  and  $R_2$ . The test can be simply done by inspecting the adjacent set of nodes in the intersection  $R_1 \cap R_2$ .

This notion of indistinguishable nodes can be applied to the minimum degree algorithm. The new enhanced algorithm can be stated as follows.

**Step 1** (*Initialization*)  $S \leftarrow \phi$ ,

$$\text{Deg}(x) = |\text{Adj}(x)|, \text{ for } x \in X.$$

**Step 2** (*Selection*) Pick a node  $y \in X - S$  such that

$$\text{Deg}(y) = \min_{x \in X - S} \text{Deg}(x).$$

**Step 3** (*Elimination*) Number the nodes in

$$Y = \{x \in X - S \mid x \text{ is indistinguishable from } y\}$$

next in the ordering.

**Step 4** (*Degree update*) For  $u \in \text{Reach}(y, S) - Y$

$$\text{Deg}(u) = |\text{Reach}(u, S \cup Y)|$$

and identify indistinguishable nodes in the set  $\text{Reach}(y, S) - Y$ .

**Step 5** (*Loop or stop*) Set  $S \leftarrow S \cup Y$ . If  $S = X$ , stop. Otherwise, go to Step 2.

#### 5.4.4 Implementation of the Minimum Degree Algorithm

The implementation of the minimum degree algorithm presented here incorporates the notion of indistinguishable nodes as described in the previous sections. Nodes identified as indistinguishable are merged together to form a supernode. They will be treated essentially *as one node* in the remainder of the algorithm. They share the same adjacent set, have the same degree, and can be eliminated together in the algorithm. In the implementation, this supernode will be referenced by a representative of the set.

The algorithm requires the determination of reachable sets for degree update. The quotient graph model (Section 5.3.2) is used for this purpose to improve the overall efficiency of the algorithm. In effect, eliminated connected nodes are merged together and the computer representation of the sequence of quotient graphs (Section 5.3.3) is utilized.

It should be emphasized that the idea of *quotient* (or merging nodes into supernodes) is applied here in two different contexts.

- a) *eliminated* connected nodes to facilitate the determination of reachable sets.
- b) *uneliminated indistinguishable* nodes to speed up elimination.

This is illustrated in Figure 5.4.8. It shows how the graph of Figure 5.4.5 is stored conceptually in this implementation by the two forms of quotient. The shaded double-circled nodes denote supernodes that have been eliminated, while blank double-circled supernodes represent those formed from indistinguishable nodes.

In this subsection, we describe a set of subroutines, which implement the minimum degree algorithm as presented earlier. Some of the parameters used are the same as those discussed in Chapter 3. We shall briefly review them here and readers are referred to Section 3.4 for details.

The graph  $\mathcal{G} = (X, E)$  is stored using the integer array pair (XADJ, ADJNCY), and the number of variables in  $X$  is given by NEQNS. The resulting minimum degree ordering is stored in the vector PERM, while INVP returns the inverse of this ordering.

This collection of subroutines requires some working vectors to implement the quotient graph model and the notion of indistinguishable nodes. The current degrees of the nodes in the (implicit) elimination graph are kept in the array DEG. The DEG value for nodes that have been eliminated is set to  $-1$ .

In the representation of the sequence of quotient graphs, connected eliminated nodes are merged to form a supernode. As mentioned in Section 5.4.2, for the purpose of reference, it is sufficient to pick a *representative* from the supernode. If  $\mathcal{G}(C)$  is such a connected component, we always choose the node  $x \in C$  last eliminated to represent  $C$ . This implies that the remaining nodes in  $C$  can be ignored in subsequent quotient graphs.

The same remark applies to indistinguishable groups of uneliminated nodes. For each group, only the representative will be considered in the present quotient structure.



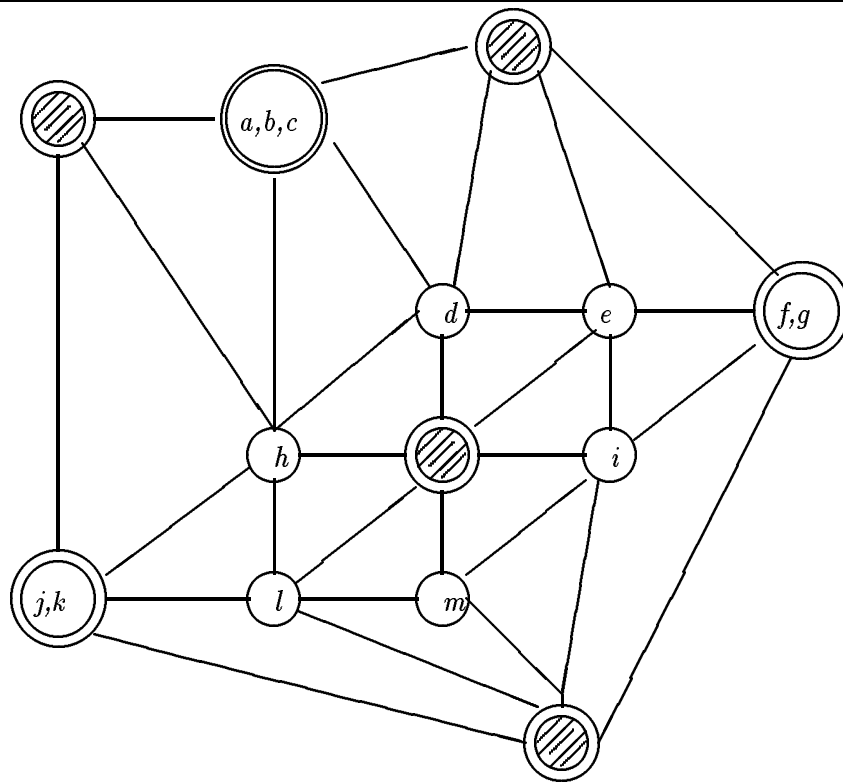


Figure 5.4.8: A quotient graph formed from two types of supernodes.

---

The working vector **MARKER** is used to mark those nodes that can be ignored in the adjacency structure. The **MARKER** values for such nodes are set to  $-1$ . This vector is also used temporarily to facilitate the generation of reachable sets.

Two more arrays **QSIZE** and **QLINK** are used to completely specify indistinguishable supernodes. If node  $i$  is the representative, the number of nodes in this supernode is given by  $\text{QSIZE}(i)$  and the nodes are given by

$$i, \text{QLINK}(i), \text{QLINK}(\text{QLINK}(i)), \dots$$

Figure 5.4.9 illustrates the use of the vectors **QSIZE**, **QLINK** and **MARKER**. The nodes  $\{2, 5, 8\}$  form an indistinguishable supernode represented by node 2. Thus, the **MARKER** values of 5 and 8 are  $-1$ . On the other hand,  $\{3, 6, 9\}$  forms an eliminated supernode. Its representative is node 9 so that **MARKER**(3) and **MARKER**(6) are  $-1$ .

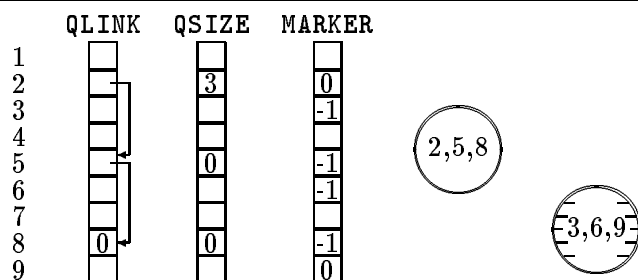


Figure 5.4.9: Illustration of the role of **QLINK**, **QSIZE** and **MARKER** working vectors.

---

There are five subroutines in this set, namely **GENQMD**, **QMDRCH**, **QMDQT**, **QMDUPD**, and **QMDMRG**. Their control relationship is as shown in Figure 5.4.10. They are described in detail in this Figure.

#### **GENQMD (GENeral Quotient Minimum Degree algorithm)**

The purpose of this subroutine is to find the minimum degree ordering for a general disconnected graph. It operates on the input graph as given by **NEQNS** and (**XADJ**, **ADJNCY**), and returns the ordering in the vectors **PERM** and **INVP**. On return, the adjacency structure will be destroyed because it is used by the subroutine to store the sequence of quotient graph structures.

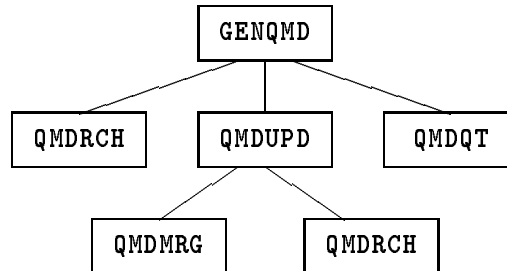


Figure 5.4.10: Control relation of subroutines for the minimum degree algorithm.

---

The subroutine begins by initializing the working arrays `QSIZE`, `QLINK`, `MARKER` and the `DEG` vector. It then prepares itself for the main loop of the algorithm. In the main loop the subroutine first determines a node of minimum degree by the technique of threshold searching. It keeps two variables `THRESH` and `MINDEG`. Any node with its current degree equal to the value of `THRESH` is one with minimum degree in the elimination graph. The variable `MINDEG` keeps the lowest degree greater than the threshold value `THRESH`, and it is used to update the value of `THRESH`.

Having found a node `NODE` of minimum degree, `GENQMD` then determines the reachable set of `NODE` through eliminated supernodes by calling the subroutine `QMDRCH`. The set is contained in the vector `RCHSET` and its size in `RCHSZE`. The nodes indistinguishable from `NODE` are then retrieved via the vector `QLINK`, and numbered (eliminated).

Next, the nodes in the reachable set have their degree updated and at the same time more indistinguishable nodes are identified. In the program, this is done by calling the subroutine `QMDUPD`. Afterwards, the threshold value is also updated.

Before the program loops back for the next node of minimum degree, the quotient graph transformation is performed by the subroutine `QMDQT`. The program exits when all the nodes in the graph have been numbered.

---

```

1. C*****
2. C*****
3. C*****      GENQMD . . . . QUOT MIN DEGREE ORDERING      *****
4. C*****
  
```

```

5.  C*****
6.  C
7.  C   PURPOSE - THIS ROUTINE IMPLEMENTS THE MINIMUM DEGREE
8.  C   ALGORITHM. IT MAKES USE OF THE IMPLICIT REPRESENT-
9.  C   ATION OF THE ELIMINATION GRAPHS BY QUOTIENT GRAPHS,
10. C   AND THE NOTION OF INDISTINGUISHABLE NODES.
11. C   CAUTION - THE ADJACENCY VECTOR ADJNCY WILL BE
12. C   DESTROYED.
13. C
14. C   INPUT PARAMETERS -
15. C   NEQNS - NUMBER OF EQUATIONS.
16. C   (XADJ, ADJNCY) - THE ADJACENCY STRUCTURE.
17. C
18. C   OUTPUT PARAMETERS -
19. C   PERM - THE MINIMUM DEGREE ORDERING.
20. C   INVP - THE INVERSE OF PERM.
21. C
22. C   WORKING PARAMETERS -
23. C   DEG - THE DEGREE VECTOR. DEG(I) IS NEGATIVE MEANS
24. C   NODE I HAS BEEN NUMBERED.
25. C   MARKER - A MARKER VECTOR, WHERE MARKER(I) IS
26. C   NEGATIVE MEANS NODE I HAS BEEN MERGED WITH
27. C   ANOTHER NODE AND THUS CAN BE IGNORED.
28. C   RCHSET - VECTOR USED FOR THE REACHABLE SET.
29. C   NBRHD - VECTOR USED FOR THE NEIGHBORHOOD SET.
30. C   QSIZE - VECTOR USED TO STORE THE SIZE OF
31. C   INDISTINGUISHABLE SUPERNODES.
32. C   QLINK - VECTOR TO STORE INDISTINGUISHABLE NODES,
33. C   I, QLINK(I), QLINK(QLINK(I)) ... ARE THE
34. C   MEMBERS OF THE SUPERNODE REPRESENTED BY I.
35. C
36. C   PROGRAM SUBROUTINES -
37. C   QMDRCH, QMDQT, QMDUPD.
38. C
39. C*****
40. C
41. C
42. C   SUBROUTINE GENQMD ( NEQNS, XADJ, ADJNCY, PERM, INVP, DEG,
43. C   1   MARKER, RCHSET, NBRHD, QSIZE, QLINK,
44. C   1   NOFSUB )
45. C
46. C*****
47. C
48. C   INTEGER ADJNCY(1), PERM(1), INVP(1), DEG(1), MARKER(1),
49. C   1   RCHSET(1), NBRHD(1), QSIZE(1), QLINK(1)
50. C   INTEGER XADJ(1), INODE, IP, IRCH, J, MINDEG, NDEG,
51. C   1   NEQNS, NHDSZE, NODE, NOFSUB, NP, NUM, NUMP1,

```

```

52.      1          NXNODE, RCHSZE, SEARCH, THRESH
53.      C
54.      C*****
55.      C
56.      C          -----
57.      C          INITIALIZE DEGREE VECTOR AND OTHER WORKING VARIABLES.
58.      C          -----
59.      C          MINDEG = NEQNS
60.      C          NOFSUB = 0
61.      C          DO 100 NODE = 1, NEQNS
62.      C              PERM(NODE) = NODE
63.      C              INVP(NODE) = NODE
64.      C              MARKER(NODE) = 0
65.      C              QSIZE(NODE) = 1
66.      C              QLINK(NODE) = 0
67.      C              NDEG = XADJ(NODE+1) - XADJ(NODE)
68.      C              DEG(NODE) = NDEG
69.      C              IF ( NDEG .LT. MINDEG ) MINDEG = NDEG
70.      C          100 CONTINUE
71.      C          NUM = 0
72.      C          -----
73.      C          PERFORM THRESHOLD SEARCH TO GET A NODE OF MIN DEGREE.
74.      C          VARIABLE SEARCH POINTS TO WHERE SEARCH SHOULD START.
75.      C          -----
76.      C          200 SEARCH = 1
77.      C              THRESH = MINDEG
78.      C              MINDEG = NEQNS
79.      C          300 NUMP1 = NUM + 1
80.      C              IF ( NUMP1 .GT. SEARCH ) SEARCH = NUMP1
81.      C              DO 400 J = SEARCH, NEQNS
82.      C                  NODE = PERM(J)
83.      C                  IF ( MARKER(NODE) .LT. 0 ) GOTO 400
84.      C                  NDEG = DEG(NODE)
85.      C                  IF ( NDEG .LE. THRESH ) GO TO 500
86.      C                  IF ( NDEG .LT. MINDEG ) MINDEG = NDEG
87.      C          400 CONTINUE
88.      C          GO TO 200
89.      C          -----
90.      C          NODE HAS MINIMUM DEGREE. FIND ITS REACHABLE SETS BY
91.      C          CALLING QMDRCH.
92.      C          -----
93.      C          500 SEARCH = J
94.      C              NOFSUB = NOFSUB + DEG(NODE)
95.      C              MARKER(NODE) = 1
96.      C              CALL QMDRCH (NODE, XADJ, ADJNCY, DEG, MARKER,
97.      C          1              RCHSZE, RCHSET, NHDSZE, NBRHD )
98.      C          -----

```

```

 99. C          ELIMINATE ALL NODES INDISTINGUISHABLE FROM NODE.
100. C          THEY ARE GIVEN BY NODE, QLINK(NODE), ....
101. C          -----
102. C          NXNODE = NODE
103.   600      NUM = NUM + 1
104. C          NP = INVP(NXNODE)
105. C          IP = PERM(NUM)
106. C          PERM(NP) = IP
107. C          INVP(IP) = NP
108. C          PERM(NUM) = NXNODE
109. C          INVP(NXNODE) = NUM
110. C          DEG(NXNODE) = - 1
111. C          NXNODE = QLINK(NXNODE)
112. C          IF (NXNODE .GT. 0) GOTO 600
113. C
114. C          IF ( RCHSZE .LE. 0 ) GO TO 800
115. C          -----
116. C          UPDATE THE DEGREES OF THE NODES IN THE REACHABLE
117. C          SET AND IDENTIFY INDISTINGUISHABLE NODES.
118. C          -----
119. C          CALL QMDUPD ( XADJ, ADJNCY, RCHSZE, RCHSET, DEG,
120.   1          QSIZE, QLINK, MARKER, RCHSET(RCHSZE+1),
121.   1          NBRHD(NHDSZE+1) )
122. C          -----
123. C          RESET MARKER VALUE OF NODES IN REACH SET.
124. C          UPDATE THRESHOLD VALUE FOR CYCLIC SEARCH.
125. C          ALSO CALL QMDQT TO FORM NEW QUOTIENT GRAPH.
126. C          -----
127. C          MARKER(NODE) = 0
128. C          DO 700 IRCH = 1, RCHSZE
129. C             INODE = RCHSET(IRCH)
130. C             IF ( MARKER(INODE) .LT. 0 ) GOTO 700
131. C             MARKER(INODE) = 0
132. C             NDEG = DEG(INODE)
133. C             IF ( NDEG .LT. MINDEG ) MINDEG = NDEG
134. C             IF ( NDEG .GT. THRESH ) GOTO 700
135. C             MINDEG = THRESH
136. C             THRESH = NDEG
137. C             SEARCH = INVP(INODE)
138.   700      CONTINUE
139. C          IF ( NHDSZE .GT. 0 ) CALL QMDQT ( NODE, XADJ,
140.   1          ADJNCY, MARKER, RCHSZE, RCHSET, NBRHD )
141.   800      IF ( NUM .LT. NEQNS ) GO TO 300
142. C          RETURN
143. C          END

```

---



```

31. C
32. C*****
33. C
34.     INTEGER ADJNCY(1), DEG(1), MARKER(1),
35.     1       RCHSET(1), NBRHD(1)
36.     INTEGER XADJ(1), I, ISTRT, ISTOP, J, JSTRT, JSTOP,
37.     1       NABOR, NHDSZE, NODE, RCHSZE, ROOT
38. C
39. C*****
40. C
41. C     -----
42. C     LOOP THROUGH THE NEIGHBORS OF ROOT IN THE
43. C     QUOTIENT GRAPH.
44. C     -----
45.     NHDSZE = 0
46.     RCHSZE = 0
47.     ISTRT = XADJ(ROOT)
48.     ISTOP = XADJ(ROOT+1) - 1
49.     IF ( ISTOP .LT. ISTRT ) RETURN
50.     DO 600 I = ISTRT, ISTOP
51.         NABOR = ADJNCY(I)
52.         IF ( NABOR .EQ. 0 ) RETURN
53.         IF ( MARKER(NABOR) .NE. 0 ) GO TO 600
54.         IF ( DEG(NABOR) .LT. 0 ) GO TO 200
55. C     -----
56. C     INCLUDE NABOR INTO THE REACHABLE SET.
57. C     -----
58.         RCHSZE = RCHSZE + 1
59.         RCHSET(RCHSZE) = NABOR
60.         MARKER(NABOR) = 1
61.         GO TO 600
62. C     -----
63. C     NABOR HAS BEEN ELIMINATED. FIND NODES
64. C     REACHABLE FROM IT.
65. C     -----
66.     200     MARKER(NABOR) = -1
67.             NHDSZE = NHDSZE + 1
68.             NBRHD(NHDSZE) = NABOR
69.     300     JSTRT = XADJ(NABOR)
70.             JSTOP = XADJ(NABOR+1) - 1
71.             DO 500 J = JSTRT, JSTOP
72.                 NODE = ADJNCY(J)
73.                 NABOR = - NODE
74.                 IF (NODE) 300, 600, 400
75.     400     IF ( MARKER(NODE) .NE. 0 ) GO TO 500
76.             RCHSZE = RCHSZE + 1
77.             RCHSET(RCHSZE) = NODE

```



```

78 .                MARKER(NODE) = 1
79 .    500          CONTINUE
80 .    600          CONTINUE
81 .                RETURN
82 .                END

```

---

### QMDQT (Quotient MD Quotient graph Transformation)

This subroutine performs the quotient graph transformation on the adjacency structure (XADJ, ADJNCY). The new eliminated supernode contains the node ROOT and the nodes in the array NBRHD, and it will be represented by ROOT in the new structure. Its adjacent set in the new quotient graph is given in (RCHSZE, RCHSET).

After initialization, the new adjacent set in (RCHSZE, RCHSET) will be placed in the adjacency list of ROOT in the structure (D0 200 ...). If there is not enough space, the program will use the space provided by the nodes in the set NBRHD. We know from Section 5.3.3 that there are always enough storage locations.

Before exit, the representative node ROOT is added to the neighbor list of each node in RCHSET. This is done in the D0 600 ... loop.

---

```

1. C*****
2. C*****
3. C***** QMDQT ..... QUOT MIN DEG QUOT TRANSFORM *****
4. C*****
5. C*****
6. C
7. C PURPOSE - THIS SUBROUTINE PERFORMS THE QUOTIENT GRAPH
8. C TRANSFORMATION AFTER A NODE HAS BEEN ELIMINATED.
9. C
10. C INPUT PARAMETERS -
11. C ROOT - THE NODE JUST ELIMINATED. IT BECOMES THE
12. C REPRESENTATIVE OF THE NEW SUPERNODE.
13. C (XADJ, ADJNCY) - THE ADJACENCY STRUCTURE.
14. C (RCHSZE, RCHSET) - THE REACHABLE SET OF ROOT IN THE
15. C OLD QUOTIENT GRAPH.
16. C NBRHD - THE NEIGHBORHOOD SET WHICH WILL BE MERGED
17. C WITH ROOT TO FORM THE NEW SUPERNODE.
18. C MARKER - THE MARKER VECTOR.
19. C
20. C UPDATED PARAMETER -
21. C ADJNCY - BECOMES THE ADJNCY OF THE QUOTIENT GRAPH.
22. C

```

```

23. C*****
24. C
25.     SUBROUTINE QMDQT ( ROOT, XADJ, ADJNCY, MARKER,
26.     1                RCHSZE, RCHSET, NBRHD )
27. C
28. C*****
29. C
30.     INTEGER ADJNCY(1), MARKER(1), RCHSET(1), NBRHD(1)
31.     INTEGER XADJ(1), INHD, IRCH, J, JSTRT, JSTOP, LINK,
32.     1        NABOR, NODE, RCHSZE, ROOT
33. C
34. C*****
35. C
36.     IRCH = 0
37.     INHD = 0
38.     NODE = ROOT
39.     100  JSTRT = XADJ(NODE)
40.         JSTOP = XADJ(NODE+1) - 2
41.         IF ( JSTOP .LT. JSTRT ) GO TO 300
42. C
43. C     PLACE REACH NODES INTO THE ADJACENT LIST OF NODE
44. C
45.     DO 200 J = JSTRT, JSTOP
46.         IRCH = IRCH + 1
47.         ADJNCY(J) = RCHSET(IRCH)
48.         IF ( IRCH .GE. RCHSZE ) GOTO 400
49.     200  CONTINUE
50. C
51. C     LINK TO OTHER SPACE PROVIDED BY THE NBRHD SET.
52. C
53.     300  LINK = ADJNCY(JSTOP+1)
54.         NODE = - LINK
55.         IF ( LINK .LT. 0 ) GOTO 100
56.         INHD = INHD + 1
57.         NODE = NBRHD(INHD)
58.         ADJNCY(JSTOP+1) = - NODE
59.         GO TO 100
60. C
61. C     ALL REACHABLE NODES HAVE BEEN SAVED.  END THE ADJ LIST.
62. C     ADD ROOT TO THE NBR LIST OF EACH NODE IN THE REACH SET.
63. C
64.     400  ADJNCY(J+1) = 0
65.         DO 600 IRCH = 1, RCHSZE
66.             NODE = RCHSET(IRCH)
67.             IF ( MARKER(NODE) .LT. 0 ) GOTO 600
68.             JSTRT = XADJ(NODE)
69.             JSTOP = XADJ(NODE+1) - 1

```

```

70.                DO 500 J = JSTRT, JSTOP
71.                NABOR = ADJNCY(J)
72.                IF ( MARKER(NABOR) .GE. 0 ) GO TO 500
73.                ADJNCY(J) = ROOT
74.                GOTO 600
75.    500          CONTINUE
76.    600          CONTINUE
77.                RETURN
78.                END

```

---

### QMDUPD (Quotient MD UPDate)

This subroutine performs the degree update step in the minimum degree algorithm. The nodes whose new degrees are to be determined are given by the pair (NLIST, LIST). The subroutine also merges indistinguishable nodes in this subset by using Theorem 5.4.7.

The first loop DO 200 ... and the call to the subroutine QMDMRG determine groups of indistinguishable nodes in the given set. They will be merged together and have their degrees updated.

For those nodes not being merged, the loop DO 600 ... determines their new degrees by calling the subroutine QMDRCH. The vectors RCHSET and NBRHD are used as temporary arrays.

---

```

1.  C*****
2.  C*****
3.  C*****      QMDUPD . . . . QUOT MIN DEG UPDATE      *****
4.  C*****
5.  C*****
6.  C
7.  C      PURPOSE - THIS ROUTINE PERFORMS DEGREE UPDATE FOR A SET
8.  C      OF NODES IN THE MINIMUM DEGREE ALGORITHM.
9.  C
10. C      INPUT PARAMETERS -
11. C      (XADJ, ADJNCY) - THE ADJACENCY STRUCTURE.
12. C      (NLIST, LIST) - THE LIST OF NODES WHOSE DEGREE HAS TO
13. C      BE UPDATED.
14. C
15. C      UPDATED PARAMETERS -
16. C      DEG - THE DEGREE VECTOR.
17. C      QSIZE - SIZE OF INDISTINGUISHABLE SUPERNODES.
18. C      QLINK - LINKED LIST FOR INDISTINGUISHABLE NODES.
19. C      MARKER - USED TO MARK THOSE NODES IN REACH/NBRHD SETS.
20. C
21. C      WORKING PARAMETERS -

```

```

22. C      RCHSET - THE REACHABLE SET.
23. C      NBRHD - THE NEIGHBORHOOD SET.
24. C
25. C      PROGRAM SUBROUTINES -
26. C      QMDMRG.
27. C
28. C*****
29. C
30. C      SUBROUTINE QMDUPD ( XADJ, ADJNCY, NLIST, LIST, DEG,
31. C      1                QSIZE, QLINK, MARKER, RCHSET, NBRHD )
32. C
33. C*****
34. C
35. C      INTEGER ADJNCY(1), LIST(1), DEG(1), MARKER(1),
36. C      1                RCHSET(1), NBRHD(1), QSIZE(1), QLINK(1)
37. C      INTEGER XADJ(1), DEGO, DEG1, IL, INHD, INODE, IRCH,
38. C      1                J, JSTRT, JSTOP, MARK, NABOR, NHDSZE, NLIST,
39. C      1                NODE, RCHSZE, ROOT
40. C
41. C*****
42. C
43. C      -----
44. C      FIND ALL ELIMINATED SUPERNODES THAT ARE ADJACENT
45. C      TO SOME NODES IN THE GIVEN LIST. PUT THEM INTO
46. C      (NHDSZE, NBRHD). DEGO CONTAINS THE NUMBER OF
47. C      NODES IN THE LIST.
48. C      -----
49. C      IF ( NLIST .LE. 0 ) RETURN
50. C      DEGO = 0
51. C      NHDSZE = 0
52. C      DO 200 IL = 1, NLIST
53. C          NODE = LIST(IL)
54. C          DEGO = DEGO + QSIZE(NODE)
55. C          JSTRT = XADJ(NODE)
56. C          JSTOP = XADJ(NODE+1) - 1
57. C          DO 100 J = JSTRT, JSTOP
58. C              NABOR = ADJNCY(J)
59. C              IF ( MARKER(NABOR) .NE. 0 .OR.
60. C      1                DEG(NABOR) .GE. 0 ) GO TO 100
61. C                  MARKER(NABOR) = - 1
62. C                  NHDSZE = NHDSZE + 1
63. C                  NBRHD(NHDSZE) = NABOR
64. C      100          CONTINUE
65. C      200          CONTINUE
66. C      -----
67. C      MERGE INDISTINGUISHABLE NODES IN THE LIST BY
68. C      CALLING THE SUBROUTINE QMDMRG.

```

```

69. C -----
70. IF ( NHDSZE .GT. 0 )
71.   1 CALL QMDMRG ( XADJ, ADJNCY, DEG, QSIZE, QLINK,
72.   1           MARKER, DEGO, NHDSZE, NBRHD, RCHSET,
73.   1           NBRHD(NHDSZE+1) )
74. C -----
75. C FIND THE NEW DEGREES OF THE NODES THAT HAVE NOT BEEN
76. C MERGED.
77. C -----
78. DO 600 IL = 1, NLIST
79.   NODE = LIST(IL)
80.   MARK = MARKER(NODE)
81.   IF ( MARK .GT. 1 .OR. MARK .LT. 0 ) GO TO 600
82.   MARKER(NODE) = 2
83.   CALL QMDRCH ( NODE, XADJ, ADJNCY, DEG, MARKER,
84.   1           RCHSZE, RCHSET, NHDSZE, NBRHD )
85.   DEG1 = DEGO
86.   IF ( RCHSZE .LE. 0 ) GO TO 400
87.   DO 300 IRCH = 1, RCHSZE
88.     INODE = RCHSET(IRCH)
89.     DEG1 = DEG1 + QSIZE(INODE)
90.     MARKER(INODE) = 0
91.   300 CONTINUE
92.   400 DEG(NODE) = DEG1 - 1
93.   IF ( NHDSZE .LE. 0 ) GO TO 600
94.   DO 500 INHD = 1, NHDSZE
95.     INODE = NBRHD(INHD)
96.     MARKER(INODE) = 0
97.   500 CONTINUE
98.   600 CONTINUE
99.   RETURN
100.  END

```

### QMDMRG (Quotient MD MeRGe)

This subroutine implements a check for the condition (5.4.2) to determine indistinguishable nodes. Let  $C_1, C_2, R_1, R_2$  and  $Y$  be as in Lemma 5.4.6. The subroutine assumes that  $C_1$  and  $R_1$  have already been determined elsewhere. Nodes in  $R_1$  have their `MARKER` values set to 1.

There may be more than one  $C_2$  input to `QMDMRG`. They are contained in `(NHDSZE, NBRHD)`, where each `NBRHD(i)` specifies one eliminated supernode (that is, connected component).

The loop `DO 1400 ...` applies the condition on each given connected component. It first determines the set  $R_2 - R_1$  in `(RCHSZE, RCHSET)` and the

intersection set  $R_2 \cap R_1$  in (NOVRLP, OVRLP) in the loop DO 600 . . . . For each node in the intersection, the condition (5.4.2) is tested in the loop DO 1100 . . . . If the condition is satisfied, the node is included in the merged supernode by placing it in the QLINK vector. The size of the new supernode is also computed.

---

```

1. C*****
2. C*****
3. C*****      QMDMRG . . . . QUOT MIN DEG MERGE      *****
4. C*****
5. C*****
6. C
7. C      PURPOSE - THIS ROUTINE MERGES INDISTINGUISHABLE NODES IN
8. C                THE MINIMUM DEGREE ORDERING ALGORITHM.
9. C                IT ALSO COMPUTES THE NEW DEGREES OF THESE
10. C               NEW SUPERNODES .
11. C
12. C      INPUT PARAMETERS -
13. C          (XADJ, ADJNCY) - THE ADJACENCY STRUCTURE.
14. C          DEGO - THE NUMBER OF NODES IN THE GIVEN SET.
15. C          (NHDSZE, NBRHD) - THE SET OF ELIMINATED SUPERNODES
16. C                ADJACENT TO SOME NODES IN THE SET.
17. C
18. C      UPDATED PARAMETERS -
19. C          DEG - THE DEGREE VECTOR.
20. C          QSIZE - SIZE OF INDISTINGUISHABLE NODES.
21. C          QLINK - LINKED LIST FOR INDISTINGUISHABLE NODES.
22. C          MARKER - THE GIVEN SET IS GIVEN BY THOSE NODES WITH
23. C                MARKER VALUE SET TO 1. THOSE NODES WITH DEGREE
24. C                UPDATED WILL HAVE MARKER VALUE SET TO 2.
25. C
26. C      WORKING PARAMETERS -
27. C          RCHSET - THE REACHABLE SET.
28. C          OVRLP - TEMP VECTOR TO STORE THE INTERSECTION OF TWO
29. C                REACHABLE SETS.
30. C
31. C*****
32. C
33. C      SUBROUTINE QMDMRG ( XADJ, ADJNCY, DEG, QSIZE, QLINK,
34. C          1              MARKER, DEGO, NHDSZE, NBRHD, RCHSET,
35. C          1              OVRLP )
36. C
37. C*****
38. C
39. C      INTEGER  ADJNCY(1), DEG(1), QSIZE(1), QLINK(1),
40. C          1      MARKER(1), RCHSET(1), NBRHD(1), OVRLP(1)
41. C      INTEGER  XADJ(1), DEGO, DEG1, HEAD, INHD, IOV, IRCH,
```

```

42.      1          J, JSTRT, JSTOP, LINK, LNODE, MARK, MRGSZE,
43.      1          NABOR, NHDSZE, NODE, NOVRLP, RCHSZE, ROOT
44.  C
45.  C*****
46.  C
47.  C      -----
48.  C      INITIALIZATION ...
49.  C      -----
50.      IF ( NHDSZE .LE. 0 ) RETURN
51.      DO 100 INHD = 1, NHDSZE
52.          ROOT = NBRHD(INHD)
53.          MARKER(ROOT) = 0
54. 100    CONTINUE
55.  C      -----
56.  C      LOOP THROUGH EACH ELIMINATED SUPERNODE IN THE SET
57.  C      (NHDSZE, NBRHD).
58.  C      -----
59.      DO 1400 INHD = 1, NHDSZE
60.          ROOT = NBRHD(INHD)
61.          MARKER(ROOT) = - 1
62.          RCHSZE = 0
63.          NOVRLP = 0
64.          DEG1 = 0
65. 200    JSTRT = XADJ(ROOT)
66.          JSTOP = XADJ(ROOT+1) - 1
67.  C      -----
68.  C      DETERMINE THE REACHABLE SET AND ITS INTERSECT-
69.  C      ION WITH THE INPUT REACHABLE SET.
70.  C      -----
71.      DO 600 J = JSTRT, JSTOP
72.          NABOR = ADJNCY(J)
73.          ROOT = - NABOR
74.          IF (NABOR) 200, 700, 300
75.  C
76. 300    MARK = MARKER(NABOR)
77.          IF ( MARK ) 600, 400, 500
78. 400    RCHSZE = RCHSZE + 1
79.          RCHSET(RCHSZE) = NABOR
80.          DEG1 = DEG1 + QSIZE(NABOR)
81.          MARKER(NABOR) = 1
82.          GOTO 600
83. 500    IF ( MARK .GT. 1 ) GOTO 600
84.          NOVRLP = NOVRLP + 1
85.          OVRLP(NOVRLP) = NABOR
86.          MARKER(NABOR) = 2
87. 600    CONTINUE
88.  C      -----

```

```

89.  C          FROM THE OVERLAPPED SET, DETERMINE THE NODES
90.  C          THAT CAN BE MERGED TOGETHER.
91.  C          -----
92.  700      HEAD = 0
93.          MRGSZ = 0
94.          DO 1100 IOV = 1, NOVRLP
95.             NODE = OVRLP(IOV)
96.             JSTRT = XADJ(NODE)
97.             JSTOP = XADJ(NODE+1) - 1
98.             DO 800 J = JSTRT, JSTOP
99.                NABOR = ADJNCY(J)
100.                IF ( MARKER(NABOR) .NE. 0 ) GOTO 800
101.                MARKER(NODE) = 1
102.                GOTO 1100
103.  800      CONTINUE
104.  C          -----
105.  C          NODE BELONGS TO THE NEW MERGED SUPERNODE.
106.  C          UPDATE THE VECTORS QLINK AND QSIZE.
107.  C          -----
108.          MRGSZ = MRGSZ + QSIZE(NODE)
109.          MARKER(NODE) = - 1
110.          LNODE = NODE
111.  900      LINK = QLINK(LNODE)
112.          IF ( LINK .LE. 0 ) GOTO 1000
113.          LNODE = LINK
114.          GOTO 900
115.  1000     QLINK(LNODE) = HEAD
116.          HEAD = NODE
117.  1100     CONTINUE
118.          IF ( HEAD .LE. 0 ) GOTO 1200
119.          QSIZE(HEAD) = MRGSZ
120.          DEG(HEAD) = DEGO + DEG1 - 1
121.          MARKER(HEAD) = 2
122.  C          -----
123.  C          RESET MARKER VALUES.
124.  C          -----
125.  1200     ROOT = NBRHD(INHD)
126.          MARKER(ROOT) = 0
127.          IF ( RCHSZ .LE. 0 ) GOTO 1400
128.          DO 1300 IRCH = 1, RCHSZ
129.             NODE = RCHSET(IRCH)
130.             MARKER(NODE) = 0
131.  1300     CONTINUE
132.  1400     CONTINUE
133.          RETURN
134.          END

```

---



**Exercises**

- 5.4.1) Let  $x_i$  be the node selected from  $\mathcal{G}_{i-1}$  in the minimum degree algorithm. Let  $y \in \text{Adj}_{\mathcal{G}_{i-1}}(x_i)$  with

$$\text{Deg}_{\mathcal{G}_i}(y) = \text{Deg}_{\mathcal{G}_{i-1}}(x_i) - 1.$$

Show that  $y$  is a node of minimum degree in  $\mathcal{G}_i$ .

- 5.4.2) Let  $x_i$  and  $\mathcal{G}_{i-1}$  be as in Exercise 5.4.1 on page 160, and  $y \in \text{Adj}_{\mathcal{G}_{i-1}}(x_i)$ . Prove that if

$$\text{Adj}_{\mathcal{G}_{i-1}}(y) \subset \text{Adj}_{\mathcal{G}_{i-1}}(x_i) \cup \{x_i\}$$

then  $y$  is a node of minimum degree in  $\mathcal{G}_i$ .

**5.5 Sparse Storage Schemes****5.5.1 The Uncompressed Scheme**

The data structure for the general sparse methods should only store (logical) nonzeros of the factored matrix. The scheme discussed here is oriented to the inner-product formulation of the factorization algorithm (see Section 2.2.2) and can be found in, for example, Gustavson (1972) and Sherman (1975).

The scheme has a main storage array LNZ which contains all the nonzero entries in the lower triangular factor. A storage location is provided for each *logical* nonzero in the factor. The nonzeros in  $\mathbf{L}$ , excluding the diagonal, are stored column after column in LNZ. An accompanying vector NZSUB is provided, which gives the row subscripts of the nonzeros. In addition, an index vector XLNZ is used to point to the start of nonzeros in each column in LNZ (or equivalently NZSUB). The diagonal entries are stored separately in the vector DIAG.

To access a nonzero component  $a_{ij}$  or  $l_{ij}$ , there is no direct method of calculating the corresponding index in the vector LNZ. Some testing on the subscripts in NZSUB has to be done. The following portion of a program can be used for that purpose. Note that any entry not represented by the data structure is zero.

```

KSTRT = XLNZ(J)
KSTOP = XLNZ(J+1) - 1
AIJ = 0.0
IF (KSTOP.LT.KSTRT) GO TO 300

```

$$\mathbf{A} = \begin{pmatrix} a_{11} & & & & & & \\ a_{21} & a_{22} & & & & & \\ & & a_{33} & & & & \\ a_{41} & & & a_{44} & & & \\ & & a_{53} & a_{54} & a_{55} & & \\ & & a_{63} & & & a_{66} & \\ & & & & a_{75} & a_{76} & a_{77} \end{pmatrix}$$

$$\mathbf{L} = \begin{pmatrix} l_{11} & & & & & & \\ l_{21} & l_{22} & & & & & \\ & & l_{33} & & & & \\ l_{41} & l_{42} & & l_{44} & & & \\ & & l_{53} & l_{54} & l_{55} & & \\ & & l_{63} & & l_{65} & l_{66} & \\ & & & & l_{75} & l_{76} & l_{77} \end{pmatrix}$$

Figure 5.5.1: A 7 by 7 matrix  $\mathbf{A}$  and its factor  $\mathbf{L}$ .

```

      DO 100 K = KSTRT, KSTOP
          IF (NZSUB(K).EQ.I) GO TO 200
100    CONTINUE
          GO TO 300
200    AIJ = LNZ(K)
300    .
      .
      .

```

Although this scheme is not particularly well suited for random access of nonzero entries, it lends itself quite readily to sparse factorization and solution. The primary storage of the scheme is  $|\text{Nonz}(\mathbf{F})| + n$  for the vectors LNZ and DIAG, and the overhead storage is  $|\text{Nonz}(\mathbf{F})| + n$  for NZSUB and XLNZ.

### 5.5.2 Compressed Scheme

This scheme, which is a modification of the uncompressed scheme, is due to Sherman [47]. The motivation can be provided by considering the minimum

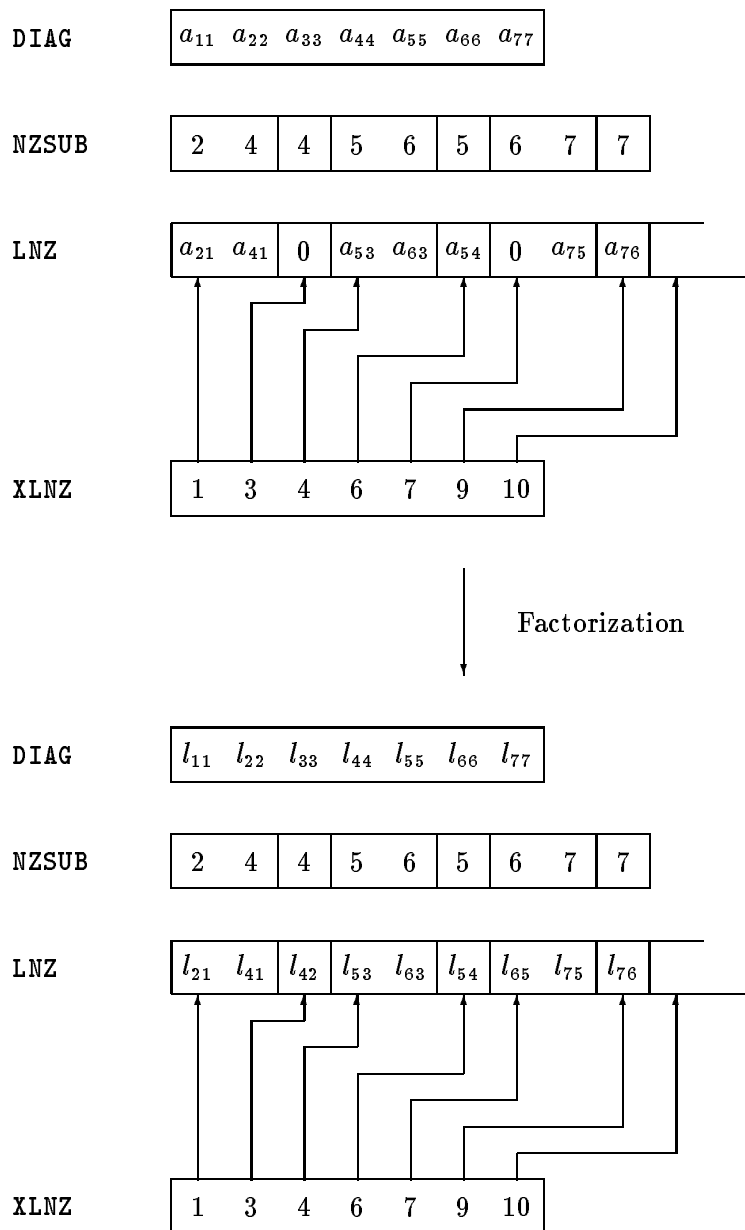


Figure 5.5.2: Uncompressed data storage scheme for the matrix and its factor in Figure 5.5.1.

---

degree ordering as discussed in Section 5.4.3. We saw that it was possible to simultaneously number or eliminate a *set*  $Y$  of nodes. The nodes in  $Y$  satisfy the indistinguishable condition

$$\text{Reach}(x, S) \cup \{x\} = \text{Reach}(y, S) \cup \{y\},$$

for all  $x, y \in Y$ . In terms of the matrix factor  $L$ , this means all the row subscripts below the block corresponding to  $Y$  are identical, as shown in Figure 5.5.3.

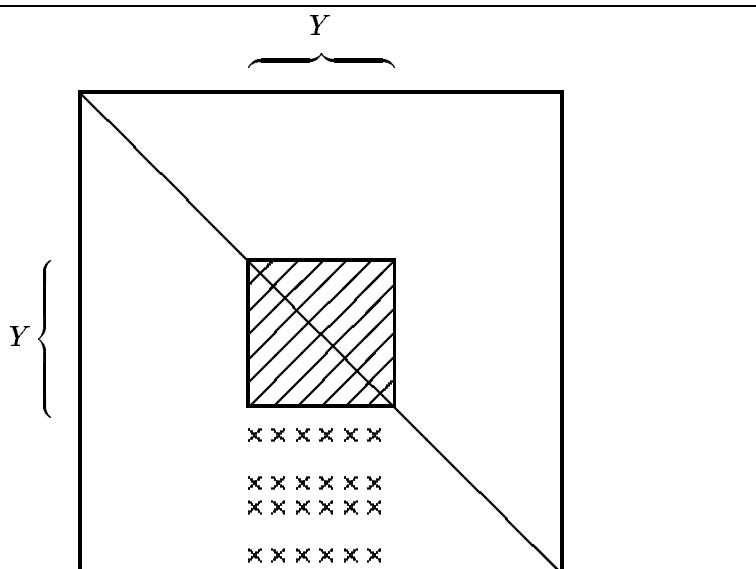


Figure 5.5.3: Motivation for the compressed storage scheme.

---

If the structure is stored using the uncompressed scheme, the row subscripts of all but the first column in this block are final subsequences of that of the previous column. Naturally, the subscript vector  $\text{NZSUB}$  can be compressed so that redundant information is not stored. It is done by removing the row subscripts for a column if they appear as a final subsequence of the previous column.

In exchange for the compression, we need to have an auxiliary index vector  $\text{XNZSUB}$  which points to the start of row subscripts in  $\text{NZSUB}$  for each column. The compressed scheme for the example in Figure 5.5.1 is shown in Figure 5.5.4.

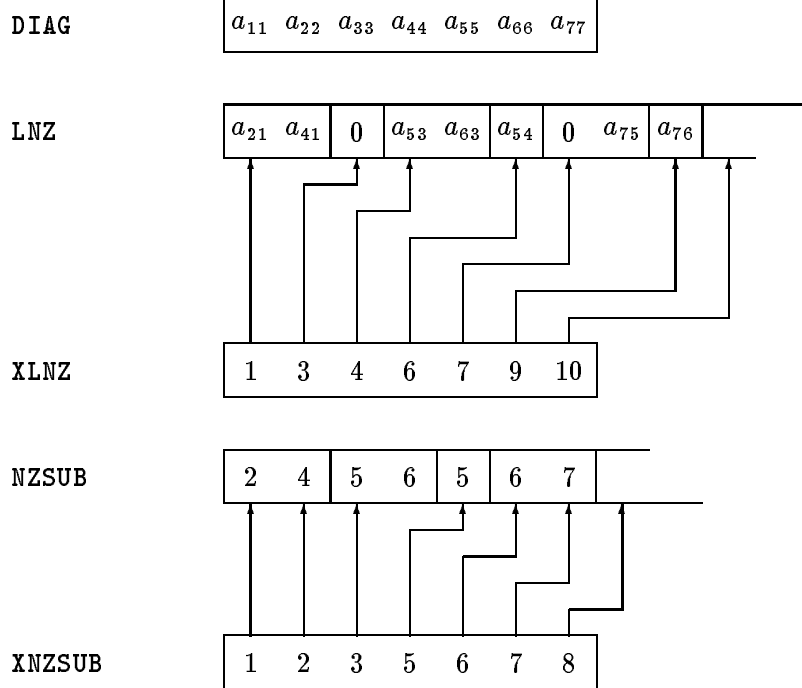


Figure 5.5.4: Compressed storage scheme for the matrix in Figure 5.5.1.

---

In this case, the way to access a nonzero entry in the  $(i, j)$ -th position is as follows.

```

      KSTRT = XLNZ(J)
      KSTOP = XLNZ(J+1) - 1
      AIJ = 0.0
      IF (KSTOP.LT.KSTRT) GO TO 300
      KSUB = XNZSUB(J)
      DO 100 K = KSTRT, KSTOP
          IF(NZSUB(KSUB).EQ.I) GO TO 200
          KSUB = KSUB + 1
100    CONTINUE
      GO TO 300
200  AIJ = LNZ(K)
300    .
      .
      .

```

In the compressed scheme, the primary storage remains the same, but the overhead storage is changed and it is less than or equal to  $|Nonz(\mathbf{F})| + 2n$ . The example given is too small to bring out the significance of the compressed scheme. In Table 5.5.1, we provide some numbers that are obtained from nine larger problems which comprise one of the test sets considered in Chapter 9. The ordering which was used in generating these results was provided by a minimum degree algorithm similar to the one described in the previous section. Typically, for problems of this size and larger, the overhead storage is reduced by at least fifty percent, compared to the uncompressed scheme.

### 5.5.3 On Symbolic Factorization

As its name implies, *symbolic factorization* is the process of simulating the numerical factorization of a given matrix  $\mathbf{A}$  in order to obtain the zero-nonzero structure of its factor  $\mathbf{L}$ . Since the numerical values of the matrix components are of no significance in this connection, the problem can be conveniently studied using a graph theory approach.

Let  $\mathcal{G}^\alpha = (X^\alpha, E)$  be an ordered graph, where  $|X^\alpha| = n$  and for convenience let  $\alpha(i) = x_i$ . In view of Theorem 5.2.2, symbolic factorization may be regarded as determination of the sets

$$Reach(x_i, \{x_1, \dots, x_{i-1}\}), \quad i = 1, \dots, n.$$

Number of Equations	$ Nonz(\mathbf{A}) $	$ Nonz(\mathbf{F}) $	Overhead for Uncompressed	Overhead for Compressed
936	2664	13870	14806	6903
1009	2928	19081	20090	8085
1089	3136	18626	19715	8574
1440	4032	19047	20487	10536
1180	3285	14685	15865	8436
1377	3808	16793	18170	9790
1138	3156	15592	16730	8326
1141	3162	15696	16837	8435
1349	3876	23726	25075	10666

Table 5.5.1: Comparison of uncompressed and compressed storage schemes. The primary storage is equal to the overhead for the uncompressed scheme.

Define  $S_i = \{x_1, \dots, x_i\}$ .

We prove the following result about reachable sets.

**Lemma 5.5.1**

$$Reach(x_i, S_{i-1}) = Adj(x_i) \cup (\cup \{Reach(x_k, S_{k-1}) \mid x_i \in Reach(x_k, S_{k-1})\}) - S_i.$$

**Proof:** Let  $j > i$ . Then by Lemma 5.2.1 and Theorem 5.2.2

$$\begin{aligned} x_j \in Reach(x_i, S_{i-1}) &\iff \{x_i, x_j\} \in E^{\mathbf{F}} \\ &\iff \{x_i, x_j\} \in E^{\mathbf{A}}, \text{ or } \{x_i, x_k\} \in E^{\mathbf{F}} \text{ and } \{x_j, x_k\} \in E^{\mathbf{F}} \text{ for some } k < j \\ &\iff x_j \in Adj(x_i) \text{ or } x_i, x_j \in Reach(x_k, S_{k-1}) \text{ for some } k. \end{aligned}$$

The lemma then follows.  $\square$

Lemma 5.5.1 suggests an algorithm for finding the reachable sets (and hence the structure of the factor  $\mathbf{L}$ ). It may be described as follows.

**Step 1** (*Initialization*) **for**  $k = 1, \dots, n$  **do**

$$Reach(x_k, S_{k-1}) \leftarrow Adj(x_k) - S_{k-1}.$$

**Step 2** (*Symbolic factorization*)

```

for  $k = 1, 2, \dots, n$  do
  if  $x_i \in \text{Reach}(x_k, S_{k-1})$  then
     $\text{Reach}(x_i, S_{i-1}) \leftarrow \text{Reach}(x_i, S_{i-1}) \cup$ 
       $\text{Reach}(x_k, S_{k-1}) - S_i.$ 

```

A pictorial illustration of the scheme is shown in Figure 5.5.5. This scheme is hardly satisfactory, since it essentially simulates the entire factorization, and its cost will be proportional to the operation count as given in Theorem 2.2.2. Let us look into possible ways of improving the efficiency of the algorithm.

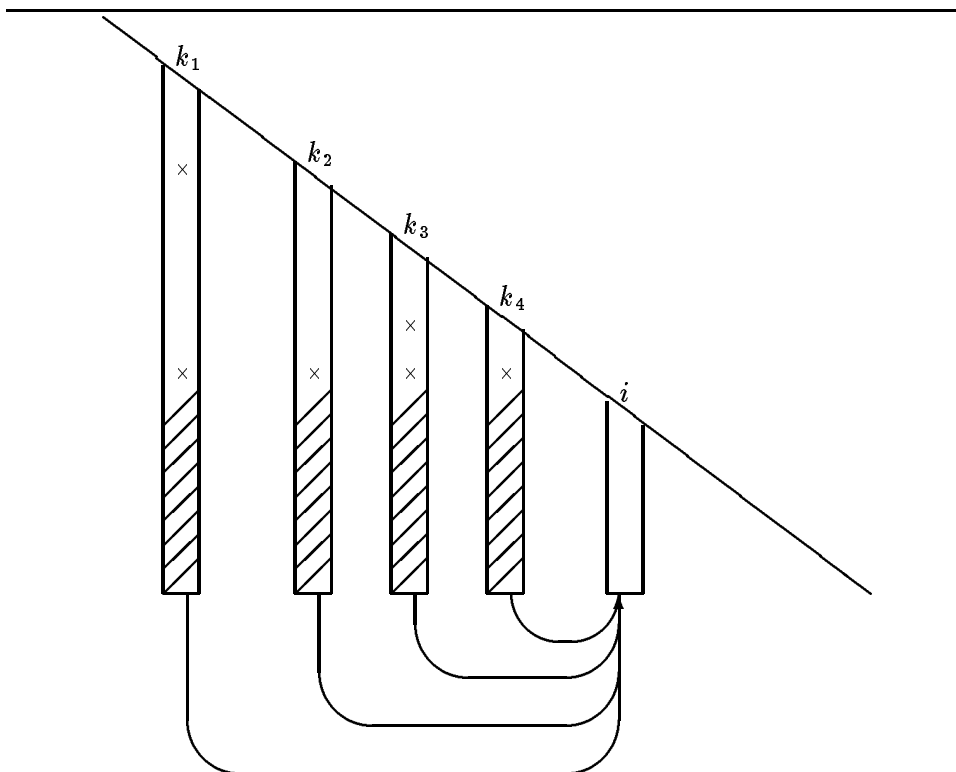


Figure 5.5.5: Merging of reachable sets to obtain  $\text{Reach}(x_i, S_{i-1})$ .

Consider the stage when the set  $S_{i-1} = \{x_1, \dots, x_{i-1}\}$  has been eliminated. For the purpose of this discussion, assume that  $x_i$  has two connected components in  $\mathcal{G}(S_{i-1})$  adjacent to it. Let their node sets be  $C_1$  and  $C_2$ .



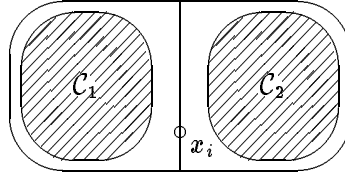


Figure 5.5.6: Determination of the reachable set of  $x_i$ .

In this case, it can be seen that

$$\text{Reach}(x_i, S_{i-1}) = \text{Adj}(x_i) \cup \text{Adj}(C_1) \cup \text{Adj}(C_2) - S_i.$$

However, representatives  $x_{r_1}$  and  $x_{r_2}$  can be chosen from  $C_1$  and  $C_2$  respectively so that

$$\text{Adj}(C_1) = \text{Reach}(x_{r_1}, S_{r_1-1})$$

and

$$\text{Adj}(C_2) = \text{Reach}(x_{r_2}, S_{r_2-1}).$$

(See Exercise 5.3.5 on page 131.) Indeed, the representative is given by (5.3.4); specifically, the node in the component last eliminated. In this way, the reachable set can be written as

$$\text{Reach}(x_i, S_{i-1}) = \text{Adj}(x_i) \cup \text{Reach}(x_{r_1}, S_{r_1-1}) \cup \text{Reach}(x_{r_2}, S_{r_2-1}) - S_i.$$

Thus, instead of having to merge many reachable sets as given in Lemma 5.5.1, we can select representatives. The ideas presented below are motivated by this observation.

For  $k = 1, \dots, n$ , define

$$m_k = \min\{j \mid x_j \in \text{Reach}(x_k, S_{k-1}) \cup \{x_k\}\}. \quad (5.5.1)$$

In terms of the matrix,  $m_k$  is the subscript of the first nonzero in the column vector  $L_{*k}$  excluding the diagonal component.

**Lemma 5.5.2**

$$\text{Reach}(x_k, S_{k-1}) \subset \text{Reach}(x_{m_k}, S_{m_k-1}) \cup \{x_{m_k}\}.$$

**Proof:** For any  $x_i \in Reach(x_k, S_{k-1})$ , then  $k < m_k \leq i$ . If  $i = m_k$ , there is nothing to prove. Otherwise, by Lemma 5.2.1 and Theorem 5.2.2,

$$x_i \in Reach(x_{m_k}, S_{m_k-1}).$$

□

Lemma 5.5.2 has the following important implication. For  $x_i \in Reach(x_k, S_{k-1})$  and  $i > m_k$ , it is redundant to consider  $Reach(x_k, S_{k-1})$  in determining  $Reach(x_i, S_{i-1})$  in the algorithm, since all the reachable nodes via  $x_k$  can be found in  $Reach(x_{m_k}, S_{m_k-1})$ . Thus, it is sufficient to merge the reachable sets of some representative nodes. Figure 5.5.7 shows the improvement on the example in Figure 5.5.5. Lemma 5.5.1 can be improved as follows.

**Theorem 5.5.3**

$$Reach(x_i, S_{i-1}) = Adj(x_i) \cup \left( \bigcup_k \{Reach(x_k, S_{k-1}) \mid m_k = i\} \right) - S_i.$$

**Proof:** Consider any  $x_k$  with  $x_i \in Reach(x_k, S_{k-1})$ . Putting  $m(k) = m_k$ , we have an ascending sequence of subscripts bounded above by  $i$ :

$$k < m(k) < m(m(k)) < m^3(k) < \dots < i.$$

There exists an integer  $p$  such that  $m^{p+1}(k) = i$ . It follows from Lemma 5.5.2 that

$$\begin{aligned} Reach(x_k, S_{k-1}) - S_i &\subset Reach(x_{m(k)}, S_{m(k)-1}) - S_i \\ &\subset \dots \\ &\subset Reach(x_{m^p(k)}, S_{m^p(k)-1}) - S_i. \end{aligned}$$

The result then follows. □

Consider the determination of  $Reach(x_5, S_4)$  in the graph example of Figure 5.5.8. If Lemma 5.5.1 is used, we see that the sets

$$Reach(x_1, S_0),$$

$$Reach(x_2, S_1),$$

and

$$Reach(x_4, S_3)$$

have to be merged with  $Adj(x_5)$ . On the other hand, by Theorem 5.5.3, it is sufficient to consider

$$Reach(x_1, S_0)$$

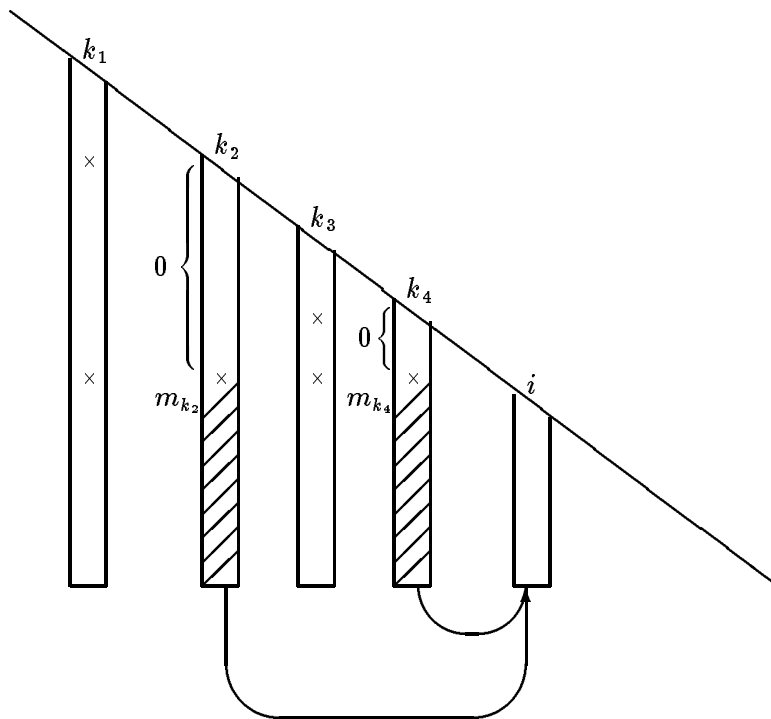
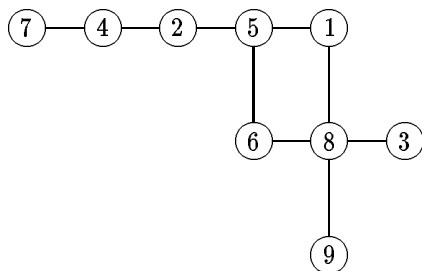


Figure 5.5.7: Improvement in merging reachable sets for  $Reach(x_i, S_{i-1})$ .



$k$	$m_k$	$Reach(x_k, S_{k-1})$
1	5	$x_5, x_8$
2	4	$x_4, x_5$
3	8	$x_8$
4	5	$x_5, x_7$

Figure 5.5.8: Illustration of  $m_k$ .

and

$$\text{Reach}(x_4, S_3).$$

Note that  $m_2 = 4$  and

$$\text{Reach}(x_2, S_1) = \{x_4, x_5\} \subset \text{Reach}(x_4, S_3) \cup \{x_4\}.$$

The symbolic factorization algorithm can now be refined as:

**Step 1** (*Initialization*)

**for**  $k = 1, \dots, n$  **do**  
      $\text{Reach}(x_k, S_{k-1}) = \text{Adj}(x_k) - S_k.$

**Step 2** (*Symbolic Factorization*)

**for**  $k = 1, 2, \dots, n$  **do**  
      $m = \min\{j \mid x_j \in \text{Reach}(x_k, S_{k-1})\}$   
      $\text{Reach}(x_m, S_{m-1}) \leftarrow \text{Reach}(x_m, S_{m-1}) \cup$   
      $(\text{Reach}(x_k, S_{k-1}) - \{x_m\}).$

**Theorem 5.5.4** *Symbolic factorization can be performed in  $O(|E^F|)$  operations.*

**Proof:** For each column  $k$ , the value  $m_k$  is unique. This means that  $\text{Reach}(x_k, S_{k-1})$  is accessed only when  $\text{Reach}(x_{m_k}, S_{m_k-1})$  is being determined. That is, the set  $\text{Reach}(x_k, S_{k-1})$  is examined exactly once throughout the entire process. Moreover, the union of two reachable sets can be performed in time proportional to the sum of their sizes (see Exercise 5.5.3 on page 178). Therefore, symbolic factorization can be done in  $O(|E^F|)$  operations, where  $|E^F| = \sum_{k=1}^n |\text{Reach}(x_k, S_{k-1})|$ .  $\square$

#### 5.5.4 Storage Allocation for the Compressed Scheme and the Subroutine SMBFCT

In this section, we describe the subroutine which performs symbolic factorization as described in the previous section. The result of the process is a data structure for the compressed scheme of Section 5.5.2.

The implementation is due to Eisenstat et. al. [13], and can be found in the Yale Sparse Matrix Package. Essentially, it implements the refined algorithm as described in the previous section, with a rearrangement of the order in which reachable sets are merged together.

**Step 1** (*Initialization*) **for**  $i = 1, \dots, n$  **do**  $R_i = \phi$ .

**Step 2** (*Symbolic Factorization*)

```

for  $k = 1, 2, \dots, n$  do
   $Reach(x_i, S_{i-1}) = Adj(x_i) - S_i$ 
  for  $k \in R_i$  do
     $Reach(x_i, S_{i-1}) \leftarrow Reach(x_i, S_{i-1}) \cup$ 
       $Reach(x_k, S_{k-1}) - S_i$ 
     $m = \min\{j \mid x_j \in Reach(x_i, S_{i-1})\}$ 
     $R_m \leftarrow R_m \cup \{x_i\}$ .

```

In this algorithm, the set  $R_i$  is used to accumulate the representatives whose reachable sets affect that of  $x_i$ . There is an immediate result from Theorem 5.5.3 that can be used to speed up the algorithm. Moreover, it is useful in setting up the compressed storage scheme.

**Corollary 5.5.5** *If there is only one  $m_k = i$ , and*

$$Adj(x_i) - S_i \subset Reach(x_k, S_{k-1})$$

*then*

$$Reach(x_i, S_{i-1}) = Reach(x_k, S_{k-1}) - \{x_i\}.$$

The subroutine **SMBFCT** accepts as input the graph of the matrix stored in the array pair (**XADJ**, **ADJNCY**), together with the permutation vector **PERM** and its inverse **INVP**. The objective of the subroutine is to set up the data structure for the compressed sparse scheme; that is, to compute the compressed subscript vector **NZSUB** and the index vectors **XLNZ** and **XNZSUB**. Also returned are the values **MAXLNZ** and **MAXSUB** which contain the number of off-diagonal nonzeros in the triangular factor and the number of subscripts for the compressed scheme respectively.

Three working vectors **RCHLNK**, **MRGLNK** and **MARKER** are used by the subroutine **SMBFCT**. The vector **RCHLNK** is used to facilitate the merging of the reachable sets, while the vector **MRGLNK** is used to keep track of the non-overlapping representative sets  $\{R_i\}$  as introduced above. The vector **MARKER** is used to detect the condition as given in Corollary 5.5.5.

The subroutine begins by initializing the working vectors **MRGLNK** and **MARKER**. It then executes the main loop finding the reachable set for each node. The set  $Adj(x_k) - S_k$  is first determined and assigned to the vector **RCHLNK**. At

the same time, the condition in Corollary 5.5.5 is tested. If it is satisfied, the merging of reachable sets can be skipped. Otherwise, based on the information in MRGLNK, previous reachable sets are merged into RCHLNK. With the new reachable set completely formed in RCHLNK, the subroutine checks for possible compression of subscripts and sets up the corresponding portion of the data structure accordingly. Finally, it updates the vector MRGLNK to reflect the changes in the sets  $\{R_i\}$ .

By merging a set of carefully selected reachable sets, the subroutine SMBFCT is able to find a new reachable set in a very efficient manner. Since the number of subscripts required in the compressed scheme is not known beforehand, the size of the vector NZSUB may not be large enough to accommodate all the subscripts. In that case, the subroutine will abort and the error flag FLAG will be set to 1.

---

```

1. C*****
2. C*****
3. C*****      SMBFCT . . . . SYMBOLIC FACTORIZATION      *****
4. C*****
5. C*****
6. C
7. C      PURPOSE - THIS ROUTINE PERFORMS SYMBOLIC FACTORIZATION
8. C      ON A PERMUTED LINEAR SYSTEM AND IT ALSO SETS UP THE
9. C      COMPRESSED DATA STRUCTURE FOR THE SYSTEM.
10. C
11. C      INPUT PARAMETERS -
12. C      NEQNS - NUMBER OF EQUATIONS.
13. C      (XADJ, ADJNCY) - THE ADJACENCY STRUCTURE.
14. C      (PERM, INVP) - THE PERMUTATION VECTOR AND ITS INVERSE.
15. C
16. C      UPDATED PARAMETERS -
17. C      MAXSUB - SIZE OF THE SUBSCRIPT ARRAY NZSUB.  ON RETURN,
18. C      IT CONTAINS THE NUMBER OF SUBSCRIPTS USED
19. C
20. C      OUTPUT PARAMETERS -
21. C      XLNZ - INDEX INTO THE NONZERO STORAGE VECTOR LNZ.
22. C      (XNZSUB, NZSUB) - THE COMPRESSED SUBSCRIPT VECTORS.
23. C      MAXLNZ - THE NUMBER OF NONZEROS FOUND.
24. C      FLAG - ERROR FLAG.  POSITIVE VALUE INDICATES THAT.
25. C      NZSUB ARRAY IS TOO SMALL.
26. C
27. C      WORKING PARAMETERS -
28. C      MRGLNK - A VECTOR OF SIZE NEQNS.  AT THE KTH STEP,
29. C      MRGLNK(K), MRGLNK(MRGLNK(K)) , . . . . .
30. C      IS A LIST CONTAINING ALL THOSE COLUMNS L(*,J)
31. C      WITH J LESS THAN K, SUCH THAT ITS FIRST OFF-
```

```

32. C          DIAGONAL NONZERO IS L(K,J).  THUS, THE
33. C          NONZERO STRUCTURE OF COLUMN L(*,K) CAN BE FOUND
34. C          BY MERGING THAT OF SUCH COLUMNS L(*,J) WITH
35. C          THE STRUCTURE OF A(*,K).
36. C          RCHLNK - A VECTOR OF SIZE NEQNS.  IT IS USED TO ACCUMULATE
37. C          THE STRUCTURE OF EACH COLUMN L(*,K).  AT THE
38. C          END OF THE KTH STEP,
39. C          RCHLNK(K), RCHLNK(RCHLNK(K)), .....
40. C          IS THE LIST OF POSITIONS OF NONZEROS IN COLUMN K
41. C          OF THE FACTOR L.
42. C          MARKER - AN INTEGER VECTOR OF LENGTH NEQNS.  IT IS USED
43. C          TO TEST IF MASS SYMBOLIC ELIMINATION CAN BE
44. C          PERFORMED.  THAT IS, IT IS USED TO CHECK WHETHER
45. C          THE STRUCTURE OF THE CURRENT COLUMN K BEING
46. C          PROCESSED IS COMPLETELY DETERMINED BY THE SINGLE
47. C          COLUMN MRGLNK(K).
48. C
49. C*****
50. C
51. C          SUBROUTINE SMBFCT ( NEQNS, XADJ, ADJNCY, PERM, INVP,
52. C          1                    XLNZ, MAXLNZ, XNZSUB, NZSUB, MAXSUB,
53. C          1                    RCHLNK, MRGLNK, MARKER, FLAG )
54. C
55. C*****
56. C
57. C          INTEGER ADJNCY(1), INVP(1), MRGLNK(1), NZSUB(1),
58. C          1                    PERM(1), RCHLNK(1), MARKER(1)
59. C          INTEGER XADJ(1), XLNZ(1), XNZSUB(1),
60. C          1                    FLAG, I, INZ, J, JSTOP, JSTRT, K, KNZ,
61. C          1                    KXSUB, MRGK, LMAX, M, MAXLNZ, MAXSUB,
62. C          1                    NABOR, NEQNS, NODE, NP1, NZBEG, NZEND,
63. C          1                    RCHM, MRKFLG
64. C
65. C*****
66. C
67. C          -----
68. C          INITIALIZATION ...
69. C          -----
70. C          NZBEG = 1
71. C          NZEND = 0
72. C          XLNZ(1) = 1
73. C          DO 100 K = 1, NEQNS
74. C             MRGLNK(K) = 0
75. C             MARKER(K) = 0
76. C          100 CONTINUE
77. C          -----
78. C          FOR EACH COLUMN ..... .  KNZ COUNTS THE NUMBER

```

```

79. C      OF NONZEROS IN COLUMN K ACCUMULATED IN RCHLNK.
80. C      -----
81.      NP1 = NEQNS + 1
82.      DO 1500 K = 1, NEQNS
83.          KNZ = 0
84.          MRGK = MRGLNK(K)
85.          MRKFLG = 0
86.          MARKER(K) = K
87.          IF (MRGK .NE. 0 ) MARKER(K) = MARKER(MRGK)
88.          XNZSUB(K) = NZEND
89.          NODE = PERM(K)
90.          JSTRT = XADJ(NODE)
91.          JSTOP = XADJ(NODE+1) - 1
92.          IF (JSTRT.GT.JSTOP) GO TO 1500
93. C      -----
94. C      USE RCHLNK TO LINK THROUGH THE STRUCTURE OF
95. C      A(*,K) BELOW DIAGONAL
96. C      -----
97.      RCHLNK(K) = NP1
98.      DO 300 J = JSTRT, JSTOP
99.          NABOR = ADJNCY(J)
100.         NABOR = INVP(NABOR)
101.         IF ( NABOR .LE. K ) GO TO 300
102.         RCHM = K
103.         M = RCHM
104.         RCHM = RCHLNK(M)
105.         IF ( RCHM .LE. NABOR ) GO TO 200
106.         KNZ = KNZ+1
107.         RCHLNK(M) = NABOR
108.         RCHLNK(NABOR) = RCHM
109.         IF ( MARKER(NABOR) .NE. MARKER(K) ) MRKFLG = 1
110.     300 CONTINUE
111. C      -----
112. C      TEST FOR MASS SYMBOLIC ELIMINATION ...
113. C      -----
114.      LMAX = 0
115.      IF ( MRKFLG .NE. 0 .OR. MRGK .EQ. 0 ) GO TO 350
116.      IF ( MRGLNK(MRGK) .NE. 0 ) GO TO 350
117.      XNZSUB(K) = XNZSUB(MRGK) + 1
118.      KNZ = XLNZ(MRGK+1) - (XLNZ(MRGK) + 1)
119.      GO TO 1400
120. C      -----
121. C      LINK THROUGH EACH COLUMN I THAT AFFECTS L(*,K).
122. C      -----
123.     350 I = K
124.     400 I = MRGLNK(I)
125.      IF (I.EQ.0) GO TO 800

```



```

126.      INZ = XLNZ(I+1) - (XLNZ(I)+1)
127.      JSTRT = XNZSUB(I) + 1
128.      JSTOP = XNZSUB(I) + INZ
129.      IF (INZ.LE.LMAX) GO TO 500
130.      LMAX = INZ
131.      XNZSUB(K) = JSTRT
132.  C      -----
133.  C      MERGE STRUCTURE OF L(*,I) IN NZSUB INTO RCHLNK.
134.  C      -----
135.      500      RCHM = K
136.      DO 700 J = JSTRT, JSTOP
137.      NABOR = NZSUB(J)
138.      600      M = RCHM
139.      RCHM = RCHLNK(M)
140.      IF (RCHM.LT.NABOR) GO TO 600
141.      IF (RCHM.EQ.NABOR) GO TO 700
142.      KNZ = KNZ+1
143.      RCHLNK(M) = NABOR
144.      RCHLNK(NABOR) = RCHM
145.      RCHM = NABOR
146.      700      CONTINUE
147.      GO TO 400
148.  C      -----
149.  C      CHECK IF SUBSCRIPTS DUPLICATE THOSE OF ANOTHER COLUMN.
150.  C      -----
151.      800      IF (KNZ.EQ.LMAX) GO TO 1400
152.  C      -----
153.  C      OR IF TAIL OF K-1ST COLUMN MATCHES HEAD OF KTH.
154.  C      -----
155.      IF (NZBEG.GT.NZEND) GO TO 1200
156.      I = RCHLNK(K)
157.      DO 900 JSTRT=NZBEG,NZEND
158.      IF (NZSUB(JSTRT)-I) 900, 1000, 1200
159.      900      CONTINUE
160.      GO TO 1200
161.      1000     XNZSUB(K) = JSTRT
162.      DO 1100 J=JSTRT,NZEND
163.      IF (NZSUB(J).NE.I) GO TO 1200
164.      I = RCHLNK(I)
165.      IF (I.GT.NEQNS) GO TO 1400
166.      1100     CONTINUE
167.      NZEND = JSTRT - 1
168.  C      -----
169.  C      COPY THE STRUCTURE OF L(*,K) FROM RCHLNK
170.  C      TO THE DATA STRUCTURE (XNZSUB, NZSUB).
171.  C      -----
172.      1200     NZBEG = NZEND + 1

```

```

173.          NZEND = NZEND + KNZ
174.          IF (NZEND.GT.MAXSUB) GO TO 1600
175.          I = K
176.          DO 1300 J=NZBEG,NZEND
177.             I = RCHLNK(I)
178.             NZSUB(J) = I
179.             MARKER(I) = K
180. 1300      CONTINUE
181.          XNZSUB(K) = NZBEG
182.          MARKER(K) = K
183.  C          -----
184.  C          UPDATE THE VECTOR MRGLNK.  NOTE COLUMN L(*,K) JUST FOUND
185.  C          IS REQUIRED TO DETERMINE COLUMN L(*,J), WHERE
186.  C          L(J,K) IS THE FIRST NONZERO IN L(*,K) BELOW DIAGONAL.
187.  C          -----
188. 1400      IF (KNZ.LE.1) GO TO 1500
189.             KXSUB = XNZSUB(K)
190.             I = NZSUB(KXSUB)
191.             MRGLNK(K) = MRGLNK(I)
192.             MRGLNK(I) = K
193. 1500      XLNZ(K+1) = XLNZ(K) + KNZ
194.             MAXLNZ = XLNZ(NEQNS) - 1
195.             MAXSUB = XNZSUB(NEQNS)
196.             XNZSUB(NEQNS+1) = XNZSUB(NEQNS)
197.             FLAG = 0
198.             RETURN
199.  C          -----
200.  C          ERROR - INSUFFICIENT STORAGE FOR NONZERO SUBSCRIPTS.
201.  C          -----
202. 1600      FLAG = 1
203.             RETURN
204.             END

```

---

## Exercises

5.5.1) Let  $\mathbf{A}$  be a matrix satisfying  $f_i(\mathbf{A}) < i$  for  $2 \leq i \leq n$ . Show that for each  $k < n$ ,  $m_k = k + 1$ . Hence or otherwise, show that for  $1 < i < n$ ,

$$\text{Reach}(x_i, S_{i-1}) = (\text{Adj}(x_i) \cup \text{Reach}(x_{i-1}, S_{i-2})) - S_i.$$

5.5.2) Let  $\mathbf{A}$  be a band matrix with bandwidth  $\beta$ . Assume that the matrix has a full band.

- a) Compare the uncompressed and compressed sparse storage schemes for  $\mathbf{A}$ .

- b) Compare the two symbolic factorization algorithms as given by Lemma 5.5.1 and Theorem 5.5.3.
- 5.5.3) Let  $R_1$  and  $R_2$  be two given sets of integers whose values are less than or equal to  $n$ . Assume that a temporary array of size  $n$  with all zero entries is provided. Show that the union  $R_1 \cup R_2$  can be determined in time proportional to  $|R_1| + |R_2|$ .

## 5.6 The Numerical Subroutines for Factorization and Solution

In this section, we describe the subroutines that perform the numerical factorization and solution for linear systems stored using the compressed sparse scheme. The factorization subroutine **GSFCT** (for general sparse symmetric factorization) uses the inner product form of the factorization algorithm. Since the nonzeros in the lower triangle of  $\mathbf{A}$  (or the factor  $\mathbf{L}$ ) are stored column by column, the inner product version of the algorithm must be implemented to adapt to this storage mode. The implementation **GSFCT** is a minor modification of the one in the Yale Sparse Matrix Package.

### The Subroutine **GSFCT** (General sparse Symmetric FaCTorization)

The subroutine **GSFCT** accepts as input the data structure of the compressed scheme (**XLNZ**, **XNZSUB**, **NZSUB**) and the primary storage vectors **DIAG** and **LNZ**. The vectors **DIAG** and **LNZ**, on input, contain the nonzeros of the matrix  $\mathbf{A}$ . On return, the nonzeros of the factor  $\mathbf{L}$  are overwritten on those of the matrix  $\mathbf{A}$ . The subroutines use three temporary vectors **LINK**, **FIRST** and **TEMP**, all of size  $n$ .

To compute a column  $\mathbf{L}_{*i}$  of the factor, the columns that are involved in the formation of  $\mathbf{L}_{*i}$  are exactly those  $\mathbf{L}_{*j}$  with  $l_{ij} \neq 0$ . The modification can be done one column at a time as follows:

**for**  $\mathbf{L}_{*j}$  with  $l_{ij} \neq 0$  **do**

$$\begin{pmatrix} l_{ii} \\ \vdots \\ l_{ni} \end{pmatrix} \leftarrow \begin{pmatrix} l_{ii} \\ \vdots \\ l_{ni} \end{pmatrix} - l_{ij} \begin{pmatrix} l_{ij} \\ \vdots \\ l_{nj} \end{pmatrix}.$$

At step  $i$ , all the columns that affect  $\mathbf{L}_{*i}$  are given by the list **LINK**( $i$ ), **LINK**(**LINK**( $i$ )),  $\dotsc$ . To minimize subscript searching, a work vector **FIRST** is

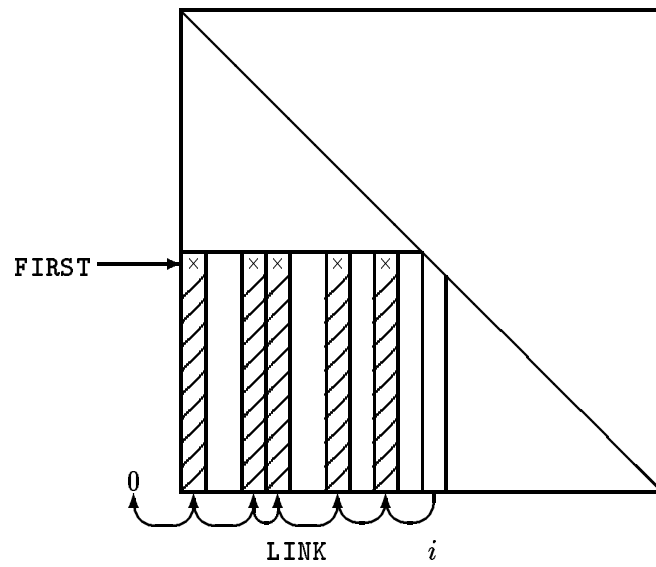


Figure 5.6.1: Illustration of the use of FIRST and LINK in GSFCT.

---

used so that  $\text{FIRST}(j)$  points to the location in the storage vector LNZ, where the nonzero  $l_{ij}$  resides for  $j = \text{LINK}(i), \text{LINK}(\text{LINK}(i)), \dots$ . In this way, the modification of  $L_{*i}$  by  $L_{*j}$  can start at the location  $\text{FIRST}(j)$  in LNZ. The third working vector TEMP is used to accumulate the modifications to the column  $L_{*i}$ .

The subroutine GSFCT begins by initializing the working vectors LINK and TEMP. The loop DO 600 J . . . processes each column. It accumulates the modifications to the current column in the variable DIAGJ and the vector TEMP. At the same time, it updates the temporary vectors FIRST and LINK. Finally, the modification is applied to the entries in the present column.

---

```

1. C*****
2. C*****
3. C*****      GSFCT . . . . GENERAL SPARSE SYMMETRIC FACT      *****
4. C*****
5. C*****
6. C
7. C      PURPOSE - THIS SUBROUTINE PERFORMS THE SYMMETRIC
8. C      FACTORIZATION FOR A GENERAL SPARSE SYSTEM, STORED IN
9. C      THE COMPRESSED SUBSCRIPT DATA FORMAT.
10. C
11. C      INPUT PARAMETERS -
12. C      NEQNS - NUMBER OF EQUATIONS.
13. C      XLNZ - INDEX VECTOR FOR LNZ. XLNZ(I) POINTS TO THE
14. C      START OF NONZEROS IN COLUMN I OF FACTOR L.
15. C      (XNZSUB, NZSUB) - THE COMPRESSED SUBSCRIPT DATA
16. C      STRUCTURE FOR FACTOR L.
17. C
18. C      UPDATED PARAMETERS -
19. C      LNZ - ON INPUT, CONTAINS NONZEROS OF A, AND ON
20. C      RETURN, THE NONZEROS OF L.
21. C      DIAG - THE DIAGONAL OF L OVERWRITES THAT OF A.
22. C      IFLAG - THE ERROR FLAG. IT IS SET TO 1 IF A ZERO OR
23. C      NEGATIVE SQUARE ROOT OCCURS DURING THE
24. C      FACTORIZATION.
25. C      OPS - A DOUBLE PRECISION COMMON PARAMETER THAT IS
26. C      INCREMENTED BY THE NUMBER OF OPERATIONS
27. C      PERFORMED BY THE SUBROUTINE.
28. C
29. C      WORKING PARAMETERS -
30. C      LINK - AT STEP J, THE LIST IN
31. C      LINK(J), LINK(LINK(J)), . . . . .
32. C      CONSISTS OF THOSE COLUMNS THAT WILL MODIFY
33. C      THE COLUMN L(*,J).
34. C      FIRST - TEMPORARY VECTOR TO POINT TO THE FIRST

```

## 5.6. NUMERICAL SUBROUTINES FOR FACTORIZATION AND SOLUTION 181

```

35. C          NONZERO IN EACH COLUMN THAT WILL BE USED
36. C          NEXT FOR MODIFICATION.
37. C          TEMP - A TEMPORARY VECTOR TO ACCUMULATE MODIFICATIONS.
38. C
39. C*****
40. C
41. C          SUBROUTINE GSFCT ( NEQNS, XLNZ, LNZ, XNZSUB, NZSUB, DIAG,
42. C          1             LINK, FIRST, TEMP, IFLAG )
43. C
44. C*****
45. C
46. C          DOUBLE PRECISION COUNT, OPS
47. C          COMMON /SPKOPS/ OPS
48. C          REAL DIAG(1), LNZ(1), TEMP(1), DIAGJ, LJK
49. C          INTEGER LINK(1), NZSUB(1)
50. C          INTEGER FIRST(1), XLNZ(1), XNZSUB(1),
51. C          1             I, IFLAG, II, ISTOP, ISTRT, ISUB, J,
52. C          1             K, KFIRST, NEQNS, NEWK
53. C
54. C*****
55. C
56. C          -----
57. C          INITIALIZE WORKING VECTORS ...
58. C          -----
59. C          DO 100 I = 1, NEQNS
60. C             LINK(I) = 0
61. C             TEMP(I) = 0.0E0
62. C 100    CONTINUE
63. C          -----
64. C          COMPUTE COLUMN L(*,J) FOR J = 1, ..., NEQNS.
65. C          -----
66. C          DO 600 J = 1, NEQNS
67. C          -----
68. C          FOR EACH COLUMN L(*,K) THAT AFFECTS L(*,J).
69. C          -----
70. C          DIAGJ = 0.0E0
71. C          NEWK = LINK(J)
72. C 200    K = NEWK
73. C          IF ( K .EQ. 0 ) GO TO 400
74. C          NEWK = LINK(K)
75. C          -----
76. C          OUTER PRODUCT MODIFICATION OF L(*,J) BY
77. C          L(*,K) STARTING AT FIRST(K) OF L(*,K).
78. C          -----
79. C          KFIRST = FIRST(K)
80. C          LJK = LNZ(KFIRST)
81. C          DIAGJ = DIAGJ + LJK*LJK

```

```

82.          OPS = OPS + 1.0D0
83.          ISTRT = KFIRST + 1
84.          ISTOP = XLNZ(K+1) - 1
85.          IF ( ISTOP .LT. ISTRT ) GO TO 200
86. C -----
87. C     BEFORE MODIFICATION, UPDATE VECTORS FIRST,
88. C     AND LINK FOR FUTURE MODIFICATION STEPS.
89. C -----
90.          FIRST(K) = ISTRT
91.          I = XNZSUB(K) + (KFIRST-XLNZ(K)) + 1
92.          ISUB = NZSUB(I)
93.          LINK(K) = LINK(ISUB)
94.          LINK(ISUB) = K
95. C -----
96. C     THE ACTUAL MOD IS SAVED IN VECTOR TEMP.
97. C -----
98.          DO 300 II = ISTRT, ISTOP
99.             ISUB = NZSUB(I)
100.            TEMP(ISUB) = TEMP(ISUB) + LNZ(II)*LJK
101.            I = I + 1
102. 300      CONTINUE
103.          COUNT = ISTOP - ISTRT + 1
104.          OPS = OPS + COUNT
105.          GO TO 200
106. C -----
107. C     APPLY THE MODIFICATIONS ACCUMULATED IN TEMP TO
108. C     COLUMN L(*,J).
109. C -----
110. 400     DIAGJ = DIAG(J) - DIAGJ
111.         IF ( DIAGJ .LE. 0.0E0 ) GO TO 700
112.         DIAGJ = SQRT(DIAGJ)
113.         DIAG(J) = DIAGJ
114.         ISTRT = XLNZ(J)
115.         ISTOP = XLNZ(J+1) - 1
116.         IF ( ISTOP .LT. ISTRT ) GO TO 600
117.         FIRST(J) = ISTRT
118.         I = XNZSUB(J)
119.         ISUB = NZSUB(I)
120.         LINK(J) = LINK(ISUB)
121.         LINK(ISUB) = J
122.         DO 500 II = ISTRT, ISTOP
123.            ISUB = NZSUB(I)
124.            LNZ(II) = ( LNZ(II)-TEMP(ISUB) ) / DIAGJ
125.            TEMP(ISUB) = 0.0E0
126.            I = I + 1
127. 500     CONTINUE
128.         COUNT = ISTOP - ISTRT + 1

```

## 5.6. NUMERICAL SUBROUTINES FOR FACTORIZATION AND SOLUTION 183

```
129.             OPS = OPS + COUNT
130.    600    CONTINUE
131.             RETURN
132.    C      -----
133.    C      ERROR - ZERO OR NEGATIVE SQUARE ROOT IN FACTORIZATION.
134.    C      -----
135.    700    IFLAG = 1
136.             RETURN
137.             END
```

---

### 5.6.1 The Subroutine GSSLV (General sparse Symmetric SoLve)

The subroutine GSSLV is used to perform the numerical solution of a factored system, where the matrix is stored in the compressed subscript sparse format as discussed in Section 5.5.2. It accepts as input the number of equations NEQNS, together with the data structure and numerical components of the matrix factor. This includes the compressed subscript structure (XNZSUB, NZSUB), the diagonal components DIAG of the factor and the off-diagonal nonzeros in the factor stored in the array pair (XLNZ, LNZ).

Since the nonzeros in the lower triangular factor are stored column by column, the solution method should be arranged so that access to the components is made column-wise. The forward substitution uses the “outer-product” form, whereas the backward substitution loop performs the solution by “inner-products” as discussed in Section 2.3.1 in Chapter 2.

---

```
1.  C*****
2.  C*****
3.  C*****    GSSLV . . . . GENERAL SPARSE SYMMETRIC SOLVE    *****
4.  C*****
5.  C*****
6.  C
7.  C    PURPOSE - TO PERFORM SOLUTION OF A FACTORED SYSTEM, WHERE
8.  C    THE MATRIX IS STORED IN THE COMPRESSED SUBSCRIPT
9.  C    SPARSE FORMAT.
10. C
11. C    INPUT PARAMETERS -
12. C    NEQNS - NUMBER OF EQUATIONS.
13. C    (XLNZ, LNZ) - STRUCTURE OF NONZEROS IN L.
14. C    (XNZSUB, NZSUB) - COMPRESSED SUBSCRIPT STRUCTURE.
15. C    DIAG - DIAGONAL COMPONENTS OF L.
16. C
17. C    UPDATED PARAMETER -
```



```

18. C          RHS - ON INPUT, IT CONTAINS THE RHS VECTOR, AND ON
19. C          OUTPUT, THE SOLUTION VECTOR.
20. C
21. C*****
22. C
23. C          SUBROUTINE GSSLV ( NEQNS, XLNZ, LNZ, XNZSUB, NZSUB,
24. C          1          DIAG, RHS )
25. C
26. C*****
27. C
28. C          DOUBLE PRECISION COUNT, OPS
29. C          COMMON /SPKOPS/ OPS
30. C          REAL DIAG(1), LNZ(1), RHS(1), RHSJ, S
31. C          INTEGER NZSUB(1)
32. C          INTEGER XLNZ(1), XNZSUB(1), I, II, ISTOP,
33. C          1          ISTRT, ISUB, J, JJ, NEQNS
34. C
35. C*****
36. C
37. C          -----
38. C          FORWARD SUBSTITUTION ...
39. C          -----
40. C          DO 200 J = 1, NEQNS
41. C             RHSJ = RHS(J) / DIAG(J)
42. C             RHS(J) = RHSJ
43. C             ISTRT = XLNZ(J)
44. C             ISTOP = XLNZ(J+1) - 1
45. C             IF ( ISTOP .LT. ISTRT ) GO TO 200
46. C             I = XNZSUB(J)
47. C             DO 100 II = ISTRT, ISTOP
48. C                ISUB = NZSUB(I)
49. C                RHS(ISUB) = RHS(ISUB) - LNZ(II)*RHSJ
50. C                I = I + 1
51. C          100          CONTINUE
52. C          200          CONTINUE
53. C          COUNT = 2*(NEQNS + ISTOP)
54. C          OPS = OPS + COUNT
55. C          -----
56. C          BACKWARD SUBSTITUTION ...
57. C          -----
58. C          J = NEQNS
59. C          DO 500 JJ = 1, NEQNS
60. C             S = RHS(J)
61. C             ISTRT = XLNZ(J)
62. C             ISTOP = XLNZ(J+1) - 1
63. C             IF ( ISTOP .LT. ISTRT ) GO TO 400
64. C             I = XNZSUB(J)

```

```
65.          DO 300 II = ISTRT, ISTOP
66.             ISUB = NZSUB(I)
67.             S = S - LNZ(II)*RHS(ISUB)
68.             I = I + 1
69.    300      CONTINUE
70.    400      RHS(J) = S / DIAG(J)
71.             J = J - 1
72.    500      CONTINUE
73.             RETURN
74.            END
```

---

## 5.7 Additional Notes

The *element model* (George [18], Eisenstat [14]) is also used in the study of elimination. It models the factorization process in terms of the clique structure in the elimination graphs. It is motivated by finite element applications, where the clique structure of the matrix graph arises in a natural way. The model is closely related to the quotient graph model studied in Section 5.3. An implementation of the minimum degree algorithm using the element model can be found in George and McIntyre [28]. In [27] the authors have implemented the minimum degree algorithm using the implicit model via reachable sets on the original graph. Refinements have been included to speed up the execution time.

There are other ordering algorithms that are designed to reduce fill-in. The *minimum deficiency algorithm* (Rose [44]) numbers a node next if its elimination incurs the least number of fills. It involves substantially more work than the minimum degree algorithm and experience has shown that in practice the ordering produced is rarely much better than the one produced by the minimum degree algorithm.

In (George [20]), a different storage scheme is proposed for general sparse orderings. It makes use of the observation that off-diagonal nonzeros form dense blocks. Only a few items of information are needed to store each non-null block, and standard dense matrix methods can be used to operate on them.



## Chapter 6

# Quotient Tree Methods for Finite Element and Finite Difference Problems

### 6.1 Introduction

In this and the subsequent two chapters we study methods designed primarily for matrix problems arising in connection with finite difference and finite element methods for solving various problems in structural analysis, fluid flow, elasticity, heat transport and related problems (Zienkiewicz [58]). For our purposes here, the problems we have in mind can be characterized as follows.

Let  $\mathcal{M}$  be a planar mesh consisting of the union of triangles and/or quadrilaterals called *elements*, with adjacent elements sharing a common side or a common vertex. There is a node at each vertex of the mesh  $\mathcal{M}$ , and there may also be nodes lying on element sides and element faces, as shown in the example of Figure 6.1.1. Associated with each node is a variable  $x_i$  and for some labelling of the nodes or variables from 1 to  $n$ , we define a *finite element system*  $\mathbf{A}\mathbf{x} = \mathbf{b}$  associated with  $\mathcal{M}$  as one where  $\mathbf{A}$  is symmetric and positive definite and for which  $a_{ij} \neq 0$  implies variables  $x_i$  and  $x_j$  are associated with nodes of the same element. The graph associated with  $\mathbf{A}$  will be referred to as the finite element graph associated with  $\mathcal{M}$ , as shown in Figure 6.1.1.

In many practical settings, this definition of “finite element system” is not quite general enough, since sometimes more than one variable is associated

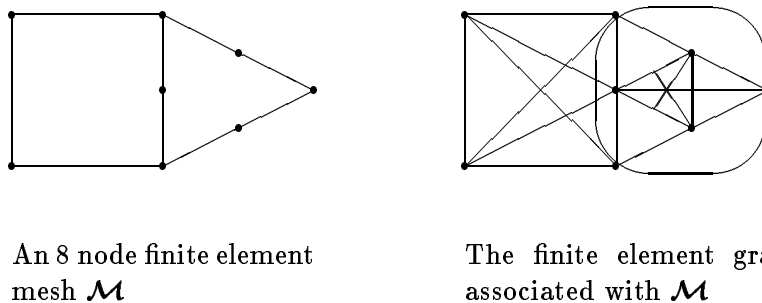


Figure 6.1.1: An 8 node finite element mesh and its associated finite element graph.

---

with some or all of the nodes. However, our definition captures the essential features of such problems and simplifies the presentation of the ideas. Moreover, the extension of the basic ideas to the more general case is immediate, and since our algorithms and programs operate on the associated graph, they work for the general case anyway.

Finite element matrix problems are often solved using the band or profile methods described in Chapter 4, and for relatively small problems these methods are often the most efficient, particularly for one-shot problems where the relatively high cost of finding low fill orderings offsets their lower arithmetic and storage costs. For fairly large problems, and/or in situations where numerous problems having identical structure must be solved, the more sophisticated orderings which attempt to minimize fill, such as the minimum degree ordering of Chapter 5 or the nested dissection orderings of Chapter 8, are attractive.

The methods of Chapters 4 and 5 in a sense represent extremes in the “sophistication spectrum;” the envelope methods do not attempt to exploit much of the structure of  $\mathbf{A}$  and  $\mathbf{L}$ , while the methods of Chapter 5 attempt to exploit it all. In this chapter we investigate methods which lie somewhere in between these two extremes, and for certain sizes and types of finite element problems, they turn out to be more efficient than either of the other two strategies. The ordering times and the operation counts are usually comparable with envelope orderings, but the storage requirements are usually substantially lower.

## 6.2 Solution of Partitioned Systems of Equations

The methods we consider in this chapter rely heavily on the use of partitioned matrices, and some techniques to exploit sparsity in such systems. All the partitionings we consider will be symmetric in the sense that the row and column partitionings will be identical.

### 6.2.1 Factorization of a Block Two by Two Matrix

In order to illustrate most of the important ideas about computations involving sparse partitioned matrices, we consider a block two by two linear system  $\mathbf{A}\mathbf{x} = \mathbf{b}$ :

$$\begin{pmatrix} \mathbf{B} & \mathbf{V} \\ \mathbf{V}^T & \bar{\mathbf{C}} \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{pmatrix}, \quad (6.2.1)$$

where  $\mathbf{B}$  and  $\bar{\mathbf{C}}$  are  $r$  by  $r$  and  $s$  by  $s$  submatrices respectively, with  $r + s = n$ . The Cholesky factor  $\mathbf{L}$  of  $\mathbf{A}$ , correspondingly partitioned, is given by

$$\mathbf{L} = \begin{pmatrix} \mathbf{L}_B & \mathbf{O} \\ \mathbf{W}^T & \mathbf{L}_C \end{pmatrix}, \quad (6.2.2)$$

where  $\mathbf{L}_B$  and  $\mathbf{L}_C$  are the Cholesky factors of the matrices  $\mathbf{B}$  and  $\mathbf{C} = \bar{\mathbf{C}} - \mathbf{V}^T \mathbf{B}^{-1} \mathbf{V}$  respectively, and  $\mathbf{W} = \mathbf{L}_B^{-1} \mathbf{V}$ . Here the “modification matrix” subtracted from  $\bar{\mathbf{C}}$  to obtain  $\mathbf{C}$  can be written as

$$\mathbf{V}^T \mathbf{B}^{-1} \mathbf{V} = \mathbf{V}^T \mathbf{L}_B^{-T} \mathbf{L}_B^{-1} \mathbf{V} = \mathbf{W}^T \mathbf{W}.$$

The determination of the factor  $\mathbf{L}$  can be done as described below. For reasons which will be obvious later in this section we refer to it as the *symmetric* block factorization scheme.

**Step 1** Factor the matrix  $\mathbf{B}$  into  $\mathbf{L}_B \mathbf{L}_B^T$ .

**Step 2** Solve the triangular systems

$$\mathbf{L}_B \mathbf{W} = \mathbf{V}.$$

**Step 3** Modify the submatrix remaining to be factored:

$$\mathbf{C} = \bar{\mathbf{C}} - \mathbf{W}^T \mathbf{W}.$$

**Step 4** Factor the matrix  $\mathbf{C}$  into  $\mathbf{L}_C \mathbf{L}_C^T$ .

This computational sequence is depicted pictorially in Figure 6.2.1. Does this block-oriented computational scheme have any advantage over the ordinary step by step factorization, in terms of operations? The following result is quoted from George [19].

**Theorem 6.2.1** *The number of operations required to compute the factor  $L$  of  $A$  is the same whether the step by step elimination scheme or the symmetric block factorization scheme is used.*

Intuitively, the result holds because the two methods perform exactly the same numerical operations, but in a different order. There is, however, a different way to perform the block factorization, where the arithmetic requirement may decrease or increase. The alternative way depends on the observation that the modification matrix  $V^T B^{-1} V$  can be computed in two distinctly different ways, namely as the conventional product

$$(V^T L_B^{-T})(L_B^{-1} V) = W^T W, \quad (6.2.3)$$

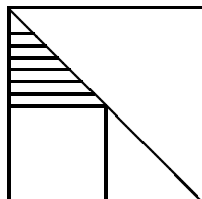
or as

$$V^T (L_B^{-T} (L_B^{-1} V)) = V^T (L_B^{-T} W) = V^T \tilde{W}. \quad (6.2.4)$$

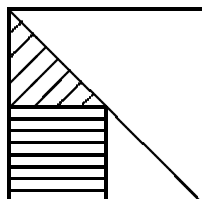
We shall refer to this latter way of performing the computation as the *asymmetric block factorization* scheme.

The difference in the two computations hinges on the cost of computing  $W^T W$  compared to the cost of solving  $L_B^T \tilde{W} = W$ , and then computing  $V^T \tilde{W}$ . As an example illustrating the difference in the arithmetic cost, consider a partitioned matrix  $A$  having the structure indicated in Figure 6.2.2. Since the matrix  $W$  is full (see Exercise 2.3.3 on page 30), by Corollary 2.3.2, the cost of solving the equations  $L_B^T \tilde{W} = W$  is  $4 \times 19 = 76$ . The cost of computing  $V^T \tilde{W}$  is 10, yielding a total of 86. On the other hand, the cost of computing  $W^T W$  is  $10 \times 10 = 100$ .

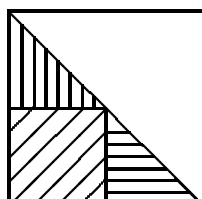
In addition to potentially reducing arithmetic operations, this asymmetric scheme may allow us to substantially reduce storage requirements over that for the standard scheme. The key observation is that we do not need  $W$  in order to solve for  $x$ , provided that  $V$  is available. Whenever we need to compute a product such as  $W^T z$  or  $Wz$ , we can do so by computing  $V^T (L_B^{-T} z)$  or  $L_B^{-1} (Vz)$ ; that is, we solve a triangular system and multiply by a sparse matrix. If  $V$  is much sparser than  $W$ , as it often is, we save storage and perhaps operations as well. The important point to note in terms of computing the factorization is that if we plan to discard  $W$  anyway,



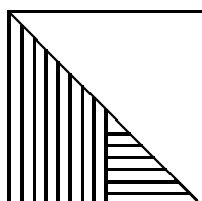
Factorization of  $B$  into  $L_B L_B^T$ .



Solution of the system  $L_B W = V$ .



Computation of  $C = \bar{C} - W^T W$ .



Factorization of  $C$  into  $L_C L_C^T$ .



accessed only



not accessed  
and not modified



accessed and  
modified

Figure 6.2.1: Diagram indicating the sequence of computations for the symmetric block factorization scheme, and the modes in which the data is processed.



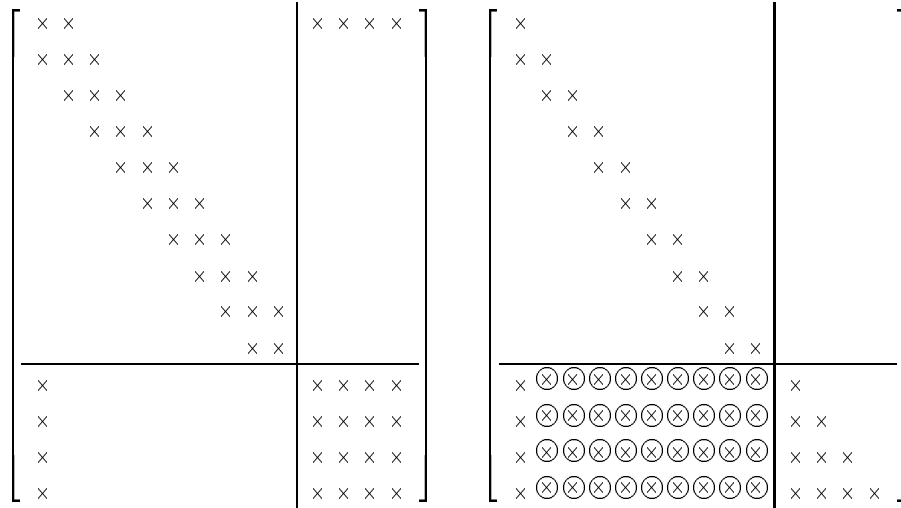


Figure 6.2.2: Structure of a 2 by 2 partitioned matrix  $A$  and its Cholesky factor  $L$ .

computing  $C$  in the asymmetric fashion implied by (6.2.4) allows us to avoid ever storing  $W$ . We can compute the product  $V^T \tilde{W}$  one column at a time, discarding each as soon as it has been used to modify a column of  $\bar{C}$ . Only one temporary vector of length  $r$  is required. By comparison, if we compute  $V^T B^{-1} V$  as  $W^T W$ , there appears to be no way to avoid storing all of  $W$  at some point, even if we do not intend to retain it for later use.

This asymmetric version of the factorization algorithm can be described as follows.

**Step 1** Factor the matrix  $B$  into  $L_B L_B^T$ .

**Step 2** For each column  $v = V_{*i}$  of  $V$ ,

2.1) Solve  $L_B w = v$ .

2.2) Solve  $L_B^T \tilde{w} = w$ .

2.3) Set  $C_{*i} = \bar{C}_{*i} - V^T \tilde{w}$ .

**Step 3** Factor the matrix  $C$  into  $L_C L_C^T$ .

Of course, the symmetry of  $\mathbf{C}$  is exploited in forming  $\mathbf{C}_{*i}$  in Step 2.3. Regardless of which of the above ways we actually employ in calculating the product  $\mathbf{V}^T \mathbf{B}^{-1} \mathbf{V}$ , there is still some freedom in the order in which we calculate the components. Assuming that we compute the lower triangle, we can compute the elements row by row or column by column, as depicted in Figure 6.2.3; each requires a different order of access to the columns of  $\mathbf{W}$  or  $\mathbf{V}$ .

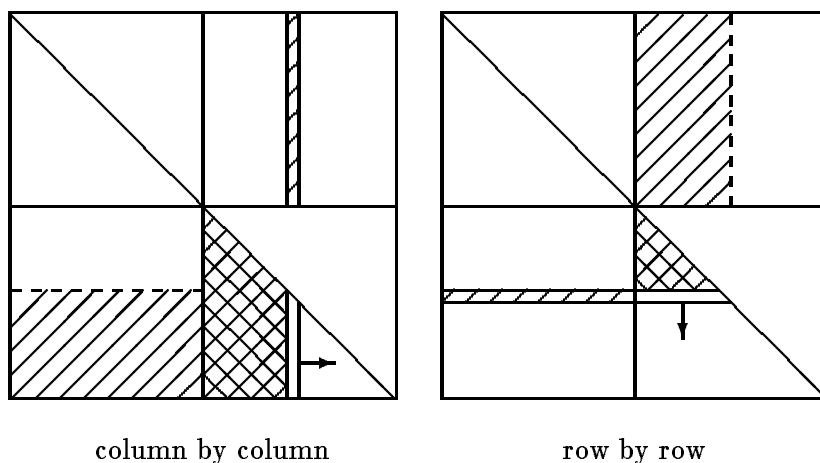


Figure 6.2.3: Diagram showing the access to columns of  $\mathbf{W}$  or  $\mathbf{V}$ , when the lower triangle of  $\mathbf{V}^T \mathbf{B}^{-1} \mathbf{V}$  is computed column by column and row by row.

### 6.2.2 Triangular Solution of a Block Two by Two System

With the Cholesky factor of the partitioned matrix available, the solution of the linear system is straightforward, as shown below.

*Forward Solve*

$$\text{Solve } \mathbf{L}_B \mathbf{y}_1 = \mathbf{b}_1.$$

$$\text{Compute } \tilde{\mathbf{b}}_2 = \mathbf{b}_2 - \mathbf{W}^T \mathbf{y}_1.$$

$$\text{Solve } \mathbf{L}_C \mathbf{y}_2 = \tilde{\mathbf{b}}_2.$$

*Backward Solve*

Solve  $\mathbf{L}_C^T \mathbf{x}_2 = \mathbf{y}_2$ .

Compute  $\tilde{\mathbf{y}}_1 = \mathbf{y}_1 - \mathbf{W} \mathbf{x}_2$ .

Solve  $\mathbf{L}_B^T \mathbf{x}_1 = \tilde{\mathbf{y}}_1$ .

This solution method will be referred to as the *standard solution* scheme. However, as we noted in Section 6.2.1, it may be desirable to discard  $\mathbf{W}$  in favour of storing only  $\mathbf{V}$ , and use the definition  $\mathbf{W} = \mathbf{L}_B^{-1} \mathbf{V}$  whenever we need to operate with the matrix  $\mathbf{W}$ . If we do this, we obtain the following algorithm, which we refer to as the *implicit solution* scheme. Here  $\mathbf{t}_1$  is a temporary vector.

*Forward Solve*

Solve  $\mathbf{L}_B \mathbf{y}_1 = \mathbf{b}_1$ .

Solve  $\mathbf{L}_B^T \mathbf{t}_1 = \mathbf{y}_1$ .

Compute  $\tilde{\mathbf{b}}_2 = \mathbf{b}_2 - \mathbf{V}^T \mathbf{t}_1$ .

Solve  $\mathbf{L}_C \mathbf{y}_2 = \tilde{\mathbf{b}}_2$ .

*Backward Solve*

Solve  $\mathbf{L}_C^T \mathbf{x}_2 = \mathbf{y}_2$ .

Solve  $\mathbf{L}_B \mathbf{t}_1 = \mathbf{V} \mathbf{x}_2$ .

Compute  $\tilde{\mathbf{y}}_1 = \mathbf{y}_1 - \mathbf{t}_1$ .

Solve  $\mathbf{L}_B^T \mathbf{x}_1 = \tilde{\mathbf{y}}_1$ .

In the implicit scheme, only the submatrices

$$\{\mathbf{L}_B, \mathbf{L}_C, \mathbf{V}\}$$

are required, compared to

$$\{\mathbf{L}_B, \mathbf{L}_C, \mathbf{W}\}$$

in the standard scheme. By Corollary 2.3.5,

$$\eta(\mathbf{V}) \leq \eta(\mathbf{W})$$

and for sparse matrices,  $\mathbf{V}$  may have *substantially* fewer nonzeros than  $\mathbf{W}$ . In the matrix example of Figure 6.2.2,

$$\eta(\mathbf{V}) = 4$$

while

$$\eta(\mathbf{W}) = 40.$$

Thus, in terms of primary storage requirements, the use of the implicit solution scheme may be quite attractive.

In terms of operations, the relative merits of the two schemes depend on the sparsity of the matrices  $\mathbf{L}_B$ ,  $\mathbf{V}$  and  $\mathbf{W}$ . Since the cost of computing  $\mathbf{W}\mathbf{z}$  is  $\eta(\mathbf{W})$  and the cost of computing  $\mathbf{L}_B^{-1}(\mathbf{V}\mathbf{z})$  is  $\eta(\mathbf{V}) + \eta(\mathbf{L}_B)$ , we easily obtain the following result. Here, the sparsity of the vector  $\mathbf{z}$  is not exploited.

**Lemma 6.2.2** *The cost of performing implicit solution is no greater than that of doing the standard block solution if and only if  $\eta(\mathbf{V}) + \eta(\mathbf{L}_B) \leq \eta(\mathbf{W})$ .*

In the next section, we shall extend these ideas to block  $p$  by  $p$  linear systems, for  $p > 2$ . In sparse partitioned systems, it is typical that the asymmetric version of block factorization and the implicit form of the solution is superior in terms of computation and storage. Thus, we consider only this version in the remainder of this chapter.

## Exercises

6.2.1) Let  $\mathbf{A}$  be a symmetric positive definite block two by two matrix of the form

$$\mathbf{A} = \begin{pmatrix} \mathbf{B} & \mathbf{V} \\ \mathbf{V}^T & \bar{\mathbf{C}} \end{pmatrix},$$

where both  $\mathbf{B}$  and  $\bar{\mathbf{C}}$  are  $m$  by  $m$  and tridiagonal, and  $\mathbf{V}$  is diagonal. In your answers to the question below, assume  $m$  is large, and ignore low order terms in your operation counts.

a) Denote the triangular factor of  $\mathbf{A}$  by

$$\mathbf{L} = \begin{pmatrix} \mathbf{L}_B & \mathbf{O} \\ \mathbf{W}^T & \mathbf{L}_C \end{pmatrix}.$$

Describe the nonzero structures of  $\mathbf{L}_B$ ,  $\mathbf{L}_C$  and  $\mathbf{W}$ .

b) Determine the number of operations (multiplications and divisions) required to compute  $\mathbf{L}$  using the symmetric and asymmetric factorization algorithms described in Section 6.2.1.

- c) Compare the costs of the explicit and implicit solution schemes of Section 6.2.2 for this problem, where you may assume that the right hand side  $\mathbf{b}$  in the matrix problem  $\mathbf{A}\mathbf{x} = \mathbf{b}$  is full.
- d) Answer a), b) and c) above when  $\mathbf{B}$  and  $\bar{\mathbf{C}}$  are full, and  $\mathbf{V}$  is diagonal.
- e) Answer a), b) and c) above when  $\mathbf{B}$  and  $\bar{\mathbf{C}}$  are full, and  $\mathbf{V}$  is null except for its first row, which is full.
- 6.2.2) The asymmetric factorization scheme can be viewed as computing the factorization shown below.

$$\mathbf{A} = \begin{pmatrix} \mathbf{B} & \mathbf{V} \\ \mathbf{V}^T & \bar{\mathbf{C}} \end{pmatrix} = \begin{pmatrix} \mathbf{L}_B \mathbf{L}_B^T & \mathbf{O} \\ \mathbf{V}^T & \mathbf{L}_C \mathbf{L}_C^T \end{pmatrix} \begin{pmatrix} \mathbf{I} & \bar{\mathbf{W}} \\ \mathbf{O} & \mathbf{I} \end{pmatrix}$$

where  $\bar{\mathbf{W}} = \mathbf{B}^{-1}\mathbf{V}$ , and it is understood that the *factors* of  $\mathbf{B}$  and  $\mathbf{C}$  are stored, rather than  $\mathbf{B}$  and  $\mathbf{C}$ . Write down explicit and implicit solution procedures analogous to those described in Section 6.2.2, using this factorization. Is there any reduction in the operation counts over those of Section 6.2.2? What about storage requirements if we store the off-diagonal blocks of the factors in each case?

- 6.2.3) Prove Theorem 6.2.1.

### 6.3 Quotient Graphs, Trees, and Tree Partitionings

It should be clear that the success of the implicit solution scheme we considered in Section 6.2.2 was due to the very simple form of the off-diagonal block  $\bar{\mathbf{W}}$ . For a general  $p$  by  $p$  partitioned matrix the off-diagonal blocks of its factor will not have such a simple form; to discard them in favor of the original block of  $\mathbf{A}$ , and then to effectively recompute them when needed, would in general be prohibitively costly in terms of computation. This immediately leads us to ask what characteristics a partitioned matrix should have in order that the off-diagonal blocks of its factor have this simple form. In this section we answer this question, and lay the foundations for an algorithm for finding such partitionings.

### 6.3.1 Partitioned Matrices and Quotient Graphs

We have already established the connection between symmetric matrices and graphs. In this section we introduce some additional graph theory ideas to allow us to deal with partitioned matrices.

Let  $\mathbf{A}$  be partitioned into  $p^2$  submatrices  $\mathbf{A}_{ij}$ ,  $1 \leq i, j \leq p$ , and suppose we view each block as a single component which is zero if the block is null, and nonzero otherwise. We can then associate a  $p$ -node graph with the  $p$  by  $p$  block matrix  $\mathbf{A}$ , having edges joining nodes if the corresponding off-diagonal blocks are non-null. Figure 6.3.1 illustrates these ideas. Note that just as in the scalar case, an ordering of this new graph is implied by the matrix to which it corresponds. Also just as before, we are interested in finding partitionings and orderings of *unlabelled* graphs. This motivates the definition of quotient graphs, which we introduced in Chapter 5.

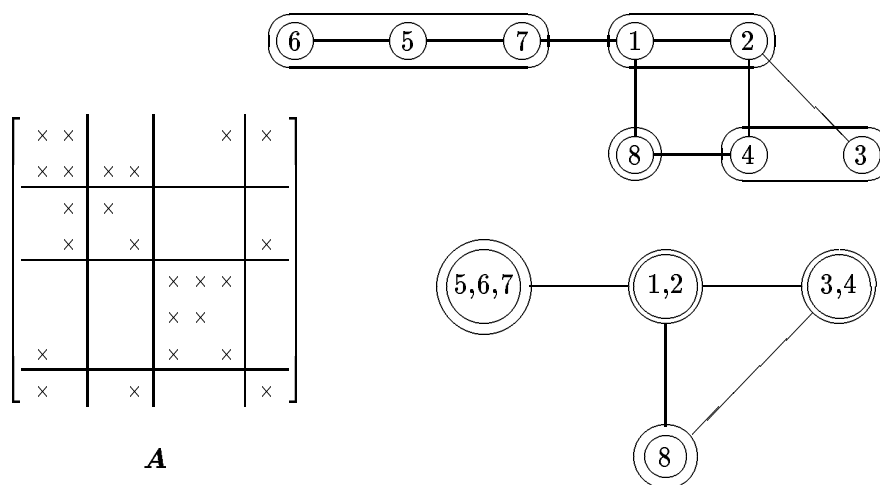


Figure 6.3.1: A partitioned matrix  $\mathbf{A}$ , the implied partitioning of the node set of its graph, and the graph of its zero-nonzero block structure.

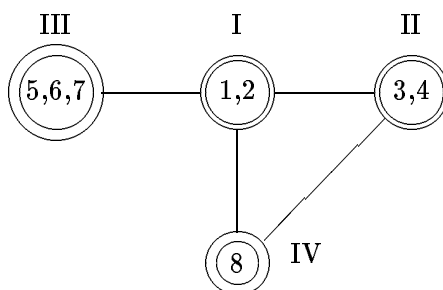
Let  $\mathcal{G} = (X, E)$  be a given unlabelled graph, and let  $\mathcal{P}$  be a partition of its node set  $X$ :

$$\mathcal{P} = \{Y_1, Y_2, \dots, Y_p\}.$$

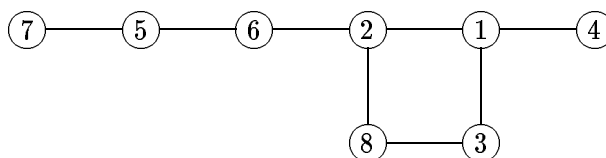
Recall from Chapter 5 that the *quotient graph* of  $\mathcal{G}$  with respect to  $\mathcal{P}$  is the graph  $(\mathcal{P}, \mathcal{E})$ , where  $\{Y_i, Y_j\} \in \mathcal{E}$  if and only if  $Adj(Y_i) \cap Y_j \neq \phi$ . We denote

this graph by  $\mathcal{G}/\mathcal{P}$ .

Note that our definition of quotient graph is for an unlabelled graph. An ordering  $\alpha$  of  $\mathcal{G}$  is said to be *compatible* with a partitioning  $\mathcal{P}$  of  $\mathcal{G}$  if each member  $Y_i$  of  $\mathcal{P}$  is numbered *consecutively* by  $\alpha$ . Clearly orderings and partitionings of graphs corresponding to partitioned matrices *must* have this property. An ordering  $\alpha$  which is compatible with  $\mathcal{P}$  induces or implies an ordering on  $\mathcal{G}/\mathcal{P}$ ; conversely, an ordering  $\alpha_{\mathcal{P}}$  of  $\mathcal{G}/\mathcal{P}$  induces a *class* of orderings on  $\mathcal{G}$  which are compatible with  $\mathcal{P}$ . Figure 6.3.2 illustrates these notions. Unless we explicitly state otherwise, whenever we refer to an ordering of a partitioned graph, we assume that the ordering is compatible with the partitioning, since our interest is in ordering partitioned matrices.



Labelling of  $\mathcal{G}/\mathcal{P}$  induced by the original ordering in Figure 6.3.1



A different ordering compatible with  $\mathcal{P}$

Figure 6.3.2: An example of induced orderings for the graph example in Figure 6.3.1.

In general, when we perform (block) Gaussian elimination on a partitioned matrix, zero blocks may become nonzero. That is, “block fill” may occur

just as fill occurs in the scalar case. For example, in the partitioned matrix of Figure 6.3.1, there are two null off-diagonal blocks which will become non-null after factorization. The structure of the triangular factor is given in Figure 6.3.3.

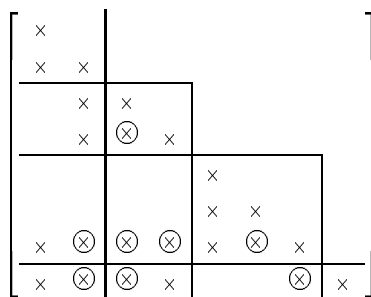


Figure 6.3.3: Structure of the factor of the matrix in Figure 6.3.1.

---

However, for a given partitioning of a symmetric matrix, the way block fill occurs does not necessarily correspond exactly to the scalar situation. In other words, we cannot simply carry out symbolic elimination on the quotient graph, and thereby obtain the block structure of the factor  $L$ . What this provides is the *worst case fill*, which of course could always happen, since each non-null block could be full. Figure 6.3.4 illustrates this point.

Thus, symbolically factoring the quotient graph of a partitioned matrix may yield a higher block fill than actually occurs. Intuitively, the reason is clear: the elimination model assumes that the product of two nonzero quantities will always be nonzero, which is true for scalars. However, it is quite possible to have two non-null matrices whose product is logically zero.

### 6.3.2 Trees, Quotient Trees, and Tree Partitionings

A *tree*  $T = (X, E)$  is a connected graph with no cycles. It is easy to verify that for a tree  $T$ ,  $|X| = |E| + 1$ , and every pair of distinct nodes is connected by exactly one path. A *rooted tree* node of  $T$  called the *root*. Since every pair of nodes in  $T$  is connected by exactly one path, the path from  $r$  to any node  $x \in X$  is unique. If this path passes through  $y$ , then  $x$  is a descendant of  $y$ , and  $y$  is an ancestor of  $x$ . If  $\{x, y\} \in E$ , then  $x$  is a *son* of  $y$  and  $y$  is a *father* of  $x$ . If  $Y$  consists of a node  $y$  and all its descendants, the section



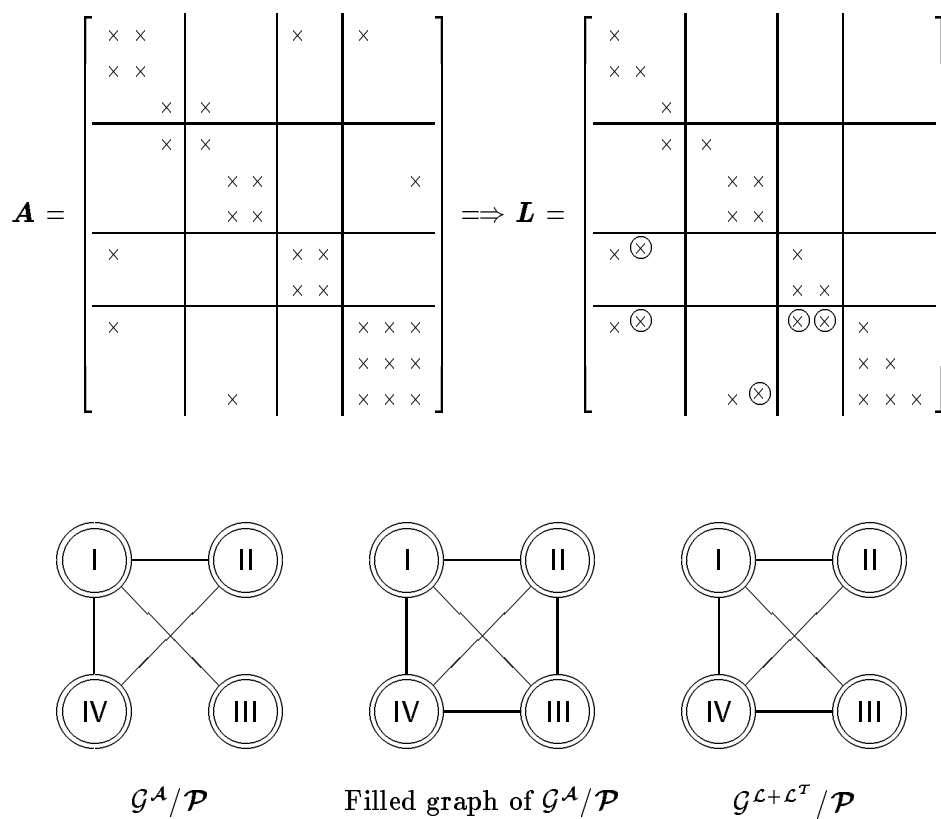


Figure 6.3.4: Example showing that symbolic elimination on  $\mathcal{G}/\mathcal{P}$  may overestimate block fill.

---

graph  $T(Y)$  is a *subtree* of  $T$ . Note that the ancestor-descendant relationship is only defined for rooted trees. These notions are illustrated in Figure 6.3.5.

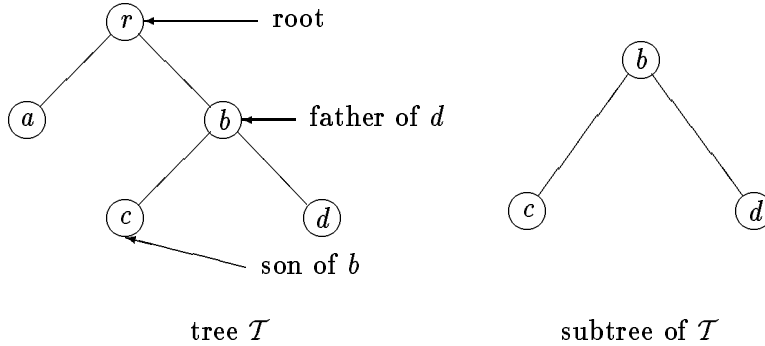


Figure 6.3.5: A rooted tree  $T$  and a subtree.

A *monotone ordering* each node is numbered before its father. Obviously the root must be numbered last. Given an unrooted tree  $T$ , an ordering  $\alpha$  is monotone if it is monotone for the rooted tree  $(\alpha(|X|), T)$ . The significance of monotonely ordered trees is that the corresponding matrices *suffer no fill during their factorization*. The following lemma is due to Parter [43].

**Lemma 6.3.1** *Let  $A$  be an  $n$  by  $n$  symmetric matrix whose labelled graph is a monotonely ordered tree. If  $A = LL^T$ , where  $L$  is the Cholesky factor of  $A$ , then  $a_{ij} = 0$  implies  $l_{ij} = 0, i > j$ .*

The proof is left as an exercise.

**Lemma 6.3.2** *Let  $A$  and  $L$  be as in Lemma 6.3.1. Then  $l_{ij} = a_{ij}/l_{jj}, i > j$ .*

**Proof:** Recall from Section 2.2.2, that the components of  $L$  are given by

$$l_{ij} = \left( a_{ij} - \sum_{k=1}^{j-1} l_{ik}l_{jk} \right) / l_{jj}, \quad i > j.$$

To prove the result we show that  $\sum_{k=1}^{j-1} l_{ik}l_{jk} = 0$ . Suppose for a contradiction that  $l_{im}l_{jm} \neq 0$  for some  $m$  satisfying  $1 \leq m \leq j - 1$ . By Lemma 6.3.1, this means that  $a_{im}a_{jm} \neq 0$ , which implies nodes  $i$  and  $j$  are both connected to

node  $m$  in the corresponding tree, with  $i > m$  and  $j > m$ . But this implies that the tree is not monotonely ordered.  $\square$

Lemmas 6.3.1 and 6.3.2 are not as significant as they might at first seem because matrices which arise in applications seldom have graphs which are trees. Their importance lies in the fact that they extend immediately to *partitioned matrices*.

Suppose  $\mathbf{A}$  is as before, partitioned into  $p^2$  submatrices  $\mathbf{A}_{ij}$ ,  $1 \leq i, j \leq p$ , and let  $\mathbf{L}_{ij}$  be the corresponding submatrices of its Cholesky factor  $\mathbf{L}$ . Suppose further that the labelled quotient graph of the partitioned matrix  $\mathbf{A}$  is a monotonely ordered tree, as illustrated in Figure 6.3.6. Then it is straightforward to verify the analog of Lemma 6.3.2 for such a partitioned matrix, that is  $\mathbf{L}_{ij} = \mathbf{A}_{ij}\mathbf{L}_{jj}^{-T} = (\mathbf{L}_{jj}^{-1}\mathbf{A}_{ji})^T$  for each non-null submatrix  $\mathbf{A}_{ij}$  in the lower triangle of  $\mathbf{A}$ . When a partitioning  $\mathcal{P}$  of a graph  $\mathcal{G}$  is such that  $\mathcal{G}/\mathcal{P}$  is a tree, we call  $\mathcal{P}$  a *tree partitioning* of  $\mathcal{G}$ .

We have now achieved what we set out to do, namely, to determine the characteristics a partitioned matrix must have in order to apply the ideas developed in Section 6.2. The answer is that we want its labelled quotient graph to be a monotonely ordered tree. If it has this property, we can reasonably discard all the off-diagonal blocks of  $\mathbf{L}$ , saving only its diagonal blocks and the off-diagonal blocks of the lower triangle of  $\mathbf{A}$ .

### 6.3.3 Asymmetric Block Factorization and Implicit Block Solution of Tree-partitioned Systems

Let  $\mathbf{A}$  be  $p$  by  $p$  partitioned with blocks  $\mathbf{A}_{ij}$ ,  $1 \leq i, j \leq p$ , and  $\mathbf{L}_{ij}$  be the corresponding blocks of  $\mathbf{L}$  for  $i > j$ . If the quotient graph of  $\mathbf{A}$  is a monotonely ordered tree, there is exactly one non-null block below the diagonal block in  $\mathbf{A}$  and  $\mathbf{L}$  (Why?); let this block be  $\mathbf{A}_{\mu_k, k}$   $1 \leq k \leq p - 1$ . The *asymmetric block factorization* algorithm for such problems is as follows.

**Step 1** For  $k = 1, 2, \dots, p - 1$  do the following

1.1) Factor  $\mathbf{A}_{kk}$ .

1.2) For each column  $\mathbf{u}$  of  $\mathbf{A}_{k, \mu_k}$ , solve  $\mathbf{A}_{kk}\mathbf{v} = \mathbf{u}$ , compute  $\mathbf{w} = \mathbf{A}_{k, \mu_k}^T \mathbf{v}$  and subtract it from the appropriate column of  $\mathbf{A}_{\mu_k, \mu_k}$ .

**Step 2** Factor  $\mathbf{A}_{pp}$ .

A pictorial illustration of the modification of  $\mathbf{A}_{\mu_k, \mu_k}$  is shown in Figure 6.3.7. Note that in the algorithm, temporary storage for the vectors  $\mathbf{u}$  and  $\mathbf{w}$  of

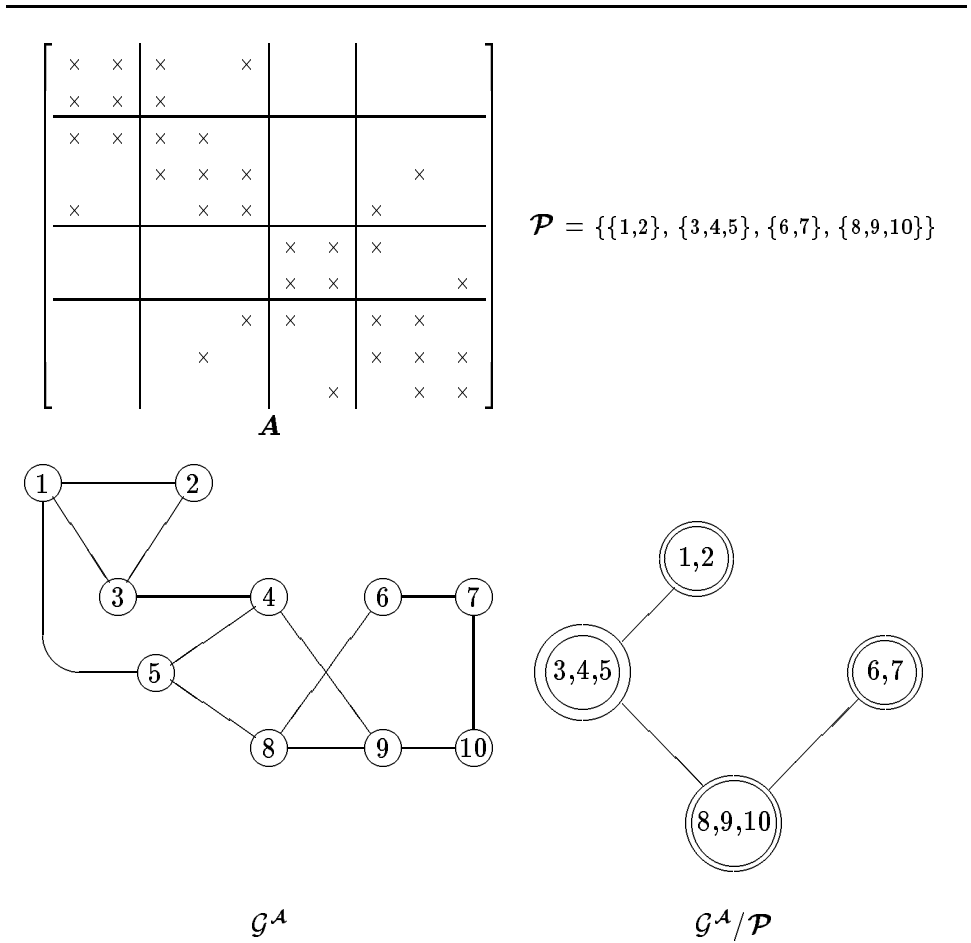


Figure 6.3.6: Illustration of a partitioned matrix, its graph, and its quotient graph which is a tree.

---

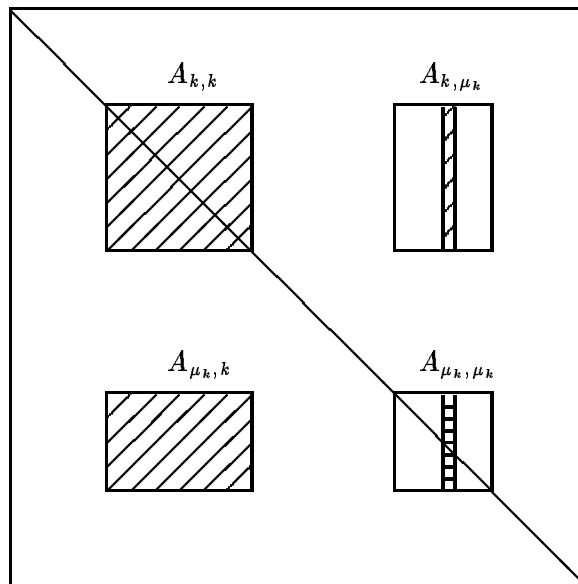


Figure 6.3.7: Pictorial illustration of asymmetric block factorization.

length equal to the largest block size is all that is needed. (The vector  $\mathbf{v}$  can overwrite  $\mathbf{u}$ .) Of course, the symmetry of the diagonal blocks can be exploited.

The implicit solution scheme for such block systems is also straightforward. Here  $\mathbf{t}$  and  $\tilde{\mathbf{t}}$  are temporary vectors which can share the same space.

*Forward implicit block solve: ( $\mathbf{L}\mathbf{y} = \mathbf{b}$ )*

**Step 1** For  $k = 1, \dots, p-1$ , do the following:

- 1.1) Solve  $\mathbf{L}_{kk}\mathbf{y}_k = \mathbf{b}_k$ .
- 1.2) Solve  $\mathbf{L}_{kk}^T\mathbf{t} = \mathbf{y}_k$ .
- 1.3) Compute  $\mathbf{b}_{\mu_k} \leftarrow \mathbf{b}_{\mu_k} - \mathbf{A}_{k,\mu_k}^T\mathbf{t}$ .

**Step 2** Solve  $\mathbf{L}_{pp}\mathbf{y}_p = \mathbf{b}_p$ .

*Backward implicit block solve: ( $\mathbf{L}^T\mathbf{x} = \mathbf{y}$ )*

**Step 1** Solve  $\mathbf{L}_{pp}^T\mathbf{x}_p = \mathbf{y}_p$ .

**Step 2** For  $k = p-1, p-2, \dots, 1$  do the following:

- 2.1) Compute  $\mathbf{t} = \mathbf{A}_{k,\mu_k}\mathbf{x}_{\mu_k}$ .
- 2.2) Solve  $\mathbf{L}_{kk}\tilde{\mathbf{t}} = \mathbf{t}$ .
- 2.3) Replace  $\mathbf{y}_k \leftarrow \mathbf{y}_k - \tilde{\mathbf{t}}$ .
- 2.4) Solve  $\mathbf{L}_{kk}^T\mathbf{x}_k = \mathbf{y}_k$ .

Figure 6.3.8 gives the steps on the forward block solve of a block four by four system.

## Exercises

6.3.1) Prove Lemma 6.3.1.

6.3.2) Let  $\mathcal{G}^{\mathbf{F}}/\mathcal{P} = (\mathcal{P}, \mathcal{E}^{\mathbf{F}})$  be the quotient graph of the filled graph of  $\mathcal{G}$  with respect to a partitioning  $\mathcal{P}$ , and let  $(\mathcal{G}/\mathcal{P})^{\mathbf{F}} = (\mathcal{P}, \tilde{\mathcal{E}}^{\mathbf{F}})$  be the filled graph of  $\mathcal{G}/\mathcal{P}$ . The example in Figure 6.3.4 shows that  $\mathcal{E}^{\mathbf{F}}$  may have fewer members than  $\tilde{\mathcal{E}}^{\mathbf{F}}$ . That is, the block structure of the factor  $\mathbf{L}$  of a partitioned matrix  $\mathbf{A}$  may be sparser than the filled graph of the quotient graph  $\mathcal{G}/\mathcal{P}$  would suggest. Show that if the diagonal blocks of  $\mathbf{L}$  have the propagation property (see Exercise 2.3.3 on page 30), then the filled graph of  $\mathcal{G}/\mathcal{P}$  will correctly reflect the block structure of  $\mathbf{L}$ . That is, show that  $\mathcal{E}^{\mathbf{F}} = \tilde{\mathcal{E}}^{\mathbf{F}}$  in this case.

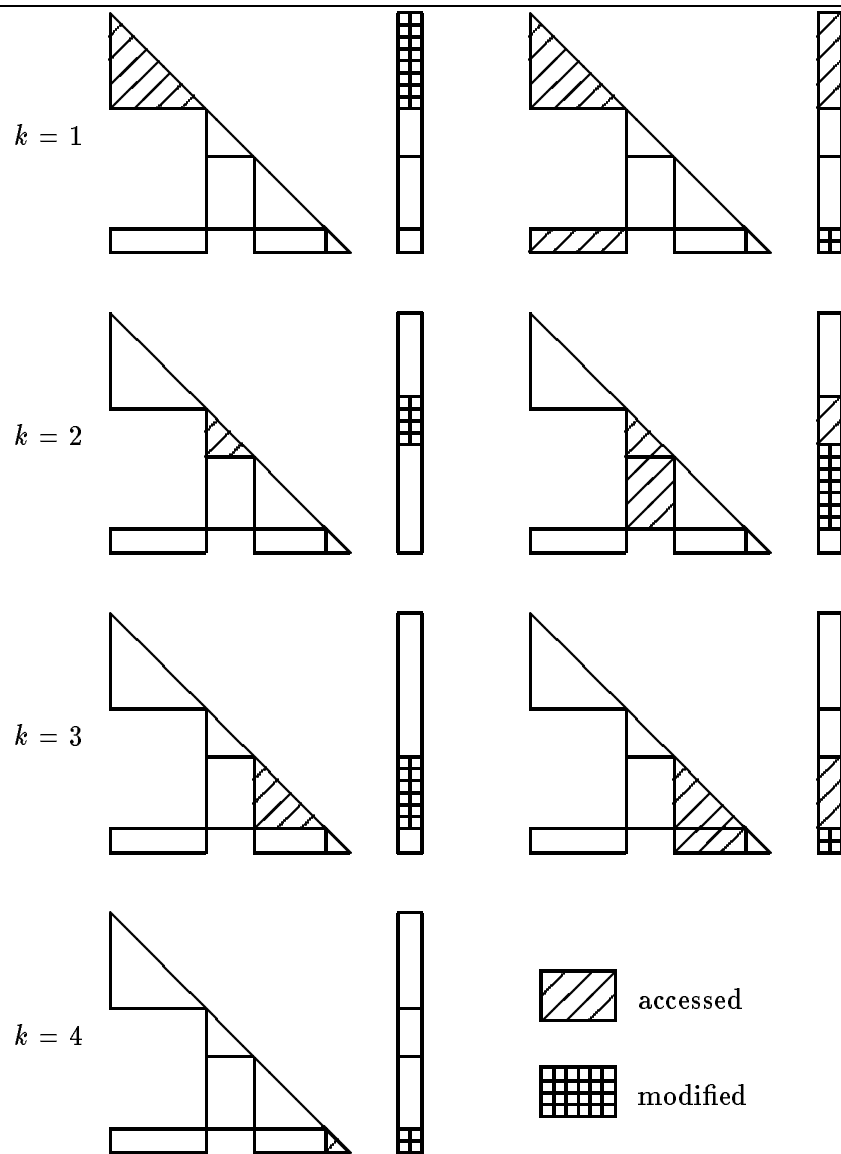


Figure 6.3.8: Forward implicit block solve for a block 4 by 4 system.

---

6.3. QUOTIENT GRAPHS, TREES, AND TREE PARTITIONINGS 207

6.3.3) Let  $\mathcal{E}^{\mathbf{F}}$  and  $\tilde{\mathcal{E}}^{\mathbf{F}}$  be as defined in Exercise 6.3.2 on page 205, for the labelled graph  $\mathcal{G}$ , and partitioning  $\mathcal{P} = \{Y_1, Y_2, \dots, Y_p\}$ .

a) Prove that if the subgraphs  $\mathcal{G}(Y_i)$ ,  $i = 1, 2, \dots, p$  are connected, then  $\mathcal{E}^{\mathbf{F}} = \tilde{\mathcal{E}}^{\mathbf{F}}$ .

b) Give an example to show that the converse need not hold.

c) Prove that if the subgraphs  $\mathcal{G}(\cup_{i=1}^l Y_i)$ ,  $l = 1, 2, \dots, p-1$  are connected, then  $\mathcal{E}^{\mathbf{F}} = \tilde{\mathcal{E}}^{\mathbf{F}}$ .

6.3.4) A tree partitioning  $\mathcal{P} = \{Y_1, Y_2, \dots, Y_p\}$  of a graph  $\mathcal{G} = (X, E)$  is *maximal* if there does not exist a tree partitioning  $\mathcal{Q} = \{Z_1, Z_2, \dots, Z_t\}$  such that  $p < t$ , and for each  $i$ ,  $Z_i \subset Y_k$  for some  $1 \leq k \leq p$ . In other words, it is maximal if we cannot subdivide one or more of the  $Y_i$ 's and still maintain a quotient tree. Suppose that for every pair of distinct nodes in any  $Y_i$ , there exist two distinct paths between  $x$  and  $y$ :

$$x, x_1, x_2, \dots, x_s, y$$

and

$$x, y_1, y_2, \dots, y_t, y$$

such that

$$S = \bigcup \{Z \in \mathcal{P} \mid x_i \in Z, 1 \leq i \leq s\}$$

$$T = \bigcup \{Z \in \mathcal{P} \mid y_i \in Z, 1 \leq i \leq t\}$$

are disjoint. Show that  $\mathcal{P}$  is maximal.

6.3.5) Let  $\mathbf{A}$  be a *block tridiagonal* matrix,

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{V}_2 & & & & \\ \mathbf{V}_2^T & \mathbf{A}_{22} & \mathbf{V}_3 & & & \\ & \mathbf{V}_3^T & \mathbf{A}_{33} & \ddots & & \\ & & & \ddots & \mathbf{V}_p & \\ & & & & \mathbf{V}_p^T & \mathbf{A}_{pp} \end{pmatrix}$$

where each  $\mathbf{A}_{ii}$  is an  $m$  by  $m$  square full matrix and each  $\mathbf{V}_i$  is a diagonal matrix.



- a) What is the arithmetic cost for performing the asymmetric block factorization? How does it compare with that of the symmetric version?
- b) What if each submatrix  $V_i$  has this sparse form

$$\begin{pmatrix} \times & \times & \cdots & \times & \times \\ & & \mathbf{0} & & \end{pmatrix}?$$

Assume  $m$  is large, and ignore low order terms in your calculations.

## 6.4 A Quotient Tree Partitioning Algorithm

### 6.4.1 A Heuristic Algorithm

The results of Section 6.3 suggest that we would like to find a partitioning  $\mathcal{P} = \{Y_1, Y_2, \dots, Y_p\}$  with as many members as possible, consistent with the requirement that  $\mathcal{G}/\mathcal{P}$  be a tree. In this section we provide an algorithm for finding a tree partitioning of a graph.

The algorithm we propose is closely related to level structures (see Section 4.4), so we begin by observing the connection between a level structure on a graph and the partitioning it induces on the corresponding matrix. Let  $\mathcal{G}^{\mathbf{A}}$  be the unlabelled graph associated with  $\mathbf{A}$ , and let  $\mathcal{L} = \{L_0, L_1, \dots, L_l\}$  be a level structure in  $\mathcal{G}^{\mathbf{A}}$ . From the definition of level structure, it is clear that the quotient graph  $\mathcal{G}/\mathcal{L}$  is a chain, so if we number the nodes in each level  $L_i$  consecutively, from  $L_0$  to  $L_l$ , the levels of  $\mathcal{L}$  induce a block tridiagonal structure on the correspondingly ordered matrix. An example appears in Figure 6.4.1.

The algorithm we will ultimately present in this section begins with a rooted level structure and then attempts to make the partitioning finer by refining the levels of the level structure. Let  $\mathcal{L} = \{L_0, L_1, \dots, L_l\}$  be a rooted level structure and let  $\mathcal{P} = \{Y_1, Y_2, \dots, Y_p\}$  be the partitioning obtained by subdividing each  $L_j$  as follows. Letting  $\mathcal{B}_j$  be the section graph prescribed by

$$\mathcal{B}_j = \mathcal{G} \left( \bigcup_{i=j}^l L_i \right), \quad (6.4.1)$$

each  $L_j$  is partitioned according to the sets specified by

$$\{Y \mid Y = L_j \cap C, \mathcal{G}(C) \text{ is a connected component of } \mathcal{B}_j\}. \quad (6.4.2)$$

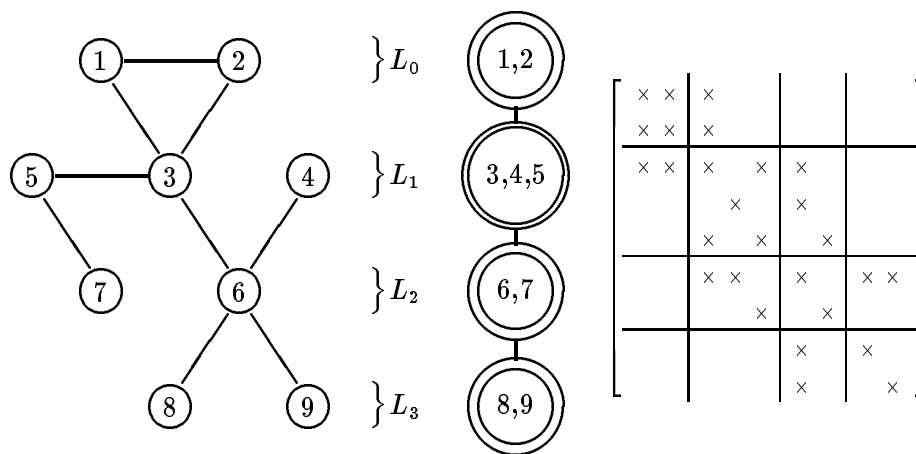


Figure 6.4.1: Block tridiagonal partitioning induced by a level structure.

Figure 6.4.2 illustrates this refinement for a simple graph.

Consider the refinement on  $L_2 = \{d, e\}$ . Note that

$$B_2 = \mathcal{G}(\{d, e, i, g, f, h\})$$

and it has two connected components with node sets:

$$\{d, i, g\}$$

and

$$\{e, f, h\}.$$

Therefore, the level  $L_2$  can be partitioned according to (6.4.2) into  $\{d\}$  and  $\{e\}$ .

We are now ready to describe the algorithm for finding a tree partitioning. Our description makes use of the definition  $SPAN(Y)$ , which is defined for a subset  $Y$  of  $X$  by

$$SPAN(Y) = \{x \in X \mid \text{there exists a path from } y \text{ to } x, \text{ for some } y \in Y\}. \tag{6.4.3}$$

When  $Y$  is a single node  $y$ ,  $SPAN(Y)$  is simply the connected component containing  $y$ .

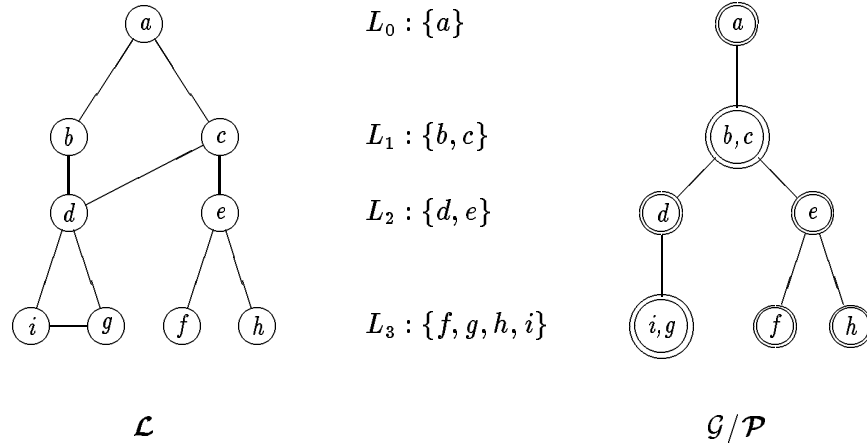


Figure 6.4.2: An example showing the refined partitioning  $\mathcal{P}$  obtained from the level structure  $\mathcal{L}$ .

---

The algorithm we now describe makes use of a stack, and thereby avoids explicitly finding the connected components of the  $B_j$  which appeared in the description of the level refinement (6.4.2). We assume that we are given a root  $r$  for the rooted level structure to be refined. We discuss the choice of  $r$  later in this section.

**Step 0** (Initialization): Empty the stack. Generate the level structure  $\mathcal{L}(r) = \{L_0, L_1, L_2, \dots, L_{l(r)}\}$  rooted at  $r$ , and choose any node  $y \in L_{l(r)}$ . Set  $l = l(r)$  and  $S = \{y\}$ .

**Step 1** (Pop stack): If the node set  $T$  on the top of the stack belongs to  $L_l$ , pop  $T$  and set  $S \leftarrow S \cup T$ .

**Step 2** (Form possible partition member): Determine the set  $Y = \text{SPAN}(S)$  in the subgraph  $\mathcal{G}(L_l)$ . If  $l < l(r)$  and some node in  $\text{Adj}(Y) \cap L_{l+1}$  has not yet been placed in a partition member, go to Step 5.

**Step 3** (New partition member): Put  $Y$  in  $\mathcal{P}$ .

**Step 4** (Next level): Determine the set  $S = \text{Adj}(Y) \cap L_{l-1}$ , and set  $l \leftarrow l-1$ . If  $l \geq 0$ , go to Step 1, otherwise stop.

**Step 5** (Partially formed partition member): Push  $S$  onto the stack. Pick  $y_{l+1} \in \text{Adj}(Y) \cap L_{l+1}$  and trace a path  $y_{l+1}, y_{l+2}, \dots, y_{l+t}$ , where

$y_{l+i} \in L_{l+i}$  and  $Adj(y_{l+t}) \cap L_{l+t+1} = \phi$ . Set  $S = \{y_{l+t}\}$  and  $l \leftarrow l + t$ , and then go to Step 2.

The example in Figure 6.4.4, taken from George and Liu [23] illustrates how the algorithm operates. The level structure rooted at node 1 is refined to obtain a quotient tree having 10 nodes. In the example,  $Y_1 = \{20\}$ ,  $Y_2 = \{18, 19\}$ ,  $Y_3 = \{16\}$ ,  $Y_4 = \{10, 15\}$ ,  $Y_5 = \{9, 14, 17\}$ , and  $Y_6 = \{5, 11\}$ , with  $L_4 = Y_5 \cup Y_6$ ,  $L_5 = Y_2 \cup Y_4$ , and  $L_6 = Y_1 \cup Y_3$ .

In order to complete the description of the tree partitioning algorithm we must specify how to find the root node  $r$  for the level structure. We obtain  $r$  and  $\mathcal{L}(r)$  by using the subroutine `FNRROOT`, described in Section 4.4.3. Since we want a partitioning with as many members as possible, this seems to be a sensible choice since it will tend to provide a level structure having relatively many levels.

#### 6.4.2 Subroutines for Finding a Quotient Tree Partitioning

In this section, a set of subroutines which implements the quotient tree algorithm is discussed. The parameters `NEQNS`, `XADJ` and `ADJNCY`, as before, are used to store the adjacency structure of the given graph. The vector `PERM` returns the computed quotient tree ordering. In addition to the ordering, the partitioning information is returned in the variable `NBLKS` and the vector `XBLK`. The number of partition blocks is given in `NBLKS`, while the node numbers of a particular block, say block  $k$ , are given by

$$\{\text{PERM}(j) \mid \text{XBLK}(k) \leq j < \text{XBLK}(k + 1)\}$$

Figure 6.4.6 contains the representation of the quotient tree ordering for the example in Figure 6.4.3.

As we see from the example, the vector `XBLK` has size `NBLKS + 1`. The last extra pointer is included so that blocks can be retrieved in a uniform manner. In the example, to obtain the fifth block, we note that

$$\begin{aligned} \text{XBLK}(5) &= 7, \\ \text{XBLK}(6) &= 10. \end{aligned}$$

Thus, the nodes in this block are given by `PERM(7)`, `PERM(8)` and `PERM(9)`. There are seven subroutines in this set, two of which have been considered in detail in Chapter 4. We first consider their control relationship as shown in Figure 6.4.7.

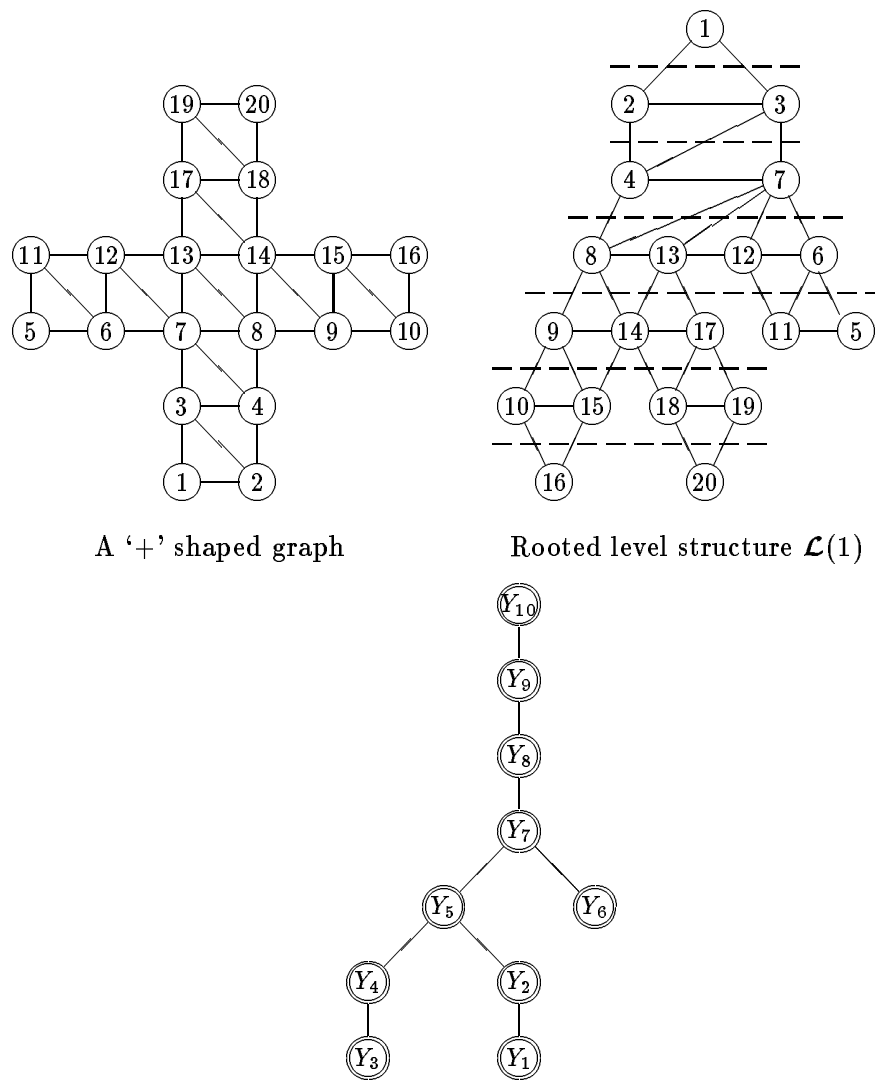


Figure 6.4.3: A graph, rooted level structure, and refined quotient tree.

---

$k$	Partitioning	$Y_k$	Level	Adjacent Set	Stack
1	$Y_1$	{20}	6	{18,19}	$\emptyset$
2	$Y_2$   $Y_1$	{18,19}	5	{14,17}	$\emptyset$
3	$Y_3$ $Y_2$   $Y_1$	{16}	6	{10,15}	{14,17}
4	$Y_4$ $Y_2$   $Y_3$   $Y_1$	{10,15}	5	{9,14}	{14,17}
5	$Y_5$ / $Y_4$ \ $Y_2$   $Y_3$   $Y_1$	{9,14,17}	4	{8,13}	$\emptyset$
6	$Y_5$ $Y_6$ / $Y_4$ \ $Y_2$   $Y_3$   $Y_1$	{5,11}	4	{6,12}	{8,13}
7	$Y_7$ / $Y_5$ \ $Y_6$ / $Y_4$ \ $Y_2$   $Y_3$   $Y_1$	{8,13,12,6}	3	{4,7}	$\emptyset$

Figure 6.4.4: Numbering in the refined quotient tree algorithm.

$k$	Partitioning	$Y_k$	Level	Adjacent Set	Stack
8		$\{4,7\}$	2	$\{2,3\}$	$\emptyset$
9		$\{2,3\}$	1	$\{1\}$	$\emptyset$
10		$\{1\}$	0	$\emptyset$	$\emptyset$

Figure 6.4.5: Continuation of Figure 6.4.4.

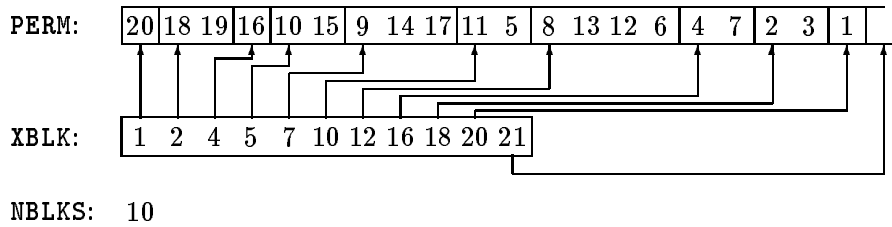


Figure 6.4.6: An example of the data structure for a partitioning.

---

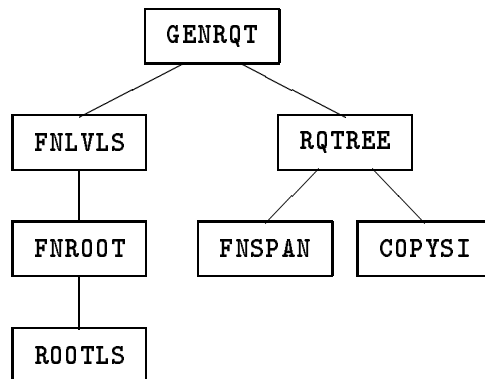


Figure 6.4.7: Control relation of subroutines for the refined quotient tree algorithm.

---



The subroutines `FNRROOT` and `ROOTLS` are used to determine a pseudo-peripheral node of a connected component of a given graph. For details of these two subroutines, readers are referred back to Section 4.4.3. The subroutine `COPYSI` is a simple utility program that copies an integer array into another one. (A listing of the subroutine appears after that of `RQTREE`.) We now describe in detail the remaining subroutines in this group.

### GENRQT (GENeral Refined Quotient Tree)

This subroutine is the driver subroutine for finding the quotient tree ordering of a general disconnected graph. It goes through the graph and calls the subroutine `RQTREE` to number each connected component in the graph. It requires three working arrays `XLS`, `LS` and `NODLVL`. The array pair (`XLS`, `LS`) is used by `FNLVLS` to obtain a level structure rooted at a pseudo-peripheral node, while the vector `NODLVL` is used to store the level number of nodes in the level structure.

The subroutine begins by initializing the vector `NODLVL` and the variable `NBLKS`. It then goes through the graph until it finds a node  $i$  not yet numbered. Note that numbered nodes have their `NODLVL` values set to zero. This node  $i$  together with the array `NODLVL` defines a connected subgraph of the original graph. The subroutines `FNLVLS` and `RQTREE` are then called to order the nodes of this subgraph. The subroutine returns after it has processed all the components of the graph.

---

```

1. C*****
2. C*****
3. C*****      GENRQT . . . . GENERAL REFINED QUOTIENT TREE      *****
4. C*****
5. C*****
6. C
7. C      PURPOSE - THIS ROUTINE IS A DRIVER FOR DETERMINING A
8. C      PARTITIONED ORDERING FOR A POSSIBLY DISCONNECTED
9. C      GRAPH USING THE REFINED QUOTIENT TREE ALGORITHM.
10. C
11. C      INPUT PARAMETERS -
12. C      NEQNS - NUMBER OF VARIABLES.
13. C      (XADJ, ADJNCY) - THE ADJACENCY STRUCTURE.
14. C
15. C      OUTPUT PARAMETERS -
16. C      (NBLKS, XBLK) - THE QUOTIENT TREE PARTITIONING.
17. C      PERM - THE PERMUTATION VECTOR.
18. C
19. C      WORKING PARAMETERS -

```

```

20. C      (XLS, LS) - THIS LEVEL STRUCTURE PAIR IS USED BY
21. C          FNROOT TO FIND A PSEUDO-PERIPHERAL NODE.
22. C      NODLVL - A TEMPORARY VECTOR TO STORE THE LEVEL
23. C          NUMBER OF EACH NODE IN A LEVEL STRUCTURE.
24. C
25. C      PROGRAM SUBROUTINES -
26. C          FNLVLS, RQTREE.
27. C
28. C*****
29. C
30. C      SUBROUTINE GENRQT ( NEQNS, XADJ, ADJNCY, NBLKS, XBLK,
31. C          1          PERM, XLS, LS, NODLVL )
32. C
33. C*****
34. C
35. C          INTEGER ADJNCY(1), LS(1), NODLVL(1), PERM(1),
36. C          1          XBLK(1), XLS(1)
37. C          INTEGER XADJ(1), I, IXLS, LEAF, NBLKS, NEQNS, NLVL,
38. C          1          ROOT
39. C
40. C*****
41. C
42. C      -----
43. C      INITIALIZATION ...
44. C      -----
45. C          DO 100 I = 1, NEQNS
46. C              NODLVL(I) = 1
47. C          100 CONTINUE
48. C          NBLKS = 0
49. C          XBLK(1) = 1
50. C      -----
51. C      FOR EACH CONNECTED COMPONENT, FIND A ROOTED LEVEL
52. C      STRUCTURE, AND THEN CALL RQTREE FOR ITS BLOCK ORDER.
53. C      -----
54. C          DO 200 I = 1, NEQNS
55. C              IF (NODLVL(I) .LE. 0) GO TO 200
56. C              ROOT = I
57. C              CALL FNLVLS ( ROOT, XADJ, ADJNCY, NODLVL,
58. C          1              NLVL, XLS, LS )
59. C              IXLS = XLS(NLVL)
60. C              LEAF = LS(IXLS)
61. C              CALL RQTREE ( LEAF, XADJ, ADJNCY, PERM,
62. C          1              NBLKS, XBLK, NODLVL, XLS, LS )
63. C          200 CONTINUE
64. C          RETURN
65. C      END

```

---

**FNLVLS (FiNd LeVeL Structure)**

This subroutine FNLVLS generates a rooted level structure for a component, specified by NODLVL and rooted at a pseudo-peripheral node. In addition, it also records the level number of the nodes in the level structure.

The connected component is specified by the input parameters ROOT, XADJ, ADJNCY and NODLVL. The subroutine first calls the subroutine FNROOT to obtain the required rooted level structure, given by (NLVL, XLS, LS). It then loops through the level structure to determine the level numbers and puts them into NODLVL (loop DO 200 LVL = ...).

```

1.  C*****
2.  C*****
3.  C*****      FNLVLS . . . . FIND LEVEL STRUCTURE      *****
4.  C*****
5.  C*****
6.  C
7.  C      PURPOSE - FNLVLS GENERATES A ROOTED LEVEL STRUCTURE FOR
8.  C      A MASKED CONNECTED SUBGRAPH, ROOTED AT A PSEUDO-
9.  C      PERIPHERAL NODE.  THE LEVEL NUMBERS ARE RECORDED.
10. C
11. C      INPUT PARAMETERS -
12. C      (XADJ, ADJNCY) - THE ADJACENCY STRUCTURE.
13. C
14. C      OUTPUT PARAMETERS -
15. C      NLVL - NUMBER OF LEVELS IN THE LEVEL STRUCTURE FOUND.
16. C      (XLS, LS) - THE LEVEL STRUCTURE RETURNED.
17. C
18. C      UPDATED PARAMETERS -
19. C      ROOT - ON INPUT, WITH THE ARRAY NODLVL, SPECIFIES
20. C      THE COMPONENT WHOSE PSEUDO-PERIPHERAL NODE IS
21. C      TO BE FOUND. ON OUTPUT, IT CONTAINS THAT NODE.
22. C      NODLVL - ON INPUT, IT SPECIFIES A SECTION SUBGRAPH.
23. C      ON RETURN, IT CONTAINS THE NODE LEVEL NUMBERS.
24. C
25. C      PROGRAM SUBROUTINES -
26. C      FNROOT.
27. C
28. C*****
29. C
30. C      SUBROUTINE FNLVLS ( ROOT, XADJ, ADJNCY, NODLVL,
31. C      1                      NLVL, XLS, LS )
32. C
33. C*****
34. C
35. C      INTEGER ADJNCY(1), LS(1), NODLVL(1), XLS(1)

```

```

36 .           INTEGER XADJ(1), J, LBEGIN, LVL, LVLEND, NLVL,
37 .           1           NODE, ROOT
38 . C
39 . C*****
40 . C
41 .           CALL FNROOT ( ROOT, XADJ, ADJNCY, NODLVL,
42 .           1           NLVL, XLS, LS )
43 .           DO 200 LVL = 1, NLVL
44 .               LBEGIN = XLS(LVL)
45 .               LVLEND = XLS(LVL + 1) - 1
46 .               DO 100 J = LBEGIN, LVLEND
47 .                   NODE = LS(J)
48 .                   NODLVL(NODE) = LVL
49 .           100       CONTINUE
50 .           200       CONTINUE
51 .           RETURN
52 .           END

```

---

### RQTREE (Refined Quotient TREE)

This is the subroutine that actually applies the quotient tree algorithm as described in Section 6.4.1. Throughout the procedure, it maintains a stack of node subsets. Before going into the details of the subroutine, we first consider the organization of the stack.

For each node subset in the stack, we need to store its size and the level number of its nodes. In the storage array called **STACK**, we store the nodes in the subset in contiguous locations, and then the subset size and the level number in the next two locations. We also keep a variable **TOPSTK** that stores the current number of locations used in **STACK**. Figure 6.4.8 contains an illustration of the organization of the vector **STACK**.

To push a subset  $S$  of level  $i$  into the stack, we simply copy the nodes in  $S$  into the vector **STACK** starting at location **TOPSTK**+1. We then enter the size  $|S|$  and the level number  $i$  and finally update the value of **TOPSTK**. On the other hand, to pop a node subset from the stack, we first obtain the size of the subset from **STACK**(**TOPSTK**-1) and then the subset can be retrieved from **STACK** starting at **TOPSTK**-size-1. The value of **TOPSTK** is also updated to reflect the current status of the stack.

We now consider the details of the subroutine **RQTREE**. It operates on a connected subgraph as specified by **LEAF**, **XADJ**, **ADJNCY** and **NODLVL**. It implicitly assumes that a level structure has been formed on this component, where **NODLVL** contains the level numbers for its nodes and **LEAF** is a *leaf node* in

the level structure. In a level structure  $\mathcal{L} = \{L_0, L_1, \dots, L_l\}$ , a node  $x$  is said to be a *leaf* in  $\mathcal{L}$  if  $Adj(x) \cap L_{i+1} = \phi$  where  $x \in L_i$ .

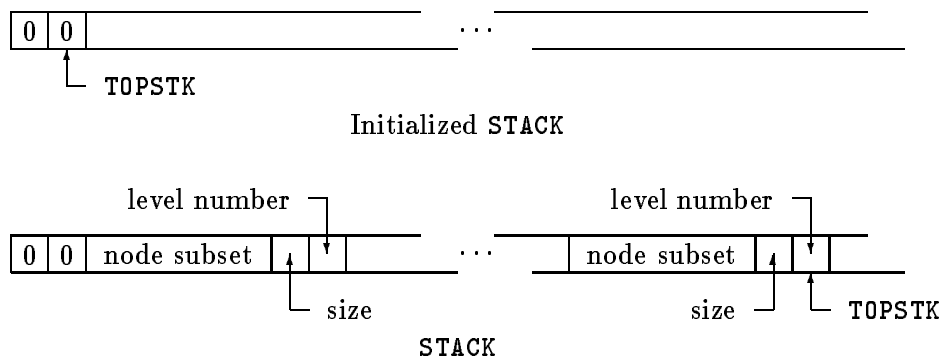


Figure 6.4.8: Organization of the stack in the subroutine RQTREE.

---

In addition to **STACK**, the subroutine uses a second working vector **ADJS**. This vector is used to store the adjacent set of the current block in the lower level, and it is a potential subset for the next block.

The subroutine starts by initializing the **STACK** vector, its pointer **TOPSTK** and the variable **TOPLVL**. The variable **TOPLVL** is local to the subroutine and it stores the level number of the top subset in the stack. A leaf block is then determined by calling the subroutine **FNSPAN** on the node **LEAF**. (A *leaf block* is a subset  $Y$  such that  $Adj(Y) \cap L_{i+1} = \phi$  where  $Y \subset L_i$ .) It is numbered as the next block (statement labelled 300).

We then march onto the next lower level (**LEVEL = LEVEL - 1** and following). The adjacent set of the previous block in this level is used to start building up the next potential block. If the node subset at the top of the **STACK** vector belongs to the same level, it is popped from the stack and included into the potential block. Then, the subroutine **FNSPAN** is called (statement labelled 400) to obtain the span of this subset. If the span does not have any unnumbered neighbors in the higher level, it becomes the next block to be numbered. Otherwise, the span is pushed into the stack and instead a leaf block is determined as the next one to be numbered.

The subroutine goes through all the levels until it comes to the first one. By this time, all the nodes in the component should have been numbered and the subroutine returns.

---

```

1. C*****
2. C*****
3. C*****      RQTREE . . . . REFINED QUOTIENT TREE      *****
4. C*****
5. C*****
6. C
7. C      PURPOSE - THIS SUBROUTINE FINDS A QUOTIENT TREE ORDERING
8. C      FOR THE COMPONENT SPECIFIED BY LEAF AND NODLVL.
9. C
10. C      INPUT PARAMETERS -
11. C          (XADJ, ADJNCY) - THE ADJACENCY STRUCTURE.
12. C          LEAF - THE INPUT NODE THAT DEFINES THE CONNECTED
13. C          COMPONENT. IT IS ALSO A LEAF NODE IN THE
14. C          ROOTED LEVEL STRUCTURE PASSED TO RQTREE.
15. C          I.E. IT HAS NO NEIGHBOR IN THE NEXT LEVEL.
16. C
17. C      OUTPUT PARAMETERS -
18. C          PERM - THE PERMUTATION VECTOR CONTAINING THE ORDERING.
19. C          (NBLKS, XBLK) - THE QUOTIENT TREE PARTITIONING.
20. C
21. C      UPDATED PARAMETERS -
22. C          NODLVL - THE NODE LEVEL NUMBER VECTOR.  NODES IN THE
23. C          COMPONENT HAVE THEIR NODLVL SET TO ZERO AS
24. C          AS THEY ARE NUMBERED.
25. C
26. C      WORKING PARAMETERS -
27. C          ADJS - TEMPORARY VECTOR TO STORE THE ADJACENT SET
28. C          OF NODES IN A PARTICULAR LEVEL.
29. C          STACK - TEMPORARY VECTOR USED TO MAINTAIN THE STACK
30. C          OF NODE SUBSETS.  IT IS ORGANISED AS -
31. C          ( SUBSET NODES, SUBSET SIZE, SUBSET LEVEL ) . . . .
32. C
33. C      PROGRAM SUBROUTINES -
34. C          FNSPAN, COPYSI.
35. C
36. C*****
37. C
38. C      SUBROUTINE RQTREE ( LEAF, XADJ, ADJNCY, PERM,
39. C          1          NBLKS, XBLK, NODLVL, ADJS, STACK )
40. C
41. C*****
42. C
43. C          INTEGER ADJNCY(1), ADJS(1), NODLVL(1), PERM(1),
44. C          1          STACK(1), XBLK(1)
45. C          INTEGER XADJ(1), BLKSZE, IP, J, JP, LEAF, LEVEL,
46. C          1          NADJS, NBLKS, NODE, NPOP, NULEAF,

```

```

47.      1          NUM, TOPLVL, TOPSTK
48.  C
49.  C*****
50.  C
51.  C      -----
52.  C      INITIALIZE THE STACK VECTOR AND ITS POINTERS.
53.  C      -----
54.      STACK(1) = 0
55.      STACK(2) = 0
56.      TOPSTK = 2
57.      TOPLVL = 0
58.      NUM = XBLK(NBLKS+1) - 1
59.  C      -----
60.  C      FORM A LEAF BLOCK, THAT IS, ONE WITH NO NEIGHBORS
61.  C      IN ITS NEXT HIGHER LEVEL.
62.  C      -----
63.      100      LEVEL = NODLVL(LEAF)
64.      NODLVL(LEAF) = 0
65.      PERM(NUM+1) = LEAF
66.      BLKSZE = 1
67.      CALL FNSPAN ( XADJ, ADJNCY, NODLVL, BLKSZE, PERM(NUM+1),
68.      1          LEVEL, NADJS, ADJS, NULEAF )
69.      IF ( NULEAF .LE. 0 ) GO TO 300
70.      JP = NUM
71.      DO 200 J = 1, BLKSZE
72.      JP = JP + 1
73.      NODE = PERM(JP)
74.      NODLVL(NODE) = LEVEL
75.      200      CONTINUE
76.      LEAF = NULEAF
77.      GO TO 100
78.  C      -----
79.  C      A NEW BLOCK HAS BEEN FOUND ...
80.  C      -----
81.      300      NBLKS = NBLKS + 1
82.      XBLK(NBLKS) = NUM + 1
83.      NUM = NUM + BLKSZE
84.  C      -----
85.  C      FIND THE NEXT POSSIBLE BLOCK BY USING THE ADJACENT
86.  C      SET IN THE LOWER LEVEL AND THE TOP NODE SUBSET (IF
87.  C      APPROPRIATE) IN THE STACK.
88.  C      -----
89.      LEVEL = LEVEL - 1
90.      IF ( LEVEL .LE. 0 ) GO TO 500
91.      CALL COPYSI ( NADJS, ADJS, PERM(NUM+1) )
92.      BLKSZE = NADJS
93.      IF ( LEVEL .NE. TOPLVL ) GO TO 400

```

```

94. C -----
95. C THE LEVEL OF THE NODE SUBSET AT THE TOP OF THE
96. C STACK IS THE SAME AS THAT OF THE ADJACENT SET.
97. C POP THE NODE SUBSET FROM THE STACK.
98. C -----
99. C NPOP = STACK(TOPSTK-1)
100. C TOPSTK = TOPSTK - NPOP - 2
101. C IP = NUM + BLKSZE + 1
102. C CALL COPYSI ( NPOP, STACK(TOPSTK+1), PERM(IP) )
103. C BLKSZE = BLKSZE + NPOP
104. C TOPLVL = STACK(TOPSTK)
105. C 400 CALL FNSPAN ( XADJ, ADJNCY, NODLVL, BLKSZE,
106. C 1 PERM(NUM+1), LEVEL, NADJS, ADJS, NULEAF )
107. C IF ( NULEAF .LE. 0 ) GO TO 300
108. C -----
109. C PUSH THE CURRENT NODE SET INTO THE STACK.
110. C -----
111. C CALL COPYSI ( BLKSZE, PERM(NUM+1), STACK(TOPSTK+1) )
112. C TOPSTK = TOPSTK + BLKSZE + 2
113. C STACK(TOPSTK-1) = BLKSZE
114. C STACK(TOPSTK) = LEVEL
115. C TOPLVL = LEVEL
116. C LEAF = NULEAF
117. C GO TO 100
118. C -----
119. C BEFORE EXIT ...
120. C -----
121. C 500 XBLK(NBLKS+1) = NUM + 1
122. C RETURN
123. C END

```

---

```

1. C*****
2. C*****
3. C*****      COPYSI . . . . COPY INTEGER VECTOR      *****
4. C*****
5. C*****
6. C
7. C PURPOSE - THIS ROUTINE COPIES THE N INTEGER ELEMENTS FROM
8. C THE VECTOR A TO B. (ARRAYS OF SHORT INTEGERS)
9. C
10. C INPUT PARAMETERS -
11. C N - SIZE OF VECTOR A.
12. C A - THE INTEGER VECTOR.
13. C
14. C OUTPUT PARAMETER -
15. C B - THE OUTPUT INTEGER VECTOR.

```



```

16. C
17. C*****
18. C
19.     SUBROUTINE COPYSI ( N, A, B )
20. C
21. C*****
22. C
23.     INTEGER A(1), B(1)
24.     INTEGER I, N
25. C
26. C*****
27. C
28.     IF ( N .LE. 0 ) RETURN
29.     DO 100 I = 1, N
30.         B(I) = A(I)
31.     100 CONTINUE
32.     RETURN
33.     END

```

---

### **FNSPAN (FiNd SPAN)**

This subroutine is used by the subroutine **RQTREE** and has several functions, one of which is to find the span of a set. Let  $\mathcal{L} = \{L_0, L_1, \dots, L_l\}$  be a given level structure and let  $S$  be a subset in level  $L_i$ . This subroutine determines the span of  $S$  in the subgraph  $\mathcal{G}(L_i)$  and finds the adjacent set of  $S$  in level  $L_{i-1}$ . Moreover, if the span of  $S$  has some unnumbered neighbors in level  $L_{i+1}$ , the subroutine returns an unnumbered leaf node and in that case, the span of  $S$  may only be partially formed.

Inputs to this subroutine are the graph structure in the array pair (**XADJ**, **ADJNCY**), the level structure stored implicitly in the vector **NODLVL**, and the subset (**NSPAN**, **SET**) in level **LEVEL** of the level structure. On return, the vector **SET** is expanded to accommodate the span of this given set. The variable **NSPAN** will be increased to the size of the span set.

After initialization, the subroutine goes through each node in the partially spanned set. Here, the variable **SETPTR** points to the current node in the span set under consideration. The loop **DO 500 J = . . .** is then executed to inspect the level numbers of its neighbors. Depending on the level number, the neighbor is either bypassed or included in the span set or included in the adjacent set. A final possibility is when the neighbor belongs to a higher level. In this case, a path through unnumbered nodes is traced down the level

structure until we hit a leaf node. The subroutine returns after recovering the nodes in the partially formed adjacent set (loop DO 900 I = ...). A normal return from FNSPAN will have the span set in (NSPAN, SET) and the adjacent set in (NADJS, ADJS) completely formed, and have zero in the variable LEAF.

---

```

1. C*****
2. C*****
3. C*****      FNSPAN ..... FIND SPAN SET      *****
4. C*****
5. C*****
6. C
7. C   PURPOSE - THIS SUBROUTINE IS ONLY USED BY RQTREE. ITS
8. C   MAIN PURPOSE IS TO FIND THE SPAN OF A GIVEN SUBSET
9. C   IN A GIVEN LEVEL SUBGRAPH IN A LEVEL STRUCTURE.
10. C   THE ADJACENT SET OF THE SPAN IN THE LOWER LEVEL IS
11. C   ALSO DETERMINED. IF THE SPAN HAS AN UNNUMBERED NODE
12. C   IN THE HIGHER LEVEL, AN UNNUMBERED LEAF NODE (I.E. ONE
13. C   WITH NO NEIGHBOR IN NEXT LEVEL) WILL BE RETURNED.
14. C
15. C   INPUT PARAMETERS -
16. C   (XADJ, ADJNCY) - THE ADJACENT STRUCTURE.
17. C   LEVEL - LEVEL NUMBER OF THE CURRENT SET.
18. C
19. C   UPDATED PARAMETERS -
20. C   (NSPAN, SET) - THE INPUT SET. ON RETURN, IT CONTAINS
21. C   THE RESULTING SPAN SET.
22. C   NODLVL - THE LEVEL NUMBER VECTOR. NODES CONSIDERED
23. C   WILL HAVE THEIR NODLVL CHANGED TO ZERO.
24. C
25. C   OUTPUT PARAMETERS -
26. C   (NADJS, ADJS) - THE ADJACENT SET OF THE SPAN IN THE
27. C   LOWER LEVEL.
28. C   LEAF - IF THE SPAN HAS AN UNNUMBERED HIGHER LEVEL NODE,
29. C   LEAF RETURNS AN UNNUMBERED LEAF NODE IN THE LEVEL
30. C   STRUCTURE, OTHERWISE, LEAF IS ZERO.
31. C
32. C
33. C*****
34. C
35. C   SUBROUTINE FNSPAN ( XADJ, ADJNCY, NODLVL, NSPAN, SET,
36. C   1                      LEVEL, NADJS, ADJS, LEAF )
37. C
38. C*****
39. C
40. C   INTEGER ADJNCY(1), ADJS(1), NODLVL(1), SET(1)
41. C   INTEGER XADJ(1), I, J, JSTOP, JSTRT, LEAF, LEVEL,

```

```

42.      1          LVL, LVL1, NADJS, NBR, NBRLVL, NODE,
43.      1          NSPAN, SETPTR
44.  C
45.  C*****
46.  C
47.  C          -----
48.  C          INITIALIZATION ...
49.  C          -----
50.          LEAF = 0
51.          NADJS = 0
52.          SETPTR = 0
53.      100      SETPTR = SETPTR + 1
54.          IF ( SETPTR .GT. NSPAN ) RETURN
55.  C          -----
56.  C          FOR EACH NODE IN THE PARTIALLY SPANNED SET ...
57.  C          -----
58.          NODE = SET(SETPTR)
59.          JSTRT = XADJ(NODE)
60.          JSTOP = XADJ(NODE + 1) - 1
61.          IF ( JSTOP .LT. JSTRT ) GO TO 100
62.  C          -----
63.  C          FOR EACH NEIGHBOR OF NODE, TEST ITS NODLVL VALUE ...
64.  C          -----
65.          DO 500 J = JSTRT, JSTOP
66.              NBR = ADJNCY(J)
67.              NBRLVL = NODLVL(NBR)
68.              IF (NBRLVL .LE. 0) GO TO 500
69.              IF (NBRLVL - LEVEL) 200, 300, 600
70.  C          -----
71.  C          NBR IS IN LEVEL-1, ADD IT TO ADJS.
72.  C          -----
73.      200      NADJS = NADJS + 1
74.              ADJS(NADJS) = NBR
75.              GO TO 400
76.  C          -----
77.  C          NBR IS IN LEVEL, ADD IT TO THE SPAN SET.
78.  C          -----
79.      300      NSPAN = NSPAN + 1
80.              SET(NSPAN) = NBR
81.      400      NODLVL(NBR) = 0
82.      500      CONTINUE
83.          GO TO 100
84.  C          -----
85.  C          NBR IS IN LEVEL+1. FIND AN UNNUMBERED LEAF NODE BY
86.  C          TRACING A PATH UP THE LEVEL STRUCTURE. THEN
87.  C          RESET THE NODLVL VALUES OF NODES IN ADJS.
88.  C          -----

```

```

89.   600   LEAF = NBR
90.       LVL = LEVEL + 1
91.   700   JSTRT = XADJ(LEAF)
92.       JSTOP = XADJ(LEAF+1) - 1
93.       DO 800 J = JSTRT, JSTOP
94.           NBR = ADJNCY(J)
95.           IF ( MODLVL(NBR) .LE. LVL ) GO TO 800
96.           LEAF = NBR
97.           LVL = LVL + 1
98.           GO TO 700
99.   800   CONTINUE
100.      IF (NADJS .LE. 0) RETURN
101.      LVL1 = LEVEL - 1
102.      DO 900 I = 1, NADJS
103.          NODE = ADJS(I)
104.          NODLVL(NODE) = LVL1
105.   900   CONTINUE
106.      RETURN
107.      END

```

---

### Exercises

6.4.1) Let  $\mathcal{P} = \{Y_1, Y_2, \dots, Y_p\}$  be the partitioning of  $\mathcal{G}$  generated by the algorithm described in Section 6.4.1.

- a) Show that  $\mathcal{G}/\mathcal{P}$  is a quotient tree.
- b) Prove that the quotient tree generated by the algorithm of Section 6.4.1 is maximal, as defined in Exercise 6.3.4 on page 207. (Hint: Let  $Y \in \mathcal{P}$  and  $Y \subset L_j(r)$ , where  $L_j(r)$  is defined in Section 6.4.1. Show that for any two nodes  $x$  and  $y$  in  $Y$ , there exists a path joining them in  $\mathcal{G}(\bigcup_{i=0}^{j-1} L_i(r))$  and one in  $\mathcal{G}(\bigcup_{i=j}^l L_i(r))$ . Then use the result of Exercise 6.3.4 on page 207.)

## 6.5 A Storage Scheme and Storage Allocation Procedure

In this section we describe a storage scheme which is specially designed for solving partitioned matrix problems whose quotient graphs are monotonely ordered trees. The assumption is that all the off-diagonal blocks of the triangular factor  $L$  are to be discarded in favor of the blocks of the original

matrix  $\mathbf{A}$ . In other words, the implicit solution scheme described at the end of Section 6.3.3 is to be used.

### 6.5.1 The Storage Scheme

For illustrative purposes we again assume  $\mathbf{A}$  is partitioned into  $p^2$  submatrices  $\mathbf{A}_{ij}$ ,  $1 \leq i, j \leq p$ , and let  $\mathbf{L}_{ij}$  be the corresponding submatrices of  $\mathbf{L}$ , where  $\mathbf{A} = \mathbf{L}\mathbf{L}^T$ . Since we do not know whether  $\mathbf{A}$  will have the form

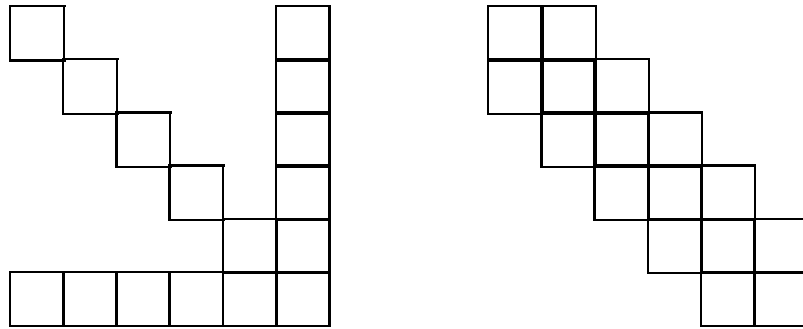


Figure 6.5.1: Examples of matrices whose quotient graphs are trees.

(which correspond to quite different quotient trees), or something in between, the storage scheme must be quite flexible. Define the matrices

$$\mathbf{V}_k = \begin{pmatrix} \mathbf{A}_{1k} \\ \mathbf{A}_{2k} \\ \vdots \\ \mathbf{A}_{k-1,k} \end{pmatrix}, \quad 2 \leq k \leq p. \quad (6.5.1)$$

Thus,  $\mathbf{A}$  can be viewed as follows, where  $p$  is chosen to be 5.

Now our computational scheme requires that we store the diagonal blocks  $\mathbf{L}_{kk}$ ,  $1 \leq k \leq p$ , and the non-null off-diagonal blocks of  $\mathbf{A}$ . The storage scheme we use is illustrated in Figure 6.5.3. The diagonal blocks of  $\mathbf{L}$  are viewed as forming a single block diagonal matrix which is stored using the envelope storage scheme already described in Section 4.5.1. That is, the diagonal is stored in the array `DIAG`, and the rows of the lower envelope are

$$\left[ \begin{array}{c|c|c|c|c} \mathbf{A}_{11} & \mathbf{V}_2 & \mathbf{V}_3 & & \\ \hline \mathbf{A}_{12}^T & \mathbf{A}_{22} & & \mathbf{V}_4 & \\ \mathbf{A}_{13}^T & \mathbf{A}_{23}^T & \mathbf{A}_{33} & & \mathbf{V}_5 \\ \hline \mathbf{A}_{14}^T & \mathbf{A}_{24}^T & \mathbf{A}_{34}^T & \mathbf{A}_{44} & \\ \hline \mathbf{A}_{15}^T & \mathbf{A}_{25}^T & \mathbf{A}_{35}^T & \mathbf{A}_{45}^T & \mathbf{A}_{55} \end{array} \right]$$

Figure 6.5.2: A partitioned matrix.

stored using the array pair (XENV, ENV). In addition, an array XBLK of length  $p + 1$  is used to record the partitioning  $\mathcal{P}$ :  $\text{XBLK}(k)$  is the number of the first row of the  $k$ -th diagonal block, and for convenience we set  $\text{XBLK}(p+1) = n+1$ . The nonzero components of the  $\mathbf{V}_k$ ,  $1 < k \leq p$  are stored in a single one dimensional array NONZ, column by column, beginning with those of  $\mathbf{V}_2$ . A parallel integer array NZSUBS is used to store the row subscripts of the numbers in NONZ, and a vector XNONZ of length  $n + 1$  contains the positions in NONZ where each column resides. For programming convenience, we set  $\text{XNONZ}(n + 1) = \eta + 1$ , where  $\eta$  denotes the number of components in NONZ. Note that  $\text{XNONZ}(i + 1) = \text{XNONZ}(i)$  implies that the corresponding column of  $\mathbf{V}_k$  is null.

Suppose  $\text{XBLK}(k) \leq i < \text{XBLK}(k+1)$ , and we wish to print the  $(i - \text{XBLK}(k) + 1)$ -st column of  $\mathbf{A}_{jk}$ , where  $j < k$ . The following code segment illustrates how this could be done. The elements of each row in NONZ are assumed to be stored in order of increasing row subscript.

```

MSTRT = XNONZ(I)
MSTOP = XNONZ(I+1)-1
IF (MSTOP.LT.MSTRT) GO TO 200
DO 100 M = MSTRT, MSTOP
  ROW = NZSUBS(M)
  IF (ROW.LT.XBLK(J)) GO TO 100
  IF (ROW.GT.XBLK(J+1)) GO TO 200
  VALUE = NONZ(M)

```

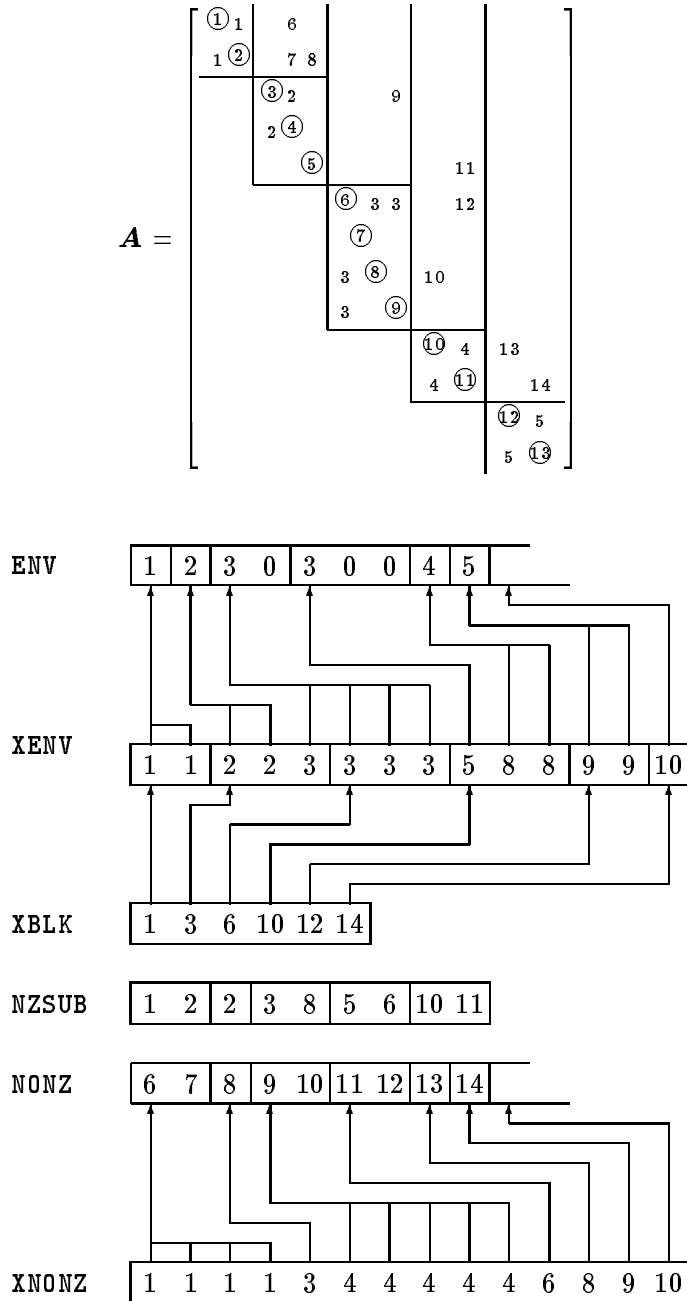


Figure 6.5.3: Example showing the arrays used in the quotient tree storage scheme.

```

        WRITE (6,3000) ROW, VALUE
3000    FORMAT (1X,15H ROW SUBSCRIPT=,I3,7H VALUE=,F12.6)
100    CONTINUE
200    CONTINUE
      .
      .
      .

```

The storage required for the vectors `XENV`, `XBLK`, `XNONZ` and `NZSUBS` should be regarded as overhead storage, (recall our remarks in Section 2.4.1) since it is not used for actual data. In addition we will need some temporary storage to implement the factorization and solution procedures. We discuss this aspect in Section 6.6, where we deal with the numerical computation subroutines `TSFCT` (Tree Symmetric FaCTorization) and `TSSLV` (Tree Symmetric SoLVe).

### 6.5.2 Internal Renumbering of the Blocks

The ordering algorithm described in Section 6.4 determines a tree partitioning for a connected graph. So far, we have assumed that nodes *within* a block (or a partition member) are labelled arbitrarily. This certainly does not affect the number of off-block-diagonal nonzeros in the original matrix. However, since the storage scheme stores the diagonal blocks using the envelope structure, the way nodes are arranged within a block can affect the primary storage for the diagonal envelope. It is the purpose of this section to discuss an internal numbering strategy and describe its implementation. The strategy should use some envelope/profile reduction scheme on each block, and the reverse Cuthill-McKee algorithm, which is simple and quite effective (see Section 4.4.1), seems to be suitable for this purpose. The method is described below. Let  $\mathcal{P} = \{Y_1, Y_2, \dots, Y_p\}$  be a given monotonely ordered quotient tree partitioning.

For each block  $Y_k$  in  $\mathcal{P}$ , do the following:

**Step 1** Determine the subset

$$U = \{y \in Y_k \mid \text{Adj}(y) \cap (Y_1 \cup \dots \cup Y_{k-1}) = \phi\}.$$

**Step 2** Reorder the nodes in  $\mathcal{G}(U)$  by the reverse Cuthill-McKee algorithm.

**Step 3** Number the nodes in  $Y_k - U$  after  $U$  in arbitrary order.



The example in Figure 6.5.5 serves to demonstrate the effect of this renumbering step. The envelope of the diagonal blocks for the ordering  $\alpha_1$  has size 24, whereas the diagonal blocks for  $\alpha_2$  have only a total of 11 entries in their envelopes. Indeed, the relabelling can yield a significant reduction of the storage requirement.

The implementation of this internal-block renumbering scheme is quite straightforward. It consists of two new subroutines `BSHUFL` and `SUBRCM`, along with the use of three others which have already been discussed in previous chapters.

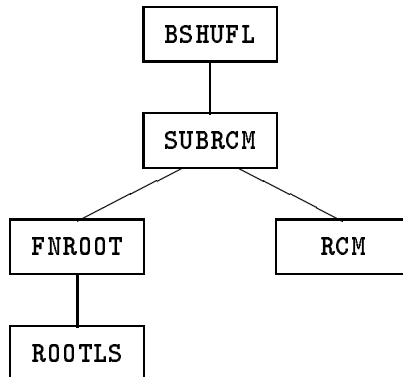


Figure 6.5.4: Control relation of subroutines for the refined quotient tree algorithm.

---

They are discussed in detail below.

#### **BSHUFL (Block SHUFfLe)**

Inputs to this subroutine are the graph structure in (`XADJ`, `ADJNCY`), the quotient tree partitioning in (`NBLKS`, `XBLK`) and `PERM`. The subroutine will shuffle the permutation vector `PERM` according to the scheme described earlier in this section. It needs four working vectors: `BNUM` for storing the block number of each node, `SUBG` for accumulating nodes in a subgraph, and `MASK` and `XLS` for the execution of the subroutine `SUBRCM`.

The subroutine begins by initializing the working vectors `BNUM` and `MASK` (loop `D0 200 K = ...`). The loop `D0 500 K = ...` goes through each block in the partitioning. For each block, all those nodes with no neighbors in the

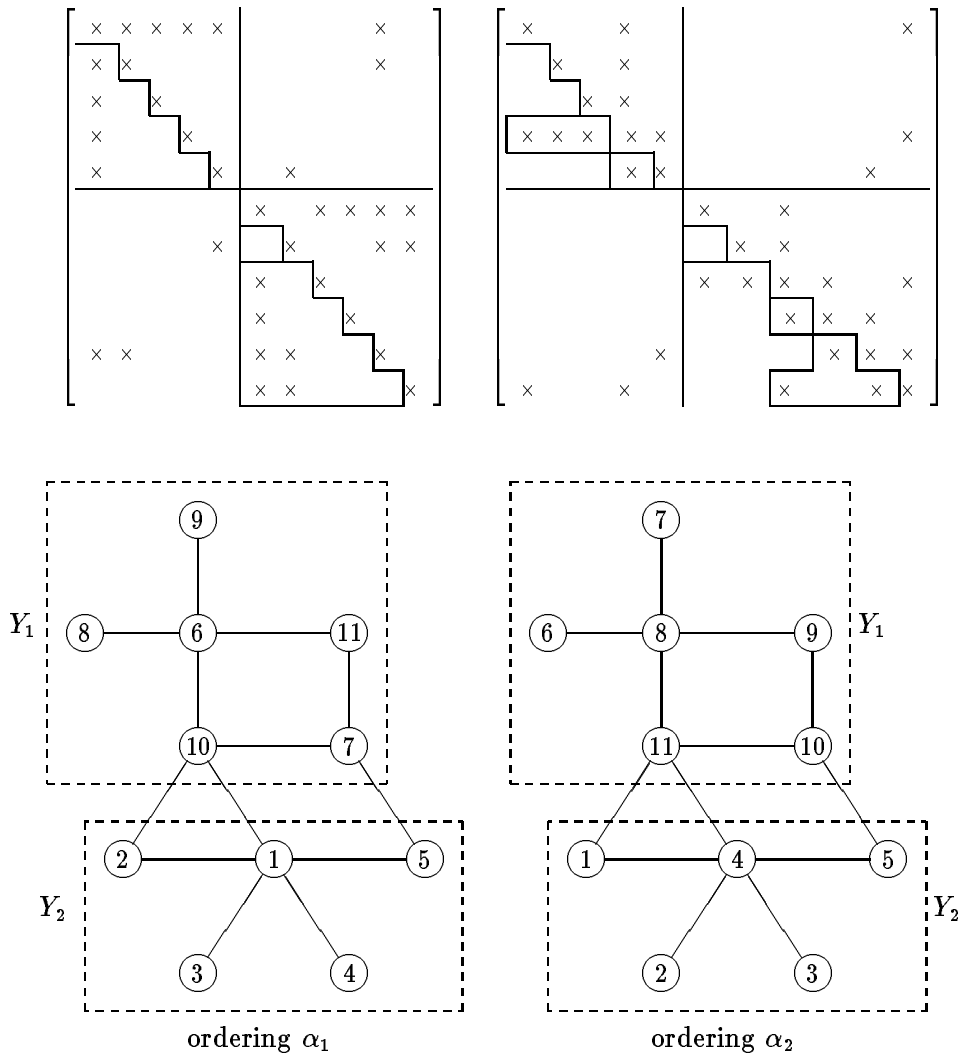


Figure 6.5.5: Example to show the effect of within-block relabelling.

previous blocks are accumulated in the vector SUBG (loop D0 400 ...). The variable NSUBG keeps the number of nodes in the subgraph. The subroutine SUBRCM is then called to renumber this subgraph using the RCM algorithm. The program returns after all the blocks have been processed.

---

```

1. C*****
2. C*****
3. C*****      BSHUFL . . . . INTERNAL BLOCK SHUFFLE      *****
4. C*****
5. C*****
6. C
7. C      PURPOSE - TO  RENUMBER THE NODES OF EACH BLOCK
8. C      SO AS TO REDUCE ITS ENVELOPE.
9. C      NODES IN A BLOCK WITH NO NEIGHBORS IN PREVIOUS
10. C      BLOCKS ARE RENUMBERED BY SUBRCM BEFORE THE OTHERS.
11. C
12. C      INPUT PARAMETERS -
13. C      (XADJ, ADJNCY) - THE GRAPH ADJACENCY STRUCTURE.
14. C      (NBLKS, XBLK ) - THE TREE PARTITIONING.
15. C
16. C      UPDATED PARAMETER -
17. C      PERM - THE PERMUTATION VECTOR. ON RETURN, IT CONTAINS
18. C      THE NEW PERMUTATION.
19. C
20. C      WORKING VECTORS -
21. C      BNUM - STORES THE BLOCK NUMBER OF EACH VARIABLE.
22. C      MASK - MASK VECTOR USED TO PRESCRIBE A SUBGRAPH.
23. C      SUBG - VECTOR USED TO CONTAIN A SUBGRAPH.
24. C      XLS - INDEX VECTOR TO A LEVEL STRUCTURE.
25. C
26. C      PROGRAM SUBROUTINE -
27. C      SUBRCM.
28. C
29. C*****
30. C
31. C      SUBROUTINE BSHUFL ( XADJ, ADJNCY, PERM, NBLKS, XBLK,
32. C      1                      BNUM, MASK, SUBG, XLS )
33. C
34. C*****
35. C
36. C      INTEGER ADJNCY(1), BNUM(1), MASK(1), PERM(1),
37. C      1          SUBG(1), XBLK(1), XLS(1)
38. C      INTEGER XADJ(1), I, IP, ISTOP, ISTRT, J,
39. C      1          JSTRT, JSTOP, K, NABOR, NBLKS, NBRBLK,
40. C      1          NODE, NSUBG
41. C
42. C*****

```

```

43. C
44.     IF ( NBLKS .LE. 0 ) RETURN
45. C -----
46. C     INITIALIZATION . . . . FIND THE BLOCK NUMBER FOR EACH
47. C     VARIABLE AND INITIALIZE THE VECTOR MASK.
48. C -----
49.     DO 200 K = 1, NBLKS
50.         ISTRT = XBLK(K)
51.         ISTOP = XBLK(K+1) - 1
52.         DO 100 I = ISTRT,ISTOP
53.             NODE = PERM(I)
54.             BNUM(NODE) = K
55.             MASK(NODE) = 0
56.     100     CONTINUE
57.     200     CONTINUE
58. C -----
59. C     FOR EACH BLOCK, FIND THOSE NODES WITH NO NEIGHBORS
60. C     IN PREVIOUS BLOCKS AND ACCUMULATE THEM IN SUBG.
61. C     THEY WILL BE RENUMBERED BEFORE OTHERS IN THE BLOCK.
62. C -----
63.     DO 500 K = 1,NBLKS
64.         ISTRT = XBLK(K)
65.         ISTOP = XBLK(K+1) - 1
66.         NSUBG = 0
67.         DO 400 I = ISTRT, ISTOP
68.             NODE = PERM(I)
69.             JSTRT = XADJ(NODE)
70.             JSTOP = XADJ(NODE+1) - 1
71.             IF (JSTOP .LT. JSTRT) GO TO 400
72.             DO 300 J = JSTRT, JSTOP
73.                 NABOR = ADJNCY(J)
74.                 NBRBLK = BNUM(NABOR)
75.                 IF (NBRBLK .LT. K) GO TO 400
76.     300     CONTINUE
77.             NSUBG = NSUBG + 1
78.             SUBG(NSUBG) = NODE
79.             IP = ISTRT + NSUBG - 1
80.             PERM(I) = PERM(IP)
81.     400     CONTINUE
82. C -----
83. C     CALL SUBRCM TO RENUMBER THE SUBGRAPH STORED
84. C     IN (NSUBG, SUBG).
85. C -----
86.     IF ( NSUBG .GT. 0 )
87.     1     CALL SUBRCM ( XADJ, ADJNCY, MASK, NSUBG,
88.     1     SUBG, PERM(ISTRT), XLS )
89.     500     CONTINUE

```



```

36.          INTEGER ADJNCY(1), MASK(1), PERM(1), SUBG(1),
37.      1          XLS(1)
38.          INTEGER XADJ(1), CCSIZE, I, NLVL, NODE, NSUBG, NUM
39.      C
40.      C*****
41.      C
42.          DO 100 I = 1, NSUBG
43.              NODE = SUBG(I)
44.              MASK(NODE) = 1
45.      100  CONTINUE
46.          NUM = 0
47.          DO 200 I = 1, NSUBG
48.              NODE = SUBG(I)
49.              IF ( MASK(NODE) .LE. 0 ) GO TO 200
50.      C          -----
51.      C          FOR EACH CONNECTED COMPONENT IN THE SUBGRAPH,
52.      C          CALL FNROOT AND RCM FOR THE ORDERING.
53.      C          -----
54.          CALL FNROOT ( NODE, XADJ, ADJNCY, MASK,
55.      1              NLVL, XLS, PERM(NUM+1) )
56.          CALL      RCM ( NODE, XADJ, ADJNCY, MASK,
57.      1              PERM(NUM+1), CCSIZE, XLS )
58.          NUM = NUM + CCSIZE
59.          IF ( NUM .GE. NSUBG ) RETURN
60.      200  CONTINUE
61.          RETURN
62.          END

```

---

### 6.5.3 Storage Allocation and the Subroutines FNTENV, FNOFNZ, and FNTADJ

We now describe two subroutines **FNTENV** (FiNd Tree ENvelope) and **FNOFNZ** (FiNd OFF-diagonal NonZeros) which are designed to accept as input a graph  $\mathcal{G}$ , an ordering  $\alpha$ , and a partitioning  $\mathcal{P}$ , and set up the data structure we described in Section 6.5.1. In addition, in order to obtain an efficient implementation of the numerical factorization procedure, it is necessary to construct a vector containing the adjacency structure of the associated quotient tree  $\mathcal{G}/\mathcal{P}$ . This is the function of the third subroutine **FNTADJ** (FiNd Tree ADJacency) which we also describe in this section.

**FNTENV (FiNd Tree ENVELOPE)**

This subroutine finds the envelope structure of the diagonal blocks in a partitioned matrix. It accepts as input the adjacency structure (XADJ, ADJNCY), the ordering (PERM, INVP) and the quotient tree partitioning (NBLKS, XBLK). The structure in XENV produced by FNTENV may not be exactly the envelope structure of the diagonal blocks, although it always contains the actual envelope structure. For the sake of simplicity and efficiency, it uses the following observation in the construction of the envelope structure.

Let  $\mathcal{P} = \{Y_1, \dots, Y_p\}$  be the given tree partitioning, and  $x_i, x_j \in Y_k$ . If

$$Adj(x_i) \cap \{Y_1, \dots, Y_{k-1}\} \neq \phi$$

and

$$Adj(x_j) \cap \{Y_1, \dots, Y_{k-1}\} \neq \phi,$$

the subroutine will include  $\{x_i, x_j\}$  in the envelope structure of the diagonal blocks.

Although this algorithm can yield an unnecessarily large envelope for the diagonal block, (Why? Give an example.) for orderings generated by the RQT algorithm, it usually comes very close to obtaining the exact envelope. Because it works so well, we use it rather than a more sophisticated (and more expensive) scheme which would find the exact envelope. For other quotient tree ordering algorithms, such as the one-way dissection algorithm described in Chapter 7, a more sophisticated scheme is required. (See Section 7.4.3.)

---

```

1. C*****
2. C*****
3. C*****      FNTENV . . . . FIND TREE DIAGONAL ENVELOPE *****
4. C*****
5. C*****
6. C
7. C      PURPOSE - THIS SUBROUTINE DETERMINES THE ENVELOPE INDEX
8. C      VECTOR FOR THE ENVELOPE OF THE DIAGONAL BLOCKS OF A
9. C      TREE PARTITIONED SYSTEM.
10. C
11. C      INPUT PARAMETERS -
12. C      (XADJ, ADJNCY) - ADJACENCY STRUCTURE PAIR FOR THE GRAPH.
13. C      (PERM, INVP) - THE PERMUTATION VECTORS.
14. C      (NBLKS, XBLK) - THE TREE PARTITIONING.
15. C
16. C      OUTPUT PARAMETERS -
17. C      XENV - THE ENVELOPE INDEX VECTOR.
```

6.5. A STORAGE SCHEME AND ALLOCATION PROCEDURE 239

```

18. C          ENVSZE - THE SIZE OF THE ENVELOPE FOUND.
19. C
20. C*****
21. C
22. C          SUBROUTINE FNTENV ( XADJ, ADJNCY, PERM, INVP,
23. C      1                      NBLKS, XBLK, XENV, ENVSZE )
24. C
25. C*****
26. C
27. C          INTEGER ADJNCY(1), INVP(1), PERM(1), XBLK(1)
28. C          INTEGER XADJ(1), XENV(1), BLKBEG, BLKEND,
29. C      1          I, IFIRST, J, JSTOP, JSTRT, K, KFIRST,
30. C      1          ENVSZE, NBLKS, NBR, NODE
31. C
32. C*****
33. C
34. C          ENVSZE = 1
35. C          -----
36. C          LOOP THROUGH EACH BLOCK IN THE PARTITIONING ...
37. C          -----
38. C          DO 400 K = 1, NBLKS
39. C              BLKBEG = XBLK(K)
40. C              BLKEND = XBLK(K+1) - 1
41. C          -----
42. C          KFIRST STORES THE FIRST NODE IN THE K-TH BLOCK
43. C          THAT HAS A NEIGHBOUR IN THE PREVIOUS BLOCKS.
44. C          -----
45. C          KFIRST = BLKEND
46. C          DO 300 I = BLKBEG, BLKEND
47. C              XENV(I) = ENVSZE
48. C              NODE = PERM(I)
49. C              JSTRT = XADJ(NODE)
50. C              JSTOP = XADJ(NODE+1) - 1
51. C              IF ( JSTOP .LT. JSTRT ) GO TO 300
52. C          -----
53. C          IFIRST STORES THE FIRST NONZERO IN THE
54. C          I-TH ROW WITHIN THE K-TH BLOCK.
55. C          -----
56. C          IFIRST = I
57. C          DO 200 J = JSTRT, JSTOP
58. C              NBR = ADJNCY(J)
59. C              NBR = INVP(NBR)
60. C              IF ( NBR .LT. BLKBEG ) GO TO 100
61. C              IF ( NBR .LT. IFIRST ) IFIRST = NBR
62. C              GO TO 200
63. C      100          IF ( KFIRST .LT. IFIRST ) IFIRST = KFIRST
64. C              IF ( I .LT. KFIRST ) KFIRST = I

```



```

65.   200           CONTINUE
66.           ENVSZ = ENVSZ + I - IFIRST
67.   300           CONTINUE
68.   400   CONTINUE
69.           XENV(BLKEND+1) = ENVSZ
70.           ENVSZ = ENVSZ - 1
71.           RETURN
72.           END

```

---

### FNOFNZ (FiNd Off-diagonal NonZeros)

The subroutine FNOFNZ is used to determine the structure of the off-block-diagonal nonzeros in a given partitioned matrix. With respect to the storage scheme in Section 6.5.1, this subroutine finds the subscript vector NZSUBS and the subscript or nonzero index vector XNONZ. It also returns a number in MAXNZ which is the number of off-block-diagonal nonzeros in the matrix. Input to the subroutine is the adjacency structure of the graph (XADJ, ADJNCY), the quotient tree ordering (PERM, INVP), the quotient tree partitioning (NBLKS, XBLK), and the size of the array NZSUBS, contained in MAXNZ. The subroutine loops through the blocks in the partitioning. Within each block, the loop DO 200 J = ... is executed to consider each node in the block. Each neighbor belonging to an earlier block corresponds to an off-diagonal nonzero and it is added to the data structure. After the subscripts in a row have been determined, they are sorted (using SORTS1) into ascending sequence. The subroutine SORTS1 is straightforward and needs no explanation. A listing of it follows that of FNOFNZ.

Note that if the user does not provide a large enough subscript vector, the subroutine will detect this from the input parameter MAXNZ. It will continue to count the nonzeros, but will not store their column subscripts. Before returning, MAXNZ is set to the number of nonzeros found. Thus, the user should check that the value of MAXNZ has not been *increased* by the subroutine, as this indicates that not enough space in NZSUBS was provided.

---

```

1. C*****
2. C*****
3. C*****   FNOFNZ . . . . FIND OFF-BLOCK-DIAGONAL NONZEROS   ****
4. C*****
5. C*****
6. C
7. C   PURPOSE - THIS SUBROUTINE FINDS THE COLUMN SUBSCRIPTS OF
8. C   THE OFF-BLOCK-DIAGONAL NONZEROS IN THE LOWER TRIANGLE

```

6.5. A STORAGE SCHEME AND ALLOCATION PROCEDURE 241

```

 9. C      OF A PARTITIONED MATRIX.
10. C
11. C      INPUT PARAMETERS -
12. C      (XADJ, ADJNCY) - ADJACENCY STRUCTURE PAIR FOR THE GRAPH.
13. C      (PERM, INVP) - THE PERMUTATION VECTORS.
14. C      (NBLKS, XBLK) - THE BLOCK PARTITIONING.
15. C
16. C      OUTPUT PARAMETERS -
17. C      (XNONZ, NZSUBS) - THE COLUMN SUBSCRIPTS OF THE NONZEROS
18. C      OF A TO THE LEFT OF THE DIAGONAL BLOCKS ARE
19. C      STORED ROW BY ROW IN CONTINGUOUS LOCATIONS IN THE
20. C      ARRAY NZSUBS. XNONZ IS THE INDEX VECTOR TO IT.
21. C
22. C      UPDATED PARAMETER -
23. C      MAXNZ - ON INPUT, IT CONTAINS THE SIZE OF THE VECTOR
24. C      NZSUBS; AND ON OUTPUT, THE NUMBER OF NONZEROS
25. C      FOUND.
26. C
27. C*****
28. C
29. C      SUBROUTINE FNOFNZ ( XADJ, ADJNCY, PERM, INVP,
30. C      1          NBLKS, XBLK, XNONZ, NZSUBS, MAXNZ )
31. C
32. C*****
33. C
34. C      INTEGER ADJNCY(1), INVP(1), NZSUBS(1), PERM(1),
35. C      1          XBLK(1)
36. C      INTEGER XADJ(1), XNONZ(1), BLKBEG, BLKEND, I, J,
37. C      1          JPERM, JXNONZ, K, KSTOP, KSTRT, MAXNZ,
38. C      1          NABOR, NBLKS, NZCNT
39. C
40. C*****
41. C
42. C      NZCNT = 1
43. C      IF ( NBLKS .LE. 0 ) GO TO 400
44. C      -----
45. C      LOOP OVER THE BLOCKS ....
46. C      -----
47. C      DO 300 I = 1, NBLKS
48. C          BLKBEG = XBLK(I)
49. C          BLKEND = XBLK(I+1) - 1
50. C      -----
51. C      LOOP OVER THE ROWS OF THE I-TH BLOCK ...
52. C      -----
53. C      DO 200 J = BLKBEG, BLKEND
54. C          XNONZ(J) = NZCNT
55. C          JPERM = PERM(J)

```

```

56.           KSTRT = XADJ(JPERM)
57.           KSTOP = XADJ(JPERM+1) - 1
58.           IF ( KSTRT .GT. KSTOP ) GO TO 200
59. C         -----
60. C         LOOP OVER THE NONZEROS OF ROW J ...
61. C         -----
62.           DO 100 K = KSTRT, KSTOP
63.             NABOR = ADJNCY(K)
64.             NABOR = INVP(NABOR)
65. C         -----
66. C         CHECK TO SEE IF IT IS TO THE LEFT OF THE
67. C         I-TH DIAGONAL BLOCK.
68. C         -----
69.           IF ( NABOR .GE. BLKBEG ) GO TO 100
70.           IF ( NZCNT .LE. MAXNZ ) NZSUBS(NZCNT) = NABOR
71.           NZCNT = NZCNT + 1
72.     100    CONTINUE
73. C         -----
74. C         SORT THE SUBSCRIPTS OF ROW J
75. C         -----
76.           JXNONZ = XNONZ(J)
77.           IF ( NZCNT - 1 .LE. MAXNZ )
78.     1      CALL SORTS1 (NZCNT - JXNONZ, NZSUBS(JXNONZ))
79.     200    CONTINUE
80.     300    CONTINUE
81.           XNONZ(BLKEND+1) = NZCNT
82.     400    MAXNZ = NZCNT - 1
83.           RETURN
84.           END

```

---

```

1. C*****
2. C*****
3. C***** SORTS1 .... LINEAR INSERTION SORT *****
4. C*****
5. C*****
6. C
7. C   PURPOSE - SORTS1 USES LINEAR INSERTION TO SORT THE
8. C   GIVEN ARRAY OF SHORT INTEGERS INTO INCREASING ORDER.
9. C
10. C   INPUT PARAMETER -
11. C   NA - THE SIZE OF INTEGER ARRAY.
12. C
13. C   UPDATED PARAMETER -
14. C   ARRAY - THE INTEGER VECTOR, WHICH ON OUTPUT WILL BE
15. C   IN INCREASING ORDER.
16. C

```

```

17. C*****
18. C
19.     SUBROUTINE SORTS1 ( NA, ARRAY )
20. C
21. C*****
22. C
23.     INTEGER ARRAY(1)
24.     INTEGER K, L, NA, NODE
25. C
26. C*****
27. C
28.     IF (NA .LE. 1) RETURN
29.     DO 300 K = 2, NA
30.         NODE = ARRAY(K)
31.         L = K - 1
32. 100     IF (L .LT. 1) GO TO 200
33.         IF ( ARRAY(L) .LE. NODE ) GO TO 200
34.         ARRAY(L+1) = ARRAY(L)
35.         L = L - 1
36.         GO TO 100
37. 200     ARRAY(L+1) = NODE
38. 300     CONTINUE
39.     RETURN
40.     END

```

---

### FNTADJ (FiNd Tree ADJacency)

The purpose of this subroutine is to determine the adjacency structure of a given monotonely-ordered quotient tree. Recall from Section 6.3.2 that the structure of a monotonely ordered rooted tree is completely characterized by the *Father* function, where for a node  $x$ ,  $Father(x) = y$  means that  $y \in Adj(x)$  and that the (unique) path from the root to  $x$  goes through  $y$ . Our representation of the structure of our quotient tree is in a vector, called **FATHER**, of size  $p$ , where  $p$  is the number of blocks. Figure 6.5.6 contains the **FATHER** vector for the quotient tree ordering in Figure 6.5.5. Note that **FATHER**(  $p$  ) is always set to zero.

The subroutine **FNTADJ** accepts as input the adjacency structure of the graph (**XADJ**, **ADJNCY**), the quotient tree ordering (**PERM**, **INVP**), and the quotient tree partitioning (**NBLKS**, **XBLK**). It uses a working vector **BNUM** of size  $n$  to store the block numbers of the nodes in the partitioning.

The subroutine begins by setting up the **BNUM** vector for each node (loop **DO 200 K = . . .**). It then loops through each block in the partitioning in the

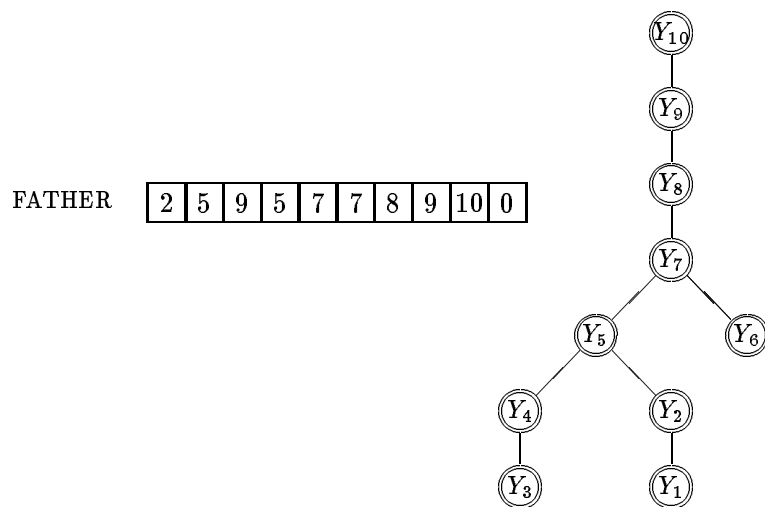


Figure 6.5.6: An example of the **FATHER** vector.

---

loop DO 600 K = ... to obtain its father block number. If it does not have any father block, the corresponding FATHER value is set to 0.

---

```

1. C*****
2. C*****
3. C*****      FNTADJ . . . . FIND TREE ADJACENCY      *****
4. C*****
5. C*****
6. C
7. C      PURPOSE - TO DETERMINE THE QUOTIENT TREE
8. C      ADJACENCY STRUCTURE OF A GRAPH. THE STRUCTURE IS
9. C      REPRESENTED BY THE FATHER VECTOR.
10. C
11. C      INPUT PARAMETERS -
12. C      (XADJ, ADJNCY) - ADJACENCY STRUCTURE PAIR FOR THE GRAPH.
13. C      (PERM, INVP) - THE PERMUTATION VECTORS.
14. C      (NBLKS, XBLK) - THE TREE PARTITIONING.
15. C
16. C      OUTPUT PARAMETERS -
17. C      FATHER - THE FATHER VECTOR OF THE QUOTIENT TREE.
18. C
19. C      WORKING PARAMETERS -
20. C      BNUM - TEMPORARY VECTOR TO STORE THE BLOCK NUMBER OF
21. C      OF EACH VARIABLE.
22. C
23. C*****
24. C
25. C      SUBROUTINE FNTADJ ( XADJ, ADJNCY, PERM, INVP,
26. C      1          NBLKS, XBLK, FATHER, BNUM )
27. C
28. C*****
29. C
30. C      INTEGER ADJNCY(1), BNUM(1), FATHER(1), INVP(1),
31. C      1          PERM(1), XBLK(1)
32. C      INTEGER XADJ(1), I, ISTOP, ISTRT, J, JSTOP, JSTRT,
33. C      1          K, NABOR, NBLKS, NBM1, NBRBLK, NODE
34. C
35. C*****
36. C
37. C      -----
38. C      INITIALIZE THE BLOCK NUMBER VECTOR.
39. C      -----
40. C      DO 200 K = 1, NBLKS
41. C          ISTRT = XBLK(K)
42. C          ISTOP = XBLK(K+1) - 1
43. C          DO 100 I = ISTRT, ISTOP
44. C              NODE = PERM(I)

```

```

45 .           BNUM(NODE) = K
46 .   100     CONTINUE
47 .   200     CONTINUE
48 . C       -----
49 . C       FOR EACH BLOCK ...
50 . C       -----
51 .         FATHER(NBLKS) = 0
52 .         NBM1 = NBLKS - 1
53 .         IF ( NBM1 .LE. 0 ) RETURN
54 .         DO 600 K = 1, NBM1
55 .           ISTRT = XBLK(K)
56 .           ISTOP = XBLK(K+1) - 1
57 . C       -----
58 . C       FIND ITS FATHER BLOCK IN THE TREE STRUCTURE.
59 . C       -----
60 .         DO 400 I = ISTRT, ISTOP
61 .           NODE = PERM(I)
62 .           JSTRT = XADJ(NODE)
63 .           JSTOP = XADJ(NODE+1) -1
64 .           IF ( JSTOP .LT. JSTRT ) GO TO 400
65 .           DO 300 J = JSTRT, JSTOP
66 .             NABOR = ADJNCY(J)
67 .             NBRBLK = BNUM(NABOR)
68 .             IF ( NBRBLK .GT. K ) GO TO 500
69 .   300     CONTINUE
70 .   400     CONTINUE
71 .           FATHER(K) = 0
72 .           GO TO 600
73 .   500     FATHER(K) = NBRBLK
74 .   600     CONTINUE
75 .         RETURN
76 .       END

```

---

## 6.6 The Numerical Subroutines TSFCT (Tree Symmetric FaCTorization) and TSSLV (Tree Symmetric SoLVe)

In this section, we describe the subroutines that implement the numerical factorization and solution for partitioned linear systems associated with quotient trees, stored in the sparse scheme as introduced in Section 6.5.1. The subroutine **TSFCT** employs the asymmetric version of the factorization, so we begin by first re-examining the asymmetric block factorization procedure of Section 6.2.1 and studying possible improvements.

### 6.6.1 Computing the Block Modification Matrix

Let the matrix  $\mathbf{A}$  be partitioned into

$$\begin{pmatrix} \mathbf{B} & \mathbf{V} \\ \mathbf{V}^T & \bar{\mathbf{C}} \end{pmatrix}$$

as in Section 6.2.1. Recall that in the factorization scheme, the modification matrix  $\mathbf{V}^T \mathbf{B}^{-1} \mathbf{V}$  used to form  $\mathbf{C} = \bar{\mathbf{C}} - \mathbf{V}^T \mathbf{B}^{-1} \mathbf{V}$  is obtained as follows.

$$\mathbf{V}^T (\mathbf{L}_B^{-T} (\mathbf{L}_B^{-1} \mathbf{V})) = \mathbf{V}^T (\mathbf{L}_B^{-T} \mathbf{W}) = \mathbf{V}^T \tilde{\mathbf{W}}.$$

Note also that the modification matrix  $\mathbf{V}^T \tilde{\mathbf{W}}$  is symmetric and that  $\text{Nonz}(\mathbf{V}) \subset \text{Nonz}(\mathbf{W})$ . We now investigate an efficient way to compute  $\mathbf{V}^T \tilde{\mathbf{W}}$ .

Let  $\mathbf{G}$  be an  $r$  by  $s$  sparse matrix and  $\mathbf{H}$  be an  $s$  by  $r$  matrix. For the  $i$ -th row of  $\mathbf{G}$ , let

$$f_i(\mathbf{G}) = \min\{j \mid g_{ij} \neq 0\}, \quad 1 \leq i \leq r. \quad (6.6.1)$$

That is,  $f_i(\mathbf{G})$  is the column subscript of the first nonzero component in row  $i$  of  $\mathbf{G}$ . Assume that the matrix product  $\mathbf{GH}$  is *symmetric*. In what follows, we show that only a portion of the matrix  $\mathbf{H}$  is needed in computing the product. Figure 6.6.1 contains an example with  $r = 4$  and  $s = 8$ . If the product  $\mathbf{GH}$  is symmetric, the next lemma says that the crosshatched part of  $\mathbf{H}$  can be ignored in the evaluation of  $\mathbf{GH}$ .

**Lemma 6.6.1** *If the matrix product  $\mathbf{GH}$  is symmetric, the product is completely determined by the matrix  $\mathbf{G}$  and the matrix subset*

$$\{h_{jk} \mid f_k(\mathbf{G}) \leq j \leq s\}$$

*of  $\mathbf{H}$ .*

**Proof:** It is sufficient to show that every entry in the matrix product can be computed from  $\mathbf{G}$  and the given subset of  $\mathbf{H}$ . Since the product is symmetric, its  $(i, k)$ -th and  $(k, i)$ -th entries are given by

$$\sum_{j=f_i(\mathbf{G})}^s g_{ij} h_{jk}$$

or

$$\sum_{j=f_k(\mathbf{G})}^s g_{kj} h_{ji}.$$



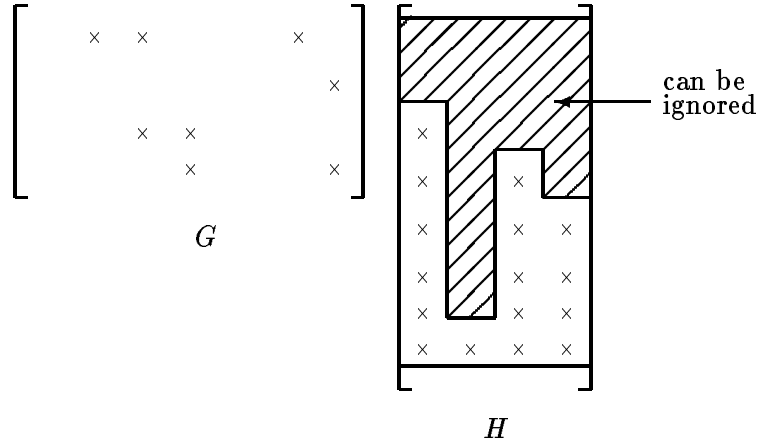


Figure 6.6.1: Sparse symmetric matrix product.

For definiteness, let  $f_k(\mathbf{G}) \leq f_i(\mathbf{G})$ . The entry can then be obtained using the first expression, which involves components in  $\mathbf{G}$  and those  $h_{jk}$  with  $f_k(\mathbf{G}) \leq f_i(\mathbf{G}) \leq j \leq s$ . They belong to the given matrix subset. On the other hand, if  $f_k(\mathbf{G}) > f_i(\mathbf{G})$ , the second expression can be used which involves matrix components in the subset. This proves the lemma.  $\square$

The order in which the components in the product are computed depends on the structure of the matrix (or more specifically, on the column subscripts  $f_i(\mathbf{G})$ ,  $1 \leq i \leq r$ ). For example, in forming  $\mathbf{GH}$  for the matrices in Figure 6.6.1, the order of computation is given in Figure 6.6.2.

With this framework, we can study changing the submatrix  $\bar{\mathbf{C}}$  into  $\mathbf{C} = \bar{\mathbf{C}} - \mathbf{V}^T \mathbf{B}^{-1} \mathbf{V} = \bar{\mathbf{C}} - \mathbf{V}^T (\mathbf{L}_B^{-T} (\mathbf{L}_B^{-1} \mathbf{V}))$ . As pointed out in Section 6.2.1, the modification can be carried out one column at a time as follows:

- 1) Unpack a column  $\mathbf{v} = \mathbf{V}_{*i}$  of  $\mathbf{V}$ .
- 2) Solve  $\mathbf{B}\tilde{\mathbf{w}} = \mathbf{v}$  by solving the triangular systems  $\mathbf{L}_B \mathbf{w} = \mathbf{v}$  and  $\mathbf{L}_B^T \tilde{\mathbf{w}} = \mathbf{w}$ .
- 3) Compute the vector  $\mathbf{z} = \mathbf{V}^T \tilde{\mathbf{w}}$  and set  $\mathbf{C}_{*i} = \bar{\mathbf{C}}_{*i} - \mathbf{z}$ .

Now since  $\mathbf{V}^T \tilde{\mathbf{W}}$  is symmetric, Lemma 6.6.1 applies to this modification process, and it is unnecessary to compute the entire vector  $\tilde{\mathbf{w}}$  from  $\mathbf{v}$  in

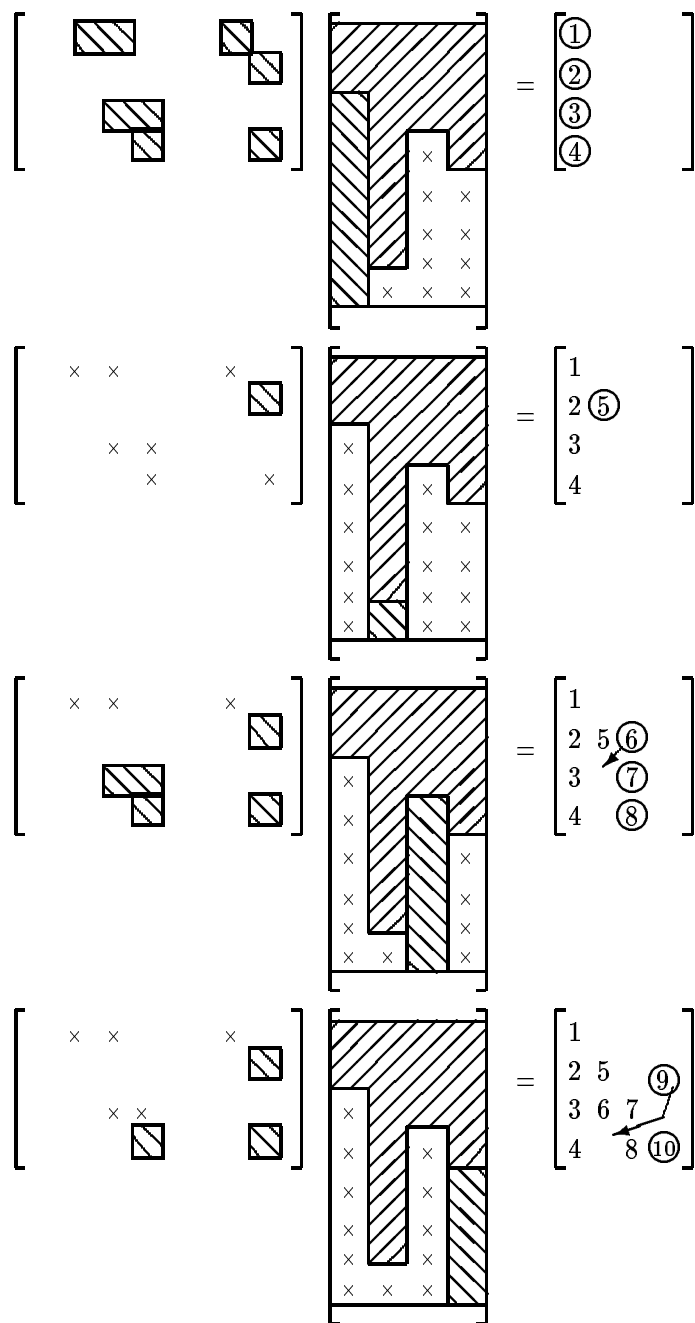


Figure 6.6.2: Illustration of the computation of the product  $GH$ .

Step 2. The components in  $\tilde{w}$  above the first nonzero subscript of  $v$  do not have to be computed when solving  $L_B(L_B^T \tilde{w}) = v$ .

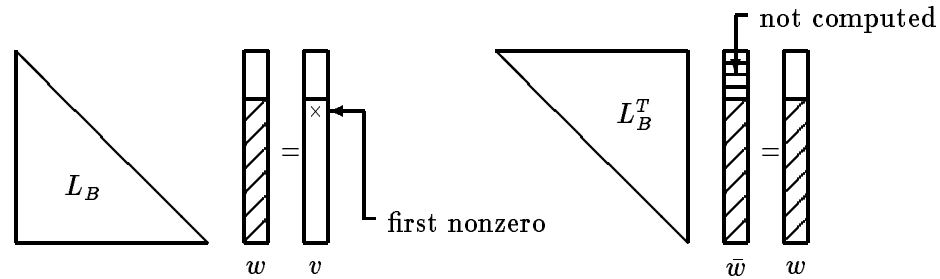


Figure 6.6.3: Illustration of computing the modifications.

In effect, a smaller system than  $L_B(L_B^T \tilde{w}) = v$  needs to be solved. This can have a significant effect on the amount of computation required for the factorization. For example, see Exercise 6.6.1 on page 263.

### 6.6.2 The Subroutine TSFCT (Tree Symmetric FaCTORization)

The subroutine TSFCT performs the asymmetric block factorization for tree-partitioned systems. The way it computes the block modification matrix is as described in the previous section.

The subroutine accepts as input the tree partitioning information in (NBLKS, XBLK) and FATHER, the data structure in XENV, XNONZ and NZSUBS, and the primary storage vectors DIAG, ENV and NONZ. The vectors DIAG and ENV, on input, contain the nonzeros of the block diagonals of the matrix  $A$ . On return, the corresponding nonzeros of the block diagonals of  $L$  are overwritten on those of  $A$ . Since the implicit scheme is used, the off-block-diagonal nonzeros of  $A$  stored in NONZ remain unchanged.

Two temporary vectors of size  $n$  are used. The real vector TEMP is used for unpacking off-diagonal block columns so that numerical solution on the unpacked column can be done in the vector TEMP. The second temporary vector FIRST is an integer array used to facilitate indexing into the subscript vector NZSUBS. (See remarks about FIRST below.)

The subroutine TSFCT begins by initializing the temporary vectors TEMP and FIRST (loop DO 100 I = ...). The main loop DO 1600 K = ... is then executed for each block in the partitioning. Within the main loop, the subroutine ESFCT is first called to factor the K-th diagonal block. The next step

is to find out where the off-diagonal block is, and it is given by `FATHER(K)`. The loops `D0 200 ...` and `D0 400 ...` are then executed to determine the first and last non-null columns respectively in the off-diagonal block so that modification can be performed within these columns. Figure 6.6.4 depicts the role of some of the important local variables in the subroutine.

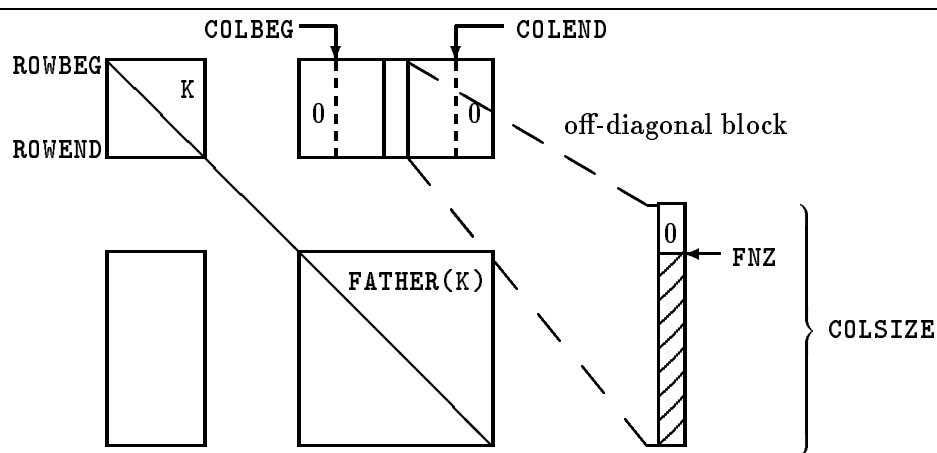


Figure 6.6.4: Illustration of some of the important local variables used in TSFCT.

The loop `D0 1300 COL = ...` applies the modification to the diagonal block given by `FATHER(K)`. Each column in the off-diagonal block is unpacked into the vector `TEMP` (loop `D0 600 J = ...`), after which the envelope solvers `ELSLV` and `EUSLV` are invoked. The inner loop `D0 1100 COL1 = ...` then performs the modification in the same manner as discussed in Section 6.6.1. Before the subroutine proceeds to consider the next block, it updates the temporary vector `FIRST` for columns in the `FATHER(K)`-th block, so that the corresponding elements of `FIRST` point to the next numbers to be used in those columns (loop `D0 1500 COL = ...`). When all the diagonal blocks have been processed, the subroutine returns.

```

1. C*****
2. C*****
3. C*****      TSFCT . . . . TREE SYMMETRIC FACTORIZATION *****
4. C*****
5. C*****
6. C
7. C      PURPOSE - THIS SUBROUTINE PERFORMS THE SYMMETRIC

```

```

8. C          FACTORIZATION OF A TREE-PARTITIONED SYSTEM.
9. C
10. C        INPUT PARAMETERS -
11. C          (NBLKS, XBLK, FATHER) - THE TREE PARTITIONING.
12. C          XENV - THE ENVELOPE INDEX VECTOR.
13. C          (XNONZ, NONZ, NZSUBS) - THE OFF-DIAGONAL NONZEROS IN
14. C            THE ORIGINAL MATRIX.
15. C
16. C        UPDATED PARAMETERS -
17. C          (DIAG, ENV) - STORAGE ARRAYS FOR THE ENVELOPE OF
18. C            THE DIAGONAL BLOCKS OF THE MATRIX. ON OUTPUT,
19. C            CONTAINS THE DIAGONAL BLOCKS OF THE FACTOR.
20. C          IFLAG - THE ERROR FLAG. IT IS SET TO 1 IF A ZERO OR
21. C            NEGATIVE SQUARE ROOT IS DETECTED DURING THE
22. C            FACTORIZATION.
23. C
24. C        WORKING PARAMETER -
25. C          TEMP - TEMPORARY ARRAY REQUIRED TO IMPLEMENT THE
26. C            ASYMMETRIC VERSION OF THE FACTORIZATION.
27. C          FIRST - TEMPORARY VECTOR USED TO FACILITATE THE
28. C            INDEXING TO THE VECTOR NONZ (OR NZSUBS)
29. C            FOR NON-NULL SUBCOLUMNS IN OFF-DIAGONAL
30. C            BLOCKS.
31. C
32. C        PROGRAM SUBROUTINES -
33. C          ESFCT, ELSLV, EUSLV.
34. C
35. C *****
36. C
37. C          SUBROUTINE TSFCT ( NBLKS, XBLK, FATHER, DIAG, XENV, ENV,
38. C            1              XNONZ, NONZ, NZSUBS, TEMP, FIRST, IFLAG )
39. C
40. C *****
41. C
42. C          DOUBLE PRECISION OPS
43. C          COMMON /SPKOPS/ OPS
44. C          REAL DIAG(1), ENV(1), NONZ(1), TEMP(1), S
45. C          INTEGER FATHER(1), NZSUBS(1), XBLK(1)
46. C          INTEGER FIRST(1), XENV(1), XNONZ(1),
47. C            1          BLKSZE, COL, COL1, COLBEG, COLEND,
48. C            1          COLSZE, FNZ, FNZ1, I, IFLAG, ISTRT, ISTOP,
49. C            1          ISUB, J, JSTOP, JSTRT, K, KENV, KENVO, KFATHR,
50. C            1          NBLKS, NEQNS, ROW, ROWBEG, ROWEND
51. C
52. C *****
53. C
54. C          -----

```

```

55. C      INITIALIZATION ...
56. C      -----
57.      NEQNS = XBLK(NBLKS+1) - 1
58.      DO 100 I = 1,NEQNS
59.          TEMP(I) = 0.0E0
60.          FIRST(I) = XNONZ(I)
61.      100  CONTINUE
62. C      -----
63. C      LOOP THROUGH THE BLOCKS ...
64. C      -----
65.      DO 1600 K = 1, NBLKS
66.          ROWBEG = XBLK(K)
67.          ROWEND = XBLK(K+1) - 1
68.          BLKSZE = ROWEND - ROWBEG + 1
69.          CALL ESFCT ( BLKSZE, XENV(ROWBEG), ENV,
70.              1      DIAG(ROWBEG), IFLAG )
71.          IF ( IFLAG .GT. 0 ) RETURN
72. C      -----
73. C      PERFORM MODIFICATION OF THE FATHER DIAGONAL BLOCK
74. C      A(FATHER(K),FATHER(K)) FROM THE OFF-DIAGONAL BLOCK
75. C      A(K,FATHER(K)).
76. C      -----
77.      KFATHR = FATHER(K)
78.      IF ( KFATHR .LE. 0 ) GO TO 1600
79.      COLBEG = XBLK(KFATHR)
80.      COLEND = XBLK(KFATHR+1) - 1
81. C      -----
82. C      FIND THE FIRST AND LAST NON-NULL COLUMN IN
83. C      THE OFF-DIAGONAL BLOCK. RESET COLBEG,COLEND.
84. C      -----
85.      DO 200 COL = COLBEG, COLEND
86.          JSTRT = FIRST(COL)
87.          JSTOP = XNONZ(COL+1) - 1
88.          IF ( JSTOP .GE. JSTRT .AND.
89.              1      NZSUBS(JSTRT) .LE. ROWEND ) GO TO 300
90.      200  CONTINUE
91.      300  COLBEG = COL
92.          COL = COLEND
93.          DO 400 COL1 = COLBEG, COLEND
94.              JSTRT = FIRST(COL)
95.              JSTOP = XNONZ(COL+1) - 1
96.              IF ( JSTOP .GE. JSTRT .AND.
97.                  1      NZSUBS(JSTRT) .LE. ROWEND ) GO TO 500
98.          COL = COL - 1
99.      400  CONTINUE
100.     500  COLEND = COL
101.     DO 1300 COL = COLBEG, COLEND

```

```

102.          JSTRT = FIRST(COL)
103.          JSTOP = XNONZ(COL+1) - 1
104. C -----
105. C TEST FOR NULL SUBCOLUMN. FNZ STORES THE
106. C FIRST NONZERO SUBSCRIPT IN THE BLOCK COLUMN.
107. C -----
108.          IF ( JSTOP .LT. JSTRT ) GO TO 1300
109.          FNZ = NZSUBS(JSTRT)
110.          IF ( FNZ .GT. ROWEND ) GO TO 1300
111. C -----
112. C UNPACK A COLUMN IN THE OFF-DIAGONAL BLOCK
113. C AND PERFORM UPPER AND LOWER SOLVES ON THE
114. C UNPACKED COLUMN.
115. C -----
116.          DO 600 J = JSTRT, JSTOP
117.             ROW = NZSUBS(J)
118.             IF ( ROW .GT. ROWEND ) GO TO 700
119.             TEMP(ROW) = NONZ(J)
120.          600 CONTINUE
121.          700 COLSZ = ROWEND - FNZ + 1
122.          CALL ELSLV ( COLSZ, XENV(FNZ), ENV,
123.                    1     DIAG(FNZ), TEMP(FNZ) )
124.          CALL EUSLV ( COLSZ, XENV(FNZ), ENV,
125.                    1     DIAG(FNZ), TEMP(FNZ) )
126. C -----
127. C DO THE MODIFICATION BY LOOPING THROUGH
128. C THE COLUMNS AND FORMING INNER PRODUCTS.
129. C -----
130.          KENVO = XENV(COL+1) - COL
131.          DO 1100 COL1= COLBEG, COLEND
132.             ISTRT = FIRST(COL1)
133.             ISTOP = XNONZ(COL1+1) - 1
134. C -----
135. C CHECK TO SEE IF SUBCOLUMN IS NULL.
136. C -----
137.          FNZ1 = NZSUBS(ISTRT)
138.          IF ( ISTOP .LT. ISTRT .OR.
139.            1     FNZ1 .GT. ROWEND ) GO TO 1100
140. C -----
141. C CHECK IF INNER PRODUCT SHOULD BE DONE.
142. C -----
143.          IF ( FNZ1 .LT. FNZ ) GO TO 1100
144.          IF ( FNZ1 .EQ. FNZ .AND.
145.            1     COL1 .LT. COL ) GO TO 1100
146.          S = 0.0E0
147.          DO 800 I = ISTRT, ISTOP
148.             ISUB = NZSUBS(I)

```

```

149.             IF ( ISUB .GT. ROWEND ) GO TO 900
150.             S = S + TEMP(ISUB) * NONZ(I)
151.             OPS = OPS + 1.0D0
152.      800      CONTINUE
153.      C      -----
154.      C      MODIFY THE ENV OR THE DIAG ENTRY.
155.      C      -----
156.      900      IF ( COL1 .EQ. COL ) GO TO 1000
157.             KENV = KENVO + COL1
158.             IF ( COL1 .GT. COL )
159.      1        KENV = XENV(COL1+1) - COL1 + COL
160.             ENV(KENV) = ENV(KENV) - S
161.             GO TO 1100
162.      1000     DIAG(COL1) = DIAG(COL1) - S
163.      1100     CONTINUE
164.      C      -----
165.      C      RESET PART OF THE TEMP VECTOR TO ZERO.
166.      C      -----
167.             DO 1200 ROW = FNZ, ROWEND
168.             TEMP(ROW) = 0.0E0
169.      1200     CONTINUE
170.      1300     CONTINUE
171.      C      -----
172.      C      UPDATE THE FIRST VECTOR FOR COLUMNS IN
173.      C      FATHER(K) BLOCK, SO THAT IT WILL INDEX TO
174.      C      THE BEGINNING OF THE NEXT OFF-DIAGONAL
175.      C      BLOCK TO BE CONSIDERED.
176.      C      -----
177.             DO 1500 COL = COLBEG, COLEND
178.             JSTRT = FIRST(COL)
179.             JSTOP = XNONZ(COL+1) - 1
180.             IF ( JSTOP .LT. JSTRT ) GO TO 1500
181.             DO 1400 J = JSTRT, JSTOP
182.             ROW = NZSUBS(J)
183.             IF ( ROW .LE. ROWEND ) GO TO 1400
184.             FIRST(COL) = J
185.             GO TO 1500
186.      1400     CONTINUE
187.             FIRST(COL) = JSTOP + 1
188.      1500     CONTINUE
189.      1600     CONTINUE
190.             RETURN
191.             END

```

---



### 6.6.3 The Subroutine TSSLV (Tree Symmetric SoLve)

The implementation of the solver for tree-partitioned linear systems does not follow the same execution sequence as specified in Section 6.3.3. Instead, it uses the alternative decomposition for the asymmetric factorization as given in Exercise 6.2.2 on page 196:

$$\mathbf{A} = \begin{pmatrix} \mathbf{B} & \mathbf{V} \\ \mathbf{V}^T & \bar{\mathbf{C}} \end{pmatrix} = \begin{pmatrix} \mathbf{B} & \mathbf{O} \\ \mathbf{V}^T & \mathbf{C} \end{pmatrix} \begin{pmatrix} \mathbf{I} & \tilde{\mathbf{W}} \\ \mathbf{O} & \mathbf{I} \end{pmatrix}, \quad (6.6.2)$$

where  $\tilde{\mathbf{W}} = \mathbf{B}^{-1}\mathbf{V}$  is not explicitly stored, and  $\mathbf{C} = \bar{\mathbf{C}} - \mathbf{V}^T\mathbf{B}^{-1}\mathbf{V}$ . Written in this form, the solution to

$$\begin{pmatrix} \mathbf{B} & \mathbf{V} \\ \mathbf{V}^T & \bar{\mathbf{C}} \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{pmatrix}$$

can be computed by solving

$$\begin{pmatrix} \mathbf{B} & \mathbf{O} \\ \mathbf{V}^T & \mathbf{C} \end{pmatrix} \begin{pmatrix} \mathbf{z}_1 \\ \mathbf{z}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{pmatrix}$$

and

$$\begin{pmatrix} \mathbf{I} & \tilde{\mathbf{W}} \\ \mathbf{O} & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{z}_1 \\ \mathbf{z}_2 \end{pmatrix}.$$

It is assumed that the submatrices  $\mathbf{B}$  and  $\mathbf{C}$  have been factored into  $\mathbf{L}_B\mathbf{L}_B^T$  and  $\mathbf{L}_C\mathbf{L}_C^T$  respectively. The scheme can hence be written as follows.

#### *Forward Solve*

Solve  $\mathbf{L}_B(\mathbf{L}_B^T\mathbf{z}_1) = \mathbf{b}_1$ .

Compute  $\tilde{\mathbf{b}}_2 = \mathbf{b}_2 - \mathbf{V}^T\mathbf{z}_1$ .

Solve  $\mathbf{L}_C(\mathbf{L}_C^T\mathbf{z}_2) = \tilde{\mathbf{b}}_2$ .

#### *Backward Solve*

Assign  $\mathbf{x}_2 = \mathbf{z}_2$ .

Compute  $\mathbf{t}_1 = \mathbf{V}\mathbf{x}_2$ .

Solve  $\mathbf{L}_B(\mathbf{L}_B^T\tilde{\mathbf{t}}_1) = \mathbf{t}_1$ .

Compute  $\mathbf{x}_1 = \mathbf{z}_1 - \tilde{\mathbf{t}}_1$ .

This scheme is simply a rearrangement of the operation sequences as given in Section 6.2.2. The only difference is that no temporary vector is required in the forward solve (no real advantage though! Why?). We choose to use this scheme because it simplifies the program organization when the general block  $p$  by  $p$  tree-partitioned system is being solved.

We now consider the generalization of the above asymmetric scheme. Let  $\mathbf{A}$  be a  $p$  by  $p$  tree-partitioned matrix with blocks  $\mathbf{A}_{ij}$ ,  $1 \leq i, j \leq p$ . Let  $\mathbf{L}_{ij}$  be the corresponding submatrices of the triangular factor  $\mathbf{L}$  of  $\mathbf{A}$ .

Recall from Section 6.3.2, that since  $\mathbf{A}$  is tree-partitioned, the lower off-diagonal blocks  $\mathbf{L}_{ij}$  ( $i > j$ ) are given by

$$\mathbf{L}_{ij} = \mathbf{A}_{ij} \mathbf{L}_{jj}^{-T}. \quad (6.6.3)$$

We want to define an asymmetric block factorization

$$\mathbf{A} = \tilde{\mathbf{L}} \mathbf{U} \quad (6.6.4)$$

similar to that of (6.6.2). Obviously the factor  $\tilde{\mathbf{L}}$  is well defined and its blocks are given by,

$$\tilde{\mathbf{L}}_{ij} = \begin{cases} \mathbf{L}_{ii} \mathbf{L}_{ii}^T & \text{if } i = j \\ \mathbf{A}_{ij} & \text{if } i > j \\ \mathbf{O} & \text{otherwise.} \end{cases}$$

The case when  $p = 4$  is given below.

$$\tilde{\mathbf{L}} = \begin{pmatrix} \mathbf{L}_{11} \mathbf{L}_{11}^T & \mathbf{O} & \mathbf{O} & \mathbf{O} \\ \mathbf{A}_{21} & \mathbf{L}_{22} \mathbf{L}_{22}^T & \mathbf{O} & \mathbf{O} \\ \mathbf{A}_{31} & \mathbf{A}_{32} & \mathbf{L}_{33} \mathbf{L}_{33}^T & \mathbf{O} \\ \mathbf{A}_{41} & \mathbf{A}_{42} & \mathbf{A}_{43} & \mathbf{L}_{44} \mathbf{L}_{44}^T \end{pmatrix}$$

**Lemma 6.6.2**

$$\tilde{\mathbf{L}} = \mathbf{L} \begin{pmatrix} \mathbf{L}_{11}^T & & & \mathbf{O} \\ & \mathbf{L}_{22}^T & & \\ & & \ddots & \\ \mathbf{O} & & & \mathbf{L}_{pp}^T \end{pmatrix}$$

**Proof:** The result follows directly from the relation (6.6.3) between off-diagonal blocks  $\mathbf{A}_{ij}$  and  $\mathbf{L}_{ij}$  for tree-partitioned systems.  $\square$

By this lemma, the upper triangular factor  $U$  in (6.6.4) can then be obtained simply as

$$U = \begin{pmatrix} \mathbf{L}_{11}^T & & & \mathbf{O} \\ & \mathbf{L}_{22}^T & & \\ & & \ddots & \\ \mathbf{O} & & & \mathbf{L}_{pp}^T \end{pmatrix}^{-1} \mathbf{L}^T$$

so that we have

$$U_{ik} = \begin{cases} \mathbf{I} & \text{if } i = k \\ \mathbf{L}_{ii}^{-T} \mathbf{L}_{ki}^T & \text{if } i < k \\ \mathbf{O} & \text{otherwise.} \end{cases}$$

and for  $i < k$ , the expression can be simplified by (6.6.3) to

$$\begin{aligned} U_{ik} &= \mathbf{L}_{ii}^{-T} (\mathbf{A}_{ki} \mathbf{L}_{ii}^{-T})^T \\ &= \mathbf{L}_{ii}^{-T} \mathbf{L}_{ii}^{-1} \mathbf{A}_{ki}^T \\ &= (\mathbf{L}_{ii} \mathbf{L}_{ii}^T)^{-1} \mathbf{A}_{ik}. \end{aligned}$$

Therefore, the asymmetric factorization (6.6.4) for the case  $p = 4$  can be expressed explicitly as shown by the following:

$$\begin{aligned} \tilde{\mathbf{L}} &= \begin{pmatrix} \mathbf{L}_{11} \mathbf{L}_{11}^T & \mathbf{O} & \mathbf{O} & \mathbf{O} \\ \mathbf{A}_{21} & \mathbf{L}_{22} \mathbf{L}_{22}^T & \mathbf{O} & \mathbf{O} \\ \mathbf{A}_{31} & \mathbf{A}_{32} & \mathbf{L}_{33} \mathbf{L}_{33}^T & \mathbf{O} \\ \mathbf{A}_{41} & \mathbf{A}_{42} & \mathbf{A}_{43} & \mathbf{L}_{44} \mathbf{L}_{44}^T \end{pmatrix} \\ U &= \begin{pmatrix} \mathbf{I} & (\mathbf{L}_{11} \mathbf{L}_{11}^T)^{-1} \mathbf{A}_{12} & (\mathbf{L}_{11} \mathbf{L}_{11}^T)^{-1} \mathbf{A}_{13} & (\mathbf{L}_{11} \mathbf{L}_{11}^T)^{-1} \mathbf{A}_{14} \\ \mathbf{O} & \mathbf{I} & (\mathbf{L}_{22} \mathbf{L}_{22}^T)^{-1} \mathbf{A}_{23} & (\mathbf{L}_{22} \mathbf{L}_{22}^T)^{-1} \mathbf{A}_{24} \\ \mathbf{O} & \mathbf{O} & \mathbf{I} & (\mathbf{L}_{33} \mathbf{L}_{33}^T)^{-1} \mathbf{A}_{34} \\ \mathbf{O} & \mathbf{O} & \mathbf{O} & \mathbf{I} \end{pmatrix}. \end{aligned}$$

To consider the actual solution phase on this factorization, we have to relate it to our block storage scheme as described in Section 6.5.1. As before, let

$$\mathbf{V}_k = \begin{pmatrix} \mathbf{A}_{1k} \\ \mathbf{A}_{2k} \\ \vdots \\ \mathbf{A}_{k-1,k} \end{pmatrix}, \quad 2 \leq k \leq p.$$

Since the nonzero components outside the diagonal blocks are stored column by column as in

$$\mathbf{V}_2, \mathbf{V}_3, \dots, \mathbf{V}_p,$$

the solution method must be tailored to this storage scheme. The method to be discussed makes use of the observation that

$$\begin{pmatrix} \mathbf{U}_{1k} \\ \mathbf{U}_{2k} \\ \vdots \\ \mathbf{U}_{k-1,k} \end{pmatrix} = \begin{pmatrix} (\mathbf{L}_{11}\mathbf{L}_{11}^T)^{-1} & & & \mathbf{O} \\ & (\mathbf{L}_{22}\mathbf{L}_{22}^T)^{-1} & & \\ & & \ddots & \\ \mathbf{O} & & & (\mathbf{L}_{k-1,k-1}\mathbf{L}_{k-1,k-1}^T)^{-1} \end{pmatrix} \mathbf{V}_k.$$

*Forward Solve*  $\tilde{\mathbf{L}}\mathbf{z} = \mathbf{b}$

**Step 1** Solve  $(\mathbf{L}_{11}\mathbf{L}_{11}^T)\mathbf{z}_1 = \mathbf{b}_1$ .

**Step 2** For  $k = 2, \dots, p$  do the following

2.1) Compute

$$\mathbf{b}_k \leftarrow \mathbf{b}_k - \mathbf{V}_k^T \begin{pmatrix} \mathbf{z}_1 \\ \vdots \\ \mathbf{z}_{k-1} \end{pmatrix}$$

2.2) Solve  $(\mathbf{L}_{kk}\mathbf{L}_{kk}^T)\mathbf{z}_k = \mathbf{b}_k$ .

*Backward Solve*  $\mathbf{U}\mathbf{x} = \mathbf{z}$

**Step 1** Initialize temporary vector  $\tilde{\mathbf{z}} = \mathbf{0}$ ,

**Step 2**  $\mathbf{x}_p = \mathbf{z}_p$

**Step 3** For  $k = p-1, p-2, \dots, 1$  do the following

3.1)

$$\begin{pmatrix} \tilde{\mathbf{z}}_1 \\ \vdots \\ \tilde{\mathbf{z}}_k \end{pmatrix} \leftarrow \begin{pmatrix} \tilde{\mathbf{z}}_1 \\ \vdots \\ \tilde{\mathbf{z}}_k \end{pmatrix} + \mathbf{V}_{k+1}\mathbf{x}_{k+1}$$

3.2) Solve  $(\mathbf{L}_{kk}\mathbf{L}_{kk}^T)\tilde{\mathbf{x}}_k = \tilde{\mathbf{z}}_k$ .

3.3) Compute  $\mathbf{x}_k = \mathbf{z}_k - \tilde{\mathbf{x}}_k$ .

Note that in the back solve, a temporary vector  $\tilde{z}$  is used to accumulate the products and its use is illustrated in Figure 6.6.5.

The subroutine TSSLV implements this solution scheme. Unlike TSFCT, it does not require the FATHER vector of the tree partitioning, although implicitly it depends on it. Inputs to TSSLV are the tree partitioning (nBLKS, XBLK), the diagonal DIAG, the envelope (XENV, ENV) of the diagonal blocks, and the off-diagonal nonzeros in (XNONZ, NONZ, NZSUBS).

There are two main loops in the subroutine TSSLV; one to perform the forward substitution and the other to do the backward solve. In the forward solve, the loop DO 200 ROW = ... is executed to modify the right hand vector before the subroutines ELSLV and EUSLV are called. In the backward solve, the temporary real vector TEMP accumulates the products of off-diagonal blocks and parts of the solution, in preparation for calling ELSLV and EUSLV. At the end, the vector RHS contains the solution vector.

---

```

1. C*****
2. C*****
3. C***** TSSLV .... TREE SYMMETRIC SOLVE *****
4. C*****
5. C*****
6. C
7. C PURPOSE - TO PERFORM SOLUTION OF A TREE-PARTITIONED
8. C FACTORED SYSTEM BY IMPLICIT BACK SUBSTITUTION.
9. C
10. C INPUT PARAMETERS -
11. C (NBLKS, XBLK) - THE PARTITIONING.
12. C (XENV, ENV) - ENVELOPE OF THE DIAGONAL BLOCKS.
13. C (XNONZ, NONZ, NZSUBS) - DATA STRUCTURE FOR THE OFF-
14. C BLOCK DIAGONAL NONZEROS.
15. C
16. C UPDATED PARAMETERS -
17. C RHS - ON INPUT IT CONTAINS THE RIGHT HAND VECTOR.
18. C ON OUTPUT, THE SOLUTION VECTOR.
19. C
20. C WORKING VECTOR -
21. C TEMP - TEMPORARY VECTOR USED IN BACK SUBSTITUTION.
22. C
23. C PROGRAM SUBROUTINES -
24. C ELSLV, EUSLV.
25. C
26. C*****
27. C
28. SUBROUTINE TSSLV ( NBLKS, XBLK, DIAG, XENV, ENV,
29. 1 XNONZ, NONZ, NZSUBS, RHS, TEMP )

```

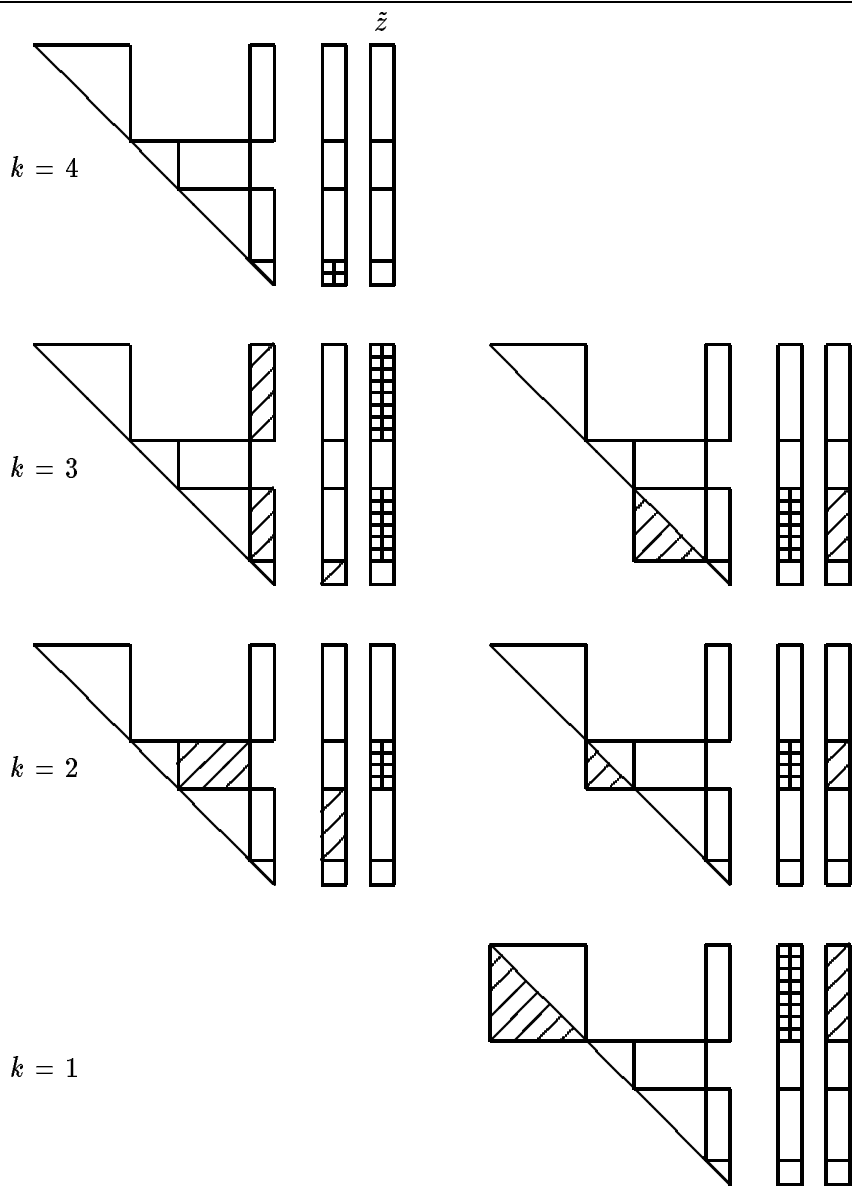


Figure 6.6.5: Backward solve for asymmetric factorization.

---

```

30. C
31. C*****
32. C
33.     DOUBLE PRECISION COUNT, OPS
34.     COMMON /SPKOPS/ OPS
35.     REAL DIAG(1), ENV(1), NONZ(1), RHS(1), TEMP(1), S
36.     INTEGER NZSUBS(1), XBLK(1)
37.     INTEGER XENV(1), XNONZ(1), COL, COL1, COL2, I, J,
38.     1         JSTOP, JSTRT, LAST, NBLKS, NCOL, NROW, ROW,
39.     1         ROW1, ROW2
40. C
41. C*****
42. C
43. C -----
44. C FORWARD SUBSTITUTION ...
45. C -----
46.     DO 400 I = 1, NBLKS
47.         ROW1 = XBLK(I)
48.         ROW2 = XBLK(I+1) - 1
49.         LAST = XNONZ(ROW2+1)
50.         IF ( I .EQ. 1 .OR. LAST .EQ. XNONZ(ROW1) ) GO TO 300
51. C -----
52. C     MODIFY RHS VECTOR BY THE PRODUCT OF THE OFF-
53. C     DIAGONAL BLOCK WITH THE CORRESPONDING PART OF RHS.
54. C -----
55.     DO 200 ROW = ROW1, ROW2
56.         JSTRT = XNONZ(ROW)
57.         IF ( JSTRT .EQ. LAST ) GO TO 300
58.         JSTOP = XNONZ(ROW+1) - 1
59.         IF ( JSTOP .LT. JSTRT ) GO TO 200
60.         S = 0.0E0
61.         COUNT = JSTOP - JSTRT + 1
62.         OPS = OPS + COUNT
63.         DO 100 J = JSTRT, JSTOP
64.             COL = NZSUBS(J)
65.             S = S + RHS(COL)*NONZ(J)
66.     100     CONTINUE
67.             RHS(ROW) = RHS(ROW) - S
68.     200     CONTINUE
69.     300     NROW = ROW2 - ROW1 + 1
70.             CALL ELSLV ( NROW, XENV(ROW1), ENV, DIAG(ROW1),
71.     1                 RHS(ROW1) )
72.             CALL EUSLV ( NROW, XENV(ROW1), ENV, DIAG(ROW1),
73.     1                 RHS(ROW1) )
74.     400     CONTINUE
75. C -----
76. C BACKWARD SOLUTION ...

```

```

77. C      -----
78.      IF ( NBLKS .EQ. 1 ) RETURN
79.      LAST = XBLK(NBLKS) - 1
80.      DO 500 I = 1, LAST
81.          TEMP(I) = 0.0E0
82.      500 CONTINUE
83.      I = NBLKS
84.      COL1 = XBLK(I)
85.      COL2 = XBLK(I+1) - 1
86.      600 IF ( I .EQ. 1 ) RETURN
87.          LAST = XNONZ(COL2+1)
88.          IF ( LAST .EQ. XNONZ(COL1) ) GO TO 900
89. C      -----
90. C      MULTIPLY OFF-DIAGONAL BLOCK BY THE CORRESPONDING
91. C      PART OF THE SOLUTION VECTOR AND STORE IN TEMP.
92. C      -----
93.      DO 800 COL = COL1, COL2
94.          S = RHS(COL)
95.          IF ( S .EQ. 0.0E0 ) GO TO 800
96.          JSTRT = XNONZ(COL)
97.          IF ( JSTRT .EQ. LAST ) GO TO 900
98.          JSTOP = XNONZ(COL+1) - 1
99.          IF ( JSTOP .LT. JSTRT ) GO TO 800
100.             COUNT = JSTOP - JSTRT + 1
101.             OPS = OPS + COUNT
102.             DO 700 J = JSTRT, JSTOP
103.                 ROW = NZSUBS(J)
104.                 TEMP(ROW) = TEMP(ROW) + S*NONZ(J)
105.      700 CONTINUE
106.      800 CONTINUE
107.      900 I = I - 1
108.          COL1 = XBLK(I)
109.          COL2 = XBLK(I+1) - 1
110.          NCOL = COL2 - COL1 + 1
111.          CALL ELSLV ( NCOL, XENV(COL1), ENV,
112.      1              DIAG(COL1), TEMP(COL1) )
113.          CALL EUSLV ( NCOL, XENV(COL1), ENV, DIAG(COL1),
114.      1              TEMP(COL1) )
115.          DO 1000 J = COL1, COL2
116.              RHS(J) = RHS(J) - TEMP(J)
117.      1000 CONTINUE
118.      GO TO 600
119.      END

```

---

## Exercises



- 6.6.1) Let  $\mathbf{L}$  and  $\mathbf{V}$  be as described in Exercise 4.3.5 on page 65, where  $\mathbf{V}$  has only 3 nonzeros per column. Compare the operation costs of computing  $\mathbf{V}^T \mathbf{L}^{-T} \mathbf{L}^{-1} \mathbf{V}$  as  $\mathbf{V}^T (\mathbf{L}^{-T} (\mathbf{L}^{-1} \mathbf{V}))$  and  $(\mathbf{V}^T \mathbf{L}^{-T}) (\mathbf{L}^{-1} \mathbf{V})$ . Assume  $n$  and  $p$  are large, so lower order terms can be ignored.

## 6.7 Additional Notes

The idea of “throwing away” the off-diagonal blocks of the factor  $\mathbf{L}$  of  $\mathbf{A}$ , as discussed in this chapter, can be recursively applied (George and Liu [23]). To explain the strategy suppose  $\mathbf{A}$  is  $p$  by  $p$  partitioned, with  $\mathbf{x}$  and  $\mathbf{b}$  partitioned correspondingly. Let  $\mathbf{A}_{(k)}$  denote the leading block- $k$  by block- $k$  principal submatrix of  $\mathbf{A}$ , and let  $\mathbf{x}_{(k)}$  and  $\mathbf{b}_{(k)}$  denote the corresponding parts of  $\mathbf{x}$  and  $\mathbf{b}$  respectively. Finally, define submatrices of  $\mathbf{A}$  as in (6.5.1), with  $\mathbf{L}$  correspondingly partitioned, as shown in Figure 6.7.1 for  $p = 5$ .

$$\begin{array}{c}
 \left[ \begin{array}{c|c|c|c|c}
 \mathbf{A}_{11} & \mathbf{V}_2 & \mathbf{V}_3 & & \\
 \hline
 \mathbf{V}_2^T & \mathbf{A}_{22} & & & \\
 \hline
 & & \mathbf{V}_4 & & \\
 \hline
 \mathbf{V}_3^T & & \mathbf{A}_{33} & & \\
 \hline
 & & & \mathbf{V}_5 & \\
 \hline
 & & \mathbf{V}_4^T & & \mathbf{A}_{44} \\
 \hline
 & & & & \mathbf{V}_5^T \\
 \hline
 & & & & \mathbf{A}_{55}
 \end{array} \right]
 \quad
 \left[ \begin{array}{c|c|c|c|c}
 \mathbf{L}_{11} & & & & \\
 \hline
 \mathbf{W}_2^T & \mathbf{L}_{22} & & & \\
 \hline
 & & \mathbf{L}_{33} & & \\
 \hline
 & & & \mathbf{L}_{44} & \\
 \hline
 & & & & \mathbf{W}_5^T \\
 \hline
 & & & & \mathbf{L}_{55}
 \end{array} \right]
 \end{array}$$

$\mathbf{A} \qquad \qquad \mathbf{L}$

Figure 6.7.1: A recursively partitioned matrix and its Cholesky factor.

Using this notation, the system  $\mathbf{A}\mathbf{x} = \mathbf{b}$  can be expressed as

$$\begin{pmatrix} \mathbf{A}_{(4)} & \mathbf{V}_5 \\ \mathbf{V}_5^T & \mathbf{A}_{55} \end{pmatrix} \begin{pmatrix} \mathbf{x}_{(4)} \\ \mathbf{x}_5 \end{pmatrix} = \begin{pmatrix} \mathbf{b}_{(4)} \\ \mathbf{b}_5 \end{pmatrix},$$

and the factorization of  $\mathbf{A}$  can be expressed as

$$\begin{pmatrix} \mathbf{A}_{(4)} & \mathbf{O} \\ \mathbf{V}_5^T & \tilde{\mathbf{A}}_{55} \end{pmatrix} \begin{pmatrix} \mathbf{I} & \mathbf{A}_{(4)}^{-1} \mathbf{V}_5 \\ \mathbf{O} & \mathbf{I} \end{pmatrix},$$

where  $\tilde{\mathbf{A}}_{55} = \mathbf{A}_{55} - \mathbf{V}_5^T \mathbf{A}_{(4)}^{-1} \mathbf{V}_5$ .

Formally, we can solve  $\mathbf{A}\mathbf{x} = \mathbf{b}$  as follows:

- a) *Factorization:* Compute and factor  $\tilde{\mathbf{A}}_{55}$  into  $\mathbf{L}_{55} \mathbf{L}_{55}^T$ . (Note that  $\mathbf{V}_5^T \mathbf{A}_{(4)}^{-1} \mathbf{V}_5$  can be computed one column at a time, and the columns discarded after use.)
- b) *Solution:*
  - b.1) Solve  $\mathbf{A}_{(4)} \mathbf{y}_{(4)} = \mathbf{b}_{(4)}$ .
  - b.2) Solve  $\tilde{\mathbf{A}}_{55} \mathbf{x}_5 = \mathbf{b}_5 - \mathbf{V}_5^T \mathbf{y}_{(4)}$ .
  - b.3) Solve  $\mathbf{A}_{(4)} \tilde{\mathbf{x}}_{(4)} = \mathbf{V}_5 \mathbf{x}_5$ .
  - b.4) Compute  $\mathbf{x}_{(4)} = \mathbf{y}_{(4)} - \tilde{\mathbf{x}}_{(4)}$ .

Note that we have only used the ideas presented in Section 6.2 and Exercise 6.1.2 to avoid storing  $\mathbf{W}_5$ ; only  $\mathbf{V}_5$  is required. The crucial point is that all that is required for us to solve the five by five partitioned system without storing  $\mathbf{W}_5$  is that we be able to solve four by four partitioned systems. Obviously, we can use exactly the same strategy as shown above, to solve the block four by four systems without storing  $\mathbf{W}_4$ , and so on. Thus, we obtain a method which apparently solves a  $p$  by  $p$  block system requiring storage only for the diagonal blocks of  $\mathbf{L}$  and the off-diagonal blocks  $\mathbf{V}_i$  of the original matrix. However, note that each level of recursion requires a temporary vector  $\tilde{\mathbf{x}}_{(i)}$  (in Step b.3 above), so there is a point where a finer partitioning no longer achieves a reduction in storage requirement. There are many interesting unexplored questions related to this procedure, and the study of the use of these partitioning and throw-away ideas appears to be a potentially fertile research area.

Partitioning methods have been used successfully in utilizing auxiliary storage (Von Fuchs et al. [54]). The value of  $p$  is chosen so that the amount of main storage available is some convenient multiple of  $(n/p)^2$ . Since  $\mathbf{A}$  is sparse, some of the blocks will be all zeros. A pointer array is held in main store, with each pointer component either pointing to the current location of the corresponding block, if the block contains nonzeros, or else is zero. If the  $p$  by  $p$  pointer matrix is itself too large to be held in main store, then it can also be partitioned and the idea recursively applied. This storage management scheme obviously entails a certain amount of overhead, but experience suggests that it is a viable alternative to other out-of-core solution schemes such as band or frontal methods. One advantage is that the actual matrix

operations involve simple data structures; only square or rectangular arrays are involved.

Shier [48] has considered the use of tree partitionings in the context of explicitly inverting a matrix, and provides an algorithm different from ours for finding a tree partitioning of a graph.

## Chapter 7

# One-Way Dissection Methods for Finite Element Problems

### 7.1 Introduction

In this chapter we consider an ordering strategy designed primarily for problems arising in finite element applications. The strategy is similar to the method of Chapter 6 in that a quotient tree partitioning is obtained, and the computational ideas of implicit solution and asymmetric factorization are exploited. The primary advantage of the one-way dissection algorithm developed in this chapter is that the storage requirements are usually much less than those for either the band or quotient tree schemes described in previous chapters. Indeed, unless the problems are very large, for finite element problems the methods of this chapter are often the best methods in terms of storage requirements of *any* we discuss in this book. They also yield very low solution times, although their factorization times tend to be larger than those of some other methods.

Since the orderings studied in this chapter are quotient tree orderings, the storage and computing methods of Chapter 6 are appropriate, so we do not have to deal with these topics in this chapter. However, the one-way dissection schemes do demand a somewhat more sophisticated storage allocation procedure than that described in Section 6.5.3. This more general allocation procedure is the topic of Section 7.4.

## 7.2 An Example – The $s \times t$ Grid Problem

### 7.2.1 A One-Way Dissection Ordering

In this section we consider a simple  $s \times t$  grid problem which motivates the development of the algorithm of Section 7.3. Consider an  $s \times t$  grid or mesh as shown in Figure 7.2.1, having  $n = st$  nodes with  $s \leq t$ . The corresponding finite element matrix problem  $\mathbf{A}\mathbf{x} = \mathbf{b}$  we consider has the property that for some numbering of the equations (nodes) from 1 to  $n$ , we have that  $a_{ij} \neq 0$  implies node  $i$  and node  $j$  belong to the same small square.

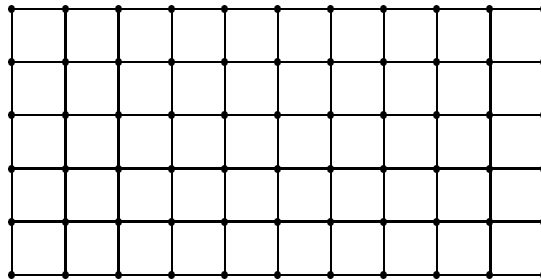


Figure 7.2.1: An  $s$  by  $t$  grid with  $s = 6$  and  $t = 11$ .

Now let  $\sigma$  be an integer satisfying  $1 \leq \sigma \ll t$ , and choose  $\sigma$  vertical grid lines (which we will refer to as *separators*) which dissect the mesh into  $\sigma + 1$  independent blocks of about the same size, as depicted in Figure 7.2.2, where  $\sigma = 4$ . The  $\sigma + 1$  independent blocks are numbered row by row, followed by the separators, as indicated by the arrows in Figure 7.2.2. The matrix structure this ordering induces in the triangular factor  $\mathbf{L}$  is shown in Figure 7.2.3, where the off-diagonal blocks with fills are hatched. We let

$$\delta = \frac{l - \sigma}{\sigma + 1},$$

that is, the length between dissectors.

Regarding  $\mathbf{A}$  and  $\mathbf{L}$  as partitioned into  $q^2$  submatrices, where  $q = 2\sigma + 1$ , we first note the dimensions of the various blocks:

$$\mathbf{A}_{kk} \text{ is } s\delta \text{ by } s\delta \text{ for } 1 \leq k \leq \sigma + 1. \quad (7.2.1)$$

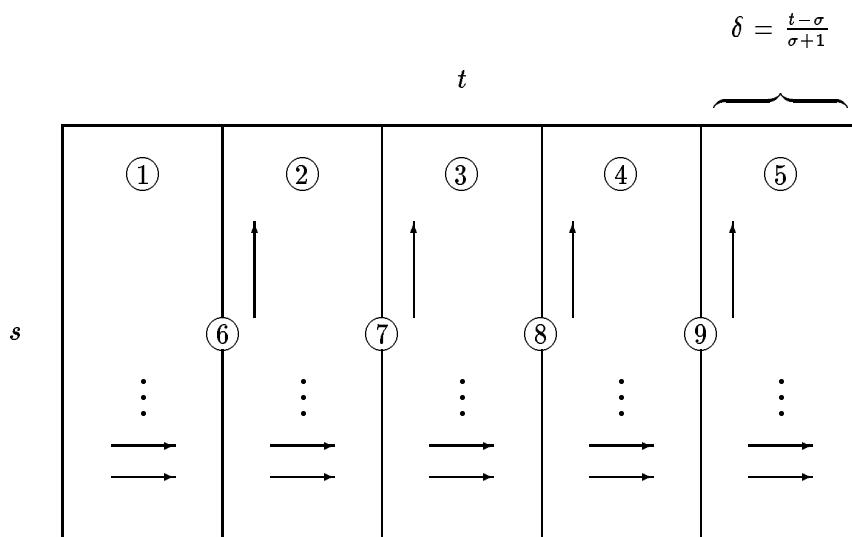


Figure 7.2.2: One-way dissection ordering of an  $s$  by  $t$  grid, indicated by the arrows. Here  $\sigma = 4$ , yielding a partitioning with  $2\sigma + 1 = 9$  members indicated by the circled numbers.

---

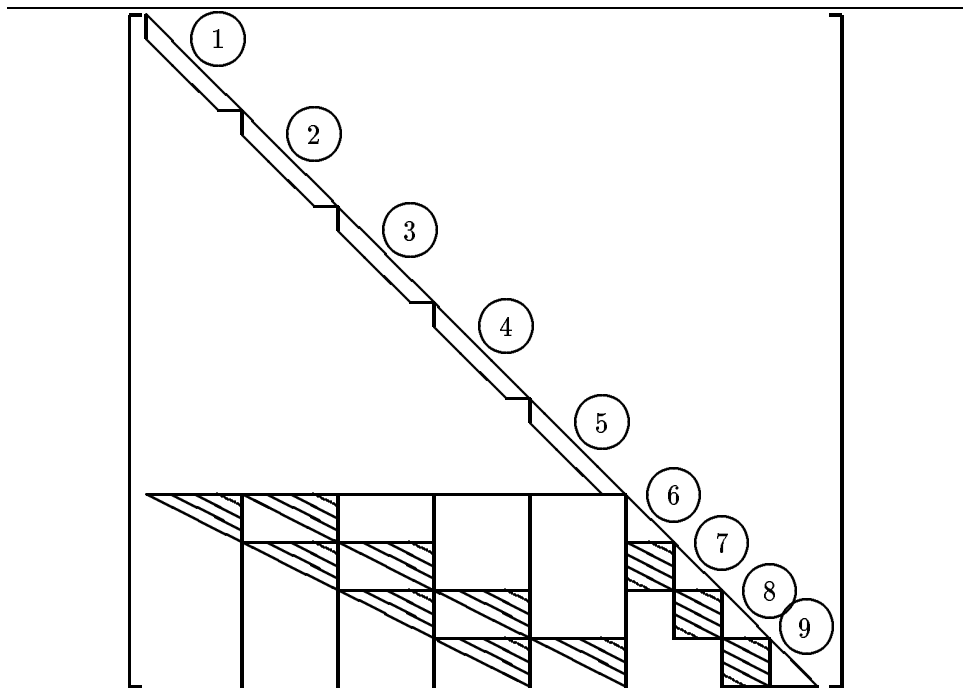


Figure 7.2.3: Matrix structure of  $L$  induced by the one-way dissection ordering of Figure 7.2.2. The hatched areas indicate where fill occurs in the off-diagonal blocks.

$$\mathbf{A}_{kj} \text{ is } s \text{ by } s\delta \text{ for } k > \sigma + 1 \text{ and } j \leq \sigma + 1. \quad (7.2.2)$$

$$\mathbf{A}_{kj} \text{ is } s \text{ by } s \text{ for } j > \sigma + 1 \text{ and } k > \sigma + 1. \quad (7.2.3)$$

Of course in practice  $\sigma$  must be chosen to be an integer, and unless  $\delta$  is also an integer, the leading  $\sigma + 1$  diagonal blocks will not all be exactly the same size. However, we will see later that these aberrations are of little practical significance; in any case, our objective in this section is to present some *basic ideas* rather than to study this  $s \times t$  grid problem in meticulous detail. For our purposes, we assume that  $\sigma$  and  $\delta$  are integers, and that  $s$  and  $t$  are large enough that  $s \ll s^2$  and  $t \ll t^2$ .

As we have already stated, the utility of this ordering hinges on using the partitioned matrix techniques developed in Chapter 6. Indeed, it is not difficult to determine that this ordering is no better or even worse than the standard band ordering if these techniques are not used. (see Exercises 7.2.4 and 7.2.5.)

The key observation which allows us to use the quotient tree techniques developed in Chapter 6 is that if we view the  $\sigma$  separator blocks as forming a *single* partition member, then the resulting partitioning, now with  $p = \sigma + 2$  members, is a *monotonely ordered tree partitioning*. This is depicted in Figure 7.2.4 for the example of Figure 7.2.2.

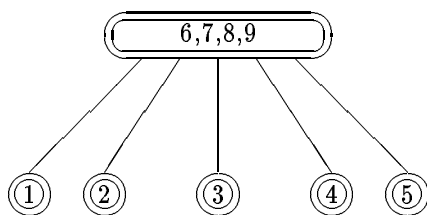


Figure 7.2.4: Quotient tree corresponding to one-way dissection ordering, obtained by placing the separators together in one partition.

---

Thus, we will use the storage scheme developed in Section 6.5, and thereby store only the diagonal blocks of  $\mathbf{L}$ , and the off-diagonal blocks of  $\mathbf{A}$ . For discussion purposes we will continue to regard  $\mathbf{A}$  and  $\mathbf{L}$  as  $q \times q$  partitioned where  $q = 2\sigma + 1$ , although the reader should understand that for computational purposes the last  $\sigma$  partition members are combined into one, so that



in effect  $p = \sigma + 2$  members are involved.

### 7.2.2 Storage Requirements

Denoting the partitions of  $L$  corresponding to  $A_{ij}$  by  $L_{ij}$ ,  $1 \leq i, j \leq 2\sigma + 1$ , we now derive an *estimate* for the storage requirements of this one-way dissection ordering, using the implicit storage scheme described in Section 6.5.1. The primary storage requirements are as follows:

- i)  $L_{kk}$ ,  $1 \leq k \leq \sigma + 1$ . The bandwidth of these band matrices is  $(t+1)/(\sigma+1)$ , yielding a combined storage requirement of

$$\frac{s(t-\sigma)(t+1)}{(\sigma+1)} \approx \frac{st^2}{\sigma}.$$

- ii)  $L_{kj}$ ,  $\sigma + 1 < j, k \leq 2\sigma + 1$ ,  $j < k$ . There are  $\sigma - 1$  fill blocks, and  $\sigma$  lower triangular blocks, all of which are  $s \times s$ , yielding a total storage requirement of

$$\frac{(\sigma-1)s^2 + \sigma s(s+1)}{2} \approx \frac{3\sigma s^2}{2}.$$

- iii)  $A_{kj}$ ,  $k \leq \sigma + 1$ ,  $j > \sigma + 1$ . Except for nodes near the boundary of the grid, all nodes on the separators are connected to 6 nodes in the leading  $\sigma + 1$  blocks. Thus, primary storage for these matrices totals about  $6\sigma s$ .

The overhead storage for items i) and ii) is  $ts + \sigma + 3$  (for the array **XENV** and **XBLK**), and about  $6\sigma s + ts$  for **XNONZ** and **NZSUBS**. Thus, if lower order terms are ignored, the storage requirement for this ordering, using the implicit storage scheme of Section 6.5.1, is approximately

$$S(\sigma) = \frac{st^2}{\sigma} + \frac{3\sigma s^2}{2}. \quad (7.2.4)$$

If our objective is to minimize storage, then we want to choose  $\sigma$  to minimize  $S(\sigma)$ . Differentiating with respect to  $\sigma$ , we have

$$\frac{dS}{d\sigma} = -\frac{st^2}{\sigma^2} + \frac{3s^2}{2}.$$

Using this, we find that  $S$  is approximately minimized by choosing  $\sigma = \sigma^*$  where

$$\sigma^* = t \left( \frac{2}{3m} \right)^{1/2} \quad (7.2.5)$$

yielding

$$S(\sigma^*) = \sqrt{6}s^{3/2}t + O(st). \quad (7.2.6)$$

Note that the corresponding optimal  $\delta^*$  is given by

$$\delta^* = \left(\frac{3s}{2}\right)^{1/2}.$$

It is interesting to compare this result with the storage requirements we would expect if we used a standard band or envelope scheme. Since  $s \leq t$ , we would number the grid column by column, yielding a matrix whose bandwidth is  $s + 1$ , and for large  $s$  and  $t$  the storage required for  $L$  would be  $s^2t + O(st)$ . Thus, asymptotically, this one-way dissection scheme reduces storage requirements by a factor of  $\sqrt{6/s}$  over the standard schemes of Chapter 4.

### 7.2.3 Operation Count for the Factorization

Let us now consider the computational requirements for this one-way dissection ordering. Basically, we simply have to count the operations needed to perform the factorization and solution algorithm described in Section 6.3. However, the off-diagonal blocks  $A_{kj}$ , for  $k \leq \sigma + 1$  and  $j > \sigma + 1$ , have rather special “pseudo tri-diagonal” structure, which is exploited by the subroutines `TSFCT` and `TSSLV`. Thus, determining an approximate operation count is far from trivial. In this Section we consider the factorization; Section 7.2.4 contains a derivation of an approximate operation count for the solution.

It is helpful in the derivation to break the computation into the three categories, where again we ignore low order terms in the calculations.

1. The factorization of the  $\sigma + 1$  leading diagonal blocks.

(In our example of Figures 7.2.1–7.2.3, where  $\sigma = 4$ , this is the computation of  $L_{kk}$ ,  $1 \leq k \leq 5$ .) Observing that the bandwidth of these matrices is  $(t + 1)/(\sigma + 1)$ , and using Theorem 4.2.1, we conclude that the operation count for this category is approximately

$$\frac{st^3}{2\sigma^2}.$$

2. The computation of the  $L_{kj}$ , for  $k \geq j$  and  $j > \sigma + 1$ .

This corresponds to factoring an  $s\sigma \times s\sigma$  block tri-diagonal matrix having blocks of size  $s$ . Using the results of Sections 2.2 and 2.3, we find that operation count is approximately

$$\frac{7\sigma s^3}{6}.$$

3. The modifications to  $\mathbf{A}_{jj}$ ,  $\mathbf{A}_{j+1,j}$ , and  $\mathbf{A}_{j+1,j+1}$  for  $j > \sigma + 1$  involving the off-diagonal blocks  $\mathbf{A}_{kj}$  and the computed  $\mathbf{L}_{kk}$ ,  $k \leq \sigma + 1$ , as depicted in Figure 7.2.5.

The operation count for this computation is discussed below.

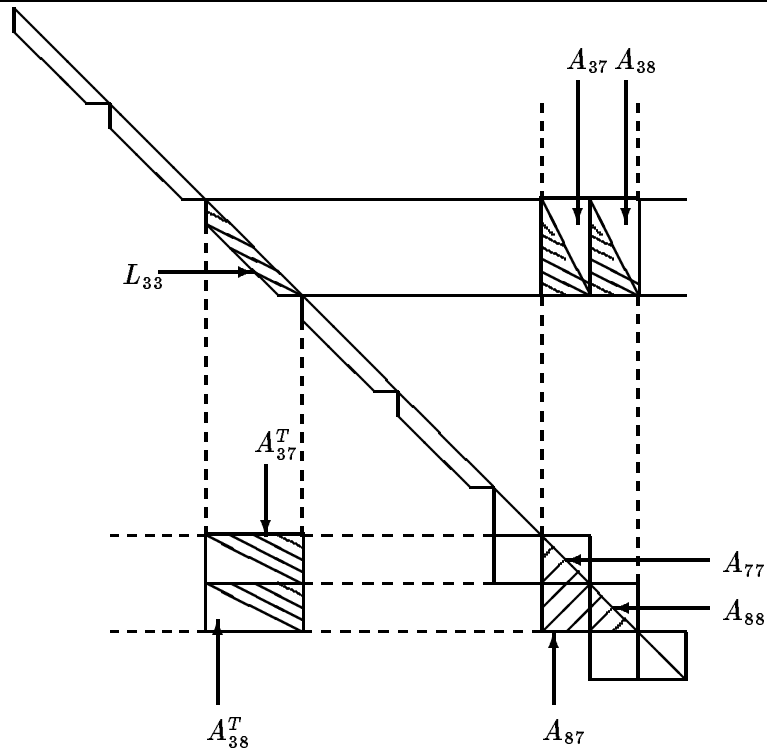


Figure 7.2.5: Matrices which interact with and/or are modified by  $\mathbf{L}_{kk}$ ,  $\mathbf{A}_{kj}$  and  $\mathbf{A}_{k,j+1}$ , where  $k = 3$  and  $j = 7$ .

In computing the modification matrix in the asymmetric way, we have to compute

$$\mathbf{L}_{kk}^{-T} (\mathbf{L}_{kk}^{-1} \mathbf{A}_{kj}), \quad (7.2.7)$$

for  $j > \sigma + 1$  and  $k \leq \sigma + 1$ . In view of the results in Section 6.6, it is not necessary to compute that part of  $L_{kk}^{-T}(L_{kk}^{-1}A_{kj})$  which is above the first nonzero in each column of  $A_{kj}$ . Thus, when computing  $W = L_{kk}^{-1}A_{kj}$ , we exploit leading zeros in the columns of  $A_{kj}$ , and when computing  $\tilde{W} = L_{kk}^{-T}W$  we stop the computation as soon as the last *required* element of  $\tilde{W}$  has been computed, as depicted in Figure 7.2.6.

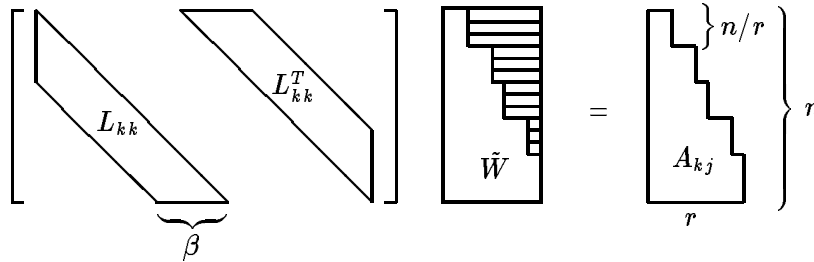


Figure 7.2.6: Structure of  $A_{kj}$  and the part of  $\tilde{W}$  that needs to be computed.

It is straightforward to show that the number of operations required to compute this part of  $\tilde{W}$  is approximately given by

$$n(\beta + 1)(r - 1), \tag{7.2.8}$$

where  $A_{kj}$  is  $n \times r$  and  $L_{kk} + L_{kk}^T$  is an  $n \times n$  band matrix with bandwidth  $\beta \ll n$  (see Exercise 7.2.5 on page 278 ). Here,  $n \approx st/\sigma, \beta \approx t/\sigma$  and  $r = m$ ; thus the expression 7.2.8 becomes

$$\frac{s^2 t^2}{\sigma^2}.$$

Note that there are in total  $2\sigma$  such off-diagonal blocks, so the computation required to compute all

$$L_{kk}^{-T}(L_{kk}^{-1}A_{kj}) \quad \text{for } j > \sigma + 1 \text{ and } k \leq \sigma + 1$$

is approximately

$$\frac{2s^2 t^2}{\sigma}. \tag{7.2.9}$$

We now estimate the cost of computing the modifications to  $A_{kj}, k > \sigma + 1, j > \sigma + 1$ . With (7.2.7) computed, we note that the modification to each

entry in the diagonal blocks  $\mathbf{A}_{kk}$ ,  $k > \sigma + 1$  can be computed in six operations, while that to the off-diagonal blocks  $\mathbf{A}_{k,k-1}$  requires three operations. That is, the cost for modification is  $O(\sigma s^2)$ . Thus, an estimate for the total number of operations required for the factorization, using this one-way dissection ordering, is given by

$$\theta_F(\sigma) = \frac{st^3}{2\sigma^2} + \frac{7\sigma s^3}{6} + \frac{2s^2t^2}{\sigma}. \quad (7.2.10)$$

If our objective is to minimize the operation count for the factorization, using this one-way dissection ordering, we want to find the  $\sigma_F$  which minimizes  $\theta_F(\sigma)$ . For large  $s$  and  $t$ , it can be shown that choosing

$$\sigma_F = t \left( \frac{12}{7s} \right)^{1/2}$$

approximately minimizes (7.2.10), yielding (see Exercise 7.2.6 on page 278 )

$$\theta_F(\sigma_F) = \left( \frac{28}{3} \right)^{1/2} s^{5/2}t + O(s^2t). \quad (7.2.11)$$

The corresponding  $\delta_F$  is given by

$$\delta_F = \left( \frac{7m}{12} \right)^{1/2}.$$

Again it is interesting to compare this result with the operation count if we use a standard band or envelope scheme as described in Chapter 4. For this  $s \times t$  grid problem, the factorization operation count would be  $\approx \frac{1}{2}s^3t$ . Thus, asymptotically this one-way dissection scheme reduces the factorization count by a factor of roughly  $4\sqrt{7/(3s)}$ .

### 7.2.4 Operation Count for the Solution

We now derive an estimate of the operation count required to solve  $\mathbf{Ax} = \mathbf{b}$ , given the “factorization” as computed in the preceding subsection.

First observe that each of the  $\sigma + 1$  leading diagonal blocks  $\mathbf{L}_{kk}$ ,  $1 \leq k \leq \sigma + 1$ , is used four times, twice in the lower solve and twice again in the upper solve. This yields an operation count of approximately

$$\frac{4st^2}{\sigma}.$$

The non-null blocks  $\mathbf{L}_{kj}$ , for  $k > \sigma + 1$  and  $j > \sigma + 1$ , are each used twice, for a total operation count of  $3\sigma s^2 + O(\sigma s)$ . Each matrix  $\mathbf{A}_{kj}$ , for  $k > \sigma + 1$  and  $j \leq \sigma + 1$ , is used twice, yielding an operation count of about  $12\sigma s$ . Thus, an estimate for the operation count associated with the solution, using this one-way dissection ordering, is

$$\theta_s(\sigma) = \frac{4st^2}{\sigma} + 3\sigma s^2. \quad (7.2.12)$$

If we wish to minimize  $\theta_s$  with respect to  $\sigma$ , we find  $\sigma$  should be approximately

$$\sigma_s = \frac{2t}{\sqrt{3s}},$$

whence

$$\theta_s(\sigma_s) = 4\sqrt{3}s^{3/2}t + O(st). \quad (7.2.13)$$

Again it is interesting to compare (7.1.11) with the corresponding operation count if we were to use standard band or envelope schemes, which would be about  $2s^2t$ . Thus, asymptotically, the one-way dissection ordering reduces the solution operation count by a factor of about  $2\sqrt{3/s}$ .

Of course in practice we cannot choose  $\sigma$  to simultaneously minimize storage, factorization operation count, and solution operation count;  $\sigma$  must be fixed. Since the main attraction for these methods is their low storage requirements, in the algorithm of the next section  $\sigma$  is chosen to attempt to minimize storage.

### Exercises

- 7.2.1) What are the coefficients of the high order terms in (7.2.10) and (7.2.13) if  $\sigma$  is chosen to be  $\sigma^*$ , given by (7.2.5)?
- 7.2.2) Suppose we use the one-way dissection ordering of this section with  $\sigma$  chosen to be  $O(t/\sqrt{s})$ , but we do not use the implicit storage technique; that is, we actually store the off-diagonal blocks  $\mathbf{L}_{ij}$ ,  $i > \sigma + 1$ ,  $j \leq \sigma + 1$ . What would the storage requirements be then? If we used these blocks in the solution scheme, what would the operation count corresponding to  $\theta_s$  now be?
- 7.2.3) Suppose we use the one-way dissection ordering of this section with  $\sigma$  chosen to be  $t/\sqrt{s}$ , but we use the symmetric version of the factorization scheme rather than the asymmetric version. (See Section 6.2.1)

Show that now the factorization operation count is  $O(s^3t)$  rather than  $O(s^{5/2}t)$ . How much temporary storage is required to carry out the computation?

- 7.2.4) Throughout Section 7.2 we assume that  $s \leq t$ , although we did not explicitly use that fact anywhere. Do our results still apply for  $s > t$ ? Why did we assume  $s \leq t$ ?
- 7.2.5) Let  $M$  be an  $n \times n$  symmetric positive definite band matrix with bandwidth  $\beta \ll n$  and Cholesky factorization  $LL^T$ . Let  $V$  be an  $n \times r$  ( $r \ll n$ ) “pseudo-tridiagonal” matrix, for which the leading nonzero in column  $i$  is in position

$$\mu_i = \left\lceil \frac{(i-1)(n-1)}{r-1} \right\rceil$$

and let  $\tilde{W} = L^{-T}(L^{-1}V)$ . Show that the number of operations required to compute  $\tilde{w}_{ij}$ ,  $1 \leq i \leq r$ ,  $\mu_i \leq j \leq n$ , is approximately  $n(\beta+1)(r-1)$ . (Note that this is approximately the pseudo-lower triangle of  $\tilde{W}$ , described in Exercise 2.3.8 on page 30.)

- 7.2.6) Let  $\sigma_F$  minimize  $\theta_F(\sigma)$  in (7.2.10). Show that a lower bound for  $\sigma_F$  is given by

$$\bar{\sigma}_F = t \left( \frac{12}{7s} \right)^{1/2},$$

whence

$$\theta_F(\bar{\sigma}_F) = \frac{7}{24}s^2t + 2 \left( \frac{7}{3} \right)^{1/2} s^{5/2}t.$$

- 7.2.7) In the description of the one-way dissection ordering of the  $s \times t$  grid given in this section, the separator blocks were numbered “end to end.” It turns out that the order in which these separator blocks are numbered is important. For example, the blocks in the example of Figure 7.2.2 might have been numbered as indicated in the diagram in Figure 7.2.7.
- a) Draw a figure similar to Figure 7.2.5 showing the structure of  $L$  corresponding to this ordering. Is there more or fewer fill blocks than the ordering shown in Figure 7.2.2?

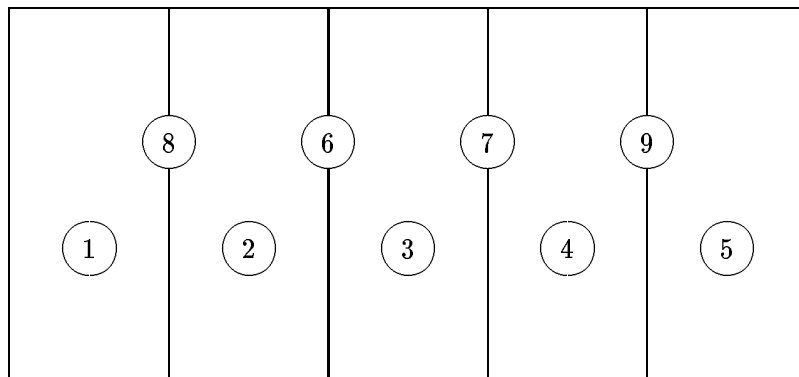


Figure 7.2.7: Different way of labelling the separators in the one-way dissection algorithm.

- b) For the one-way dissection ordering of the  $s \times t$  grid shown in Figure 7.2.2, the number of fill blocks is  $\sigma - 1$ . Show that for some orderings of the separator blocks, as many as  $2\sigma - 3$  fill blocks may result.

### 7.3 An Algorithm for Finding One-Way Dissection Orderings of Irregular Mesh Problems

#### 7.3.1 The Algorithm

The description and analysis of the previous section suggests that in general terms, we would like to find a set of “parallel” separators having relatively few nodes. These separators should disconnect the graph or mesh into components which can be ordered so as to have small envelopes. This is essentially what the following heuristic algorithm attempts to do.

The algorithm operates on a given graph  $\mathcal{G} = (X, E)$ , which we assume to be connected. The extension to disconnected graphs is obvious. Recall from Chapter 3 that the set  $Y \subset X$  is a *separator* of the connected graph  $\mathcal{G}$  if the section graph  $\mathcal{G}(X - Y)$  is disconnected.

We now give a step-by-step description of the algorithm, followed by some



explanatory remarks for the important steps. In the algorithm  $n = |X|$ , and  $s$  and  $t$  correspond roughly to  $s$  and  $t - 1$  in Section 7.2. The algorithm attempts to choose  $\sigma$  to minimize storage, but it can easily be modified so as to attempt to minimize the factorization or solution operation count.

**Step 1** (Generate level structure) Find a pseudo-peripheral node  $x$  by the algorithm of Section 4.4.2, and generate the level structure rooted at  $x$ .

$$\mathcal{L}(x) = \{L_0, L_1, \dots, L_t\}.$$

**Step 2** (Estimate  $\delta$ ) Calculate  $s = n/(t + 1)$ , and set

$$\delta = \left( \frac{3s + 13}{2} \right)^{1/2}.$$

**Step 3** (Limiting case) If  $\delta < t/2$ , and  $|X| > 50$  go to Step 4. Otherwise, set  $p = 1$ ,  $Y_p = X$  and go to Step 6.

**Step 4** (Find separator) Set  $i = 1$ ,  $j = \lfloor \delta + 0.5 \rfloor$ , and  $T = \phi$ .  
While  $j < t$  do the following

4.1) Choose  $T_i = \{x \in L_j \mid \text{Adj}(x) \cap L_{j+1} \neq \phi\}$ .

4.2) Set  $T = T \cup T_i$

4.3) Set  $i \leftarrow i + 1$  and  $j = \lfloor i\delta + 0.5 \rfloor$ .

**Step 5** (Define blocks) Let  $Y_k$ ,  $k = 1, \dots, p-1$  be the connected components of the section graph  $\mathcal{G}(X - T)$ , and set  $Y_p = T$ .

**Step 6** (Internal numbering) Number each  $Y_k$ ,  $k = 1, \dots, p$  consecutively using the method described in Section 6.5.2.

Step 1 of the algorithm produces a (hopefully) long, narrow level structure. This is desirable because the separators are selected as subsets of some of the levels  $L_i$ .

The calculation of the numbers  $s$  and  $t$  computed in Step 2 is motivated directly by the analysis of the  $s \times t$  grid in Section 7.2. Since  $s$  is the average number of nodes per level, it serves as a measure of the width of the level structure. The derivation of  $\sigma^*$  given in (7.2.5) was obtained in a fairly crude way, since our objective was simply to convey the basic ideas. A more careful

analysis along with some experimentation suggests that a better value for  $\sigma^*$  is

$$t \left( \frac{2}{3s + 13} \right)^{1/2}.$$

The corresponding  $\delta^*$  is given by the formula used in Step 2.

Step 3 is designed to handle anomalous situations where  $m \ll t$ , or when  $n$  is simply too small to make the use of the one-way dissection method worthwhile. Experiments indicate that for small finite element problems, and/or “long slender” problems, the methods of Chapter 4 are more efficient, regardless of the basis for comparison. In these cases, the entire graph is processed as one block ( $p = 1$ ). That is, an ordinary envelope ordering as discussed in Chapter 4 is produced for the graph.

Step 4 performs the actual selection of the separators, and is done essentially as though the graph corresponds to an  $s \times t$  grid as studied in Section 7.2. As noted earlier, each  $L_i$  of  $\mathcal{L}$  is a separator of  $\mathcal{G}$ . In Step 4, approximately equally spaced levels are chosen from  $\mathcal{L}$ , and subsets of these levels (the  $T_i$ ) which are possibly smaller separators are then found.

Finally, in Step 6 the  $p \geq \sigma + 1$  independent blocks created by removing the separators from the graph are numbered, using the internal renumbering scheme described in Section 6.5.2.

Although the choice of  $\sigma$  and the method of selection of the separators seems rather crude, we have found that attempts at more sophistication do not often yield significant benefits (except for some unrealistic, contrived examples). Just as in the regular rectangular grid case, the storage requirement, as a function of  $\sigma$ , is very flat near its minimum. Even relatively large perturbations in the value of  $\sigma$ , and in the selection of the separators, usually produce rather small changes in storage requirements.

In Figure 7.3.1 we have an example of an irregular mesh problem, along with some indications of the steps carried out by the algorithm. (For purposes of this illustration, we assume that the test for  $|X| > 50$  in Step 3 of the algorithm has been removed.) Figure 7.3.1 (a) contains the level numbers of the original level structure, while Figure 7.3.1 (b) displays the nodes chosen as the separators. Here  $s = n/(t + 1) = 25/11 \approx 2.27$ ,  $\delta = \sqrt{9.91} \approx 3.12$ . The levels chosen from which to pick the separators are levels 4 and 8.

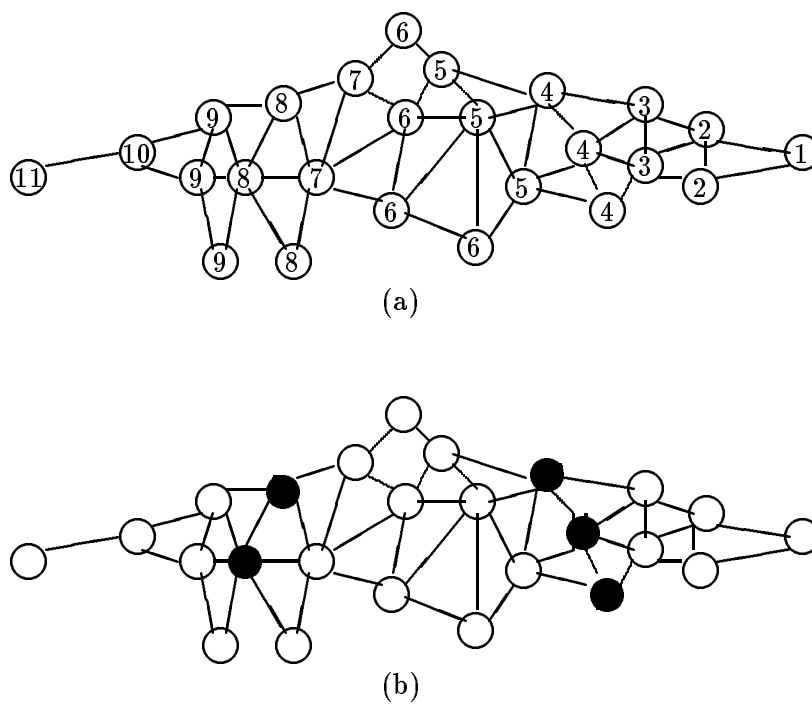


Figure 7.3.1: Diagram of an irregular mesh showing the separators chosen by the algorithm.

---

### 7.3.2 Subroutines for Finding a One-Way Dissection Partitioning

The set of subroutines which implements the one-way dissection algorithm is given in the control diagram in Figure 7.3.2. The subroutines **FNROOT** and **ROOTLS** are used together to determine a pseudo-peripheral node of a connected component in a given graph. They have been discussed in detail in Section 4.4.3. The subroutine **REVRSE** is a utility subroutine that is used to reverse an integer array. The execution of the calling statement

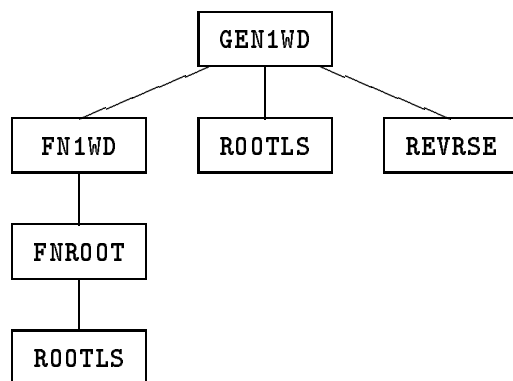


Figure 7.3.2: Control relation of subroutines for the one-way dissection algorithm.

---

```
CALL REVRSE ( NV, V )
```

will interchange the entries in the integer vector **V** of size **NV** in the following way:

$$V(i) \leftrightarrow V(NV-i+1) \quad \text{for } 1 \leq i \leq \lfloor NV \rfloor / 2.$$

The remaining two subroutines **GEN1WD** and **FN1WD** are described in detail below.

#### **GEN1WD (GENERAL 1-Way Dissection)**

This is the driver subroutine for finding a one-way dissection partitioning of a general disconnected graph. The input and output parameters of **GEN1WD** follow the same notations as the implementations of other ordering algorithms.

The parameters `NEQNS`, `XADJ` and `ADJNCY` are for the adjacency structure of the given graph. Returned from the subroutine are the one-way dissection ordering in the vector `PERM`, and the partitioning information in `(NBLKS, XBLK)`. Three working vectors `MASK`, `XLS` and `LS` are used by `GEN1WD`. The array pair `(XLS, LS)` is used by `FN1WD` to store a level structure rooted at a pseudo-peripheral node, and the vector `MASK` is used by the subroutine to mask off nodes that have been numbered.

The subroutine begins by initializing the vector `MASK` so that all nodes are considered unnumbered. It then goes through the graph and obtains a node  $i$  not yet numbered. The node defines an unnumbered connected component in the graph and the subroutine `FN1WD` is called to find a one-way dissector for the component. The set of dissecting nodes forms a block in the partitioning. Each component in the remainder of the dissected subgraph also constitutes a block, and they are found by calling the subroutine `ROOTLS`.

After going through all the connected components in the graph, the subroutine reverses the permutation vector `PERM` and block index vector `XBLK`, since the one-way dissectors which are found first should be ordered after the remaining nodes.

---

```

1. C*****
2. C*****
3. C*****      GEN1WD . . . . GENERAL ONE-WAY DISSECTION *****
4. C*****
5. C*****
6. C
7. C      PURPOSE - GEN1WD FINDS A ONE-WAY DISSECTION PARTITIONING
8. C      FOR A GENERAL GRAPH. FN1WD IS USED FOR EACH CONNECTED
9. C      COMPONENT.
10. C
11. C      INPUT PARAMETERS -
12. C      NEQNS - NUMBER OF EQUATIONS.
13. C      (XADJ, ADJNCY) - THE ADJACENCY STRUCTURE PAIR.
14. C
15. C      OUTPUT PARAMETERS -
16. C      (NBLKS, XBLK) - THE PARTITIONING FOUND.
17. C      PERM - THE ONE-WAY DISSECTION ORDERING.
18. C
19. C      WORKING VECTORS -
20. C      MASK - IS USED TO MARK VARIABLES THAT HAVE
21. C      BEEN NUMBERED DURING THE ORDERING PROCESS.
22. C      (XLS, LS) - LEVEL STRUCTURE USED BY ROOTLS.
23. C
24. C      PROGRAM SUBROUTINES -

```

```

25. C          FN1WD, REVRSE, ROOTLS.
26. C
27. C*****
28. C
29. C          SUBROUTINE GEN1WD ( NEQNS, XADJ, ADJNCY, MASK,
30. C          1              NBLKS, XBLK, PERM, XLS, LS )
31. C
32. C*****
33. C
34. C          INTEGER ADJNCY(1), LS(1), MASK(1), PERM(1),
35. C          1              XBLK(1), XLS(1)
36. C          INTEGER XADJ(1), CCSIZE, I, J, K, LNUM,
37. C          1              NBLKS, NEQNS, NLVL, NODE, NSEP,
38. C          1              NUM, ROOT
39. C
40. C*****
41. C
42. C          DO 100 I = 1, NEQNS
43. C              MASK(I) = 1
44. C          100 CONTINUE
45. C          NBLKS = 0
46. C          NUM = 0
47. C          DO 400 I = 1, NEQNS
48. C              IF ( MASK(I) .EQ. 0 ) GO TO 400
49. C          -----
50. C          FIND A ONE-WAY DISSECTOR FOR EACH COMPONENT.
51. C          -----
52. C          ROOT = I
53. C          CALL FN1WD ( ROOT, XADJ, ADJNCY, MASK,
54. C          1              NSEP, PERM(NUM+1), NLVL, XLS, LS )
55. C          NUM = NUM + NSEP
56. C          NBLKS = NBLKS + 1
57. C          XBLK(NBLKS) = NEQNS - NUM + 1
58. C          CCSIZE = XLS(NLVL+1) - 1
59. C          -----
60. C          NUMBER THE REMAINING NODES IN THE COMPONENT.
61. C          EACH COMPONENT IN THE REMAINING SUBGRAPH FORMS
62. C          A NEW BLOCK IN THE PARTITIONING.
63. C          -----
64. C          DO 300 J = 1, CCSIZE
65. C              NODE = LS(J)
66. C              IF ( MASK(NODE) .EQ. 0 ) GO TO 300
67. C              CALL ROOTLS ( NODE, XADJ, ADJNCY, MASK,
68. C          1              NLVL, XLS, PERM(NUM+1) )
69. C              LNUM = NUM + 1
70. C              NUM = NUM + XLS(NLVL+1) - 1
71. C              NBLKS = NBLKS + 1

```

```

72.                XBLK(NBLKS) = NEQNS - NUM + 1
73.                DO 200 K = LNUM, NUM
74.                  NODE = PERM(K)
75.                  MASK(NODE) = 0
76.    200          CONTINUE
77.                IF ( NUM .GT. NEQNS ) GO TO 500
78.    300          CONTINUE
79.    400          CONTINUE
80.    C           -----
81.    C           SINCE DISSECTORS FOUND FIRST SHOULD BE ORDERED LAST,
82.    C           ROUTINE REVRSE IS CALLED TO ADJUST THE ORDERING
83.    C           VECTOR, AND THE BLOCK INDEX VECTOR.
84.    C           -----
85.    500          CALL REVRSE ( NEQNS, PERM )
86.                CALL REVRSE ( NBLKS, XBLK )
87.                XBLK(NBLKS+1) = NEQNS + 1
88.                RETURN
89.    END

```

---

### FN1WD (FiNd 1-Way Dissection ordering)

This subroutine applies the one-way dissection algorithm described in Section 7.3.1 to a connected component of a subgraph. It operates on a component specified by the input parameters `ROOT`, `MASK`, `XADJ` and `ADJNCY`. Output from this subroutine is the set of dissecting nodes given by (`NSEP`, `SEP`).

The first step in the subroutine is to find a level structure rooted at a pseudo-peripheral node which it does by calling `FNROOT`. Based on the characteristics of the level structure (`NLVL`, the number of levels and `WIDTH`, the average width), the subroutine determines the level increment `DELTA` to be used. If the number of levels `NLVL` or the size of the component is too small, the whole component is returned as the “dissector”.

With `DELTA` determined, the subroutine then marches along the level structure picking up levels, subsets of which form the set of parallel dissectors.

---

```

1.  C*****
2.  C*****
3.  C*****      FN1WD . . . . FIND ONE-WAY DISSECTORS      *****
4.  C*****
5.  C*****
6.  C
7.  C      PURPOSE - THIS SUBROUTINE FINDS ONE-WAY DISSECTORS OF
8.  C      A CONNECTED COMPONENT SPECIFIED BY MASK AND ROOT.
9.  C

```

```

10. C     INPUT PARAMETERS -
11. C     ROOT - A NODE THAT DEFINES (ALONG WITH MASK) THE
12. C     COMPONENT TO BE PROCESSED.
13. C     (XADJ, ADJNCY) - THE ADJACENCY STRUCTURE.
14. C
15. C     OUTPUT PARAMETERS -
16. C     NSEP - NUMBER OF NODES IN THE ONE-WAY DISSECTORS.
17. C     SEP - VECTOR CONTAINING THE DISSECTOR NODES.
18. C
19. C     UPDATED PARAMETER -
20. C     MASK - NODES IN THE DISSECTOR HAVE THEIR MASK VALUES
21. C     SET TO ZERO.
22. C
23. C     WORKING PARAMETERS-
24. C     (XLS, LS) - LEVEL STRUCTURE USED BY THE ROUTINE FNROOT.
25. C
26. C     PROGRAM SUBROUTINE -
27. C     FNROOT.
28. C
29. C*****
30. C
31. C     SUBROUTINE FN1WD ( ROOT, XADJ, ADJNCY, MASK,
32. C     1             NSEP, SEP, NLVL, XLS, LS )
33. C
34. C*****
35. C
36. C     INTEGER ADJNCY(1), LS(1), MASK(1), SEP(1), XLS(1)
37. C     INTEGER XADJ(1), I, J, K, KSTOP, KSTRT, LP1BEG, LP1END,
38. C     1             LVL, LVLBEG, LVLEND, NBR, NLVL, NODE,
39. C     1             NSEP, ROOT
40. C     REAL DELTP1, FNLVL, WIDTH
41. C
42. C*****
43. C
44. C     CALL FNROOT ( ROOT, XADJ, ADJNCY, MASK,
45. C     1             NLVL, XLS, LS )
46. C     FNLVL = FLOAT(NLVL)
47. C     NSEP = XLS(NLVL + 1) - 1
48. C     WIDTH = FLOAT(NSEP) / FNLVL
49. C     DELTP1 = 1.0 + SQRT((3.0*WIDTH+13.0)/2.0)
50. C     IF (NSEP .GE. 50 .AND. DELTP1 .LE. 0.5*FNLVL) GO TO 300
51. C     -----
52. C     THE COMPONENT IS TOO SMALL, OR THE LEVEL STRUCTURE
53. C     IS VERY LONG AND NARROW. RETURN THE WHOLE COMPONENT.
54. C     -----
55. C     DO 200 I = 1, NSEP
56. C         NODE = LS(I)

```



```

57.          SEP(I) = NODE
58.          MASK(NODE) = 0
59.    200    CONTINUE
60.          RETURN
61.  C      -----
62.  C      FIND THE PARALLEL DISSECTORS.
63.  C      -----
64.    300    NSEP = 0
65.          I = 0
66.    400    I = I + 1
67.          LVL = IFIX (FLOAT(I)*DELTP1 + 0.5)
68.          IF ( LVL .GE. NLVL ) RETURN
69.          LVLBEG = XLS(LVL)
70.          LP1BEG = XLS(LVL + 1)
71.          LVLEND = LP1BEG - 1
72.          LP1END = XLS(LVL + 2) - 1
73.          DO 500 J = LP1BEG, LP1END
74.            NODE = LS(J)
75.            XADJ(NODE) = - XADJ(NODE)
76.    500    CONTINUE
77.  C      -----
78.  C      NODES IN LEVEL LVL ARE CHOSEN TO FORM DISSECTOR.
79.  C      INCLUDE ONLY THOSE WITH NEIGHBORS IN LVL+1 LEVEL.
80.  C      XADJ IS USED TEMPORARILY TO MARK NODES IN LVL+1.
81.  C      -----
82.          DO 700 J = LVLBEG, LVLEND
83.            NODE = LS(J)
84.            KSTRT = XADJ(NODE)
85.            KSTOP = IABS(XADJ(NODE+1)) - 1
86.            DO 600 K = KSTRT, KSTOP
87.              NBR = ADJNCY(K)
88.              IF ( XADJ(NBR) .GT. 0 ) GO TO 600
89.              NSEP = NSEP + 1
90.              SEP(NSEP) = NODE
91.              MASK(NODE) = 0
92.              GO TO 700
93.    600    CONTINUE
94.    700    CONTINUE
95.          DO 800 J = LP1BEG, LP1END
96.            NODE = LS(J)
97.            XADJ(NODE) = - XADJ(NODE)
98.    800    CONTINUE
99.          GO TO 400
100.        END

```

---

## 7.4 On Finding the Envelope Structure of Diagonal Blocks

In Chapter 4, the envelope structure of a symmetric matrix  $\mathbf{A}$  has been studied. It has been shown that the envelope structure is preserved under symmetric factorization; in other words, if  $\mathbf{F}$  is the filled matrix of  $\mathbf{A}$ , then

$$\text{Env}(\mathbf{A}) = \text{Env}(\mathbf{F}).$$

In this section, we consider the envelope structure of the diagonal block submatrices of the filled matrix with respect to a given partitioning. This is important in setting up the data structure for the storage scheme described in Section 6.5.1.

### 7.4.1 Statement of the Problem

Let  $\mathbf{A}$  be a sparse symmetric matrix partitioned as

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \cdots & \mathbf{A}_{1p} \\ \mathbf{A}_{12}^T & \mathbf{A}_{22} & \cdots & \mathbf{A}_{2p} \\ \vdots & & \ddots & \vdots \\ \mathbf{A}_{1p}^T & \mathbf{A}_{2p}^T & \cdots & \mathbf{A}_{pp} \end{pmatrix}, \quad (7.4.1)$$

where each  $\mathbf{A}_{kk}$  is a square submatrix. The *block diagonal matrix* of  $\mathbf{A}$  with respect to the given partitioning is defined to be

$$\text{Bdiag}(\mathbf{A}) = \begin{pmatrix} \mathbf{A}_{11} & & & \mathbf{O} \\ & \mathbf{A}_{22} & & \\ & & \ddots & \\ \mathbf{O} & & & \mathbf{A}_{pp} \end{pmatrix}. \quad (7.4.2)$$

Let the triangular factor  $\mathbf{L}$  of  $\mathbf{A}$  be correspondingly partitioned as

$$\mathbf{L} = \begin{pmatrix} \mathbf{L}_{11} & & & \mathbf{O} \\ \mathbf{L}_{21} & \mathbf{L}_{22} & & \\ \vdots & & \ddots & \\ \mathbf{L}_{p1} & \mathbf{L}_{p2} & \cdots & \mathbf{L}_{pp} \end{pmatrix}.$$

Then the associated block diagonal matrix of the filled matrix  $\mathbf{F}$  will be

$$Bdiag(\mathbf{F}) = \begin{pmatrix} \mathbf{F}_{11} & & \mathbf{O} \\ & \mathbf{F}_{22} & \\ & & \ddots \\ \mathbf{O} & & & \mathbf{F}_{pp} \end{pmatrix}$$

where  $\mathbf{F}_{kk} = \mathbf{L}_{kk} + \mathbf{L}_{kk}^T$  for  $1 \leq k \leq p$ .

Our objective is to determine the envelope structure of  $Bdiag(\mathbf{F})$ . Although  $Env(\mathbf{A}) = Env(\mathbf{F})$ , the result does not hold in general for  $Bdiag(\mathbf{A})$  and  $Bdiag(\mathbf{F})$  due to the possible creation of nonzeros outside  $Env(Bdiag(\mathbf{A}))$  during the factorization.

#### 7.4.2 Characterization of the Block Diagonal Envelope via Reachable Sets

Recall from Chapter 4 that the envelope structure of a matrix  $\mathbf{A}$  is characterized by the column subscripts

$$f_i(\mathbf{A}) = \min\{j \mid a_{ij} \neq 0\}, \quad 1 \leq i \leq n.$$

In terms of the associated graph  $\mathcal{G}^{\mathbf{A}} = (X^{\mathbf{A}}, E^{\mathbf{A}})$ , where  $X^{\mathbf{A}} = \{x_1, \dots, x_n\}$ , these numbers are given by

$$f_i(\mathbf{A}) = \min\{s \mid x_s \in Adj(x_i) \cup \{x_i\}\}. \quad (7.4.3)$$

In this subsection, we shall study the envelope structure of  $Bdiag(\mathbf{F})$  by relating the first nonzero column subscript with the corresponding graph structure.

Let  $\mathcal{G}^{\mathbf{A}} = (X^{\mathbf{A}}, E^{\mathbf{A}})$  and  $\mathcal{G}^{\mathbf{F}} = (X^{\mathbf{F}}, E^{\mathbf{F}})$  be the undirected graphs associated with the symmetric matrices  $\mathbf{A}$  and  $\mathbf{F}$  respectively. Let  $\mathcal{P} = \{Y_1, Y_2, \dots, Y_p\}$  be the set partitioning of  $X^{\mathbf{A}}$  that corresponds to the matrix partitioning of  $\mathbf{A}$ . It is useful to note that

$$\mathcal{G}^{\mathbf{A}_{kk}} = \mathcal{G}^{\mathbf{A}}(Y_k),$$

$$\mathcal{G}^{\mathbf{F}_{kk}} = \mathcal{G}^{\mathbf{F}}(Y_k),$$

and

$$\mathcal{G}^{Bdiag(\mathbf{F})} = (X^{\mathbf{A}}, E^{Bdiag(\mathbf{F})})$$

where

$$E^{Bdiag(\mathbf{F})} = \bigcup \{E^{\mathbf{F}}(Y_k) \mid 1 \leq k \leq p\}.$$

In what follows, we shall use  $f_i$  to stand for  $f_i(Bdiag(\mathbf{F}))$ . Let row  $i$  belong to the  $k$ -th block in the partitioning. In other words, we let  $x_i \in Y_k$ . In terms of the filled graph, the quantity  $f_i$  is given by

$$f_i = \min\{s \mid s = i \text{ or } \{x_s, x_i\} \in E^{\mathbf{F}}(Y_k)\}.$$

We now relate it to the original graph  $\mathcal{G}^{\mathbf{A}}$  through the use of reachable sets introduced in Section 5.2.2. By Theorem 5.2.2 which characterizes the fill via reachable sets, we have

$$f_i = \min\{s \mid x_s \in Y_k, x_i \in Reach(x_s, \{x_1, \dots, x_{s-1}\}) \cup \{x_s\}\} \quad (7.4.4)$$

In Theorem 7.4.2, we prove a stronger result.

**Lemma 7.4.1** *Let  $x_i \in Y_k$ , and let*

$$S = Y_1 \cup \dots \cup Y_{k-1}.$$

*That is,  $S$  contains all the nodes in the first  $k - 1$  blocks. Then*

$$x_i \in Reach(x_{f_i}, S) \cup \{x_{f_i}\}.$$

**Proof:** By definition of  $f_i$ ,  $\{x_i, x_{f_i}\} \in E^{\mathbf{F}}$  so that by Theorem 5.2.2,  $x_i \in Reach(x_{f_i}, \{x_1, \dots, x_{f_i-1}\})$ . We can then find a path  $x_i, x_{r_1}, \dots, x_{r_t}, x_{f_i}$  where  $\{x_{r_1}, \dots, x_{r_t}\} \subset \{x_1, \dots, x_{f_i-1}\}$ .

We now prove that  $x_i$  can also be reached from  $x_{f_i}$  through  $S$ , which is a subset of  $\{x_1, \dots, x_{f_i-1}\}$ . If  $t = 0$ , clearly  $x_i \in Reach(x_{f_i}, S)$ . On the other hand, if  $t \neq 0$ , let  $x_{r_s}$  be the node with the largest index number in  $\{x_{r_1}, \dots, x_{r_t}\}$ . Then  $x_i, x_{r_1}, \dots, x_{r_{s-1}}, x_{r_s}$  is a path from  $x_i$  to  $x_{r_s}$  through  $\{x_1, x_2, \dots, x_{r_s-1}\}$  so that

$$\{x_i, x_{r_s}\} \in E^{\mathbf{F}}.$$

But  $r_s < f_i$ , so by the definition of  $f_i$  we have  $x_{r_s} \notin Y_k$ , or in other words  $x_{r_s} \in S$ . The choice of  $r_s$  implies

$$\{x_{r_1}, \dots, x_{r_t}\} \subset S$$

and thus  $x_i \in Reach(x_{f_i}, S)$ . □

**Theorem 7.4.2** *Let  $x_i \in Y_k$  and  $S = Y_1 \cup \dots \cup Y_{k-1}$ . Then*

$$f_i = \min\{s \mid x_s \in Y_k, x_i \in \text{Reach}(x_s, S) \cup \{x_s\}\}.$$

**Proof:** By Lemma 7.4.1, it remains to show that  $x_i \notin \text{Reach}(x_r, S)$  for  $x_r \in Y_k$  and  $r < f_i$ . Assume for contradiction that we can find  $x_r \in Y_k$  with  $r < f_i$  and  $x_i \in \text{Reach}(x_r, S)$ . Since

$$S \subset \{x_1, \dots, x_{r-1}\},$$

we have  $x_i \in \text{Reach}(x_r, \{x_1, \dots, x_{r-1}\})$  so that  $\{x_i, x_r\} \in E^{\mathbf{F}}(Y_k)$ . This contradicts the definition of  $f_i$ .  $\square$

**Corollary 7.4.3** *Let  $x_i$  and  $S$  be as in Theorem 7.4.2. Then*

$$f_i = \min\{s \mid x_s \in \text{Reach}(x_i, S) \cup \{x_i\}\}.$$

**Proof:** It follows from Theorem 7.4.2 and the symmetry of the “Reach” operator.  $\square$

It is interesting to compare this result with that given by (7.4.4). To illustrate the result, we consider the partitioned matrix example in Figure 7.4.1.

Consider  $Y_2 = \{x_5, x_6, x_7, x_8\}$ . Then the associated set  $S$  is  $\{x_1, x_2, x_3, x_4\}$ . We have

$$\begin{aligned} \text{Reach}(x_5, S) &= \{x_{10}, x_{11}\} \\ \text{Reach}(x_6, S) &= \{x_7, x_8, x_9, x_{10}\} \\ \text{Reach}(x_7, S) &= \{x_6, x_8\} \\ \text{Reach}(x_8, S) &= \{x_6, x_7, x_{10}, x_{11}\}. \end{aligned}$$

By Corollary 7.4.3,

$$\begin{aligned} f_5(\text{Bdiag}(\mathbf{F})) &= 5 \\ f_6(\text{Bdiag}(\mathbf{F})) &= f_7(\text{Bdiag}(\mathbf{F})) = f_8(\text{Bdiag}(\mathbf{F})) = 6. \end{aligned}$$

### 7.4.3 An Algorithm and Subroutines for Finding Diagonal Block Envelopes

Corollary 7.4.3 readily provides a method for finding  $f_i(\text{Bdiag}(\mathbf{F}))$  and hence the envelope structure of  $\text{Bdiag}(\mathbf{F})$ . However, in the actual implementation, Lemma 7.4.1 is more easily applied. The algorithm can be described as follows.

Let  $\mathcal{P} = \{Y_1, \dots, Y_p\}$  be the partitioning. For each block  $k$  in the partitioning, do the following:



**Step 1** (Initialization)  $S = Y_1 \cup \dots \cup Y_{k-1}$ ,  $T = S \cup Y_k$ .

**Step 2** (Main loop) For each node  $x_r$  in  $Y_k$  do:

2.1) Determine  $Reach(x_r, S)$  in the subgraph  $\mathcal{G}(T)$ .

2.2) For each  $x_i \in Reach(x_r, S)$ , set  $f_i = r$ .

2.3) Reset  $T \leftarrow T - (Reach(x_r, S) \cup \{x_r\})$ .

The implementation of this algorithm consists of two subroutines, which are discussed below.

#### REACH (find REACHable sets)

Given a set  $S$  and a node  $x \notin S$  in a graph. To study the reachable set  $Reach(x, S)$ , it is helpful to introduce the related notion of neighborhood set. Formally, the *neighborhood set of  $x$  in  $S$*  is defined to be

$$nbrhd(x, S) = \{s \in S \mid s \text{ is reachable from } x \text{ through a subset of } S\}$$

Reachable and neighborhood sets are related by the following lemma.

#### Lemma 7.4.4

$$Reach(x, S) = Adj(nbrhd(x, S) \cup \{x\}).$$

The subroutine REACH applies this simple relation to determine the reachable set of a node via a subset in a given *subgraph*. The subgraph is specified by the input parameters XADJ, ADJNCY and MARKER, where a node belongs to the subgraph if its MARKER value is zero. The subset  $S$  is specified by the mask vector SMASK, where a node belongs to  $S$  if its SMASK value is nonzero. The variable ROOT is the input node, whose reachable set is to be determined.

It returns the reachable set in (RCHSZE, RCHSET). As a by-product, the neighborhood set in (NHDSZE, NBRHD) is also returned. On exit, nodes in the reachable or neighborhood sets will have their MARKER value set to ROOT. After initialization, the subroutine loops through the neighbors of the given ROOT. Neighbors not in the subset  $S$  are included in the reach set, while neighbors in the subset  $S$  are put into the neighborhood set. Furthermore, each neighbor in the subset  $S$  is examined to obtain new reachable nodes. This process is repeated until no neighbors in  $S$  can be found.

---

```

1. C*****
2. C*****
3. C*****      REACH . . . . REACHABLE SET      *****
4. C*****
5. C*****
6. C
7. C      PURPOSE - THIS SUBROUTINE IS USED TO DETERMINE THE
8. C      REACHABLE SET OF A NODE Y THROUGH A SUBSET S
9. C      (I.E. REACH(Y,S) ) IN A GIVEN SUBGRAPH. MOREOVER,
10. C      IT RETURNS THE NEIGHBORHOOD SET OF Y IN S, I.E.
11. C      NBRHD(Y,S), THE SET OF NODES IN S THAT CAN BE
12. C      REACHED FROM Y THROUGH A SUBSET OF S.
13. C
14. C      INPUT PARAMETERS -
15. C      ROOT - THE GIVEN NODE NOT IN THE SUBSET S.
16. C      (XADJ, ADJNCY) - THE ADJACENCY STRUCTURE PAIR.
17. C      SMASK - THE MASK VECTOR FOR THE SET S.
18. C      = 0, IF THE NODE IS NOT IN S,
19. C      > 0, IF THE NODE IS IN S.
20. C
21. C      OUTPUT PARAMETERS -
22. C      (NHDSZE, NBRHD) - THE NEIGHBORHOOD SET.
23. C      (RCHSZE, RCHSET) - THE REACHABLE SET.
24. C
25. C      UPDATED PARAMETERS -
26. C      MARKER - THE MARKER VECTOR USED TO DEFINE THE SUBGRAPH,
27. C      NODES IN THE SUBGRAPH HAVE MARKER VALUE 0.
28. C      ON RETURN, THE REACHABLE AND NEIGHBORHOOD NODE
29. C      SETS HAVE THEIR MARKER VALUES RESET TO ROOT.
30. C
31. C*****
32. C
33. C      SUBROUTINE REACH ( ROOT, XADJ, ADJNCY, SMASK, MARKER,
34. C      1      RCHSZE, RCHSET, NHDSZE, NBRHD )
35. C
36. C*****
37. C
38. C      INTEGER ADJNCY(1), MARKER(1), NBRHD(1), RCHSET(1),
39. C      1      SMASK(1)
40. C      INTEGER XADJ(1), I, ISTOP, ISTRT, J, JSTOP, JSTRT,
41. C      1      NABOR, NBR, NHDBEG, NHDPTR, NHDSZE, NODE,
42. C      1      RCHSZE, ROOT
43. C
44. C*****
45. C
46. C      -----
47. C      INITIALIZATION ...

```



```

48. C -----
49. NHDSZ = 0
50. RCHSZ = 0
51. IF ( MARKER(ROOT) .GT. 0 ) GO TO 100
52. RCHSZ = 1
53. RCHSET(1) = ROOT
54. MARKER(ROOT) = ROOT
55. 100 ISTRT = XADJ(ROOT)
56. ISTOP = XADJ(ROOT+1) - 1
57. IF ( ISTOP .LT. ISTRT ) RETURN
58. C -----
59. C LOOP THROUGH THE NEIGHBORS OF ROOT ...
60. C -----
61. DO 600 I = ISTRT, ISTOP
62. NABOR = ADJNCY(I)
63. IF ( MARKER(NABOR) .NE. 0 ) GO TO 600
64. IF ( SMASK(NABOR) .GT. 0 ) GO TO 200
65. C -----
66. C NABOR IS NOT IN S, INCLUDE IT IN THE REACH SET.
67. C -----
68. RCHSZ = RCHSZ + 1
69. RCHSET(RCHSZ) = NABOR
70. MARKER(NABOR) = ROOT
71. GO TO 600
72. C -----
73. C NABOR IS IN SUBSET S, AND HAS NOT BEEN CONSIDERED.
74. C INCLUDE IT INTO THE NBRHD SET AND FIND THE NODES
75. C REACHABLE FROM ROOT THROUGH THIS NABOR.
76. C -----
77. 200 NHDSZ = NHDSZ + 1
78. NBRHD(NHDSZ) = NABOR
79. MARKER(NABOR) = ROOT
80. NHDBEG = NHDSZ
81. NHDPTR = NHDSZ
82. 300 NODE = NBRHD(NHDPTR)
83. JSTRT = XADJ(NODE)
84. JSTOP = XADJ(NODE+1) - 1
85. DO 500 J = JSTRT, JSTOP
86. NBR = ADJNCY(J)
87. IF ( MARKER(NBR) .NE. 0 ) GO TO 500
88. IF ( SMASK(NBR) .EQ. 0 ) GO TO 400
89. NHDSZ = NHDSZ + 1
90. NBRHD(NHDSZ) = NBR
91. MARKER(NBR) = ROOT
92. GO TO 500
93. 400 RCHSZ = RCHSZ + 1
94. RCHSET(RCHSZ) = NBR

```

```

95 .                MARKER(NBR) = ROOT
96 .   500          CONTINUE
97 .                NHDPTR = NHDPTR + 1
98 .                IF ( NHDPTR .LE. NHDSZE ) GO TO 300
99 .   600          CONTINUE
100 .              RETURN
101 .              END

```

---

### **FNBENV (FiNd diagonal Block ENVELOpe)**

This subroutine serves the same purpose as `FNTENV` in Section 6.5.3. They are both used to determine the envelope structure of the factored diagonal blocks in a partitioned matrix. Unlike `FNTENV`, this subroutine `FNBENV` finds the *exact* envelope structure. Although it works for general partitioned matrices, it is more expensive to use than `FNTENV`, and for the orderings provided by the RQT algorithm, the output from `FNTENV` is satisfactory. However, for one-way dissection orderings the more sophisticated `FNBENV` is essential.

Inputs to `FNBENV` are the adjacency structure (`XADJ`, `ADJNCY`), the ordering (`PERM`, `INVP`) and the partitioning (`NBLKS`, `XBLK`). The subroutine will produce the envelope structure in the index vector `XENV` and the variable `MAXENV` will contain the size of the envelope.

Three temporary vectors are required. The vector `SMASK` is used to specify those nodes in the subset  $S$  (see the above algorithm). On the other hand, the nodes in the set  $T$  are given by those with `MARKER` value 0. The vector `MARKER` is also used temporarily to store the first neighbor in each row of a block. The third temporary vector `RCHSET` is used to contain both the reachable and neighborhood sets. Since the two sets do not overlap, we can organize the vector `RCHSET` as follows.

The subroutine begins by initializing the temporary vectors `SMASK` and `MARKER`. The main loop goes through and processes each block. For each block, its nodes are added to the subgraph by turning their `MARKER` values to zeros. For each node  $i$  in the block, the subroutine `REACH` is called so that nodes in the thus-determined reachable sets will have node  $i$  as their first neighbor. Before the next block is processed, the `MARKER` values are reset and nodes in the current block are added to the subset  $S$ .

---

```

1. C*****
2. C*****
3. C*****      FNBENV . . . . FIND DIAGONAL BLOCK ENVELOPE *****

```

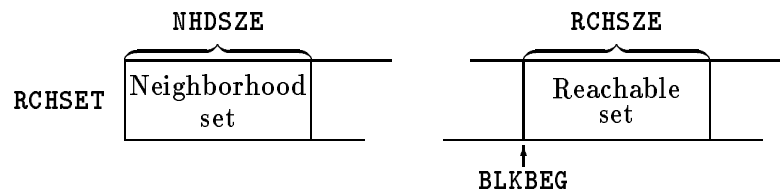


Figure 7.4.2: Organization of the RCHSET array.

```

4. C*****
5. C*****
6. C
7. C   PURPOSE - THIS SUBROUTINE FINDS THE EXACT ENVELOPE
8. C     STRUCTURE OF THE DIAGONAL BLOCKS OF THE CHOLESKY
9. C     FACTOR OF A PERMUTED PARTITIONED MATRIX.
10. C
11. C   INPUT PARAMETERS -
12. C     (XADJ, ADJNCY) - ADJACENCY STRUCTURE OF THE GRAPH.
13. C     (PERM, INVP) - THE PERMUTATION VECTOR AND ITS INVERSE.
14. C     (NBLKS, XBLK) - THE PARTITIONING.
15. C
16. C   OUTPUT PARAMETERS -
17. C     XENV - THE ENVELOPE INDEX VECTOR.
18. C     ENVSZ - THE SIZE OF THE ENVELOPE.
19. C
20. C   WORKING PARAMETERS -
21. C     SMASK - MARKS NODES THAT HAVE BEEN CONSIDERED.
22. C     MARKER - IS USED BY ROUTINE REACH.
23. C     RCHSET - IS USED BY THE SUBROUTINE REACH.
24. C     STORES BOTH REACHABLE AND NEIGHBORHOOD SETS.
25. C
26. C   PROGRAM SUBROUTINES -
27. C     REACH.
28. C
29. C*****
30. C
31. C   SUBROUTINE FNBENV ( XADJ, ADJNCY, PERM, INVP, NBLKS, XBLK,
32. C     1           XENV, ENVSZ, SMASK, MARKER, RCHSET )
33. C
34. C*****
35. C
36. C     INTEGER ADJNCY(1), INVP(1), MARKER(1), PERM(1),
37. C     1           RCHSET(1), SMASK(1), XBLK(1)

```

```

38.          INTEGER XADJ(1), XENV(1), BLKBEG, BLKEND, I,
39.          1          IFIRST, INHD, K, ENVSZE, NBLKS, NEQNS,
40.          1          NEWNHD, NHDSZE, NODE, RCHSZE
41. C
42. C*****
43. C
44. C          -----
45. C          INITIALIZATION ...
46. C          -----
47.          NEQNS = XBLK(NBLKS+1) - 1
48.          ENVSZE = 1
49.          DO 100 I = 1, NEQNS
50.              SMASK(I) = 0
51.              MARKER(I) = 1
52.          100    CONTINUE
53. C          -----
54. C          LOOP OVER THE BLOCKS ...
55. C          -----
56.          DO 700 K = 1, NBLKS
57.              NHDSZE = 0
58.              BLKBEG = XBLK(K)
59.              BLKEND = XBLK(K+1) - 1
60.              DO 200 I = BLKBEG, BLKEND
61.                  NODE = PERM(I)
62.                  MARKER(NODE) = 0
63.          200    CONTINUE
64. C          -----
65. C          LOOP THROUGH THE NODES IN CURRENT BLOCK ...
66. C          -----
67.          DO 300 I = BLKBEG, BLKEND
68.              NODE = PERM(I)
69.              CALL REACH ( NODE, XADJ, ADJNCY, SMASK,
70.          1              MARKER, RCHSZE, RCHSET(BLKBEG),
71.          1              NEWNHD, RCHSET(NHDSZE+1) )
72.              NHDSZE = NHDSZE + NEWNHD
73.              IFIRST = MARKER(NODE)
74.              IFIRST = INVP(IFIRST)
75.              XENV(I) = ENVSZE
76.              ENVSZE = ENVSZE + I - IFIRST
77.          300    CONTINUE
78. C          -----
79. C          RESET MARKER VALUES OF NODES IN NBRHD SET.
80. C          -----
81.          IF ( NHDSZE .LE. 0 ) GO TO 500
82.          DO 400 INHD = 1, NHDSZE
83.              NODE = RCHSET(INHD)
84.              MARKER(NODE) = 0

```

```

85.   400           CONTINUE
86.   C           -----
87.   C           RESET MARKER AND SMASK VALUES OF NODES IN
88.   C           THE CURRENT BLOCK.
89.   C           -----
90.   500           DO 600 I = BLKBEG, BLKEND
91.                   NODE = PERM(I)
92.                   MARKER(NODE) = 0
93.                   SMASK(NODE) = 1
94.   600           CONTINUE
95.   700           CONTINUE
96.                   XENV(NEQNS+1) = ENVSIZE
97.                   ENVSIZE = ENVSIZE - 1
98.                   RETURN
99.           END

```

---

#### 7.4.4 Execution Time Analysis of the Algorithm

For general partitioned matrices, the complexity of the diagonal block envelope algorithm depends on the partitioning factor  $p$ , the sparsity of the matrix and the way blocks are connected. However, for one-way dissection partitionings, we have the following result.

**Theorem 7.4.5** *Let  $\mathcal{G} = (X, E)$  and  $\mathcal{P} = \{Y_1, \dots, Y_p\}$  be a one-way dissection partitioning. The complexity of the algorithm **FNBENV** is  $O(|E|)$ .*

**Proof:** For a node  $x_i$  in the first  $p - 1$  blocks, the subroutine **REACH**, when called, merely looks through the adjacency list for the node  $x_i$ . On the other hand, when nodes in the last block  $Y_p$  are processed, the adjacency lists for all the nodes in the graph are inspected at most once. Hence, in the entire algorithm, the adjacency structure is gone through at most twice.  $\square$

#### Exercises

- 7.4.1) Construct an example of a tree-partitioned matrix structure  $\mathbf{A}$  to show that **FNTENV** is not adequate to determine the *exact* envelope structure of the block diagonal matrix  $Bdiag(\mathbf{F})$ , where  $\mathbf{F}$  is the filled matrix of  $\mathbf{A}$ .
- 7.4.2) Give an example to show that Theorem 7.4.5 does not hold for all tree-partitionings  $\mathcal{P}$ .

- 7.4.3) This question involves solving a sequence of  $st$  by  $st$  finite element matrix problems  $\mathbf{Ax} = \mathbf{b}$  of the type studied in Section 7.3, with  $m = 5, 10, 15,$  and  $20,$  and  $t = 2m.$  Set the diagonal elements of  $\mathbf{A}$  to 8, the off-diagonal elements to  $-1,$  and arrange the right hand side  $\mathbf{b}$  so that the solution to the system is a vector of all ones. Use the programs provided in Chapter 4 to solve these problems, taking care to record the storage required and the execution times for each phase of the solution of each problem. Repeat the procedure using the one-way dissection ordering subroutines provided in this chapter, along with the appropriate subroutines from Chapter 6. Compare the two methods for solving these problems with respect to the criteria discussed in Section 2.4 of Chapter 2.

## 7.5 Additional Notes

It is interesting to speculate about more sophisticated ways of choosing the one-way dissectors. For example, instead of using a fixed  $\delta,$  one might instead use a sequence  $\delta_i, i = 1, 2, \dots,$  where  $\delta_i$  is obtained from local information about the part of the level structure that remains to be processed after the first  $i - 1$  dissectors have been chosen. Investigations such as these, aimed at the development of robust heuristics, are good candidates for senior projects and masters theses.

The fundamental idea that makes the one-way dissection method effective is the use of the “throw-away” technique introduced in Section 6.3. This technique can be recursively applied, as described in the *additional notes* at the end of Chapter 6, which implies that the one-way dissection scheme of this chapter may also be similarly generalized. In its simplest form the idea is to also apply the one-way dissection technique to the  $\sigma + 1$  independent blocks, rather than ordering them using the RCM algorithm. The basic approach for this two-level scheme is depicted in Figure 7.5.1.

Of course the idea can be generalized to more than two levels, but apparently in practice using more than two levels does not yield significant benefit. It can be shown that for an  $k \times k$  grid problem ( $s = t = k$ ), if the optimal  $\sigma_1$  and  $\sigma_2$  are chosen, the storage and operation counts for this two level ordering are  $O(k^{7/3})$  and  $O(k^{10/3})$  respectively, compared to  $O(k^{5/2})$  and  $O(k^{7/2})$  for the ordinary (one-level) one-way dissection scheme as described in this chapter (Ng [42]).

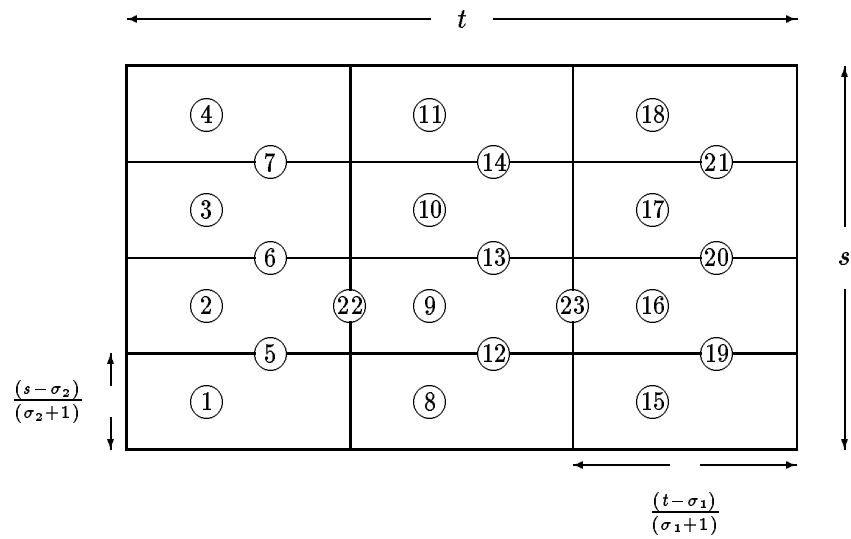


Figure 7.5.1: A two-level one-way dissection ordering, having  $\sigma_1 = 2$  level-1 dissectors,  $\sigma_2 = 3$  level-2 dissectors, and  $(\sigma_1 + 1)(\sigma_2 + 1) = 12$  independent blocks, which are numbered grid column by column.

---

## Chapter 8

# Nested Dissection Methods

### 8.1 Introduction

In Chapter 7, we have studied the so-called one-way dissection method, and we have seen that it lends itself readily to the implicit tree-partitioning scheme of Chapter 6. In this chapter, we consider a different dissection method, which attempts to minimize fill, just as the minimum degree algorithm described in Chapter 5 attempts to do.

The nested dissection for matrix problems arising in finite difference and finite element applications. The main advantage of the algorithm of Section 8.3, compared to the minimum degree algorithm, is its speed, and its modest and predictable storage requirements. The orderings produced are similar in nature to those provided by the minimum degree algorithm, and for this reason we do not deal with a storage scheme, allocation procedure, or numerical subroutines in this chapter. Those of Chapter 5 are appropriate for nested dissection orderings.

Separators, which we defined in Section 3.2, play a central role in the study of sparse matrix factorization. Let  $\mathbf{A}$  be a symmetric matrix and  $\mathcal{G}^{\mathbf{A}}$  be its associated undirected graph. Consider a separator  $S$  in  $\mathcal{G}^{\mathbf{A}}$ , whose removal disconnects the graph into two parts whose node sets are  $C_1$  and  $C_2$ .

If the nodes in  $S$  are numbered after those of  $C_1$  and  $C_2$ , this induces a partitioning on the correspondingly ordered matrix and it has the form shown in Figure 8.1.1. The crucial observation is that the zero block in the matrix remains zero after the factorization. Since one of the primary purposes in the study of sparse matrix computation is to preserve as many zero entries as possible, the use of separators in this way is central. When appropriately



chosen, a (hopefully large) submatrix is guaranteed to stay zero. Indeed, the idea can be recursively applied, so that zeros can be preserved in the same manner in the submatrices.

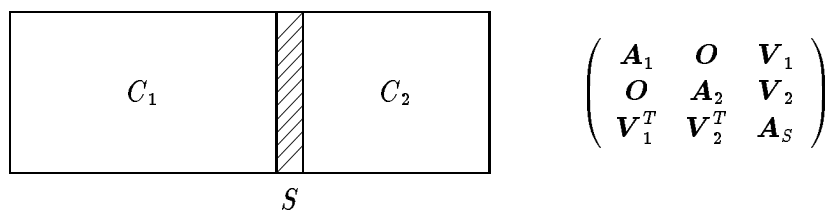


Figure 8.1.1: Use of a separator to partition a matrix.

---

The recursive application of this basic observation has come to be known as the *nested dissection method*. George in 1973 [18] applied this technique to sparse systems associated with an  $s \times s$  regular grid or mesh consisting of  $(s - 1)^2$  small elements. In the next section, we shall give a careful analysis of the method for this special problem.

## 8.2 Nested Dissection of a Regular Grid

### 8.2.1 The Ordering

Let  $X$  be the set of vertices of the  $s \times s$  regular grid. Let  $S^0$  consist of the vertices on a mesh line which as nearly as possible divides  $X$  into two equal parts  $R^1$  and  $R^2$ . Figure 8.2.1 shows the case when  $s = 10$ .

If we number the nodes of the two components  $R^1$  and  $R^2$  row by row, followed by those in  $S^0$ , a matrix structure as shown in Figure 8.2.2 is obtained. Let us call this the *one-level dissection ordering*.

To get a nested dissection ordering, we continue dissecting the remaining two components. Choose vertex sets

$$S^j \subset R^j, \quad j = 1, 2$$

consisting of nodes lying on mesh lines which as nearly as possible divide  $R^j$  into equal parts. If the variables associated with vertices in  $R^j - S^j$  are num-

---

86	87	88	89	90	100	40	39	38	37
81	82	83	84	85	99	36	35	34	33
76	77	78	79	80	98	32	31	30	29
71	72	73	74	75	97	28	27	26	25
66	67	68	69	70	96	24	23	22	21
61	62	63	64	65	95	20	19	18	17
56	57	58	59	60	94	16	15	14	13
51	52	53	54	55	93	12	11	10	9
46	47	48	49	50	92	8	7	6	5
41	42	43	44	45	91	4	3	2	1

Figure 8.2.1: A one-level dissection ordering of a 10 by 10 grid.

---

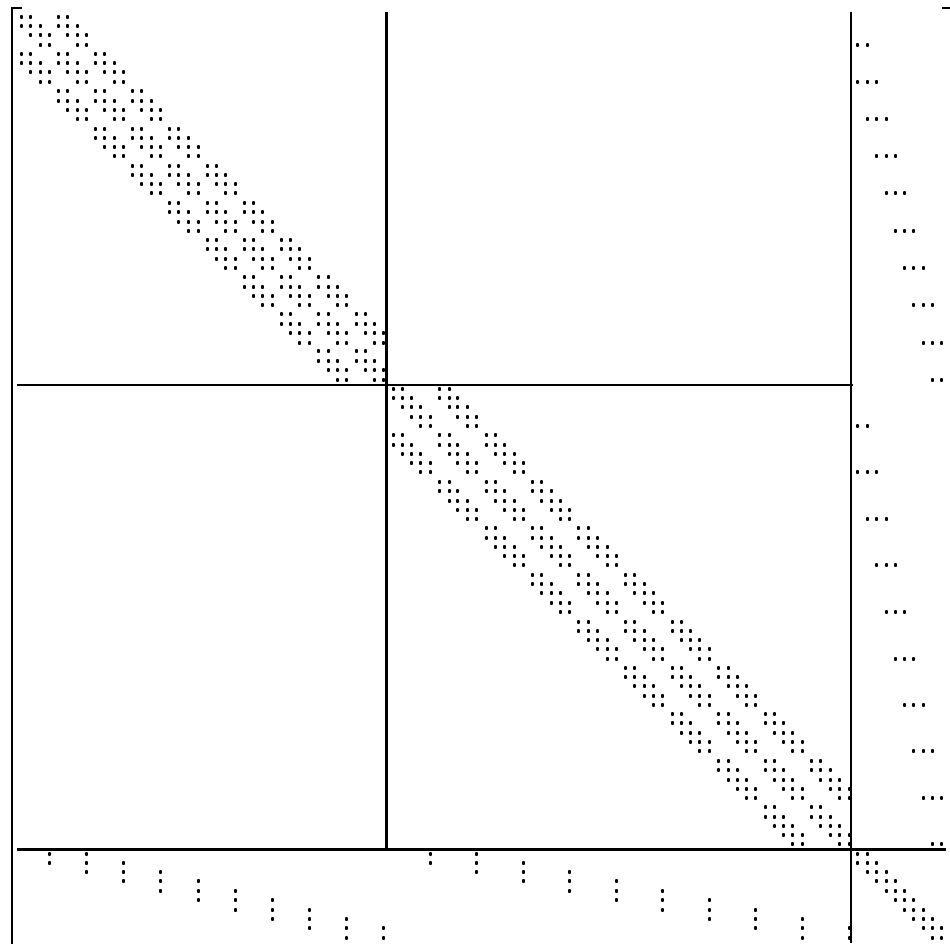


Figure 8.2.2: Matrix structure associated with a one-level dissection ordering.

---

bered before those associated with  $S^j$ , we induce in the two leading principal submatrices exactly the same structure as that of the overall matrix.

---

78	77	85	68	67	100	29	28	36	20
76	75	84	66	65	99	27	26	35	19
80	79	83	70	69	98	31	30	34	21
74	73	82	64	63	97	25	24	33	18
72	71	81	62	61	96	23	22	32	17
90	89	88	87	86	95	40	39	38	37
54	53	60	46	45	94	10	9	16	3
52	51	59	44	43	93	8	7	15	2
56	55	58	48	47	92	12	11	14	4
50	49	57	42	41	91	6	5	13	1

Figure 8.2.3: A nested dissection ordering of a 10 by 10 grid.

The process can be repeated until the components left are not dissectable. This yields a *nested dissection ordering*. Figure 8.2.3 shows such an ordering on the 10 by 10 grid problem and Figure 8.2.4 shows the correspondingly ordered matrix structure. Note the recursive pattern in the matrix structure.

### 8.2.2 Storage Requirements

Nested dissection employs a strategy commonly known as *divide and conquer*. The strategy splits a problem into smaller subproblems whose individual solutions can be combined to yield the solution to the original problem. Moreover, the subproblems have structures similar to the original one so that the process can be repeated recursively until the solutions to the subproblems are trivial.

In the study of such strategies, some forms of recursive equations need to be solved. We now provide some results in preparation for the analysis of the

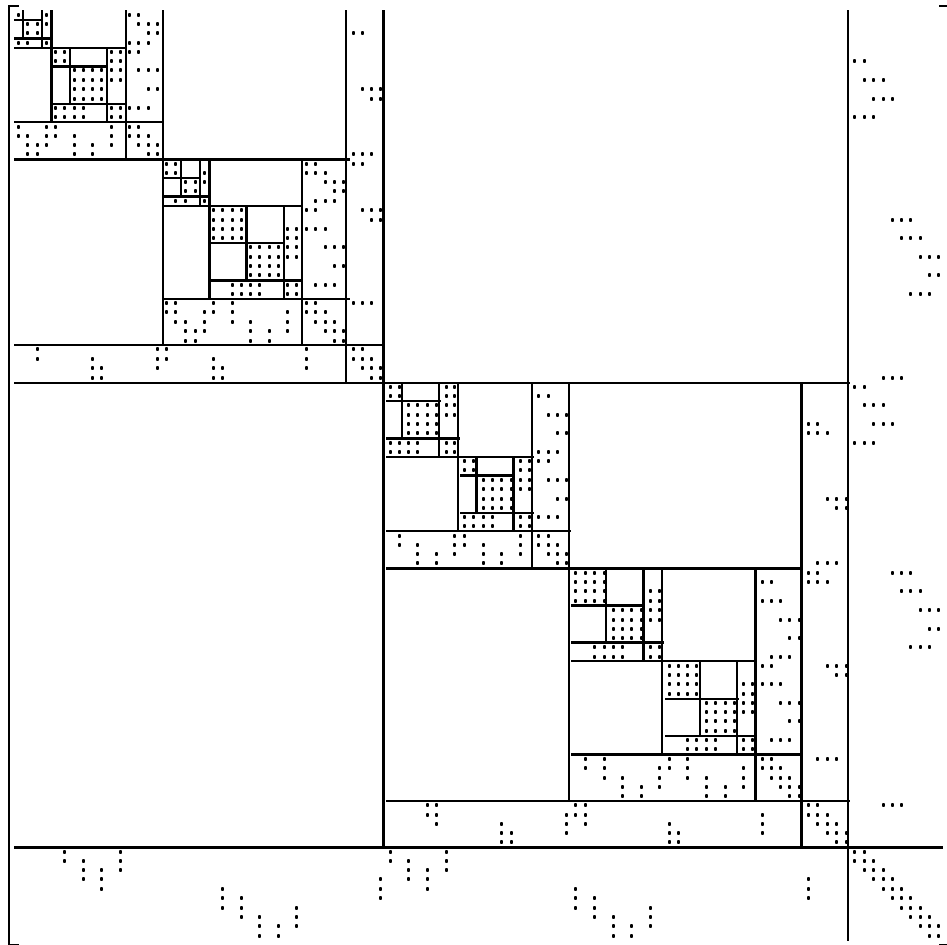


Figure 8.2.4: Matrix structure associated with a nested dissection ordering.

---

storage requirement for nested dissection orderings. The proofs of these are left as exercises.

**Lemma 8.2.1** *Let  $f(s) = 4f(s/2) + ks^2 + O(s)$ . Then*

$$f(s) = ks^2 \log_2 s + O(s^2).$$

**Lemma 8.2.2** *Let  $g(s) = g(s/2) + ks^2 \log_2 s + O(s^2)$ . Then*

$$g(s) = \frac{4}{3}ks^2 \log_2 s + O(s^2).$$

**Lemma 8.2.3** *Let  $h(s) = 2h(s/2) + ks^2 \log_2 s + O(s^2)$ . Then*

$$h(s) = 2ks^2 \log_2 s + O(s^2).$$

In order to give an analysis of the nested dissection orderings recursively, we introduce bordered  $s \times s$  grids. A *bordered  $s \times s$  grid* contains an  $s \times s$  subgrid, where one or more sides of this subgrid is bordered by an additional grid line. Figure 8.2.5 contains some examples of bordered 3 by 3 grids.

We are now ready to analyze the storage requirement for the nested dissection ordering. Let  $S(s, i)$  be the number of nonzeros in the factor of a matrix associated with an  $s \times s$  grid ordered by nested dissection, where the grid is bordered along  $i$  sides. Clearly, what we are after is the quantity  $S(s, 0)$ . For our purposes, when  $i = 2$ , we always refer to the one as shown in Figure 8.2.5(c), rather than the grid in Figure 8.2.6:

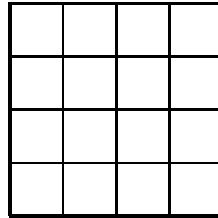
In what follows, we relate the quantities  $S(s, i)$ ,  $0 \leq i \leq 4$ . Consider first  $S(s, 0)$ . In Figure 8.2.7, a “+” shaped separator is used to divide the  $s \times s$  grid into 4 smaller subgrids. The variables in regions  $\boxed{1}$ ,  $\boxed{2}$ ,  $\boxed{3}$  and  $\boxed{4}$  are to be numbered before those in  $\boxed{5}$  so that a matrix structure of the form in Figure 8.2.8 is induced.

The number of nonzeros in the factor comes from the  $L_{ii}$ 's ( $1 \leq i \leq 4$ ) and the  $L_{5i}$  for  $1 \leq i \leq 5$ . Now since the strategy is applied recursively on the smaller subgrids, we have

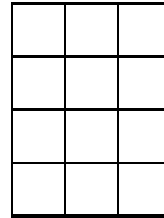
$$\eta(L_{ii}) + \eta(L_{5i}) \approx S(s/2, 2)$$

for  $1 \leq i \leq 4$ . As for  $L_{55}$  which corresponds to the nodes in the “+” separator, we can determine the number of nonzeros using Theorem 5.2.2. It is given by

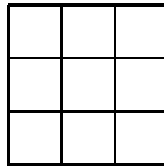
$$\eta(L_{55}) = 2 \sum_{i=s}^{3s/2} i + s^2/2 + O(s) = 7s^2/4 + O(s).$$



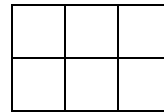
(a)



(b)



(c)



(d)

Figure 8.2.5: Some bordered 3 by 3 grids.

---

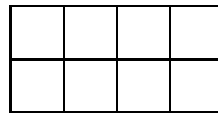
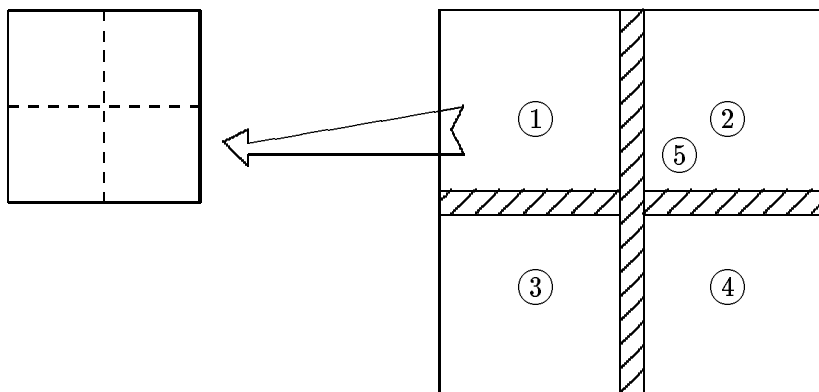


Figure 8.2.6: A different type of bordered 3 by 3 grid.

---

Figure 8.2.7: Dissection of an  $s$  by  $s$  grid.

Thus, we obtain the first recursion equation:

$$S(s, 0) = 4S\left(\frac{s}{2}, 2\right) + \frac{7}{4}s^2 + O(s). \quad (8.2.1)$$

The other recursion equations can be established in the same way. In general, it can be expressed as

$$S(s, i) = \text{cost to store the 4 bordered } s/2 \times s/2 \text{ subgrids} + \\ \text{cost to store the " + "separator.}$$

We leave it to the reader to verify the following results.

$$S(s, 2) = S(s/2, 2) + 2S(s/2, 3) + S(s/2, 4) + 19s^2/4 + O(s) \quad (8.2.2)$$

$$S(s, 3) = 2S(s/2, 3) + 2S(s/2, 4) + 25s^2/4 + O(s) \quad (8.2.3)$$

$$S(s, 4) = 4S(s/2, 4) + 31s^2/4 + O(s). \quad (8.2.4)$$

**Theorem 8.2.4** *The number of nonzeros in the triangular factor  $\mathbf{L}$  of a matrix associated with a regular  $s \times s$  grid ordered by nested dissection is given by*

$$\eta(\mathbf{L}) = 31(s^2 \log_2 s)/4 + O(s^2).$$

**Proof:** The result follows from the recurrence relations (8.2.1)-(8.2.4). Applying Lemma 8.2.1 to equation (8.2.4), we get

$$S(s, 4) = 31(s^2 \log_2 s)/4 + O(s^2),$$



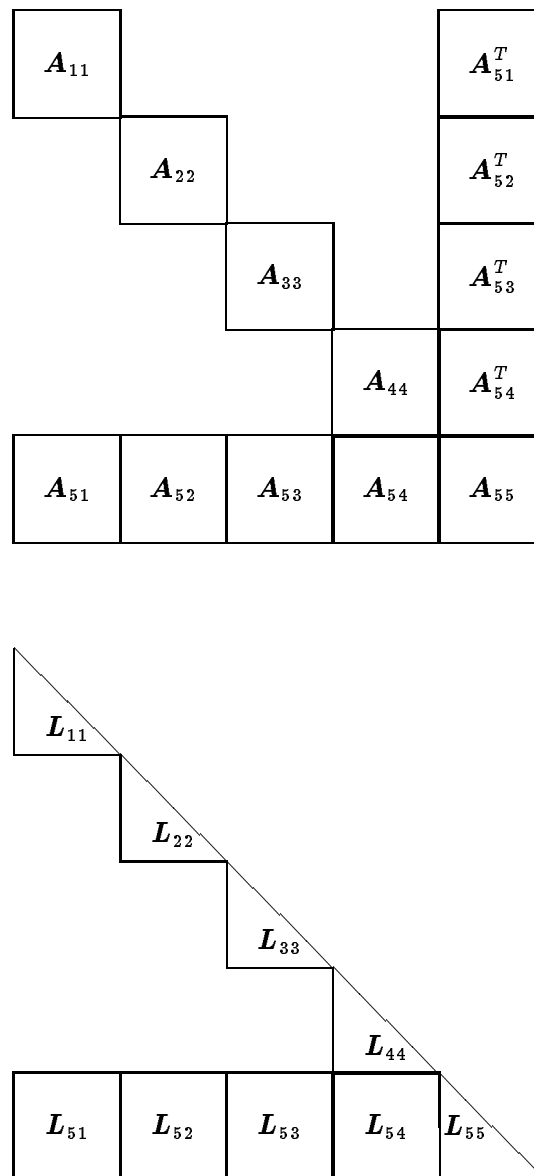


Figure 8.2.8: Matrix structure for dissection in Figure 8.2.7.

---

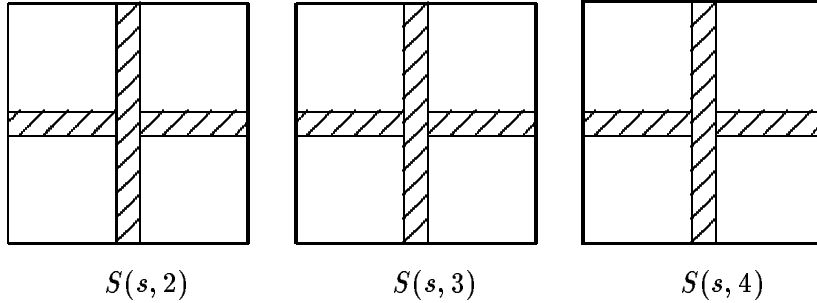


Figure 8.2.9: Illustrations of  $S(s, 2)$ ,  $S(s, 3)$  and  $S(s, 4)$ .

---

so that (8.2.3) becomes

$$S(s, 3) = 2S(s/2, 3) + 31(s^2 \log_2 s)/8 + O(s^2).$$

The solution to it gives, by Lemma 8.2.3,

$$S(s, 3) = 31(s^2 \log_2 s)/4 + O(s^2).$$

Substituting  $S(s, 3)$  and  $S(s, 4)$  into equation (8.2.2), we have

$$S(s, 2) = S(s/2, 2) + 93(s^2 \log_2 s)/16 + O(s^2).$$

Again, the solution is

$$S(s, 2) = 31(s^2 \log_2 s)/4 + O(s^2)$$

so that

$$\eta(\mathbf{L}) = S(s, 0) = 31(s^2 \log_2 s)/4 + O(s^2)$$

□

It is interesting to note from the proof of Theorem 8.2.4 that the asymptotic bounds for  $S(s, i)$ ,  $i = 0, 2, 3, 4$  are all  $31(s^2 \log_2 s)/4$ . (What about  $i = 1$ ?)

### 8.2.3 Operation Counts

Let  $\mathbf{A}$  be a matrix associated with an  $s \times s$  grid ordered by nested dissection. To estimate the number of operations required to factor  $\mathbf{A}$ , we can follow the same approach as used in the previous section. We first state some further results on recursive equations.

**Lemma 8.2.5** *Let  $f(s) = f(s/2) + ks^3 + O(s^2 \log_2 s)$ . Then*

$$f(s) = 8ks^3/7 + O(s^2 \log_2 s).$$

**Lemma 8.2.6** *Let  $g(s) = 2g(s/2) + ks^3 + O(s^2 \log_2 s)$ . Then*

$$g(s) = 4ks^3/3 + O(s^2 \log_2 s).$$

**Lemma 8.2.7** *Let  $h(s) = 4h(s/2) + ks^3 + O(s^2)$ . Then*

$$h(s) = 2ks^3 + O(s^2 \log_2 s).$$

In parallel to  $S(s, i)$ , we introduce  $\theta(s, i)$  to be the number of operations required to factor a matrix associated with an  $s \times s$  grid ordered by nested dissection, where the grid is bordered on  $i$  sides. To determine  $\theta(s, 0)$ , we again consider Figure 8.2.7; clearly  $\theta(s, 0)$  is the cost of eliminating the four  $s/2 \times s/2$  bordered subgrids, together with the cost of eliminating the nodes in the “+” dissector. Applying Theorem 2.2.2, we have

$$\theta(s, 0) \approx 4\theta(s/2, 2) + \sum_{i=s}^{3s/2} i^2 + \frac{1}{2} \sum_{i=1}^s i^2 \quad (8.2.5)$$

$$\begin{aligned} &= 4\theta(s/2, 2) + 19s^3/24 + s^3/6 + O(s^2) \\ &= 4\theta(s/2, 2) + 23s^3/24 + O(s^2). \end{aligned} \quad (8.2.6)$$

We leave it to the reader to verify the following equations:

$$\theta(s, 2) = \theta(s/2, 2) + 2\theta(s/2, 3) + \theta(s/2, 4) + 35s^3/6 + O(s^2) \quad (8.2.7)$$

$$\theta(s, 3) = 2\theta(s/2, 3) + 2\theta(s/2, 4) + 239s^3/24 + O(s^2) \quad (8.2.8)$$

$$\theta(s, 4) = 4\theta(s/2, 4) + 371s^3/24 + O(s^2). \quad (8.2.9)$$

**Theorem 8.2.8** *The number of operations required to factor a matrix associated with an  $s$  by  $s$  grid ordered by nested dissection is given by*

$$829s^3/84 + O(s^2 \log_2 s).$$

**Proof:** All that is required is to determine  $\theta(s, 0)$ . Applying Lemma 8.2.7 to equation (8.2.9), we obtain

$$\theta(s, 4) = 371s^3/12 + O(s^2 \log_2 s).$$

This means equation (8.2.8) can be rewritten as

$$\theta(s, 3) = 2\theta(s/2, 3) + 849s^3/48 + O(s^2 \log_2 s).$$

By Lemma 8.2.6, we have

$$\theta(s, 3) = 283s^3/12 + O(s^2 \log_2 s).$$

Substituting  $\theta(s, 3)$  and  $\theta(s, 4)$  into (8.2.7), we get

$$\theta(s, 2) = \theta(s/2, 2) + 1497s^3/96 + O(s^2 \log_2 s),$$

which is, by Lemma 8.2.5,

$$\theta(s, 2) = 499s^3/28 + O(s^2 \log_2 s).$$

Finally, from equation (8.2.5),

$$\theta(s, 0) = 829s^3/84 + O(s^2 \log_2 s).$$

□

#### 8.2.4 Optimality of the Ordering

In this section, we establish lower bounds on the number of nonzero entries in the factor (primary storage) and the number of operations required to effect the symmetric factorization for *any* ordering of the matrix system associated with an  $s \times s$  regular grid. We show that at least  $O(s^3)$  operations are required for its factorization and the corresponding lower triangular factor must have at least  $O(s^2 \log_2 s)$  nonzero components. The nested dissection ordering described in Section 8.2.1 attains these lower bounds, so that the ordering can be regarded as optimal in the order of magnitude sense.

We first consider the lower bound on operations.

**Lemma 8.2.9** *Let  $\mathcal{G} = (X, E)$  be the graph associated with the  $s \times s$  grid. Let  $x_1, x_2, \dots, x_n$  be any ordering on  $\mathcal{G}$ . Then there exists an  $x_i$  such that*

$$|\text{Reach}(x_i, \{x_1, \dots, x_{i-1}\})| \geq n - 1.$$

**Proof:** Let  $x_i$  be the *first* node to be removed which completely vacates a row or column of the grid. For definiteness, let it be a column {row}. At this stage, there are at least  $(s - 1)$  mesh rows {columns} with uneliminated nodes. At least one in each of these rows {columns} can be reached from  $x_i$  through the subset  $\{x_1, \dots, x_{i-1}\}$ . This proves the lemma. □

**Theorem 8.2.10** *The factorization of a matrix associated with an  $s \times s$  grid requires at least  $O(s^3)$  operations.*

**Proof:** By Lemma 8.2.9, there exists an  $x_i$  such that

$$\text{Reach}(x_i, \{x_1, \dots, x_{i-1}\}) \cup \{x_i\}$$

is a clique of size at least  $s$  in the filled graph  $\mathcal{G}^{\mathbf{F}(\mathbf{A})}$  (see Exercise 5.2.4 on page 116). This corresponds to a full  $s \times s$  submatrix in the filled matrix  $\mathbf{F}$  so that symmetric factorization requires at least  $s^3/6 + O(s^2)$  operations.  $\square$  The proof for the lower bound on primary storage follows a different argument. For each  $k \times k$  subgrid, the following lemma identifies a special edge in the resulting filled graph.

**Lemma 8.2.11** *Consider any  $k \times k$  subgrid in the given  $s \times s$  grid. There exists an edge in  $\mathcal{G}^{\mathbf{F}}$  joining a pair of parallel boundary lines in the subgrid.*

**Proof:** There are four boundary mesh lines in the  $k \times k$  subgrid. Let  $x_i$  be the first boundary node in the subgrid to be removed that completely vacates a boundary line (not including the corner vertices).

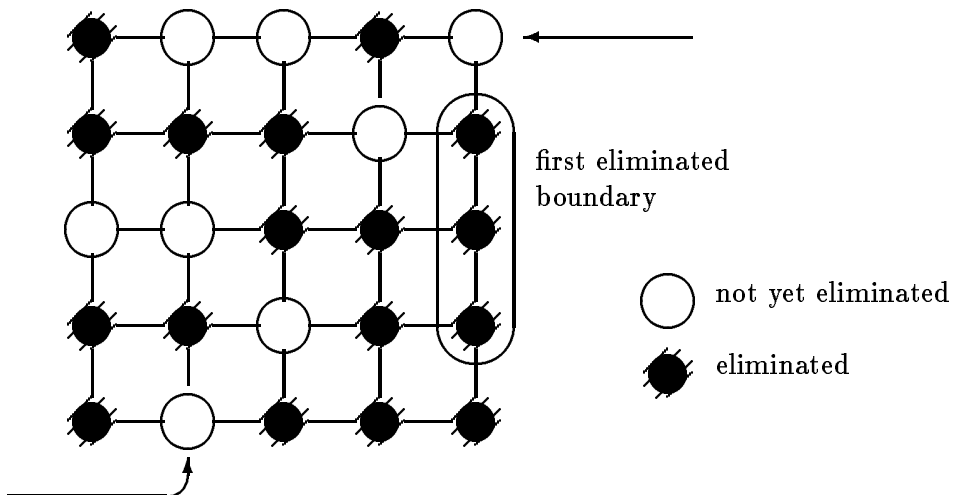


Figure 8.2.10: The status of a grid when the first boundary is eliminated.

---

Then there always exist two nodes in the remaining parallel boundary lines that are linked through

$$\{x_1, x_2, \dots, x_{i-1}, x_i\}.$$

(See the nodes pointed to in the Figure 8.2.10.) In other words, there is an edge joining them in  $\mathcal{G}^F$ .  $\square$

**Theorem 8.2.12** *The triangular factor of a matrix associated with an  $s \times s$  grid has at least  $O(s^2 \log_2 s)$  nonzeros.*

**Proof:** Consider each subgrid of size  $k$ . It follows from Lemma 8.2.11 that there is an edge in  $\mathcal{G}^F$  joining a pair of parallel boundary lines in the subgrid. Each such edge can be chosen for at most  $k$  subgrids of size  $k$ . Since the number of subgrids of size  $k$  is  $(s - k + 1)^2$ , the number of such distinct edges is bounded below by

$$\frac{(s - k + 1)^2}{k}.$$

Futhermore, for subgrids of different sizes, the corresponding edges must be different. So, we have

$$|E^F| \geq \sum_{k=1}^s \frac{(s - k + 1)^2}{k} \approx s^2 \log_2 s.$$

$\square$

## Exercises

8.2.1) Let  $\mathbf{A}$  be the matrix associated with an  $s \times s$  grid, ordered by the one-level dissection scheme. Show that

a) the number of operations required to perform the symmetric factorization is  $\frac{13}{24}s^4 + O(s^3)$

b) the number of nonzeros in the factor  $\mathbf{L}$  is  $s^3 + O(s^2)$ .

8.2.2) Prove the recursive equations in Lemmas 8.2.1-8.2.3 and Lemmas 8.2.5-8.2.7.

8.2.3) In establishing equation (8.2.7) for  $\theta(s, 2)$ , we assumed that the “+” separator is ordered as in (a).

Assume  $\theta'(s, 2)$  is the corresponding cost if (b) is used. Show that

$$\theta'(s, 2) = \theta'(s/2, 2) + 2\theta(s/2, 3) + \theta(s/2, 4) + 125s^3/24 + O(s^2).$$

How does it compare to  $\theta(s, 2)$  ?

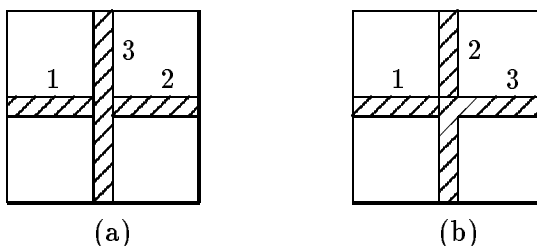


Figure 8.2.11: Different ways of labelling the ‘+’ separator.

---

- 8.2.4) Prove results similar to Theorems 8.2.4 and 8.2.9 for an  $s \times t$  grid where  $s$  is large and  $s < t$ .
- 8.2.5) Prove that any ordering of an  $s \times s$  grid must yield a matrix whose bandwidth is at least  $s - 1$ .
- 8.2.6) Consider the  $s \times s$  grid. It is known that the associated graph  $\mathcal{G} = (X, E)$  satisfies the *isoparametric inequality*: for any subset  $S$ , if  $|S| \leq s^2/2$  then  $|Adj(S)| \geq |S|^{1/2}$ . Prove that any ordering on  $\mathcal{G}$  yields a profile of at least  $O(s^3)$ .
- 8.2.7) Suppose one carries out “incomplete nested dissection” on the  $s \times s$  grid problem (George et al. [22]). That is, one only carries out the dissection  $l$  levels, where  $l < \log_2 s$ , and numbers the remaining independent grid subarrays row by row. Show that if  $l \geq \log_2(\sqrt{s})$  then the operation count for this ordering remains  $O(s^3)$ . Show that the number of nonzeros in the corresponding factor  $L$  is  $O(s^2\sqrt{s})$ .
- 8.2.8) Using a method due to Strassen [52], and extended by Bunch and Hopcroft [5], it is possible to solve a dense  $s \times s$  system of linear equations, and to multiply two dense  $s \times s$  matrices together, in  $O(s^{\log_2 7})$  operations. Using this result, along with modifications to Lemmas 8.2.5-8.2.7, show that the  $s \times s$  grid problem can be solved in  $O(s^{\log_2 7})$  operations, using the nested dissection ordering (Rose [45]).

## 8.3 Nested Dissection of General Problems

### 8.3.1 A Heuristic Algorithm

The optimality of the nested dissection ordering for the  $s \times s$  grid problem has been established in the previous section. The underlying idea of splitting the grid into two pieces of roughly equal size with a small separator is clearly important. In this section, we describe a heuristic algorithm that applies this strategy for orderings of general graphs.

How do we find a small separator to disconnect a given graph into components of approximately equal size? The method is to generate a long level structure of the graph and then choose a small separator from a “middle” level. The overall dissection ordering algorithm is described below. Let  $\mathcal{G} = (X, E)$  be the given graph.

**Step 1** (Initialization) Set  $R = X$ , and  $n = |X|$ .

**Step 2** (Generate a level structure) Find a connected component  $\mathcal{G}(C)$  in  $\mathcal{G}(R)$  and construct a level structure of the component  $\mathcal{G}(C)$  rooted at a pseudo-peripheral node  $r$  :

$$\mathcal{L}(r) = \{L_0, L_1, \dots, L_l\}.$$

**Step 3** (Find separator) If  $l \leq 2$ , set  $S = C$  and go to Step 4. Otherwise let  $j = \lfloor (l + 1)/2 \rfloor$ , and determine the set  $S \subset L_j$ , where

$$S = \{y \in L_j \mid \text{Adj}(y) \cap L_{j+1} \neq \phi\}.$$

**Step 4** (Number separator and loop) Number the nodes in the separator  $S$  from  $n - |S| + 1$  to  $n$ . Reset  $R \leftarrow R - S$  and  $n \leftarrow n - |S|$ . If  $R \neq \phi$ , go to Step 2.

In Step 3 of the algorithm, the separator set  $S$  can be obtained by simply discarding nodes in  $L_j$  which are not adjacent to any node in  $L_{j+1}$ . In many cases, this reduces the size of the separators.

### 8.3.2 Computer Implementation

The set of subroutines which implements the nested dissection ordering algorithm consists of those shown in Figure 8.3.1:

The subroutines `FNROOT` and `ROOTLS` have been described in Section 4.4.3 and the utility subroutine `REVRSE` was described in Section 7.3.2. The other two are described below.



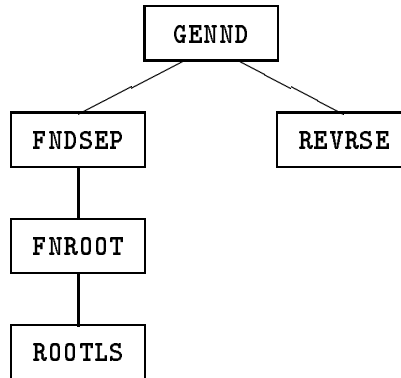


Figure 8.3.1: Control relation of subroutines for the nested dissection algorithm.

---

### GENND (GENERAL NESTED DISSECTION ordering)

This is the driver subroutine for this set of subroutines. It is used to determine a nested dissection ordering for a general disconnected graph. The input graph is given by `NEQNS` and `(XADJ, ADJNCY)`, and the output ordering is returned in the vector `PERM`. The working vector `MASK` is used to mask off nodes that have been numbered during the ordering process. Two more working vectors (`XLS, LS`) are required and they are used by the called subroutine `FNDSEP`.

The subroutine begins by initializing the vector `MASK`. It then goes through the graph until it finds a node  $i$  not yet numbered. This node  $i$  defines a component in the unnumbered portion of the graph. The subroutine `FNDSEP` is then called to find a separator in the component. Note that the separator is collected in the vector `PERM` starting at position `NUM + 1`. So, after all nodes have been numbered, the vector `PERM` has to be reversed to get the final ordering.

---

```

1. C*****
2. C*****
3. C*****      GENND . . . . GENERAL NESTED DISSECTION      *****
4. C*****
5. C*****
6. C
  
```

```

 7. C      PURPOSE - SUBROUTINE GENND FINDS A NESTED DISSECTION
 8. C      ORDERING FOR A GENERAL GRAPH.
 9. C
10. C
11. C      INPUT PARAMETERS -
12. C      NEQNS - NUMBER OF EQUATIONS.
13. C      (XADJ, ADJNCY) - ADJACENCY STRUCTURE PAIR.
14. C
15. C      OUTPUT PARAMETERS -
16. C      PERM - THE NESTED DISSECTION ORDERING.
17. C
18. C      WORKING PARAMETERS -
19. C      MASK - IS USED TO MASK OFF VARIABLES THAT HAVE
20. C      BEEN NUMBERED DURING THE ORDERNG PROCESS.
21. C      (XLS, LS) - THIS LEVEL STRUCTURE PAIR IS USED AS
22. C      TEMPORARY STORAGE BY FNROOT.
23. C
24. C      PROGRAM SUBROUTINES -
25. C      FNDSEP, REVRSE.
26. C
27. C*****
28. C
29. C      SUBROUTINE GENND ( NEQNS, XADJ, ADJNCY, MASK,
30. C      1          PERM, XLS, LS )
31. C
32. C*****
33. C
34. C      INTEGER ADJNCY(1), MASK(1), LS(1), PERM(1),
35. C      1          XLS(1)
36. C      INTEGER XADJ(1), I, NEQNS, NSEP, NUM, ROOT
37. C
38. C*****
39. C
40. C      DO 100 I = 1, NEQNS
41. C          MASK(I) = 1
42. C      100 CONTINUE
43. C          NUM = 0
44. C      DO 300 I = 1, NEQNS
45. C          -----
46. C          FOR EACH MASKED COMPONENT ...
47. C          -----
48. C      200 IF ( MASK(I) .EQ. 0 ) GO TO 300
49. C          ROOT = I
50. C          -----
51. C          FIND A SEPARATOR AND NUMBER THE NODES NEXT.
52. C          -----
53. C          CALL FNDSEP ( ROOT, XADJ, ADJNCY, MASK,

```

```

54.          1          NSEP, PERM(NUM+1), XLS, LS )
55.          NUM = NUM + NSEP
56.          IF ( NUM .GE. NEQNS ) GO TO 400
57.          GO TO 200
58.      300  CONTINUE
59.  C      -----
60.  C      SINCE SEPARATORS FOUND FIRST SHOULD BE ORDERED
61.  C      LAST, ROUTINE REVRSE IS CALLED TO ADJUST THE
62.  C      ORDERING VECTOR.
63.  C      -----
64.      400  CALL REVRSE ( NEQNS, PERM )
65.          RETURN
66.          END

```

---

### FNDSEP (FiND SEParator)

This subroutine is used by GENND to find a separator for a connected subgraph. The connected component is specified by the input parameters ROOT, XADJ, ADJNCY and MASK. Returned from FNDSEP is the separator in (NSEP, SEP). The array pair (XLS, LS) is used to store a level structure of the component.

The subroutine first generates a level structure rooted at a pseudo-peripheral node by calling FNROOT. If the number of levels is less than 3, the whole component is returned as the “separator.” Otherwise, a middle level, given by MIDLVL is determined. The loop DO 500 I = ... goes through the nodes in this middle level. A node is included in the separator if it has some neighbor in the next level. The separator is then returned in (NSEP, SEP).

---

```

1.  C*****
2.  C*****
3.  C*****      FNDSEP . . . . FIND SEPARATOR      *****
4.  C*****
5.  C*****
6.  C
7.  C      PURPOSE - THIS ROUTINE IS USED TO FIND A SMALL
8.  C      SEPARATOR FOR A CONNECTED COMPONENT SPECIFIED
9.  C      BY MASK IN THE GIVEN GRAPH.
10. C
11. C      INPUT PARAMETERS -
12. C      ROOT - IS THE NODE THAT DETERMINES THE MASKED
13. C      COMPONENT.
14. C      (XADJ, ADJNCY) - THE ADJACENCY STRUCTURE PAIR.
15. C

```

```

16. C      OUTPUT PARAMETERS -
17. C          NSEP - NUMBER OF VARIABLES IN THE SEPARATOR.
18. C          SEP - VECTOR CONTAINING THE SEPARATOR NODES.
19. C
20. C      UPDATED PARAMETER -
21. C          MASK - NODES IN THE SEPARATOR HAVE THEIR MASK
22. C              VALUES SET TO ZERO.
23. C
24. C      WORKING PARAMETERS -
25. C          (XLS, LS) - LEVEL STRUCTURE PAIR FOR LEVEL STRUCTURE
26. C              FOUND BY FNROOT.
27. C
28. C      PROGRAM SUBROUTINES -
29. C          FNROOT.
30. C
31. C*****
32. C
33. C      SUBROUTINE FNDSEP ( ROOT, XADJ, ADJNCY, MASK,
34. C          1              NSEP, SEP, XLS , LS )
35. C
36. C*****
37. C
38. C          INTEGER ADJNCY(1), LS(1), MASK(1), SEP(1), XLS(1)
39. C          INTEGER XADJ(1), I, J, JSTOP, JSTRT, MIDBEG,
40. C          1          MIDEND, MIDLVL, MP1BEG, MP1END,
41. C          1          NBR, NLVL, NODE, NSEP, ROOT
42. C
43. C*****
44. C
45. C          CALL FNROOT ( ROOT, XADJ, ADJNCY, MASK,
46. C          1          NLVL, XLS, LS )
47. C      -----
48. C      IF THE NUMBER OF LEVELS IS LESS THAN 3, RETURN
49. C      THE WHOLE COMPONENT AS THE SEPARATOR.
50. C      -----
51. C      IF ( NLVL .GE. 3 ) GO TO 200
52. C          NSEP = XLS(NLVL+1) - 1
53. C          DO 100 I = 1, NSEP
54. C              NODE = LS(I)
55. C              SEP(I) = NODE
56. C              MASK(NODE) = 0
57. C          100 CONTINUE
58. C              RETURN
59. C      -----
60. C      FIND THE MIDDLE LEVEL OF THE ROOTED LEVEL STRUCTURE.
61. C      -----
62. C          200 MIDLVL = (NLVL + 2)/2

```

```

63.      MIDBEG = XLS(MIDLVL)
64.      MP1BEG = XLS(MIDLVL + 1)
65.      MIDEND = MP1BEG - 1
66.      MP1END = XLS(MIDLVL+2) - 1
67.  C    -----
68.  C    THE SEPARATOR IS OBTAINED BY INCLUDING ONLY THOSE
69.  C    MIDDLE-LEVEL NODES WITH NEIGHBORS IN THE MIDDLE+1
70.  C    LEVEL. XADJ IS USED TEMPORARILY TO MARK THOSE
71.  C    NODES IN THE MIDDLE+1 LEVEL.
72.  C    -----
73.      DO 300 I = MP1BEG, MP1END
74.          NODE = LS(I)
75.          XADJ(NODE) = - XADJ(NODE)
76. 300    CONTINUE
77.      NSEP = 0
78.      DO 500 I = MIDBEG, MIDEND
79.          NODE = LS(I)
80.          JSTRT = XADJ(NODE)
81.          JSTOP = IABS(XADJ(NODE+1)) - 1
82.          DO 400 J = JSTRT, JSTOP
83.              NBR = ADJNCY(J)
84.              IF ( XADJ(NBR) .GT. 0 ) GO TO 400
85.                  NSEP = NSEP + 1
86.                  SEP(NSEP) = NODE
87.                  MASK(NODE) = 0
88.                  GO TO 500
89. 400    CONTINUE
90. 500    CONTINUE
91.  C    -----
92.  C    RESET XADJ TO ITS CORRECT SIGN.
93.  C    -----
94.      DO 600 I = MP1BEG, MP1END
95.          NODE = LS(I)
96.          XADJ(NODE) = - XADJ(NODE)
97. 600    CONTINUE
98.      RETURN
99.      END

```

---

## Exercises

- 8.3.1) This problem involves modifying GENND and FNDSEP to implement a form of “incomplete nested dissection.” Add a parameter MINSZE to both subroutines, and modify FNDSEP so that it only dissects the component given to it if the number of nodes in the component is greater than MINSZE. Otherwise, the component should be numbered

using the RCM subroutine from Chapter 4. Conduct an experiment to investigate whether the result you are asked to prove in Exercise 8.2.7 on page 318 appears to hold for the heuristic orderings produced by the algorithm of this section. One way to do this would be to solve a sequence of problems of increasing size, such as the test set #2 from Chapter 9, with MINSZE set to  $\sqrt{n}$ . (For the  $s \times s$  grid problem, note that  $l \geq \log_2(\sqrt{s})$  implies that the final level independent blocks have  $O(s)$  nodes. That is,  $O(\sqrt{n})$  nodes, where  $n = s^2$ .) Monitor the operation counts for these problems, and compare them to the corresponding values for the original (complete) dissection algorithm. Similarly, you could compare storage requirements to see if they appear to grow as  $n\sqrt{n}$  for your incomplete dissection algorithm.

- 8.3.2) Show that in the algorithm of Section 8.3.1, the number of fills and factorization operation count are independent of the order the nodes in the separator are numbered.

## 8.4 Additional Notes

Lipton, Tarjan and Rose [36] have provided a major advance in the development of automatic nested dissection algorithms. The key to their algorithm is a fundamental result by Lipton and Tarjan [37] showing that the nodes of any  $n$ -node planar graph can be partitioned into three sets  $A$ ,  $B$ , and  $C$  where  $Adj(A) \cap B = \phi$ ,  $|C|$  is  $O(\sqrt{n})$ , and  $|A|$  and  $|B|$  are bounded by  $2n/3$ . They also provided an algorithm which finds  $A$ ,  $B$ , and  $C$  in  $O(n)$  time. Using this result Lipton et al. have developed an ordering algorithm for two dimensional finite element problems for which the  $O(n^{3/2})$  operation and  $O(n \log_2 n)$  storage bounds are guaranteed. Moreover, the ordering algorithm itself runs in  $O(n \log_2 n)$  time. On the negative side, their algorithm appears to be substantially more complicated than the simple heuristic one given in this chapter. A practical approach might be to combine the two methods, and use their more sophisticated scheme only if the simple approach in this chapter yields a “bad” separator.

The use of nested dissection ideas has been shown to be effective for problems associated with three dimensional structures. (George [18], Duff et al. [11], Rose [45], Eisenstat et al. [14].) Thus, research into automatic nested dissection algorithms for these non-planar problems appears to be a potentially fertile area.

The use of dissection methods on parallel and vector computers has been investigated by numerous researchers (Calahan [6, 7], George et al. [21], Lambiotte [34]). Vector computers tend to be most efficient if they can operate on “long” vectors, but the use of dissection techniques tend to produce short vectors, unless some unconventional methods of arranging the data are employed. Thus, the main issue in these studies involves balancing several conflicting criteria to produce the best solution time. Often this does not correspond at all closely to minimizing the arithmetic performed.

## Chapter 9

# Numerical Experiments

### 9.1 Introduction

In Chapter 1 we asserted that the success of algorithms for sparse matrix computations depends crucially on the quality of their computer implementations. This is why we have included computer implementations of the algorithms discussed in the previous chapters, and have provided a detailed discussion of how those programs work. In this chapter we provide results from numerical experiments where these subroutines have been used to solve some test problems.

Our primary objective here is to provide some concrete examples which illustrate the points made in Section 2.4, where “practical considerations” were discussed, and where it was pointed out how complicated it is to compare different methods. Data structures vary in their complexity, and the execution time for solving a problem consists of several components whose importance varies with the ordering strategy and the problem. The numerical results provided in this chapter give the user information to gauge the significance of some of these points.

As an attractive byproduct, the reader is supplied with data about the absolute time and storage requirements for some representative sparse matrix computations on a typical computer.

The test problems are of one specific type, typical of those arising in finite element applications. Our justification for this is that we are simply trying to provide some evidence illustrating the practical points made earlier; we regard it as far too ambitious to attempt to gather evidence about the relative merits of different methods over numerous classes of problems. It is more



or less self-evident that for some classes of problems, one method may be uniformly better than all others, or that the relative merits of the methods in our book may be entirely different for other classes of problems. Restricting our attention to problems of one class simply removes one of the variables in an already complicated study.

Nevertheless, the test problems do represent a large and important application area for sparse matrix techniques, and have the additional advantage that they are associated with physical objects (meshes) which provide us with a picture (graph) of the matrix problem.

An outline of the remaining parts of this chapter is as follows. In Section 9.2 we describe the test problems, and in Section 9.3 we describe the information supplied in some of the tables, along with the reasons for providing it. These tables, containing the “raw” experimental data, appear at the end of Section 9.3. In Section 9.4 we review the main criteria used in comparing methods, and then proceed to compare five methods, according to these criteria, when applied to the test problems. Finally, in Section 9.5 we consider the influence of the different storage schemes on the storage and computational efficiency of the numerical subroutines.

## 9.2 Description of the Test Problems

The two sets of test problems are positive definite matrix equations typical of those which might arise in structural analysis or the study of heat conduction (Zienkiewicz [58]). (For an excellent tutorial see Chapter 6 of Strang [51].) The problems are derived from the triangular meshes shown in Figure 9.2.1 as follows. The basic meshes shown are subdivided by a factor  $s$  in the obvious way, yielding a mesh having  $s^2$  as many triangles as the original, as shown in Figure 9.2.2 for the pinched hole domain with  $s = 3$ . Providing a basic mesh along with a subdivision factor determines a new mesh having  $N$  nodes. Then, for some labelling of these  $N$  nodes, we generate an  $N$  by  $N$  symmetric positive definite matrix problem  $\mathbf{Ax} = \mathbf{b}$ , where  $a_{ij} \neq 0$  if and only if nodes of the mesh are joined by an edge. Thus, the generated meshes can be viewed as the graphs of the corresponding matrix problem.

The two sets of test problems are derived from these meshes. Test set #1 is simply the nine mesh problems, subdivided by an appropriate factor so that the resulting matrix problems have about 1000–1500 equations, as shown in Table 9.2.1. The second set of problems is a sequence of nine graded-L problems obtained by subdividing the initial graded-L mesh of Figure 9.2.1

by subdivision factors  $s = 4, 5, \dots, 12$ , as indicated in Table 9.2.2.

Problem	Subdivision factor	$N$	$ E $
Square	32	1089	3136
Graded L	8	1009	2928
+ domain	9	1180	3285
H domain	8	1377	3808
Small hole	12	936	2665
Large hole	9	1440	4032
3 holes	6	1138	3156
6 holes	6	1141	3162
Pinched hole	19	1349	3876

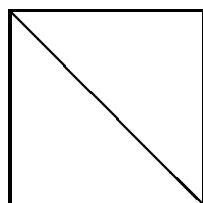
Table 9.2.1: Data on test problem set #1 with the subdivision factors used to generate the problems, the number of equations obtained, and the number of edges in the corresponding graphs.

Subdivision factor	$N$	$ E $
4	265	744
5	406	1155
6	577	1656
7	778	2247
8	1009	2928
9	1270	3699
10	1561	4560
11	1882	5511
12	2233	6552

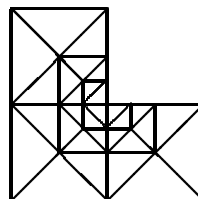
Table 9.2.2: Data on test problem set #2, which is derived from the Graded-L mesh with subdivision factors  $s = 4, 5, \dots, 12$ .

### 9.3 The Numbers Reported and What They Mean

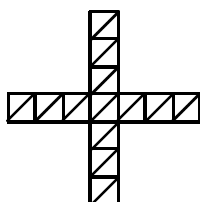
In Chapters 4 through 8 we have described five methods, which in this chapter we refer to by the mnemonics RCM (reverse Cuthill-McKee), RQT (refined quotient tree), 1WD (one-way dissection), QMD (quotient minimum



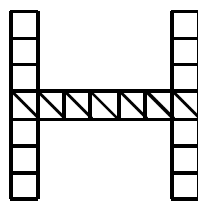
a) Square



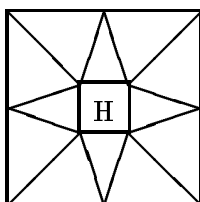
b) Graded L



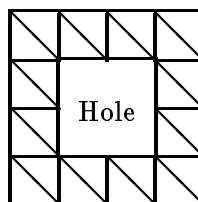
c) + shaped domain



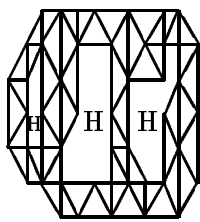
d) H-shaped domain



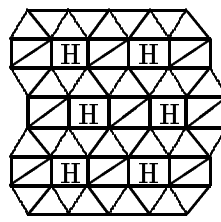
e) Hollow square (small hole)



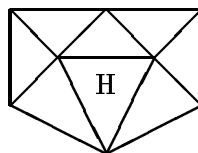
f) Hollow square (large hole)



g) 3-hole problem



h) 6-hole problem



i) Pinched hole problem

Figure 9.2.1: Mesh problems with  $s = 1$ .

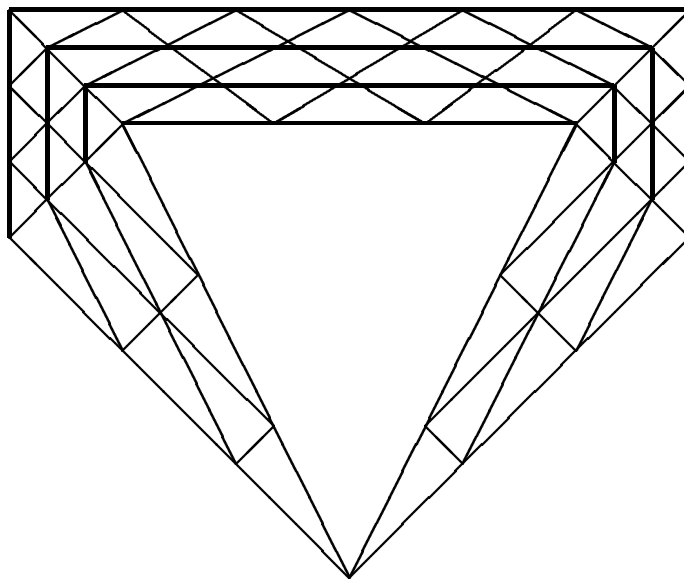


Figure 9.2.2: Pinched hole domain with subdivision factor  $s = 3$ .

---

degree), and ND (nested dissection). Recall that we described only three basic data structures and corresponding numerical subroutines, because it is appropriate to use the same data structures with the one-way dissection and refined quotient tree orderings, and similarly for the minimum degree and nested dissection orderings.

In the tables at the end of this section, *operations* mean *multiplicative* operations (multiplications and divisions). For reasons already discussed in Chapter 2, we regard this as a reasonable measure of the amount of arithmetic performed, since arithmetic operations in matrix computations typically occur in multiply-add pairs. *Execution time* is reported in seconds on an IBM 3031 computer, a fairly recent architecture using high speed cache memory, and on which typical operations take from .4 microseconds for a simple fixed-point register-to-register operation, to about 7 microseconds for a floating-point division. As is usual in multiprogrammed operating system environments, accurate timing results are difficult to obtain and may be in error by up to 10 percent. We have attempted to reduce these errors somewhat by making multiple runs, and running when the computer was lightly loaded. The programs were all compiled using the optimizing version of the compiler, which usually generates very efficient machine code.

Recall that we concluded in Section 2.4 that in some comparisons of ordering strategies, it might be reasonable to ignore one or more of the four basic steps in the overall solution procedure. For this reason, in the numerical experiments we report execution times for each of the four individual steps: order, allocate, factor, and solve.

There are four storage statistics reported in the tables: *order storage*, *allocation storage*, *total (solution) storage*, and *overhead storage*. All our experiments were performed within the framework of a sparse matrix package called SPARSPAK ([24, 25]) which allocates all array storage from a single one dimensional array. The order storage, allocation storage, and solution storage reported is the amount of storage used from that array. Thus, we feel that these numbers represent the amount of storage required when the various subroutines are used in a *practical setting*, rather than the irreducible minimum necessary to execute the subroutines. To illustrate this point, note that one does not *need* to preserve the original graph when one uses the QMD ordering subroutine, (which destroys its input graph during execution) but in most practical applications one *would* preserve the graph since it is required for the subsequent symbolic factorization step. Thus, the ordering storage entries under QMD in the tables *include* the space necessary to preserve the original graph.

As another example, it is obviously not necessary to preserve `PERM` and `INVP` after the allocation has been performed, since the numerical factorization and solution subroutines do not use these arrays. However, in most situations the arrays would be saved in order to place the numerical values of  $\mathbf{A}$  and  $\mathbf{b}$  in the appropriate places in the data structure, and to replace the values of  $\mathbf{x}$  in the original order after the (permuted) solution has been computed. In Table 9.3.1 we list the arrays included in our storage reporting, for the different phases of the computation (order, allocate, factorization, solution), and for the five methods. The notation  $\mathbf{A}(\mathbf{B})$  in Table 9.3.1 means arrays  $\mathbf{A}$  and  $\mathbf{B}$  use the same storage space, in sequence.

Strictly speaking, we should distinguish between factorization storage and triangular solution storage, since several arrays required by `TSFCT` and `GSFCT` are not required by their respective solvers. However, the storage for these arrays will usually be relatively small, compared to the total storage required for the triangular solution. Thus, we report only “solution storage” in our tables.

So far our discussion about storage reporting has dealt only with the first three categories: ordering, allocation, and numerical solution. The fourth category is “overhead storage,” which is included in order to illustrate how much of the total storage used during the factorization/solution phase is occupied by data [*other than the nonzeros in  $\mathbf{L}$  and the right hand side*]  $\mathbf{b}$  (which is overwritten by  $\mathbf{x}$ ). If a storage location is not being used to store a component of  $\mathbf{L}$  or  $\mathbf{b}$ , then we count it as overhead storage. The arrays making up the overhead storage entries are underlined in Table 9.3.1. Note that solution storage *includes* overhead storage.

There is another reason for reporting overhead storage as a separate item. On computers having a large word size, it may be sensible to pack several integers per word. Indeed, some computer manufacturers provide short integer features directly in their Fortran languages. For example, IBM Fortran allows one to declare integers as `INTEGER*2` or `INTEGER*4`, which will be represented using 16 or 32 bits respectively. Since much of the overhead storage involves integer data, the reader can gauge the potential storage savings to be realized if the Fortran processor one is using provides these short integer features. However, note that all the experiments were performed on an IBM 3031 in single precision, and both integers and floating point numbers are represented using 32 bits.

	Order	Allocate	Solution
	XADJ, ADJNCY, PERM, XLS, MASK	XADJ, ADJNCY, PERM, INVP	<u>PERM</u> , <u>INVP</u> , RHS, <u>XENV</u> , ENV, DIAG
RCM	XADJ, ADJNCY, PERM, BNUM, LS(SUBG), XBLK, MASK, XLS	XADJ, ADJNCY, PERM, INVP, XBLK, MASK, MARKER, FATHER, XENV, NZSUBS, RCHSET(XNONZ)	<u>PERM</u> , <u>INVP</u> , RHS, <u>XENV</u> , ENV, DIAG, <u>XNONZ</u> , <u>NZSUBS</u> , NONZ, <u>TEMPV</u> , <u>FIRST</u>
1WD	XADJ, ADJNCY, PERM, XBLK, MASK, NODLVL(BNUM), XLS, LS(SUBG)	XADJ, ADJNCY, PERM, INVP, XBLK, MASK, FATHER, XENV, XNONZ, NZSUBS	
RQT	XADJ, ADJNCY, PERM, LS, XLS, MASK	XADJ, ADJNCY, PERM, INVP, XLNZ, XNZSUB, NZSUB, MRGLNK, RCHLNK, MARKER	<u>PERM</u> , <u>INVP</u> , RHS, <u>XNZSUB</u> , <u>NZSUB</u> , <u>XLNZ</u> , <u>LNZ</u> , DIAG, <u>LINK</u> , <u>FIRST</u> , <u>TEMPV</u>
ND	XADJ, 2 copies of ADJNCY, PERM, MARKER, DEG, RCHSET, NBRHD, QSIZE, QLINK		

Table 9.3.1: Arrays included in reported storage requirements for each phase of the five methods. Storage required for the undeclared arrays in the *Solution* column is reported as “overhead storage.”

same as above

same as above

Problem	Order		Allocation		Solution					
	Time Store		Time Store		Time		Operations		Store	
					Fact. Solve	Fact. Solve	Solve	Solve	Total Ovrhd	
936	0.21	0.91	0.04	0.91	2.85	0.40	30.18	4.55	2.65	0.28
1009	0.27	0.99	0.04	0.99	3.43	0.46	37.49	5.25	3.03	0.30
1089	0.24	1.06	0.05	1.06	3.25	0.45	34.46	5.11	2.99	0.33
1440	0.32	1.38	0.06	1.38	4.74	0.62	53.75	7.23	4.19	0.43
1180	0.32	1.13	0.06	1.13	2.86	0.44	31.87	5.17	3.06	0.35
1377	0.30	1.31	0.06	1.31	1.99	0.40	18.64	4.34	2.72	0.41
1138	0.30	1.09	0.06	1.09	2.81	0.45	28.88	4.92	2.92	0.34
1141	0.25	1.09	0.05	1.09	4.40	0.52	54.08	6.75	3.83	0.34
1349	0.36	1.31	0.06	1.31	5.74	0.69	64.95	7.95	4.52	0.40

Table 9.3.2: Results of the RCM method applied to test set #1. (Operations and storage scaled by  $10^{-4}$ )

Problem	Order		Allocation		Solution					
	Time Store		Time Store		Time		Operations		Store	
					Fact. Solve	Fact. Solve	Solve	Solve	Total Ovrhd	
936	0.38	1.19	0.25	1.24	3.39	0.36	26.60	3.06	1.72	0.61
1009	0.47	1.29	0.28	1.35	5.47	0.40	44.90	3.66	1.94	0.66
1089	0.45	1.39	0.30	1.46	5.23	0.42	41.48	3.62	2.03	0.72
1440	0.60	1.81	0.39	1.89	4.43	0.52	33.04	4.32	2.51	0.94
1180	0.55	1.48	0.33	1.56	3.35	0.42	24.50	3.48	2.03	0.78
1377	0.57	1.73	0.37	1.82	3.25	0.49	21.41	3.57	2.21	0.92
1138	0.52	1.43	0.30	1.49	3.17	0.41	23.56	3.48	1.98	0.75
1141	0.46	1.43	0.31	1.49	5.10	0.44	41.08	3.77	2.07	0.74
1349	0.61	1.72	0.36	1.79	7.03	0.56	58.38	4.79	2.57	0.88

Table 9.3.3: Results of the 1WD method applied to test set #1. (Operations and storage scaled by  $10^{-4}$ )



Problem	Order		Allocation		Solution					
	Time Store		Time Store		Time		Operations		Store	
					Fact. Solve	Fact. Solve	Solve	Solve	Total Ovrhd	
936	0.18	1.19	0.15	1.19	4.17	0.53	32.84	4.85	2.14	0.75
1009	0.26	1.29	0.16	1.30	4.88	0.59	39.32	5.49	2.38	0.81
1089	0.22	1.39	0.17	1.40	4.68	0.61	37.04	5.43	2.45	0.88
1440	0.29	1.81	0.22	1.81	6.75	0.82	55.46	7.44	3.29	1.15
1180	0.31	1.48	0.19	1.49	2.39	0.54	13.52	3.41	2.04	0.95
1377	0.28	1.74	0.21	1.74	2.30	0.58	11.69	3.53	2.27	1.12
1138	0.30	1.43	0.17	1.43	4.32	0.60	21.98	5.13	2.41	0.91
1141	0.23	1.42	0.17	1.42	7.18	0.72	63.98	6.92	2.86	0.91
1349	0.36	1.72	0.21	1.72	7.79	0.86	67.68	8.30	3.42	1.08

Table 9.3.4: Results of the RQT method applied to test set #1. (Operations and storage scaled by  $10^{-4}$ )

Problem	Order		Allocation		Solution					
	Time Store		Time Store		Time		Operations		Store	
					Fact. Solve	Fact. Solve	Solve	Solve	Total Ovrhd	
936	0.77	1.00	0.24	1.78	2.19	0.29	16.25	2.96	2.73	1.15
1009	0.95	1.09	0.25	1.97	3.77	0.37	31.11	4.00	3.38	1.29
1089	0.92	1.17	0.27	2.10	3.40	0.37	26.82	3.91	3.43	1.36
1440	1.35	1.53	0.34	2.68	2.69	0.41	19.05	4.06	3.90	1.72
1180	1.10	1.25	0.28	2.17	2.05	0.31	14.22	3.15	3.09	1.40
1377	1.20	1.45	0.32	2.51	2.34	0.36	15.71	3.59	3.54	1.61
1138	1.15	1.20	0.27	2.11	2.32	0.32	16.89	3.32	3.14	1.37
1141	1.14	1.20	0.27	2.13	2.32	0.32	17.23	3.34	3.16	1.38
1349	1.39	1.45	0.33	2.60	4.48	0.48	35.48	4.98	4.31	1.69

Table 9.3.5: Results of the ND method applied to test set #1. (Operations and storage scaled by  $10^{-4}$ )

Problem	Order		Allocation		Solution					
	Time Store		Time Store		Time		Operations		Store	
					Fact. Solve	Fact. Solve	Solve	Solve	Total Ovrhd	
936	1.47	1.91	0.21	1.80	2.27	0.30	19.34	3.11	2.83	1.18
1009	1.57	2.08	0.24	1.97	3.37	0.37	30.91	3.95	3.36	1.29
1089	1.78	2.23	0.26	2.12	3.00	0.36	26.31	3.85	3.42	1.38
1440	2.33	2.91	0.34	2.74	2.70	0.42	21.62	4.21	4.04	1.79
1180	1.89	2.38	0.27	2.19	1.44	0.27	9.86	2.72	2.90	1.42
1377	2.09	2.76	0.30	2.52	1.50	0.30	10.00	2.99	3.25	1.62
1138	1.97	2.29	0.27	2.15	1.82	0.29	13.80	3.04	3.04	1.41
1141	2.07	2.29	0.27	2.17	2.03	0.31	16.24	3.24	3.14	1.43
1349	2.07	2.76	0.32	2.62	3.70	0.46	32.41	4.41	4.26	1.71

Table 9.3.6: Results of the QMD method applied to test set #1. (Operations and storage scaled by  $10^{-4}$ )

Problem	Order		Allocation		Solution					
	Time Store		Time Store		Time		Operations		Store	
					Fact. Solve	Fact. Solve	Solve	Solve	Total Ovrhd	
265	0.07	0.25	0.01	0.25	0.33	0.07	2.97	0.75	0.48	0.08
406	0.12	0.39	0.02	0.39	0.71	0.13	6.62	1.39	0.86	0.12
577	0.16	0.56	0.03	0.56	1.30	0.21	12.88	2.32	1.39	0.17
778	0.22	0.76	0.03	0.76	2.15	0.32	22.78	3.59	2.10	0.23
1009	0.27	0.99	0.05	0.99	3.43	0.46	37.49	5.25	3.03	0.30
1270	0.34	1.25	0.06	1.25	5.19	0.62	58.37	7.36	4.19	0.38
1561	0.42	1.54	0.07	1.54	7.50	0.83	86.95	9.97	5.61	0.47
1882	0.51	1.86	0.09	1.86	10.62	1.11	124.90	13.14	7.32	0.56
2233	0.62	2.20	0.10	2.20	14.44	1.40	174.10	16.91	9.32	0.67

Table 9.3.7: Results of the RCM method applied to test set #2. (Operations and storage scaled by  $10^{-4}$ )

Problem	Order		Allocation		Solution					
	Time Store		Time Store		Time		Operations		Store	
					Fact. Solve		Fact. Solve		Total Ovrhd	
265	0.12	0.33	0.07	0.35	0.65	0.09	4.38	0.69	0.42	0.18
406	0.18	0.52	0.11	0.54	1.29	0.15	9.25	1.18	0.69	0.27
577	0.27	0.74	0.17	0.77	2.39	0.22	18.11	1.87	1.04	0.38
778	0.35	0.99	0.20	1.04	3.69	0.32	29.22	2.68	1.45	0.51
1009	0.48	1.29	0.28	1.35	5.46	0.41	44.90	3.66	1.94	0.66
1270	0.60	1.63	0.35	1.70	7.76	0.54	66.00	4.87	2.53	0.83
1561	0.70	2.00	0.43	2.09	11.09	0.68	97.14	6.36	3.25	1.02
1882	0.91	2.42	0.51	2.51	14.73	0.85	131.13	7.90	4.00	1.22
2233	1.05	2.87	0.62	2.98	19.64	1.04	177.17	9.91	4.89	1.45

Table 9.3.8: Results of the 1WD method applied to test set #2. (Operations and storage scaled by  $10^{-4}$ )

Problem	Order		Allocation		Solution					
	Time Store		Time Store		Time		Operations		Store	
					Fact. Solve		Fact. Solve		Total Ovrhd	
265	0.07	0.33	0.04	0.34	0.57	0.12	3.24	0.81	0.47	0.21
406	0.10	0.52	0.07	0.52	1.12	0.20	7.11	1.49	0.78	0.33
577	0.16	0.74	0.09	0.74	1.94	0.29	13.70	2.46	1.19	0.46
778	0.20	0.99	0.12	1.00	3.15	0.43	24.03	3.78	1.72	0.63
1009	0.26	1.29	0.16	1.30	4.86	0.60	39.32	5.49	2.38	0.81
1270	0.33	1.63	0.20	1.63	7.15	0.78	60.94	7.67	3.19	1.02
1561	0.40	2.00	0.25	2.01	10.14	1.04	90.43	10.35	4.15	1.25
1882	0.48	2.42	0.30	2.43	14.06	1.31	129.48	13.59	5.28	1.51
2233	0.56	2.87	0.35	2.88	19.20	1.67	179.99	17.44	6.60	1.79

Table 9.3.9: Results of the RQT method applied to test set #2. (Operations and storage scaled by  $10^{-4}$ )

Problem	Order		Allocation		Solution					
	Time	Store	Time	Store	Time		Operations		Store	
					Fact. Solve	Fact. Solve	Fact. Solve	Fact. Solve	Total Ovrhd	Total Ovrhd
265	0.20	0.28	0.06	0.49	0.46	0.07	3.25	0.72	0.70	0.32
406	0.33	0.43	0.10	0.77	0.93	0.12	6.93	1.27	1.17	0.50
577	0.49	0.62	0.14	1.10	1.62	0.19	12.56	1.99	1.77	0.72
778	0.68	0.84	0.20	1.51	2.58	0.29	20.10	2.88	2.50	0.99
1009	0.95	1.09	0.26	1.97	3.80	0.37	31.11	4.00	3.38	1.29
1270	1.22	1.38	0.32	2.49	5.30	0.49	44.45	5.27	4.39	1.63
1561	1.56	1.69	0.39	3.08	7.34	0.63	62.57	6.82	5.58	2.01
1882	1.93	2.04	0.50	3.74	9.73	0.78	83.85	8.54	6.90	2.44
2233	1.35	2.43	0.58	4.45	12.79	0.94	111.18	10.57	8.42	2.92

Table 9.3.10: Results of the ND method applied to test set #2. (Operations and storage scaled by  $10^{-4}$ )

Problem	Order		Allocation		Solution					
	Time	Store	Time	Store	Time		Operations		Store	
					Fact. Solve	Fact. Solve	Fact. Solve	Fact. Solve	Total Ovrhd	Total Ovrhd
265	0.39	0.54	0.06	0.49	0.36	0.07	2.65	0.65	0.67	0.32
406	0.72	0.83	0.09	0.78	0.78	0.11	6.35	1.19	1.14	0.50
577	1.01	1.18	0.13	1.12	1.26	0.18	10.35	1.83	1.70	0.73
778	1.39	1.60	0.19	1.52	2.22	0.26	19.65	2.80	2.48	1.00
1009	1.55	2.08	0.24	1.97	3.43	0.38	30.91	3.95	3.36	1.29
1270	2.48	2.62	0.34	2.53	4.59	0.48	42.56	5.15	4.37	1.66
1561	2.56	3.23	0.39	3.09	5.90	0.60	55.43	6.53	5.45	2.03
1882	3.32	3.90	0.48	3.76	8.36	0.76	80.10	8.49	6.91	2.47
2233	3.53	4.63	0.55	4.43	12.06	0.98	119.13	10.71	8.47	2.89

Table 9.3.11: Results of the QMD method applied to test set #1. (Operations and storage scaled by  $10^{-4}$ )

## 9.4 Comparison of the Methods

### 9.4.1 Criteria for Comparing Methods

In this section we shall not attempt to answer the question “which method should we use?”. Sparse matrices vary a great deal, and the collection of test problems is of only one special class. Our objective here is to illustrate, using the data reported in Section 9.3, the issues involved in answering the question, given a particular problem or class of problems. These issues have already been discussed, or at least mentioned, in Section 2.4.

The main criteria were a) storage requirements, b) execution time, and c) cost. In some contexts, keeping storage requirements low is of overwhelming importance, while in other situations, low execution time is of primary concern. Perhaps most frequently, however, we are interested in choosing the method which results in the lowest computer charges. This charging function is typically a fairly complicated multi-parameter function of storage used ( $S$ ), execution time ( $T$ ), amount of input and output performed, . . . , etc. For our class of problems and the methods we treat, this charging function can usually be quite well approximated by a function of the form

$$COST(S, T) = T \times p(S),$$

where  $p(S)$  is a polynomial of degree  $d$ , usually equal to 1. (However, sometimes  $d = 0$ , and in other cases where large storage demands are discouraged,  $d = 2$ .) For purposes of illustration, in this book we assume  $p(S) = S$ .

Recall from Section 2.4 that the relative importance of the ordering and allocation, factorization, and solution depends on the context in which the sparse matrix problem arises. In some situations only one problem of a particular structure is to be solved, so any comparison of methods should certainly include ordering and allocation costs. In other situations where many problems having identical structure must be solved, it may be sensible to ignore the ordering and allocation costs. Finally, in still other contexts where numerous systems differing only in their right hand side must be solved, it may be appropriate to consider only the time and/or storage associated with the triangular solution, *given the factorization*.

In some of the tables appearing later we report a “minimum” and “maximum” total cost. The distinction is that the maximum cost is computed assuming that the storage used by any of the four phases (order, allocate, factor, solve) is equal to the maximum storage required by any of them (usually the factorization step). The minimum cost is obtained by assum-

ing that the storage used by each phase is the minimum required by that phase (as specified in Table 9.3.1). We report both costs to show that for some methods and problems, the costs are quite different and it is therefore worthwhile to segment the computation into its constituent parts, and use only the requisite storage for each phase.

#### 9.4.2 Comparison of the Methods Applied to Test Set #1

Now consider Table 9.4.1, which we obtained by averaging the results in the tables of Section 9.3 for test set #1, and then computing the various costs. One of the most important things that it shows is that for the nine problems of this test set, the method of choice depends very much on the criterion we wish to optimize. For example, if total execution time is the basis for choice, then RCM should be chosen. If solution time, or factorization plus solution time or factorization plus solution cost, is of primary importance, then QMD should be chosen. If storage requirements, solve cost, or total cost are the most important criteria, then 1WD is the method of choice.

Several other aspects of Table 9.4.1 are noteworthy. Apparently, QMD yields a somewhat better ordering than ND, which is reflected in lower execution times and costs for the factorization and solution, and lower storage requirements. However, the fact that the ordering time for ND is substantially lower than that for QMD results in lower *total* costs and execution time for ND, compared to QMD.

Another interesting aspect of the ND and QMD total cost entries is the substantial difference between the maximum and minimum costs. Recall from Section 9.4.1 that the maximum cost is computed assuming that the storage used during any of the phases (order, allocate, factor, solve) is equal to the maximum used by any of them, while the minimum cost is computed assuming that each phase uses only what is normally required, as prescribed by Table 9.3.1. These numbers suggest that even for “one-shot” problems, segmenting the computation into its natural components, and using only the storage required for each phase, is well worthwhile.

After examining Table 9.4.1, the reader might wonder whether methods such as RQT and ND have any merit, compared to the three other methods, since they fail to show up as winners according to any of the criteria we are considering. However, averages tend to hide differences among the problems, and to illustrate that each method does have a place, Table 9.4.2 contains a frequency count of which method was best, based on the various criteria, for the problems of set #1. Note that no row in the table is all zeros.

Method	Cost				Storage	Execution Time		
	Total (Max)	Total (Min)	Fact+ Solve	Solve		Total	Fact+ Solve	Solve
RCM	14.60	13.86	13.47	1.64	3.32	4.39	4.06	0.49
1WD	12.22	11.72	10.45	0.95	2.12	5.77	4.94	0.45
RQT	15.60	15.11	14.44	1.68	2.58	6.04	5.59	0.65
ND	15.63	12.92	10.89	1.22	3.41	6.59	3.19	0.36
QMD	16.66	14.52	9.30	1.15	3.36	4.96	2.77	0.34

Table 9.4.1: Average values of the various criteria for problem set #1. (Costs and storage scaled by  $10^{-4}$ )

This suggests that even within a particular class of problems, and for a fixed criterion (e.g., execution time, storage), the method of choice varies considerably across problems. One should also keep in mind that special combinations of criteria may make any of the methods look best, for almost any of the problems.

Method	Cost				Storage	Execution Time		
	Total (Max)	Total (Min)	Fact+ Solve	Solve		Total	Fact+ Solve	Solve
RCM	3	3	1	0	0	4	0	0
1WD	4	4	2	7	9	0	0	0
RQT	1	1	0	0	0	1	0	0
ND	1	1	1	0	0	3	2	3
QMD	0	0	5	2	0	1	7	6

Table 9.4.2: Frequency counts of which method was best on the basis of various criteria for test problem set #1.

One rather striking aspect of Table 9.4.2 is the very strong showing of 1WD in terms of cost and storage.

### 9.4.3 Comparison of the Methods Applied to Test Set #2

In order to illustrate some additional points, we include Tables 9.4.3 and reftab9.3.4, generated from Tables 9.3.7 – 9.3.11 of Section 9.3, which contain the experimental results for test problem set #2. Table 9.4.3 contains the same information as Table 9.4.1, for the Graded-L problem with  $s = 4$ , (yielding  $N = 265$ ). Table 9.4.4 is also the same, except the subdivision factor is  $s = 12$ , yielding  $N = 2233$ .

First note that for  $N = 265$ , the RCM method displays a considerable advantage in most categories, and is very competitive in the remaining ones. However, for  $N = 2233$ , it has lost its advantage in all except total execution time. (Some other experiments show that it loses to ND in this category also for somewhat larger graded-L problems.) One of the main points we wish to make here is that even for essentially similar problems such as these, the *size* of the problem can influence the method of choice. Roughly speaking, for “small problems,” the more sophisticated methods simply do not pay. It is interesting to again note how very effective the 1WD method is in terms of storage and cost.

Notice also that the relative cost of the ordering and allocation steps, compared to the total cost, is going down as  $N$  increases, for all the methods. For the RCM, 1WD and RQT methods, these first two steps have become relatively unimportant in the overall cost and execution time when  $N$  reaches about 2000. However, for the ND and QMD methods, even for  $N$  as large as 2233, the ordering and allocation steps still account for a significant fraction of the total execution time. Since these steps in general require less storage than the numerical computation steps, the difference between *MAX* cost and *MIN* cost remains important even for  $N = 2233$ .

## 9.5 The Influence of Data Structures

In several places in this book we have emphasized the importance of data structures (storage schemes) for sparse matrices. In Section 2.4 we distinguished between primary storage and overhead storage, and through a simple example showed that primary storage requirements may not be a reliable indicator of the storage actually required by different computer programs, because of differences in overhead storage. We also pointed out in Section 2.4 that differences in data structures could lead to substantial differences in the arithmetic-operations-per-second output of the numerical subroutines. The main objective of this section is to provide some experimental evidence which



Method	Cost				Storage	Execution Time		
	Total	Total	Fact+	Solve		Total	Fact+	Solve
	(Max)	(Min)	Solve					Solve
RCM	0.23	0.21	0.19	0.04	0.48	0.48	0.40	0.07
1WD	0.39	0.37	0.31	0.04	0.42	0.93	0.73	0.09
RQT	0.38	0.36	0.32	0.06	0.47	0.81	0.69	0.12
ND	0.56	0.46	0.37	0.05	0.70	0.79	0.53	0.07
QMD	0.59	0.53	0.29	0.04	0.67	0.88	0.43	0.07

Table 9.4.3: Values of the various criteria for the Graded-L problem with  $s = 4$ , yielding  $N = 265$ . (Costs and storage scaled by  $10^{-4}$ )

Method	Cost				Storage	Execution Time		
	Total	Total	Fact+	Solve		Total	Fact+	Solve
	(Max)	(Min)	Solve					Solve
RCM	154.83	149.69	148.10	13.09	9.35	16.56	15.84	1.40
1WD	109.41	106.10	101.25	5.11	4.89	22.35	20.69	1.04
RQT	143.69	140.30	137.67	10.99	6.60	21.78	20.87	0.12
ND	140.45	124.03	115.74	7.95	8.42	16.67	13.74	0.94
QMD	145.07	129.28	110.49	8.33	8.47	17.13	13.04	0.98

Table 9.4.4: Values of the various criteria for the Graded-L problem with  $s = 12$ , yielding  $N = 2233$ . (Costs and storage scaled by  $10^{-4}$ )

supports these contentions, and to illustrate the potential magnitude of the differences involved.

### 9.5.1 Storage Requirements

In Table 9.5.1 we have compiled the primary and total storage requirements for the five methods, applied to test problem set #2. Recall that primary storage is that used for the numerical values of  $L$  and  $b$ , and overhead storage is “everything else,” consisting mainly of integer pointer data associated with maintaining a compact representation of  $L$ .

		Primary Storage								
		Number of Equations								
Method	265	406	577	778	1009	1270	1561	1882	2233	
RCM	0.40	0.74	1.22	1.87	2.73	3.81	5.14	6.76	8.68	
1WD	0.24	0.42	0.66	0.94	1.28	1.70	2.23	2.78	3.45	
RQT	0.26	0.45	0.73	1.10	1.57	2.17	2.90	3.77	4.80	
ND	0.39	0.67	1.05	1.52	2.10	2.76	3.57	4.46	5.51	
QMD	0.35	0.64	0.97	1.48	2.08	2.70	3.42	4.43	5.58	

		Total Storage								
		Number of Equations								
Method	265	406	577	778	1009	1270	1561	1882	2233	
RCM	0.48	0.86	1.39	2.10	3.03	4.19	5.61	7.32	9.35	
1WD	0.42	0.69	1.04	1.45	1.94	2.53	3.25	4.00	4.89	
RQT	0.47	0.78	1.19	1.72	2.38	3.19	4.15	5.28	6.60	
ND	0.70	1.17	1.77	2.50	3.38	4.39	5.58	6.90	8.42	
QMD	0.67	1.14	1.70	2.48	3.36	4.37	5.45	6.91	8.47	

Table 9.5.1: Primary and total storage for each method, applied to test problem #2. (Operations are scaled by  $10^{-4}$ )

The numbers in Table 9.4.1 illustrate some important practical points:

1. For some methods, the overhead component in the storage requirements is very substantial, even for the larger problems where the relative importance of overhead is diminished somewhat. For example, for the QMD method, the ratio (*overhead storage*)/(*total storage*) ranges

from about .48 to .34 as  $N$  goes from 265 to 2233. Thus, while the ratio is decreasing, it is still very significant for even fairly large problems.

By way of comparison, for the RCM method, which utilizes a very simple data structure, the (*overhead storage*)/(*total storage*) ratio ranges from about .17 to .07 over the same problems.

2. Another point, (essentially a consequence of 1 above,) is that primary storage is a very unreliable indicator of a program's array storage requirements. For example, if we were comparing the RCM and QMD methods on the basis of primary storage requirements for the problems of test set #2, then QMD would be the method of choice for all  $N$ . However, in terms of *actual* storage requirements, the RCM method is superior until  $N$  is about 1500!

This comparison also illustrates the potential importance of being able to use less storage for integers than that used for floating point numbers. In many circumstances, the number of binary digits used to represent floating point numbers is at least twice that necessary to represent integers of a sufficient magnitude. If it is convenient to exploit this fact, the significance of the overhead storage component will obviously be diminished. For example, if integers required only half as much storage as floating point numbers, the cross-over point between RCM and QMD would be at  $N \simeq 600$ , rather than *simeq*1500 as stated above.

3. Generally speaking, the information in Table 9.5.1 shows that while the more sophisticated ordering algorithms do succeed in reducing primary storage over their simpler counterparts, since they also necessitate the use of correspondingly more sophisticated storage schemes, the *net* reduction in storage requirements over the simpler schemes is not as pronounced as the relative differences in primary storage indicate. For example, primary storage requirements indicate that 1WD enjoys a storage saving of more than 50 percent over RCM, for  $N \leq 778$ , and that the advantage increases with  $N$ . However, the total storage requirements, while they still indicate that the storage advantage of 1WD over RCM increases with  $N$ , also show that the point at which a 50 percent savings occurs has still not been reached at  $N = 2233$ .

### 9.5.2 Execution Time

In Table 9.5.2 we have computed and tabulated the operations-per-second performance of the factorization and solution subroutines for the five methods, applied to test problem set #2. The information in the table suggests the following:

1. Generally speaking, the efficiency (i.e., operations-per-second) of the subroutines tends to improve with increasing  $N$ . This is to be expected since loops, once initiated, will tend to be executed more often as  $N$  increases. Thus, there will be less loop initialization overhead per arithmetic operation performed.

In this connection, note that the relative improvement from  $N = 265$  to  $N = 2233$  varies considerably over the six different subroutines and five different orderings involved. (Recall that the 1WD and RQT methods use the same numerical subroutines, as do the ND and QMD methods.) For example, the operations-per-second output for the ND solver (GSSLV) only improved from  $9.44 \times 10^4$  to  $10.15 \times 10^4$  over the full range of  $N$ , while the RQT solver (TSSLV) improved from  $6.07 \times 10^4$  to  $9.50 \times 10^4$ . These differences in improvement appear to be due to the variation in the number of auxiliary subroutines used. For example, TSFCT uses subroutines ELSLV, EUSLV, and ESFCT (which in turn uses ELSLV), while GSFCT uses none at all. These subroutine calls contribute a large low order component to the execution time.

These differences in the performance of the numerical subroutines illustrate how unrealistic it is to conclude much of a practical nature from a study of operation counts alone. For example, if we were to compare the RCM method to the QMD method on the basis of factorization operation counts, for the problems of test set #2, we would choose QMD for all the problems. However, in terms of execution time, QMD does not win until  $N$  reaches about 1600.

2. We have already observed that efficiency varies across the different subroutines, and varies with  $N$ . It is also interesting that for a fixed problem and subroutine, efficiency varies with the ordering used. As an example, consider the factorization entries for the 1WD and RQT methods, for  $N = 2233$ . (Remember that both methods employ the subroutine TSFCT.) This difference in efficiency can be understood by observing that unlike the subroutines ESFCT and GSFCT, where the

Factorization									
Number of Equations									
Method	265	406	577	778	1009	1270	1561	1882	2233
RCM	9.01	9.37	9.91	10.58	10.92	11.24	11.59	11.76	12.06
1WD	6.78	7.19	7.57	7.93	8.22	8.51	8.76	8.90	9.02
RQT	5.68	6.33	7.07	7.63	8.10	8.53	8.92	9.21	9.37
ND	7.07	7.42	7.75	7.78	8.20	8.38	8.52	8.62	8.69
QMD	7.30	8.10	8.19	8.87	9.01	9.27	9.39	9.59	9.88

Solution									
Number of Equations									
Method	265	406	577	778	1009	1270	1561	1882	2233
RCM	10.21	10.69	10.87	11.21	11.50	11.81	11.97	11.87	12.08
1WD	7.96	8.05	8.50	8.27	9.00	9.01	9.31	9.30	9.50
RQT	6.79	7.45	8.38	8.78	9.16	9.83	9.95	10.35	10.47
ND	10.25	10.28	10.49	9.81	10.71	10.83	10.89	11.00	11.20
QMD	9.77	10.52	9.98	10.65	10.50	10.65	10.89	11.12	10.89

Table 9.5.2: Operations-per-second for each method, applied to test problem #2. (Operations are scaled by  $10^{-4}$ )

majority of the numerical computation is isolated in a single loop, the numerical computation performed by TSFCT is distributed among three auxiliary subroutines (which vary in efficiency), in addition to a major computational loop of its own. Thus, one ordering may yield a more efficient TSFCT than another simply because a larger proportion of the associated computation is performed by the most efficient auxiliary subroutines or loop of TSFCT.

A final complicating factor in this study of the 1WD and RQT factorization entries is that apparently, the proportions of the computation performed by the different computational loops of TSFCT varies with  $N$ , and the variation with  $N$  is different for the one-way dissection ordering than it is for the refined quotient tree ordering. For small problems, TSFCT operates more efficiently on the one-way dissection ordering than on the refined quotient tree ordering, but as  $N$  increases, the situation is reversed, with the cross-over point occurring at about  $N = 1700$ .

We should caution the reader not to infer too much from this particular example. On a different computer with a different compiler and instruction set, the relative efficiencies of the computational loop in TSFCT and its auxiliary subroutines may be quite different. However, this example *does* illustrate that efficiency is not only a function of the data structure used, but may depend in a rather subtle way on the ordering used with that data structure, and on the problem size.



## Appendix A

# Some Hints on Using the Subroutines

### 1.1 Sample Skeleton Drivers

Different sparse methods have been described in Chapters 4 – 8 for solving linear systems. They differ in storage schemes, ordering strategies, data structures, and/or numerical subroutines. However, the overall procedure in using these methods is the same. Four distinct phases can be identified:

**Step 1** Ordering

**Step 2** Data structure set-up

**Step 3** Factorization

**Step 4** Triangular solution

Subroutines required to perform these steps for each method are included in Chapters 4 – 8. In Figures 1.1.1 – 1.1.3, three skeleton drivers are provided; they are for the envelope method (Chapter 4), the tree partitioning method (Chapter 6), and the nested dissection method (Chapter 8), respectively. They represent the sequence in which the subroutines should be called in the solution of a given sparse system by the selected scheme. Note that these are just *skeleton* programs; the various arrays are assumed to have been appropriately declared, and no checking for possible errors is performed. When an ordering subroutine is called, the zero-nonzero pattern of the sparse matrix  $\mathbf{A}$  is assumed to be in the adjacency structure pair ( $\mathbf{XADJ}$ ,  $\mathbf{ADJNCY}$ ).



---

```

C -----
C   CREATE XADJ AND ADJNCY
C   CORRESPONDING TO AX = B
C -----
C
C   .
C   .
C   .
C   CALL GENRCM(N,XADJ,ADJNCY,PERM,MASK,XLS)
C   CALL INVRSE(N,PERM,INVP)
C   CALL FNENV(N,XADJ,ADJNCY,PERM,INVP,XENV,ENVSZE,BANDW)
C
C -----
C   PUT NUMERICAL VALUES IN DIAG, ENV AND RHS
C -----
C
C   .
C   .
C   .
C   CALL ESFCT(N,XENV,ENV,DIAG,IERR)
C   CALL ELSLV(N,XENV,ENV,DIAG,RHS,IERR)
C   CALL EUSLV(N,XENV,ENV,DIAG,RHS,IERR)
C
C -----
C   PERMUTED SOLUTION IS NOW IN THE ARRAY RHS
C   RESTORE IT TO THE ORIGINAL ORDERING
C -----
C
C   CALL PERMRV(N,RHS,PERM)

```

Figure 1.1.1: Skeleton driver for the envelope method.

---

---

```

C -----
C   CREATE XADJ AND ADJNCY
C   CORRESPONDING TO AX = B
C -----
C
C   .
C   .
C   .
C   CALL GENRQT(N,XADJ,ADJNCY,NBLKS,XBLK,PERM,XLS,LS,NODLVL)
C   CALL BSHUFL(XADJ,ADJNCY,PERM,NBLKS,XBLK,BNUM,MASK,SUBG,XLS)
C   CALL INVRSE(N,PERM,INVP)
C   CALL FNTADJ(XADJ,ADJNCY,PERM,INVP,NBLKS,XBLK,FATHER,MASK)
C   CALL FNTENV(XADJ,ADJNCY,PERM,INVP,NBLKS,XBLK,XENV,ENVSZE)
C   CALL FNOFNZ(XADJ,ADJNCY,PERM,INVP,NBLKS,XBLK,XNONZ,
C             NZSUBS,NOFNZ)
C
C -----
C   PUT NUMERICAL VALUES INTO NONZ,DIAG,ENV AND RHS
C -----
C
C   .
C   .
C   .
C   CALL TSFCT(NBLKS,XBLK,FATHER,DIAG,XENV,ENV,XNONZ,NONZ,
C             NZSUBS,TEMPV,FIRST,IERR)
C   CALL TSSLV(NBLKS,XBLK,DIAG,XENV,ENV,XNONZ,NONZ,NZSUBS,
C             RHS,TEMPV)
C
C -----
C   PERMUTED SOLUTION IS NOW IN THE ARRAY RHS
C   RESTORE IT TO THE ORIGINAL ORDERING
C -----
C
C   CALL PERMRV(N,RHS,PERM)

```

Figure 1.1.2: Skeleton driver for the tree partitioning method.

---

---

```

C -----
C CREATE XADJ AND ADJNCY
C CORRESPONDING TO AX = B
C -----
C
C .
C .
C .
C CALL GENND(N,XADJ,ADJNCY,MASK,PERM,XLS,LS)
C CALL INVRSE(N,PERM,INVP)
C CALL SMBFCT(N,XADJ,ADJNCY,PERM,INVP,XLNZ,NOFNZ,XNZSUB,
C           NZSUB,NOFSUB,RCHLNK,MRGLNK,MASK,FLAG)
C
C -----
C PUT NUMERICAL VALUES IN LNZ,DIAG, AND RHS
C -----
C
C .
C .
C .
C CALL GSFCT(N,XLNZ,LNZ,XNZSUB,NZSUB,DIAG,LINK,FIRST,
C           TEMPV,IERR)
C CALL GSSLV(N,XLNZ,LNZ,XNZSUB,NZSUB,DIAG,RHS)
C
C -----
C PERMUTED SOLUTION IS NOW IN THE ARRAY RHS
C RESTORE IT TO THE ORIGINAL ORDERING
C -----
C
C CALL PERMRV(N,RHS,PERM)

```

Figure 1.1.3: Skeleton driver for the nested dissection method.

---

It is rare that the user has this representation provided for him. Thus, the user must create this structure prior to the execution of the ordering step. The creation of the adjacency structure is not a trivial task, especially in situations where the  $(i, j)$  pairs for which  $a_{ij} \neq 0$  become available in random order. We shall not concern ourselves with this problem here. Exercises 3.4.1 and 3.4.6 in Chapter 3 indicate how part of this problem can be solved. The package SPARSPAK to be discussed in Appendix B provides ways to generate the adjacency structure pair in the (XADJ, ADJNCY) format. In the skeleton drivers, there are two subroutines that have not been discussed before. The subroutine INVRSE, called after the ordering PERM has been determined, is used to compute the inverse INVP of the ordering (or permutation) found. The vector INVP is required in setting up data structures for the solution scheme, and in putting numerical values into them. After the numerical subroutines for factorization and triangular solutions have been executed, the solution  $\tilde{\mathbf{a}}$  obtained is that for the *permuted* system

$$(\mathbf{PAP}^T)\tilde{\mathbf{x}} = \mathbf{Pb} .$$

The subroutine PERMRV is used to permute the vector  $\tilde{\mathbf{x}}$  back to the original given order.

After the data structure for the triangular factor has been successfully set up, the user must input the actual numerical values for the matrix  $\mathbf{A}$  and the right hand side  $\mathbf{b}$ . To insert values into the data structure, the user must understand the storage scheme in detail. In the next section, a sample subroutine is provided for matrix input. For different storage methods, these matrix input subroutines are obviously different. With the sample provided in the next section, the user should be able to write those for the other methods. It should be pointed out that they are all provided in SPARSPAK (see Appendix B).

## 1.2 A Sample Numerical Value Input Subroutine

Before the numerical subroutines for a sparse method are called, it is necessary to put the numerical values into the data structure. Here, we provide a sample subroutine for the tree-partitioning method, whereby the numerical values of an entry  $a_{ij}$  can be placed into the structure.

Recall from Chapter 6 that there are three vectors in the storage scheme containing numerical values. The vector DIAG contains the diagonal elements of the matrix. The entries within the envelope of the diagonal blocks are

stored in ENV, while the vector NONZ keeps all the nonzero off-diagonal entries. For a given nonzero entry  $a_{ij}$ , the subroutine ADAIJ updates one of the three storage vectors DIAG, ENV, or NONZ, depending on where the value  $a_{ij}$  resides in the matrix.

The calling statement to the matrix input subroutine is `CALL ADAIJ ( I,J,VALUE,INVP,DIAG, )` where I and J are the subscripts of the original matrix  $A$  (that is, unpermuted) and VALUE is the numerical value. This subroutine adds VALUE to the appropriate current value of  $a_{ij}$  in storage. This is used instead of an assignment so as to handle situations when the values of  $a_{ij}$  are obtained in an incremental fashion (such as in certain finite element applications).

The subroutine checks to see if the nonzero component lies on the diagonal or within the envelope of the diagonal blocks. If so, the value is added to the appropriate location in DIAG or ENV. Otherwise, the subscript structure (XNONZ, NZSUBS) for off-diagonal block nonzeros is searched and VALUE is then added to the appropriate entry of the vector NONZ.

Since ADAIJ only adds new values to those currently in storage, the space used for  $L$  must be initialized to zero before numerical values of  $A$  are supplied. Therefore, the input of values of  $A$  for the tree-partitioning method would be done as follows:

- Initialize the vectors DIAG, ENV and NONZ to zeros.
- {Repeated calls to ADAIJ}.

The input of values for the right hand vector  $b$  can be performed in a similar way.

---

```

1. C*****
2. C*****
3. C*****      ADAIJ      ....  ADD ENTRY INTO MATRIX      *****
4. C*****
5. C*****
6. C
7. C      PURPOSE - THIS ROUTINE ADDS A NUMBER INTO THE (I,J)-TH
8. C                  POSITION OF A MATRIX STORED USING THE
9. C                  IMPLICIT BLOCK STORAGE SCHEME.
10. C
11. C      INPUT PARAMETERS -
12. C          (ISUB, JSUB) - SUBSCRIPTS OF THE NUMBER TO BE ADDED
13. C          ASSUMPTIONS - ISUB .GE. JSUB.
14. C          DIAG - ARRAY CONTAINING THE DIAGONAL ELEMENTS
15. C                  OF THE COEFFICIENT MATRIX.
16. C          VALUE - VALUE OF THE NUMBER TO BE ADDED.

```

```

17. C          INVP - INVP(I) IS THE NEW POSITION OF THE
18. C             VARIABLE WHOSE ORIGINAL NUMBER IS I.
19. C          (XENV, ENV) - ARRAY PAIR CONTAINING THE ENVELOPE
20. C             STRUCTURE OF THE DIAGONAL BLOCKS.
21. C          (XNONZ, NONZ, NZSUBS) - LEVEL STRUCTURE CONTAINING
22. C             THE OFF-BLOCK DIAGONAL PARTS OF THE ROWS OF
23. C             THE LOWER TRIANGLE OF THE ORIGINAL MATRIX.
24. C
25. C          OUTPUT PARAMETERS -
26. C             IERR - ERROR CODE....
27. C                 0 - NO ERRORS DETECTED
28. C                 5 - NO SPACE IN DATA STRUCTURE FOR NUMBER
29. C                    WITH SUBSCRIPTS (I,J), I>J.
30. C
31. C*****
32. C
33. C          SUBROUTINE ADAIJ ( ISUB, JSUB, VALUE, INVP, DIAG,
34. C             1             XENV, ENV, XNONZ, NONZ, NZSUBS,
35. C             2             IERR )
36. C
37. C*****
38. C
39. C          REAL DIAG(1), ENV(1), NONZ(1), VALUE
40. C          INTEGER INVP(1), NZSUBS(1)
41. C          INTEGER XENV(1), XNONZ(1), KSTOP, KSTRT,
42. C             1             I, IERR, ISUB, ITEMP, J, JSUB, K
43. C
44. C*****
45. C             I = INVP(ISUB)
46. C             J = INVP(JSUB)
47. C             IF ( I .EQ. J ) GO TO 400
48. C             IF ( I .GT. J ) GO TO 100
49. C                 ITEMP = I
50. C                 I      = J
51. C                 J      = ITEMP
52. C             -----
53. C             THE COMPONENT LIES WITHIN THE DIAGONAL ENVELOPE.
54. C             -----
55. C             100  K = XENV(I+1) - I + J
56. C                 IF ( K .LT. XENV(I) ) GO TO 200
57. C                 ENV(K) = ENV(K) + VALUE
58. C                 RETURN
59. C             -----
60. C             THE COMPONENT LIES OUTSIDE DIAGONAL BLOCKS.
61. C             -----
62. C             200  KSTRT = XNONZ(I)
63. C                 KSTOP = XNONZ(I+1) - 1

```

```

64.             IF ( KSTOP .LT. KSTRT ) GO TO 500
65. C
66.             DO 300 K = KSTRT, KSTOP
67.                 IF ( NZSUBS(K) .NE. J ) GO TO 300
68.                     NONZ(K) = NONZ(K) + VALUE
69.                 RETURN
70.     300      CONTINUE
71.             GO TO 500
72. C
73. C             -----
74. C             THE COMPONENT LIES ON THE DIAGONAL OF THE MATRIX.
75. C             -----
76.     400      DIAG(I) = DIAG(I) + VALUE
77.             RETURN
78. C             -----
79. C             SET ERROR FLAG.
80. C             -----
81.     500      IERR = 5
82.             RETURN
83.             END

```

---

### 1.3 Overlaying Storage in Fortran

Consider the skeleton driver in Figure 1.1.1 for the envelope method. The ordering subroutine `GENRCM` generates an ordering `PERM` based on the adjacency structure (`XADJ`, `ADJNCY`). It also uses two working vectors `MASK` and `XLS`.

After the input of numerical values into the data structure for the envelope, note that the working vectors `MASK` and `XLS` are no longer needed. Moreover, even the adjacency structure (`XADJ`, `ADJNCY`) will no longer be used. To conserve storage, these vectors can be overlaid and re-used by the solution subroutines. Similar remarks apply to the other sparse methods.

In this section, we show how overlaying can be done in Fortran. The general technique involves the use of a large working storage array in the driver program. Storage management can be handled by this driver through the use of pointers into the main storage vector.

As an illustration, suppose that there are two subroutines `SUB1` and `SUB2`:

```

SUBROUTINE SUB1 (X,Y,Z)
SUBROUTINE SUB2 (X,Y,U,V) .

```

The subroutine `SUB1` requires two integer arrays `X` and `Y` of sizes 100 and 500 respectively, and a working integer array `Z` of size 400. On the other hand, `SUB2` requires four vectors: the `X` and `Y` output vectors from `SUB1` and two additional arrays `U` and `V` of sizes 40 and 200 respectively.

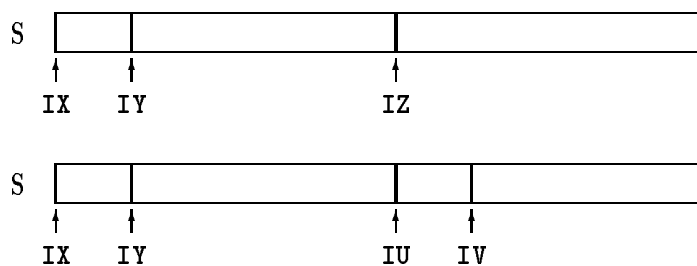


Figure 1.3.1: Storage management by pointers into the main storage vector.

---

The following skeleton driver makes use of a main storage vector `S(1000)` and calls the subroutines `SUB1` and `SUB2` in succession. It manages the storage using pointers into the array `S`.

```

      INTEGER S(1000)
      .
      .
      .
      IX = 1
      IY = IX + 100
      IZ = IY + 500
      CALL SUB1 (S(IX),S(IY),S(IZ))
      .
      .
      .
      IU = IY + 500
      IV = IU + 40
      CALL SUB2 (S(IX),S(IY),S(IU),S(IV))
      .
      .
      .

```



In this way, the storage used by the working vector  $Z$  can be overlaid by  $U$  and  $V$ .

The same overlay technique can be used in invoking the sequence of subroutines for a sparse solution method. The package SPARSPAK (Appendix B) uses essentially this same technique in a system of *user interface subroutines* which relieve the user of all the storage management tasks associated with using the subroutines in this book.

## Appendix B

# SPARSPAK: A Sparse Matrix Package

### 2.1 Motivation

The skeleton programs in Appendix A illustrate several important characteristics of sparse matrix programs and subroutines. First, the unconventional data structures employed to store sparse matrices result in subroutines which have distressingly long parameter lists, most of which have little or no meaning to the user unless he or she understands and remembers the details of the data structure being employed. Second, the computation consists of several distinct phases, with numerous opportunities to overlay (re-use) storage. In order to use the subroutines effectively, the user must determine which arrays used in one module must be preserved as input to the next, and which ones are no longer required and can therefore be re-used. Third, in all cases, the amount of storage required for the solution phase is unknown until at least part of the computation has been performed. Usually we do not know the maximum storage requirement until the allocation subroutine (e.g., `FNENV`) has been executed. In some cases, the storage requirement for the successful execution of the allocation subroutine *itself* is not predictable (e.g., `SMBFCT`). Thus, often the computation must be suspended part way through because of insufficient storage, and if the user wishes to avoid repeating the successfully completed part, then he or she must be aware of all the information required to restart the computation.

These observations, along with our experience in using sparse matrix software, have prompted us to design and implement a *user interface* for the

subroutines described in this book. This interface is simply a layer of subroutines between the user, who presumably has a sparse system of equations to solve, and subroutines which implement the various methods described in this book, as depicted in Figure 2.1.1. The interface, along with the subroutines it serves, forms a package which has been given the name SPARSPAK (George [25]). In addition to the subroutines from Chapters 4 – 8 and the interface subroutines, SPARSPAK also contains a number of utility subroutines for printing error messages, pictures of the structure of sparse matrices, etc.

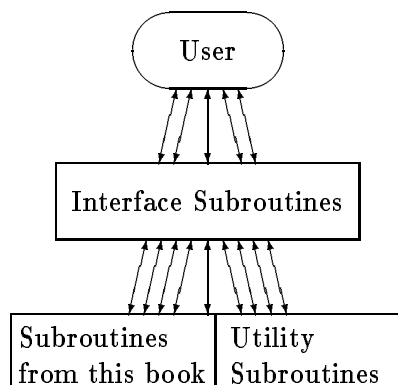


Figure 2.1.1: Schematic of the components of SPARSPAK.

---

The interface provides a number of services. First, it relieves the user of all responsibility for the allocation of array storage. All storage is allocated by the interface from a user-supplied one-dimensional array, using a technique similar to that described in Section 1.3. The interface also imposes sequencing control so that interface subroutines are called in the correct order. In addition, it provides a convenient means by which computation can be suspended and later restarted. Finally, it has comprehensive error diagnostics.

Our objective in subsequent sections is to give a brief survey of the various features of SPARSPAK, rather than to provide a detailed user guide. A comprehensive user guide and installation instructions are provided with the package. For information, the interested reader should write the authors.

## 2.2 Basic Structure of SPARSPAK

For all the methods described in Chapters 4 through 8, the user and SPARSPAK interact to solve the problem  $\mathbf{Ax} = \mathbf{b}$  through the following basic steps.

**Step 1** (Structure Input) The user supplies the nonzero structure of  $\mathbf{A}$  to the package by calling the appropriate interface subroutines.

**Step 2** (Order and Allocate) The execution by the user program of a single subroutine call instructs the package to find an ordering and set up the data structure for  $\mathbf{L}$ .

**Step 3** (Matrix Input) The user supplies the numerical values for  $\mathbf{A}$  by calling appropriate interface subroutines.

**Step 4** (Factor  $\mathbf{A}$ ) A single subroutine call tells SPARSPAK to factor  $\mathbf{A}$  into  $\mathbf{LL}^T$ .

**Step 5** (Right Hand Side Input) The user supplies the numerical values for  $\mathbf{b}$  by calling appropriate interface subroutines. (This step can be done before Step 4, and/or intermixed with Step 3.)

**Step 6** (Solution) A single subroutine call instructs SPARSPAK to compute  $\mathbf{x}$ , using  $\mathbf{L}$  from Step 4 and the  $\mathbf{b}$  supplied in Step 5.

A list of the names of some of the interface subroutines, along with their argument lists and general roles is given in Figure 2.2.1. Details are provided later in this and subsequent sections.

## 2.3 User Mainline Program and an Example

SPARSPAK allocates all its storage from a single one dimensional real array which for purposes of discussion we will denote by  $\mathbf{S}$ . In addition, the user must provide its size  $\mathbf{MAXS}$ , which is transmitted to the package via a common block `/SPKUSR/`, which has four variables:

```
COMMON /SPKUSR/ MSGLVL, IERR, MAXS, NEQNS
```

Here `MSGLVL` is the message level indicator which is used to control the amount of information printed by the package. The second variable `IERR` is an error code, which the user can examine in the mainline program for

---

SPARSPAK	}	Initialization
IJBEGN	}	Structure input (Step 1)
INIJ(I, J, S)		
INROW(I, NR, IR, S)		
INIJIJ(NIR, II, JJ, S)		
INCLQ(NCLQ, CLQ, S)		
IJEND(S)		
ORDRxi(S)	}	Ordering and Allocation (Step 2. See Figure 2.3.1 for meanings of $x$ and $i$ .)
INAIJi(I, J, VALUE, S)	}	Matrix input (Step 3)
INROWi(I, NIR, IR, VALUES, S)		
INMATi(NIJ, II, JJ, VALUES, S)		
INBI(I, VALUE, S)	}	Right hand side input (Step 5)
INBIBI(NI, II, VALUES, S)		
INRHS(RHS, S)		
SOLVEi(S)	}	Factorization and Solution (Steps 4 and 6)

---

Figure 2.2.1: List of names of some of the SPARSPAK interface subroutines.

possible errors detected by the package. The variable NEQNS is the number of equations, set by the package.

The following program illustrates how one might use the envelope method of Chapter 4 to solve a system of equations, using SPARSPAK. The problem solved is a 10 by 10 symmetric tridiagonal system  $\mathbf{Ax} = \mathbf{b}$  where the diagonal elements of  $\mathbf{A}$  are all 4, the superdiagonal and subdiagonal elements are all  $-1$ , and the entries in the right hand side vector  $\mathbf{b}$  are all ones.

The digit  $i$  and letter  $x$  in some of the interface subroutine names specify which method is to be used to solve the problem. We should note here that SPARSPAK handles both symmetric and unsymmetric  $\mathbf{A}$ , but assumes that the *structure* of  $\mathbf{A}$  is symmetric, and that no pivoting is required for numerical stability. (See Exercise 4.6.1.) The methods available are as indicated in Figure 2.3.1.

<i>ORDRxi</i>		Ordering Choices	Ref.
<i>x</i>	<i>i</i>		
<b>A</b>	1	Reverse Cuthill-McKee ordering; symmetric $\mathbf{A}$	Ch. 4
<b>A</b>	2	Reverse Cuthill-McKee ordering; unsymmetric $\mathbf{A}$	Ch. 4
<b>A</b>	3	One-way Dissection ordering; symmetric $\mathbf{A}$	Ch. 7
<b>A</b>	4	One-way Dissection ordering; unsymmetric $\mathbf{A}$	Ch. 7
<b>B</b>	3	Refined quotient tree ordering; symmetric $\mathbf{A}$	Ch. 6
<b>B</b>	4	Refined quotient tree ordering; unsymmetric $\mathbf{A}$	Ch. 6
<b>A</b>	5	Nested Dissection ordering; symmetric $\mathbf{A}$	Ch. 8
<b>A</b>	6	Nested Dissection ordering; unsymmetric $\mathbf{A}$	Ch. 8
<b>B</b>	5	Minimum Degree ordering; symmetric $\mathbf{A}$	Ch. 5
<b>B</b>	6	Minimum Degree ordering; unsymmetric $\mathbf{A}$	Ch. 5

Figure 2.3.1: Choices of methods available in SPARSPAK.

---

```

1. C      SAMPLE PROGRAM ILLUSTRATING THE USE OF SPARSPAK
2. C      -----
3. C
4.        COMMON /SPKUSR/ MSGVLV, IERR, MAXS, NEQNS
5.        REAL S(250)
6. C
7.        CALL SPRSPK
8.        MAXS = 250
9. C      -----
10. C     INPUT THE MATRIX STRUCTURE. THE DIAGONAL IS ALWAYS
11. C     ASSUMED TO BE NONZERO, AND SINCE THE MATRIX IS SYM-
```

```

12. C      METRIC, ONLY THE SUBDIAGONAL POSITIONS ARE INPUT.
13. C      -----
14. C      CALL IJBEGM
15. C      DO 100 I = 2, 10
16. C          CALL INIJ ( I, I-1, S )
17. C      100 CONTINUE
18. C      CALL IJEND ( S )
19. C      -----
20. C      FIND THE ORDERING AND ALLOCATE STORAGE ....
21. C      -----
22. C      CALL ORDRA1 ( S )
23. C      -----
24. C      INPUT THE NUMERICAL VALUES. (LOWER TRIANGLE ONLY.)
25. C      -----
26. C      DO 200 I = 1, 10
27. C          IF ( I .GT. 1 ) CALL INAIJ1 ( I, I-1, -1.0, S )
28. C          CALL INAIJ1 ( I, I, 4.0, S )
29. C          CALL INBI ( I, 1.0, S )
30. C      200 CONTINUE
31. C      -----
32. C      SOLVE THE SYSTEM. SINCE BOTH THE MATRIX AND RIGHT HAND
33. C      SIDE HAVE BEEN INPUT, BOTH THE FACTORIZATION AND THE
34. C      TRIANGULAR SOLUTION OCCUR.
35. C      -----
36. C      CALL SOLVE1 ( S )
37. C      -----
38. C      PRINT THE SOLUTION, FOUND IN THE FIRST 10 POSITIONS OF
39. C      THE WORKING STORAGE ARRAY S.
40. C      -----
41. C      WRITE ( 6, 11 ) ( S(I), I = 1, 10 )
42. C      11 FORMAT ( / 10H SOLUTION  ,/, (5F10.6) )
43. C      -----
44. C      PRINT SOME STATISTICS GATHERED BY THE PACKAGE.
45. C      -----
46. C      CALL PSTATS
47. C      STOP
48. C      END

```

---

The subroutine SPSPK must be called before any part of the package is used. Its role is to initialize some system parameters (e.g., the logical unit number for the printer), to set default values for options (e.g., the message level indicator), and to perform some installation dependent functions (e.g., initializing the timing subroutine). It needs only to be called once in the user program. Note that the only variable in the common block /SPKUSR/ that must be explicitly assigned a value by the user is MAXS.

SPARSPAK contains an interface subroutine called `PSTATS` which the user can call to obtain storage requirements, execution times, operation counts etc. for the solution of the problem.

It is assumed that the subroutines which comprise the SPARSPAK package have been compiled into a *library*, and that the user can reference them from a Fortran program just as the standard Fortran library subroutines, such as `SIN`, `COS`, etc., are referenced. Normally, a user will use only a small fraction of the subroutines provided in SPARSPAK.

## 2.4 Brief Description of the Main Interface Subroutines

### 2.4.1 Modules for Input of the Matrix Structure

SPARSPAK must know the matrix structure before it can determine an appropriate ordering for the system. SPARSPAK contains a group of subroutines which provide a variety of ways through which the user can inform the package where the nonzero entries are; that is, those subscripts  $(i, j)$  for which  $a_{ij} \neq 0$ . Before any of these input subroutines is called, the user must execute an initialization subroutine called `IJBEGN`, which tells the package that a matrix problem with a new structure is to be solved.

#### a) Input of a nonzero location

To tell SPARSPAK that the matrix component  $a_{ij}$  is nonzero, the user simply executes the statement

```
CALL INIJ ( I, J, S )
```

where `I` and `J` are the subscripts of the nonzero, and `S` is the working storage array declared by the user for use by the package.

#### b) Input of the structure of a row, or part of a row.

When the structure of a row or part of a row is available, it may be more convenient to use the subroutine `INROW`. The statement to use is

```
CALL INROW ( I, NIR, IR, S )
```

where `I` denotes the subscript of the row under consideration, `IR` is an array containing the column subscripts of some or all of the nonzeros in the `I`-th row, `NIR` is the number of subscripts in `IR`, and `S` is the



user-declared working storage array. The subscripts in the array `IR` can be in arbitrary order, and the rows can be input in any order.

c) Input of a submatrix structure

SPARSPAK allows the user to input the structure of a submatrix. The calling statement is

```
CALL INIJIJ ( NIJ, II, JJ, S ) ,
```

where `NIJ` is the number of input subscript pairs and `II`, `JJ` are the arrays containing the subscripts.

d) Input of a full submatrix structure

The structure of an entire matrix is completely specified if all the full submatrices are given. In applications where they are readily available, the subroutine `INCLQ` is useful. Its calling sequence is

```
CALL INCLQ (NCLQ, CLQ, S) ,
```

where `NCLQ` is the size of the submatrix and `CLQ` is an array containing the subscripts of the submatrix.

Thus, to inform the package that the submatrix corresponding to subscripts 1, 3, 5 and 6 is full, we execute

```
CLQ(1) = 1
CLQ(2) = 3
CLQ(3) = 5
CLQ(4) = 6
CALL INCLQ(4, CLQ, S) .
```

The type of structure input subroutine to use depends on how the user obtains the matrix structure. Anyway, one can select those that best suit the application. The package allows *mixed use* of the subroutines in inputting a matrix structure. SPARSPAK automatically removes duplications so the user does not have to worry about inputting duplicate subscript pairs.

When all pairs have been input, using one or a combination of the input subroutines, the user is required to tell the package explicitly so by calling the subroutine `IJEND`. The calling statement is

```
CALL IJEND(S)
```

and its purpose is to transform the data from the format used during the recording phase to the standard (XADJ, ADJNCY) format used by all the subroutines in the book. The user does not have to be concerned with this input representation or the transformation process.

### 2.4.2 Modules for Ordering and Storage Allocation

With an internal representation of the nonzero structure of the matrix  $A$ , SPARSPAK is now ready to reorder the matrix problem. The user initiates this by calling an ordering subroutine, whose name has the form `ORDRxi`. Here  $i$  is a numerical digit between 1 and 6 that signifies the storage method, and the character  $x$  denotes the ordering strategy as summarized in Figure 2.3.1. The subroutine `ORDRxi` determines the ordering and then sets up the data structure for the reordered matrix problem. The package is now ready for numerical inputs.

### 2.4.3 Modules for Inputting Numerical Values of $A$ and $b$

The modules in this group are similar to those for inputting the matrix structure. They provide a means of transmitting the actual numerical values of the matrix problem to the package. Since the data structures for different storage methods are different, the package must have a different matrix input subroutine for each method. SPARSPAK uses the same set of subroutine names for all the methods (except for the last digit which distinguishes the method), and the parameter lists for all the methods are the same.

There are three ways of passing the numerical values to the package. In all of them, subscripts passed to the package always refer to those of the *original* given problem. The user need not be concerned about the various permutations to the problem which may have occurred during the ordering step.

a) Input of a single nonzero component

The subroutine `INAIJi` is provided for this purpose and its calling sequence is

```
CALL INAIJi ( I, J, VALUE, S )
```

where  $I$  and  $J$  are the subscripts, and `VALUE` is the numerical value. The subroutine `INAIJi` adds the quantity `VALUE` to the appropriate current value in storage, rather than making an assignment. This is

helpful in situations (e.g., in some finite element applications) where the numerical values are obtained in an incremental fashion. For example, the execution of

```

      .
      .
      .
      INAIJ2 ( 3, 4, 9.5, S )
      INAIJ2 ( 3, 4, -4.0, S )
      .
      .
      .

```

effectively assigns 5.5 to the matrix component  $a_{34}$ .

b) Input of a row of nonzeros

The subroutine `INROW $i$`  can be used to input the numerical values of a row or part of a row in the matrix. Its calling sequence is similar to that of `INROW`, described in Section 2.4.1.

```
CALL INROW $i$  ( I, NIR, IR, VALUES, S ) .
```

Here the additional variable `VALUES` is an array containing the numerical values of the row. Again, the numerical values are added to the current values in storage.

c) Input of a submatrix

The subroutine for the input of a submatrix is called `INMAT $i$` . Its parameter list corresponds to that of `INIJIJ` with the additional parameter `VALUES` that stores the numerical quantities:

```
CALL INMAT $i$  ( NIJ, II, JJ, VALUES, S ) .
```

Again, the `VALUES` are added to those held by the package.

Mixed use of the subroutines `INAIJ $i$` , `INROW $i$` , and `INMAT $i$`  is permitted. Thus, the user is free to use whatever subroutines are most convenient.

The same convenience is provided in the input of numerical values for the right hand side vector. The package includes the subroutine `INBI` which inputs an entry to the right hand vector.

```
CALL INBI ( I, VALUE, S )
```

Here **I** is the subscript and **VALUE** is the numerical value.

The subroutine **INBIBI** can be used to input a subvector, and its calling sequence is

```
CALL INBIBI ( NI, II, VALUES, S )
```

where **II** and **VALUES** are vectors containing the subscripts and numerical values respectively. In both subroutines, incremental calculations of the numerical values are performed.

In some situations where the entire right hand vector is available, the user can use the subroutine **INRHS** which transmits the whole vector to the package. It has the form

```
CALL INRHS ( RHS, S )
```

where **RHS** is the vector containing the numerical values.

In all three subroutines, the numbers provided are added to those currently held by the package, and the use of the subroutines can be intermixed. The storage used for the right hand side by the package is initialized to zero the first time any of them is executed.

#### 2.4.4 Modules for Factorization and Solution

The numerical computation of the solution vector is initiated by the Fortran statement

```
CALL SOLVE $i$  ( S )
```

where **S** is the working storage array for the package. Again, the last digit  $i$  is used to distinguish between solvers for different storage methods.

Internally, the subroutine **SOLVE $i$**  consists of *both* the factorization and forward/backward solution steps. If the factorization has been performed in a previous call to **SOLVE $i$** , the package will automatically skip the factorization step, and perform the solution step directly. The solution vector is returned in the first **NEQNS** locations of the storage vector **S**. If **SOLVE $i$**  is called before any right hand side values are input, only the factorization will be performed. The solution returned will be all zeros.

## 2.5 Save and Restart Facilities

SPARSPAK provides two subroutines called `SAVE` and `RESTRT` which allow the user to stop the calculation at some point, save the results on an external sequential file, and then restart the calculation at exactly that point some time later. To save the results of the computation done thus far, the user executes the statement

```
CALL SAVE ( K, S )
```

where `K` is the Fortran logical unit on which the results are to be written, along with other information needed to restart the computation. If execution is then terminated, the state of the computation can be re-established by executing the statement

```
CALL RESTRT ( K, S ) .
```

When an error is detected, so that the computation cannot proceed, a positive code is assigned to `IERR`. The user can simply check the value of `IERR` to see if the execution of the module has been successful. This error flag can be used in conjunction with the save/restart feature to retain the results of successfully completed parts of the computation, as shown by the program fragment below.

```

.
.
.
CALL ORDRA1 ( S )
IF (IERR.EQ.0) GO TO 100
CALL SAVE ( 3, S )
STOP
100 CONTINUE
.
.
.
```

Another potential use of the `SAVE` and `RESTRT` modules is to make the working storage array `S` available to the user in the middle of a sparse matrix computation. After `SAVE` has been executed, the working storage array `S` can be used by some other computation.

## 2.6 Solving Many Problems Having the Same Structure or the Same Coefficient Matrix $A$

In certain applications, many problems which have the same sparsity structure, but different numerical values, must be solved. This situation can be accommodated perfectly well by the package. The control sequence is depicted by the flowchart in Figure 2.6.1. When the numerical input subroutines ( $INAIJ_i$ ,  $INBI$ , etc.) are first called after  $SOLVE_i$  has been called, this is detected by the package, and the computer storage used for  $A$  and  $b$  is initialized to zero.

---

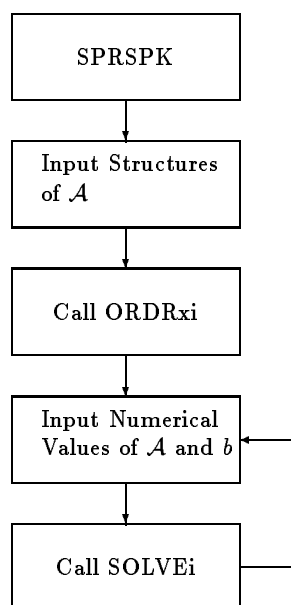


Figure 2.6.1: Flowchart for using SPARSPAK to solve numerous problems having the same structure.

---

Note that if such problems must be solved over an extended time period (i.e., in different runs), the user can execute `SAVE` after executing `ORDRxi` and thus avoid input of the structure of  $A$  and the execution of `ORDRxi` in subsequent equation solutions.

In other applications, numerous problems which differ only in their right hand sides must be solved. In this case, we only want to factor  $\mathbf{A}$  once, and use the factors repeatedly in the calculation of  $\mathbf{x}$  for each different  $\mathbf{b}$ . Again, the package can handle this in a straightforward manner, as illustrated by the flowcharts in Figure 2.6.2.

When SPARSPAK is used as indicated by flowchart (1) in Figure 2.6.2, it detects that no right hand side has been provided during the first execution of `SOLVE $i$` , and only the factorization is performed. In subsequent calls to `SOLVE $i$` , the package detects that the factorization has already been performed, and that part of the `SOLVE $i$`  module is by-passed. In flowchart (2) of Figure 2.6.2, both factorization and solution is performed during the first call to `SOLVE $i$` , with only the solve part performed in subsequent executions of `SOLVE $i$` .

Note that `SAVE` can be used after `SOLVE $i$`  has been executed, if the user wants to save the factorization for use in some future calculation.

## 2.7 Output From the Package

As noted earlier, the user supplies a one-dimensional real array `S`, from which all array storage is allocated. In particular, the interface allocates the first `NEQNS` storage locations in `S` for the solution vector of the linear system. After all the interface modules for a particular method have been successfully executed, the user can retrieve the solution from these `NEQNS` locations.

In addition to the solution  $\mathbf{x}$ , the package may print other information about the computation, depending upon the value of `MSGVLV`, whether or not errors occur, and whether or not the module `PSTATS` is called.

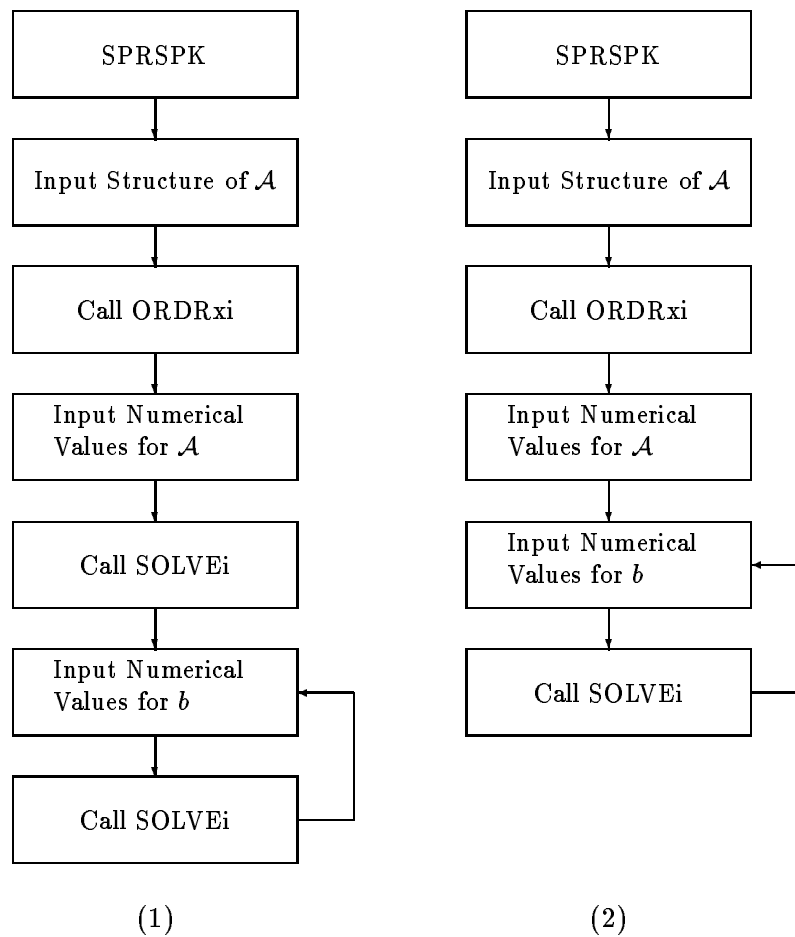


Figure 2.6.2: Flowcharts for using SPARSPAK to solve numerous problems having the same coefficient matrix but different right hand sides.

---





# Bibliography

- [1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974.
- [2] I. Arany, W.F. Smyth, and L. Szoda. An improved method for reducing the bandwidth of sparse symmetric matrices. In *Information Processing 71*, Amsterdam, 1972. North-Holland. (Proceedings of IFIP Congress).
- [3] C. Berge. *The Theory of Graphs and its Applications*. John Wiley & Sons Inc., New York, 1962.
- [4] B. Buchberger, G. E. Collins, and R. Loos, editors. *Matrix Iterative Analysis*. Prentice-Hall, Englewood Cliff, New Jersey, 1962.
- [5] J.R. Bunch and J.E. Hopcroft. Triangular factorization and inversion by fast matrix multiplication. *Math. Comp.*, 28:231–236, 1974.
- [6] D.A. Calahan. Complexity of vectorized solution of two dimensional finite element grids. Technical Report Tech. Rept. 91, Systems Engrg. Lab., University of Michigan, 1975.
- [7] D.A. Calahan, W.N. Joy, and D.A. Orbits. Preliminary report on results of matrix benchmarks on a vector processor. Technical Report Tech. Rept. 94, Systems Engrg. Lab., University of Michigan, 1976.
- [8] W.M. Chan and A. George. A linear time implementation of the reverse Cuthill-McKee algorithm. *BIT*, 20:8–14, 1980.
- [9] E. Cuthill. Several strategies for reducing bandwidth of matrices. In D. J. Rose and R. A. Willoughby, editors, *Sparse Matrices and their Applications*, New York, 1972. Plenum Press.

- [10] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings 24th ACM National Conference*, pages 157–172, aug 1969.
- [11] I.S. Duff, A.M. Erisman, and J.K. Reid. On George’s nested dissection method. *SIAM J. Numer. Anal.*, 13:686–695, 1976.
- [12] I.S. Duff and J.K. Reid. A comparison of sparsity orderings for obtaining a pivotal sequence in Gaussian elimination. *J. Inst. Maths. Appl.*, 14:281–291, 1974.
- [13] S.C. Eisenstat, M.C. Gursky, M.H. Schultz, and A. H. Sherman. The Yale sparse matrix package I. the symmetric codes. *Internat. J. Numer. Meth. Engrg.*, 18:1145–1151, 1982.
- [14] S.C. Eisenstat, M.H. Schultz, and A.H. Sherman. Applications of an element model for Gaussian elimination. In J. R. Bunch and D. J. Rose, editors, *Sparse Matrix Computations*, pages 85–96. Academic Press, 1976.
- [15] C.A. Felippa. Solution of linear equations with skyline-stored symmetric matrix. *Computers and Structures*, 5:13–29, 1975.
- [16] C.A. Felippa and R.W. Clough. The finite element method in solid mechanics. In *Numerical Solution of Field Problems in Continuum Mechanics*, volume 5, pages 210–252. SIAM-AMS Proc. Amer. Math. Soc., 1970.
- [17] J.A. George. *Computer Implementation of the Finite Element Method*. PhD thesis, Dept. of Computer Science, Stanford University, 1971.
- [18] J.A. George. Nested dissection of a regular finite element mesh. *SIAM J. Numer. Anal.*, 10:345–363, 1973.
- [19] J.A. George. On block elimination for sparse linear systems. *SIAM J. Numer. Anal.*, 11:585–603, 1974.
- [20] J.A. George. Numerical experiments using dissection methods to solve  $n$  by  $n$  grid problems. *SIAM J. Numer. Anal.*, 14:161–179, 1977.
- [21] J.A. George, W.G. Poole Jr., and R.G. Voigt. Analysis of dissection algorithms for vector computers. *J. Comp. and Maths. with Applics.*, 4:287–304, 1978.

- [22] J.A. George, W.G. Poole Jr., and R.G. Voigt. Incomplete nested dissection for solving  $n$  by  $n$  grid problems. *SIAM J. Numer. Anal.*, 15:662–673, 1978.
- [23] J.A. George and J. W-H. Liu. Algorithms for matrix partitioning and the numerical solution of finite element systems. *SIAM J. Numer. Anal.*, 15:297–327, 1978.
- [24] J.A. George and J. W-H. Liu. An automatic nested dissection algorithm for irregular finite element problems. *SIAM J. Numer. Anal.*, 15:1053–1069, 1978.
- [25] J.A. George and J. W-H. Liu. The design of a user interface for a sparse matrix package. *ACM Trans. on Math. Software*, 5:134–162, 1979.
- [26] J.A. George and J. W-H. Liu. An implementation of a pseudo-peripheral node finder. *ACM Trans. on Math. Software*, 5:286–295, 1979.
- [27] J.A. George and J. W-H. Liu. A minimal storage implementation of the minimum degree algorithm. *SIAM J. Numer. Anal.*, 17:282–299, 1980.
- [28] J.A. George and D.R. McIntyre. On the application of the minimum degree algorithm to finite element systems. *SIAM J. Numer. Anal.*, 15:90–111, 1978.
- [29] N.E. Gibbs. A hybrid profile reduction algorithm. *ACM Trans. on Math. Software*, 2:378–387, 1976.
- [30] N.E. Gibbs, W.G. Poole Jr, and P.K. Stockmeyer. An algorithm for reducing the bandwidth and profile of a sparse matrix. *SIAM J. Numer. Anal.*, 13:236–250, 1976.
- [31] B.M. Irons. A frontal solution program for finite element analysis. *Int. J. Num. Meth. Engng.*, 2:5–32, 1970.
- [32] A. Jennings. A compact storage scheme for the solution of symmetric linear simultaneous equations. *Computer J.*, 9:281–285, 1966.
- [33] I.P. King. An automatic reordering scheme for simultaneous equations derived from newtork systems. *Int. J. Numer. Meth. Engng.*, 2:523–533, 1970.

- [34] J.J. Lambiotte. *The solution of linear systems of equations on a vector computer*. PhD thesis, Dept. Appl. Math. and Comp. Sci., University of Virginia, 1975.
- [35] R. Levy. Resequencing of the structural stiffness matrix to improve computational efficiency. *Jet Prop. Lab. Quart. Tech. Review*, 1:61–70, 1971.
- [36] R.J. Lipton, D.J. Rose, and R.E. Tarjan. Generalized nested dissection. *SIAM J. Numer. Anal.*, 16:346–358, 1979.
- [37] R.J. Lipton and R.E. Tarjan. A separator theorem for planar graphs. *SIAM J. Appl. Math.*, 36:177–199, 1979.
- [38] J. W-H. Liu and A.H. Sherman. Comparative analysis of the Cuthill-McKee and reverse Cuthill-McKee ordering algorithms for sparse matrices. *SIAM J. Numer. Anal.*, 13:198–213, 1976.
- [39] H.M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Sci.*, 3:255–269, 1957.
- [40] R.S. Martin and J.H. Wilkinson. Symmetric decomposition of positive definite band matrices. In *Handbook for Automatic Computation, Volume II*. Springer Verlag, 1971.
- [41] R.J. Melosh and R.M. Bamford. Efficient solution of load deflection equations. *J. Struct. Div. ASCE, Paper No. 6510*, pages 661–676, 1969.
- [42] E. G-Y. Ng. Generalized width-2 nested dissection. unpublished, 1981.
- [43] S.V. Parter. The use of linear graphs in Gaussian elimination. *SIAM Review*, 3:364–369, 1961.
- [44] D.J. Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. In R. C. Read, editor, *Graph Theory and Computing*, pages 183–217. Academic Press, 1972.
- [45] D.J. Rose and G.F. Whitten. A recursive analysis of dissection strategies. In J. R. Bunch and D. J. Rose, editors, *Sparse Matrix Computations*, pages 59–84. Academic Press, 1976.
- [46] B.G. Ryder. The PFORT verifier. *Software Practise and Experience*, 4:359–377, 1974.

- [47] A.H. Sherman. *On the efficient solution of sparse systems of linear and nonlinear equations*. PhD thesis, Yale University, 1975.
- [48] D.R. Shier. Inverting sparse matrices by tree partitioning. *J. Res. Nat. Bur. of Standards*, 80b, 1976.
- [49] W.F. Smyth and W.M.L. Benzi. An algorithm for finding the diameter of a graph. *IFIP Congress 74*, pages 500–503, 1974.
- [50] G.W. Stewart. *Introduction to Matrix Computations*. Academic Press, New York, 1973.
- [51] G. Strang and G.J. Fix. *An Analysis of the Finite Element Method*. Prentice-Hall, Englewood Cliffs, N.J., 1973.
- [52] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969.
- [53] W.F. Tinney and J.W. Walker. Direct solution of sparse network equations by optimally ordered triangular factorization. *Proc. IEEE*, 55:1801–1809, 1967.
- [54] G. VonFuchs, J.R. Roy, and E. Schrem. Hypermatrix solution of large sets of symmetric positive-definite linear equations. *Comp. Meth. Appl. Mech. Engng.*, 1:197–216, 1972.
- [55] E.L. Wilson, K.J. Bathe, and W.P. Doherty. Direct solution of large systems of linear equations. *Computers & Structures*, 4:363–372, 1974.
- [56] M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM J. Alg. Disc. Meth.*, 2:77–79, 1981.
- [57] D.M. Young. *Iterative Solution of Large Linear Systems*. Academic Press, New York, 1971.
- [58] O.C. Zienkiewicz. *The Finite Element Method*. McGraw-Hill, London, 1977.

# Index

- A**
- adjacency linked list 118
  - adjacency list 46
  - adjacency structure 46, 351
  - adjacent 43
  - adjacent set 43
  - ancestor 199
  - arithmetic operations 15
  - asymmetric block factorization 190, 246
  - auxiliary storage 105, 265
- B**
- band 55, 56
  - band schemes 55
  - bandwidth 56
  - block diagonal 45
  - block diagonal matrix 289
  - bordered grid 309
- C**
- Chan 106
  - Cholesky's method 2, 13
  - Cholesky factor 189
  - chord 117
  - clique 44, 65
  - comparison of iterative and direct methods 11
  - comparison of strategies 37
  - compatible 198
  - components 45
  - computer charges 340
  - connected 45
  - connection table 48
- D**
- degree 44
  - descendant 199
  - diagonal storage scheme 57
  - diameter 70
  - distance 70
  - dynamic storage scheme 9
- E**
- eccentricity 70
  - edges 42
  - Eisenstat 106
  - element model 185
  - elements 187
  - elimination graphs 122
  - elimination sequence 122
  - $Env(\mathbf{A})$  59
  - envelope 55, 59, 238
  - envelope methods 55
  - envelope size 59
  - envelope structure 290
- F**
- factorization 5, 189
  - father 199, 243
  - finite element graph 187
  - finite element system 187
  - frontwidth 60
- G**
- Gibbs et al. 106
  - graph 42, 187
  - grid 304

- grid problem 268
- I
  - $i$ -th front 64
  - $i$ -th frontwidth 64
  - implicit solution 194
  - incomplete nested dissection 318, 324
  - indefinite 5
  - indefinite matrix 5
  - indefinite sparse matrix 5
  - integral approximation 15
  - interface 361
  - Irons 106
  - isoparametric inequality 318
  - iterative methods 10
- K
  - King 106
- L
  - labelling 42
  - leaf block 220
  - leaf node 219
  - length 71
  - level structure 208
  - Levy 106
  - link field 49
  - lower adjacency structure 52
  - lower triangular 3
- M
  - matrix product 247
  - Melosh 106
  - mesh 187, 304, 328
  - minimal storage band methods 105
  - monotone ordering 201
  - monotone profile 64
  - monotone profile property 88
- N
  - neighborhood set 294
  - nested dissection 303
  - nodes 42
  - nonzero counts 14
  - numbers of nonzeros 15
  - numerical experiments 327
  - numerical stability 5
- O
  - one-way dissection 267
  - operation counts 28
  - ordering 42
  - order of magnitude 14
  - outer product form 18
  - outer product version 21
  - overhead storage 33, 333
  - overlying 358
- P
  - partitioned matrices 189, 197
  - partitioning 71
  - path 45
  - peripheral 70
  - pivoting 5
  - primary storage 33
  - profile 55, 59
  - profile methods 55
  - propagation property 46
  - pseudo-peripheral 71
  - pseudo-peripheral node 72
- Q
  - quotient graph 197
  - quotient graphs 121, 122
  - quotient tree partitioning 208
- R
  - RCM 65
  - reachable sets 112
  - reducible 45
  - reordering 4
  - reverse Cuthill-McKee 65



reverse Cuthill-McKee algorithm 231  
root 199  
rooted level structure 71  
rooted tree 199

## S

section graph 44  
separator 45, 279, 303  
separators 268  
shortest path 70  
software modularity 9  
solution storage 333  
son 199  
span 224  
SPARSPAK 362  
starting node 70  
starting vector 11  
subgraph 44  
subtree 201  
supernodes 120  
symmetric block factorization 189  
symmetric permutation 4  
symmetric positive definite matrices 2

## T

test problems 2  
tree 199  
tree partitioning 202  
triangular factorization 3  
triangular solution 193  
triangular systems 26  
triangulated 117  
tridiagonal 56  
tridiagonal matrix 24

## U

unlabelled graphs 43  
user interface 361

## V

vertices 42

wave front 60  
width 71  
(XADJ, ADJNCY) 49