CWI Tracts

Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

The CWI is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

MATHEMATICAL CENTRE TRACTS 150

AN ANALYSIS OF SPARSE MATRIX STORAGE SCHEMES

M. VELDHORST

SECOND PRINTING

MATHEMATISCH CENTRUM AMSTERDAM 1982

1980 Mathematics subject classification: 52A45, 65F30, 68B99, 68C25, 68E99

1982 CR. Catgories: 4.22, 4.34, 5.14, 5.25, 5.3

ISBN 90 6196 242 0

First printing: 1982
Second printing: 1986

ACKNOWLEDGEMENTS

I am indebted to all who contributed to the completion of this book. I am particularly grateful to Prof. J. van Leeuwen, for this book would not have been completed without his stimulating support.
I also thank S.G. van der Meulen for the numerous discussions necessary for the design of TORRIX-SPARSE.
I owe much gratitude to A.A. Schoone and Prof. J. van Leeuwen for their willingness to read numerous versions of this book. Their close reading led to a great number of improvements.
Last but not least I thank J. Pannekoek for she took the terrible job of typing the manuscript off my hands.
I thank the Mathematical Centre for the opportunity to publish this monograph in their series Mathematical Centre Tracts and all those at the Mathematical Centre who have contributed to its technical realization.

CONTENTS

INTRODUCTION 1

CHAPTER I THE TORRIX LIBRARY SYSTEM 7

I.1 PRELIMINARIES FROM LINEAR ALGEBRA 7

I.1.1 Algebraic systems 7
I.1.2 Vector spaces 8
I.1.3 Linear mappings 10
I.1.4 Inner products 12

I.2 TORRIX DESIGN OBJECTIVES 14

I.2.1 Outline of the objectives 14
I.2.2 Total-arrays 15
I.2.3 Viability of the design objectives 16

I.3 TORRIX-BASIS 18

I.3.1 Implemented features 18
I.3.2 Choice of programming language 18
I.3.3 Data structures in TORRIX-BASIS 19

I.4 EXTENSIONS OF TORRIX-BASIS 22

I.4.1 Complexification of the field of scalars 22
I.4.2 Infinite fields of scalars 24
I.4.3 Triangular and band matrices 24
I.4.4 Other sparse matrices 25

CHAPTER II SPARSITY PATTERNS AND THE USE OF SPARSE MATRICES 27

II.1 SPARSE MATRICES RELATED TO DIFFERENTIAL EQUATIONS 28

II.1.1 Sparsity patterns obtained with finite difference approximations 29
II.1.2 Weighted residuals and stationary methods 32

II.2 SPARSITY PATTERNS IN OPTIMIZATION PROBLEMS 35

II.3 NUMERICAL ALGORITHMS AND SPARSE MATRICES 35

II.3.1 Operations on sparse matrices related to LU decomposition 36
II.3.2 Operations on sparse matrices in block factorization 38
II.3.3 Other direct methods to solve a linear system of equations 39
II.3.4 Operations on sparse matrices in iterative methods to solve a linear system of equations 40
II.3.5 Operations on sparse matrices related to the matrix eigenvalue problem 42
II.3.6 Operations on sparse matrices related to the linear least square problem 42
II.3.7 Operations on sparse matrices in the simplex method 44

APPENDIX SOME COMMON SPARSITY PATTERNS 47

CHAPTER III COMPLEXITY RESULTS FOR THE CONSECUTIVE ONES PROPERTY 49

III.1 PRELIMINARIES 50

III.1.1 Graph theory 50
III.1.2 Complexity of algorithms 51
III.1.2.1 NP-complete problems 53

III.2 CONSECUTIVE ONES PROPERTY AND SUBMATRICES 55

III.3 ALGORITHMIC ASPECTS OF THE EXISTENCE OF FORBIDDEN PERMUTED SUBMATRICES 57

III.3.1 The existence of a forbidden permuted submatrix 57
III.3.2 The largest forbidden permuted submatrix 59
III.3.3 Enumerating all forbidden permuted submatrices 67

III.4 MINIMIZATION PROBLEMS RELATED WITH COR 74

III.4.1 Augmentation 74
III.4.2 Storing a number of (permuted) submatrices 75

CHAPTER IV APPROXIMATION OF THE CONSECUTIVE ONES MATRIX AUGMENTATION PROBLEM 77

IV.1 INTRODUCTION AND PRELIMINARIES 78

IV.2 APPROXIMATION WITH ON-LINE COLUMN INSERTION ALGORITHMS 87

IV.3 EXTENSIONS TO OTHER CLASSES OF APPROXIMATION ALGORITHMS 95

IV.3.1 Preprocessing to block diagonal form 95
IV.3.2 Mixers and on-line column insertion algorithms 98

CHAPTER V DATA STRUCTURES FOR SPARSE MATRICES 101

V.1 DESIGN OBJECTIVES 102

V.1.1 Design objectives for the data structure for sparse matrices 103
V.1.2 Design objectives for a TORRIX-SPARSE software system 104

V.2 A REVIEW OF SPARSE MATRIX DATA STRUCTURES 106

V.2.1 Band and profile data structures 107
V.2.2 Data structures for arbitrary sparsity patterns 107
V.2.3 Data structures for block patterns 110

V.3 A NEW DATA STRUCTURE FOR SPARSE MATRICES 111

V.3.1 Sparse matrix storage trees 112
V.3.2 An acyclic directed graph storage scheme 115

V.4 SLICING IN TORRIX-SPARSE 116

V.4.1 Types of variability and operations in TORRIX-SPARSE 117
V.4.2 Slicing in ALGOL 68 and TORRIX-BASIS 118
V.4.3 Slicing operators in TORRIX-SPARSE 119

V.5 TORRIX-SPARSE: MODES AND ALGORITHMS 121

V.5.1 The SPARSEI system 124
V.5.1.1 Mode declarations in SPARSEI 124
V.5.1.2 Interface between SPARSEI and TORRIX-BASIS 127
V.5.1.3 SPARSEI operations and recursive partitions 128
V.5.2 The SPARSEII system 132
V.5.3 The SPARSEIII system 137
V.5.3.1 Mode declarations in SPARSEIII 137
V.5.3.2 Operations in SPARSEIII 139
V.5.3.3 The occurrence of side effects in SPARSEIII 143
V.5.4 The SPARSEIV system 145
V.5.4.1 Mode declarations in SPARSEIV 145
V.5.4.2 The uptodate test on element and submatrix slices 149
V.5.4.3 Sparse vectors 156
V.5.5 The SPARSEV system 159

APPENDIX TABLE OF OPERATORS 166

CHAPTER VI OPTIMAL PARTITIONS OF A SPARSE MATRIX IN TORRIX-SPARSE 179

VI.1 STORAGE TREES AND SETS OF CONCRETE DOMAINS 180

VI.2 FINDING A STORAGE TREE WITH A MINIMUM NUMBER OF LEAVES 184

VI.3 SPARSE MATRIX STORAGE TREES OF MINIMUM HEIGHT 195

CHAPTER VII SUITABILITY OF TORRIX-SPARSE FOR SYMBOLIC LU DECOMPOSITION 207

VII.1 APPLICABILITY OF TORRIX-SPARSE FOR PERMUTING A MATRIX TO ZERO-FREE MAIN DIAGONAL FORM 209

VII.2 APPLICABILITY OF TORRIX-SPARSE FOR PERMUTING A MATRIX TO SQUARE BLOCK TRIANGULAR FORM 217

VII.3 APPLICABILITY OF TORRIX-SPARSE FOR COMPUTING THE FILL IN LU DECOMPOSITION 220

REFERENCES 224

INDEX 231

SUMMARY 236

INTRODUCTION

In many fields of research numerical mathematical problems are solved with the aid of computers. Very often the matrices that arise must be stored entirely or partly in computer memory. There are many possible storage schemes, and it is often worthwhile to make a detailed study of which storage scheme can be used best for the matrices at hand. Sometimes, advantage can be taken of special properties of these matrices.

In this book we will study these aspects in detail for sparse matrices (i.e., matrices with many zero elements). The choice of a storage scheme for a sparse matrix depends heavily on the operations to be performed on the matrix. A bad choice can lead to a considerable increase in the computing time used. Often many algorithms are available for one particular operation, one algorithm for each storage scheme for sparse matrices in computer memory.

For the choice of a storage scheme and the necessary algorithms for sparse matrices, the following criteria can be used:

(i) Are the scheme and the algorithms suitable for all sparse matrices or are they only interesting in case of a special distribution of the non-zero elements over the matrix?

(ii) Are efficient algorithms available for all operations? If not, is it necessary to design new algorithms or to use several different storage schemes in the same program?

(iii) If the storage scheme is used, how large is the ratio of the required computer memory and the number of non-zero elements for the matrix under consideration?

If an analysis of a storage scheme and its algorithms does not provide satisfactory answers (e.g., the analysis does not give a decisive answer, a

proof is given that a task cannot be computed efficiently), then it may be worthwhile to permute the rows and columns of the matrix in order that satisfactory answers can be obtained with the analyses mentioned before. A related problem is then:

(iv) Design and analyze algorithms that permute an arbitrary matrix in computer memory such that satisfactory results can be obtained for the permuted matrix according to the criteria given in (ii) and (iii).

The scientific literature contains many publications already with analyses of storage methods and algorithms for sparse matrices. Mostly storage schemes and algorithms are designed such that the stated problem concerning sparse matrices can be solved efficiently in time (e.g., [35], [69]). These are analyses in the sense of (i), (ii) and possibly (iv). As for criterion (iii) combined with (iv) only a few storage schemes have been analyzed. For instance, in the case of band matrices such an analysis has been made (cf. [28]). In this book a number of additional storage schemes for sparse matrices are proposed and analyzed.

In 1976-77 S.G. van der Meulen and the author developed a software system TORRIX (cf. [75]) for matrix-vector computations over an arbitrary field of scalars in the programming language ALGOL 68. Considering the potential use of TORRIX, it was reasonable to design next to it a special system TORRIX-SPARSE geared to the manipulation of sparse matrices and vectors, based on the ideas of [75]. For this purpose the author made a detailed study of the operations and data structures which a sparse matrix package should provide (chapter II). Moreover, he developed and analyzed in detail the following (new) storage scheme for sparse matrices:

(i) The *rowmat* storage scheme: store the rows of the matrix separately such that all zero elements at both ends of each row will not be stored. This storage scheme is well suited for full triangular matrices. If the columns may be permuted, the optimality (in the sense of (iii)) of this storage scheme for other sparse matrices is closely related to the consecutive ones property for rows of a matrix (cf. [27]). As far as we know the relation of the consecutive ones property to a sparse matrix storage scheme has never before been

explored. The consecutive ones property itself has been studied thoroughly in the past (cf. [39]) and some results are important with regard to the optimality of the *rowmat* data structure. In chapter III we will review these results. Booth and Lueker (cf. [11]) presented a linear time algorithm to determine whether a matrix A has the consecutive ones property for rows, i.e., whether the columns of A can be permuted in such a way that A can be stored in a *rowmat* data structure without storing any zero element. We will prove the NP-completeness of the problem of finding of an arbitrary sparse matrix A a largest permuted submatrix A' of A that does not have the consecutive ones property for rows while every proper submatrix of A' has. Moreover, we will present an algorithm that enumerates all permuted submatrices A' of A that do not have the consecutive ones property for rows while every proper submatrix of A' has in polynomial time per submatrix.
If we want to store in a *rowmat* data structure a matrix A that does not have the consecutive ones property for rows, then zero elements will be stored, even if we do not store A itself but a column-permutation of it. In [10] the NP-completeness has been proven of the problem to find an optimum column-permutation of A, i.e., a column-permutation that can be stored in a *rowmat* data structure with at most as many stored elements as any other column-permutation of A. Knowing this, it is reasonable to try and design an algorithm that finds a near optimum column-permutation of A.
In chapter IV we will prove that near optimum column-permutations for arbitrary sparse matrices cannot be found by such simple algorithmic schemes like the on-line column insertion algorithms. In fact we will characterize a large class of reasonable algorithms that will never do to find near optimum column-permutations. We will see that there are matrices for which these algorithms find column-permutations that are even far from optimal. This provides an indication that the *rowmat* data structure is not well suited for arbitrary sparse matrices if full storage optimality is the ultimate goal.

(ii) The tree storage scheme: partition the matrix recursively into blocks and subblocks and do not store the blocks obtained by this partition which have no non-zero element at all. Sparseness of a non-zero block can be obtained in three ways:

(1) a block is represented by one (small) full matrix containing fewer elements than the block itself,
(2) if the non-zero elements of the block are situated along a (small) number of diagonals, then only these diagonals need be stored,
(3) a block can be partitioned into smaller blocks.

A further reduction in storage can be obtained in case of equal blocks. Unlike some other sparse matrix storage schemes the tree storage scheme is **not** suited for sparse matrices with a random distribution of the non-zero elements over the matrix. In most practical problems involving sparse matrices, however, there is no random distribution of the non-zero elements over the matrix (cf. [69]). The tree storage scheme will be proposed and described in chapter V. This chapter contains also a proposal for a TORRIX-SPARSE system in which this storage scheme can be implemented. A slicing mechanism as in ALGOL 68 is proposed. With such a slicing mechanism operations induce specific side effects if applied to a matrix or one of its slices. It simplifies the programming task of the user considerably. We will describe in chapter V how it can be implemented in the tree storage scheme in such a way that it leads, if used, to a decrease in computation time in many applications, at the cost of only a small increase in storage requirements.

Though in many practical problems it is clear what the recursive partition of the matrix at hand should be, the problem to find for an arbitrary sparse matrix a good recursive partition is hard. In chapter VI the question is studied in detail. The matrix is assumed to be given as a set of (small) full matrices with mutually disjoint index domains, i.e., the sparsity pattern of A is viewed as a set of mutually disjoint rectangles. Only elements of these full matrices are allowed to be stored. We will consider in chapter VI two optimality criteria for recursive partitions: the binary tree determined by the recursive partition should be of minimum height or have a minimum number of vertices, respectively. We will present and analyze algorithms that find recursive partitions of an arbitrary sparse matrix A that are optimal with regard to the latter and former criterion, respectively. It will be shown that there are matrices A with a sparsity pattern such that no recursive

partition of A induces a binary tree with a minimum number of vertices and of minimum height simultaneously.
In this chapter only a few problems concerning optimal partitions are solved. More research is needed to solve the problems mentioned in chapter V.
In chapter VII we will test the suitability of the TORRIX-SPARSE system in case it is used for the problem of solving a linear system of equations. This problem can be divided into a number of subproblems (cf. [63]) and TORRIX-SPARSE will be tested on three of them: permuting a sparse matrix to zero-free diagonal form, permuting a sparse matrix to block triangular form and performing a symbolic LU decomposition. We will try to formulate and design algorithms for these three problems that take advantage of the storage schemes proposed in chapter V. This means that these algorithms should be block oriented and take advantage of the fact that scalar values are stored in (small) full matrices or in arrays of diagonals. Especially for the problem of performing a symbolic LU decomposition without row- and column-permutations an efficient algorithm will be presented.

An introductory chapter is added for a brief explanation of the basic ideas and objectives of TORRIX (chapter I). In this chapter and chapter V the reader is assumed to be familiar with the programming language ALGOL 68, especially its data structure facilities, its slicing mechanism and the identity declaration. Knowledge about transput is not assumed. For an introduction to ALGOL 68 we refer to [52] and [59]. The other chapters can be read without any knowledge of ALGOL 68.

CHAPTER I

THE TORRIX LIBRARY SYSTEM

TORRIX is a computational system for finite sequences based on the mathematical theory of linear algebra. In this chapter we will review the necessary linear algebra (I.1), state the design objectives of TORRIX (I.2) and give some remarks on the implementation TORRIX-BASIS (I.3). In I.4 several possible extensions to TORRIX-BASIS are mentioned.

I.1 PRELIMINARIES FROM LINEAR ALGEBRA.

This section contains a survey of the main definitions and theorems of linear algebra underlying the TORRIX system. We will concentrate on fields, vector spaces, linear mappings and inner products, in this order. Whereas the concepts are likely to be well-known to mathematical readers, the specific notations and terminology will lead up to the later representations in the TORRIX system (see I.2). For a more detailed treatise on linear algebra, we refer to [20] and [41].

I.1.1 Algebraic systems.

An algebraic system is a set S with a number of (total) operations defined on the elements of S. There may be nullary operations $\emptyset=S^0\rightarrow S$, unary operations $S\rightarrow S$, binary operations $S\times S\rightarrow S$, etc. In TORRIX only nullary, unary and binary operations are used.

DEFINITION 1.1. A group $(G,\square,n,')$ is a set G with a binary operation $\square:G\times G\rightarrow G$, a neutral element $n\in G$ and a unary operation $'$ (inverse) satisfying the following three conditions:

(i) $\square$ is associative: $(a\square b)\square c = a\square(b\square c)$ for all $a,b,c\in G$ (1.1)

(ii) $a\square n = n\square a = a$ for all $a\in G$ (1.2)

(iii) $'a\square a = a\square 'a = n$ for all $a\in G$.

A group $(G,+,0,-)$ is called additive and, by definition, is commutative (i.e., $a+b = b+a$ for all $a,b\in G$). A group $(G,\times,1,^{-1})$ is called multiplicative and may or may not be commutative.
In an additive group $\sum_{i=p}^{q} a_i$ means $a_p+a_{p+1}+\ldots+a_q$ if $p\leq q$ and 0 otherwise and ra $(r\in\mathbb{N})$ means $\sum_{i=1}^{r} a$. In a multiplicative group $\prod_{i=p}^{q} a_i$ means $a_p\times a_{p+1}\times\ldots\times a_q$ if $p\leq q$ and 1 otherwise and a^r $(r\in\mathbb{N})$ means $\prod_{i=1}^{r} a$. In an additive group $a-b$ means $a+(-b)$ and in a multiplicative group $\frac{a}{b}$ means $a\times(b^{-1})$. We do not need to distinguish between left and right inverses of a group, because their equality and uniqueness are direct consequences of the definition of a group as given.

DEFINITION 1.2. A ring R is a system $(R,+,0,-,\times,1)$ in which $(R,+,0,-)$ is an additive group and $(R,\times,1)$ satisfies (1.1) and (1.2). Moreover, multiplication in R is to be distributive over addition:

$$a\times(b+c) = (a\times b)+(a\times c) \text{ and } (a+b)\times c = (a\times c)+(b\times c) \quad \text{for all } a,b,c\in R.$$

It is easy to see that $a\times 0 = 0\times a = 0$ for all $a\in R$. If its multiplication is commutative (i.e., $a\times b = b\times a$ for all $a,b\in R$), then a ring is called commutative.

DEFINITION 1.3. A field $(F,+,0,-,\times,1,^{-1})$ is a ring in which $(F\setminus\{0\},\times,1,^{-1})$ is a multiplicative group.

A field F is commutative if $F\setminus\{0\}$ is commutative as a group. Non-commutative fields are called skew. We shall always assume fields to be commutative.

Examples:

(i) The system $\mathbb{N}$ of natural numbers (including 0) is not a group. Addition and multiplication in $\mathbb{N}$ both do satisfy (1.1) and (1.2).

(ii) The system $\mathbb{Z}$ of integral numbers is a ring.

(iii) The system $\mathbb{Z}_n$ of integral numbers modulo n is a ring. If n is a prime number, $\mathbb{Z}_n$ is a field.

(iv) The systems $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{C}$ of rational, real and complex numbers, respectively, are all fields.

I.1.2 Vector spaces.

Next we consider the common algebraic system of a vector space.

DEFINITION 1.4. A vector space V over a field F is an additive group with a mapping of F×V into V (to represent scalar multiplication) satisfying (1.3).

$$\begin{aligned} \alpha(u+v) &= \alpha u+\alpha v \\ (\alpha+\beta)u &= \alpha u+\beta u \qquad\qquad \alpha,\beta\in F,\ u,v\in V \\ (\alpha\times\beta)u &= \alpha(\beta u) \\ 1u &= u \end{aligned} \tag{1.3}$$

The elements of F are called scalars and we shall denote them by Greek letters $\alpha,\beta,\gamma,\ldots$. The elements of V are called vectors and will be denoted by the letters $u,v,w,\ldots$. Given a non-empty finite sequence*) $(u_i)=(u_1,u_2,\ldots,u_n)$ of vectors, we can form linear combinations:

DEFINITION 1.5. Let V be a vector space over F and $(u_i)=(u_1,\ldots,u_n)$ a sequence of vectors of V.

(i) A vector v of V is a linear combination of (u_i) if and only if $v = \sum_{i=1}^{n}\alpha_i u_i$ for some sequence of scalars (α_i) over F.

(ii) The vectors (u_i) are linearly independent, if there is exactly one linear combination (i.e., one sequence (α_i) of scalars) yielding the zero vector.

Because the zero vector is a linear combination of each vector sequence (u_i), linear independence of (u_i) implies

$$\begin{aligned} &\sum_{i=1}^{n}\alpha_i u_i = 0 \ \Rightarrow\ \alpha_i=0 \quad (1\leq i\leq n) \\ \text{and}\quad &\sum_{i=1}^{n}\alpha_i u_i = \sum_{i=1}^{n}\beta_i u_i \ \Rightarrow\ \alpha_i=\beta_i \quad (1\leq i\leq n) \end{aligned} \tag{1.4}$$

Even for infinite sets of vectors one can define a suitable notion of linear independence: a set S of vectors is linearly independent if and only if each finite sequence of different vectors of S is linearly independent.

---

*) We explicitly did not use the notion of a set here as we wish to allow for the possibility that there are equal elements among the u_1 to u_n.

DEFINITION 1.6. A set of vectors B in a vector space V is a basis of V if and only if B is linearly independent and each vector $v \in V$ is a linear combination of a finite sequence of B. V is finite dimensional if it has a finite basis.

In a finite dimensional vector space V all bases have the same number of elements, the dimension of V or dim(V) (cf. [20]). The following result is fundamental (cf. [20]):

THEOREM 1.1. *Every finite dimensional vector space* V *over the field* F *with* dim(V) = n *is isomorphic to* F^n.

It follows that elements of V can be represented as singly subscripted sequences of scalars relative to a given basis (u_i). The sequence of n scalars $(0,\ldots,0,1,0,\ldots,0)$ with a "1" in the i^{th} coordinate position represents u_i and

$$
\begin{aligned}
&(\ \alpha_1\ ,\ \alpha_2\ ,\text{-----},\ \alpha_n\ ) \text{ represents } \sum_{i=1}^{n}\alpha_i u_i,\\
&(\alpha_1+\beta_1,\alpha_2+\beta_2,\text{-----},\alpha_n+\beta_n) \text{ represents } \sum_{i=1}^{n}\alpha_i u_i + \sum_{i=1}^{n}\beta_i u_i,\\
&(\alpha_1-\beta_1,\alpha_2-\beta_2,\text{-----},\alpha_n-\beta_n) \text{ represents } \sum_{i=1}^{n}\alpha_i u_i - \sum_{i=1}^{n}\beta_i u_i,\\
&(\ \gamma\times\alpha_1,\ \gamma\times\alpha_2,\text{-----},\ \gamma\times\alpha_n) \text{ represents } \gamma\sum_{i=1}^{n}\alpha_i u_i.
\end{aligned}
$$

A subset $V' \subseteq V$ may be a vector space on its own with the same addition and scalar multiplication as V. Such a V' is called a subspace of V. If V is finite dimensional, then so is V' and $\dim(V') \leq \dim(V)$. If $V' \neq V$, then $\dim(V') < \dim(V)$. For a set of vectors $S \subseteq V$ we define the subspace $V' \subseteq V$ spanned by the vectors of S, as the smallest subspace satisfying the following conditions:

$$u \in S \Rightarrow u \in V'$$

$$u_1 \in V',\ u_2 \in V' \Rightarrow \alpha u_1 + \beta u_2 \in V' \quad \text{for all } \alpha,\beta \in F.$$

If S is linearly independent, then S is a basis of the subspace it spans.

I.1.3 Linear mappings.

This section deals with the general concept of a linear mapping. In this section V, W and X are vector spaces over the same field F.

DEFINITION 1.7. A mapping $L:V\to W$ is linear if an only if

$$L(\alpha u+\beta v) = \alpha(Lu)+\beta(Lv) \quad \text{for all } u,v\in V,\ \alpha,\beta\in F.$$

There is a zero mapping O which maps each $u\in V$ onto $0\in W$. The sum of two linear mappings is defined as $(L+M)u = Lu+Mu$ and for all $\alpha\in F$ $(\alpha L)u = \alpha(Lu)$. With these definitions the set of linear mappings $L:V\to W$ is a vector space over F, denoted as $\mathrm{Hom}(V,W)$.

If V=W, one can say more. First there is a linear identity mapping I with $Iv=v$ for all $v\in V$, and second, one can define the product of two linear mappings $L,M:V\to V$ as functional composition: $(L\times M)u = L(Mu)$. With these definitions $\mathrm{Hom}(V,V)$ is a ring. The ring is not necessarily commutative, because linear mappings usually do not commute. Linear mappings $L:V\to V$ are often called linear transformations or operators, but as we will need the word 'operator' in another context, we will keep ourselves to mappings in the context of linear algebra.

For $\mathrm{Hom}(V,W)$ a result exists similar to theorem 1.1 (cf. [20]):

THEOREM 1.2. *Let* V *and* W *be finite dimensional vector spaces over the field* F *with* $\dim(V)=n$ *and* $\dim(W)=m$. *Then* $\mathrm{Hom}(V,W)$ *is isomorphic to* F^{mn} *and hence,* $\dim(\mathrm{Hom}(V,W))=mn$.

Thus, given a basis of $\mathrm{Hom}(V,W)$, a linear mapping $A\in\mathrm{Hom}(V,W)$ can be represented as a sequence of mn scalars. For reasons of clarity such a sequence is usually arranged as an $m\times n$ matrix with m rows and n columns:

$$(\alpha_{ij})_{1\le i\le m,1\le j\le n} = \begin{pmatrix} \alpha_{11} & \cdots & \alpha_{1n} \\ \vdots & & \vdots \\ \alpha_{m1} & \cdots & \alpha_{mn} \end{pmatrix}$$

Let $(v_1,\ldots,v_n)$ and $(w_1,\ldots,w_m)$ be bases of V and W, respectively. Related to these bases we obtain the standard basis $(E_{ij})_{1\le i\le m,1\le j\le n}$ of $\mathrm{Hom}(V,W)$ with

$$E_{ij}v_r = \begin{cases} 0 & \text{if } j\neq r \\ w_i & \text{otherwise} \end{cases} \qquad (1\le i\le m,1\le j\le n) \tag{1.5}$$

E_{ij} $(1\le i\le m,1\le j\le n)$ corresponds to the $m\times n$ matrix with 1 in the $(i,j)^{th}$ position and 0 in all other positions. If mappings $A,B\in\mathrm{Hom}(V,W)$ are represented

as $(\alpha_{ij})_{1\leq i\leq m,1\leq j\leq n}$ and $(\beta_{ij})_{1\leq i\leq m,1\leq i\leq n}$ relative to (1.5), the following statements can be easily proved:

(i) $A(\sum_{j=1}^{n} \upsilon_j v_j) = \sum_{j=1}^{n} \sum_{i=1}^{m} \upsilon_j \alpha_{ij} w_i = \sum_{i=1}^{m} (\sum_{j=1}^{n} \upsilon_j \times \alpha_{ij}) w_i$,

(ii) relative to (1.5), A+B can be represented by $(\alpha_{ij}+\beta_{ij})_{1\leq i\leq m,1\leq j\leq n}$.

Let $\dim(X)=p$, $(x_1,\ldots,x_p)$ be a basis of X and $C\in \mathrm{Hom}(W,X)$ be represented by $(\gamma_{ij})_{1\leq i\leq p,1\leq j\leq m}$ relative to the standard basis of $\mathrm{Hom}(W,X)$. Then $C\circ A \in \mathrm{Hom}(V,X)$ can be represented by a sequence $(\xi_{ij})_{1\leq i\leq p,1\leq j\leq n}$ with

$$\xi_{ij} = \sum_{k=1}^{m} \gamma_{ik} \times \alpha_{kj} \quad (1\leq i\leq p, 1\leq j\leq n)$$

Remark 1.1. A linear mapping $A:V\to V$ is usually represented relative to one basis for V:

$$Av_j = \sum_{i=1}^{n} \alpha_{ij} v_i \quad (1\leq j\leq n)$$

Another basis may give rise to a different representation for the same mapping.

Remark 1.2. In this section sequence elements were numbered from index 1. Sometimes it is appropriate to start with another index. For example:

Let $A\in\mathrm{Hom}(V,V)$ be represented by $(\alpha_{ij})_{1\leq i,j\leq n}$ relative to basis $(v_i)_{1\leq i\leq n}$. If $V'\subseteq V$ is a subspace of V, we define the restriction $A|V'$ of A as the linear mapping $B:V'\to V$ with $Bv'=Av'$ for all $v'\in V'$ (cf. [20]). If V' is the subspace spanned by the vectors $(v_k,\ldots,v_m)$, $A|V'$ can be represented by $(\alpha_{ij})_{1\leq i\leq n,k\leq j\leq m}$ relative to $(v_k,\ldots,v_m)$.

I.1.4 Inner products.

Many rings R (for instance $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$) can be ordered according to the following rules:

for all $\alpha,\beta,\gamma\in R$

(i) either $\alpha\leq\beta$ or $\beta\leq\alpha$ and (hence) $\alpha\leq\alpha$

(ii) $\alpha\leq\beta$ and $\beta\leq\alpha$ imply $\alpha=\beta$

(iii) $\alpha\leq\beta$ and $\beta\leq\gamma$ imply $\alpha\leq\gamma$

(iv) $\alpha\leq\beta$ implies $\alpha+\gamma \leq \beta+\gamma$

(v) $\alpha\leq\beta$ and $0\leq\gamma$ imply $\gamma\times\alpha \leq \gamma\times\beta$ and (hence) $\alpha\times\gamma \leq \beta\times\gamma$

Given an ordering of a field F, the concept of an inner product in vector spaces over F becomes significant:

DEFINITION 1.8. Let V be a vector space over an ordered field F. An inner (or scalar) product on V is a function $\langle,\rangle: V\times V\to F$ with the following properties:

$$\left.\begin{array}{ll} \text{(i)} & \langle u,v\rangle = \langle v,u\rangle \\ \text{(ii)} & \langle \alpha u+\beta v,w\rangle = \alpha\times\langle u,w\rangle + \beta\times\langle v,w\rangle \\ \text{(iii)} & 0 \leqq \langle u,u\rangle \qquad\qquad \alpha,\beta\in F,\ u,v,w\in V \\ \text{(iv)} & \langle u,u\rangle = 0 \text{ implies } u=0 \end{array}\right\} \quad (1.6)$$

Any vector space V with an inner product is called an inner product space. From the viewpoint of computations with finite sequences, inner products are very important. Of course there may be many inner products defined on a vector space V. For instance, if $V=\mathbb{R}^n$: let $\omega_1,\ldots,\omega_n\in\mathbb{R}$ and each $\omega_i>0$; then for each choice of (ω_i) we have another inner product:

$$\langle u,v\rangle = \sum_{i=1}^{n}\omega_i u_i v_i \text{ with } u=(u_1,\ldots,u_n),\ v=(v_1,\ldots,v_n)$$

The inner product with $\omega_i=1$ $(1\leqq i\leqq n)$ is most well-known. In I.1.2 a basis of a vector space V was defined as a linearly independent set of vectors that spanned all of V. In an inner product space it is easy to find the linear combination of a basis yielding a given vector v, provided the right basis is chosen. The following result (cf. [20]) is fundamental:

Theorem 1.3. *Let* $(V, \langle,\rangle)$ *be an* n*-dimensional inner product space over* F. *Then there is a basis* $(u_1,\ldots,u_n)$ *of* V *such that:*

(i) $(u_1,\ldots,u_n)$ *is an orthogonal set;* i.e. $\langle u_i,u_j\rangle \neq 0$ *if* $i=j$; $= 0$ *if* $i\neq j$

(ii) if $v\in V$, *then* $v = \sum_{i=1}^{n}\dfrac{\langle v,u_i\rangle}{\langle u_i,u_i\rangle}u_i$

(iii) if $v,w\in V$, *then* $\langle v,w\rangle = \sum_{i=1}^{n}\dfrac{\langle v,u_i\rangle\times\langle u_i,w\rangle}{\langle u_i,u_i\rangle^2}$

If we represent vectors v and w by finite sequences (v_i) and (w_i) of scalars relative to a basis implied by theorem 1.3, then

$$v_i = \frac{\langle v,u_i\rangle}{\langle u_i,u_i\rangle},\quad w_i = \frac{\langle w,u_i\rangle}{\langle u_i,u_i\rangle} \text{ and } \langle v,w\rangle = \sum_{i=1}^{n} v_i\times w_i.$$

Remark 1.3. There exists a proof of theorem 1.3 which does not use the ordering of F and thus the requirement $0\leqq\langle u,u\rangle$ in (1.6) (cf. [20]). We have introduced inner products with $0\leqq\langle u,u\rangle$, because this condition is

assumed in most applications of an inner product space over the real number system $\mathbb{R}$.

Remark 1.4. A result similar to theorem 1.3 exists for finite dimensional vector spaces over $\mathbb{C}$ on which a so-called hermitian form has been defined (cf. [41]).

Conclusion. In section I.1 we have put much emphasis on the concept of a basis. For computational purposes it is important that vectors and linear mappings can be represented by finite sequences of scalars (see I.1.2 and I.1.3). The particular choice of a basis can have a strong effect on how to compute with these finite sequences (see I.1.3 and I.1.4).

I.2 TORRIX DESIGN OBJECTIVES.

I.2.1 Outline of the objectives.

TORRIX has been designed as a computational system for finite dimensional vector spaces over a fixed but arbitrary field F. This means that, when someone (a "user") wants to apply TORRIX, he has to specify the field F (or more precisely: a computational system S for the field F). Such a general design allows TORRIX to be used in many areas of computation: statistics, curve fitting, differential equations, arithmetic with polynomials over $\mathbb{Q}$, and many others.
TORRIX does not contain the system S of any specific field; it is merely based on assumptions of how a system S is presented to it. When supplied with an S, TORRIX yields a computational system T for vector spaces over F. Two different systems S and S' for the same field F will yield two different systems T and T'. In the sequel we will speak about a TORRIX system T based on a system S for F. The elements of S will be called *scal*s. TORRIX assumes that the integers are element of F and moreover that $0,1 \in \mathbb{Z}$ coincide with $0,1 \in F$.
As we stated above, the user has to specify S in order to work with T. We now arrive at the first objective:

Objective 1.1. Define TORRIX in such a way that the user can write a program that is valid for different *scal*-systems S (possibly for different fields F).

Probably this objective cannot be fully met: the transput of a given programming language may require special extra specifications of S. This, however, is not a mathematical problem of S, but merely a representation problem.

Objective 1.2. Design TORRIX in such a way that it is possible to use vectors and linear mappings as autonomous objects, without the user having to worry about how they are stored or represented in memory.

Only for reasons of efficiency it may be appropriate to know how a vector or linear mapping is stored in memory.

Objective 1.3. Design TORRIX in such a way that it is independent of the dimensions of a vector space and its subspaces.

I.2.2 Total-arrays.

Let a user-specified S be given. In the design of TORRIX, the following two technical concepts were introduced.

DEFINITION 1.9.

(i) An array1 is a mathematical function, mapping a set of consecutive integers [m:n] into S; array1's are denoted by $[\nu_i],[\phi_i],\ldots$. [m:n] is called the domain of an array1 and is sometimes denoted by $[\![\upsilon_i]\!],[\![\phi_i]\!]$, ...

(ii) An array2 is a mathematical function, mapping the cartesian product of two sets of consecutive integers [m1:n1]×[m2:n2] into S; array2's are denoted by $[\alpha_{ij}],[\beta_{ij}],\ldots$. [m1:n1]×[m2:n2] is called the domain of an array2 and is sometimes denoted by $[\![\alpha_{ij}]\!],[\![\beta_{ij}]\!],\ldots$. $[\![\alpha_{ij}]\!]_1$=[m1:n1] and $[\![\alpha_{ij}]\!]_2$=[m2:n2].

Array1's can represent vectors (relative to some basis) and array2's can represent linear mappings (relative to one basis, or possibly two bases). An array is either of type array1 or of type array2.

DEFINITION 1.10. Two arrays x and y are equivalent if and only if

(i) x and y are of the same type.

(ii) x and y are equal on the intersection of their domains.

(iii) x and y have zero values outside the intersection of their domains.

The definition of equivalence can be formally stated as follows:

$$\left.\begin{array}{ll} \text{for array1's } [\upsilon_i] \text{ and } [\phi_i]: & \text{for array2's } [\alpha_{ij}] \text{ and } [\beta_{ij}]: \\ [\upsilon_i]\cong[\phi_i] \text{ if and only if} & [\alpha_{ij}]\cong[\beta_{ij}] \text{ if and only if} \\ \text{(ii) } \upsilon_i=\phi_i \text{ for all } i\in[\![\upsilon_i]\!]\cap[\![\phi_i]\!] & \text{(ii) } \alpha_{ij}=\beta_{ij} \text{ for all } (i,j)\in[\![\alpha_{ij}]\!]\cap[\![\beta_{ij}]\!] \\ \text{(iii) } \upsilon_i=0 \text{ for all } i\in[\![\upsilon_i]\!]\setminus[\![\phi_i]\!] & \text{(iii) } \alpha_{ij}=0 \text{ for all } (i,j)\in[\![\alpha_{ij}]\!]\setminus[\![\beta_{ij}]\!] \\ \quad \phi_i=0 \text{ for all } i\in[\![\phi_i]\!]\setminus[\![\upsilon_i]\!] & \quad \beta_{ij}=0 \text{ for all } (i,j)\in[\![\beta_{ij}]\!]\setminus[\![\alpha_{ij}]\!] \end{array}\right\} \quad (1.7)$$

It follows that an array1 $[\upsilon_m,\upsilon_{m+1},\ldots,\upsilon_n]$ is equivalent to the array1

$$\begin{array}{ccccccccccccc} [0,0\text{---},0, & \upsilon_m, & \upsilon_{m+1}, & \text{---}, & \upsilon_n, & 0, & 0, & \text{---}, & 0] \\ \downarrow & \downarrow & \downarrow & & \downarrow & \downarrow & \downarrow & & \downarrow \\ -t & m-1 & m & & n & n+1 & & & t \end{array}$$

for any positive integer t that is large enough. Each array1 $[\upsilon_i]$ with $[\![\upsilon_i]\!]\subseteq[-t:+t]$ is equivalent to an element of S^{2t+1}. The value of t should be as large as possible provided that all TORRIX operations could be implemented. Thus, it depends on the programming language, in which TORRIX will be implemented, and the implementation of this language on an actual machine. In practice a program will never use domains outside [-t:+t], and it is allowed to identify the space of equivalence classes of array1's as being isomorphic to S^{2t+1}. In the same way array2's are equivalent to array2's with domain $[-t:+t]\times[-t:+t]$.

DEFINITION 1.11. An element of S^{2t+1} is a total-array1. A total-array2 is an array2 that is an element of $S^{(2t+1)\times(2t+1)}$.

Summarizing, we have:

(i) a total-array1 is a mapping $[-t:+t]\rightarrow S$.

(ii) a total array2 is a mapping $[-t:+t]\times[-t:+t]\rightarrow S$.

(iii) each array (even an array with empty domain) is equivalent to a total-array.

(iv) TORRIX is a system implementing total-arrays.

I.2.3 Viability of the design objectives.

Objective 1.1 can be fulfilled when it is possible to use an abstract data

type on which operators*) like +,-,×,/,↑,etc. are available, or can be made available, in the programming language. In a programming language like ALGOL 68 this can certainly be realized if F is the field $\mathbb{R}$, $\mathbb{Q}$ or $\mathbb{Z}_n$. However, some problems may arise if the field F itself is a vector space over another field G. In such a case the user may want to work with two TORRIX-systems. This puts some demands on the programming language in which TORRIX is implemented.

As for the second objective, it is not enough to have mode(type) declarations for vectors and linear mappings. The way in which the operators are defined, plays a role too. In fact, objective 1.2 cannot be met in full generality by any TORRIX system. We show this by means of an example. Consider a programming problem using two vector spaces V_1 and V_2 over the field $\mathbb{R}$ (e.g. take $V_1 = \mathbb{R}^n$ and V_2 is the vector space of polynomials of degree$\leq$n on the segment $[0,1] \subseteq \mathbb{R}$. The elements of both V_1 and V_2 may be represented by array1's with domain [0:n-1]. TORRIX does not distinguish between any typing for array1's and thus does not exclude that the "inner product" of two total-array1's is computed, one of them representing a vector of $\mathbb{R}^n$, the other representing a polynomial. The result is meaningless but it cannot be detected by the system.
The situation may become worse in case there are two bases of one vector space in use in an application. This allows two different array1 representations $[\upsilon_i]$ and $[\phi_i]$ of one and the same vector v. Computing the inner product <v,v> by means of $\Sigma \upsilon_i \phi_i$ will give an erroneous result.

Objective 1.3 can be met by means of the concept of a total-array as defined in I.2.2. All vector spaces and their subspaces are considered to be subspaces of S^{2t+1}.

Conclusion. If a programmer is to compute with vectors of different vector spaces, he must be careful with the use of general operators. In case of one vector space, TORRIX assumes, that vectors are represented by array1's relative to one fixed basis. As for subspaces V' of a vector space V,

---

*) We will always use the word 'operator' in the context of a programming language. In this context it has much in common with procedures and/or functions (cf. [52], [77]).

TORRIX assumes that vectors v'∈V' are represented relative to the basis of V' which is a subsequence of the basis used for V.

I.3 TORRIX-BASIS.*)

Each storage scheme for total-arrays yields another implementation of TORRIX. TORRIX-BASIS is the system in which each total-array is stored in one concrete array. For other implementations we refer to I.4 and chapter V. TORRIX-BASIS has been implemented as an ALGOL 68 standard prelude (cf. [75]). This section highlights some important aspects of this implementation.

I.3.1 Implemented features.

Obviously TORRIX-BASIS contains features for declaration, addition, subtraction and multiplication of total-array1's and total-array2's. Moreover, it provides the user with "selectors" on total-arrays, like taking submatrices, rows, columns, diagonals, subvectors and elements. In some operations total-array1's are considered as polynomials. Other operations can be used to control the storage used by a total-array. These latter operations may not be present in other implementations. Finally, there are operations that deal with sequences of *scal*s, but their number is kept as small as possible. Operations for which the defining algorithm depends heavily on the choice of the *scal*-system S, are not implemented. Thus, TORRIX-BASIS does not contain transput facilities, operations to solve a linear system of equations, etc. Nevertheless, TORRIX-BASIS contains operations that can be used in order to simplify such tasks considerably (for instance: modes and operations to record permutations).

I.3.2 Choice of programming language.

TORRIX-BASIS was implemented in the programming language ALGOL 68, as it provides easy mechanisms that facilitate the creation of general routines and has gained considerable acceptance among programmers of numerical software. More specifically, ALGOL 68 was chosen for the following reasons:

---

*) Much of the terminology in this section derives from the programming language ALGOL 68 (cf. [77], [52]).

(i) It is better and easier to describe and implement a library system within an existing language, than it is to define and implement such a system as a new programming language of its own.

(ii) ALGOL 68 provides mode declarations. Thus, it is possible to write programs using an abstract data type *scal*.

(iii) It is possible to define generic procedures (called operators in ALGOL 68). For instance, the operator x can be defined for all data structures of total-arrays regardless of the underlying *scal*-system S.

(iv) The modes of ALGOL 68 multiple values are independent of the specific bounds of each subscript. Moreover, multiple values can easily be "sliced" and a programmer can easily work with vectors of (sub)spaces of different dimensions and with submatrices (representation of linear mappings restricted to subspaces).

(v) The way in which the concepts of variables and pointers are united in ALGOL 68, could be used to distinguish several kinds of variability of total-arrays.

I.3.3 Data structures in TORRIX-BASIS.

Total-arrays are implemented in the following manner in the TORRIX-BASIS system. The value t (recall that the domain of each array1 is a subset of [-t:+t]) is too big for all practical purposes to allow an ALGOL 68 multiple value with bounds [-t:+t]. Very likely t will have the maximum value that allows all kinds of ALGOL 68 slicing features on multiple values with bounds like [t:t],[-t:-t],[t:-t],etc. Now we will use the equivalence relation (1.7): in TORRIX-BASIS a total-array1 is implemented by one multiple value with one subscript. We say: a total-array1 is represented by a concrete-*array1*. A concrete-*array1* consists of a sequence of *scal*-values, neatly arranged in memory and a concrete domain [m:n] (a descriptor) with lowerbound m and upperbound n that describes this sequence of *scal*-values. In the same way a total-array2 is represented by a concrete-*array2* with two subscripts and a concrete domain [m1:n1,m2:n2].
Given a concrete-*array*, one should consider it as the total-array that is obtained by adding "many" zeros to the ends of its concrete domain.

In TORRIX-BASIS a clear distinction has been made between operations on total arrays and operations on concrete-*array*s, although both types of operations act on concrete-*array*s. All operations on total-arrays have a clear meaning derived from the mathematical theory of linear algebra. As a consequence the result of an operation on one or two total-arrays does not depend on the way a total-array is stored in memory. Several operations on concrete-*array*s can be interpreted as operations on special total-arrays. The number of operations in TORRIX-BASIS not clearly derived from corresponding operations in linear algebra, has been kept as low as possible. Needless to say, it would be in bad taste to define dyadic operators with a total-array and a concrete-*array* operand.

We can make another classification of operations, which is more related to ALGOL 68. Imagine that at some stage of execution (or elaboration) of a program using TORRIX-BASIS, there is a set of *scal*-locations in memory described by a multiset of concrete domains.

Notation:

$[m1:n1] \subseteq [m2:n2]$ if and only if $m1 \geq m2$ and $n1 \leq n2$,

$[m1:n1,p1:q1] \subseteq [m2:n2,p2:q2]$ if and only if

$[m1:n1] \subseteq [m2:n2]$ and $[p1:q1] \subseteq [p2:q2]$,

$D_1 = D_2$ if and only if $D_1 \subseteq D_2$ and $D_2 \subseteq D_1$ (D_1 and D_2 are concrete domains).

Each concrete domain D describes a set of *scal*-locations L(D). Whereas $\subseteq$ and $=$ have only been defined for concrete domains of the same type, $\subseteq_L$ and $=_L$ are defined for concrete domains of possibly different types:

$D_1 \subseteq_L D_2$ if and only if $L(D_1) \subseteq L(D_2)$,

$D_1 =_L D_2$ if and only if $D_1 \subseteq_L D_2$ and $D_2 \subseteq_L D_1$.

In ALGOL 68 it is possible to have D_1 and D_2 with:

$D_1 \cap D_2 = \emptyset$ but $D_1 =_L D_2$,

$D_1 = D_2$ but $L(D_1) \cap L(D_2) = \emptyset$,

and all situations in between.

For any TORRIX-BASIS program Π at some stage of elaboration we define L_Π as:

$L_\Pi = U\{L(D) : D \text{ is a concrete domain at this stage of elaboration}\}$.

Now we can make the following classification of operations into different types:

(i) operations that neither change a *scal*-value of a location in L_{Π} nor create a concrete domain.
Examples: questions about concrete domains,
the computation of inner products.

(ii) operations that create a concrete domain but neither change L_{Π} nor change the *scal*-value of a location in L_{Π}.
Examples: slicing,
making a concrete domain describing the main diagonal of an existing total-array2.

(iii) operations that do not create concrete domains, but change the *scal*-values of locations in L_{Π}.
Examples: assigning zero to a total-array,
adding a concrete-*array* to another concrete-*array*,
multiplying a total-array with a *scal*.

(iv) operations that change L_{Π} by adding a new concrete-*array* to it; of course there must be a new concrete domain as well to describe this concrete-*array*. In most cases *scal*-values will be assigned to this newly generated concrete-*array*.
Examples: the sum of two total-arrays,
the result of a linear mapping applied to a vector.

DEFINITION 1.12. An operation of type i is called an operation of the "i^{th} kind".

Based on this classification, and the features of the reference mechanism of ALGOL 68, we can distinguish two kinds of variability:

(i) variability at the level of the concrete-*array*: *scal*-locations get other *scal*-values. This is the variability of the first *ref*-level in ALGOL 68, (1.8)

(ii) variability at the level of the total-array: a total-array will be represented by another concrete-*array* with possibly another concrete domain. This is the variability of the second *ref*-level in ALGOL 68 and includes variability of type (1.8). (1.9)

A mapping in the mathematical sense (like +, -, etc.) may have several corresponding TORRIX-BASIS operators. Consider, for example, the problem of summing two vectors. TORRIX-BASIS provides the user with three operators to perform the addition. Let u and v be concrete-*array1* variables.

u+<v assumes the concrete domain of *v* to be a subset of the concrete domain of *u* (in the sense of ⊆); it assigns the sum of *u* and *v* to the *scal*-locations of *u*; it is an operation of the 3rd kind and an example of variability (1.8).

u + v generates a new concrete-*array1* containing the sum of *u* and *v*; neither *u* nor *v* is changed or its domain affected; it is an operation of the 4th kind.

u+:=v if the concrete domain of *v* is a subset of the concrete domain of *u*, then *u+<v* will be performed; otherwise *u* will be made to refer to the newly generated concrete-*array1* *u+v*; it is an operation of the 3rd or 4th kind and sometimes it shows variability (1.8), sometimes variability (1.9).

Finally we will give the two most important mode declarations of TORRIX-BASIS in ALGOL 68:

mode vec = *ref[]scal*,
mode mat = *ref[,]scal*;

A *vec* is a reference to a concrete-*array1* of *scal*s and a *mat* a reference to a concrete-*array2* of *scal*s. A *vec* (*mat*)-variable allows variability (1.9) and a *vec* (*mat*) on its own allows variability (1.8). For a complete listing of modes, operators and program-texts in TORRIX-BASIS, one is referred to [75].

I.4 EXTENSIONS OF TORRIX-BASIS.

TORRIX-BASIS provides the simplest possible implementation of a total-array. From here there are two possible ways to proceed. In the first direction one would try to solve problems arising from the abstract data type *scal* (see I.4.1 and I.4.2); in the second direction one would apply alternative storage schemes, especially for total-array2's (see I.4.3. and I.4.4).

I.4.1 Complexification of the field of scalars.

In practice one often has a problem in two vector spaces V and V_C over fields F and F_C, where F_C is a complexification of F.

DEFINITION 1.13. Let F be a field and $x \times x \neq -1$ for all $x \in F$. The complexification F_C of F is a field of which each element is a tuple of two elements of F and in which operations are defined as follows:

(i) $(x1,y1)+(x2,y2) = (x1+x2,y1+y2)$

(ii) $-(x1,y1) = (-x1,-y1)$

(iii) $(x,y)+(0,0) = (0,0)+(x,y) = (x,y)$

(iv) $(x1,y1) \times (x2,y2) = (x1 \times x2 - y1 \times y2,\ x1 \times y2 + y1 \times x2)$

(v) $(x,y)^{-1} = (x \times (x^2+y^2)^{-1}, -y \times (x^2+y^2)^{-1}) \quad (x,y) \neq (0,0)$

(vi) $(1,0) \times (x,y) = (x,y) \times (1,0) = (x,y)$

(vii) $\overline{(x,y)} = (x,-y)$

A theory similar to the one developed in I.1 can be developed for vector spaces over F_C. Instead of an inner product, one needs the concept of an hermitian form:

DEFINITION 1.14. Let F be an ordered field and V_C a vector space over F_C. An hermitian form is a mapping $<,>: V_C \times V_C \to F_C$ satisfying the following conditions:

(i) $<u,v> = \overline{<v,u>}$ $\qquad u,v \in V_C$

(ii) $<\alpha u+\beta v,w> = \alpha \times <u,w> + \beta \times <v,w>$ $\qquad u,v,w \in V_C,\ \alpha,\beta \in F_C$

(iii) Let $<u,u>=(x,y)$; then $y=0$ and $x \geq 0$

(iv) $<u,u>=(0,0)$ implies $u=0$

Though these concepts are commonly defined only for $F = \mathbb{R}$, the theorems can be easily generalized to arbitrary ordered fields F (cf. [74]).
Problems in the vector spaces V and V_C cannot easily be solved with TORRIX-BASIS since the implementation is based on just one field. In [76] a system TORRIX-COMPLEX has been defined which is an extension of TORRIX-BASIS. If TORRIX-BASIS is a computational system for working in vector spaces over an (ordered) field F, then TORRIX-COMPLEX is a system for working in vector spaces over F_C. If TORRIX-BASIS is based upon an implementation S of F, then TORRIX-COMPLEX is based on the implementation S_C of F_C.
From the technical point of view, there are no important new ideas in TORRIX-COMPLEX.

I.4.2 Infinite fields of scalars.

A field F with an infinite number of elements usually cannot be implemented perfectly on a computer. An implementation S of F will consist of a finite set $S \subseteq F$ and a number of operations (like +,-,×,etc.) which can be taken as adequate approximations to their mathematical counterparts. During a computation in such an S two kinds of errors (i.e., deviations from the pure mathematical computation) may occur:

(i) precision errors: operations return a *scal* $s' \in S$ which merely is a good approximation of the wanted scalar $s \in F$.

(ii) underflow/overflow: the execution halts or the operation returns a *scal* $s' \in S$ that is a very bad approximation of the wanted scalar $s \in F$.

For some infinite fields there may be several finite implementations available. Consider two finite implementations S and S' of a field F with $S \subseteq S' \subseteq F$ Usually it is assumed that a computation in S' will give a better approximation to the result of the pure mathematical computation than the corresponding computation in S, but that it requires more time. A user can decide to perform only crucial parts of the computation in S'.
In [76] an extension of TORRIX-BASIS has been given in which several operations can be performed in a better implementation of F.

I.4.3 Triangular and band matrices.

In many applications matrices (doubly subscripted finite sequences), as representations of linear mappings, have a large number of zero elements. For example:

(i) lower triangular matrices, i.e., matrices (α_{ij}) with $\alpha_{ij}=0$ for $j>i$,

(ii) band matrices (with bandwidths k_1 and k_2), i.e., matrices (β_{ij}) such that $\beta_{ij} \neq 0$ implies $-k_2 \leq i-j \leq k_1$.

These well-known classes of special matrices allow special algorithms. To implement any matrix of either kind as a total-array2 in TORRIX, one could define a primitive type *rowmat* by:

mode *rowmat* = *ref[]vec* *co* = *ref[]ref[]scal* *co*. (1.10)

Given a *rowmat* *r* we can find the corresponding total-array2 $[\alpha_{ij}]$ by:

α_{ij}=0 if r does not have an i^{th} *vec* or
if *r[i]* does not have a j^{th} *scal*,
= *r[i][j]* otherwise.

A *vec* of r corresponds to a row of $[\alpha_{ij}]$.
In the same way we can define a *diagmat* by:

mode diagmat = ref[]vec

and the k^{th} *vec* of a *diagmat* corresponds to the k^{th} diagonal of a total-array2 $[\beta_{ij}]$ (i.e., all elements β_{ij} with j-i=k).

Observe that the mode *rowmat* can also be used for representing other than triangular matrices. Neither is the use of *diagmat*s restricted to band matrices. For many matrices the storage scheme implied by using *rowmat* or *diagmat* requires less computer memory than a concrete-*array2* would. Of course there are disadvantages too. A concrete-*array2* has three kinds of substructures that are concrete-*array1*'s: rows, columns and diagonals. A *rowmat* and a *diagmat* each have only one concrete-*array1* substructure.
There is a third difference between *rowmat*s, *diagmat*s and concrete-*array2*'s. Associated with the concrete-*array2* implementation of total-array2's were two kinds of variability (see I.3.3). The *rowmat* implementation (and likewise the *diagmat* implementation) has three kinds of variability:

(i) changing the *scal*-value of a concrete *scal*-element (a *scal*-element (i,j) of the total-array2 $[\alpha_{ij}]$ is concrete if the corresponding *rowmat* contains a *scal*-location representing α_{ij}),
(ii) changing the concrete domain of a *vec* of a *rowmat*,
(iii) changing the number of *vec*s or their index in a *rowmat*.

In chapters III and IV we will give a number of results concerning algorithms working on a *rowmat* implementation of total-array2's.

I.4.4 Other sparse matrices.

The approach of I.4.3 gives good results only for matrices in which the non-zero elements are (will be, can be) situated in a special way. It is only useful for special sparse matrices.

DEFINITION 1.15 (cf. [63]). A matrix is said to be sparse if it has sufficiently many zero elements for it to be worthwhile to use special techniques that avoid storing or operating with the zeros.

Moreover, the approach of I.4.3 does not take advantage of the frequently occurring situation that many parts of a matrix are equal (see chapter II). In chapters II, V and VI we will propose a data structure for general sparse matrices to be used in an extension of the TORRIX-BASIS system, that aims at making the following desirable features explicit:

(i) each sparse matrix (α_{ij}) can be represented in this data structure, regardless of the distribution of the non-zero elements over the matrix. Thus, storing of and operating on zero elements can be avoided.

(ii) if large parts of a sparse matrix are equal, only one of these parts need to be stored in memory.

CHAPTER II

SPARSITY PATTERNS AND THE USE OF SPARSE MATRICES

Until now we have mentioned two data structures for matrices:

(i) the *mat* data structure, which can be used for full matrices,

(ii) the *rowmat* data structure, which can be used e.g. for full triangular matrices.

The choice of a data structure for a sparse matrix $A=(\alpha_{ij})_{1\leq i\leq m,1\leq j\leq n}$ in general should be determined by the following considerations:

(i) what is the sparsity pattern of A, i.e., which elements of A are zero and which are non-zero and how are the non-zero elements distributed over A?

(ii) which operations will be applied to A? The simplest operations are those that require access to each non-zero element of A once without any specific ordering in these accesses. Other operations can change values of elements of A and even change the sparsity pattern,

(iii) how "easy" is it to exploit the data structure; for example, is all software that manipulates matrices of this form available?

(iv) how much computer memory and computation time will be needed to store the sparse matrix and to perform the required operations on it?

Many algorithms for sparse matrices are designed for special sparsity patterns; thus it may be useful to change the sparsity pattern of a matrix A to let it fit one of these special formats. (For example, by using row- and column-permutations, by ignoring several zero elements and assuming tacitly that they have a non-zero value.)

In this chapter we will analyze the ways sparse matrices are manipulated in a variety of applications in order to gain insight into what a software system to manipulate sparse matrices should provide. We will identify a number of sparsity patterns that arise in practice and investigate several algorithms to solve a number of practical problems involving sparse matrices.

In the sections II.1 and II.2 we will concentrate on sparsity patterns that arise from solving differential equations and linear optimization problems with numerical methods. Block patterns like block tridiagonal form and doubly bordered block diagonal form turn out to be important. The definition of these two as well as some other sparsity patterns will be given in the appendix at the end of this chapter. In the sections of this chapter we will not include the definitions of these patterns once again. We will refer to them by abbreviations given in the appendix.
In section II.3 we will review a variety of common algorithms for sparse matrices. We will discuss methods for the solution of a linear system of equations and study their behavior for several sparsity patterns. Moreover, we will give a discussion of the simplex method, methods for the linear least square problem and the eigenvalue problem in the case of sparse matrices.
Each of these methods gives rise to a number of basic operations that a sparse matrix package should contain, like the matrix-vector product, the permutation of rows and of columns and the selection of a block of a partitioned matrix.
In the conclusion of this chapter we will present a list of important basic operations for sparse matrices.
A study of specific data structures for sparse matrices will be given in chapter V.

II.1 SPARSE MATRICES RELATED TO DIFFERENTIAL EQUATIONS.

Let L and M be appropriate differential transformations defined on a connected domain $G \subseteq \mathbb{R}^n$. ∂G is the boundary of G. The majority of differential equations arising from e.g. physics can be divided in three classes (cf. [02]):

(i) Boundary value problems: find a function u defined on G and satisfying $Lu = f$ within G, subject to certain boundary conditions $B_i u = g_i$ on the boundary ∂G. Very often G will be closed and bounded in $\mathbb{R}^n$.

(ii) Eigenvalue problems: find one or more $\lambda \in \mathbb{R}$ and corresponding functions u such that $Lu = \lambda Mu$ within G, subject to the boundary conditions $B_i u = \lambda E_i u$ on ∂G.

(iii) Initial value problems. The formulation of initial value problems is the same as for boundary value problems. However, the last parameter

of u is a time parameter and the projection of G into $\mathbb{R}^{n-1}$ is open (and can vary in time). The boundary conditions for t=0 are called the initial state.

Two well-known boundary conditions arising in the theory of differential equations are:

(i) the Dirichlet condition: $u(x) = g(x)$ for all $x \in \partial G$,

(ii) the Von Neumann condition: $\frac{\partial u}{\partial n}(x) = g(x)$ for all $x \in \partial G$, in which $\frac{\partial}{\partial n}$ denotes differentiation along the normal to the boundary directed away from the interior of G.

In practice other boundary conditions can occur as well, as can all kinds of combinations of them.

We say a differential transformation L is <u>linear</u> if

$$L(\alpha u + \beta v) = \alpha Lu + \beta Lv$$

for every α, $\beta \in \mathbb{R}$ and all functions u and v for which Lu and Lv are defined.

One can distinguish two kinds of methods for the numerical solution of differential equations:

(i) finite difference approximations (FDAs),

(ii) stationary and weighted residual methods.

In the following two sections we will identify a number of sparsity patterns that can occur if one of these methods is used. The numerical aspects of the methods are not the point at issue here and for these we refer to [02] and [57]. We always assume that a solution exists for all equations dealt with.

II.1.1 <u>Sparsity patterns obtained with finite difference approximations</u>.

In this section we will assume that L and M are linear differential transformations. Consider the following systems of differential equations:

$$Lu = f \tag{2.1}$$

$$\text{and} \quad Lu = \lambda Mu \tag{2.2}$$

on $G \subseteq \mathbb{R}^n$.

In a FDA method to solve (2.1) or (2.2) a mesh of N points is laid on the domain G. Then (2.1) (or (2.2)) is approximated in the meshpoints by means of finite differences (cf. [02]). With L (and M) linear, a linear system of the form

$$AU = F \tag{2.3}$$

$$\text{or} \quad AU = \lambda BU \tag{2.4}$$

is obtained with A (and B) N×N matrices and U and F vectors of length N. The meshpoints, the FDA and the numbering of the meshpoints are chosen in such a way that (2.3) (or (2.4)) can be solved adequately (i.e., in a reasonable amount of space, using a reasonable amount of time and yielding a reasonable approximation to the exact solution u in the meshpoints). Moreover, the choice is influenced by the domain G, by L (and M) and by the boundary conditions.

In (2.3) the number of non-zero elements in row i of A is equal to the number of meshpoints used in the FDA of (2.1) for meshpoint P_i ($1 \leq i \leq N$). From the viewpoint of error analysis it is needless in most cases to use every meshpoint in the FDA of every other meshpoint. Moreover, this would lead to a non-sparse matrix A (and B, in case of an eigenvalue problem) and computing the solution of (2.3) or (2.4) would require too much time by the sheer size of the system. Only if the distance between two meshpoints P_i and P_j (possibly i=j) is very small, can P_j be used in the FDA of the differential equation for P_i.

Example 2.1. Let $G = \{(x_1,\ldots,x_n) : l_i < x_i < u_i,\ 1 \leq i \leq n\} \subseteq \mathbb{R}^n$. Solve

$$\sum_{i=1}^{n} \left(a_i(x).\frac{\partial^2 u}{\partial x_i^2} + b_i(x).\frac{\partial u}{\partial x_i}\right) + c(x).u = f(x) \quad \text{for all } x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \in G \tag{2.5}$$

with $u(n) = g(x)$ on the boundary ∂G.

Let $N \in \mathbb{N}$ and $h_i = \frac{u_i - l_i}{N}$ ($1 \leq i \leq n$). Use a rectangular array of $(N+1)^n$ meshpoints $(P_{j_1,\ldots j_n})_{0 \leq j_i \leq N, 1 \leq i \leq n}$ in $G \cup \partial G$, with the j_i^{th}-coordinate of $P_{j_1,\ldots,j_n}$ equal to $l_i + j_i h_i$ ($1 \leq i \leq n$). The values of u in meshpoints on ∂G are already known, and we only have $(N-1)^n$ unknowns. We will order them in natural columnwise order:

$$u(P_{j_1,\ldots,j_n}) = U_k \quad \text{with } k = 1 + \sum_{i=1}^{n} (j_i - 1)(N-1)^{i-1}.$$

If in (2.5) n has value 1, we can use the following approximations:

$$\left.\begin{aligned} \frac{\partial^2 u}{\partial x_1^2}\Big/_{l_1+j_1h_1} &\approx \frac{u(l_1+(j_1+1)h_1) + u(l_1+(j_1-1)h_1) - 2u(l_1+j_1h_1)}{h_1^2} \\ \text{and } \frac{\partial u}{\partial x_1}\Big/_{l_1+j_1h_1} &\approx \frac{u(l_1+(j_1+1)h_1) - u(l_1+(j_1-1)h_1)}{2h_1} \end{aligned}\right\} \quad (2.6)$$

leading to a system AU=F with A an (N-1)×(N-1) matrix of TDF. For arbitrary functions a_1, b_1 and c, A is not symmetric.
If in (2.5) n=2, and we use the approximations (2.6) in both dimensions, we have a system AU=F with A an $(N-1)^2\times(N-1)^2$ matrix of SBTDF. Moreover, the main diagonal blocks are all (N-1)×(N-1) matrices of TDF and the other non-zero blocks are of DF.
For arbitrary n in (2.5) A is of recursive SBTDF. For arbitrary functions a_i, b_i $(1\leq i\leq n)$ and c, A is not symmetric but has a symmetric sparsity pattern. However, if $a_i(x)$ and $b_i(x)$ $(1\leq i\leq n)$ are not dependent of all coordinates of x, many non-zero elements of A may have the same value, many non-zero blocks may be equal and A may be symmetric. For this kind of sparse matrices, special techniques are designed to solve AU=F (cf. [17] and [18]).

This example is not the only case in which a number of blocks can be equal. If G can be split up in a number of subdomains $G = \bigcup_{i=1}^{p} G_p$ with $G_i \cap G_j = \emptyset$ $(i\neq j)$, such that several subdomains are of the same form, and L gives rise to the same transformation when restricted to these subdomains, while the discretizations are the same as well, then this would lead to a number of equal blocks under a suitable ordering of the meshpoints.
For initial value problems FDAs are used that result in a number of systems of equations $A_p U_p = F_p$ $(p=1,2,\ldots)$. Many of these matrices may have the same sparsity pattern (cf. [02]).

The natural ordering of the meshpoints and equations is not necessarily the most optimal for each solution algorithm. Moreover, there are algorithms that, during their execution, renumber the meshpoints and equations. We shall deal with renumbering in more detail in section II.3. Here we will restrict ourselves to two other possible ways of numbering the meshpoints:

(i) Try to partition the set $P = \{P_1,\ldots,P_N\}$ of meshpoints into $P = \bigcup_{i=1}^{k} P^{(i)}$ ($k \geq 2$) such that for any two different meshpoints P_i and P_j (with $P_i \in P^{(i')}$ and $P_j \in P^{(j')}$), $u(P_j)$ is used in the FDA of L in P_i if and only if $|i'-j'|=1$. Number the points in $P^{(i')}$ ($1 \leq i' \leq k$) consecutively, starting with $P^{(1)}$; number the equations in the same way.
The matrix A so obtained is of SBTDF, with the main diagonal blocks of DF. This ordering is important in case the SOR-method (cf. [21]) is used to solve the system AU = F (see II.3.4). Observe that the required partition of P does not always exist.

(ii) (Nested dissection, cf. [33]). Try to partition the set $P = \{P_1,\ldots,P_N\}$ of meshpoints into $P = \bigcup_{i=1}^{k+1} P^{(i)}$ ($k \geq 2$) such that for all points $P_i \in P^{(i')}$ ($1 \leq i'$ and $P_j \in P^{(j')}$ ($1 \leq j' \leq k$, $i' \neq j'$), $u(P_j)$ is not used in the FDA of L in P_i nor $u(P_i)$ in P_j. Apply this partition rule recursively to $P^{(1)},\ldots,P^{(k)}$.
If we number the meshpoints of $P^{(1)},\ldots,P^{(k+1)}$ (and the corresponding equations) consecutively, starting with $P^{(1)}$, then the matrix A so obtained is of DBSBDF and the first k main diagonal blocks have this same sparsity pattern.

We conclude that the sparsity patterns of many matrices obtained from FDAs are some combination of the following two patterns:

(i) the matrix A can be partitioned into $A=(A_{ij})$ and many blocks of this partition contain only zero elements,

(ii) many non-zero elements (or blocks) are organized along a few diagonals. These diagonals are not necessarily consecutive.

Moreover, very often the matrices are symmetric or have a symmetric sparsity pattern. Many blocks of a partitioned matrix can be equal. In many applications several sparse matrices with the same sparsity pattern must be manipulated.

II.1.2 Weighted residuals and stationary methods.

In the stationary methods and methods of weighted residuals the solution u of (2.1) is approximated by a linear combination of basis functions $(u_i)_{1 \leq i \leq k}$

$$u(x) \approx \tilde{u}(x) = \sum_{i=1}^{k} c_i . u_i(x).$$

Depending on how the boundary conditions are incorporated, one of the $(c_i)_{1\leq i\leq k}$ may be fixed. For convenience we will assume that none of the $(c_i)_{1\leq i\leq k}$ is fixed. For many differential equations a variational principle exists, i.e., there is a function $\Phi : C^0_{GU\partial G} \to \mathbb{R}$ which has an extreme value for the solution u of (2.1) ($C^0_{GU\partial G}$ is the set of continuous functions on $GU\partial G$). (For example, u minimizes the energy of a physical system.) All stationary methods have in common that the extreme value of

$$\{\Phi(\tilde{u}(x)) : c_i \in \mathbb{R}\}$$

is computed by solving the set of equations $\frac{\partial \Phi(\tilde{u}(x))}{\partial c_i} = 0 \quad (1\leq i\leq k)$.

In the methods of weighted residuals, the set of continuous functions on $GU\partial G$ is viewed as a vector space and the solution u is approximated by a function in the subspace spanned by $(u_i)_{1\leq i\leq k}$. For minimizing $u-\tilde{u}$ we need an inner product, norm or seminorm. Let (v,w) be the inner product

$$(v,w) = \int_G v(x).w(x)dx. \tag{2.7}$$

Several methods exist to obtain good approximations to u (cf. [02]):

the Galerkin methods:

Find $c_1,\ldots,c_k$ such that $(L\tilde{u}-f,u_i) = 0 \quad (1\leq i\leq k)$, (2.8)

the least square methods: Let $\|v\|=\sqrt{(v,v)}$. (2.9)

Find $c_1,\ldots,c_k$ such that $\|L\tilde{u}-f\|$ is minimal. (2.10)

A stationary method may be used to solve (2.10),

the collocation methods: Let $(P_i)_{1\leq i\leq m}$ be a set of points in $GU\partial G$.

Then we can approximate (2.7) and likewise (2.9) by

$$\langle v,w\rangle = \sum_{i=1}^{m} v(P_i).w(P_i), \qquad |v| = \left(\sum_{i=1}^{m} v^2(P_i)\right)^{\frac{1}{2}}. \tag{2.11}$$

Find $c_1,\ldots,c_k$ such that $|L\tilde{u}-f|$ is minimal. (2.12)

If L is linear, (2.8) and (2.12) (in case m=k) result in a linear system of equations Ac=F; if m>k (in (2.11)), (2.12) results in a linear least square problem. The elements of $A=(\alpha_{ij})$ are defined as:

$$\alpha_{ij} = (u_i,Lu_j) = \int_G u_i(x).(Lu_j)(x)dx \text{ in case of (2.8)}$$

and $\alpha_{ij} = (Lu_j)(P_i)$ in case of (2.12).

In many practical cases, L is symmetric and positive definite:

$(Lv,w) = (v,Lw)$, $(Lv,v) \geq 0$ for all v,w and $(Lv,v) > 0$ for all $v \neq 0$.

With a Galerkin method this results in a symmetric positive definite matrix A. If L is not symmetric, A very often has a symmetric sparsity pattern. All methods mentioned above give rise to a matrix A which may contain no zero element at all. Let us consider Galerkin methods in more detail.

In "finite element" methods (cf. [57] and [58]) the basis functions $(u_i)_{1 \leq i \leq k}$ are chosen such that the resulting matrix A is sparse. The word "element" refers to the support of one u_i. (The support of a function g on G is the set of points of G in which g has a non-zero value.) G is divided into a finite set of open subdomains such that the support of each basis function consists of one or more of these subdomains (possibly except a few points). It often happens that two different elements have the same form and, if similar basis functions are chosen, this may lead to a matrix with equal blocks.
Finite element methods are widely used because of their flexibility. If needed, the derivatives of the solution and the basis functions can be incorporated. Moreover, knowledge from physical problems can play a role in the choice of elements and basis functions. Because of this flexibility, all kinds of sparsity patterns can occur. In case of rather regular elements (triangles, rectangles, etc.) the sparsity pattern may be as in finite difference approximation methods. In [32] a choice and numbering of basis functions are proposed that lead to a matrix of DBSBDF in which all, except the last, main diagonal blocks have this same sparsity pattern.
Sometimes the solution $\tilde{u}$ obtained for the differential equation is not precise enough. Then the elements can be refined (i.e., divided into a number of subelements), points added, basis functions replaced and new basis functions added. For the matrix A this implies a change of values of some matrix elements, and the insertion of new rows and columns. It is possible that rows and columns must be inserted at every row- and column-position of the matrix.

Summarizing, finite element methods give often rise to matrices with the same sparsity pattern as matrices derived from finite difference approximation. Symmetry and symmetric sparsity patterns are important as well as the insertion of rows and columns. The matrix can have equal blocks. Rather arbitrary sparsity patterns are more likely than with finite difference approximation.

II.2 SPARSITY PATTERNS IN OPTIMIZATION PROBLEMS.

In the theory of (linear) optimization one often has to solve problems of the following type:

$$\left.\begin{array}{l}\text{minimize } \langle c,x\rangle \text{ subject to } Ax=b \text{ and } x_i \geqq 0 \quad (1 \leqq i \leqq n) \\ \text{with } c,x \in \mathbb{R}^n,\ b \in \mathbb{R}^m \text{ and } A \text{ an } m \times n \text{ matrix } (m \leqq n),\end{array}\right\} \quad (2.13)$$

where A may be sparse. Because linear optimization does not provide important new sparsity patterns, we will only briefly mention a number of them. The most important difference with matrices derived from differential equations is that A in (2.13) will never have fewer columns than rows. Frequently occurring patterns are:

(i) Block angular form (cf. [50]) (BAF, see appendix).

(ii) Dual block angular form (cf. [50]) (DBAF, see appendix).

(iii) A combination of block and dual block angular form: A can be partitioned into $A=(A_{hk})_{1 \leqq h,k \leqq p+1}$ such that $A_{hk}=0$ if $h \neq k+1$, $h \neq 1$ and $k \neq p+1$ (cf. [50]). It is clear that with a simple row-permutation, A obtains the DBBDF.

(iv) Staircase form (cf. [60]), i.e., A is of BBF with bandwidths 0 and 1.

In all these patterns the non-zero blocks can be sparse, even with a rather irregular pattern.

II.3 NUMERICAL ALGORITHMS AND SPARSE MATRICES.

Numerical methods in linear algebra and optimization theory do not all use matrices in the same way. Some methods only need a matrix-vector product, some only change the value of non-zero elements, others change the value of zero elements and (hence) the sparsity pattern, etc. In this section we will review a number of methods and will attempt to identify the basic operations that are applied to sparse matrices. We will deal with linear systems of equations (sections II.3.1 - II.3.4), matrix eigenvalue (II.3.5), linear least square (II.3.6) and linear optimization problems (II.3.7). We will see that many algorithms manipulate sparse matrices in a block-oriented fashion. Moreover, row- and column-permutations are often used and the matrix is often viewed as an ordered set of rows or an ordered set of columns.

II.3.1 Operations on sparse matrices related to LU decomposition.

Methods for solving the linear system of equations

$$Ax=b \qquad A=(\alpha_{ij})_{1\leq i,j\leq n} \tag{2.14}$$

are often divided into two classes: direct methods and iterative methods.

DEFINITION 4.1(cf. [21]). A method to solve (2.14) is direct if (in case we use exact arithmetic) the solution x is obtained after a finite number of steps (dependent on the number of rows and columns of A and the values of its elements). A method to solve (2.14) is iterative if it is not direct.

Iterative methods are often of the form:

A number of approximations $x^{(1)},\ldots,x^{(k)}$ of x are available;
if none of them is good enough, an approximation $x^{(k+1)}$ is derived from A, b, $x^{(1)},\ldots,x^{(k)}$, which is hopefully a better approximation of x.

The most commonly used iterative methods have the property that they work with the original matrix and involve no creation of additional non-zeros (cf. [62]), but there are iterative methods that create new non-zero elements (for an example, see [56]). We will also distinguish between direct and iterative parts in methods for the other matrix problems. It is clear that a direct method cannot contain an iterative part.

The system (2.14) is easy to solve when A is of LTF or UTF. In the LU decompo sition, A is factored as

$$A=L.U, \text{ with L of LTF and U of UTF.}$$

The solution of (2.14) can now be found by (back)solving the following two linear systems of equations:

$$Ly=b \quad \text{and} \quad Ux=y.$$

Sometimes, A is factored as A=L.D.U with D of DF and L of LUTF and U of UUTF. If A is symmetric, U equals the transpose L^T of L.
For sparse A, L and U can have non-zero elements in positions where A has zero elements. Let $L=(l_{ij})_{1\leq i,j\leq n}$ and $U=(u_{ij})_{1\leq i,j\leq n}$. Assume without loss of generality $u_{ii}=1$ $(1\leq i\leq n)$. Multiplying L with U leads to the following equations:

$$\left.\begin{aligned} l_{ij} &= \alpha_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj} \qquad 1\leq j\leq i\leq n \\ \text{and } u_{ij} &= l_{ii}^{-1}.(\alpha_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}) \qquad 1\leq i<j\leq n. \end{aligned}\right\} \quad (2.15)$$

From this one can derive:

$$\left.\begin{aligned} &l_{ij}\neq 0 \text{ if and only if } \alpha_{ij}\neq 0 \text{ or } l_{ik}\neq 0 \text{ and } u_{kj}\neq 0 \text{ for some k with } 1\leq k\leq j-1, \\ &u_{ij}\neq 0 \text{ if and only if } \alpha_{ij}\neq 0 \text{ or } l_{ik}\neq 0 \text{ and } u_{kj}\neq 0 \text{ for some k with } 1\leq k\leq i-1. \end{aligned}\right\} \quad (2.16)$$

We will deal only with the general case of (2.16) that subtraction of non-zero values does not lead to a zero element in L or U. Let

$$r(A,i) = \min(\{i\}\cup\{j:\alpha_{ij}\neq 0\}),$$

$$c(A,j) = \min(\{j\}\cup\{i:\alpha_{ij}\neq 0\}).$$

If $\alpha_{ii}\neq 0$ $(1\leq i\leq n)$, then r(A,i) and c(A,j) are the index of the first non-zero element in row i and column j of A, respectively. The envelope and fill of A are defined as follows:

$$Env(A) = \{(i,j) : j\geq r(A,i) \text{ and } i\geq c(A,j)\},$$

$$fill(A) = \{(i,j) : \alpha_{ij}=0 \text{ and } (l_{ij}\neq 0 \text{ or } u_{ij}\neq 0)\}.$$

The LU decomposition of A is only of practical interest if fill(A) contains only a small number of elements. Because numerical cancellation is not taken into account, we have

$$fill(A) \subseteq Env(A), \qquad Env(A) = Env(L+U).$$

There are matrices A with $fill(A) \cup \{(i,j) : \alpha_{ij}\neq 0\} \neq Env(A)$.
The envelope of a symmetric matrix or a matrix with a symmetric sparsity pattern is often called a <u>profile</u>.

How can we keep fill(A) small? The common technique is to permute rows and columns of A and to factor PAQ as PAQ=LU (for certain n×n permutation matrices P and Q). For a general review of these techniques to keep fill(A) small, we refer to [35], [37], [63] and [69]. Two of them are the so-called minimum degree algorithm (cf. [35]) and the related Markowitz criterion (cf. [63]). These require the following primitive operation:

Given i and k, find the cardinality of the set $\{j : j \geq k, \alpha_{ij} \neq 0\}$ and the cardinality of the set $\{j : j \geq k, \alpha_{ji} \neq 0\}$.

However, permuting A may be used also to obtain a numerically stable factorization. Unfortunately, these two desires cannot always be fully met simultaneously with one permutation of A. For more details we refer to [63]. As for row- and column-permutations in order to obtain a numerically stable factorization the following operations are important:

(i) Given k, find the in absolute value largest element (and its indices) of all elements with both indices at least k.

(ii) Given i and k, find the in absolute value largest element (and its column-index) of all elements in row i of A with column-index at least k.

(iii) Given j and k, find the in absolute value largest element (and its row-index) of all elements in column j of A with row-index at least k.

We conclude that the following operations are important:

(i) selecting a row or a column of a matrix,

(ii) selecting the main diagonal of a matrix,

(iii) changing the value of zero and non-zero elements,

(iv) row- and column-permutations,

(v) adding a part of a row (or column) multiplied with a scalar to the corresponding part of another row (or column),

(vi) computing the number of non-zero elements in a part of a row or column,

(vii) determining the in absolute value largest element (with its indices) in a part of a row, of a column or in a submatrix,

(viii) computing the inner product of a part of a row (or column) with a vector or a part of a column (or row).

II.3.2 Operations on sparse matrices in block factorization.

A slightly different approach to LU decomposition is block elimination. The matrix A is assumed to be partitioned into $A=(A_{ij})_{1\leq i,j\leq p}$ such that A_{ii} is square ($1\leq i\leq p$). Hopefully many blocks of A are zero (possibly after row- and column-permutations). Each main diagonal block must be non-singular.

A is factored into A=L.U with L and U of SBLTF and SBUTF. Rewriting (2.15) gives us:

$$\left.\begin{aligned} L_{ii}U_{ii} &= A_{ii} - \sum_{k=1}^{i-1} L_{ik}U_{ki} \quad (1\leq i\leq p) \\ L_{ij} &= (A_{ij} - \sum_{k=1}^{j-1} L_{ik}U_{kj}) \times U_{ii}^{-1} \quad (1\leq j<i\leq p) \\ U_{ij} &= L_{ii}^{-1} \times (A_{ij} - \sum_{k=1}^{i-1} L_{ik}U_{kj}) \quad (1\leq i<j\leq p) \end{aligned}\right\} \qquad (2.17)$$

and the backsolving equations are (with b, x and y partitioned as A):

$$\left.\begin{aligned} L_{11}y_1 &= b_1 & U_{pp}x_p &= y_p \\ L_{22}y_2 &= b_2 - L_{21}y_1 & U_{p-1,p-1}x_{p-1} &= y_{p-1} - U_{p-1,p}x_p \\ &\vdots & &\vdots \\ L_{pp}y_p &= b_p - \sum_{j=1}^{p-1} L_{pj}y_j & U_{11}x_1 &= y_1 - \sum_{j=2}^{p} U_{1j}x_j \end{aligned}\right\} \qquad (2.18)$$

From the viewpoint of programming (2.17) and (2.18) are easy to solve if each non-zero block of A does contain only a few zero elements. But if a main diagonal block of A is sparse, special techniques must be applied to this block. With several sparse main diagonal blocks different techniques can be used for these blocks depending on their sparsity pattern.

Block factorization does not give rise to many new operations, except that the operations of II.3.1 must now be applied to blocks. It requires a block oriented storage scheme in order to allow an efficient selection of blocks, especially the main diagonal blocks. For more details we refer to [13] and [31].

II.3.3 Other direct methods to solve a linear system of equations.

There are direct methods especially designed for matrices with a specific sparsity pattern. Their behavior is rather bad for general matrices or cannot be applied at all to matrices with other sparsity patterns.

Example 2.2. Marching algorithms and odd/even reductions are designed for banded or block banded matrices (cf. [04], [17], [18] and [64]). These are closely related to LU decomposition or block LU decomposition. The matrix is accessed along diagonals or block diagonals.

Example 2.3. In tearing and capacitance methods the matrix A is written as

$$A = B + C$$

where By=z is easy to solve and B differs from A in only a small number of rows and columns (cf. [14], [16]). For example, B is symmetric, B is block diagonal, or B can be solved with marching algorithms or odd/even reductions.

Conjugate gradient methods (cf. [03] for a review) are not very interesting for the design of a data structure, because the matrix A is only used for matrix-vector products. We conclude that the following operations are important:

(i) matrix-vector product,
(ii) matrix product of blocks,
(iii) selection of a diagonal,
(iv) selection of a block diagonal,
(v) deletion of a block of a sparse matrix.

II.3.4 Operations on sparse matrices in iterative methods to solve a linear system of equations.

Iterative methods are rather popular because they are easy to implement and, if convergence has been proven, no problems occur with rounding errors during the computation. Sometimes each step requires a linear system Kx=y to be solved, but K has always a rather simple sparsity pattern. Thus solving this system of equations does not require much time. Often matrix-vector products are involved and this operation allows a simple data structure for the matrix Iterative methods have a disadvantage in case (2.14) must be solved for many different right hand sides. To illustrate all this, we will give three examples. Many iterative methods to solve (2.14) use a splitting of A:

$$A=K-R \quad \text{with K non-singular.} \tag{2.19}$$

Equation (2.19) leads to the iteration scheme:

$$Kx^{(i+1)} = Rx^{(i)}+b. \tag{2.20}$$

How can K be chosen?

Example 2.4. Successive overrelaxation (SOR, cf. [21]) solves the slightly different system

$$\tilde{A}x=\tilde{b} \quad \text{with } \tilde{A}=\omega A,\ \tilde{b}=\omega b \text{ for a suitable } \omega\in\mathbb{R}.$$

Let $\tilde{A} = \tilde{L}+\tilde{D}+\tilde{U}$ with $\tilde{D}$ of DF, $\tilde{L}$ and $\tilde{U}$ of SLTF and SUTF. Then $\tilde{K} = \tilde{L}+\frac{\tilde{D}}{\omega}$ and $\tilde{R} = \tilde{D}(\frac{1}{\omega}-1)-\tilde{U}$ and we have the iteration:

$$(\tilde{L}+\frac{\tilde{D}}{\omega})x^{(i+1)} = (\tilde{D}(\frac{1}{\omega}-1)-\tilde{U})x^{(i)} +\tilde{b}. \tag{2.21}$$

However, SOR does not work with $\tilde{A}$ but with A: $L=\frac{\tilde{L}}{\omega}$, $D=\frac{\tilde{D}}{\omega}$, $U=\frac{\tilde{U}}{\omega}$. Substituting this in (2.21) leads to

$$(D+\omega L)x^{(i+1)} = ((1-\omega)D-\omega U)x^{(i)} +\omega b.$$

Observe that $D+\omega L$ is of LTF and each successive $x^{(i+1)}$ can be easily determined from $x^{(i)}$.

Block versions for this method are available: D is then of SBDF, L and U of strictly SBLTF and SBUTF. The block versions often converge faster at the cost of some complication in the computation of each iteration step (cf. [02]).

Example 2.5. Alternating direction methods (cf. [02]) are designed for matrices derived from finite difference approximations of boundary value problems in $\mathbb{R}^2$:

$$(B+V+E)x^{(i+1)} = b - (H-E)(B+H+D)^{-1}(b-(V-D)x^{(i)}) \tag{2.22}$$

with A = B+H+V and E and D some suitable matrices. B+V+E can be chosen in such a way that it consists of a full band or is of SBDF with each main diagonal block a full band. B+H+D can be permuted to this same sparsity pattern. (2.22) is a special case of (2.20) with

$$\tilde{K}=B+V+E \text{ to solve the system}$$
$$\tilde{A}x=\tilde{b} \text{ with } \tilde{A} = A - (H-E)(B+H+D)^{-1}A,$$
$$\tilde{b} = b - (H-E)(B+H+D)^{-1}b.$$

Example 2.6. Meijerink and Van der Vorst (cf. [56]) use a splitting as in (2.19) such that the L and U of the LU decomposition K=LU have only non-zero elements in positions where A has them. Moreover, R has non-zero elements only in those positions where fill could occur if A itself was factored. They

prove that this splitting exists if A is an M-matrix (i.e., $\alpha_{ij} \leqq 0$ $(1 \leqq i,j \leqq n,$ $i \neq j)$ and A^{-1} exists and has no negative elements).

We conclude that the following operations are important:

(i) selection of the main diagonal,

(ii) selection of the main block diagonal,

(iii) operations involved in LU decomposition,

(iv) changing the value of non-zero elements of a matrix.

II.3.5 Operations on sparse matrices related to the matrix eigenvalue problem.

Methods for solving the matrix eigenvalue problem

$$Ax=\lambda Bx$$

must be iterative if one can compute with the operations +, -, × and / only. Except for Householder and Givens transformations (which will be dealt with in II.3.6) they do not add new aspects concerning sparsity patterns and operations. For a bibliography we refer to [67].

II.3.6 Operations on sparse matrices related to the linear least square problem.

There are many classes of methods to solve the linear least square problem

$$\text{minimize } \|Ax-b\|_2, \quad A=(\alpha_{ij})_{1 \leqq i \leqq m, 1 \leqq j \leqq n, m \geqq n}. \tag{2.23}$$

For a review we refer to [09]. We will only deal with direct methods based on orthogonalization. For any m×n matrix A of which the columns are linearly independent, A can be factored as

$$\left.\begin{aligned} A = Q'^{T}.U \quad & \text{with } Q \text{ an } m \times m \text{ matrix } Q^T.Q=Q.Q^T=I \\ & \text{and } U \text{ an } m \times n \text{ matrix of UTF.} \end{aligned}\right\} \tag{2.24}$$

Then (2.23) reduces to the linear system of equations

$$U'x=b'$$

in which U' consists of the first n rows of U and b' consists of the first n elements of Qb. The columns of Q^T provide an orthonormal basis of $\mathbb{R}^m$.

In the Gram-Schmidt method, the first i columns of Q^T span the same subspace as the first i columns of A ($1 \leq i \leq n$). Column i of Q^T is a linear combination of the first i columns of A. In the same way, column i of A can be seen as a linear combination of the first i columns of Q^T and these coefficients form the upper triangular matrix U. This method can create many non-zero elements (cf. [70]). However, if A is of BAF, Q^T is of BAF and U is of UTF and BAF. For computational convenience the last m-n columns of Q^T do not need to be calculated.

If Householder transformations are used, Q (in (2.24)) is written as

$$Q = Q_n \cdot Q_{n-1} \cdot Q_{n-2} \cdot \ \cdots \ \cdot Q_1,$$

Q_j $(1 \leq j \leq n)$ orthonormal and of the form $Q_j = I - 2 \dfrac{W_j \cdot W_j^T}{\langle W_j[,1],\ W_j[,1] \rangle}$

with W_j an $m \times 1$ matrix.

Let $A_0 = A$, $A_j = Q_j \cdot A_{j-1}$ $(1 \leq j \leq n)$, $A_n = U$. To obtain $A_j[i,k] = 0$ for all $i > j \geq k$, a suitable choice of W_j can be made satisfying

$W_j[i,1] = 0$ if $i < j$,
$= 0$ if $i > j$ and $A_{j-1}[i,j] = 0$,
$\neq 0$ otherwise.

This method can create many non-zero elements in the A_j. However, if A is of BAF, it may be necessary to permute A such that $\alpha_{ii} \neq 0$ $(1 \leq i \leq n)$. In case this can be done with row-permutations only, only the blocks of the A_j that correspond to the non-zero blocks of A, will contain newly created non-zero elements (cf. [38])'.

In the Givens method, Q is written as

$$Q = Q_n \cdot Q_{n-1} \cdot \ \cdots \ \cdot Q_1$$

and each $Q_j = Q_j^{j-1} \cdot Q_j^{j-2} \cdot Q_j^{j-2} \cdot \ \ldots \ \cdot Q_j^1$ $(1 \leq j \leq n)$, with Q_j^i orthonormal for $1 \leq i \leq j-1$.

The Q_j^i are chosen such that multiplying A with Q_j^i means that row i and j of A are replaced by two linear combinations of these rows, creating a zero

at position (i,j). If A is of BAF, creation of new non-zero elements is restricted to the non-zero blocks. Row-permutations may be needed to obtain $\alpha_{ii} \neq 0$ $(1 \leq i \leq n)$ (cf. [38]).

These three methods of orthogonal factorization change the sparsity pattern by means of adding non-zero elements to a column (row) if another column (row) has a non-zero in a corresponding position provided that there is at least one position where they both have non-zero elements.

We conclude that the following operations are important:
(i) operations related to LU decomposition,
(ii) Givens transformation.

II.3.7 Operations on sparse matrices in the simplex method.

When applied to the linear program (2.13) the simplex method solves a sequence of linear systems

$$zB=d, \quad By=a \quad \text{and} \quad Bw=b \tag{2.25}$$

in which B consists of m linearly independent columns of A, w and d vectors of the corresponding coordinates of x and c and a is a column of A not in B determined by z. B is called a basis of A. If z, y and w are determined and $\langle c,x \rangle$ is not small enough, another basis is chosen such that $\langle c,x \rangle$ will be less with the new values of w inserted in x. The successive bases of A chosen differ only in one column: a column of B is exchanged with a. Let B and $\tilde{B}$ be two successive bases with corresponding linear systems

$$zB=d, \quad By=a \quad \text{and} \quad Bw=b, \tag{2.26}$$

$$\tilde{z}\tilde{B}=\tilde{d}, \quad \tilde{B}\tilde{y}=\tilde{a} \quad \text{and} \quad \tilde{B}\tilde{w}=b. \tag{2.27}$$

It is important to solve (2.26) in such a way that a solution of (2.27) can be easily obtained from the solution of (2.26). We will investigate a number of methods to solve (2.26) and determine how the sparsity pattern is exploited. For a general review we refer to [55].

Methods based on inverting the basis. Write B^{-1} as a product of elementary matrices $E_m.E_{m-1}. \ \ldots \ .E_1$ (n.b. an elementary matrix is the identity matrix except for one column), and for all $1 \leq j \leq i \leq m$

$$(E_i.E_{i-1}.\ \ldots\ .E_1.B)[k,j] = 1 \quad \text{if } k=j,$$
$$= 0 \quad \text{if } k\neq j.$$

Then $E_m.\ \ldots\ .E_1.\tilde{B}$ is an elementary matrix and $\tilde{B}^{-1}$ can be written as

$$\tilde{B}^{-1}=E_{m+1}.E_m.\ \ldots\ .E_1.$$

The computation of the inverse of each successive basis adds an elementary matrix to the list. From the viewpoint of computing time it may be necessary to reinvert the current basis (if the list becomes too long). When this reinversion must be done depends on the sparseness of the elementary matrices of the list. It is clear that row- and column-permutations of the bases are allowed to obtain a numerically stable inversion and/or to retain sparseness. Hellerman and Rarick (cf. [42] and [43]) permute B such that the elementary matrices remain sparse: B becomes of SBTF with many main diagonal blocks of LTF; the other main diagonal blocks (bumps) are of LTF except for a few columns (spikes). The most right upper element of a bump is always non-zero. Each time a reinversion is needed the algorithm of Hellerman and Rarick can be used.

Methods based on triangular factorization of the basis. The basis B of (2.26) can be factored into B=L.U with L and U of LTF and UTF. Commonly L^{-1} is written as a product of elementary matrices, interspersed with permutation matrices. For permutations to lower the fill, we refer to II.3.1 and II.3.2. To obtain a decomposition of $\tilde{B}$, several strategies may be used for inserting the new column $v=L^{-1}a$ in U. In general v will not be sparse.

(i) method of Bartels and Golub (cf. [06]): delete column r from B, move columns r+1,...,m one place to the left and a will be the new column m. Then $L^{-1}\tilde{B}$ is of UTF except for the positions (r+1+i,r+i) ($0\leq i\leq m-r$). The decomposition of $\tilde{B}$ can be computed by decomposing $L^{-1}\tilde{B}$ and adding m-r elementary matrices to the already obtained list. These new elementary matrices are very sparse: they are all identity matrices except for one diagonal and one off-diagonal element. New non-zero elements may occur in $\tilde{U}$.

Givens transformations (see II.3.6) may be used to decompose $L^{-1}B$.

(ii) Forrest and Tomlin (cf. [25]) propose a variant of the strategy of Bartels and Golub: after inserting the new column, row r of $\tilde{B}$ is placed

at the bottom of the matrix, moving m-r rows upwards. If $L^{-1}\tilde{B}$ is permuted in the same way, yielding $\tilde{U}'$, $\tilde{U}'$ is of UTF except for row m. The decomposition of $\tilde{U}'$ will give fill in the last row only. Hence, the upper triangular factors of the successive bases retain sparsity, except for the inserted columns.

Methods based on orthogonal factorization(cf. [38], [55]). B is factored as $B=Q.R$ with $Q^T.Q=I$ and R of UTF. This factorization has already been discussed in II.3.6. To obtain a factorization of $\tilde{B}$, insertion strategies can be used as mentioned with the methods based on triangular factorization.

Many other insertion strategies can be used. Some examples:

(i) Saunders (cf. [66]) proposes to permute all spikes obtained with the Hellerman and Rarick algorithm to the last columns of B (and permuting the rows likewise). The insertion can be done as in the methods based on triangular factorization of the basis.

(ii) In case of a staircase system (see II.2) and use of LU decomposition, L and U have the same sparsity pattern and the new column can be inserted such that this sparsity pattern is retained in U (cf. [55]).

In this section one extra sparsity pattern was mentioned: bumps and spikes. Also this pattern is block oriented. Basic operations for the simplex method are:

(i) row- and column-permutations,

(ii) cyclic row- and column-permutations,

(iii) operations related to the LU decomposition,

(iv) operations related to orthogonal factorization.

Conclusion. In this chapter we have reviewed a number of applications involving sparse matrices. We have seen that many sparsity patterns are block and/or diagonal oriented. Nevertheless, more arbitrary sparsity patterns can occur also. Symmetric matrices and matrices with a symmetric sparsity pattern are important. In many applications the sparse matrix can have equal blocks. As for operations on sparse matrices we have found that the following primitives are important:

(i) matrix-vector product,
(ii) matrix product of two blocks of a sparse matrix,
(iii) selecting a row, a column, the main diagonal or a block of a matrix,
(iv) changing the value of zero and non-zero elements,
(v) row- and column-permutations,
(vi) cyclic row- and column-permutations,
(vii) adding a part of a row (or column) multiplied with a scalar to the corresponding part in another row (or column),
(viii) Givens transformation,
(ix) computing the number of non-zero elements in a part of a row or column,
(x) determining the in absolute value largest element (with its indices) in a part of a row, of a column or of a submatrix,
(xi) computing the inner product of a part of a row (or column) with a vector or a part of a column (or row),
(xii) insertion of rows and columns,
(xiii) deletion of a block.

APPENDIX.

SOME COMMON SPARSITY PATTERNS.

Here we will list a number of sparsity patterns and their abbreviations. Most patterns are identified by the zero elements rather than by the non-zero elements. We will follow the characterization of Tewarson (cf. [71]) to a certain extent.

Let $A=(\alpha_{ij})_{1\leq i\leq m,1\leq j\leq n}$. We say that a square matrix A (i.e., m=n) has a symmetric sparsity pattern if $\alpha_{ij}\neq 0$ implies $\alpha_{ji}\neq 0$. Sometimes it is worthwhile to assume that A has a symmetric pattern by tacitly assuming that a number of zero elements are non-zero. The symmetric sparsity pattern can be combined with several, but not all, of the sparsity patterns listed below. Let A be partitioned into $A=(A_{hk})_{1\leq h\leq p,1\leq k\leq q}$. We say that this partition is square if and only if A_{hh} is a square matrix ($1\leq h\leq \min(p,q)$). Thus the main diagonal blocks are square. However, the off-diagonal blocks are not necessarily square. Even for a non-square partition we say that the A_{hh} ($1\leq h\leq \min(p,q)$) are main diagonal blocks.

DF Diagonal form: $\alpha_{ij}=0$ for all (i,j) with $i\neq j$,

BF Band form with bandwidths w_1 and w_2:

$\alpha_{ij}=0$ if $i-j<-w_2$ or $i-j>w_1$,

TDF Tridiagonal form: bandform with bandwidths 1 and 1,

LTF Lower triangular form: $\alpha_{ij}=0$ if $j>i$,

LUTF Lower unit triangular form:

$\alpha_{ij}=0$ if $j>i$ and $\alpha_{ii}=1$ $(1\leqq i\leqq \min(m,n))$,

SLTF Strictly lower triangular form: $\alpha_{ij}=0$ if $j\geqq i$,

UTF Upper triangular form: $\alpha_{ij}=0$ if $i>j$,

UUTF Upper unit triangular form:

$\alpha_{ij}=0$ if $i>j$ and $\alpha_{ii}=1$ $(1\leqq i\leqq \min(m,n))$,

SUTF Strictly upper triangular form: $\alpha_{ij}=0$ if $i\geqq j$,

BDF Block diagonal form: $A_{hk}=0$ for all (h,k) with $h\neq k$,

BBF Block banded form with bandwidths w_1 and w_2:

$A_{hk}=0$ if $h-k<-w_2$ or $h-k>w_1$,

BTDF Block tridiagonal form: block banded form with bandwidths 1 and 1,

BLTF Block lower triangular form: $A_{hk}=0$ if $k>h$,

BUTF Block upper triangular form: $A_{hk}=0$ if $h>k$,

SBBDF Singly bordered block diagonal form:

$p=q$, $A_{hk}=0$ if $h>k$ and $A_{hk}=0$ if $h<k<q$,

DBBDF Doubly bordered block diagonal form:

$p=q$, $A_{hk}=0$ if $h\neq k$ and $h\neq p$ and $k\neq q$,

BAF Block angular form:

$p=q+1$ and $A_{hk}=0$ for all (h,k) with $h\neq k$ and $h<p$,

DBAF Dual block angular form:

$q=p+1$, $A_{hk}=0$ for all (h,k) with $h\neq k$ and $k<q$.

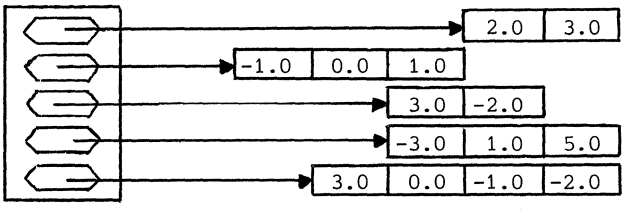
All these block patterns have a square-version in which the partition is square. The "S" of "square" is inserted just before the "B" of "block" in all abbreviations if this is the case.

CHAPTER III

COMPLEXITY RESULTS FOR THE CONSECUTIVE ONES PROPERTY

If we store a matrix A in a *rowmat* data structure of TORRIX, then this will not prevent that zero elements are stored in memory (see fig. 3.1). For many matrices more zero elements than non-zero elements will be stored. However, in many applications it is allowed to permute the columns of the matrix. Thus we can look for a column-permutation such that the permuted matrix can be stored in a *rowmat* without storing any zero elements. Unfortunately there are matrices for which such a column-permutation does not exist.

$$A=(\alpha_{ij})=\begin{pmatrix} 0.0 & 0.0 & 0.0 & 2.0 & 3.0 \\ -1.0 & 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 3.0 & -2.0 & 0.0 \\ 0.0 & 0.0 & -3.0 & 1.0 & 5.0 \\ 0.0 & 3.0 & 0.0 & -1.0 & -2.0 \end{pmatrix}$$ can be stored in



concrete domain [4:5]
" " [1:3]
" " [3:4]
" " [3:5]
" " [2:5]

bounds [1:5]

fig. 3.1. Storing a matrix in a *rowmat* data structure.

In analyzing the *rowmat* data structure, we are not interested in the exact value of a non-zero element, but only in the fact that it is non-zero. Therefore we will only consider {0,1}-matrices in the chapters III and IV.

DEFINITION 3.1(cf. [27]). An m×n {0,1}-matrix A has the consecutive ones property for rows (COR-property) if and only if there is an n×n permutation matrix P such that the ones of B = AP = $(\beta_{ij})_{1\leq i\leq m, 1\leq j\leq n}$ occur consecutively in each row, i.e., for each i ($1\leq i\leq m$) $\beta_{ij}=1$ and $\beta_{ik}=1$ imply $\beta_{ip}=1$ for all p with $j\leq p\leq k$.

In this chapter we will consider algorithmic aspects of the COR-property and several related problems. First of all we are interested in complexity results. For this purpose we need some terminology and results from graph theory (III.1.1) and from the complexity theory of algorithms (III.1.2). Then we will review the characterization of matrices without the COR-property in terms of submatrices (III.2). In section III.3 we will analyze the (often hard) problem of finding these submatrices in an arbitrary {0,1}-matrix A: we will prove several NP-completeness results and present algorithms that enumerate all these submatrices of A. In III.4 we will see how the COR-property can be exploited for two storage schemes for arbitrary sparse matrices; the *rowmat* data structure is one of them. We will present some complexity results for optimization problems related to these two storage schemes.

Further applications of the COR-property occur e.g. in the study of storage schemes for data bases (cf. e.g. [36]).

III.1 PRELIMINARIES.

III.1.1 Graph theory.

A graph G=(V,E) consists of a finite set V of vertices and a set E of unordered pairs (u,v) of vertices with $(u,u)\notin E$ for all $u\in V$. The elements of E are called edges. Two vertices u and v are adjacent if $(u,v)\in E$. The set N(v) of neighbors of v contains all vertices u adjacent to v. The degree of v is the number of neighbors of v. An edge (u,v) is incident to a vertex x if x=u or x=v; an edge (u,v) is incident to a subset $V'\subseteq V$ if either u or v is a vertex in V'. A path $\pi=(v_1,v_2,\ldots,v_n)$ ($n\geq 1$) is a sequence of vertices such that v_i is adjacent to v_{i+1} for all i with $1\leq i\leq n-1$. We say: π is a path from v_1 to v_n of length n-1. A cycle of length n is a path $(v_1,v_2,\ldots,v_n,v_1)$ with $n\geq 2$. A path $(v_1,\ldots,v_n)$ is called simple if it contains n dif-

ferent vertices. A cycle $(v_1,\ldots,v_n,v_1)$ is simple if the path $(v_1,\ldots,v_n)$ is simple. A simple path $(v_1,\ldots,v_n)$ in G has a chord if there are i and $j>i+1$ such that $(v_i,v_j)\in E$. A simple cycle $(v_1,\ldots,v_n)$ has a chord if $(v_1,\ldots,v_{n-1})$ or $(v_2,\ldots,v_n)$ has a chord. A simple path (cycle) without a chord is called chordless.

Let V' be a subset of V. The subgraph G(V') of G is the graph (V',E(V')) with $E(V') = \{(u,v)\in E : u\in V' \text{ and } v\in V'\}$. G is connected if there is a path from each $u\in V$ to each $v\in V$ with $v\neq u$. A subgraph G(V') is a connected component of G if G(V') is connected and for all $v\in V\setminus V'$ the subgraph $G(V'\cup\{v\})$ is not connected.

A graph G is bipartite if V can be partitioned into V_1 and V_2 (i.e., $V=V_1\cup V_2$ and $V_1\cap V_2=\emptyset$) such that for all $(v_1,v_2)\in E$ we have $v_1\in V_1$ and $v_2\in V_2$. A bipartite graph will subsequently be denoted as $G=(V_1,V_2,E)$. G is a clique if for all u and v in V $(v\neq u)$ we have $(u,v)\in E$.

If we number the vertices of G from v_1 to v_n, G can be represented uniquely by an $n\times n$ $\{0,1\}$-matrix $M(G)=(m_{ij})$ defined by:

$$m_{ij}=1 \text{ if and only if } (v_i,v_j)\in E.$$

M(G) is called the adjacency matrix of G. M(G) is symmetric (i.e., $m_{ij}=m_{ji}$ $1\leq i,j\leq n$). If we number the vertices of a bipartite graph $G=(V_1,V_2,E)$ in the following way: $V_1=\{v_1,\ldots,v_n\}$, $V_2=\{v_{n+1},\ldots,v_m\}$, then M(G) can be partitioned into

$$M(G) = \begin{pmatrix} O & N \\ N^T & O \end{pmatrix}$$

where N has n rows and m columns and N^T is the transpose of N. N (and not M(G)) is called the adjacency matrix of the bipartite graph G. Moreover, for each $\{0,1\}$-matrix A there is a bipartite graph G such that M(G)=A.

III.1.2 Complexity of algorithms.

In this section we will briefly review the pertinent notions from the complexity theory of algorithms. For a more complete introduction we refer to [01] and [29]. In the complexity theory of algorithms one distinguishes between a problem and an instance of that problem. An algorithm to solve a problem must return the right answer if it is applied to any instance of that problem. The input for an algorithm is a description of an instance of the

problem to be solved with this algorithm. Each input describes exactly one instance of the problem. Inputs are sequences of symbols. The length of an input is the length of this sequence of symbols. We assume that the description of instances is kept as short as possible. E.g. integral numbers are always described in binary or decimal (etc.) notation and certainly not in unary notation.

In the study of algorithms four aspects are important: the problem, an algorithm for the problem, an implementation of this algorithm and an assesment of its efficiency. We will consider only two types of problems. Problems that require a yes-or-no answer, are decision problems; problems in which a goal-function must be optimized (i.e., maximized or minimized) are optimization problems. For each optimization problem in which some integral number must be optimized, there is a natural correspondence to a decision problem.

Example 3.1.

LONGEST CHORDLESS PATH (optimization version):

Instance: a graph G.

Question: what is the length of the longest chordless path of G?

LONGEST CHORDLESS PATH (decision version):

Instance: a graph G; an integer k.

Question: is there a chordless path in G with length $\geq$ k?

If one has an algorithm for a decision version, it can be used in an algorithm for the optimization version and vice versa.

The complexity of an algorithm is measured by investigating the performance of an implementation of it on a sufficiently accurate computing model*). We will call such a computing model "the hypothetical machine".

DEFINITION 3.2.

(i) An algorithm X for a problem Y has a time (space) upperbound f(n), if there is a c>0 and an implementation Z of X on the hypothetical machine such that each instance of Y with input length n can be solved within

---

*) Usually the Random Access Machine (RAM) with the limited instruction set as in [01] is chosen. In the theory of NP-completeness the Turing machine is normally used, but if RAMs were used the results would remain essentially unchanged (cf. [01], chptr. 10).

time (space) c.f(n), often denoted by O(f(n)).

(ii) An algorithm X for a problem Y has a time (space) lowerbound f(n), if there is a c>0 such that for all implementations Z of X on the hypothetical machine and for all n>0 there is an instance I of Y with input length n and Z requires at least c.f(n) time (space) to solve I, often denoted by $\Omega(f(n))$.

(iii) An algorithm X for a problem Y has time (space) complexity f(n), if X has upperbound f(n) and lowerbound f(n), often denoted by $\Theta(f(n))$.

We say X is a polynomial time (space) algorithm, if it has upperbound p(n) for some polynomial p and X is an exponential time (space) algorithm, if it has a lowerbound f(n) with $\liminf_{n\to\infty} \frac{q(n)}{f(n)} = 0$ for each polynomial q (cf. [29], p. 6). Observe that the complexity as well as the upper and lower time and space bounds deal with the asymptotic behavior of an algorithm as $n\to\infty$. In general exponential time algorithms will be problematic in practice.

DEFINITION 3.3(cf. [29], p. 35).

(i) A problem Y has a time (space) upperbound f(n), if there is an algorithm X for Y with upperbound f(n).

(ii) A problem Y has a time (space) lowerbound f(n), if each algorithm X for Y has time (space) lowerbound f(n).

(iii) A problem Y has a time (space) complexity f(n), if it has upperbound f(n) and lowerbound f(n).

A problem can be solved in polynomial time (space), if there is a polynomial time (space) algorithm for it.

III.1.2.1 NP-complete problems.

NP-completeness is a classification of problems rather than of algorithms. Only exponential time algorithms are known to solve NP-complete problems at present, but the proven lowerbounds are all polynomial in the length of the input. Moreover, the set of NP-complete problems is defined in such a way that, should one discover a polynomial time algorithm for one NP-complete problem, there would be polynomial time algorithms for all NP-complete problems. And the other way round: if one could prove an exponential lowerbound for one NP-complete problem, then none of the NP-complete problems can be solved in polynomial time.

Essential for the theory of NP-completeness is the distinction between deterministic and nondeterministic algorithms. An algorithm is assumed to be deterministic unless it is explicitly stated to be nondeterministic. The state of a (deterministic or nondeterministic) algorithm consists of the current values of all its variables and the next instruction of the algorithm to be executed (on the hypothetical machine). An algorithm is deterministic, if the next state is uniquely determined by the current state and, if an input symbol must be read, this input symbol. The next state may be the termination of the algorithm. An algorithm is nondeterministic if the current state and the input symbol (if the instruction to be executed is a read-instruction) allow a bounded number of states to be the next state. With each execution of an instruction a guess will be made as to which of the possible next states will actually be the next state. Nondeterministic algorithms thus allow many different computations on a single input. The deterministic algorithms can be considered as a special instance of the nondeterministic algorithms.

DEFINITION 3.4.

(i) A nondeterministic algorithm X solves an instance I of a decision problem Y in time t if there is a computation according to X such that X returns "yes" if and only if I has answer "yes", and a yes-answer can be obtained in at most t computation steps.

(ii) X is a nondeterministic polynomial time algorithm to solve a decision problem Y, if there is a polynomial p such that, for all $n \in \mathbb{N}$ and all instances I of Y with length n, X solves I in time p(n).

Notation 3.1. P is the set of all polynomial time decision problems. NP is the set of all decision problems that can be solved in polynomial time with a nondeterministic algorithm.

Clearly $P \subseteq NP$.

DEFINITION 3.5. A problem Y_1 is polynomially transformable to a problem Y_2, if there is a transformation that transforms each instance I_1 of Y_1 in polynomial time (in the length of the input of Y_1) into an instance I_2 of Y_2 such that I_1 has a solution if and only if I_2 has a solution.

DEFINITION 3.6. A problem Y is NP-complete if $Y \in NP$ and if each problem $Y' \in NP$ is polynomially transformable to Y.

Cook has proven that the set of NP-complete problems is not empty (cf. [29]). Proving a problem Y NP-complete can often be done in two steps:

- prove that $Y \in NP$,
- take a suitable NP-complete problem Y'; find a polynomial transformation f that maps instances I' of Y' to instances of Y such that for all instances I' of Y': I' has a solution if and only if f(I') has a solution.

With the second step it is proven that each problem $Q \in NP$ is polynomially transformable to Y. An example of an NP-complete problem is (cf. [29]):

LONGEST PATH: (3.1)

Instance: a graph G; an integer $k \geq 0$.

Question: does G contain a simple path of length $\geq k$?

Remark: remains NP-complete if we ask for a simple cycle of length $\geq k$ (LONGEST CYCLE).

III.2 CONSECUTIVE ONES PROPERTY AND SUBMATRICES.

We recall the definition of the COR-property:

DEFINITION 3.1. An $m \times n$ $\{0,1\}$-matrix A has the consecutive ones property for rows (COR-property) if and only if there is an $n \times n$ permutation matrix P such that the ones in each row of AP occur consecutively.

DEFINITION 3.7. Let $A=(\alpha_{ij})$ be an $m \times n$ matrix and $B=(\beta_{ij})$ a $p \times q$ matrix ($p \leq m$, $q \leq n$).

B is a permuted submatrix of A, if there are sets $I=\{i_1,\ldots,i_p\}$ and $J=\{j_1,\ldots,j_q\}$ such that $\beta_{hk}=\alpha_{i_h j_k}$ for all h,k with $1 \leq h \leq p, 1 \leq k \leq q$.

If $I=\{i_1,i_1+1,\ldots,i_1+p-1\}$, $J=\{j_1,j_1+1,\ldots,j_1+q-1\}$, we say B is a submatrix of A and we write $B=A[i_1:i_1+p-1,j_1:j_1+q-1]$.

The following results are straightforward:

Lemma 3.1. *Let* $A=(\alpha_{ij})$ *be an* $m \times n$ $\{0,1\}$*-matrix.*

(i) If A *has the* COR*-property, then each permuted submatrix of* A *has the COR-property;*

(ii) Let P *be an* $m \times m$ *permutation matrix.* A *has the* COR*-property if and only if* PA *has the* COR*-property;*

(iii) If the m^{th} *row of* A *contains only non-zeros or at most one non-zero, then* A *has the* COR*-property if and only if* A[1:m-1,1:n] *has the* COR*-property;*

(iv) If $\alpha_{mj}=\alpha_{m-1,j}$ *for all* j *with* $1\leq j\leq n$*, then* A *has the* COR*-property if and only if* A[1:m-1,1:n] *has the* COR*-property;*

(v) Let P *be an* n×n *permutation matrix;* A *has the COR-property if and only if* AP *has the* COR*-property;*

(vi) If $\alpha_{in}=\alpha_{i,n-1}$ *for all* i *with* $1\leq i\leq m$*, then* A *has the* COR*-property if and only if* A[1:m,1:n-1] *has the* COR*-property.*

As we have seen above (III.1.1), each {0,1}-matrix A is the adjacency matrix of a bipartite graph $G=(V_R,V_C,E)$. A permutation of the columns (rows) corresponds to a reordering of the vertices of V_C (V_R). Permuted submatrices of A correspond to subgraphs of G and the COR-property for A corresponds to the V_C-consecutive arrangement for G:

DEFINITION 3.8 (cf. [72]). Let $G=(V_R,V_C,E)$ be a bipartite graph. G has a V_C-consecutive arrangement if the vertices of V_C can be arranged as $V_C=\{v_1,\ldots,v_n\}$ such that for each $w\in V_R$ we have

$$(w,v_i)\in E \text{ and } (w,v_j)\in E \text{ imply } (w,v_k)\in E \text{ for all } k \text{ with } i\leq k\leq j.$$

Convention 3.1. In all following figures of graphs vertices are denoted by circles and edges by lines. In case of a bipartite graph $G=(V_R,V_C,E)$ vertices of V_R are denoted by blackened circles and vertices of V_C by open circles.

The relation between matrices with the COR-property and special kinds of graphs as well as characterizations of matrices with regard to the COR-property have been studied thoroughly in the past (cf. [39]). We will only give the result needed in this chapter.

THEOREM 3.1 (Tucker, cf. [72]). *A bipartite graph* $G=(V_R,V_C,E)$ *(a* {0,1}*-matrix* A*) has a* V_C*-consecutive arrangement (the* COR*-property) if and only if* G (A) *contains none of the subgraphs (permuted submatrices) given in fig. 3.2.*

If a matrix A contains a permuted submatrix M_i, then we shall refer to A as simply "containing an M_i".

III.3 ALGORITHMIC ASPECTS OF THE EXISTENCE OF FORBIDDEN PERMUTED SUBMATRICES.

In this section we will deal with the problem of detecting the forbidden permuted submatrices (see fig. 3.2) in an arbitrary {0,1}-matrix A. The following questions arise:

(i) does A contain a forbidden permuted submatrix? This question can be solved in polynomial time with an algorithm of Booth and Lueker (cf. [11]).

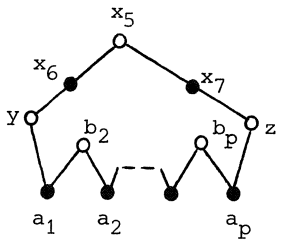
(ii) does A contain a forbidden permuted submatrix with at least k rows? We will prove that this problem is NP-complete.

(iii) list all forbidden permuted submatrices of A. We will give algorithms for each of the types of forbidden submatrices that are polynomial in the size of A and the number of forbidden submatrices that will be listed.

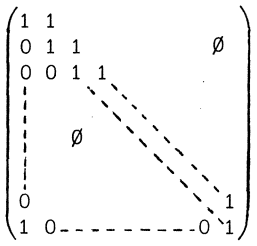
III.3.1 The existence of a forbidden permuted submatrix.

Let A be an m×n {0,1}-matrix. By Tucker's theorem the existence of a forbidden permuted submatrix is equivalent to A not having the COR-property. Booth and Lueker (cf. [11]) gave an algorithm to test whether A has the COR-property in O(m+n+f) time (f is the number of non-zeros in A). The algorithm is on-line: the rows of A are processed one by one. It starts with the set S of all column-permutations of A. Processing row i means that from S all permutations are eliminated which, when applied to row i, do not place the ones in row i in consecutive order. If S gets empty before all rows of A are processed, then A does not have the COR-property. Observe that this algorithm actually finds the largest p such that A[1:p,1:n] has the COR-property. In case p=m A has the COR-property.

The fact that S starts out as a set with an exponential (in the number of columns) number of elements, is not contradictory to the linear time bound of the algorithm. Booth and Lueker designed in their algorithm a data structure for S with only linear (in the number of columns) space.

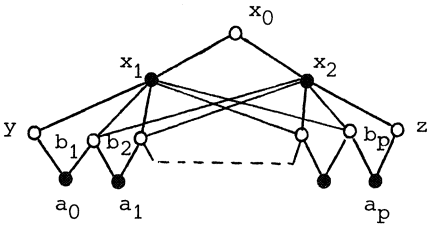


G_{I_p} with $p \geq 1$

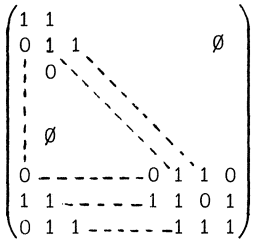


p+2 rows
p+2 columns

M_{I_p} with $p \geq 1$

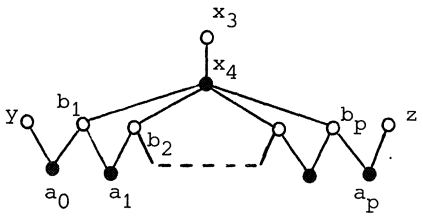


G_{II_p} with $p \geq 1$

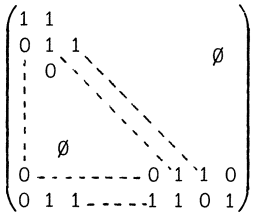


p+3 rows
p+3 columns

M_{II_p} with $p \geq 1$

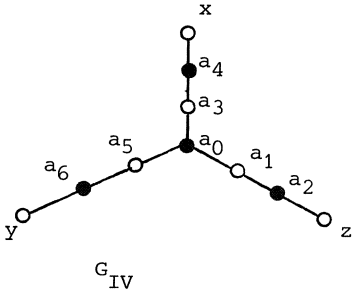


G_{III_p} with $p \geq 1$

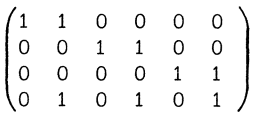


p+2 rows
p+3 columns

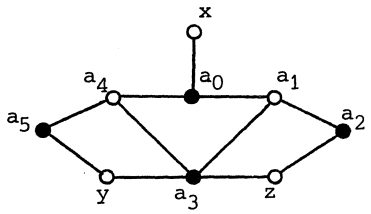
M_{III_p} with $p \geq 1$



G_{IV}



M_{IV}



G_V

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

M_V

fig. 3.2. Minimal bipartite graphs and matrices not having a V_C-consecutive arrangement or COR-property, resp.

III.3.2 The largest forbidden permuted submatrix.

Suppose A does not have the COR-property. To store A efficiently, a relevant question is how many rows the largest forbidden permuted submatrix of A has. It is easy to see that the question whether A contains an M_{IV} or M_V, can be answered in polynomial time: generate all 4×6 and 4×5 permuted submatrices of A and check whether one of them is an M_{IV} or M_V. We will see that finding the largest M_I in arbitrary {0,1}-matrices A is an NP-complete problem. The same holds for finding the largest M_{II} and M_{III}. In this section we will give these results in terms of bipartite graphs. As we have seen (III.2, fig. 3.2) the matrices M_I, M_{II}, M_{III}, M_{IV} and M_V correspond to the graphs G_I, G_{II}, G_{III}, G_{IV} and G_V. We will prove the NP-completeness of the following 6 problems:

LONGEST CHORDLESS CYCLE (INDUCED CYCLE):

Instance: a graph G; an integer $k \geq 3$.

Question: does G contain a chordless simple cycle of length $\geq k$?

LONGEST CHORDLESS PATH (INDUCED PATH):

Instance: a graph G; an integer $k \geq 2$.

Question: does G contain a chordless simple path of length $\geq k$?

MAXIMUM G_I SUBGRAPH:

Instance: a bipartite graph G; an integer $k \geq 1$.

Question: is there a $p \geq k$ such that G contains a G_{I_p} subgraph (a chordless simple cycle of length $\geq 2k+4$)?

MAXIMUM G_{II} SUBGRAPH:

Instance: a bipartite graph G; an integer $k \geq 1$.

Question: is there a $p \geq k$ such that G contains a G_{II_p} subgraph?

MAXIMUM G_{III} SUBGRAPH:

Instance: a bipartite graph G; an integer $k \geq 1$.

Question: is there a $p \geq k$ such that G contains a G_{III_p} subgraph?

MAXIMUM FORBIDDEN SUBGRAPH:

Instance: a bipartite graph G; an integer $k \geq 1$.

Question: is there a $p \geq k$ such that G contains a G_{I_p}, G_{II_p} or a G_{III_p} subgraph?

The first two problems are already mentioned in the literature and are attributed to M. Yannanakis (cf. [29]). However, as far as we know, the proofs have remained unpublished. We will provide proofs in this section.
We can always assume in the proofs of these 6 problems that k is greater than a fixed number (for example $k \geq 4$, $k \geq 7$, etc.) without loss of generality. The subgraphs G_{I_p}, G_{II_p} and G_{III_p} have in common that they contain a chordless simple path of length $\geq 2p+2$. In the NP-completeness proofs we make use of this property. We will use a basic transformation BT from a graph to a bipartite graph (see also fig. 3.3):

Given a graph $G=(V,E)$ with $V=\{v_1,\ldots,v_n\}$, $E=\{e_1,\ldots,e_m\}$. (3.2

$BT(G)=(W,F,D)$ with $W=\{w_1,w_2,\ldots,w_n\}$, $F=\{f_1,f_2,\ldots,f_m\}$,
$D=\{(f_j,w_i)$: e_j is incident to $v_i\}$

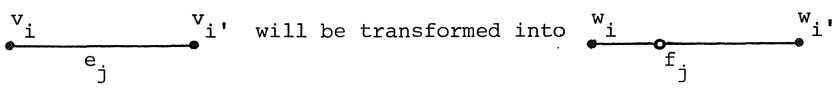


fig. 3.3. The transformation BT.

In the sequel we will often speak of the w-vertices and f-vertices of a transformed graph BT(G). Without proof we state:

Proposition 3.1. *Let* G=(V,E) *be a graph.*

(i) BT *transforms* G *into* BT(G) *in polynomial (in the length of a description of* G*) time;*

(ii) Each f-vertex of BT(G) *has degree* 2; (3.3)

(iii) BT(G) *does not contain a cycle of length* ≤ 5.

Proposition 3.2. *Let* G=(V,E) *be a graph as given in (3.2).*

(i) G *contains a simple path of length* p *if and only if* BT(G) *contains a chordless simple path of length* 2p.

(ii) G *contains a simple cycle of length* p *if and only if* BT(G) *contains a chordless simple cycle of length* 2p.

Proof: The proof of (ii) is less complicated than the proof of (i), though the same arguments are used in both proofs. Therefore, we will only give the proof of (i).

$\Rightarrow$: Suppose G contains a simple path $\pi=(v_{i_0},\ldots,v_{i_p})$ with edges $e_{h_j}=(v_{i_{j-1}}, v_{i_j})$, $1\leq j\leq p$. Then $\pi'=(w_{i_0},f_{h_1},w_{i_1},\ldots,f_{h_p},w_{i_p})$ is a path in BT(G) of length 2p. All vertices of π' are different, because π is a simple path. π' will not have a chord, because, with the fact that BT(G) is bipartite, it would contradict (3.3). Thus π' is a chordless simple path of length 2p.

$\Leftarrow$: Suppose BT(G) has a chordless simple path $\pi=(x_0,x_1,\ldots,x_{2p})$. There are two cases:

(i) x_0 is a w-vertex. Then x_{2i} $(0\leq i\leq p)$ is a w-vertex and x_{2i-1} $(1\leq i\leq p)$ is an f-vertex. The corresponding vertices and edges in G form a simple path of length p.

(ii) x_0 is an f-vertex. Then there is a unique vertex $x_{-1}\neq x_1$ in BT(G) adjacent to x_0. Because π is chordless, x_{-1} is unequal to x_{2i-1} $(2\leq i\leq p)$. Moreover, x_{-1} is not adjacent to any x_{2i} $(1\leq i\leq p-1)$ because of (3.3). Thus the path $\pi'=(x_{-1},x_0,\ldots,x_{2p-1})$ is a chordless simple path of length 2p in BT(G) and with the same arguments as given for case (i), we conclude that G contains a simple path of length p.

□

The NP-completeness proofs we give (theorem 3.2, 3.3, 3.4 and lemma 3.2, 3.3, 3.4) have the structure suggested in III.1.2.1 and all use a similar argument.

THEOREM 3.2. *THE LONGEST CHORDLESS PATH problem is NP-complete.*

Proof:

a) Given a graph G=(V,E) and an integer k, the following nondeterministic algorithm proves that LONGEST CHORDLESS PATH is in NP. Guess a vertex and with each step guess a next vertex that is adjacent to the previous vertex, and check that it is not adjacent to the other previous vertices. If G contains a chordless path of length k, the algorithm will find it with k guesses in polynomial time.

b) Recall LONGEST PATH is NP-complete (cf. (3.1)). Using proposition 3.2(i) the transformation BT easily establishes that LONGEST PATH is polynomially transformable to LONGEST CHORDLESS PATH.

□

Corollary 3.1. *The LONGEST CHORDLESS PATH problem for bipartite graphs is NP-complete.*

THEOREM 3.3. *The LONGEST CHORDLESS CYCLE problem is NP-complete.*

Proof: The proof is similar to the proof of theorem 3.2, this time using proposition 3.2(ii) to transform the LONGEST CYCLE problem (which is NP-complete by (3.1)) to the LONGEST CHORDLESS CYCLE problem.

□

Lemma 3.2. *The MAXIMUM* G_I *SUBGRAPH problem is NP-complete.*

Proof: This proof is equal to the proof of theorem 3.3.

□

Lemma 3.3. *The MAXIMUM* G_{II} *SUBGRAPH problem is NP-complete.*

Proof: Let a G_{II_p} graph have the labelling as given in fig. 3.2.

a) MAXIMUM G_{II} SUBGRAPH is in NP:

Let $G=(V_R,V_C,E)$ be a bipartite graph. Guess a number $p \in \mathbb{N}$ with $k \leq p \leq \min(|V_R|,|V_C|)-3$. Generate with 2p+6 guesses a subgraph G' of G with 2p+6 vertices. Locate for G' the vertices x_1 and x_2 (x_1 and x_2 are the only two vertices in G' with degree p+2); locate x_0 and check whether the remaining vertices of G' from a chordless path with:

(i) no vertex adjacent to x_0;

(ii) the first vertex adjacent to x_1 and not to x_2;

(iii) the last vertex adjacent to x_2 and not to x_1;

(iv) the 3^{rd},5^{th},... vertex adjacent to both x_1 and x_2.

If G contains a G_{II} subgraph with 2p+6 vertices, the given nondeterministic algorithm will find it in polynomial time.

b) We will show that LONGEST PATH can be polynomially transformed to MAXIMUM G_{II} SUBGRAPH.

Let H be an arbitrary graph with vertices $v_1,\ldots,v_n$ and edges $e_1,\ldots,e_m$. We extend BT(H) with the following vertices and edges:

vertices	edges	
t_1	(t_1,f_j)	for all j $(1\leq j\leq m)$,
t_2	(t_2,f_j)	for all j $(1\leq j\leq m)$,
t_0	(t_1,t_0) and (t_2,t_0),	
$y_1,\ldots,y_n$	(w_i,y_i) and (t_1,y_i)	for all i $(1\leq i\leq n)$,
$y'_1,\ldots,y'_n$	(w_i,y'_i) and (t_2,y'_i)	for all i $(1\leq i\leq n)$.

We call this graph H'. Claims concerning H':

(i) H' is bipartite, (3.4)

(ii) each f-vertex has degree 4, (3.5)

(iii) t_1 and t_2 have degree m+n+1; degree(t_0)=2, (3.6)

(iv) each y_i and y'_i has degree 2, (3.7)

(v) let (w_{i_0},u_1,w_{i_1}) and (w_{i_0},u_2,w_{i_1}) be two paths between two w-vertices; then $u_1=u_2$ and u_1 is an f-vertex, (3.8)

(vi) degree(w_i) = degree(v_i)+2 for all i $(1\leq i\leq n)$. (3.9)

We will prove that H has a simple path of length $p\geq k$ if and only if H' has a G_{II_p} subgraph.

$\Rightarrow$: Suppose H has a simple path $(v_{i_0},\ldots,v_{i_p})$ with edges $e_{j_1},\ldots,e_{j_p}$. Then $w_{i_0},\ldots,w_{i_p}$, $f_{j_1},\ldots,f_{j_p}$, y_{i_0}, y_{i_p}, t_0, t_1 and t_2 determine a G_{II_p} subgraph in H' in which $w_{i_0},\ldots,w_{i_p}$ are $a_0,\ldots,a_p$ and $f_{j_1},\ldots,f_{j_p}$ are $b_1,\ldots,b_p$ (see figure 3.2).

$\Leftarrow$: Suppose H' has a G_{II_p} subgraph. We have to show that $a_0,\ldots,a_p$ are w-vertices and $b_1,\ldots,b_p$ are f-vertices (see fig. 3.2). We assume that $p\geq k\geq 3$, thus degree(x_1) = degree(x_2) = $p+2\geq 5$. With (3.5) neither x_1 nor x_2 is an f-vertex. With (3.6) and (3.7) neither x_1 nor x_2 is t_0, y_i or y'_j for some i or j.

There are at least $p+1\geq 4$ distinct paths in H' of length 2 from x_1 to x_2. This means that (using (3.8)) x_1 and x_2 are not both w-vertices. There remains that x_1 and x_2 are t_1 and t_2 or only one of them is a w-vertex.

Claim: Neither x_1 nor x_2 is a w-vertex.

Proof: Suppose $x_1=w_{i_0}$ for some i_0 with $1\leq i_0\leq n$. Then x_2 must be t_1 or t_2. Without loss of generality we assume $x_2=t_2$. Now we look at y in the G_{II_p} subgraph.

case 1: $y=y_{i_0}$; then $a_0=t_1$. $b_1,\ldots,b_p$ are adjacent to x_1 (a w-vertex) and with $p\geq 3$ we have that at least two of them must be f-vertices and therefore a_0 is adjacent to at least two vertices of $b_1,\ldots,b_p$. Contradiction.

case 2: y is an f-vertex. With $x_2=t_2$, x_2 should have been adjacent to y. Contradiction.

case 3: $y=y'_{i_0}$. With the argument of case 2, we have a contradiction.

There are no other possibilities for y and we have a contradiction to the assumption that $x_1=w_{i_0}$.

Now we have $x_1=t_1$ and $x_2=t_2$. t_0 does not occur in $b_1,\ldots,b_p$ because its degree is too small. All this means that $b_1,\ldots,b_p$ must be f-vertices and $a_0,\ldots,a_p$ must be w-vertices. The corresponding vertices in H' determine a simple path of length $p\geq k$.

So, we conclude H' has a G_{II_p} subgraph if and only if H has a simple path of length $p\geq k$ and LONGEST PATH is polynomially transformable to MAXIMUM G_{I} SUBGRAPH.

□

Lemma 3.4. *The MAXIMUM* G_{III} *SUBGRAPH problem is NP-complete.*

Proof:

a) Analogous to the proof of lemma 3.3 we have that MAXIMUM G_{III} SUBGRAPH is in NP.

b) To prove that MAXIMUM G_{III} SUBGRAPH is NP-complete, we use a transformation from LONGEST PATH. Given a graph H and an integer k, we construct a bipartite graph H' by adding the following vertices and edges to BT(H):

vertices	edges
t_3	(t_3,t_4)
t_4	(t_4,f_j) for all j $(1\leq j\leq m)$
$z_1,\ldots,z_n$	(z_i,w_i) for all i $(1\leq i\leq n)$.

Claims concerning H':

(i) H' is bipartite,

(ii) each f-vertex has degree 3,

(iii) t_3 has degree 1 and t_4 has degree m+1,

(iv) for each i ($1 \leq i \leq n$) degree(z_i)=1 and degree(w_i) = degree(v_i)+1,

(v) let (w_{i_0},u,w_{i_1}) be a path between two w-vertices; then u is an f-vertex and unique.

We will show that H has a simple path of length $p \geq k$ if and only if H' has a G_{III_p} subgraph.

$\Rightarrow$: Suppose H has a simple path (v_{i_0},...,v_{i_p}) with edges e_{j_1},...,e_{j_p}. Then w_{i_0},...,w_{i_p}, f_{j_1},...,f_{j_p}, z_{i_0}, z_{i_p}, t_4 and t_3 determine a G_{III_p} subgraph in H'.

$\Leftarrow$: Suppose H' has a G_{III_p} subgraph (see fig. 3.2). Assume $k \geq 3$. Because degree(x_4)$\geq$4, x_4 must be t_4 or a w-vertex. With $k \geq 3$, it is impossible that x_4 and a_1 are both w-vertices or that x_4 and a_2 are both w-vertices, and therefore $x_4=t_4$. With $x_4=t_4$, b_1,...,b_p are all f-vertices and a_0,...,a_p are all w-vertices. The corresponding vertices in H determine a simple path of length $p \geq k$.

We conclude that LONGEST PATH is polynomially transformable to MAXIMUM G_{III} SUBGRAPH.

□

THEOREM 3.4. *The MAXIMUM FORBIDDEN SUBGRAPH problem is NP-complete.*

Proof:

a) With lemma 3.2, 3.3 and 3.4 we have that MAXIMUM FORBIDDEN SUBGRAPH is in NP.

b) To prove that MAXIMUM FORBIDDEN SUBGRAPH is NP-complete, we use a transformation from LONGEST PATH which is a combination of the transformations used in the proofs of lemma 3.3 and 3.4. Given a graph H and and an integer $k \geq 5$, we construct a bipartite graph H' by adding the following vertices and edges to BT(H):

vertices	edges	
t_1	(t_1,f_j)	for all j $(1\leq j\leq m)$
t_2	(t_2,f_j)	for all j $(1\leq j\leq m)$
t_0	(t_0,t_1) and (t_0,t_2)	
$y_1,\ldots,y_n$	(w_i,y_i) and (t_1,y_i)	for all i $(1\leq i\leq n)$
$y_1',\ldots,y_n'$	(w_i,y_i') and (t_2,y_i')	for all i $(1\leq i\leq n)$
t_4	(t_4,f_j)	for all j $(1\leq j\leq m)$
t_3	(t_4,t_3)	
$z_1,\ldots,z_n$	(z_i,w_i)	for all i $(1\leq i\leq n)$

Claims concerning H':

(i) H' is bipartite,

(ii) each f-vertex has degree 5,

(iii) degree(t_1) = degree(t_2) = m+n+1, degree(t_0)=2,

(iv) each y_i and y_i' has degree 2,

(v) each z_i has degree 1,

(vi) degree(t_4)=m+1, degree(t_3)=1,

(vii) degree(w_i) = degree(v_i)+3 for all i $(1\leq i\leq n)$,

(viii) if (w_{i_0},u,w_{i_1}) is a path in H', then u is an f-vertex and is unique.

Recall that $k\geq 5$. We will show that H has a simple path of length $p\geq k$ if and only if H' has a G_{I_p}, G_{II_p} or G_{III_p} subgraph:

$\Rightarrow$: if H has a simple path of length $p\geq k$, then H' has a G_{II_p} and a G_{III_p} subgraph. (See the proofs of lemma 3.3 and 3.4.)

$\Leftarrow$: We will deal with three cases:

case 1: H' contains a G_{III_p} subgraph (see fig. 3.2). The only possibilities for x_4 are that x_4 is a w-vertex, t_1, t_2 or t_4 (because degree(x_4) = $p+1\geq 6$). Then $b_1,\ldots,b_p$ can only be f-vertices, $y_1,\ldots,y_n$, $y_1',\ldots,y_n'$. However, with degree(b_i)≥ 3 we have that each b_i $(1\leq i\leq p)$ must be an f-vertex. The vertices $a_0,\ldots,a_p$ are w-vertices, t_1, t_2 or t_4. Because each b_i is adjacent to t_1, t_2 and t_4, each a_i $(0\leq i\leq p)$ must be a w-vertex. The vertices in H corresponding to $a_0,\ldots,a_p$ determine a simple path of length $p\geq k$.

case 2: H' contains a G_{II_p} subgraph (see fig. 3.2). degree(x_1) = degree(x_2) = $p+2>5$ and therefore x_1 and x_2 are w-vertices, t_1, t_2 or t_4; and $b_1,\ldots,b_p$ must be f-vertices. Because each f-vertex is adjacent to t_1, t_2 and t_4, each a_i $(0\leq i\leq p)$ is not t_1, t_2 or t_4, but a w-vertex. The vertices

in H corresponding to $a_0,\ldots,a_p$ determine a simple path of length $p \geq k$.

case 3: H' contains a G_{I_p} subgraph (see fig. 3.2). With $k \geq 5$, this subgraph contains at least 14 vertices. If the subgraph (which is a chordless cycle) contains t_1, t_2 and t_4, then it contains at least two f-vertices adjacent to t_4 and therefore is not chordless (contradiction).

If the subgraph (as a chordless cycle) contains t_1 and t_2 but not t_4, then it contains at most one of the $y_1,\ldots,y_n$, one of the $y'_1,\ldots,y'_n$ and t_0; thus the other ≥ 9 vertices are all w- or f-vertices with at least 4 f-vertices (contradiciton).

If the subgraph contains t_1 and not t_2 and t_4, then it contains at least 6 f-vertices (contradiction).

If the subgraph does not contain t_1 and t_2, but does contain t_4, then it contains at least 6 f-vertices (contradiction).

We conclude that the subgraph contains neither t_1 nor t_2 nor t_4 and therefore contains only w- and f-vertices. The graph H contains a simple cycle of length p+2 and therefore a simple path of length $p \geq k$.

Hence LONGEST PATH is polynomially transformable to MAXIMUM FORBIDDEN SUBGRAPH.

□

III.3.3 Enumerating all forbidden permuted submatrices.

In this section we consider the enumeration of all forbidden subgraphs of a bipartite graph $G=(V_R,V_C,E)$. According to [73] an enumeration problem is P-enumerable if all solutions can be listed in time $N.p(n)$ where N is the number of solutions found, p a polynomial and n the length of the input. In this section we will present an algorithm to enumerate all forbidden subgraphs of a bipartite graph G in time $O(|E|^2.N+p(n))$, thus proving that this problem is P-enumerable. We will make no distinction between a chordless path π in a graph G and the subgraph of G determined by the vertices of π. We will first prove that the number of solutions to be enumerated can be exponential in the size of G.

Fact 3.1. There is a sequence $(G_i)_{i\geq 3}$ of bipartite graphs such that each G_i has $\frac{(i+2)(i+5)}{2}$ vertices and each G_i contains at least $(i-1)!$ G_I subgraphs, at least $\frac{i!}{2}$ G_{II} subgraphs and at least $\frac{i!}{2}$ G_{III} subgraphs.

Proof: Let C_i be the clique with i vertices. G_i is the graph generated by applying to C_i the transformation used in the proof of theorem 3.4. So G_i has $\frac{(i+2)(i+5)}{2}$ vertices. To each simple cycle in C_i corresponds a G_I subgraph and to each simple path in C_i corresponds a G_{II} and a G_{III} subgraph in G_i and vice versa. C_i contains (i-1)! simple cycles of length i and $\frac{i!}{2}$ simple paths of length i-1 (for $i \geq 4$).

□

To detect all forbidden subgraphs in a bipartite graph we use a subroutine ENUMCHORDLESSPATHS that finds all chordless paths between two specified vertices. The following result is essential for the design of this subroutine.

Lemma 3.5. *Let G be a graph and* $\pi=(x=x_0,x_1,\ldots,x_n=y)$ *be a chordless path from x to y. Then:*

(i) for each j $(0 \leq j \leq n)$ *and* $k>j$, x_k *is not adjacent to* $x_0,\ldots,x_{j-1}$. (3.10)

(ii) Let π' *be another chordless path from x to y; then there is an* i $(1 \leq i \leq n-1)$ *such that* $x_i \notin \pi'$.

(iii) Let $\pi'=(x=q_0,q_1,\ldots,q_m=y)$ *be a chordless path from x to y; let* i_0 *be the least index with* $x_{i_0} \notin \pi'$; *then* $q_1=x_j$ *for all* j *with* $0 \leq j \leq i_0-1$.

Proof:

(i) Immediate from the definition of a chordless path.

(ii) Suppose π' contains $x_0,\ldots,x_n$. π' is different from π, so two possibilities arise:

(a) π' contains also a vertex $z \notin \pi$. Then two adjacent vertices of π are at different sides of z in π'. Thus π' is not chordless. Contradiction.

(b) π' is a permutation of the vertices of π. Then there is an i_0 $(1 \leq i_0 \leq n-1)$ such that x_{i_0} is in π' adjacent to x_j with $j \neq i_0-1$ and $j \neq i_0+1$. But then neither π nor π' is a chordless path. Contradiction

(iii) Similar to the proof of (ii).

□

Let $N(x)=\{a_1,\ldots,a_k\}$ be the set of neighbors of x; let $G-x-N(x)+a_i$ denote the graph obtained from G by deleting x and N(x) except a_i from G. The subroutine ENUMCHORDLESSPATHS finds all chordless paths from x to y ($x \neq y$) as follows:

for i=1,...,k generate all chordless paths from a_i to y in the graph $G-x-N(x)+a_i$; add x to all these paths π (denoted by $x+\pi$). (3.11)

Lemma 3.6. *Procedure (3.11) finds all chordless paths in a graph* G=(V,E) *from* x *to* y (x,y∈V, x≠y).

Proof: We have to prove that each detected path is chordless and that each chordless path in G from x to y will be found by this procedure.

(i) Suppose $\pi=(a_i,x_2,\ldots,x_n=y)$ is a chordless path of length $n-1\geq 0$ in G-x-N(x)+a_i for some i ($1\leq i\leq k$). By the construction of G-x-N(x)+a_i, $(x,a_i,x_2,\ldots,y)$ must be chordless in G.

(ii) Suppose $\pi=(x=x_0,x_1,\ldots,x_n=y)$ is a chordless path in G from x to y. Then $x_1=a_i$ for some i ($1\leq i\leq k$). With (3.10), $(x_1,\ldots,x_n)$ is a chordless path in G-x-N(x)+a_i of length at least 0.

□

Fact 3.2. Let G=(V,E) be a graph and x,y∈V (x≠y). If y is a neighbor of x, then (x,y) is the only chordless path from x to y.

Fact 3.3. Let G=(V,E) be a graph and x,y∈V (x≠y). If x and y are connected then there is a chordless path from x to y.

Proof: A shortest path connecting x and y is always chordless.

□

In view of these facts it is reasonable to let ENUMCHORDLESSPATHS make use of the following stopcriteria with regard to a recursive call:

(i) y is a neighbor of x, (3.12)

(ii) there is no path from x to y. (3.13)

As long as neither (3.12) nor (3.13) is satisfied, the subroutine will call itself recursively and will find a chordless path. It returns a *family**) of subsets of vertices. Each returned subset constitutes another chordless path from x to y.

algorithm 3.1.

```
proc ENUMCHORDLESSPATHS = (graph G, vertex x,y)family:
  co let G=(V,E) be a graph. ENUMCHORDLESSPATHS returns
     a family of chordless paths from x to y.          co
  if y∈N(x)                          then return ({x,y})          (1)
elif the connected component of G containing x does not contain y  (2)
```

---

*) *family* is neither an ALGOL 68 nor a TORRIX concept.

```
then return ∅                                                          (3)
else family S := ∅;                                                    (4)
     for all a∈N(x)                                                    (5)
      do family T = ENUMCHORDLESSPATHS(G-x-N(x)+a,a,y);                (6)
         for all π∈T do  S := SU{x+π}  od                              (7)
      od;                                                              (8)
     return S                                                          (9)
fi
```

Observe that when a *family* S is returned in line (9), it will not be empty. If each path $\pi \in S$ is represented as a linked list and a *family* as a linked list of subsets of vertices, line (7) uses an amount of time linear in the sum of the lengths of the paths found, when considered over all recursive calls of ENUMCHORDLESSPATHS. Line (2) can be performed in time linear in the number of edges in E (cf. [68]). If in line (6) integers are assigned to the vertices in G-x-N(x)+a, denoting the depth of recursion, computing the parameters requires at most $O(|V|)$ time (x and N(x) are not actually deleted from the data structure for G). Hence:

Lemma 3.7. *Let* $G=(V,E)$ *be a graph and* $x_0, y_0 \in V$ $(x_0 \neq y_0)$. *Algorithm 3.1 executes in time* $O(|E|^2.N)$, *where* N *is the number of chordless paths found.*

Proof: If N=0, then the algorithm halts in line (3) without any recursive call in time at most $O(|E|)$. For each occurrence of a vertex $v \neq y_0$ on a detected path, there was a call of ENUMCHORDLESSPATHS with x=v. In this call $O(|E|)$ time was needed for line (2). If line (6) was executed, then at least one $w \in N(v)$ is a vertex on a detected path. As for v, line (6) can be executed at most degree(v)-1 times without finding a path from v to y_0. The total time needed for this occurrence of v in a detected path (without counting line (7)) is bounded by $O(|E|) + (\text{degree}(v)-1).O(|E|)$. The total time needed by this algorithm can now be estimated at

$$\sum_{\substack{\text{all occurrences of } v \in V \\ \text{in a detected path}}} O(|E|).\text{degree}(v) \;+\; \sum_{\pi \in S} \text{length}(\pi) \;\leq$$

$$\leq\; O(|E|).\sum_{\pi \in S} \sum_{v \in \pi} \text{degree}(v) \;\leq\; O(|E|^2).\sum_{\pi \in S} 1 \;=\; O(|E|^2).N$$

□

Given ENUMCHORDLESSPATHS, it is easy to generate all forbidden G_I, G_{II} and G_{III} subgraphs of a bipartite graph $G=(V_R,V_C,E)$. We will use the word *bipgraph* to denote a data structure for a bipartite graph. Algorithm 3.2 generates all chordless cycles of length at least 6 in a bipartite graph:

algorithm 3.2.

```
proc GI = (bipgraph BG)family:
  co let BG = (V_R,V_C,E); GI returns the family of all subsets V'⊆ V_R∪V_C
     such that BG(V') is a G_I subgraph with at least 6 vertices.
  co
  (family S:=∅;                                                    (1)
   while E≠∅                                                       (2)
     do let (v,w)∈E;  E:=E\{(v,w)};                                (3)
        S := S ∪ ENUMCHORDLESSPATHS( (V_R∪V_C,E),v,w)              (4)
     od;                                                           (5)
   for all π∈S do if length(π)<6 then S := S\{π} fi od;            (6)
   return S                                                        (7)
  )
```

Note that, when $(v,w)\in E$, a chordless path from v to w in the graph $(V_R \cup V_C, E\setminus\{(v,w)\})$ (cf. line (4)) is a chordless cycle in the bipartite graph BG, and vice versa.

Lemma 3.8. *Let* $G=(V_R,V_C,E)$ *be a bipartite graph. Algorithm 3.2 enumerates all its* G_I *subgraphs in time* $O(|E|^2.(N+|V_R|^2.|V_C|^2))$.

Proof: Let $N_{(v,w)}$ be the number of subsets resulting from the call of ENUMCHORDLESSPATHS in line (4). The number N of chordless cycles in G is

$$N = \sum_{(v,w)\in E} N_{(v,w)} - |\{\pi_4 : \pi_4 \text{ is a cycle in G of length 4}\}|.$$

The amount of time needed for a call of ENUMCHORDLESSPATHS is $O(|E|^2).N_{(v,w)}$. Hence the total amount of time of *proc* GI is at most

$$\sum_{(v,w)\in E} O(|E|^2)N_{(v,w)} = O(|E|^2).(N + |V_R|^2.|V_C|^2).$$

□

Algorithm 3.3 generates all G_{II}-subgraphs of a bipartite graph $G=(V_R,V_C,E)$. The x_0 vertex of a G_{II}-subgraph is in V_C (see fig. 3.2).

algorithm 3.3(see fig. 3.2).

```
proc GII = (bipgraph BG)family:
  co let BG=(V_R,V_C,E), G=(V_R∪V_C,E), V_R={w_1,...,w_n}, V_C={v_1,...,v_m};
     GII returns the family of all subsets V'⊆ V_R∪V_C such that BG(V') is
     a G_II subgraph with x_0,y,z∈V_C
  co
  (family S := ∅;                                                    (1)
   for i to m                                                        (2)
    do co find all G_II subgraphs with v_i=x_0 co                    (3)
       for all w_j and w_k in N(v_i) with j<k                        (4)
        do co find all G_II subgraphs with v_i=x_0, w_j=x_1, w_k=x_2 co   (5)
           set B = (N(v_j)∩N(v_k))\{v_i};                            (6)
           set A = {w∈V_R : ∃v∈B with (v,w)∈E}\N(v_i);                (7)
           for all y ∈ N(w_j)\N(w_k)                                 (8)
            do for all z ∈ N(w_k)\N(w_j)                             (9)
                do graph G' = G(AUBU{y,z});                          (10)
                   family T = ENUMCHORDLESSPATHS(G',y,z);            (11)
                   for all π∈T do S := SU{πU{v_i,w_j,w_k} } od       (12)
                od
            od
        od
    od; return S
  )
```

Lemma 3.9. *Let* $G=(V_R,V_C,E)$ *be a bipartite graph. Algorithm 3.3 enumerates the* G_{II} *subgraphs with* $x_0 \in V_C$ *in time* $O(|E|^2.N + n.|E|.(n+m+|E|))$ *with* N *the number of* G_{II} *subgraphs of* G.

Proof: $O(|E|^2.N)$ time is used for all calls of ENUMCHORDLESSPATHS together. $O(n.|E|.(n+m+|E|))$ time is used to perform all loop-clauses without line (11). □

Lemma 3.10. *Let* $G=(V_R,V_C,E)$ *be a bipartite graph. The* G_{III} *subgraphs of* G *(see fig. 3.2) with* $x_3 \in V_C$ *can be enumerated in time* $O(|E|^2.N+|E|^2.m^2)$ *where* $m=|V_C|$ *and* N *is the number of* G_{III} *subgraphs of* G.

Proof: An algorithm similar to algorithm 3.3 can be used. It consists of a number of nested loop-clauses in which four vertices are chosen as x_3, x_4, y, z of G_{III}. Then (as in algorithm 3.3) a suitable subgraph G' of G is

determined and G' is searched for all chordless paths from y to z in G'. $O(|E|^2.N)$ time is used for all calls of ENUMCHORDLESSPATHS together and $O(|E|^2.m^2)$ time is used to perform the loop-clauses and the computation of all subgraphs G'.

□

Lemma 3.11. *Let* $G=(V_R,V_C,E)$ *be a bipartite graph. The* G_{IV} *subgraphs of* G *(see fig. 3.2) with* $x \in V_C$ *can be enumerated in time* $O(|E|^4.|V_C|)$.

Proof: In the algorithm to be used, every four edges of G are chosen as (x,a_4), (y,a_6), (a_5,a_0) and (z,a_2). Thus, all vertices except a_3 and a_1 are chosen. By computing $N(a_0) \cap N(a_4)$ and $N(a_0) \cap N(a_2)$ in G all possibilities for a_3 and a_1 are determined. To check whether the adjacency matrix of the subgraph of G determined by the chosen vertices of G, equals M_{IV} can be done in $O(|V_C|)$. This check must be performed partially before all possibilities for a_3 and a_1 are enumerated. Thus, this algorithm runs in time $O(|E|^4.|V_C|)$.

□

Lemma 3.12. *Let* $G=(V_R,V_C,E)$ *be a bipartite graph. The* G_V *subgraphs of* G *with* $x \in V_C$ *can be enumerated in time* $O(|E|^4.|V_C|)$.

Proof: Use an algorithm similar to the algorithm explained in lemma 3.11.

□

Theorem 3.5. *Let* $G=(V_R,V_C,E)$ *be a bipartite graph. The forbidden subgraphs of* G *(see fig. 3.2) can be enumerated in time* $O(|E|^2.N+f)$ *where* N *is the number of forbidden subgraphs of* G *and*

$$f=|E|.\max(|E|.|V_R|^2.|V_C|^2,\ |V_R|(|V_R|+|V_C|+|E|),\ |E|.|V_C|^2,\ |E|^3.|V_C|).$$

Proof: Apply the above algorithms subsequently to enumerate the forbidden subgraphs.

□

Observe in theorem 3.5 that $f=O(|V_R|^4.|V_C|^5)$, hence polynomially bounded.

Corollary 3.2. *Let* $G=(V_R,V_C,E)$ *be a bipartite graph. The problem to enumerate all forbidden subgraphs of* G *is* P*-enumerable.*

III.4 MINIMIZATION PROBLEMS RELATED WITH COR.

In this section we will investigate the consequences for the optimization of storage if a {0,1}-matrix does not have the COR-property. If we want to store such a matrix in a *rowmat* data structure, then we shall have to compromise and store embedded zero elements. In the following sections we will deal briefly with two minimization problems. For related minimization problems we refer to [10], [36] and [49].

III.4.1 Augmentation.

DEFINITION 3.9. Let $A=(\alpha_{ij})$ and $B=(\beta_{ij})$ be $m\times n$ {0,1}-matrices. B is a k-augmentation ($k\in\mathbb{N}$) of A if $\alpha_{ij}=1$ implies $\beta_{ij}=1$ and moreover there are exactly k different pairs (i_p,j_p), $1\leq p\leq k$, such that $\alpha_{i_p,j_p}=0$ and $\beta_{i_p,j_p}=1$ $(1\leq p\leq k)$.

$Aug^0(A) = \{A\}$, $Aug^k(A) = \{B : B \text{ is a k-augmentation of } A\}$.

Suppose there is a $B\in Aug^k(A)$ that has the COR-property. Let P be an $n\times n$ permutation matrix such that the ones of BP occur consecutively in each row. Then, if we store AP in a *rowmat* data structure, at most k zero elements of A will be stored. Moreover, if k is the least integer such that $Aug^k(A)$ contains a matrix with the COR-property, then AP requires k zero elements to be stored. This leads to the following problem:

CONSECUTIVE ONES MATRIX AUGMENTATION:

Instance: a {0,1}-matrix A; an integer $k\geq 0$.

Question: is there a p $(0\leq p\leq k)$ such that $Aug^p(A)$ contains a matrix with the COR-property?

THEOREM 3.6 (Booth, cf. [10]). *The CONSECUTIVE ONES MATRIX AUGMENTATION problem is NP-complete.*

This means that it will be very difficult to design a practical algorithm to find the column-permutation that is optimal with regard to the number of stored zero elements. In the next chapter we will prove negative results concerning the existence of simple schemes for even finding near optimal column-permutations. Observe that for every fixed k one can determine whether a {0,1}-matrix has a k-augmentation with the COR-property in polynomial time. There is only a polynomial number (in the size of the matrix A) of k-

augmentations of A, and each k-augmentation can be tested in linear time (cf. [11], III.3.1) for the COR-property.

III.4.2 Storing a number of (permuted) submatrices.

If an m×n {0,1}-matrix A does not have the COR-property, then we may try to divide A into a minimum number of submatrices $A[1:p_1,1:n]$, $A[p_1+1:p_2,1:n]$, ..., such that each submatrix has the COR-property. Each submatrix has its own column-permutation such that the ones in each row of the submatrix occur consecutively.

If it is not allowed to permute the rows of A, this problem can be solved in polynomial time (apply the algorithm of Booth and Lueker (cf. [11], III.3.1) several times), but if the rows of A can be permuted, then this problem is NP-complete (cf. [49]). Even to find a maximum set of rows of A that has the COR-property is NP-complete (cf. [10]). Nevertheless we will show that the following problem can be solved in polynomial time (in the size of A):

Let A have rows $r_1,\ldots,r_m$. Partition $R=\{r_1,\ldots,r_m\}$ into sets $R_1,\ldots,R_p$ such that:

(i) $|R_i| \geq |R_{i+1}|$ $(1\leq i\leq p-1)$, (3.14)

(ii) each R_i $(1\leq i\leq p)$ has the COR-property, (3.15)

(iii) for each i $(1\leq i\leq p-1)$ and for each row $r\in R_j$ with $j>i$ the set $R_i\cup\{r\}$ does not have the COR-property. (3.16)

The following algorithm will solve the problem.

algorithm 3.4.

(initialize $R_i:=\{r_i\}$ $(1\leq i\leq m)$;
while there is a j and an $r\in R_j$ such that for some $i<j$
$R_i\cup\{r\}$ has the COR-property
do $R_j := R_j\setminus\{r\}$; $R_i := R_i\cup\{r\}$;
sort the sequence $(R_i)_{1\leq i\leq m}$ according to decreasing number of rows
od;
let p be the greatest index such that $R_p\neq\emptyset$
)

If this algorithm terminates, $R_1,\ldots,R_p$ satisfy the requirements (3.14)-(3.16).

Proposition 3.3. *Algorithm 3.4 terminates and does so within polynomial time.*

Proof: Define:

$$f(R_1,\ldots,R_m) = \sum_{i=1}^{m} |R_i|.i \quad (|R_i| \text{ is the number of rows in } R_i)$$

f can only have nonnegative integer values. With each action consisting of a deletion, an addition and an ordering, the value of f decreases. The algorithm has to terminate otherwise f should get negative values. When the algorithm has processed the first line, $f(R_1,\ldots,R_m) = \frac{m(m+1)}{2}$ and the outer loop will be executed at most $\frac{m(m+1)}{2}$ times. It requires at most polynomial time to perform the instructions in the loop-clause. Thus the algorithm will halt within polynomial time in the size of A.

□

Conclusion. We have studied the COR-property of {0,1}-matrices as it is of interest for the storage efficiency to be achieved in TORRIX. The theorem of Tucker, characterizing the COR-property in terms of forbidden permuted submatrices, is important in the mathematical sense, but can hardly be used to design efficient algorithms: finding the largest forbidden permuted submatrix is proved to be NP-complete. (Indeed, the COR-test of Booth and Lueker (cf. [11]) does not rely on Tucker's theorem.) Unfortunately many optimization problems related to the COR-property are NP-complete as well. In this chapter we have not given any indication that the *rowmat* data structure is an economical storage scheme for each sparse matrix. Even if a sparse matrix may be permuted, it is hard to find an appropriate column-permutation to store a minimum augmentation matrix, or to find a suitable row-permutation to split up the matrix into a minimum number of permuted submatrices, each with the COR-property.

CHAPTER IV

APPROXIMATION OF THE CONSECUTIVE ONES MATRIX AUGMENTATION PROBLEM

In chapter III we saw that not every sparse matrix can be stored in a *rowmat* data structure without storing any zero elements, even if the columns are permuted. Moreover, the problem to find a column-permutation such that a minimum number of zero elements would be stored, turned out to be very hard: the problem is NP-complete. However, these results do not justify the conclusion that the *rowmat* data structure should not be used in practice. For such a conclusion there should be negative results in the following four directions:

a). We restrict ourselves to a special class C of matrices (which reflects the matrices used in practice) and try to design a polynomial time algorithm that finds for each matrix of C a column-permutation such that a minimum number of zero elements will be stored.

b). We try to design a probabilistic ("usually efficient") algorithm that finds for each matrix a column-permutation such that a minimum number of zero elements will be stored. For most matrices such an algorithm should run in polynomial time, but for a (hopefully small) number of matrices it may need exponential time.

c). We try to design a polynomial time algorithm that finds for each matrix a column-permutation such that a (hopefully) small, but not necessarily minimum, number of zero elements will be stored. These algorithms are called polynomial time approximation algorithms.

d). Some combination of a), b), c).

In this chapter we will concentrate only upon approximation algorithms. The main result is given in section IV.2 (theorem 4.3 and 4.4) and states that any algorithm from the class of on-line column insertion algorithms must give arbitrarily bad approximations for an infinite number of matrices. This means that no on-line column insertion algorithm can guarantee a bounded error factor with regard to the number of stored elements of a minimum augmentation. Section IV.1 serves as an introduction for this theorem; in section IV.2 the

theorem will be proved and in section IV.3 we will extend the result to much wider classes of approximation algorithms.

IV.1 INTRODUCTION AND PRELIMINARIES.

Each sparse matrix A can be stored in a *rowmat* data structure without permuting its columns. To lower the number of stored zero elements we allow that columns are permuted.

DEFINITION 4.1. Let $A=(\alpha_{ij})$ be an $m\times n$ $\{0,1\}$-matrix.

$\underline{ones(A)} = \sum_{i=1}^{m}\sum_{j=1}^{n} \alpha_{ij}$ (the number of non-zero elements of A),

$\underline{store(A)} = \sum_{i=1}^{m} (\max\{j:\alpha_{ij}\neq 0\} - \min\{j:\alpha_{ij}\neq 0\} +1)$ with $\max(\emptyset)=0$ and $\min(\emptyset)=1$,

$\underline{optstore(A)} = \min\{store(AP):P \text{ is an } n\times n \text{ permutation matrix}\}$.

An $m\times n$ matrix B is a column-permutation of A if $B=AP$ for some $n\times n$ permutation matrix P. B is an optimum column-permutation of A if $store(B) = optstore(A)$.

DEFINITION 4.2. Let $A=(\alpha_{ij})$ be an $m\times n$ $\{0,1\}$-matrix and B a column-permutation of A. Let column i of A be column p_i of B $(1\leq i\leq n)$. We say that α_{i_0,j_0} of A is stored in B if there are j_1 and j_2 with

$$p_{j_1}\leq p_{j_0},\ p_{j_2}\geq p_{j_0} \text{ and } \alpha_{i_0,j_1}=\alpha_{i_0,j_2}=1.$$

As we have seen in III.4.1, the problem to find for each matrix A an optimum column-permutation B of A, is NP-complete. But we may be content if we have a polynomial time algorithm that finds a column-permutation B of A (for each A) such that e.g.

$$\frac{store(B)}{ones(A)} \leq c$$

for some fixed constant c, because otherwise there are other more appropriate data structures for sparse matrices.

Proposition 4.1. *If such an algorithm exists, then* $c\geq\frac{3}{2}$.

Proof: Consider the special case of M_{I_n} (cf. III.2).

$$optstore(M_{I_n}) = store(M_{I_n}) = 2(n+1)+n+2, \quad ones(M_{I_n}) = 2n+4.$$

Therefore, for every column-permutation B_n of M_{I_n} we have

$$\frac{\text{store}(B_n)}{\text{ones}(M_{I_n})} \geq \frac{\text{optstore}(M_{I_n})}{\text{ones}(M_{I_n})} = \frac{3n+4}{2n+4}.$$

Note that for each $c' < \frac{3}{2}$ there is an n such that $\frac{\text{store}(B_n)}{\text{ones}(M_{I_n})} > c'$.

Hence $c \geq \frac{3}{2}$.

□

However, it is more realistic to compare store(B) with optstore(A) than with ones(A). If store(B) is close to optstore(A), then B is considered a "good" column-permutation of A. Therefore, to analyze an algorithm X, we will use as a criterion the magnitude of the ratio

$$\frac{\text{store}(B)}{\text{optstore}(A)} \text{ with B a column-permutation of A found by X.} \qquad (4.1)$$

Moreover, we are interested in the asymptotic behavior of X:

DEFINITION 4.3. Let $f: \mathbb{N} \to \mathbb{R}$ be a mapping. Let X be an algorithm that takes {0,1}-matrices as input and that returns a column-permutation of its input. X is an f-approximation algorithm (for the CONSECUTIVE ONES MATRIX AUGMENTATION problem) if there is an $N \in \mathbb{N}$ such that for all $m \times n$ {0,1}-matrices A $(m, n \geq N)$

$$\frac{\text{store}(X(A))}{\text{optstore}(A)} \leq f(\text{optstore}(A)).$$

As we have seen, we are only interested in c-approximation algorithms with c some constant. It should be nice if there are algorithms of that kind that have a running time that is polynomial in the length of their input.

DEFINITION 4.4. Let A be a {0,1}-matrix. A is said to be clean if the following five conditions are satisfied:

(i) each column of A contains at least one non-zero element,
(ii) each row of A contains at least two non-zero elements,
(iii) each row of A contains at least one zero element,
(iv) no two rows of A are equal,
(v) no two columns of A are equal.

DEFINITION 4.5. Let A be an $m \times n$, B an $m \times p$, C an $m \times (n+1)$ and D an $m \times (n+p)$ {0,1}-matrix and let u be a {0,1}-sequence of length m.

C = A concat u if C[,1:n] = A and C[,n+1] = u.
D = A concat B if D[,1:n] = A and D[,n+1:n+p] = B.

Now we will give two examples of straightforward approximation algorithms that are not c-approximation algorithms for any c≧1.

algorithm 4.1.

```
proc BESTFIT = (matrix A)matrix:
co let A be an m×n {0,1}-matrix co
( initialize B as the m×0 {0,1}-matrix;
 for k from 1 to n
  do initialize C as the m×k {0,1}-matrix with only non-zero elements;
     for j from 1 to k
      do matrix D = B[ ,1:j-1] concat A[ ,k] concat B[ ,j:k-1];
         if store(D) < store(C) then C:=D fi
      od; B:=C
  od;
  return B
)
```

BESTFIT processes the columns of A one by one; processing column k means that it will be inserted in B such that the current columns of B do not loose their relative order; column k is inserted in B just there where it causes the least number of elements of B (including column k) to be stored.

Proposition 4.2. BESTFIT *is not a c-approximation algorithm for any* c≧1.
Proof: We have to prove that for each N∈ ℕ and each c∈ ℝ (c≧1) there is a {0,1}-matrix A with at least N rows and N columns such that

$$\frac{\text{store(BESTFIT(A))}}{\text{optstore(A)}} > c.$$

Let N∈ ℕ, c∈ ℝ (c≧1); let

$$A_{p,k} = \begin{pmatrix} 0\ldots\ldots0 & 1\ldots\ldots1 & 1\ldots\ldots1 \\ 1\ldots\ldots1 & \vdots\quad\vdots & 0\ldots\ldots0 \\ \vdots\quad\vdots & \vdots\quad\vdots & \vdots\quad\vdots \\ \vdots\quad\vdots & 1\ldots\ldots1 & 0\ldots\ldots0 \\ 1\ldots\ldots1 & 0\ldots\ldots0 & 1\ldots\ldots1 \end{pmatrix} \begin{matrix} \\ \\ \\ \\ \uparrow \\ p+2 \\ \downarrow \end{matrix}$$

$$\underbrace{\qquad}_{p+1} \quad \underbrace{\qquad}_{p+1} \quad \underbrace{\qquad}_{k}$$

p+2 rows
2p+k+2 columns
k≧2, p≧1

optstore($A_{p,k}$) = store($A_{p,k}$) = $(p+1)^2 + (p+1)(p+2) + 2k$. If BESTFIT is applied to $A_{p,k}$, it returns

$$\text{BESTFIT}(A_{p,k}) = \begin{pmatrix} 1\ldots\ldots1 & 1\ldots\ldots1 & 0\ldots\ldots0 \\ \vdots\ \ \vdots & 0\ldots\ldots0 & 1\ldots\ldots1 \\ \vdots\ \ \vdots & \vdots\ \ \vdots & \vdots\ \ \vdots \\ 1\ldots\ldots1 & 0\ldots\ldots0 & \vdots\ \ \vdots \\ 0\ldots\ldots0 & 1\ldots\ldots1 & 1\ldots\ldots1 \end{pmatrix}$$

$\overleftrightarrow{p+1}$ $\overleftrightarrow{k}$ $\overleftrightarrow{p+1}$

and $$\frac{\text{store}(\text{BESTFIT}(A_{p,k}))}{\text{optstore}(A_{p,k})} = \frac{2(p+1)^2 + (p+2)k}{(p+1)^2 + (p+1)(p+2) + 2k} = \frac{2\frac{p+1}{p+2} + \frac{k}{p+1}}{\frac{p+1}{p+2} + 1 + \frac{2k}{(p+1)(p+2)}} \approx \frac{p+2}{2}$$

if k large compared with p. Thus, with p and k large enough, we have $p+2 \geq N$, $2p+2+k \geq N$ and $\frac{p+2}{2} > c$. We conclude that BESTFIT is not a c-approximation algorithm for any $c \in \mathbb{R}$.

□

Observe that the columns of $A_{p,k}$ are sorted by non-increasing number of non-zero elements. BESTFIT has a rather bad performance because one column u can occur many times and u can have less zero elements than a few columns together with many non-zero elements per column. Thus algorithm 4.2 below in which the columns are ordered by increasing number of non-zero elements, may perform better. As for identical columns we have the following lemma.

Lemma 4.1. *Let* A *be an* m×n {0,1}*-matrix. Then there is an optimum column-permutation* B *of* A *such that identical columns of* B *are consecutive in* B.

Proof: Let C be an optimum column-permutation of A and suppose not all identical columns of C are consecutive. Thus there are $j_0, j_1, j_2 \in \mathbb{N}$ $(j_0 < j_1 < j_2)$ such that the columns j_0 and j_2 of C are identical and the columns j_0 and j_1 of C are not identical. Without loss of generality we can assume that the number k of stored elements of column j_0 is not greater than the number of stored elements of column j_2. If we delete column j_2 and insert it just beside column j_0 (thus obtaining a {0,1}-matrix C') then exactly k elements of column j_2 of C are stored in C'. Then:

$$\text{store}(C') = k + \text{store}(C[\,,1:j_2-1] \text{ concat } C[\,,j_2+1:n]) \leq \text{store}(C).$$

Because C was assumed to be an optimum column-permutation of A, we have store(C') = store(C). Thus we can permute the columns of C in such a way

that the number of stored elements will not increase and identical columns will be consecutive. This gives us another optimum column-permutation of A. □

DEFINITION 4.6. Let A be an m×n matrix. We say that A has a row-partition $K^r = (0=k_0^r<k_1^r<k_2^r<\ldots<k_p^r=m)$ and a column-partition $K^c = (0=k_0^c<k_1^c<\ldots<k_q^c=n)$ if A can be written as $(A_{ij})_{1\leq i\leq p,1\leq j\leq q}$ (see fig. 4.1) such that for each i $(1\leq i\leq p)$ and j $(1\leq j\leq q)$ A_{ij} is a $(k_i^r-k_{i-1}^r)\times(k_j^c-k_{j-1}^c)$ matrix. In this case $\underline{\text{rowstrip}}(A,i,K^r) = A[k_{i-1}^r+1:k_i^r, \ ]$ and $\underline{\text{colstrip}}(A,j,K^c) = A[\ ,k_{j-1}^c+1:k_j^c]$.

$$A=(A_{ij})_{1\leq i\leq p,1\leq j\leq q} = \left(\begin{array}{c|c|c|c} A_{11} & A_{12} & & A_{1q} \\ \hline A_{21} & A_{22} & & A_{2q} \\ \hline & & \diagdown & \\ \hline A_{p1} & A_{p2} & & A_{pq} \end{array}\right)$$

fig. 4.1. A partitioned matrix.

DEFINITION 4.7.

(i) Let $A=(A_{ij})_{1\leq i\leq p,1\leq j\leq q}$ be a block diagonal matrix (cf. II. Appendix). The i^{th} block of A is the submatrix A_{ii} $(1\leq i\leq\min(p,q))$.

(ii) Let $A=(\alpha_{ij})$ be an m×n matrix. A is a permuted block diagonal matrix with k blocks if there are m×m and n×n permutation matrices P and Q such that PAQ can be partitioned into a block diagonal matrix with k blocks.

Without proof we state:

Lemma 4.2. *Let* A *be an* m×n {0,1}*-matrix with row-partition* $K^r = (k_0^r,\ldots,k_p^r)$. *Then:*

$$\text{store}(A) = \sum_{i=1}^{p} \text{store}(\text{rowstrip}(A,i,K^r)).$$

If A *is a block diagonal matrix with row-partition* K^r *and column-partition* $K^c = (k_0^c,\ldots,k_q^c)$, *then*

(i) for each i $(1\leq i\leq\min(p,q))$:

$$\text{store}(\text{rowstrip}(A,i,K^r)) = \text{store}(\text{colstrip}(A,i,K^c)) = \text{store}(A_{ii}),$$

(ii) $\text{optstore}(A) = \sum_{i=1}^{p} \text{optstore}(\text{rowstrip}(A,i,K^r))$

$= \sum_{i=1}^{q} \text{optstore}(\text{colstrip}(A,i,K^c))$

$= \sum_{i=1}^{\min(p,q)} \text{optstore}(A_{ii})$.

algorithm 4.2.

proc BESTFITDECR = (matrix A)matrix:

(let B be a column-permutation of A such that the number of zero elements per column in B is non-increasing. Moreover, identical columns of A are consecutive in B.

Then perform BESTFIT(B) with the modification that identical columns are inserted simultaneously and are kept in consecutive order in the result of BESTFIT(B)

)

BESTFITDECR processes the matrices $A_{p,k}$ ($p \geq 2$) of the proof of proposition 4.2 very well:

$$\frac{\text{store}(\text{BESTFITDECR}(A_{p,k}))}{\text{optstore}(A_{p,k})} = \frac{\text{store}(A_{p,k})}{\text{optstore}(A_{p,k})} = \frac{\text{store}(A_{p,k})}{\text{store}(A_{p,k})} = 1.$$

BESTFIT works rather well for a block diagonal matrix A: the columns of each columnstrip of A remain consecutive in BESTFIT(A). However, BESTFIT can have a bad performance for permuted block diagonal matrices, which will not necessarily be improved by BESTFITDECR.

Proposition 4.3. BESTFITDECR *is not a c-approximation algorithm for any* $c \geq 1$.

Proof: We have to prove that for each $N \in \mathbb{N}$ and each $c \in \mathbb{R}$ ($c \geq 1$) there is an $m \times n$ $\{0,1\}$-matrix A ($m,n \geq N$) such that

$$\frac{\text{store}(\text{BESTFITDECR}(A))}{\text{optstore}(A)} > c.$$

Let $N \in \mathbb{N}$, $c \in \mathbb{R}$ ($c \geq 1$). Let A_k be a block diagonal matrix with 2k blocks and each block equal to

$$A_{ii} = M_{I_1} = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \qquad (1 \leq i \leq 2k).$$

Let $B_k = (\beta_{ij})$ be the $6k \times 7k$ $\{0,1\}$-matrix with $B_k[\,,1:6k] = A_k$ and for each i ($1 \leq i \leq k$) $\beta_{3(i-1)+1,6k+i} = \beta_{3k+3(i-1)+1,6k+i} = 1$ (see fig. 4.2). Let $K^c = (0,3,6,\ldots,6k,7k)$.

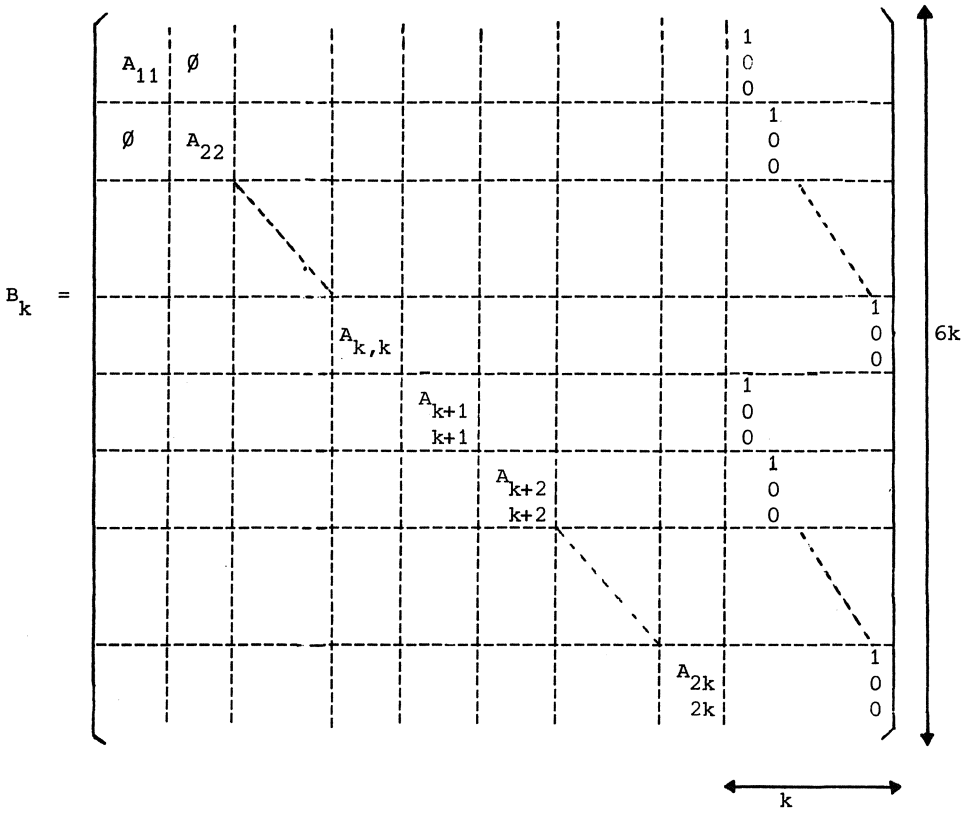


fig. 4.2.

B_k is a permuted block diagonal matrix with k blocks. All columns of B_k are different and each column contains two non-zero elements. BESTFITDECR returns the columns of B_k in the ordering:

$$\text{colstrip}(B_k,2k,K^C),\ B_k[\ ,7k],\ \text{colstrip}(B_k,2k-1,K^C),\ B_k[\ ,7k-1],\ \ldots,$$
$$B_k[\ ,6k+2],\ \text{colstrip}(B_k,k+1,K^C),\ B_k[\ ,6k+1],\ \text{colstrip}(B_k,k,K^C),$$
$$\text{colstrip}(B_k,k-1,K^C),\ \ldots,\ \text{colstrip}(B_k,1,K^C).$$

Thus store(BESTFITDECR(B_k)) =

$$\sum_{i=1}^{k} \text{store}(A_{ii}) + \sum_{i=k+1}^{2k} (\text{store}(A_{ii})+1) + \text{ones}(B_k[\ ,6k+1:7k]) +$$

$$+ \sum_{i=1}^{k} (3(k-1)+i-1) = \frac{k}{2}(7k+27).$$

On the other hand, $\text{optstore}(B_k) = 2k + \sum_{i=1}^{2k} \text{optstore}(A_{ii}) = 2k+14k = 16k$.
With $6k \geq N$ and k large enough we have

$$\frac{\text{store}(\text{BESTFITDECR}(B_k))}{\text{optstore}(B_k)} = \frac{7k^2+27k}{32k} > c.$$

Hence, BESTFITDECR is not a c-approximation algorithm for any $c \geq 1$.

□

DEFINITION 4.8. Let A be an m×n {0,1}-matrix, u a {0,1}-sequence of length m and B an m×(n+1) {0,1}-matrix. B is an insertion matrix for u in A if for some $y \in \mathbb{N}$ $(0 \leq y \leq n)$

B = A[,1:y] concat u concat A[,y+1:n].

DEFINITION 4.9. Let X be an algorithm that takes {0,1}-matrices as input and that returns a column-permutation of its input. X is an on-line column insertion algorithm if for each m×(n+1) {0,1}-matrix A, X(A) is an insertion matrix for A[,n+1] into X(A[,1:n]).

Proposition 4.4. BESTFITDECR *is not an on-line column insertion algorithm.*
Proof:

$$\text{Let } A = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}. \quad \text{BESTFITDECR}(A) = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

$$\text{BESTFITDECR}(A[\ ,1:3]) = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \text{and therefore, BESTFITDECR}(A)$$

is not an insertion matrix for A[,4] into BESTFITDECR(A[,1:3]).

□

We could have defined the notion of an on-line column insertion algorithm in another way:

DEFINITION 4.10. Let X be an algorithm that takes {0,1}-matrices as input and that returns a column-permutation of its input. X has property P if for every m×n {0,1}-matrix A and every k∈ IN with 1≦k≦n, X(A) and X(A[,1:k]) satisfy (4.2):

$$\left.\begin{array}{l}\text{let column } i\ (1\leqq i\leqq n) \text{ of A be column } p_i \text{ of X(A); let column } i\ (1\leqq i\leqq k) \\ \text{of A[,1:k] be column } q_i \text{ of X(A[,1:k]); for all i and j } (1\leqq i,j\leqq k) \text{ we} \\ \text{have} \\ \qquad p_i<p_j \text{ if and only if } q_i<q_j.\end{array}\right\} (4.2)$$

Lemma 4.3. *X is an on-line column insertion algorithm if and only if X has property* P.

Proof:

⇐: Suppose X has property P. Let A be an m×n {0,1}-matrix and k=n-1. Let $(p_i)_{1\leqq i\leqq n}$ and $(q_i)_{1\leqq i\leqq n-1}$ as in (4.2) and $p_n=y$ for some y. Then for all i with $q_i\leqq y-1$ we have $p_i=q_i$ and for all i with $q_i\geqq y$, $p_i=q_i+1$. Thus X(A) is an insertion matrix for A[,n] into X(A[,1:n-1]). This holds for every matrix A and therefore, X is an on-line column insertion algorithm.

⇒: Suppose X is an on-line column insertion algorithm. Let A be an m×n {0,1}-matrix and k∈ IN (1≦k≦n). Consider A[,1:h] with k≦h≦n. Let column i (1≦i≦h) of A[,1:h] be column $p_{h,i}$ of X(A[,1:h]).
X(A[,1:h+1]) is an insertion matrix for A[,h+1] into X(A[,1:h]), thus:

for all i,j $(1\leqq i,j\leqq h)$ $p_{h,i}<p_{h,j}$ if and only if $p_{h+1,i}<p_{h+1,j}$.

With induction it is easy to prove that

for all i,j $(1\leqq i,j\leqq k)$ $p_{k,i}<p_{k,j}$ if and only if $p_{n,i}<p_{n,j}$.

Hence, X has property P. □

Using this lemma, one can for every m×n {0,1}-matrix A determine X(A[,1:k]) from X(A) for every k≦n, since the relative ordering of columns of A [,1:k] as they appear in X(A) is inherited.

Corollary 4.1. *Let* $A=(\alpha_{ij})$ *be an* m×n {0,1}*-matrix and* X *an on-line column insertion algorithm. If for some* k≦n *and* i_0,j_0 $(1\leqq i_0\leqq m,\ 1\leqq j_0\leqq k)$ α_{i_0,j_0} *(considered as an element) of* A[,1:k] *is stored in* X(A[,1:k]), *then* α_{i_0,j_0} *(considered as an element) of* A *is stored in* X(A).

IV.2 APPROXIMATION WITH ON-LINE COLUMN INSERTION ALGORITHMS.

Now the question arises whether there is some on-line column insertion algorithm that is a polynomial time c-approximation algorithm for the CONSECUTIVE ONES MATRIX AUGMENTATION problem. In theorem 4.3 we will answer this question negatively. But corollary 4.3 states more. To obtain a negative answer, it will be sufficient to show that for each on-line column insertion algorithm X and each $c \geq 1$ and each $N \in \mathbb{N}$ there is an $m \times n$ matrix $A_{X,c,N}$ $(m,n \geq N)$ such that

$$\frac{\text{store}(X(A_{X,c,N}))}{\text{optstore}(A_{X,c,N})} > c$$

which proves that, in fact, no on-line column insertion algorithm can be a c-approximation (for some fixed c) at all. Theorem 4.3 states that $A_{X,c,N}$ can be chosen to be of arbitrary size (for fixed X and c) and corollary 4.3 states that for fixed c and N $A_{X,c,N}$ can be chosen such that the two following conditions are satisfied:

(i) the number of rows and columns of $A_{X,c,N}$ does not depend on the on-line column insertion algorithm X,

(ii) let X and X' be two on-line column insertion algorithms. Then $A_{X,c,N}$ and $A_{X',c,N}$ are equal possibly except for their last columns.

THEOREM 4.1. *For all* $N \in \mathbb{N}$ *and for all* $\varepsilon > 0$ $(\varepsilon \in \mathbb{R})$ *there is a clean square* {0,1}*-matrix* A *with* $n \geq N$ *rows such that for each column-permutation* B *of* A *we have:*

if $\frac{\text{store}(B)}{\text{optstore}(A)} \leq \frac{13}{7} - \varepsilon$ *then there is a* {0,1}*-sequence* u_B *of length* n *such that for every* $n \times (n+1)$ {0,1}*-matrix* C *that is an insertion matrix for* u_B *into* B $\frac{\text{store}(C)}{\text{optstore}(A \underline{\text{concat}}\ u_B)} > \frac{13}{7} - \varepsilon$.

Proof: Let $k \in \mathbb{N}$ with $3k \geq N$, $k > \frac{8}{7\varepsilon}$ and $\frac{13k-6}{7k+8} > \frac{13}{7} - \varepsilon$. (4.3)

Let A be the block diagonal matrix with k blocks $M_1, \ldots, M_k$ and $M_i = M_{I_1}$ $(1 \leq i \leq k)$ (for M_{I_1}, see section III.2). Let $K = (0,3,6,\ldots,3k)$ be the corresponding row- and column-partition of A.

With lemma 4.2: $\text{optstore}(A) = k \times \text{optstore}(M_{I_1}) = 7k$.

Let B be any column-permutation of A and let column i of A be column p_i of B. Then for all i $(1 \leq i \leq k)$:

$$\text{store}(\text{rowstrip}(B,i,K)) = 3 + |p_{3i}-p_{3i-1}| + |p_{3i-1}-p_{3i-2}| + |p_{3i-2}-p_{3i}|$$

$$= 3 + 2(\max\{p_{3i},p_{3i-1},p_{3i-2}\} - \min\{p_{3i},p_{3i-1},p_{3i-2}\}). \qquad (4.4)$$

Let $\text{pmax}(i) = \max\{p_{3i},p_{3i-1},p_{3i-2}\}$ $1 \leqq i \leqq k$ and
$\text{pmin}(i) = \min\{p_{3i},p_{3i-1},p_{3i-2}\}$ $1 \leqq i \leqq k$.

Let s_0 and s_1 be such that $\text{pmin}(s_0)=1$ and $\text{pmax}(s_1)=3k$. Thus the first and last column of B come from $\text{colstrip}(A,s_0,K)$ and $\text{colstrip}(A,s_1,K)$. Now we look at the position in B of the other columns of these two column-strips. We distinguish three cases:

Case 1: $s_0=s_1$.

Claim 1: $\dfrac{\text{store}(B)}{\text{optstore}(A)} > \dfrac{13}{7} - \varepsilon$.

Proof of claim 1: In this case the first and last column of B are from the same column-strip of A. Then (with (4.4)):

$$\text{store}(B) = \text{store}(\text{rowstrip}(B,s_0,K)) + \sum_{\substack{i=1 \\ i \neq s_0}}^{k} \text{store}(\text{rowstrip}(B,i,K))$$

$$\geqq 3 + 2(\text{pmax}(s_0)-\text{pmin}(s_0)) + 7(k-1) \geqq 13k-6$$

and with (4.3): $\dfrac{\text{store}(B)}{\text{optstore}(A)} \geqq \dfrac{13k-6}{7k} \geqq \dfrac{13}{7} - \dfrac{8}{7k} > \dfrac{13}{7} - \varepsilon$.

Case 2: $s_0 \neq s_1$, but $\text{pmax}(s_0) > \text{pmin}(s_1)$.

Claim 2: $\dfrac{\text{store}(B)}{\text{optstore}(A)} > \dfrac{13}{7} - \varepsilon$.

Proof of claim 2: With (4.4) we now have:

$$\text{store}(B) =$$

$$\text{store}(\text{rowstrip}(B,s_0,K)) + \text{store}(\text{rowstrip}(B,s_1,K)) + \sum_{\substack{i=1 \\ i \neq s_0,s_1}}^{k} \text{store}(\text{rowstrip}(B,i,K))$$

$$\geqq 6 + 2(\text{pmax}(s_0)-1) + 2(3k-\text{pmin}(s_1)) + (k-2)7 \geqq 13k-8,$$

and with (4.3): $\dfrac{\text{store}(B)}{\text{optstore}(A)} \geqq \dfrac{13}{7} - \dfrac{8}{7k} > \dfrac{13}{7} - \varepsilon$.

Case 3: $s_0 \neq s_1$ and $\text{pmax}(s_0) < \text{pmin}(s_1)$.

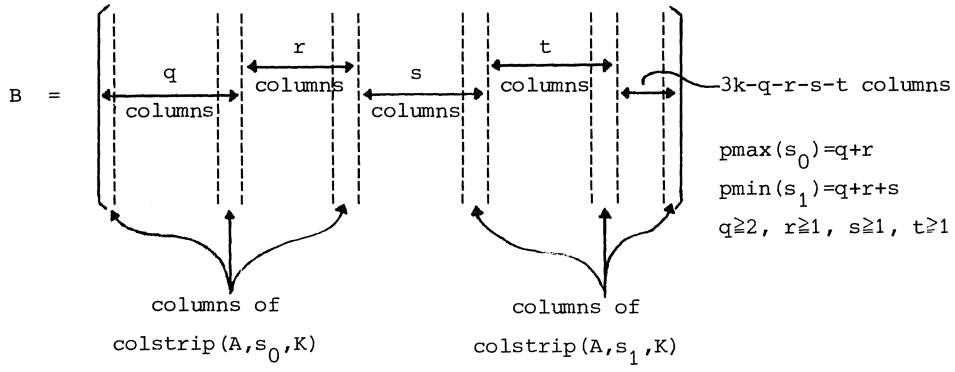
Claim 3: There is a $\{0,1\}$-sequence u_B of length 3k such that for each $\{0,1\}$-matrix C that is an insertion matrix for u_B in B

$$\frac{\text{store}(C)}{\text{optstore}(A \underline{\text{concat}}\ u_B)} > \frac{13}{7} - \varepsilon.$$

Proof of claim 3: As in the proof of proposition 4.3, define u_B to be the sequence that "connects" the blocks M_{s_0} and M_{s_1} in A:

$$\begin{aligned} u_B[i] &= 1 \text{ if } 3s_0-2 \leqq i \leqq 3s_0 \\ &= 1 \text{ if } 3s_1-2 \leqq i \leqq 3s_1 \\ &= 0 \text{ otherwise.} \end{aligned}$$

A concat u_B is a permuted block diagonal matrix with k-1 blocks. It is easy to see that optstore(A concat u_B) = 7k+8. Now we look at matrices that are insertion matrices for u_B in B. Let the columns of colstrip(A,s_0,K) and colstrip(A,s_1,K) be located in B as below



Let C be an insertion matrix for u_B in B, i.e., there is a $y \in \mathbb{N}$ $(0 \leqq y \leqq 3k)$ such that

$$\begin{aligned} C[\ ,1:y] &= B[\ ,1:y] \\ C[\ ,y+1] &= u_B \\ C[\ ,y+2:3k+1] &= B[\ ,y+1:3k]. \end{aligned}$$

Now we distinguish 7 cases: y=0, $0<y<q$, $q \leqq y<q+r$, $q+r \leqq y<q+r+s$, $q+r+s \leqq y<q+r+s+t$, $q+r+s+t \leqq y<3k$ and y=3k. In each of these 7 cases we have: store(C) $\geqq$ 13k-6. And with (4.3):

$$\frac{\text{store}(C)}{\text{optstore}(A \underline{\text{concat}}\ u_B)} \geqq \frac{13k-6}{7k+8} > \frac{13}{7} - \varepsilon.$$

Thus, claim 1, claim 2 and claim 3 have been proved and hence, theorem 4.1 has been proved.

□

In the above proof we used a block diagonal matrix A with k 3×3 blocks. We can prove more by using matrices A that have k m×m blocks. The number of elements in a block is bounded from above by m^2 and in a row-strip by km^2. In the proof of the next theorem we will use a block diagonal matrix A with k m×m blocks that all have an optstore value bounded by 3m, but which is such that for each column-permutation B of A it is possible that many elements of a row-strip of B will be stored if one extra column is inserted in B.

Theorem 4.2. *For all* $N \in \mathbb{N}$ *and for all* $c \geq 1$ $(c \geq \mathbb{R})$ *there is a clean square matrix* A *with* $n \geq N$ *rows such that for each column-permutation* B *of* A *we have:*

if $\frac{\text{store}(B)}{\text{optstore}(A)} \leq c$ *then there is a* $\{0,1\}$*-sequence* u_B *of length* n *such that for each matrix* C *that is an insertion matrix for* u_B *into* B, $\frac{\text{store}(C)}{\text{optstore}(A \underline{\text{concat}}\ u_B)} > c$.

Proof: Let $N \in \mathbb{N}$ and $c \in \mathbb{R}$ $(c \geq 1)$. Let $f: \mathbb{N} \to \mathbb{N}$ be any function such that

(i) $\lim_{k \to \infty} f(k) = \infty$

(ii) there is a $c' > 0$ and a $K' \in \mathbb{N}$ such that for all $k \geq K'$ $\frac{f(k)}{k} < \frac{1}{3} - c'$.

For example: $f(k) = \lceil \frac{k}{4} \rceil$, $f(k) = \lceil {}^2\log k \rceil$, etc.

Choose $k' \in \mathbb{N}$ $(k' \geq 3)$ such that for all $k \geq k'$ we have

$$f(k) > c, \quad \frac{1}{3} + \frac{2}{3} f(k) > c \quad \text{and} \quad \frac{k - 3f(k)}{3k}(1 + 2f(k)) > c.$$

Such a k' exists. Let $k \geq k'$.

Choose $m' \in \mathbb{N}$ $(m' \geq 3)$ such that for all $m \geq m'$ we have

$$mk \geq N \quad \text{and} \quad \frac{k(3m-2) + m + m^2 f(k)}{k(3m-2) + m^2 - m + 2} > c.$$

Let $m \geq m'$. Thus:

$$\left.\begin{array}{l} mk \geq N, \quad m \geq 3, \quad k \geq 3, \quad \dfrac{k(3m-2) + m + m^2 f(k)}{k(3m-2) + m^2 - m + 2} > c, \\ \dfrac{1 + 2f(k)}{3} > c \quad \text{and} \quad \left(\dfrac{1}{3} - \dfrac{f(k)}{k}\right)(2f(k) + 1) > c. \end{array}\right\} \quad (4.5)$$

Let A be a block diagonal matrix with k blocks $M_1, \ldots, M_k$ and $M_i = M_{I_{m-2}}$ $(1 \leq i \leq k)$. Let $K = (0, m, 2m, \ldots, mk)$ be the corresponding row- and column-partition. Then we have: (4.6

$$\text{optstore}(M_i) = 3m-2 \quad (1 \leq i \leq k), \qquad \text{optstore}(A) = (3m-2)k. \quad (4.7$$

Let B be any column-permutation of A with column j of A column p_j of B $(1 \leq j \leq mk)$.

Let $pmin(i) = min\{p_j : (i-1)m<j\leq im\}$, $(1\leq i\leq k)$ and

$$pmax(i) = max\{p_j : (i-1)m<j\leq im\},\ (1\leq i\leq k). \qquad (4.8)$$

Then we have:

$$store(rowstrip(B,i,K)) \geq m+2(pmax(i)-pmin(i)) \text{ for all } 1\leq i\leq k. \qquad (4.9)$$

As in the preceding proof we are interested in the values of $pmin(i)-pmax(j)$.

Let $q_0 = min\{pmax(i) : 1\leq i\leq k\}$, $q_1 = max\{pmin(i) : 1\leq i\leq k\}$.

There are three cases:

Case 1: $q_1 \leq q_0-m.f(k)$.

Claim 1: With m and k satisfying (4.5) and $q_1 \leq q_0-m.f(k)$ we have

$$\frac{store(B)}{optstore(A)} > c.$$

Proof of claim 1: $q_1 \leq q_0-m.f(k)$. Thus:

for all i $pmax(i) \geq pmin(i)+m.f(k)$ $(1\leq i\leq k)$.

Using (4.9) this results in $store(rowstrip(B,i,K)) \geq m+2m.f(k)$ so that (with (4.5) and (4.7)):

$$\frac{store(B)}{optstore(A)} \geq \frac{mk+2mk.f(k)}{(3m-2)k} \geq \frac{m+2m.f(k)}{3m} = \frac{1+2f(k)}{3} > c.$$

Hence claim 1 is proved and case 1 satisfies the theorem.

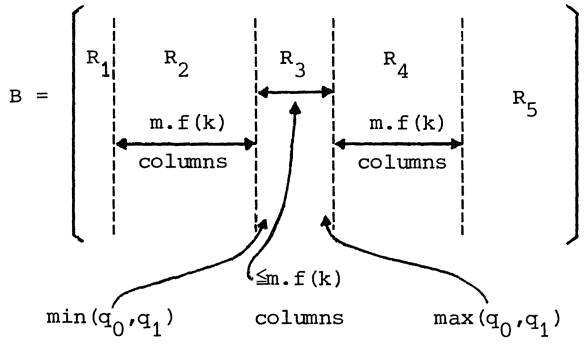
Case 2: $|q_1-q_0| < m.f(k)$.

Claim 2: With m and k satisfying (4.5) and $|q_1-q_0| < m.f(k)$ we have

$$\frac{store(B)}{optstore(A)} > c.$$

Proof of claim 2: First assume $m.f(k) < min(q_0,q_1) < max(q_0,q_1) < mk-m.f(k)$.

We divide B in five parts:



$$R_1 = B[\ ,1 : \min(q_0,q_1)-m.f(k)-1],$$
$$R_2 = B[\ ,\min(q_0,q_1)-m.f(k) : \min(q_0,q_1)-1],$$
$$R_3 = B[\ ,\min(q_0,q_1) : \max(q_0,q_1)],$$
$$R_4 = B[\ ,\max(q_0,q_1)+1 : \max(q_0,q_1)+m.f(k)],$$
$$R_5 = B[\ ,\max(q_0,q_1)+m.f(k)+1 : mk].$$

R_2, R_3 and R_4 together contain at most $3m.f(k)$ columns. B contains mk columns so that R_1 and R_5 together contain at least $mk-3m.f(k)$ columns. Each column-strip of A has m columns; thus there are at least $\frac{mk-3m.f(k)}{m}$ different column-strips of A that have a column in R_1 or R_5. More formally:

Let $I_{MIN} = \{i : pmin(i) < \min(q_0,q_1)-m.f(k)\}$ and

$I_{MAX} = \{i : pmax(i) > \max(q_0,q_1)+m.f(k)\}$.

Then: $|I_{MIN} \cup I_{MAX}| \geq k-3f(k)$. For each $i \in I_{MIN} \cup I_{MAX}$: $pmax(i)-pmin(i) \geq m.f(k)$.

With (4.9) this gives the result:

for all $i \in I_{MIN} \cup I_{MAX}$: $store(rowstrip(B,i,K)) \geq m+2m.f(k)$.

Hence: $store(B) \geq \sum_{i \in I_{MIN} \cup I_{MAX}} store(rowstrip(B,i,K))$

$$\geq |I_{MIN} \cup I_{MAX}| . (m+2m.f(k)) \geq (k-3f(k)).(m+2m.f(k)).$$

With (4.5): $\frac{store(B)}{optstore(A)} \geq \frac{(k-3f(k))(m+2m.f(k))}{(3m-2)k} > \frac{(k-3f(k))(1+2f(k))}{3k} > c.$

Now we have proved claim 2 for the general case that $m.f(k) < \min(q_0,q_1) < \max(q_0,q_1) < mk-m.f(k)$.

Next assume $\min(q_0,q_1) \leq m.f(k)$ or $\max(q_0,q_1) \geq mk-m.f(k)$. The two special cases ($\min(q_0,q_1) \leq m.f(k)$ and $\max(q_0,q_1) \geq mk-m.f(k)$) are similar; thus we will only deal with $\min(q_0,q_1) \leq m.f(k)$. Now we divide B in 4 parts:

$$R_2 = B[\ ,1 : \min(q_0,q_1)-1],$$
$$R_3 = B[\ ,\min(q_0,q_1) : \max(q_0,q_1)],$$
$$R_4 = B[\ ,\max(q_0,q_1)+1 : \max(q_0,q_1)+m.f(k)] \text{ and}$$
$$R_5 = B[\ ,\max(q_0,q_1)+m.f(k)+1 : mk].$$

R_2, R_3 and R_4 together contain at most $3m.f(k)$ columns, and therefore R_5 contains at least $mk-3m.f(k)$ columns and there are at least $k-3f(k)$ column-strips of A with a column in R_5. Let

$I_{MAX} = \{i : pmax(i) > \max(q_0,q_1)+m.f(k)\}$.

Then $|I_{MAX}| \geq k-3f(k)$. For each $i \in I_{MAX}$: $pmax(i)-pmin(i) > m.f(k)$. With (4.9):

$$\text{store}(B) \geq \sum_{i \in I_{MAX}} \text{store}(\text{rowstrip}(B,i,K)) \geq (k-3f(k)).(m+2m.f(k))$$

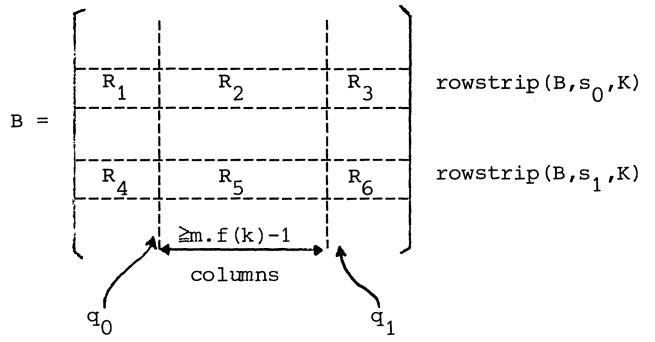
and with (4.5): $\dfrac{\text{store}(B)}{\text{optstore}(A)} > c.$

Now claim 2 is proven, and hence case 2 satisfies the theorem.

Case 3: $q_1 \geq q_0+m.f(k)$.

Claim 3: With m and k satisfying (4.5) and $q_1 \geq q_0+m.f(k)$ there is a $\{0,1\}$-sequence u_B of length mk such that for each matrix C that is an insertion matrix for u_B into B, $\dfrac{\text{store}(C)}{\text{optstore}(A \text{ concat } u_B)} > c$.

Proof of claim 3: Let $q_1-q_0 \geq m.f(k)$. With the definition of q_0 and q_1 there are s_0 and s_1 $(1 \leq s_0 \leq k,\ 1 \leq s_1 \leq k)$ such that $q_0=\text{pmax}(s_0)$ and $q_1=\text{pmin}(s_1)$. For s_0 and s_1 we have: $s_0 \neq s_1$ and $\text{pmin}(s_1)-\text{pmax}(s_0) \geq m.f(k)$. Let us divide B as follows:



(4.10)

Because of (4.8) and the choice of s_0 and s_1, R_2, R_3, R_4 and R_5 are all zero matrices. Let u_B be a $\{0,1\}$-sequence of length mk defined by:

$$\left.\begin{array}{l} u_B[j]=1 \quad \text{if } (s_0-1)m<j \leq s_0.m \\ \quad\ =1 \quad \text{if } (s_1-1)m<j \leq s_1.m \\ \quad\ =0 \quad \text{otherwise.} \end{array}\right\} \quad (4.11)$$

A concat u_B is a permuted block diagonal matrix with k-1 blocks. (4.12)

It is easy to see that

$$\text{optstore}(A \text{ concat } u_B) \leq \sum_{\substack{i=1 \\ i \neq s_0,s_1}}^{k} \text{optstore}(\text{rowstrip}(A,i,K)) + 2\left(m+1+\sum_{i=3}^{m+1} i\right)$$

$$= (k-2)(3m-2) + m^2+5m-2. \quad (4.13)$$

Let C be an insertion matrix for u_B into B. Thus, there is a $y \in \mathbb{N}$ $(0 \leq y \leq mk)$ such that

$$\begin{aligned} B[\ ,1:y] &= C[\ ,1:y] \\ u_B &= C[\ ,y+1] \\ B[\ ,y+1:mk] &= C[\ ,y+2:mk+1]. \end{aligned}$$

If $y \leq q_0$ then all zero elements of R_5 are stored in C though they are not stored in B. R_5 contains at least $m^2.f(k)-m$ zero elements.
If $y \geq q_1-1$ then all zero elements of R_2 are stored in C though they are not stored in B. R_2 contains at least $m^2.f(k)-m$ zero elements.
If $q_0 < y \leq q_1$, then the same number of zero elements will be stored in C, but now they are distributed over R_2 and R_5. None of these zero elements are stored in B.
Thus:

$$\begin{aligned} \text{store}(C) &\geq \text{optstore}(B) + \text{ones}(u_B) + |\{\text{elements of } R_2 \text{ and } R_5 \text{ stored in } C\}| \\ &\geq k(3m-2)+2m+m^2.f(k)-m = k(3m-2)+m^2.f(k)+m. \end{aligned}$$

With (4.13) and (4.5), this gives:

$$\frac{\text{store}(C)}{\text{optstore}(A \underline{\text{concat}}\ u_B)} \geq \frac{k(3m-2)+m+m^2.f(k)}{k(3m-2)+m^2-m+2} > c$$

This ends the proof of claim 3 and the proof of theorem 4.2.

□

As a direct consequence we have:

THEOREM 4.3. *Let* X *be an algorithm that takes* {0,1}*-matrices as input and that returns a column-permutation of its input. Suppose there is a* $c \in \mathbb{R}$ *such that* X *is a c-approximation algorithm. Then* X *is not an on-line column insertion algorithm.*

However, theorem 4.2 states more about c-approximation algorithms:

Corollary 4.2. *Let* X *satisfy the conditions of theorem 4.3. Then for all* $N \in \mathbb{N}$ *and for every algorithm* Y *that takes* {0,1}*-matrices as input and that returns a column-permutation of its input, there is an* m×n *matrix* B *with* $m,n \geq N$ *such that* X(B) *is not an insertion matrix for* B[,n] *into* Y(B[,1:n-1]).

Corollary 4.3. *Let* $N \in \mathbb{N}$ *and* $c \in \mathbb{R}$ $(c \geq 1)$. *Then there are* $m,n \geq N$ *and an* m×n *matrix* A *such that for every two on-line column insertion algorithms* X *and* X' *there are* m×(n+1) *matrices* B *and* C *with*

$$\frac{\text{store}(X(B))}{\text{optstore}(B)} > c, \quad \frac{\text{store}(X'(C))}{\text{optstore}(C)} > c \quad \textit{and} \quad B[\,,1{:}n] = C[\,,1{:}n] = A.$$

Theorem 4.3 is the special case of corollary 4.2 with X substituted for Y. Observe that corollary 4.2 holds even if Y finds the optimum column-permutation of its input. With this corollary we have described a large class of approximation algorithms that does not contain any c-approximation algorithm for any $c \in \mathbb{R}$.

IV.3 EXTENSIONS TO OTHER CLASSES OF APPROXIMATION ALGORITHMS.

In the previous section we saw that many approximation algorithms can provide bad approximations to the optimal storage (in a *rowmat* data structure) of a column-permutation of a matrix. In particular this bad result is obtained for some permuted block diagonal matrices. Knowing this, one can try to design more sophisticated approximation algorithms. In this section we will extend theorem 4.2 and, consequently, theorem 4.3 and the corollaries 4.2 and 4.3, and will describe larger classes of approximation algorithms that still do not contain a c-approximation algorithm for any $c \in \mathbb{R}$.

IV.3.1 Preprocessing to block diagonal form.

The class of matrices used in the proof of theorem 4.2 merely consists of block diagonal and permuted block diagonal matrices (see (4.6) and (4.12)). For each block diagonal matrix there is an optimum column-permutation that is block diagonal too. Moreover, as we have seen before (algorithm 4.1), there is an on-line column insertion algorithm that preserves this block diagonal structure if applied to a block diagonal matrix. Thus, it seems reasonable to permute the rows and columns of a permuted block diagonal matrix to block diagonal form before applying any on-line column insertion algorithm. Theorem 4.2 can be proven using matrices $A_{m,k}$ which are not block diagonal with $p \geq 2$ blocks. $A_{m,k}$ can be chosen in such a way that it will become block diagonal by deletion of two columns. To prove negative results concerning more sophisticated algorithms, we introduce the concept of a separator of a matrix.

DEFINITION 4.11. Let $p \in \mathbb{N}$ and A an $m \times n$ $\{0,1\}$-matrix with $p \leq n$.

(i) A set of p columns of A is a separator of A if the matrix A with the separator columns deleted, is a permuted block diagonal matrix with at least two blocks.

(ii) Let S be a smallest separator of A. A is in connected order if the deletion of the columns of S from A results in a block diagonal matrix.

The next theorem states that if an on-line column insertion algorithm is combined with a preprocessing algorithm that permutes matrices in connected order, this will not give rise to c-approximation algorithms for any $c \in \mathbb{R}$.

THEOREM 4.4. *Let* $N \in \mathbb{N}$, $\varepsilon, c \in \mathbb{R}$ $(c \geq 1, \varepsilon > 0)$. *Then there is an* $n \in \mathbb{N}$ *and a square* $\{0,1\}$*-matrix* A *such that for each column-permutation* B *of* A*:*

if $\frac{\text{store}(B)}{\text{optstore}(A)} \leq c$ *then there is a* $\{0,1\}$*-sequence* u_B *of length* $n^{1+\varepsilon}$ *such that* $\frac{\text{store}(C)}{\text{optstore}(A \text{ concat } u_B)} > c$ *for any matrix* C *that is an insertion matrix for* u_B *into* B. A *and* A concat u_B *satisfy the following conditions:*

(i) they are clean,

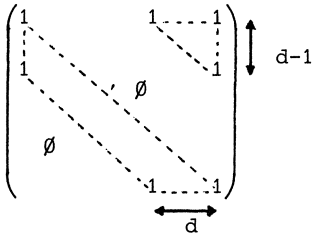
(ii) they have $n^{1+\varepsilon} \geq N$ *rows,*

(iii) they are in connected order,

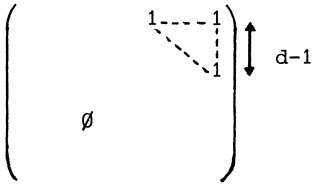
(iv) they do not contain a separator of size $< n^{1-\varepsilon}$.

Proof: Because the proof of this theorem is similar to the proof of theorem 4.2 and uses the same kind of arguments, we will only give an outline.

Let $f: \mathbb{N} \to \mathbb{N}$ be a function with $f(n) = \Omega(n)$ and there are $\delta > 0$ and $K \in \mathbb{N}$ such that for all $k \geq K$ $\frac{3f(k)-1}{3k} - \frac{1}{3} > \delta$.

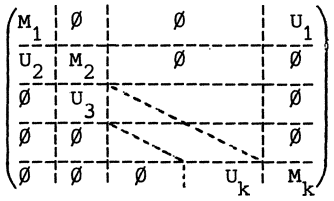
Let $M_{n,d}$ = 

an $n \times n$ $\{0,1\}$-matrix in which each row and column has exactly d non-zero elements,

$U_{n,d}$ = 

an $n \times n$ $\{0,1\}$-matrix,

and $A_{n,k,d}$ =



with $M_i = M_{n,d}$ $(1 \leq i \leq k)$
$U_i = U_{n,d}$ $(1 \leq i \leq k)$.

Then we have:

(i) The smallest separator of $A_{n,k,d}$ contains 2d-2 columns,

(ii) $nd \leq \text{optstore}(M_i) \leq (d-1)n+(n-d+1)d$,

(iii) $kd(\frac{d-1}{2}+n) \leq \text{optstore}(A_{n,k,d}) \leq k\{3nd-\frac{d^2}{2}+\frac{d}{2}-2n\}-n(d-1)-\frac{d(d-1)}{2}$.

Let B be a column-permutation of $A_{n,k,d}$. pmax(i) and pmin(i), $1 \leq i \leq k$, are defined as in (4.8). Without proof we state:

Claim: $\text{store}(\text{rowstrip}(B,i,K)) \geq d(\text{pmax}(i)-\text{pmin}(i)+1)$ for all i $(1 \leq i \leq k)$.

Let $q_0 = \max\{\text{pmin}(i):2 \leq i \leq k\}$, $q_1 = \min\{\text{pmax}(i):2 \leq i \leq k\}$.

The rest of the proof is quite the same as the proof of theorem 4.2, except that it requires more calculations. We only show where the connected order plays a role. $A_{n,k,d}$ does not have a separator of size less than 2d-2. Deletion of the last d-1 columns of the first and last column-strip of $A_{n,k,d}$, results in a block diagonal matrix with two blocks. Moreover, the sequence u_B that is defined will not have non-zero elements in the first n positions, because pmin(1) and pmax(1) do not play any role in the values of q_0 and q_1. Therefore, deletion of the same (2d-2) columns of $A_{n,k,d}$ concat u_B once again results in a block diagonal matrix with two blocks.

□

Remark 4.1. Theorem 4.4 even holds if we put the additional restriction on A (and A concat u_B) that the columns of a smallest separator are consecutive in A (and A concat u_B).

Thus, preprocessing a matrix into connected order does not help us to find a c-approximation if we use an on-line column insertion algorithm. Of course, it is impossible to prove negative results for all combinations of preprocessing and on-line column insertion algorithms: one of these combinations gives for each matrix the optimum column-permutation. But each preprocessing algorithm that does not change the column number of u_B if applied to A concat u_B (see (4.11)), does not improve the performance of any on-line column in-

sertion algorithm.

IV.3.2 Mixers and on-line column insertion algorithms.

Another kind of more sophisticated approximation algorithms can be obtained by incorporating a "small mixer" in an on-line column insertion algorithm. On-line column insertion algorithms often can be formulated in such a way that the columns are processed one by one. The columns that are processed do not loose their relative order with the insertion of a next column. If we incorporate a "small mixer", we allow that the columns already processed are permuted slightly just before column i will be inserted.

DEFINITION 4.12. Let A be an m×n {0,1}-matrix, u a {0,1}-sequence of length m and $s: \mathbb{N} \to \mathbb{N}$ a function with $s(k) \leq k$ for all $k \in \mathbb{N}$. Let B be a column-permutation of A concat u and column i of A concat u is column p_i of B $(1 \leq i \leq n+1)$.

B is an s-mix1 insertion matrix for u in A if there are n-s(n) numbers $i_1 < i_2 < \ldots < i_{n-s(n)} < n$ such that

$$i_j < i_k \text{ if and only if } p_{i_j} < p_{i_k} \quad (1 \leq j \leq n-s(n),\ 1 \leq k \leq n-s(n)). \tag{4.14}$$

B is an s-mix2 insertion matrix for u in A if for all j $(1 \leq j \leq n)$ we have:

if $p_j < p_{n+1}$ then $|p_j - j| \leq s(n)$ and if $p_j > p_{n+1}$ then $|p_j - 1 - j| \leq s(n)$.

DEFINITION 4.13. Let X be an algorithm that takes {0,1}-matrices as input and that returns a column-permutation of its input. Let $s: \mathbb{N} \to \mathbb{N}$ be a function with $s(k) \leq k$ for all $k \in \mathbb{N}$. X is an on-line column s-mix1 (resp. s-mix2) insertion algorithm if for each m×n {0,1}-matrix A, X(A) is an s-mix1 (resp. s-mix2) insertion matrix for A[,n] into X(A[,1:n-1]).

In an on-line column s-mix1 insertion algorithm at most s(n-1) columns may be deleted from X(A[,1:n-1]) and inserted somewhere else in this matrix before inserting the n^{th} column. In particular, an s-mix1 insertion algorithm allows that the first column of X(A[,1:n-1]) will be the last column of X(A), in case $s(n-1) \geq 1$. The way these s(n-1) columns are mixed and once again inserted, will have to depend on the n^{th} column, otherwise corollary 4.2 states that we do not have a better approximation algorithm.

In an on-line column s-mix2 insertion algorithm all columns of X(A[,1:n-1]) are allowed to be permuted, but the new column number will differ at most

s(n-1) from the old column number. With $s(n-1)<n-2$, this means that the first column of X(A[,1:n-1]) will never be the last column of X(A). For the same reason as above we are only interested in mix2 algorithms that depend on column n of A.
Unfortunately, if s(n) is not large enough, no on-line column s-mix1 or s-mix2 insertion algorithm is a c-approximation algorithm:

THEOREM 4.5. *For all* $N\in\mathbb{N}$, $c\in\mathbb{R}$ *and* $\varepsilon\in\mathbb{R}$ $(\varepsilon>0)$ *there is an* $n\in\mathbb{N}$ *and a clean square* {0,1}*-matrix* A *with* $n^{1+\varepsilon}\geq N$ *rows such that for each column-permutation* B *of* A *we have:*
if $\frac{\text{store(B)}}{\text{optstore(A)}} \leq c$ *then there is a* {0,1}*-sequence* u_B *of length* $n^{1+\varepsilon}$ *such that for every function* $s:\mathbb{N}\to\mathbb{N}$ *with* $s(k)\leq k^{1-\varepsilon}$ $(k\geq 2)$,
$\frac{\text{store(C)}}{\text{optstore(A concat } u_B)} > c$ *for every matrix* C *that is an* s-mix1 *insertion matrix for* u_B *in* B.

THEOREM 4.6. *Theorem 4.5 with* "mix1" *replaced by* "mix2".

We will only sketch the proofs of these theorems, because they are similar to the proof of theorem 4.2.

Sketch of proof of theorem 4.5:
Use the matrix A given in (4.6), with M_i an n×n matrix. Let B be a column-permutation of A. Let $k=n^{\varepsilon}$.
We only have to look at the case $\frac{\text{store(B)}}{\text{optstore(A)}} \leq c$ in which we have to provide u_B. Define u_B as in (4.11). Let B be divided as in (4.10).
Because $s(n^{1+\varepsilon})<n$, there is at least one column of each column-strip of A of which the column number is one of the $i_1,\ldots,i_{n^{1+\varepsilon}-s(n^{1+\varepsilon})}$ of (4.14). Let us delete all columns p_i of B of which i does not occur in $i_1,\ldots,i_{n^{1+\varepsilon}-s(n^{1+\varepsilon})}$, resulting in a matrix B'. Then there surely is a column of colstrip(A,s_0,K) in the left part of B' and one of colstrip(A,s_1,K) in the right part of B'. Inserting u_B in B' causes at least $n(n.f(k)-s(n^{1+\varepsilon})-1)$ zero elements to be stored. Inserting the deleted columns can make things only worse. With suitable f and n large enough, ratio (4.1) is violated.

□

Sketch of proof of theorem 4.6:
Let A, B and u_B as above in the proof of theorem 4.5. After the mixing of B has been done (resulting in B'), column $pmax(s_0)$ of B is at most $s(n^{1+\varepsilon})$ positions to the right and column $pmin(s_1)$ of B $s(n^{1+\varepsilon})$ to the left. Thus inserting u_B causes at least $n(n.f(k)-2s(n^{1+\varepsilon})-1)$ zero elements to be stored. With suitable f and n large enough, ratio (4.1) is violated.

□

Corollary 4.4. *Let there be an* $N \in \mathbb{N}$, *a* $c \in \mathbb{R}$ *and a c-approximation algorithm* X *for all matrices with more than* N *rows and* N *columns. Then* X *is neither an on-line column* s-mix1 *nor an on-line column* s-mix2 *insertion algorithm for any* $\varepsilon \in \mathbb{R}$ $(\varepsilon>0)$ *and* $s: \mathbb{N} \to \mathbb{N}$ *with* $s(k) \leq k^{1-\varepsilon}$.

Corollary 4.5. *Let* X *satisfy the conditions of corollary 4.4. Then for every algorithm* Y *that takes* {0,1}*-matrices as input and that returns a column-permutation of its input, there is an* m×n *matrix* B *with* $m \geq N$ *and* $n \geq N$ *such that* X(B) *is neither an* s-mix1 *nor an* s-mix2 *insertion matrix for* B[,n] *into* Y(B[,1:n-1]) *for any* $\varepsilon \in \mathbb{R}$ $(\varepsilon>0)$ *and* $s: \mathbb{N} \to \mathbb{N}$ *with* $s(k) \leq k^{1-\varepsilon}$.

Conclusion. In this chapter we have given negative results concerning the existence of approximation algorithms of the CONSECUTIVE ONES MATRIX AUGMENTATION problem. No algorithm of a wide class of algorithms, containing e.g. all on-line column insertion algorithms can guarantee a c-approximation for any $c \in \mathbb{R}$ and for all matrices. Even the addition of a small mixer does not help.
In all these algorithms we looked at a matrix as a set of columns. Further research can be done on approximation algorithms that are row-oriented. Needless to say that research in the direction of probabilistic algorithms (see the introduction of this chapter) will perhaps give positive results, even for on-line column insertion algorithms. Thus, it remains open whether the *rowmat* data structure could be used for arbitrary sparse matrices. This chapter has only provided one more indication that this data structure may not be suited for this purpose if full storage optimality is the ultimate goal.

The results presented in this chapter have also appeared in [78].

CHAPTER V

DATA STRUCTURES FOR SPARSE MATRICES

In this chapter we will design TORRIX-SPARSE, a system of primitive operations to manipulate sparse matrices. Implicit in this venture is the design of one or more data structures for sparse matrices. Unfortunately most commonly used data structures for sparse matrices are rather simple and not geared to the sparsity patterns and desired operations mentioned in chapter II. In this chapter we will review a number of data structures for sparse matrices and propose a new one. The new data structure is obtained by an extension of the hypermatrix approach (cf. [26]), in which a sparse matrix is considered a block and each block is either stored as a full matrix or as a zero matrix, or is partitioned into smaller blocks (that may be partitioned also). New features of the data structure presented in this chapter are:

(i) sparseness of a non-zero block can be obtained in three ways:
  (1) a block is represented by one (small) full matrix containing fewer elements than the block itself,
  (2) if the non-zero elements of the block are situated along a (small) number of diagonals, then only these diagonals need be stored,
  (3) a block is partitioned into smaller blocks.

(ii) a slicing mechanism similar to that of ALGOL 68 and TORRIX-BASIS (cf. [77] and [75]) is implemented in TORRIX-SPARSE. It prescribes which side effects must occur if operations are applied to a sparse matrix or slices of it. Obviously this puts some demands on the data structure used in TORRIX-SPARSE.

(iii) the user can specify which blocks of a matrix are structurally equal such that only one of them will be actually stored. This may save storage as well as computation time.

This chapter consists of five sections. In section V.1 we will state the design objectives of a TORRIX-SPARSE system. In section V.2 we will investi-

gate briefly several common data structures for sparse matrices and see to what extent they satisfy the design objectives. In V.3 we will propose and explain the general features of the new data structure and in V.4 we will explain the slicing mechanism of TORRIX-SPARSE. The large section V.5 is a technical explanation of the TORRIX-SPARSE system and contains e.g. the mode declarations for the implemented data structure. In an appendix at the end of this chapter a table of operations will be given of all operations of TORRIX-SPARSE used in this chapter. A short description of these operations is part of this table. Because TORRIX-SPARSE is an extension of TORRIX-BASIS, we will freely use terms of ALGOL 68 and TORRIX-BASIS (cf. chapter I, or [77] and [75]).

V.1 DESIGN OBJECTIVES.

In stating the design objectives of TORRIX-SPARSE we will try to make a difference between design objectives for the data structure and design objectives for the software system. Possibly the latter objectives cannot be fully met with each data structure satisfying the first objectives. Data structure design objectives only deal with sparsity patterns to be implemented and operations to be performed and their efficiency, whereas software system design objectives deal e.g. with the provision of a slicing mechanism, the occurrence of side effects and the need of a garbage collector.
Each software system for sparse matrices contains one or more data structures for a sparse matrix and a number of operations. The user of such a system can access the data structures only by means of these operations. There may be operations hidden from the user, which only the system manager can apply. Often these operations are hidden from the user because they are not fully safe, i.e., there might be a syntactically correct application returning a wrong answer. However, if applied in the right way, these operations may do their task in a more efficient way than a syntactically safe equivalent. In proposing the software system this difference between the instructions available to the system manager and the user is important, especially with regard to the control of the side effects.

V.1.1 Design objectives for the data structure for sparse matrices.

Each data structure for a (sparse) matrix consists of storage for values of matrix elements and storage for pointers, bounds, indices, etc. (administration) (cf. [30]). The total amount of storage needed should be as low as possible without preventing an efficient use of the data structure. The data structure *mat* for full matrices contains very litle administration. Data structures for sparse matrices are designed to decrease the amount of storage for values of matrix elements considerably at the expense of a (small) increase in administration. There are data structures in which only the non-zero elements are stored and for which the ratio of the number of stored elements and the amount of administration is independent of the number and the distribution of the non-zero elements in a matrix (see V.2.2). In chapter II we saw that non-zero elements often occur in (combinations of) blocks and (block) diagonals. From this we deduce the first design objective:

Objective 5.1. Design a data structure for sparse matrices that allows the storage of every sparse matrix, such that advantage is taken from the block patterns and the (block) diagonal patterns of these matrices (i.e., the administration should be rather small, if these patterns occur in the matrix).

As TORRIX-SPARSE is to be an extension of TORRIX-BASIS, a good interface between TORRIX-BASIS and TORRIX-SPARSE is necessary. Thus, it may be needed that a matrix stored in a *mat* must be considered as a sparse matrix. If design objective 5.1 can be met, the administration to be added to a *mat* to view it as a sparse matrix, may be very modest and of a size that is hopefully independent of the size of the *mat*.

In chapter II we mentioned the following commonly occurring properties of sparse matrices:

(i) often matrices are symmetric or have a symmetric sparsity pattern, (5.1)
(ii) several blocks of a matrix may be equal, (5.2)

and we identified the following important operations on sparse matrices:

(i) matrix-vector product, (5.3)
(ii) matrix product of two blocks of a sparse matrix, (5.4)
(iii) changing the value of zero and non-zero elements, (5.5)
(iv) row- and column-permutations, (5.6)
(v) cyclic row- and column-permutations, (5.7)

(vi) adding a part of a row (or column) multiplied with a scalar to the corresponding part in another row (or column), (5.8)

(vii) multiplying a matrix with a Givens transformation, (5.9)

(viii) computing the number of non-zero elements in a part of a row or column, (5.10)

(ix) determining the in absolute value largest element (with its indices) in a part of a row, of a column or in a submatrix, (5.11)

(x) computing the inner product of a part of a row (or column) with a vector or a part of a column (or row), (5.12)

(xi) insertion of rows and columns, (5.13)

(xii) deletion of a block. (5.14)

Objective 5.2. Design a data structure for sparse matrices such that properties (5.1) and (5.2) can be taken into account and that operations (5.3) ... (5.14) are implemented easily and efficiently.

V.1.2 Design objectives for a TORRIX-SPARSE software system.

TORRIX-SPARSE is meant to be a software system containing a number of primitive operations involved in the manipulation of sparse matrices. Thus, the user will write programs using the primitives defined in TORRIX-SPARSE. This leads to the following objective:

Objective 5.3. Each algorithm for sparse matrices can be implemented by the user.

Each algorithm acting on a sparse matrix can be formulated in such a way that the access to the matrix is performed as a number of accesses to individual matrix elements. Thus, every algorithm can be implemented provided the system contains operators to change the value of an arbitrary matrix element and to return to the user the value of an arbitrary matrix element. Of course we cannot prevent that an algorithm implemented by the user and applying only operations on single matrix elements, runs rather slowly. However, the use of some operators defined in TORRIX-SPARSE (for example slicing operators) could reduce computation time in such a case.

In chapter II we have seen that many sparse matrices in one program have the same sparsity pattern. This leads to:

Objective 5.4. Design a software system TORRIX-SPARSE for sparse matrices in which it is taken into account that many sparse matrices in one user program have the same sparsity pattern.

In our view TORRIX-SPARSE is an extension of TORRIX-BASIS. Therefore, we have:

Objective 5.5. Design a software system TORRIX-SPARSE for sparse matrices such that the user can freely apply operators and use data structures from TORRIX-BASIS and TORRIX-SPARSE in one program. Moreover, he can write programs in TORRIX-SPARSE in the same style as he was used to with TORRIX-BASIS.

The latter part of this objective is rather vague and we shall indicate what we consider essential aspects of TORRIX-BASIS programming style in four refinements of objective 5.5.

Objective 5.5.1. The software system should make no assumptions about the field of scalars used and (thus) provide a mode *scal* for the user to specify, as in TORRIX-BASIS.

ALGOL 68 contains a slicing mechanism with which the user can select rows, columns, submatrices and elements of a matrix and subvectors and elements of a vector. In addition to these selections, TORRIX-BASIS contains the selection of a diagonal. Users of ALGOL 68 or TORRIX-BASIS can apply these features e.g. to reduce computation time.

Objective 5.5.2. The software system should contain a slicing mechanism and provide means such that ascription of a slice to an identifier reduces computation time if the slice is used several times in the user program.

This objective leads into the occurrence of side effects. Unexpected (and therefore undesired) side effects should be avoided. On the other hand, it should be impossible for side effects not to occur when the user expects and desires them.

Objective 5.5.3. Design the slicing mechanism in TORRIX-SPARSE in such a way that the occurring side effects are similar to the side effects of ALGOL 68 and TORRIX-BASIS and moreover, that they occur in similar situations as in ALGOL 68 or TORRIX-BASIS user programs.

Objective 5.5.4. The user does not need to program the garbage collector for (parts of) the data structure in TORRIX-SPARSE. As soon as storage generated on the (ALGOL 68) heap cannot be used anymore during the elaboration of the user program, this storage should be available for the garbage collector of the ALGOL 68 implementation.

Of course design objectives 1.2 and 1.3 of TORRIX-BASIS (see chapter I) also hold for TORRIX-SPARSE. Obviously a clear distinction must be made in TORRIX-SPARSE as well between operations with a meaning derived from linear algebra and operations with a meaning related to the data structure. For operations derived from linear algebra we can use the same operator symbols as in TORRIX-BASIS.
The slicing mechanism is not the only feature of TORRIX-SPARSE that leads into the occurrence of side effects. TORRIX-SPARSE is designed in such a way that storage may be saved in case of equal blocks because, if specified by the user, only one of them will be actually stored (see (5.2)). This gives rise to side effects for which no equivalent exists in TORRIX-BASIS. We will provide a solution for this side effect problem, not by excluding side effects, but by controlling them.

V.2 A REVIEW OF SPARSE MATRIX DATA STRUCTURES.

Most data structures for sparse matrices are built of linear arrays. The reason is the rather poor data structure facility of FORTRAN and ALGOL 60 (in which most software for sparse matrices is written). In this section we will review a number of data structures for sparse matrices, provide ALGOL 68 mode declarations for them and see to what extent our design objectives are met. In V.2.1 we will mention two data structures for special sparsity patterns, in V.2.2 we will deal with data structures suitable for arbitrary sparsity patterns and in V.2.3 we will deal with block patterns. We will see that none of them provides a good slicing mechanism nor has the possibility for more economic storage in case of equal blocks.

V.2.1 Band and profile data structures.

A full n×n band matrix A with bandwidths w1 and w2 is often stored in a doubly subscripted array a with bounds [1:n,-w1:w2]. The rows of a correspond to the rows of A and the columns of a correspond to the diagonals of A. The ratio between administration overhead and the number of stored elements is quite good, even for large w1 and w2. It is clear that this data structure should not be used for matrices with arbitrary sparsity patterns. It provides efficient row and diagonal access, but column access is rather poor. No advantage can be taken of the equality of blocks to reduce storage.

For the case of symmetric matrices Jennings (cf. [46]) proposed to store only the lower triangle of the profile of the matrix. There is one array containing the elements of the profile and one array of pointers, one for each main diagonal element, to offset the row-indexing (see fig. 5.1). In ALGOL 68 a more appropriate data structure for Jennings' method is the *rowmat* data structure (see (1.10)). Most of our design objectives cannot be met even with a modest change of this data structure.

V.2.2 Data structures for arbitrary sparsity patterns.

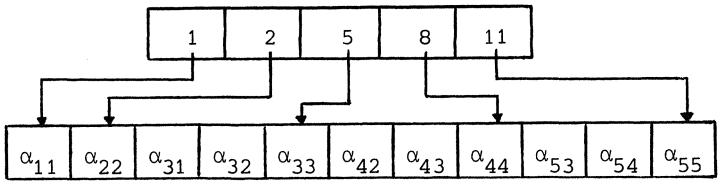
A data structure for general use has been mentioned in [47]. Many other data structures are restrictions, extensions or adaptations of it (cf. [30], [63]). In this data structure a matrix is viewed as a set of elements, each one uniquely determined by two integers: the row-index and the column-index. Each non-zero element is stored as a 3-tuple (its value, row- and column-index) and two pointers, one referring to the next non-zero element in the same row and the other to the next non-zero element in the same column. In most applications of sparse matrices it is very improbable that some column or row of the matrix to be stored, does not contain any non-zero element. Thus, the pointers to the first stored elements of the rows and columns can be stored in arrays. We can implement the structure with the following ALGOL 68 modes:

```
mode element = struct(int rowind, colind, scal value
                     ,ref element right, down
                     );
mode spmat = struct(ref[]ref element rows, cols);
```

(5.15)

$$A = (\alpha_{ij}) = \begin{pmatrix} \alpha_{11} & 0 & \alpha_{31} & 0 & 0 \\ 0 & \alpha_{22} & 0 & \alpha_{42} & 0 \\ \alpha_{31} & 0 & \alpha_{33} & \alpha_{43} & \alpha_{53} \\ 0 & \alpha_{42} & \alpha_{43} & \alpha_{44} & 0 \\ 0 & 0 & \alpha_{53} & 0 & \alpha_{55} \end{pmatrix}$$

Jennings' storage scheme for A:



The corresponding ALGOL 68 data structure:

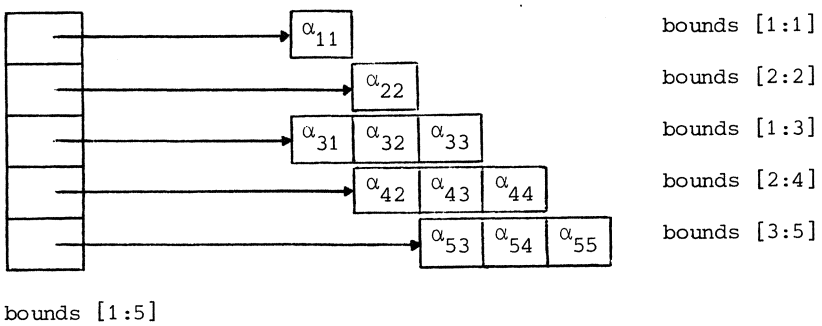


fig., 5.1. The Jennings' storage scheme.

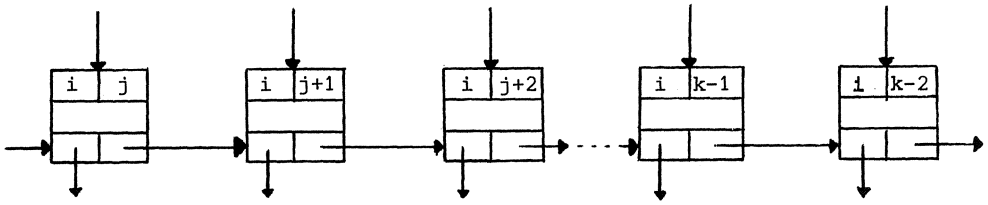
The bounds of the matrix are the bounds of the *rows*- and *cols*-fields. Selecting a row of an *spmat* *a* is very easy:

(rows of a)[i]

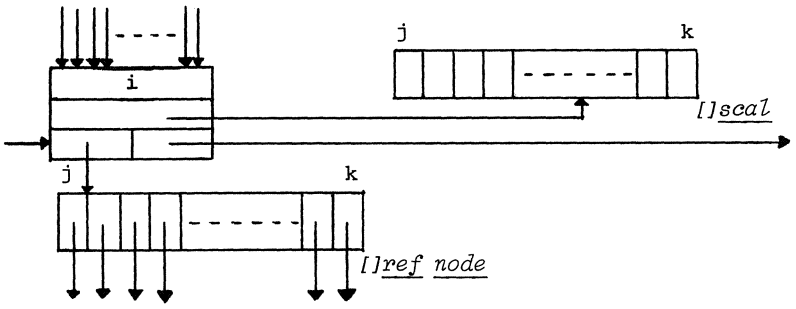
However, the size of the administration overhead can be too large (cf. [63]). In [23] other disadvantages are mentioned and it was tried to avoid them by storing the non-zero elements of each row in an array with deletion of the

row- and column-pointers. Thus column access becomes time consuming and it will require more time to insert newly created non-zero elements in this restricted data structure.

If diagonal access is needed, data structure (5.15) can be easily extended by adding to each *element* a pointer to the next non-zero in the same diagonal. The above mentioned disadvantages can be partly avoided if a number of stored elements in a row have consecutive column-indices. These elements can be stored in one array. Column-pointers will be retained and no problem arises with the insertion of new non-zero elements (see fig. 5.2).



Non-compacted form.



Compacted form.

fig. 5.2. Compaction of stored elements with consecutive column-indices.

The mode declarations (using TORRIX-BASIS) can be:

```
mode rowelem = struct(int colind, scal value, ref node down);
mode rowcomp = struct(vec values, ref[]ref node downs);
mode node = struct(int rowind
                   ,union(ref rowelem, ref rowcomp) rowpart
                   ,ref node right
                   );
mode spmat = struct(ref[]ref node rows, cols);
```

(5.16)

Compared with (5.15), k-j column-indices and k-j row-pointers are not stored in this compaction. Obviously (5.16) can be easily adapted to allow column or even submatrix compaction. Efficient cyclic row- and column-permutations are not possible with data structure (5.15).
In order to implement a slicing mechanism with side effects as mentioned in objective 5.5.3, a severe modification of (5.15) (or (5.16)) is required, in particular if additional efficiency requirements for the use of slices are stated. It is not difficult to obtain a reduction in storage in case of equal blocks, but a reasonable solution (as presented in V.5.3.3) with regard to the occurrence of side effects due to the equality of blocks cannot easily be implemented. It is not surprising indeed, that slicing mechanisms or features for a reduction in storage in case of equal blocks are not implemented (as far as we know) in sparse matrix packages using data structures related to (5.15) (cf. [23], [34] and [82]).

V.2.3 Data structures for block patterns.

A sparse matrix with only full and zero blocks can be stored as an array of blocks (cf. [26], [12]):

```
mode spmat = struct(ref[]int rowpartition, colpartition
                   ,ref[,]mat blocks
                   );
```

(5.17)

A zero block is represented with *zeromat*.
In case of a block diagonal pattern with only full and zero blocks, one can use a *ref[]ref[]mat blocks* in (5.17). If some non-zero blocks are sparse, other mode declarations are needed; a block is an *spmat* itself:

```
mode matrix = union(mat, spmat);
mode spmat = struct(ref[]int rowpartition, colpartition
                    ,ref[,]matrix blocks
                    );
```

In more recent publications (cf. [30], [62]), this latter data structure is only mentioned without any reasoning about advantages or disadvantages. The data structure has been used for Gaussian elimination of symmetric positive definite systems (cf. [26]) and QR decomposition (cf. [12]). We will modify it in the next sections in such a way that our design objectives can be met rather well.

Conclusion. In this section we have reviewed a number of data structures for sparse matrices. None of them provide facilities for an economical storage of equal blocks. This is left to the user and it seems that the pattern of equal blocks will only be reflected in his program. Several data structures allow a slicing mechanism. However, no attention has been given to the side effect problem.

V.3 A NEW DATA STRUCTURE FOR SPARSE MATRICES.

In this section we will present a new data structure for sparse matrices. It allows a reduction in storage in case of equal blocks. If this reduction is not required, this data structure is a rooted tree with leaves that are *mat*s or *diagmat*s (see chapter I) and its subtrees correspond to blocks of the matrix. If a reduction in storage in case of equal blocks is required, only one of the corresponding subtrees will be actually stored. This identification of two subtrees leads to a data structure that is an acyclic directed graph.
Section V.3 consists of two subsections. In V.3.1 we will give basic definitions and briefly explain the rooted tree data structure for sparse matrices. In V.3.2 we will see how a reduction in storage can be obtained in case of equal blocks. We will give no definitions for rooted trees and related notions. These can be found in [01].

V.3.1 Sparse matrix storage trees.

Let a scalar field F and a computational system S for F be given. Thus, if we speak about total-arrays, we mean total-arrays over S. In order to relate trees to matrices, we have to define precisely what is meant by blocks, partitions, etc.

DEFINITION 5.1. A block with bounds (bnds) $[m_1:n_1,m_2:n_2]$ is a total-array2 $[\alpha_{ij}]$ with $\alpha_{ij}=0$ for all $(i,j) \in [-t:t]\times[-t:t]\setminus[m_1:n_1]\times[m_2:n_2]$. Its lowerbounds are m_1 and m_2; its upperbounds are n_1 and n_2.

Observe that each total-array2 is a block with bounds [-t:t,-t:t]. Moreover, a block does not necessarily have lowerbounds 1.

DEFINITION 5.2. Let $B=[\beta_{ij}]$ be a block with $bnds(B)=[m_1:n_1,m_2:n_2]$. Let $R=(r_h)_{0\leq h\leq p}$, $C=(c_k)_{0\leq k\leq q}$ be strictly increasing sequences over $\mathbb{Z}$ satisfying

$$p,q\geq 1,\ p\times q\geq 2,\ r_0=m_1-1,\ c_0=m_2-1,\ r_p=n_1 \text{ and } c_q=n_2. \tag{5.18}$$

A partition of B according to R and C consists of a doubly subscripted sequence of blocks $(B_{hk})_{1\leq h\leq p,1\leq k\leq q}$ with $B_{hk}=[\beta^{hk}_{ij}]$ and $bnds(B_{hk})=[r_{h-1}+1:r_h, c_{k-1}+1:c_k]$ such that

$$\beta^{hk}_{ij}=\beta_{ij} \text{ if } (i,j)\in bnds(B_{hk}).$$

We say that any B_{hk} $(1\leq h\leq p,1\leq k\leq q)$ is a direct subblock of B and that the size of the partition of B is pq.

DEFINITION 5.3. Let B be a block. If B is not partitioned, we say that B is recursively partitioned with depth 0. B is recursively partitioned with depth s $(s\in\mathbb{N},s>0)$ if B is partitioned into $(B_{hk})_{1\leq h\leq p,1\leq k\leq q}$ (in the sense of definition 5.2) and its direct subblocks satisfy the following two conditions:

(i) each direct subblock of B is recursively partitioned with depth at most s-1,

(ii) there is at least one direct subblock of B that is recursively partitioned with depth s-1.

The size of a recursive partition P of B with depth s equals:

$$size(P) = \begin{cases} 1 & \text{if } s=0 \\ 1 + \sum_{h=1}^{p}\sum_{k=1}^{q} size(P_{hk}) & \text{otherwise } (s>0) \text{ with } P_{hk} \text{ the recursive partition of } B_{hk}. \end{cases}$$

DEFINITION 5.4. Let B and B' be blocks. Let a recursive partition of B be given. B' is a subblock of B if there are blocks $B=B_0,B_1,\ldots,B_n=B'$ ($n\geq 0$) such that B_i is a direct subblock of B_{i-1} ($1\leq i\leq n$).

Obviously, if $B\neq B'$ and B' is a subblock of B, then we can define another recursive partition for B in which B' is a direct subblock of B. However, this can lead to a considerable increase in the size of the partition or the sizes of the *mat*s and *diagmat*s representing non-partitioned subblocks (see proposition 5.1) and, hence, to a considerable increase in the storage required for the (non-empty) blocks of the given matrix. Now we will see how non-partitioned blocks can be represented with *mat*s and *diagmat*s (as defined in I.4.3).

DEFINITION 5.5. A *diagarray* is an ALGOL 68 multiple value of *vec*s. The k^{th} *vec* is considered to represent one diagonal of a total-array2 $[\alpha_{ij}]$ (i.e., all elements $\alpha_{i,i+k}$ $(-t\leq i,i+k\leq t)$).

Because each total-array2 has a domain $[-t:t]\times[-t:t]$, each *diagarray* d must satisfy the following conditions:

(i) $-2t\leq lwb\ d$ and $upb\ d\leq 2t$,

(ii) for each k ($lwb\ d\leq k\leq upb\ d$) we have

if $k<0$ then $-t-k\leq lwb\ d[k]$ and $upb\ d[k]\leq t$ and

if $k\geq 0$ then $-t\leq lwb\ d[k]$ and $upb\ d[k]\leq t-k$.

In the remainder of this chapter we will seldom speak of concrete-*array2*'s and *diagarray*s, but only of *mat*s (i.e., references to concrete-*array2*'s) and *diagmat*s (i.e., references to *diagarray*s (see I.4.3)). Without proof we state (see also I.3.3):

Proposition 5.1. *Let* B *be a non-partitioned block with* $\mathrm{bnds}(B)=[m_1:n_1,m_2:n_2]$. *Then* B *can be represented by a mat or a diagmat* d. *The concrete domain* D *of the mat satisfies* $D\subseteq \mathrm{bnds}(B)$ *and* d *satisfies:* $lwb\ d\geq m_2-n_1$, $upb\ d\leq n_2-m_1$ *and for all* k ($lwb\ d\leq k\leq upb\ d$) *we have*

$$lwb\ d[k]\geq \max\{m_1,m_2-k\},\quad upb\ d[k]\leq \min\{n_1,n_2-k\}.$$

DEFINITION 5.6. Let B be a block and let a recursive partition of B be given. Moreover, let each non-partitioned subblock of B be represented by a *mat* or *diagmat*.

(i) A *mat* *m* has a concrete element with indices (i,j) if and only if *1 lwb m*≤i≤*1 upb m* and *2 lwb m*≤j≤*2 upb m* (cf. [75]).

(ii) A *diagmat d* has a concrete element with indices (i,j) if and only if *lwb d*≤j-i≤*upb d* and *lwb d*[j-i]≤i≤*upb d*[j-i].

(iii) B has a concrete element with indices (i,j) if and only if there is a non-partitioned subblock B' of B, represented by a *mat* or *diagmat* with a concrete element with indices (i,j).

(iv) An element is virtual if it is not concrete.

DEFINITION 5.7. Let T be a tree with leaves that are *mat*s or *diagmat*s. Let A be a matrix or block; let a recursive partition P of A be given and let each non-partitioned block of A be represented by a *mat* or *diagmat*. T is a sparse matrix storage tree for A if

(i) there is a unique correspondence between vertices of T and subblocks of A,

(ii) the sons of an internal vertex N precisely correspond to the direct subblocks of the block associated with N,

(iii) each leaf of T is equal to a *mat* or *diagmat* that is a representation of a non-partitioned block of A, and vice versa.

The block B corresponding to a vertex of T will be completely determined by associating with the vertex the four integers l_1, u_1, l_2, u_2 such that bnds(B)=[l_1:u_1,l_2:u_2]. Note that, if B' is a subblock of B and their corresponding vertices in T are N' and N, respectively, then N' is a descendant of N. Because each partitioned block has at least two direct subblocks, each internal vertex of T has at least two sons. Thus, the number of vertices in a sparse matrix storage tree for A is equal to the size of the recursive partition of A. Observe that even a block represented by a *mat* (or *diagmat*) without any concrete element requires a vertex in the storage tree. Thus, if storage should be minimized, blocks with all zeros should not be partitioned. With definition 5.7 several notions related to sparse matrix storage trees correspond to notions related to recursive partitions. A rather important concept is given in the following definition:

DEFINITION 5.8. Let A be a block and let a recursive partition P of A be given. The sequence $(A=B_0,B_1,\ldots,B_n)$ is a search path for the $(i,j)^{th}$ element of A if B_p is a direct subblock of B_{p-1} $(1\leq p\leq n)$, B_n is not partitioned and $(i,j)\in bnds(B_n)$.

In a sparse matrix storage tree T for A the vertices corresponding to B_0, $\ldots,B_n$ constitute the path in T from the root of T to the leaf containing the $(i,j)^{th}$ element (possibly virtual). Observe that the search path of every element of A is uniquely determined.

V.3.2 An acyclic directed graph storage scheme.

In this section we introduce a modification of sparse matrix storage trees that gives a reduction in storage in case of equal blocks. First we will review some necessary concepts of graph theory.
A directed graph G=(V,E) consists of a finite set V of vertices and a set $E\subseteq V\times V$ with $(u,u)\notin E$ for each $u\in V$. The elements of E are called edges. The indegree of a vertex v is the size of $\{u\in V : (u,v)\in E\}$ and the outdegree of v is the size of $\{u\in V : (v,u)\in E\}$. A path $\pi=(v_1,v_2,\ldots,v_n)$ of length n-1 $(n\geq 1)$ is a sequence of vertices such that $(v_i,v_{i+1})\in E$ $(1\leq i\leq n-1)$. A cycle of length n is a path $(v_1,\ldots,v_n,v_1)$. A directed graph G=(V,E) is acyclic if it does not contain a cycle.
For acyclic graphs we use the following notions: a vertex v is a root if indegree(v)=0; a vertex v is a leaf if outdegree(v)=0; a vertex v is a son of a vertex u if (u,v) is an edge; a vertex v is a descendant of a vertex u if there is a path from u to v.

DEFINITION 5.9. Let A be a block and let a recursive partition P of A be given. Let B and B' be subblocks of A with $bnds(B)=[l_1:u_1,l_2:u_2]$ and $bnds(B')=[l_1':u_1',l_2':u_2']$. B and B' are equal blocks if $u_1-l_1=u_1'-l_1'$, $u_2-l_2=u_2'-l_2'$ and the value of element (i,j) of B equals the value of element $(i+l_1'-l_1,j+l_2'-l_2)$ of B' for all $(i,j)\in bnds(B)$.

Let A, P, B and B' be as in definition 5.9. Let T be a sparse matrix storage tree for A according to P. Suppose B and B' are equal $(B\neq B')$ and let N(B) and N(B') be the vertices of T corresponding to B and B', respectively. Let N(B) and N(B') be sons of F(B) and F(B'), respectively. Consider T as an acyclic directed graph. Then a reduction in storage (in the size of T) can be obtained

by deleting N(B') and all its descendants from T and adding the edge (F(B'), N(B)). Moreover, the bounds of B' must be added as a label to N(B) (more precisely: l_1'-l_1 and l_2'-l_2 must be added as labels to the edge (F(B'),N(B)). In this way, B and B' are made structurally equal. Obviously, this deletion and addition of vertices can be done for all equal blocks of A, starting with the larger ones. Suppose this is done for a number of equal blocks and we obtained an acyclic directed graph G. G satisfies the following properties:

(i) G has exactly one root r,

(ii) for each vertex x in G, there is a path from r to x. If for each x, this path is unique, then G is a tree and the matrix A does not contain structurally equal blocks.

Observe that the search path for each element of A is uniquely determined if A is stored in an acyclic directed graph.

Above we have given the general way in which equal blocks can be made structurally equal. Many questions are not solved. For instance, how to deal with the situation in which one of the deleted vertices had indegree >1? For this kind of details as well as many others, we refer to V.5.

Conclusion. In section V.3 we have presented a new data structure for sparse matrices based on a recursive partition of the matrix. This leads to a storage scheme that is represented by a tree. A reduction in storage in case of equal blocks can be obtained by identifying vertices and thus transforming the tree into an acyclic directed graph in which several vertices have indegree >1.

V.4 SLICING IN TORRIX-SPARSE.

In this section we will explain in more detail what, in our view, a slicing mechanism should contain and what the requirements are concerning the occurrence of side effects that come with a slicing mechanism. For this purpose we need a short survey of how values of modes for sparse matrices as defined in TORRIX-SPARSE can be manipulated. This survey is given in section V.4.1. In section V.4.2 we will discuss the typical properties of the ALGOL 68 slicing mechanism and see what the slicing features are in ALGOL 68 and TORRIX-BASIS. Then, in V.4.3, we will present the slicing operators as defined in TORRIX-SPARSE.

V.4.1 Types of variability and operations in TORRIX-SPARSE.

In TORRIX-SPARSE several modes will be declared to represent sparse matrices in computer memory (see V.5). Because we do not need to distinguish between these modes in this section, we will denote them all by simply the word *sparmat*. To manipulate *sparmat*-values the user can only use the following instructions:

(i) operations defined in the software system. We will not distinguish between procedures and operators. In ALGOL 68 an operation can be implemented with an operator or a procedure. The word 'operand' in V.4 and V.5 will sometimes denote 'parameter'. All this will not lead to ambiguities,

(ii) an ALGOL 68 assignment involving *sparmat*s,

(iii) an ALGOL 68 identity declaration involving *sparmat*s. This also includes the parameter mechanism of procedure calls.

Note that, although a *sparmat* is an ALGOL 68 structured value, the user cannot select its separate fields.

All data structures used in TORRIX-SPARSE are based on a recursive partition of the matrix to be represented in memory. Therefore, we can distinguish four types of variability of *sparmat*s:

(i) changing the value of concrete *scal*-elements, (5.19)

(ii) replacing *mat*s or *diagmat*s of non-partitioned blocks by other *mat*s or *diagmat*s, (5.20)

(iii) changing the recursive partition of a block or replacing a block by another block with the same bounds, (5.21)

(iv) changing the bounds of the *sparmat*. (5.22)

The mode declarations for *sparmat* are chosen in such a way that a change in a *sparmat* is always of type (5.19), (5.20) or (5.21). A change of type (5.22) cannot be performed on *sparmat*s. For (5.22) *ref sparmat*s are necessary. Observe that variability (5.22) in TORRIX-SPARSE corresponds to variability (1.9) in TORRIX-BASIS. Operations for the modes *sparmat* can be divided into three types:

I Queries:

(i) Operations of which the result is determined by the total-array of the operand.

Examples: the sum of the values of all elements;
generating a *(vec-)*copy of row i.

(ii) Operations of which the result is determined by the storage scheme.

Examples: the number of concrete elements;
queries about the recursive partition.

II Operations to generate new *sparmat*s.

Examples: the sum of two *sparmat*s;
generating a *sparmat* without concrete elements, but with the same recursive partition as the operand.

III Operations that change the total-array, the recursive partition or the concrete domain(s) of a *mat* or *diagmat* of a *sparmat*. (5.23

TORRIX-SPARSE contains type III operations that are examples of variability (5.19), (5.20) and (5.21) simultaneously (for instance, permutations of rows, cyclic row-permutations). However, none of the operations is an example of variability (5.22). This variability is left to the user. As a consequence, no operation provided in TORRIX-SPARSE has an operand of the mode *ref sparmat*.

V.4.2 Slicing in ALGOL 68 and TORRIX-BASIS.

In this section we will use concepts like submatrix, row, diagonal, *scal*-element, etc. of a matrix with the usual meaning. Clearly submatrices, rows, etc. are involved in a slicing mechanism. Merely defining a number of operations that act only on parts (i.e., on a submatrix, a row, etc.) of a matrix does not necessarily provide a slicing mechanism. If a software system contains a slicing mechanism then the user is able to apply an operation (possibly defined by himself) defined on a vector, to a row(, column or diagonal) of a matrix or to a subvector of a vector. Thus the user can freely use rows and columns of a matrix and subvectors of a vector without the obligation to distinguish between a vector that is part of another data structure and an autonomous vector. We describe a slice of a data structure for a matrix or vector as the selection of a part of it, of which the elements may have quite different indices compared with their indices of the matrix or vector. Examples: selection of a submatrix, the lower triangle, the elements of a vector

with even index, etc. Moreover, a slice can be sliced again. It is important to note that a change in a slice constitutes a change in the original matrix or vector. And vice versa: if the original data structure has been changed in an element selected by a slice, then the slice is changed accordingly. Observe that a slicing mechanism is not introduced because of the efficiency with which operations can be performed, but primarily as a part of the interface between a software system and its user.

ALGOL 68 provides four kinds of slicing of references to doubly subscripted multiple values:

(i) submatrix slicing: $a[h_1:k_1 \ \underline{at}\ h_1,\ h_2:k_2 \ \underline{at}\ h_2]$,

(ii) revised lowerbound: $a[h_1:k_1 \ \underline{at}\ p_1,\ h_2:k_2 \ \underline{at}\ p_2]$,

(iii) vector slicing: row slicing $a[i,\ ]$,

column slicing $a[\ ,j]$,

(iv) element slicing: $a[i,j]$.

TORRIX-BASIS contains two new kinds of slicing a *mat*:

(v) transpose slicing: *trnsp* a,

(iii) vector slicing: diagonal slicing k *diag* a.

Similar kinds of slicing exists in ALGOL 68 and TORRIX-BASIS for references to singly subscripted multiple values: subvector slicing, revised lowerbound and element slicing. An important aspect of the slicing mechanism in ALGOL 68 and TORRIX-BASIS is, that slices have fixed bounds (fixed at execution). Slices in TORRIX-BASIS do not allow variability (1.9).

V.4.3 Slicing operators in TORRIX-SPARSE.

It is often the case in a program that a slice must be used several times. It is rather easy to design a slicing mechanism in such a way that with each use of the same slice, the whole data structure of the original matrix or vector is searched for the elements contained in the slice (even if the slice has been ascribed to an identifier). Thus we have a new objective:

Objective 5.5.5. Design a slicing mechanism such that, if a slice is used several times and the original data structure has not been changed between these usages, then the original data structure will be explored only once (namely when the slice is created). (5.24)

In TORRIX-SPARSE we have the following slicing operators:

(i) the operator *?* for submatrix, row and column slicing (see TORRIX-BASIS), (5.25)

(ii) the operator *diag* for diagonal slicing, (5.26)

(iii) *a!(i?j)*: if $(i,j) \in \text{bnds}(a)$ then a reference to the (possibly newly generated) concrete $(i,j)^{\text{th}}$ element of *a* will be returned; otherwise a fatal error will be reported, (5.27)

(iv) *a newlwb h* is equivalent with the ALGOL 68 `A[ : at h, ]`, (5.28)

(v) *k newlwb a* is equivalent with the ALGOL 68 `A[ , : at k]`, (5.29)

(vi) *a symslice(h//k)* is equivalent with the ALGOL 68 `A[h:k at h, h:k at h]` but can only be applied to a symmetric matrix or a matrix with a symmetric sparsity pattern. (5.30)

Moreover, two new modes are declared:

spvec to represent sparse vectors,

scalel to represent a reference to an element of a *sparmat* or an *spvec*.

An *spvec* is not necessarily a slice of a *sparmat*. It represents a total-array1 and is recursively partitioned according to rules similar to those for matrices (see V.3). The recursive partition of a submatrix or vector slice is the restriction of the recursive partition of the original matrix. *spvec*s can be sliced also, with the following operations:

(i) the operator *?* for subvector slicing (see also TORRIX-BASIS), (5.31)

(ii) *u newlwb h*, equivalent with the ALGOL 68 `U[ : at h]`, (5.32)

(iii) *u!i*: if $i \in \text{bnds}(u)$ then a reference to the (possibly newly generated) concrete i^{th} element of *u* will be returned; otherwise a fatal error will be reported. (5.33)

Observe that the indices determine whether an element belongs to a slice. Which elements belong to a slice depends neither on the recursive partition nor on the concreteness of elements. In V.5 we will see that the operations (5.25)-(5.33) form a slicing mechanism in TORRIX-SPARSE, i.e., TORRIX-SPARSE satisfies the following criterion:

Let π be a TORRIX-SPARSE user program. Let, at some stage of the elaboration of π, v be a *sparmat*- or *spvec*-value and sl be a value that is just created as a result of an application of one of the operators (5.25)-(5.33) applied to v. Let SL be the set of indices of elements selected by sl. Let after the elaboration of an arbitrary number of TORRIX-SPARSE operations applied to sl and/or v, a value tl be created with an identical application of the operator with which sl was created. Then sl and tl are of the same mode and if sl is a *scalel*, sl and tl have the same *scal*-value. If sl is an *spvec* or *sparmat*, then the following four conditions hold:

(i) the total-arrays of sl and tl are equal,

(ii) the recursive partitions of sl and tl are equal,

(iii) the bounds of sl and tl are equal,

(iv) the i^{th} (or $(i,j)^{th}$) element of sl is concrete if and only if the i^{th} (or $(i,j)^{th}$) element of tl is concrete. (5.34)

Thus, if a and u are *sparmat*- or *spvec*-variables, and u refers to a slice of a, then during the elaboration of π, u continues to refer to this slice of a, until a or u is the destination of an assignment.

Conclusion. In this section we have discussed the properties of a slicing mechanism. Operators that form a slicing mechanism must induce specific side effects in the matrix (or vector) and its defined slices, if other operations are applied to this matrix (or vector) and its slices.

V.5 TORRIX-SPARSE: MODES AND ALGORITHMS.

In this section we will give a rather detailed description of the operators and data structures TORRIX-SPARSE provides. We will develop TORRIX-SPARSE in five stages (SPARSEI, SPARSEII, SPARSEIII, SPARSEIV and SPARSEV (which is the complete TORRIX-SPARSE system)), because otherwise the explanation of TORRIX-SPARSE would become too complicated. Moreover, we can show in each stage what additional problems arise for the implementor. Each SPARSE system is an extension of a former one. The separate SPARSE systems can be considered sparse matrix packages on their own, but in each (except in SPARSEV) a number of the design objectives (see V.1) are not met. In table 5.1 one can find which objectives are met in each of these systems.

SPARSEI is a system merely using the sparse matrix storage tree data structure for sparse matrices (see V.3.1). It has no additional features like provisions for structurally equal blocks or a slicing mechanism. In SPARSEII the "shift" will be incorporated in order to allow the insertion of rows and columns and to have efficient cyclic permutations. SPARSEIII incorporates the feature of structurally equal blocks. In SPARSEIV a slicing mechanism is implemented but structurally equal blocks are not possible. SPARSEV (i.e., TORRIX-SPARSE) contains both a slicing mechanism and a feature to reduce storage in case of equal blocks.
In each SPARSE system (except SPARSEI) we will only deal with the problems caused by the new features added by the extension. Thus, for instance, to comprehend SPARSEIII, one has to read the sections V.5.1-V.5.3. In an appendix to chapter V, a table is given containing a formal description of all operator symbols used in this section. TORRIX-SPARSE contains more operators and procedures than described in this appendix.

system	extension of	design objectives that are met
SPARSEI		objectives 5.1, 5.3, 5.4, 5.5.1, 5.5.4 and (5.1), (5.3)-(5.6), (5.8)-(5.12), (5.14)
SPARSEII	SPARSEI	objectives 5.1, 5.3, 5.4, 5.5.1, 5.5.4 and (5.1), (5.3)-(5.14)
SPARSEIII	SPARSEII	objectives 5.1, 5.2, 5.3, 5.4, 5.5.1 and 5.5.4
SPARSEIV	SPARSEII	objectives 5.1, 5.3, 5.4, 5.5 and (5.1), (5.3)-(5.14)
SPARSEV	SPARSEIII and SPARSEIV	objectives 5.1, 5.2, 5.3, 5.4 and 5.5.

table 5.1. Organization of V.5.

TORRIX-SPARSE provides the user with three modes to represent sparse matrices: the mode *spsym* is used for sparse symmetric matrices, *sympat* for matrices with symmetric sparsity pattern and *spmat* for other sparse matrices. If we do not want to distinguish the modes of the various systems, we will use *spsym*, *sympat* and *spmat*; otherwise we will use modes like *spsymi* (in SPARSEI), *spsymiv* (in SPARSEIV), *sympatii* (in SPARSEII), *spmati* (in SPARSEI), etc. If we do not want to distinguish between *spsym*, *sympat* and *spmat*, we will use

sparmat (or *sparmati*, *sparmatii*, etc.). These distinctions are only used in V.5. The user can only manipulate *spsym*s, *sympat*s and *spmat*s. Depending on the system used, these modes are declared differently. In all SPARSE systems the identifiers *zerospsym*, *zerosympat* and *zerospmat* are declared to denote a block with bounds [t:-t,t:-t] having no concrete element at all.

Remark 5.1. Obviously all operations (5.3)-(5.14) acting on a part of a matrix can be implemented (and objective 5.3 be met) by using the slicing mechanism. However, we will provide in TORRIX-SPARSE procedures for them, so that the user can avoid the generation of a slice in case such a slice of the matrix would be used only once.

In this section V.5 we will present a number of ALGOL 68 programs for TORRIX-SPARSE operations. These are meant to illustrate how the data structures can be manipulated.

The slicing mechanism is not the only cause of the occurrence of side effects. In V.1 we have already mentioned a second cause: structurally equal blocks. Moreover, the reference mechanism of ALGOL 68 provides us with side effect problems also. In order to control the occurrence of side effects, we have hidden from the user a number of operations and modes defined in TORRIX-SPARSE.

Before we start the explanation of TORRIX-SPARSE, we will define a number of concepts that we will use frequently.

DEFINITION 5.10.

(i) A block B with bnds(B) = $[m_1:n_1,m_2:n_2]$ is square if $m_1=m_2$ and $n_1=n_2$.

(ii) A partition of a square block B into $(B_{hk})_{1\leq h\leq p,1\leq k\leq q}$ is square if for each i B_{ii} is square $(1\leq i\leq \min(p,q))$.

(iii) A recursive partition of a square block B is symmetric if B contains a subblock B_1 with bnds(B_1) = $[m_1:n_1,m_2:n_2]$ if and only if B contains a subblock B_2 with bnds(B_2) = $[m_2:n_2,m_1:n_1]$.

DEFINITION 5.11. Let B be a block, partitioned according to (5.18). Then r_i $(0\leq i\leq p)$ resp. c_j $(0\leq j\leq q)$ is a direct horizontal resp. vertical partition line of B with endpoints m_2 and n_2, resp. m_1 and n_1. Moreover, r_i $(0\leq i\leq p)$ and c_j $(0\leq j\leq q)$ are horizontal and vertical partition lines with endpoints m_2 and n_2, resp. m_1 and n_1, of all blocks of which B is a subblock.

DEFINITION 5.12. Let *a* be a *sparmat*. A rectangular subdomain of *a* consists of a four tuple $[p_1:q_1,p_2:q_2]$ with

1 lwb bnds a$\leq p_1 \leq$t, $-t \leq q_1 \leq$*1 upb bnds a*, *2 lwb bnds a*$\leq p_2 \leq$t and $-t \leq q_2 \leq$*2 upb bnds a*.

For reasons of clarity we repeat all modes declared in TORRIX-BASIS:

value modes: *mode scal* = *c* a user specified mode *c*;
mode vec = *ref[]scal*,
mat = *ref[,]scal*;

modes concerning indices and bounds:
mode index = *ref[]int*,
pair = *struct(int rowsub, colsub)*,
trimmer = *struct(int lower, upper)*;

Each SPARSE system contains the modes
mode matbnds = *struct(trimmer rowbnds, colbnds)*;
mode diagmat = *ref[]vec* *co* = *ref[]ref[]scal co*;

The mode *matbnds* serves for bounds of matrices, submatrices and blocks. In case of *spsym*s and *sympat*s, *trimmer*s suffice. Contrary to TORRIX-BASIS, TORRIX-SPARSE does not contain operators concerning bounds that have an operand of the mode *sparmat*. This kind of operation acts on *matbnds*.

V.5.1 The SPARSEI system.

V.5.1.1 Mode declarations in SPARSEI.

The SPARSEI system provides the user with three modes (*spsymi*, *sympati* and *spmati*) and operators for them. Their declarations are given in fig. 5.3. First we will develop the mode *spmati* and then adapt it such that the data structures for *sympati* and *spsymi* can be easily defined.
We have already stated (V.3.1) that a block is either a *mat*, a *diagmat* or that it is partitioned. The direct subblocks of such a block can be stored in rectangular or diagonal form. Thus we arrive at the following declaration of a *blocki*:

mode †*blocki* = *union(ref mat, ref diagmat, ref rectblocki, ref bandblocki)*;

The symbol † is used to denote that this mode, identifier or operator cannot

be applied by the user. A *ref mat* (*ref diagmat*) is incorporated in a *blocki* instead of a *mat* (*diagmat*) in order to avoid problems in the applications of operators with respect to the *union*-mechanism of ALGOL 68.

```
mode †rectblocki = struct(index rowpartit, colpartit
                          ,ref[,]blocki subblocks
                          );
mode †bandblocki = struct(index rowpartit, colpartit
                          ,ref[]ref[]blocki subblocks
                          );
```

In the routine-texts of the operators a *rectblocki b* is assumed to satisfy the following conditions:

(i) *lwb rowpartit* = *lwb colpartit* = *0* and
1 lwb subblocks = *2 lwb subblocks* = *1*,

(ii) *1 upb subblocks* = *upb rowpartit*, *2 upb subblocks* = *upb colpartit*,

(iii) with m = *rowpartit*[*upb rowpartit*], n = *colpartit*[*upb colpartit*]

$$\text{bnds}(b) = [\mathit{rowpartit}[0]+1\text{:m}, \mathit{colpartit}[0]+1\text{:n}], \tag{5.35}$$

(iv) the bounds of the direct subblock *subblocks*[*i*,*j*] are

$$\text{bnds}(\mathit{subblocks}[i,j]) = [\mathit{rowpartit}[i-1]+1\text{:}\mathit{rowpartit}[i], \mathit{colpartit}[j-1]+1\text{:}\mathit{colpartit}[j]\ ]. \tag{5.36}$$

Similar assumptions hold for a *bandblocki*. With (5.35) and (5.36) the bounds of a *blocki* can be computed, except when this *blocki* is not partitioned and is not a direct subblock. This exception will only occur if the *blocki* is the root of the stored sparse matrix:

```
mode spmati = struct(matbnds †bounds, ref blocki †root);          (5.37)
```

The bounds of the *root*-field of an *spmati* are given with the *bounds*-field. Observe that, in order to change a non-partitioned into a partitioned matrix, a *ref blocki*-field is needed in (5.37) instead of a *blocki*-field. If a *blocki* is neither partitioned nor contains any concrete element, it is assumed to be a *ref*erence to *zeromat* (defined in TORRIX-BASIS).

Example 5.1. The routine-text of the procedure *genspmat* is rather simple:

```
proc genspmati = (int lwb1, upb1, lwb2, upb2)spmati:
  co returns an spmati A with bnds(A) = [lwb1:upb1,lwb2:upb2] without any
     concrete element                                              co
    if lwb1>upb1 or lwb2>upb2      then zerospmati
  elif lwb1<-t or upb1>t or lwb2<-t or upb2>t
  then co error co   stop
       co or any other action that halts the program co
  else ( ( (lwb1,upb1), (lwb2,upb2) )  ,  heap blocki := heap mat := zeromat)
    fi
```

As for *spsymi*s it is assumed that their recursive partition is symmetric. Only the lower triangle of a symmetric matrix will be stored in an *spsymi*, except for square *mat*s (that are on the main diagonal of the matrix). The declaration of the mode *spsymi* is simple:

```
mode spsymi = struct(trimmer †bounds, ref blocki †symroot)
```

Each *mat*, *diagmat* or *blocki* of an *spsymi* that is not on the main diagonal, represents two submatrices of the matrix stored with this *spsymi*. The *rowpartit*- and *colpartit*-fields of a partitioned square *blocki b* of an *spsymi* are assumed to be equal as *index*es, i.e.,

ref[]int(rowpartit of b) is colpartit of b returns *true*. (5.3

For a square *blocki b* we can distinguish four cases:

(i) *b* is a reference to a *mat m*. Then *m* will be square and the strictly upper triangular part of *m* will never be used by any SPARSE operation,

(ii) *b* is a reference to a *diagmat d*. Then *upb d*≦*0* and thus, no strictly upper triangular element of the matrix will be stored,

(iii) *b* is a reference to a *rectblocki rb*. Then the strictly upper triangular direct subblocks of *rb* are *ref mat(nil)* values,

(iv) *b* is a reference to a *bandblocki bb*. Then *upb(subblocks of bb)*≦*0* (5.: and thus, no strictly upper triangular direct subblock of *bb* is stored.

The mode *sympati* can be used to store a sparse matrix with a symmetric sparsity pattern. As with *spsymi*s the recursive partition of a *sympati* is symmetric and, in order to decrease the need for administration overhead, each

```
mode †blocki = union(ref mat , ref diagmat
                    ,ref struct(mat m,trnspm), ref struct(diagmat m,trnspm)
                    ,ref rectblocki , ref bandblocki
                    );
mode †rectblocki = struct(index rowpartit, colpartit, ref[,]blocki subblocks
                         );
mode †bandblocki = struct(index rowpartit , colpartit
                         ,ref[]ref[]blocki subblocks
                         );
mode spmati = struct(matbnds †bounds,  ref blocki †root);
mode spsymi = struct(trimmer †bounds,  ref blocki †symroot);
mode sympati = struct(trimmer †bounds,  ref blocki †root);
```

fig. 5.3. Mode declarations in SPARSEI.

off-diagonal *blocki* of a *sympati* represents two submatrices of the matrix. However, each *mat* or *diagmat* of a *sympati* can only represent one submatrix. This cannot be fulfilled with the mode declarations given above, so we have to adapt them (see fig. 5.3). All we have said about *spmati*s and *spsymi*s is still valid with the declarations of fig. 5.3. In SPARSEI their *blocki*s will never be a reference to structured values with *mat*s or *diagmat*s. Partitioned *blocki*s of a *sympati* satisfy (5.38) and (5.39). The symmetric sparsity pattern puts some demands on the bounds of the *mat*s and *diagmat*s of a non-partitioned *blocki* of a *sympati*.

V.5.1.2 Interface between SPARSEI and TORRIX-BASIS.

Obviously SPARSEI contains a number of operators with *mat* or *vec*-operands. Otherwise, even a matrix-vector product of a *sparmat* and a *vec* could not be computed. Moreover, in order to build up a *sparmat*, the user should be able to insert a *mat* or *diagmat* in a *sparmat*. To avoid side effects later on in the user program, such a *mat* or *diagmat* itself will not be inserted but a reference to a copy of its concrete-*array*(s) will. This means that the storage cells used for the concrete-*array*s of *mat*s and *diagmat*s in a *sparmat* are neither generated (with a *loc* or *heap*-generator) in the user program nor are

they generated by the user himself with generating operations on TORRIX-BASIS.

V.5.1.3 SPARSEI operations and recursive partitions.

In V.4 we have divided operations in three types: queries, generating operations and operations that change a *sparmat*. Most operations in SPARSEI are rather easy to implement, because one can declare recursive operators in ALGOL 68. In program 5.1 an ALGOL 68 program is presented for the matrix-vector product of an *spmati* and a *vec*, using TORRIX-BASIS. In this program the advantage of design objective 1.3 of TORRIX (see I.2) is clear: as soon as computations with *scal*-elements are involved, an operator of TORRIX-BASIS is applied. Almost all lines deal with manipulating values of the administration of an *spmati*.
The meaning of query operations will be clear from the short description in a table of operators (see the appendix of this chapter).

The meaning of operations of type (5.23) that deal only with the recursive partition and the concrete domains of *mat*s and *diagmat*s (but do not change the total-array of the *sparmat*), will be clear from the specification. We must be more explicit about operations that may change both the total-array and the recursive partition. Except for the operators *<:=*, *assigndel* and *plusabdel* none of these operators changing the total-array2, will delete partition lines of the left (first) operand. Depending on the recursive partition and concrete domains of *mat*s and *diagmat*s of the other operand(s), new partition lines may be added. Among these operators we have *+<* and *-<*. The assign operation *a<:=b* (with *a* and *b* *sparmat*s) assigns a copy of *b* to *a*. Thus *a* obtains the recursive partition of *b* and a copy of the *scal*-values of *b*. It is an important operator in SPARSEIV and SPARSEV.
Making copies of *mat*s and *diagmat*s in the operations *a<:=b* and *a+<b* may be very inefficient. Thus we have delete-versions of these operations (*assigndel*, *plusabdel*) in which the right operand will have virtual elements only after the elaboration. In *a<:=b* the tree of *a* is replaced by a copy of the tree of *b*, whereas in *a assigndel b* the tree of *b* is deleted from *b* and assigned to *a*. Thus, *a assigndel b* needs a constant amount of time, while the amount of time to elaborate *a<:=b* depends heavily on the number of concrete elements of *b*. A similar distinction exists between *a+<b* and

```
op bnds = (int k, spmati a)trimmer:
   case k in  rowbnds of bounds of a,  colbnds of bounds of a  esac;
prio bnds = 8;
op × = (spmati a, vec u)vec:
    if emptyt(2 bnds a meet bnds u)      then zerovec
  else op × = (blocki b, vec u)vec:
        case b
         in (ref mat mm): mm×u
          , (ref diagmat dd): dd×u
          , (ref rectblocki rr):
             (index colp = colpartit of rr; loc int lwb:=1, upb := upb colp;
              to upb while  colp[lwb]<lwb u  do  lwb+:=1  od;
              to upb-lwb while  colp[upb-1]≥upb u  do  upb-:=1  od;
              loc vec result := zerovec;
              for j from lwb to upb
               do for i to  1 upb(subblocks of rr)
                    do  result +:= (subblocks of rr)[i,j]×u  od
               od; result
             )
          , (ref bandblocki bb):
             (index colp = colpartit of bb; loc int lwb := 1, upb := upb colp;
              to upb while  colp[lwb]<lwb u  do  lwb+:=1  od;
              to upb-lwb while  colp[upb-1]≥upb u  do  upb-:=1  od;
              loc vec result := zerovec;
              for k from  lwb subblocks of bb  to  upb subblocks of bb
               do ref[]blocki blockdiagk = (subblocks of bb)[k];
                  for j from  (lwb-k)max lwb blockdiagk
                          to  (upb-k)min upb blockdiagk
                   do result +:= blockdiagk[j]×u  od
               od; result
             )
        esac;
       (root of a)×u
   fi;
```

program 5.1. Matrix-vector product in SPARSEI.

a plusabdel b. In *a plusabdel b* subtrees of *a* may be replaced by subtrees of *b* and then the former subtree of *a* is added into the new subtree. This prevents large *mat*s (*diagmat*s) of *b* to be copied. Because in *a+<b* a copy of *b* will be added into *a*, no partition lines of *a* will vanish (see fig. 5.4)

Fig. 5.4 also reveals one way in which redundant partition lines may arise. We distinguish two kinds of redundant partition lines:

(i) partition lines that separate *block*s without any concrete elements, (5.4

(ii) partition lines that separate *block*s of which the *mat*s (or *diagmat*s) could be added without making virtual elements concrete (this type may arise after *a+<b* as in fig. 5.4). (5.4

To delete redundant partition lines, two operators are available (see also the appendix): *clear* and *cleargen*.

As for the generating operations, these are designed in such a way that two operands will not be changed and a new *sparmat* will be generated. In general, it is clear from the TORRIX point of view what the total-array of the resulting *sparmat* is. Only the recursive partition must be specified as well as which elements are concrete. Of each generating operation with two *sparmat*-

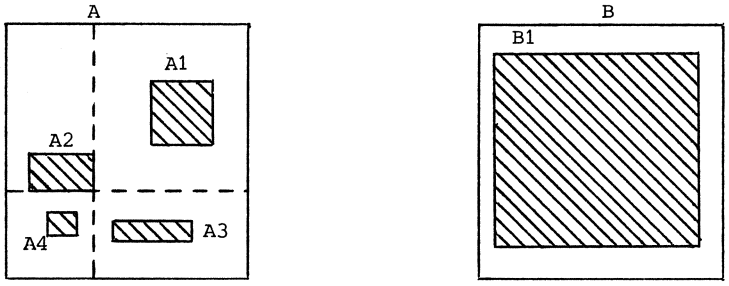


fig. 5.4. The additions A+<B and A *plusabdel* B.

In this case with A+<B four *mat*s B11, B12, B13 and B14 of B1 according to the partition of A are added to A1, A2, A3 and A4, whereas in A *plusabdel* B the *mat*s A1, A2, A3 en A4 are added into B1, then B1 is deleted from B and A is made to refer to B1 (thus deleting A1, A2, A3 and A4 from A).

operands, two versions are available: one version (*proc*) in which the recursive partition of the result *sparmat* is determined by the user and one version in which the recursive partition will be determined dynamically. Among these operations we have the sum, difference and product of two *sparmats*. As for the dynamically determined recursive partitions of the results of the operations $a+b$, $a-b$, $a\times b$, etc., we will present here only a few heuristics. Because $a+b$ and $a-b$ do not differ with respect to the choice of the recursive partition, we will only deal with $a+b$ and $a\times b$.

(i) let each partition line of a and b be a partition line in $a+b$ and $a\times b$. This will often give rise to many redundant partition lines in $a+b$,

(ii) as in (i), but now followed by a call of *cleargen*. This of course can be implemented more efficiently by performing the *cleargen* operation during the addition or multiplication,

(iii) let each direct partition line of the roots of a and b be partition lines of $a+b$. Determine the other partition lines with a strategy similar to the one used in *plusabdel*,

(iv) let each direct horizontal partition line of the root of a and each direct vertical partition line of the root of b be partition lines of $a\times b$. Determine the other partition lines with a strategy similar to the one used in *plusabdel*.

Other heuristics are certainly possible. In chapter VI we will once again deal with recursive partitions. One of the problems concerning the choice of recursive partitions is the contrast between dynamically created and user specified recursive partitions. In an operation like $a+b$ (a and b *sparmats*), the recursive partitions of a and b may be (partly) specified by the user and the question is to what extent these recursive partitions appear in the result *sparmat* of $a+b$.

There are a number of monadic operators that return a *sparmat* with the same recursive partition as the operand. For example: *copy a* and *concrcopy a* (in *copy a* a copy is made of all *mats* and *diagmats* of a). These operations can be coded such that the *indexes* *rowpartit* and *colpartit* are equal (in the sense of the identity relation *is*) to *rowpartit*- and *colpartit*-fields in the operand. Thus, one *[]int* can serve as the row-partition and/or column-partition of many *sparmats*. This was already the case with *spsyms* and *sympats*. In this way storage can be saved if in one user program several matrices with

the same sparsity pattern will be manipulated (see objective 5.4). However, in order to avoid the occurrence of side effects, a new *[]int* must be generated with each change in a row- or column-partition. These (or this) new *[]int*(s) will contain the new partition(s). This will be done automatically in all TORRIX-SPARSE operators. Observe that this kind of storage optimization is not necessarily confined to monadic operators like *copy*, but can occur also in many other operators.

In SPARSEI the user can access a sparse matrix storage tree T only by its root. Moreover, if he has access to the root of T, then he is allowed to access each vertex of T. Thus, if the user has access to the root, then no vertex can be given to the garbage collector of the ALGOL 68 implementation. On the other hand, as soon as the user has lost access to the root of T, he cannot access any other vertex of T. Therefore, all storage for T can be given to the garbage collector. This is automatically done in SPARSEI and the implementor does not need to write a garbage collector for SPARSEI.

This concludes the description of SPARSEI. Clearly the data structure underlying a *sparmati*-value, is a tree. An important feature is that two *sparmati*-values in SPARSEI have a *scal*-storage cell in common if and only if these *sparmati*-values are equal. For two equal *sparmati*-values a change in one of them will occur if and only if this change will occur in the other. Another important feature is that if a *sparmati*-value a contains in its underlying data structure a *mat* or *diagmat* with a concrete element (i,j), then this element is the only *scal* representing the $(i,j)^{th}$ element of a. Moreover, if the $(i,j)^{th}$ element of a is concrete, then a contains in its underlying data structure exactly one *mat* or *diagmat* with a concrete element (i,j).
In SPARSEI the user does not need to write his own garbage collector. As soon as (parts of) a *sparmati* cannot be used in the further elaboration of a user program, these parts will be given to the garbage collector of the ALGOL 68 implementation.

V.5.2 The SPARSEII system.

To allow efficient cyclic row exchange, block exchange and insertion of rows and columns, we extend the mode *blocki* with a shift. In terms of TORRIX-BASIS:

```
mode †blockii = struct(pair shift , shblockii block);
mode †shblockii = union(ref mat , ref diagmat
                        , ref struct(mat m, trnspm)
                        , ref struct(diagmat m, trnspm)
                        , ref rectblockii , ref bandblockii
                        );
mode †rectblockii = struct(index rowpartit , colpartit
                           , ref[,]blockii subblocks
                           );
mode †bandblockii = struct(index rowpartit , colpartit
                           , ref[]ref[]blockii subblocks
                           );
mode spmatii = struct(matbnds †bounds , ref shblockii †root);
mode spsymii = struct(trimmer †bounds , ref shblockii †symroot);
mode sympatii = struct(trimmer †bounds , ref shblockii †root);
```

fig. 5.5. Mode declarations in SPARSEII.

```
op shift = (mat a, pair shift)mat:
    a[at 1 lwb a + rowsub of shift , at 2 lwb a + colsub of shift]   (5.42)
```

Thus, the *mat* *a* itself is not considered, but the slice as given in (5.42). In order to allow efficient cyclic exchanges it is not enough that only *mat*s and *diagmat*s can be shifted. A block can be shifted also. The mode declarations in SPARSEII are given in fig. 5.5. We say that a *shblockii* of a *blockii* is shifted if and only if the *shift*-field is unequal to (0,0). The total-array2 of a *shblockii* and *blockii* can easily be defined. The user can make all shifted *shblockii*s of a *sparmatii* unshifted with the operator *unshift*: the shifts are incorporated in the *mat*s and *diagmat*s. No other operators, except for *clear* and *cleargen*, make shifted *shblockii*s unshifted. Program 5.2 is an implementation of the matrix-vector product.
Block exchange means subtree exchange and changing the *shift*-fields of the roots of both subtrees. Cyclic exchanges can now be performed by deletion of a number of rows (columns), shifting and insertion of rows (columns).

```
op × = (spmatii a, vec u)vec:
    if emptyt(cbnds bnds a meet bnds u)    then zerovec
  else op × = (shblockii b, vec u)vec:
        case b
         in (ref mat mm): mm×u
          , (ref diagmat dd): dd×u
          , (ref rectblockii rr):
             (index colp = colpartit of rr; loc int lwb:=1, upb := upb colp;
              to upb while  colp[lwb]<lwb u  do lwb+:=1  od;
              to upb-lwb while  colp[upb-1]≧upb u  do  upb-:=1  od;
              loc vec result := zerovec;
              for j from lwb to upb
               do vec usl = u?((colp[j-1]+1)//colp[j]);
                  for i to  1 upb subblocks of rr
                   do  result +:= (subblocks of rr)[i,j]×usl  od
               od; result
             )
          , (ref bandblockii bb):
             (index colp = colpartit of bb; loc int lwb:=1, upb := upb colp;
              to upb while  colp[lwb]<lwb u  do  lwb+:=1  od;
              to upb-lwb while  colp[upb-1]≧upb u  do  upb-:=1  od;
              loc vec result := zerovec;
              for k from  lwb subblocks of bb  to  upb subblocks of bb
               do ref[]blockii blockdiagk = (subblocks of bb)[k];
                  for j from  (lwb-k)max lwb blockdiagk
                          to  (upb-k)min upb blockdiagk
                   do result +:=
                          blockdiagk[j] × u?((colp[j+k-1]+1)//colp[j+k])
                   od
               od; result
             )
        esac;
      op × = (blockii b, vec u)vec:
         if vec result =
                if colsub of shift of b = 0     then  (block of b)×u
```

```
            else (block of b) × u[at  lwb u - colsub of shift of b]
              fi;
          rowsub of shift of b = 0  or  zero result
      then result
      else result[at lwb result + rowsub of shift of b]
        fi;
    (root of a)×u
fi;
```

program 5.2. Matrix-vector product in SPARSEII.

Obviously the recursive partition of an *sparmatii* can change if *shblockii*s will be shifted. With deletion and insertion of rows and columns partition lines may be deleted, others may be created. Suppose we have to insert k columns at column-position j_0 in a block B. These k columns are represented as an m×k block M (see fig. 5.6). The columns at the right of j_0 will be shifted to the right. If k is small, the partition of B will not change with the insertion, though partitions of subblocks of B can change. The row-partition of B will not change with the insertion. Ik k is large it may be worthwhile to create two new vertical partition lines in B at j_0 and (with the insertion) at j_0+k. We will present here only an heuristic. Let B have b_j columns in the j^{th} column-strip B_j (see fig. 5.6). Let $\min(B)=\min_j(b_j)$, $\max(B)=\max_j(b_j)$.

Case 1: $k \geq \min(B)$.

(i) If $c_1 \geq \min(B)$ and $c_2 \geq \min(B)$, then two new partition lines will be created.

(ii) If $c_1 \geq \min(B)$ and $c_2 < \min(B)$, then a new partition line will be created at j_0.

(iii) If $c_2 \geq \min(B)$ and $c_1 < \min(B)$, then a new partition line will be created at j_0+k.

(iv) If $c_1, c_2 < \min(B)$, then the row-partition of B is laid on M and the separate rowstrips of M are inserted in the corresponding direct subblocks of B.

Case 2: $k<\min(B)$.

(i) If $c_1 \geq c_2$ and $c_1 \geq \min(B)$, then one new partition line of B will be created at j_0.

(ii) If $c_2 \geq c_1$ and $c_2 \geq \min(B)$, then one new partition line of B will be created (with the insertion) at j_0+k.

(iii) If $c_1, c_2 < \min(B)$, then no new partition line will be created in B: the row-partition of B is laid on M and the direct subblocks of M are inserted in the corresponding direct subblocks of B.

The strategy attempts to avoid a huge increase in the ratio $\frac{\max(B)}{\min(B)}$.

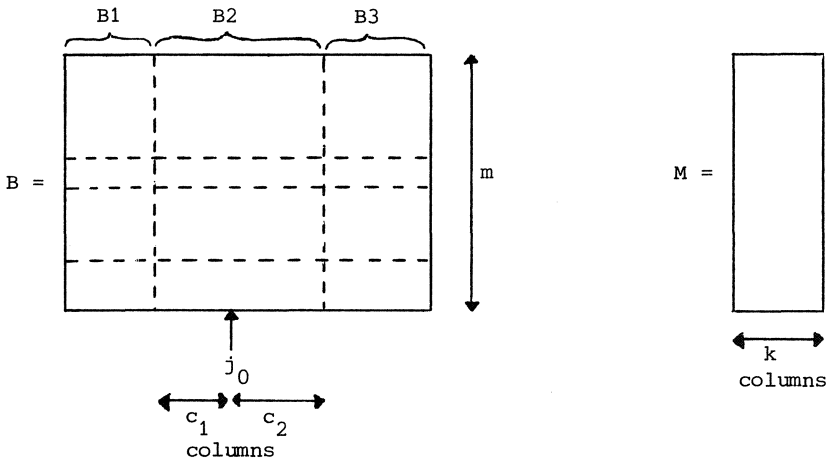


fig. 5.6. Insertion of columns and recursive partitions.

Observe that an *spmatii* is a tree data structure. Creating a new partition line in a *shblockii* means that the number of sons of this tree vertex increases or that a leaf is changed into an internal vertex. With the shift-feature more operations can be performed on subtrees, whereas all tree vertices must be accessed in case such an operation would be implemented in SPARSEI.

V.5.3 The SPARSEIII system.

V.5.3.1 Mode declarations in SPARSEIII.

Compared with SPARSEII, SPARSEIII contains only one new feature: the user can specify structurally equal subblocks in one *sparmat*. More precisely, he can specify structurally equal submatrices, and automatically a (new) recursive partition is derived such that the bounds of each of these submatrices equal the bounds of a subblock. Moreover, he can specify several finite sets of (structurally) equal submatrices. Only the concrete elements of one submatrix of a set of structurally equal submatrices will be stored actually. However, there are restrictions. Let

$$S_1 = \{D_1,\ldots D_{p_1}\},\quad S_2 = \{D_{p_1+1},\ldots,D_{p_2}\},\ \ldots,\quad S_k = \{D_{p_{k-1}+1},\ldots,D_{p_k}\} \tag{5.43}$$

be a family of sets of subdomains. Suppose the user specifies that all submatrices (corresponding to these subdomains) of an *spmatiii* a should be structurally equal. Then $\{S_1,\ldots,S_k\}$ must satisfy the following conditions:

(i) for all i $(1\leq i\leq p_k)$ $D_i \subseteq \text{bnds}(a)$, (5.44)

(ii) for all h $(1\leq h\leq k)$ and all i, j $(p_{h-1}+1\leq i,j\leq p_h)$, the submatrices of a corresponding to D_i and D_j have the same number of rows and columns, (5.45)

(iii) for all h $(1\leq h\leq k)$ and all i, j $(p_{h-1}+1\leq i,j\leq p_h,\ i\neq j)$ $D_i\cap D_j=\emptyset$, (5.46)

(iv) for all i, j $(1\leq i,j\leq p_k,\ i\neq j)$ we have

$$D_i\cap D_j = \emptyset,\quad D_i \not\subseteq D_j \quad \text{or} \quad D_j \not\subseteq D_i. \tag{5.47}$$

Unfortunately there are families of sets of subdomains satisfying (5.44)-(5.47), that do not allow a corresponding recursive partition. For details we refer to chapter VI. Equal submatrices as they occur in practice (see chapter II) are already related to a recursive partition so that these cases can be implemented with the SPARSEIII and SPARSEV systems. Obviously, these systems contain operators to detect whether a family of sets of subdomains satisfies (5.44)-(5.47) and whether a recursive partition can be found such that the bounds of each subdomain of the family equal the bounds of a subblock.

We will adapt the data structure of SPARSEII such that a number of internal vertices in the tree have a subtree in common. For all these internal vertices (except at most one) this common subtree is shifted, with a different

```
mode †blockiii = struct(pair shift , ref comblockiii cblock);
mode †comblockiii = struct(int refcount , intmem , scal scalmem
                           , shblockiii shblock
                           );
mode †shblockiii = union(ref mat , ref diagmat
                         , ref struct(mat m, trnspm)
                         , ref struct(diagmat m, trnspm)
                         , ref rectblockiii , ref bandblockiii
                         );
mode †rectblockiii = struct(index rowpartit , colpartit
                            , ref[,]blockiii subblocks
                            );
mode †bandblockiii = struct(index rowpartit , colpartit
                            , ref[]ref[]blockiii subblocks
                            );
mode spmatiii = struct(matbnds †bounds , ref shblockiii †root);
mode spsymiii = struct(trimmer †bounds , ref shblockiii †symroot);
mode sympatiii = struct(trimmer †bounds , ref shblockiii †root);
```

fig. 5.7. Mode declarations in SPARSEIII.

shift for all these internal vertices. The mode declarations for SPARSEIII are given in fig. 5.7.

Two *blockiii*s are structurally equal if and only if their *cblock*-fields are equal: these refer to the same *comblockiii*. The *refcount*-field in a *comblockiii* counts the number of references referring to this *comblockiii*. The fields *intmem* and *scalmem* are used for an efficient implementation of many algorithms (see V.5.3.2).

Because no advantage can be taken from equal blocks in the algorithm for the matrix-vector product, only the modes and selections from structured values in program 5.2 must be adapted in order to make it valid for SPARSEIII. Therefore, we will not present the SPARSEIII version here in detail.

An *spsymiii* can have structurally equal subblocks also. The user has to specify only the subblocks in the lower triangle and on the main block diagonal.

As for *sympatiiis* there is an extra restriction in order to make the data structure of *sympat*s not too complicated.

DEFINITION 5.13. Let $D=[m_1:n_1,m_2:n_2]$ be a subdomain. The transpose of D is the subdomain $D^T=[m_2:n_2,m_1:n_1]$.

The additional restrictions for *sympat*s on the family of sets of subdomains (5.43) are given by:

(v) for all h ($1\leq h\leq k$) and for all i, j ($p_{h-1}+1\leq i,j\leq p_h$) with $D_i=[m_1:n_1,m_2:n_2]$ and $D_j=[m_3:n_3,m_4:n_4]$ we have:
if $m_1>n_2$ then either $m_3>n_4$ or $m_3=m_4$ and $n_3=n_4$,
if $m_2>n_1$ then either $m_4>n_3$ or $m_3=m_4$ and $n_3=n_4$,

(vi) for all h ($1\leq h\leq k$) we have that if there are i, j ($p_{h-1}+1\leq i,j\leq p_h$) with $i\neq j$, then there is an $h'\neq h$ such that $D_i^T,D_j^T\in S_{h'}$.

Obviously SPARSEIII contains an operator to test whether these restrictions are valid and to test whether a family of sets of subdomains allow a symmetric recursive partition such that each subdomain is equal to the bounds of a subblock.

V.5.3.2 Operations in SPARSEIII.

In this section we will deal with the question how structurally equal blocks are involved in several operations. We will justify the fields *intmem* and *scalmem* in a *comblockiii* and show how these fields can be used to design algorithms for operations.

In the matrix-vector product no advantage can be taken from structurally equal blocks. The common subtrees have to be explored several times. In an operation like *sigmabs a* (returning the sum of the absolute values of the elements of *a*) each common subtree may be explored several times, but it will require less computation time, if a common subtree will be explored only once. In an operation like *a*×*<2.14* (multiply each *scal*-element of *a* with *2.14*) it is even forbidden that the multiplication will be performed on one common subtree more than once. Moreover, there is an operator *unshare* available to the user that changes the equality structure of a *sparmatiii*:

given a *blockiii*-operand B; after performing *unshare* applied to B, the equality structure of the *spmatiii* containing B must satisfy the following conditions:

(i) There is no subblock B' (B'≠B) such that B and B' are structurally equal, i.e., *refcount* *of* *cblock* *of* B has value 1. (5.48)

(ii) If B_1 and B_2 are structurally equal blocks and B_1 is a subblock of B, then B_2 is a subblock of B. (5.49)

Program 5.4 is an implementation of the operation that tests whether these two conditions are satisfied.

In all other operations the way in which common subtrees are involved, is similar to one of the above examples.

Now we will see how these operations can be implemented. As for the matrix-vector product we refer to the last sentence of V.5.3.1. As for *sigmabs* *a* and $a \times < n$ we can explain them simultaneously. In the routine-text of each SPARSEIII operator it is assumed that all *intmem*-fields have value 0. During the elaboration these values can change, but if the elaboration halts in the right way, these values are 0 once again. Thus in the routine-text this field can be used to record the number of times this *comblockiii* has been explored. If a *scal*-value obtained with the exploration of a *comblockiii*, must be used several times, then it can be assigned to the *scalmem*-field (see program 5.3). In the routine-text of *shared* an additional array has been declared, of which each element will contain information about one shared *comblockiii*. The *intmem*-field of a *comblockiii* (with *refcount*-field >1) contains, if it has been accessed, the index of the array element containing information about this *comblockiii*. Such an array element can contain information like the number of *blockiii*s with this *comblockiii* that are a subblock of B, a list of pointers to these *blockiii*s, a pointer to the involved *comblockiii*, etc. Such an array makes efficient algorithms possible (for example, see program 5.4).

Remark 5.2. The only additional information stored with each block is the *refcount*. For some operations global information (like "does this block contain a subblock with *refcount*>1?") seems appropriate. However, if the equality structure changes (for example with *unshare*) with the application of an operation on a part B of the *spmatiii*, it requires this global information to be adapted, even outside the part B (recall that the path from the root to a vertex in the directed graph is not necessarily uniquely determined). Thus an extensive search on the whole *spmatiii* is necessary or the data structure must be extended with still more information.

```
op sigmabs = (spmatiii a)scal:
  (op sigmabs = (shblockiii b)scal:
     case b
      in (ref mat mm): sigmabs mm
       , (ref diagmat dd): sigmabs dd
       , (ref rectblockiii rr):
         (loc scal sum := widen 0;
          for i to  1 upb subblocks of rr
           do for j to  2 upb subblocks of rr
               do  sum +:= sigmabs(cblock of(subblocks of rr)[i,j]) od
           od; sum
         )
       , (ref bandblockiii bb):
          (loc scal sum := widen 0;
           for i from  lwb subblocks of bb  to  upb subblocks of bb
            do ref[]blockiii blockdiagi = (subblocks of bb)[i];
               for j from  lwb blockdiagi  to  upb blockdiagi
                do  sum +:= sigmabs(cblock of blockdiagi[j])  od
           od; sum
          )
     esac;
   op sigmabs = (ref comblockiii b)scal:
     (scal sum = if intmem of b = 0  then  sigmabs(shblock of b)
                                     else  scalmem of b
                 fi;
       if refcount of b > 1
      then  if (intmem of b +:= 1) = 1       then scalmem of b := sum
            elif intmem of b = refcount of b  then intmem of b := 0
              fi
       fi; sum
     );
   sigmabs(root of a)
  );
```

program 5.3. Computing the sum of the absolute values of the elements of an *spmatiii* in SPARSEIII.

```
op shared = (ref blockiii b)bool:
  co returns false if b satisfies (5.48) and (5.49).
     Otherwise it returns true.                                    co
  (mode info = struct(int locrefcount, ref comblockiii this);
   op extend = (ref[]info row, info nextelement)ref[]info:
     (heap[lwb row : upb row +1]info newrow;
      newrow[upb newrow] := nextelement;
      newrow[lwb row : upb row  at  lwb row] := row;   newrow
     );      prio extend = 1;
   loc ref[]info count := heap[1:0]info;
   proc fillcount = (ref comblockiii cb)void:
       if intmem of cb > 0
     then co this block has already been explored; count this visit.  co
          locrefcount of count[intmem of cb] +:= 1
     elif co cb is not explored; we will do it now.                   co
          fillcnt(shblock of cb); refcount of cb > 1
     then co cb itself is a common block; record this first visit.    co
          count := count extend info((1,cb));
          intmem of cb := upb count
       fi;
   proc fillcnt = (shblockiii b)void:
     case b
      in (ref rectblockiii rr):
          for i to  1 upb subblocks of rr
           do for j to  2 upb subblocks of rr
                do  fillcount(cblock of(subblocks of rr)[i,j])  od
           od
       , (ref bandblockiii bb):
          for i from  lwb subblocks of bb  to  upb subblocks of bb
           do for j from lwb(subblocks of bb)[i]  to  upb(subblocks of bb)[i]
                do  fillcount(cblock of(subblocks of bb)[i][j])  od
           od
     esac;
```

```
fillcnt(shblock of cblock of b);
loc int i := 1;
to upb count while locrefcount of count[i] = refcount of this of count[i]
                do  i+:=1  od;
for j to upb count  do  intmem of this of count[j] := 0  od;
upb count ≧ i
);
```

program 5.4. Test whether a block B contains or is a subblock that is shared with other blocks outside B.

Remark 5.3. Instead of generating an additional array in the routine-text of an operator containing information about structurally equal blocks, a linked list or even a search tree can be generated to avoid that the applications of the operator *extend* requires too much time.

V.5.3.3 The occurrence of side effects in SPARSEIII.

Obviously, structurally equal blocks give rise to the occurrence of side effects. A value can only be assigned to a (virtual or concrete) *scal*-element of one of two structurally equal subblocks if it is assigned also to a *scal*-element in the other subblock. In SPARSEIII and TORRIX-SPARSE this "multiple" assignation will automatically be performed. However, in other operations applied to a subblock that is structurally equal to another block, it would be a matter of bad design to allow the occurrence of all kinds of uncontrolled side effects. The actual change in the sparse matrix could have no counterpart in linear algebra. For example, the exchange of two rows could lead to a bunch of exchanges of or/and assignations to rows of many subblocks of the matrix. Therefore, for many operators it will be checked at runtime whether it can be applied, depending on the equality structure. Important is the concept of an active domain of an application of an operation. It consists of all virtual and concrete elements of the matrix that can be changed or (if the operator involves only changes in the recursive partition) the virtual and concrete elements of the smallest subblock in which all changes will occur. For example, the active domain of a permutation of row h and row k of an *spmatiii a* consists of

$\{(i,j)$: i=h or i=k and 2 *lwb bnds* $a \leq j \leq 2$ *upb bnds* $a\}$

if 1 *lwb bnds* $a \leq h,k \leq 1$ *upb bnds* a.

Unfortunately we have to make a distinction with regard to the occurrence of side effects between operators in which blocks may be shifted and operators in which blocks will not be shifted. As for the occurrence of side effects in applications of the latter operators, we have the following rules: Let such an application have an active domain E of an *spmatiii* a; suppose a has an equality structure as specified in (5.43);

(i) for all i $(1 \leq i \leq p_k)$ $D_i \cap E = \emptyset$. (5.50)
Then after the elaboration of the operation, there is no change in the equality structure of a,

(ii) there is an i $(1 \leq i \leq p_k)$ such that $E \subseteq D_i$. (5.51)
Then after the elaboration, the equality structure of a has not been changed. Thus a change in the subblock B specified by D_i is a change in all subblocks structurally equal to B,

(iii) there are h $(1 \leq h \leq k)$, i and j $(p_{h-1}+1 \leq i,j \leq p_h)$ such that
$$D_j \cap E = \emptyset \text{ and } D_i \subsetneq E. \quad (5.52)$$
Then after the elaboration, D_i and D_j are disconnected and the block specified by D_j has not been changed. Whether two subblocks with bounds D_i and D_j both in E will remain structurally equal to each other, depends on the total-array2 of a and the other operands,

(iv) there is an i $(1 \leq i \leq p_k)$ with $D_i \setminus E \neq \emptyset$, $E \setminus D_i \neq \emptyset$ and $E \neq D_i$. (5.53)
Then a fatal error will be reported and the elaboration of the user program will halt.

Observe that a can have an equality structure such that (5.51) and (5.53) occur simultaneously.

Clearly (5.50)-(5.53) must be adapted in order to hold for *spsyms* and *sympats*.

As a consequence of the rules mentioned above, many operators change the equality structure of its *sparmat*-operand. Subblocks that are structurally equal before elaboration are possibly not structurally equal after the elaboration : they are disconnected. For instance, in $a+<b$ structurally equal subblocks in a will be disconnected if the corresponding submatrices in b are not structurally equal and if these submatrices in b have concrete elements. It will not happen that subblocks in a are made structurally equal by an application $a+<b$. For all operations (except <:=, *assigndel*, *plusabdel*

and a number of generating operators) holds that no subblocks will be made structurally equal.

In generating operators the result *sparmat* will have as many structurally equal blocks as possible. However, which blocks will be structurally equal depends on only four aspects:

(i) two blocks will not be structurally equal if the corresponding subblocks in one of the operands are not structurally equal,

(ii) the total-array2 of the result *sparmat*,

(iii) the set of virtual elements of the operands,

(iv) will the recursive partition of the result *sparmat* be dynamically determined or is it specified by the user?

How the equality structure changes if an operator is applied in which a shift can be performed, must be clear from the description in a table of operators.

Conclusion. The increase in administration overhead to implement SPARSEIII (compared with SPARSEII) is only linear in the number of stored subblocks. We have tried to control the occurrence of side effects related to structurally equal blocks. Several operations disconnect structurally equal blocks.

V.5.4 The SPARSEIV system.

V.5.4.1 Mode declarations in SPARSEIV.

In the SPARSEIV system a slicing mechanism is implemented. There is no possibility to specify structurally equal blocks. In V.4 we have explained what a slicing mechanism provides and what side effects must occur if operators are applied to slices, matrices or vectors (see (5.34)). Because of this requirement it is not allowed that an operation like *a!(i?j)* returns a *ref scal* to the $(i,j)^{th}$ element (that will be (made) concrete) of *a*. Suppose it does, then the following program part would be a violation of (5.34).

```
ref scal aij = a!(i?j); aij := widen 1;        (1)
a <:= zerospmativ;                             (2)
print(a?(i?j));                                (3)
print(aij)                                     (4)
```

With the tree terminology, the following happens during the elaboration: with line 1 a pointer is made to refer to (a part of) a leaf of *a*; with line

2, *a* is replaced by the tree of *zerospmativ*, but this is not recorded in *aij* with *a?(i?j)*, the new tree of *a* is searched for the value of the $(i,j)^{th}$ element of *a*, returning value 0; with line 4, *aij* still refers to the leaf of the former tree of *a*, thus printing value 1.

Another implementation of *a!(i?j)* could be to store the values *i* and *j* together with a pointer to the root of *a*. Each time *aij* is used, *a* would be searched for its $(i,j)^{th}$ element. However, now the efficiency requirement (5.24) is violated.

Our solution is a combination of these rejected implementations: we store with *a!(i?j)* the slice information (containing e.g. the values *i* and *j* and a pointer to the root of *a*) together with a *ref scal* to the concrete $(i,j)^{th}$ element of *a*. Let the following program part be given (with *a*, *b* and *c* of the mode *spmativ* and *i*, *j* of the mode *int*):

```
scalel aij = a!(i?j);                    (1)
a +< b;                                  (2)
a +< c;                                  (3)
print(a?(i?j));                          (4)
print(aij);                              (5)
aij <:= widen 0                          (6)
```

With our implementation this program part will be elaborated as follows. In line 1 a *scalel* is ascribed to the identifier *aij*, containing slice information as indicated above; in line 2 and 3, *a* may change dramatically; as for *aij*, these changes are not important until line 5 will be elaborated. Thus, it does not matter whether, after the elaboration of line 3, *aij* actually refers to the concrete $(i,j)^{th}$ element of *a*. We will use the concept 'up to date' ('out of date') for a slice at some stage of the elaboration of a user program, if this slice actually refers (does not refer) to the involved element(s) of the original *sparmativ* or *spvec*. As a consequence of (5.32), a slice must be checked automatically whether it is up to date, before it will be used. If a slice turns out to be out of date, then it must be updated. Therefore, line 5 is elaborated behind the screens as

```
(if not uptodate aij then update aij fi;
 scal value = c scalar value of aij c;
 print(value)
)
```

The uptodate test, possibly followed by an update must be performed before every operation using a slice. Line 6 will be elaborated as

```
(if not uptodate aij then update aij fi;
 value of aij := widen 0
)
```

Because of (5.24) the elaboration of *uptodate aij* in line 6 must return *true*, requiring a small amount of computation time.
Observe that the application of an operation of type (5.23) does not necessarily make slices out of date. Slices can be made out of date only by a change in the sense of (5.20) and (5.21). These kinds of changes we will call *ref*-changes.

The mode declarations of SPARSEIV are given in fig. 5.8. Here we briefly mention for what purposes several fields and modes are used (see also fig. 5.9):

(i) *eventcnt* and *allcnt* in a *dateblockiv* will be used to record *ref*-changes and are needed in the uptodate test.

(ii) an *mslinfo* is used to store slice infomation about a submatrix slice: *slbnds* denotes the indices in the original matrix of the elements contained in the slice, *birth* is used in the uptodate test, *origin* refers to the original matrix, *roottype* denotes the type of the original matrix: 0 for an *spmativ*, 1 for an *spsymiv* and -1 for a *sympativ*.

(iii) an *mroot* contains a pointer to the slice information which is *nil* in case this *mroot* belongs to an original *sparmativ* and *root* refers to the tree or subtree data structure.

(iv) a *ref[]ref scal* in a *vblockiv* will only be used for a "row" or "column" slice of a *diagmat*.

(v) *eventbirth* and *allcnt* in a *dvblockiv* have the same purpose as *eventcnt* and *allcnt* in a *dateblockiv*.

(vi) a *vslinfom* contains slice information of a vector slice of a *sparmativ*: *sltype* denotes the slice type (0 for rows, 1 for columns and 2 for diagonals), *ind* denotes the ind^{th} row (column, diagonal); the other fields are clear from *mslinfom*.

(vii) a *vslinfov* contains slice information of a subvector slice of an *spveciv*

(viii) an *scslinfom* and an *scslinfov* contain slice information of element slices of *sparmativ*s and *spvecivs*.

Modes for sparse matrices:

```
mode †shblockiv = union(ref mat , ref diagmat
                        , ref struct(mat m, trnspm)
                        , ref struct(diagmat m, trnspm)
                        , ref rectblockiv , ref bandblockiv
                        );
mode †dateblockiv = struct(int eventcnt , bool allcnt , shblockiv shblock);
mode †blockiv = struct(pair shift , ref dateblockiv dblock);
mode †rectblockiv = struct(index rowpartit , colpartit
                           , ref[,]blockiv subblocks
                           );
mode †bandblockiv = struct(index rowpartit , colpartit
                           , ref[]ref[]blockiv subblocks
                           );
mode †mslinfo = struct(matbnds slbnds , int birth , roottype
                       , ref mroot origin
                       );
mode †mroot = struct(ref mslinfo slicinfo , ref blockiv root);
mode spmativ = struct(matbnds †bounds , ref mroot †root);
mode spsymiv = struct(trimmer †bounds , ref mroot †symroot);
mode sympativ = struct(trimmer †bounds , ref mroot †root);
```

Modes for sparse vectors:

```
mode †vblockiv = union(ref vec , ref[]ref scal , ref vblockrowiv);
mode †dvblockiv = struct(int eventbirth , bool allcnt , vblockiv vblock);
mode †vblockrowiv = struct(index partit , ref[]dvblockiv subblocks);
mode †vslinfom = struct(int sltype , ind , trimmer slbnds
                        , int roottype , ref mroot origin
                        );
mode †vslinfov = struct(trimmer slbnds , ref vroot origin);
mode †vroot = struct(int birth , union(ref vslinfom , ref vslinfov) slicinfo
                     , ref dvblockiv root
                     );
mode spveciv = struct(trimmer †bounds , ref vroot †root);
```

Modes for element slices:

```
mode †scslinfom = struct(pair index , ref mroot origin , int roottype);
mode †scslinfov = struct(int ind , ref vroot origin);
mode scalel = struct(ref int †birth
                     , union(ref scslinfom , ref scslinfov) †slicinfo
                     , ref scal †elem
                     );
```

fig. 5.8. Mode declarations in SPARSEIV.

In the next section we will deal with element and submatrix slices and the uptodate test on these slices. In V.5.4.3 we will deal with vector slices of *sparmat*s, slices of *spveciv*s and their uptodate tests.

V.5.4.2 The uptodate test on element and submatrix slices.

An element slice consists of slice information and a pointer to a (part of a) leaf of the tree T of a *sparmativ*. A submatrix slice consists of slice information and a pointer to the smallest subtree of T containing all (virtual and concrete) elements of this slice. Before we can explain the algorithm for the uptodate test, we have to say how *ref*-changes are recorded in the tree. For this we need a distinction between two kinds of *ref*-changes. A *ref*-change is a leaf change if a *mat*, *diagmat* or a *vec* of a *diagmat* is replaced by another *mat*, *diagmat* or *vec*. Any other *ref*-change is a subtree change. Leaf changes are recorded in the *eventcnt*s, subtree changes are recorded in *allcnt*s and *eventcnt*s. If a leaf change occurs in leaf X, then the *eventcnt* of the root R of the tree T is raised with 1 and the *eventcnt*s of all other vertices on the search path to X are set to the *eventcnt* of R. If an operation of type (5.23) constitutes several leaf changes, then the *eventcnt* of R is raised only once. If no subtree change is performed on T, then the *eventcnt*s of the vertices on the search path of any leaf X form a non-increasing sequence of integers.
Subtree changes occur with changing the recursive partition or a shift or an assignation of a subtree. A subtree change could be recorded in the right way by adapting all *eventcnt*s in all vertices of this subtree. However, this would take too much time and therefore, each vertex N contains a boolean

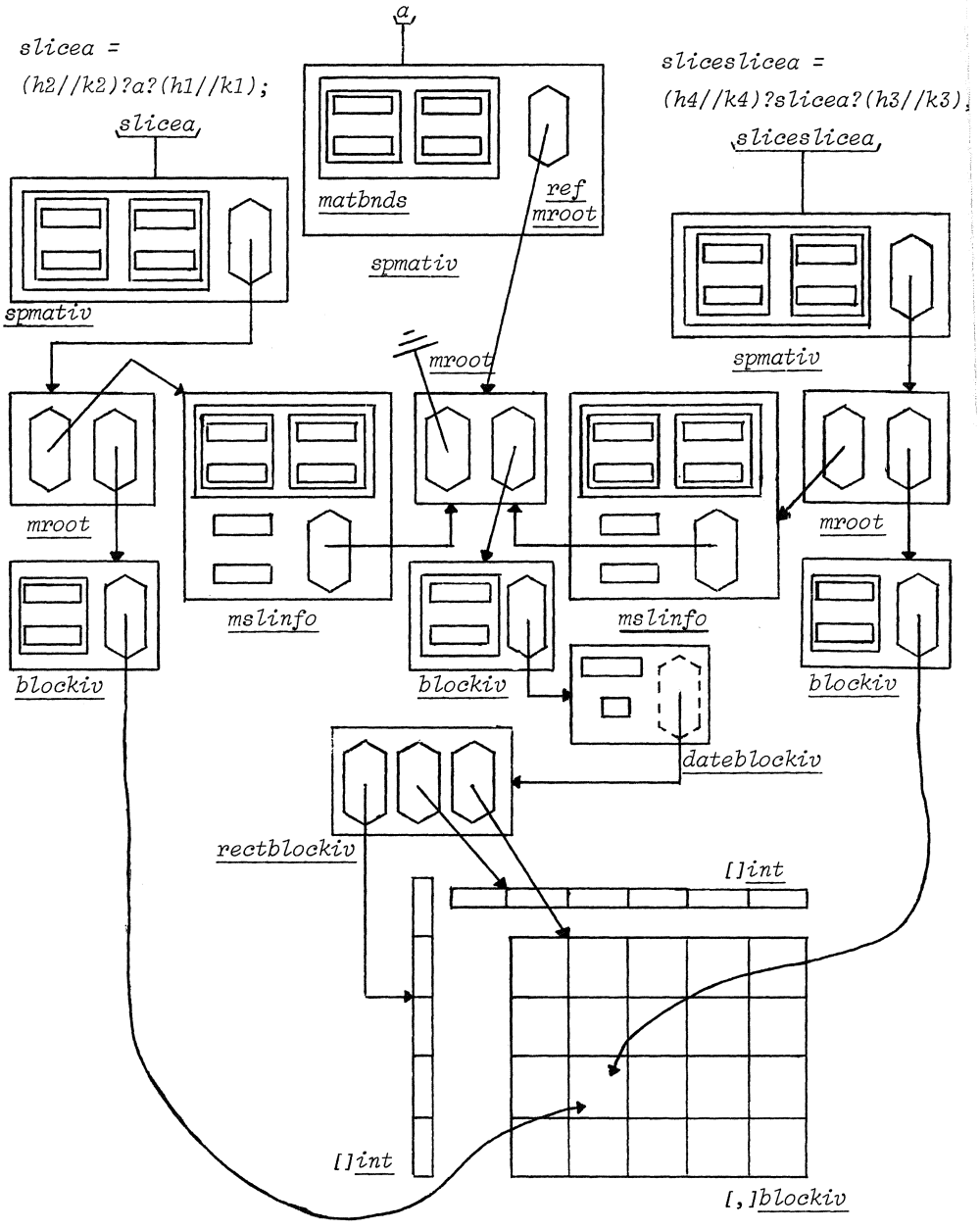


fig. 5.9. Data structure of an *spmativ a* and two of its submatrix slices

Straight lines are ALGOL 68 references, a curve denotes an ALGOL 68 referen
that points to some node in this tree data structure; values of a *union* mod
are dotted.

value *allcnt* with the following meaning: if *allcnt*(N) has value *true* then the *eventcnt*s of all descendants of N should be considered to have value *eventcnt*(N), though actually they may have another value.

Proposition 5.2. *Let* R *be the root of a tree* T *of a sparmativ and let* X *be any vertex in* T. *Let* $(R=X_0,X_1,\ldots,X_n=X)$ *be the search path of* X. *If* *allcnt*(X_i) $(0\leq i\leq n)$ *has value false, then* $(eventcnt(X_i))_{0\leq i\leq n}$ *is a non-increasing sequence of integers.*

If an element or submatrix slice *sl* is created, its *birth*-field obtains the value of *eventcnt* of R. As long as *eventcnt*(R) ≦ *birth of sl*, the slice *sl* is up to date. If *eventcnt*(R) > *birth of sl*, then in the uptodate test there will be a search along the path from R to the root of the subtree containing *sl*, for a vertex N with

eventcnt(N) ≦ *birth of sl*

taking into account the value of the *allcnt*-fields. If such an N exists, then the *ref*-changes recorded in R have not occurred in the subtree N containing *sl*. Thus *sl* is up to date. If such an N does not exist, *sl* will be created once again before use to ensure that it is up to date at the moment it will be used. With each uptodate test (and update) applied to *sl*, the *birth*-field of *sl* is assigned the current value of *eventcnt*(R). Program 5.5 contains an ALGOL 68 implementation of the uptodate test. In order to make this program valid, *ref*-changes arising with the application of an operator on a slice must be recorded in all vertices on the path from the root R to the vertex of this slice.
Observe that we do not record the "birthdates" of slices but actually the last time that a slice turns out to be or had been made up to date.

In the remainder of this section we will concentrate on the operations *a<:=b* and *a assigndel b* with respect to the uptodate question. *a* and *b* can be slices of other *sparmativ*s A and B (see fig. 5.10). In *a<:=b* copies of the *mat*s and *diagmat*s of *b* replace *mat*s and *diagmat*s of *a*. Thus all leaves of the (sub)tree of *a* must be explored at least once, and all *ref*-changes can be recorded with *eventcnt*s without use of the *allcnt*s. In *a assigndel b*, however, subtrees of *a* may be replaced by subtrees deleted from *b*. Subtree changes, therefore, can occur in both *a* and *b* (see fig. 5.10). In case the slices *a* and *b* coincide with two blocks only the *eventcnt*s on the paths from

```
op unshift = (trimmer t1, int h)trimmer:
   (lower of t1 -h)//(upper of t1 -h);      prio unshift = 6;
op unshift = (matbnds bnds, pair shift)matbnds:
   (rowbnds of bnds unshift rowsub of shift
   ,colbnds of bnds unshift colsub of shift
   );
proc lowsearch = (index row, int here)int:
   co assumes that  lwb row < upb row,  row[i]>row[i-1] for all
      lwb row <i≦ upb row,  row[lwb row]<here≦row[upb row].
      returns the largest h such that row[h-1]<here                    co
   (loc int h := lwb row +1;
    to upb row - lwb row -1  while  row[h]<here  do  h+:=1  od; h
   );
op uptodate = (spmativ a)bool:
   co returns true if, since the last use of a, no ref-change has
      occurred affecting a.                                            co
     if ref mslinfo(slicinfo of root of a) is nil
   then co  a is an original sparmativ  co      true
   else ref mslinfo slicinfo = slicinfo of root of a;
        int birthdate = birth of slicinfo, type = roottype of sliciinfo,
        loc matbnds slbounds := slbnds of slicinfo,
        loc ref dateblockiv pntr := dblock of root of origin of slicinfo,
        loc bool result := true;
        case type+2
         in co enclosed-clause for the case a is a slice of a sympativ co
            skip
          , (while if eventcnt of pntr <= birthdate     then false
                elif allcnt of pntr                     then result:=false
                else
```

```
case shblock of pntr
 in (ref mat): result:=false
  , (ref diagmat): result:=false
  , (ref rectblockiv rr):
       if index rowp = rowpartit of rr;
          int h = lowsearch(rowp, lower of rowbnds of slbounds);
          rowp[h] < upper of rowbnds of slbounds
     then result:=false
     elif index colp = colpartit of rr;
          int k = lowsearch(colp, lower of colbnds of slbounds);
          colp[k] < upper of colbnds of slbounds
     then result:=false
     else blockiv newbl = (subblocks of rr)[h,k];
          slbounds := slbounds unshift shift of newbl;
          pntr := dblock of newbl;   true
       fi
  , (ref bandblockiv bb):
       if index rowp = rowpartit of bb;
          int h = lowsearch(rowp, lower of rowbnds of slbounds);
          rowp[h] < upper of rowbnds of slbounds
     then result:=false
     elif index colp = colpartit of bb;
          int k = lowsearch(colp, lower of colbnds of slbounds);
          colp[k] < upper of colbnds of slbounds
     then result:=false
     elif k-h < lwb subblocks of bb  or  k-h > upb subblocks of bb
     then pntr:=nil;  result:=false
     elif ref[]blockiv blockdiag = (subblocks of bb)[k-h];
          h < lwb blockdiag  or  h > upb blockdiag
     then pntr:=nil; result:=false
     else slbounds := slbounds unshift shift of blockdiag[h];
          pntr := dblock of blockdiag[h];   true
       fi
esac
```

```
                fi
            do skip od;
              if result=true
            then birth of slicinfo :=
                        eventcnt of dblock of root of origin of slicinfo;
                 result
            elif ref dateblockiv(pntr) is dblock of root of root of a
            then birth of slicinfo :=
                        eventcnt of dblock of root of origin of slicinfo;
                 true
            else result
              fi
          )
        , co  enclosed-clause for the case a is a slice of an spsymiv co
          skip
      esac
  fi;
```

program 5.5. The uptodate test of an *spmativ* that is a slice of an *spmativ*.

these blocks to r(A) and r(B) must be adapted as well as *allcnt* of these two blocks. However, if a and b do not coincide with two blocks (i.e., the bounds of a and the block to which a refers, are different) all *eventcnt*s and *allcnt*s of all blocks in A that are partly part of the slice a, must be adapted also. The same holds for b. The subtree of the block to which a refer must be split in order to delete the slice a. Because sons of a tree vertex are stored in rectangular or diagonal arrays, this splitting may involve much more leaves than those two denoted in fig. 5.10. Thus, the time to elaborate a *assigndel* b depends on the number of blocks of the (sub)tree of a (and b) of which the bounds have a non-empty intersection with but are not included in the bounds of a.

Remark 5.4. The uptodate test *uptodate* sometimes returns *false* while the slice is actually up to date. However, this is not an error because *uptodate* is not available to the user and occurs only in the choice-clause

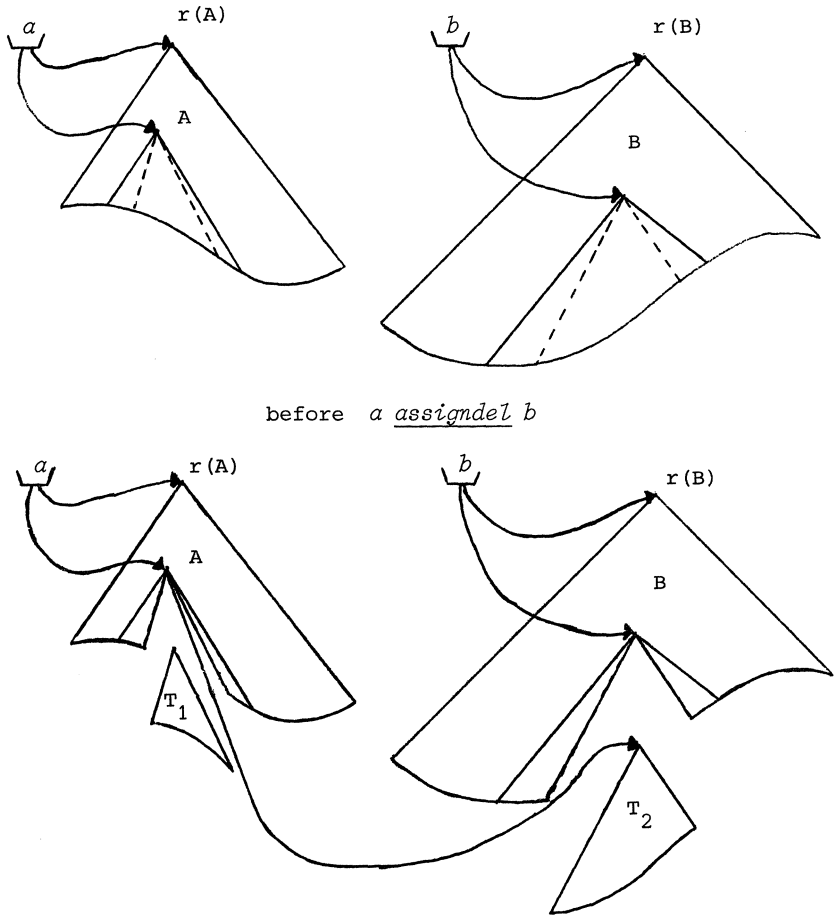


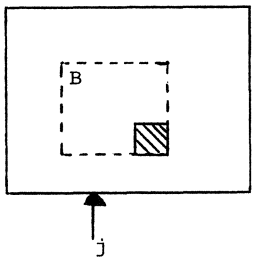
fig. 5.10 The operation *a assigndel b* in case *a* and *b* are slices.

if not uptodate slice then update slice fi

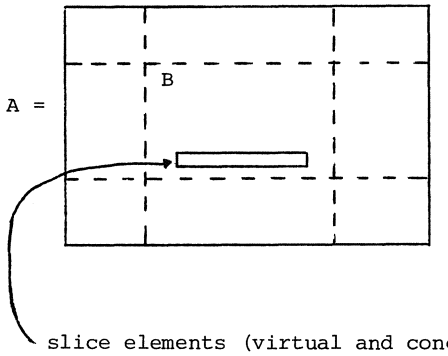
Therefore, after the elaboration of this clause, the slice certainly is up to date. Moreover, it is better to have a fast operator *uptodate* that in a few cases returns *false* while the slice is up to date, than to have an unefficient *uptodate* returning always the right answer.

V.5.4.3 Sparse vectors.

Row, column and diagonal slices could be implemented in a way similar to submatrix slices: slice information, a pointer to the original matrix, a birth date and a pointer to the smallest subtree containing all (virtual and concrete) elements of the slice. However, in many cases this smallest subtree would be the root of the tree of the original matrix. Thus for each use of the slice a search must be done in the *sparmativ* to see which of its sons contain *scal*-elements of the slice, and objective 5.5.5 can hardly be met. Therefore, we have chosen another implementation of vector slices of a *sparmativ*: a new tree will be built of which the leaves (*vecs*) are part of the leaves of the tree of the original *sparmativ*. In principle each block of the *sparmativ* containing elements of the slice corresponds to an internal vertex in the slice tree. This can lead to a degenerated tree for the slice in two ways (see fig. 5.11):



block B contains only one small *mat m* with j < 2 *lwb m*. The block in the tree of the j^{th} column slice would contain virtual elements only.



slice elements (virtual and concrete)

The slice is contained in the subblock B of A. The vertex in the slice tree corresponding to block A would have only one son that corresponds to block B.

fig. 5.11. Slices that give rise to degenerated slice trees.

(i) there are leaves in the slice tree that do not contain a concrete *scal*-element,

(ii) there are vertices in the slice tree with only one child.

Thus we can implement a more efficient data structure for vector slices by reducing the slice tree as explained above in the following way:

(i) delete all leaves without any concrete *scal*-element, (5.54)

(ii) if a vertex p has only one son q (with a son r), then r is made a son of p and q is deleted. (5.55)

Each vertex in the slice tree contains an event counter (*eventbirth*). It will be used for the uptodate test and has the same purposes as both *birth* and *eventcnt*s in *sparmativ*s. When a vector slice is created the *eventbirth*s in the slice tree obtain the value of *eventcnt* of the root of the original *sparmativ*. Observe that the shift feature has not been implemented for sparse vectors. Shifts are not needed for vector slices or slices of vectors. An *spveciv* can be sliced. If u is a vector slice of a *sparmativ* a, its slices are considered to be slices of a. If u is not a slice, its slices are implemented in a way similar to submatrix slices: slice information, a birth date, a pointer to the root of u and a pointer to the smallest subtree containing the slice. The *allcnt*-fields in vertices of the tree of an *spveciv* u are only used in case u is not a slice of a *sparmativ*. As for operators of type (5.23) applied to an *spveciv* u that is a slice, these are actually not applied to u but to the original *sparmativ* in the right way. Thus, u can be made out of date. This does not matter for the user, because u will be made up to date if a query or generating operator will be applied to u.

As for the uptodate test for vector slices we can be rather short. In fact a number of search paths in the tree of the original *sparmativ* must be compared with the corresponding (possibly shorter) search paths in the slice tree. The comparison must be done between *eventcnt* and *eventbirth* (proposition 5.2 also holds for *spveciv*s (if terms concerning *sparmativ*s are replaced by the corresponding terms for *spveciv*s)). Observe that updating an out-of-date vector slice does not necessarily result in a new creation of the whole tree of the slice. Only those subtrees that are out of date should be renewed.

As for the efficiency of the use of slices we have the following result:

Proposition 5.3.

(i) Updating a slice will not require more time than creating this slice once again.

(ii) Let a be a sparmat in SPARSEII and b an original sparmat in SPARSEIV (with the same recursive partition, bounds, concrete scal-elements, total-array2 as a). If an operation applied to a requires time t, then this same operation applied to b requires time at most $c.t$ for a constant $c \in \mathbb{R}$ independent of a, b and the applied operation.

The only additional work in operations in SPARSEIV applied to an original *sparmativ* (compared with SPARSEIV) consists of an extra dereferencing during the visit of a block and of setting *allcnt*s and *eventcnt*s.

Remark 5.5. With the implementation of slices a new problem arises. We will illustrate it with an example. SPARSEIV contains an operator =:= to exchange *scal*-values of *spveciv*s. These two *spveciv*s can be slices, say, a column and a diagonal slice. In writing the routine-text for =:=, the system manage must be careful not to change the value of an element before it is used or preserved. Fortunately, checking whether *sparmativ*s, *spveciv*s or *scalel*s contain a common element can be done rather efficiently. This can be decided from the slice information.
For the user an operator *slalias* is available: *sl1 slalias sl2* returns the smallest slice of *sl1* (if *sl1* is not a *scalel*) containing all elements of *sl1* (virtual or concrete) that are element of *sl2* also. If no element of *sl1* is element of *sl2*, then *zerosparmat* (or *zerospvec*) will be returned. If *sl1* is a *scalel* a boolean will be returned.

Remark 5.6. Unfortunately design objective 5.5.4 concerning garbage collecti is not fully met in SPARSEIV. Because of the slicing mechanism in SPARSEIV the user can have access to a (part of a) sparse matrix storage tree T not only by the root but by any other vertex of T. On the other hand, if slices have been created in the user program, there are pointers from these slices to the root of T. Thus, the whole tree T remains in memory even if the user can access a very small part of T only. By means of the system routines of SPARSEIV the user is prevented from accessing the other parts of T.
This situation can occur if slices of a *sparmat* a are assigned to variables declared in a range containing the range of a. After leaving the range of a the user can still access the slices of a, though he cannot access a itself

(see example 5.2). The whole storage tree for *a* remains allocated. However, if before leaving the range of *a*, the user assigns *zerospsym* (with *assigndel*) to the parts of *a* not being one of the slices, then only a small amount of storage will remain allocated which the user cannot access.

Example 5.2.

```
loc spvec ai;
begin spmat a = genspmat(1000,1000);
      .......;
      ai := a?i
end;
.......
```

Conclusion. A slicing mechanism can be implemented such that the problem of side effects is solvable in the sense of (5.34). The implementation in SPARSEIV requires only a modest increase in storage. Also objective 5.5.5 is satisfied. Unfortunately, the user must take care of garbage collection but does not need to program a garbage collector.

V.5.5 The SPARSEV system.

The SPARSEV system contains both a slicing mechanism and the feature of structurally equal blocks. The data structures in SPARSEIII and SPARSEIV seem difficult to combine. In the former each vertex of the directed graph obtained only local information (i.e., information about this very vertex). Storing "global" information was rejected because it could cause that this information had to be adapted in the vertices on all paths from the root to a given vertex. In the SPARSEIV data structure global information was added to every vertex in order to have a fast uptodate test.
Suppose we add an *eventcnt* and *allcnt* to all vertices of the SPARSEIII data structure. What kind of information should be stored with each slice and what is the algorithm for the uptodate test? To answer these two questions we first have to explain how *ref*-changes in a *sparmatv* are recorded. If a *ref*-change occurs at a vertex X in the directed graph of a *sparmatv* *a*, then X has been found by searching along the search path π of an element of the matrix. Observe that there may be a submatrix slice *sl* in the user program pointing to X, while the search path of *sl* is unequal to π. The *ref*-change in X will be recorded in the vertices on π only. Thus proposition 5.2 is not valid in

SPARSEV. However, a weaker form of this proposition does hold in SPARSEV:

Proposition 5.4. *Let* R *be the root of the directed graph* T *of a* sparmatv *and let* X *be a vertex of* T. *Let* $(R=X_0,X_1,\ldots,X_n=X)$ *be a path from* R *to* X. *Let* $X_{p_1},X_{p_2},\ldots,X_{p_k}$ $(p_i<p_{i+1}, 1\leq i\leq k-1)$ *be the vertices of this path with refcount*>1 *and let* $p_0=0$. (5.56)

If allcnt(X_i) *has value false* $(0\leq i\leq n)$, *then eventcnt*$(X_j))_{p_i\leq j\leq p_{i+1}-1}$ *is a non-increasing sequence of integers for each* i $(0\leq i\leq k-1)$.

No problems arise with *allcnt*s on other paths from R to X with value *true*. Such an *allcnt* does not exist because of (5.52) except in case it is set *true* with a shift. However, a shift at another vertex than $X_0,\ldots,X_n$ does not involve the blocks of the matrix corresponding to $X_0,\ldots,X_n$. Thus we have the following fact:

Proposition 5.5. *Let* T *be the directed graph of a* sparmatv. *Let* R *be the root of* T *and sl a slice of* T *pointing at vertex* s. *Let* s' *be the vertex of* T *to which sl has to point if sl is up to date. Let* b *be the value of birth of sl and* $R=X_0,X_1,\ldots,X_n=s'$ *be the search path (according to the slice sl). If there is an* i $(1\leq i\leq n)$ *such that*

eventcnt$(X_i)\leq b$ and
allcnt(X_j)=*false* $(1\leq j\leq \min(i,n-1))$ and
eventcnt$(X_{p_j})\leq b$ for all j with $p_j>i$ and p_j as in (5.56) (5.57)

then sl is up to date (i.e., s=s').

Each operator of type (5.23) affecting *sl* could only be performed along a path that contained X_i with i as in (5.56) or one of the X_{p_j} (with $p_j>i$). Thus if a *ref*-change has affected *sl*, one of these vertices has *eventcnt*>b. With proposition 5.5 we arrive at the data structure for submatrix and element slices of a *sparmatv*: the corresponding data structure in SPARSEIV, extended with a list of pointers to the vertices $X_{p_1},\ldots,X_{p_k}$ as in (5.56). The algorithm for the operator *uptodate* for submatrix and element slices should now be clear:

algorithm 5.1. UPTODATE (in SPARSEV) for an element or submatrix slice *sl*.

```
if initialize pointer pntr := root of origin of sl; int b = birth(sl);
   eventcnt(pntr)≦b
then true
else while while   if eventcnt(pntr)≦b or allcnt(pntr)=true   then false
                   else pntr has a direct subblock b' with
                        slbounds(sl) ⊆ bnds(b')          co pntr isnt s' co
                   fi
           do  pntr := b'  od;
             if pntr has not a direct subblock b' with
                slbounds(sl) ⊆ bnds(b')                  co pntr is s' co
           then false
           elif allcnt(pntr)=true and eventcnt(pntr)>b
           then false
           elif there is no pointer y in the additional list of sl with
                eventcnt(y)>b and slbounds(sl) ⊊ bnds(y) ⊊ bnds(pntr)
           then false
           else let y have maximal bounds; pntr:=y; true
             fi
    do skip od;
      if pntr has not a direct subblock b' with slbounds(sl) ⊆ bnds(b')
    then  co pntr is s' co   pntr is s
    elif eventcnt(pntr)>b          then false
    else  co eventcnt(pntr)≦b co        true
      fi
fi
```

Vector slices of a *sparmatv* can be implemented as an *spvec* of SPARSEIV together with a tree of pointer nodes. Each node of this additional tree contains two pointers: one points at a vertex in the original tree (that contains, let us say, part [p:q] of the vector slice) with *refcount*>1 and the other at the vertex (block) in the slice tree with bounds [p:q]. Because of the reduction (5.54) and (5.55) two pointers in two nodes of this additional tree may refer to one vertex in the slice tree. As a consequence, this additional tree may require more space than the slice tree itself. Observe that we have implemented neither a feature of structurally equal blocks nor a

shift-feature for *spvecv*s. By its very nature a vector slice of a *sparmatv* can have equal leaves. All this yields the mode declarations for SPARSEV as given in fig. 5.12.

Modes for sparse matrices:

```
mode †shblockv = union(ref mat , ref diagmat
                       , ref struct(mat m, trnspm)
                       , ref struct(diagmat m, trnspm)
                       , ref rectblockv , ref bandblockv
                       );
mode †comblockv = struct(bool allcnt , int eventcnt , refcount , intmem
                         , scal scalmem , shblockv shblock
                         );
mode †blockv = struct(pair shift , ref comblockv cblock);
mode †rectblockv = struct(index rowpartit , colpartit
                          , ref[,]blockv subblocks
                          );
mode †bandblockv = struct(index rowpartit , colpartit
                          , ref[]ref[]blockv subblocks
                          );
mode †comnode = struct(ref comblockv thiscom , ref comnode next);
mode †mslinfo = struct(matbnds slbnds , int birth , roottype
                       , ref comnode comlist , ref mroot origin
                       );
mode †mroot = struct(ref mslinfo slicinfo , ref blockv root);
mode spmatv = struct(matbnds †bounds , ref mroot †root);
mode spsymv = struct(trimmer †bounds , ref mroot †symroot);
mode sympatv = struct(trimmer †bounds , ref mroot †root);
```

Modes for sparse vectors:

```
mode †vblockv = union(ref vec , ref[]ref scal , vblockrowv);
mode †dvblockv = struct(int eventbirth , bool allcnt , vblockv vblock);
mode †vblockrowv = struct(index partit , ref[]dvblockv subblocks);
mode †vslinfov = struct(trimmer slbnds , ref vroot origin);
mode †vcomnode = struct(ref comblockv thiscomorig , ref dvblockv thiscomslice
                        , ref[]ref vcomnode nexts
                        );
mode †vslinfom = struct(int sltype , ind , trimmer slbnds , int roottype
                        , ref vcomnode comtree , ref mroot origin
                        );
mode †vroot = struct(int birth , ref dvblockv root
                     , union(ref vslinfom , ref vslinfov) sliceinfo
                     );
mode spvecv = struct(trimmer †bounds , ref vroot †root);
```

Modes for element slices:

```
mode †scslinfom = struct(pair index , int roottype
                         , ref comnode comlist , ref mroot origin
                         );
mode †scslinfov = struct(int ind , ref vroot origin);
mode scalel = struct(ref int †birth , ref scal †elem
                     , union(ref scslinfom , ref scslinfov) †sliceinfo
                     );
```

fig. 5.12. Mode declarations in SPARSEV.

Remark 5.7. In the slicing mechanism as explained in V.5.4 element slices could not be implemented as *ref scal*s. As a consequence many operations for *ref scal*s and *scal*s must be defined for *scalel*s also. A similar consequence arises with the row and column slices of *diagmat*s: many operators of TORRIX-BASIS must also be defined for *ref[]ref scal*. This will lead to a considerable increase in the number of operators to be defined.

Remark 5.8. In the programs 5.1 and 5.2 for a matrix-vector product $a \times u$ a result vector *result* has been declared with each recursive call. These vectors are added with the operator *+:=* . Thus, this may lead to the generation of many concrete-*array1*s. There are at least two methods to increase the efficiency. Moreover, they can be combined.

(i) Only one result vector will be declared and is passed as parameter to all recursive calls. This single result vector will be the only left operand in *+:=* in the whole program.

(ii) The sequence of recursive calls is permuted such that the result vector will be large after a few recursive calls.

Remark 5.9. In the data structures proposed in V.5 the *scal*s of a sparse matrix are organized in *mat*s and *diagmat*s. It is not difficult to extend these data structures with other storage schemes for *scal*s. For example,

mode melem = struct(pair indices , scal value)

can serve as a representation of a block with only one concrete *scal*-element. Of course, the mode declaration for an *spvec* and a *scalel* must be adapted too. Such other storage schemes for *scal*s may decrease the amount of administration overhead of a sparse matrix, but it certainly increases the length of the routine-texts of the operators.

Remark 5.10. Sparse total-array3's, sparse total-array4's, etc. can be stored in a way similar to sparse total-array2's. A tree data structure can be used of which the leaves consist of arrays of *scal*s with 3 (4, 5, ...) subscripts; sons are organized in arrays with 3 (4, 5, ...) subscripts. Slices can be implemented with slice information and a pointer to a subtree or a tree of newly created internal vertices.

Conclusion. In this chapter we have proposed a new data structure for sparse matrices based on a recursive partition of the matrices. With this data structure an arbitrary number of slices of a sparse matrix can be maintained without problems concerning side effects. We have solved the side effect problem by stating in which situations what side effects occur (see (5.34) and (5.50)-(5.53)). Side effects that occur with respect to the slicing mechanism in TORRIX-SPARSE, are similar to the side effects based on slicing in ALGOL 68. Moreover, the proposed data structure allows an efficient storage scheme for sparse matrices with equal blocks.

Both features can be implemented with an increase in administration that is linear in the number of blocks of the matrix. A submatrix or element slice requires an amount of administration that depends linearly on the number of structurally equal blocks containing this slice. As for vector slices, we have proposed a tree data structure in which each slice requires an amount of administration that depends linearly on the number of blocks of the matrix containing *scal*-elements of this slice. This is an upperbound and there are vector slices for which this administration is far less in size. The data structure has been designed in such a way that, if slices of a sparse matrix are created, the operations applied to the matrix require hardly more computation time than in the situation that no slices are created. In the same way, if an operation in which no advantage can be taken from possible equal blocks, is applied to a matrix with structurally equal blocks, then it will hardly require more computation time than in the case that this matrix was stored without structurally equal blocks (but with the same recursive partition).

APPENDIX. TABLE OF OPERATORS.

This appendix contains only those operators of TORRIX-SPARSE that are mentioned in chapter V or are needed in the description of other operators. We will use the notion *sparmat* to denote an *spsym*, *sympat* or *spmat*. The mode *sparmat* is not declared in TORRIX-SPARSE. If one operand of an operator in this table is a *sparmat*. then in fact there are three operators, one for each choice of a *sparmat*. If both operands are *sparmat*s, then there are 9 operators.
Each *sparmat* and *spvec* is determined by its total-array, its bound, its recursive partition, its set of concrete elements and its equality structure. As for the occurrence of side effects with the application of operators, it is important to describe their effect on equality structures. Because no operator in this table has an operand of the mode *ref sparmat* or *ref spvec*, the side effects due to slicing have already been described precisely in (5.34). In the description of the meaning of operators, we will only describe their effect on the total-array, the bounds and the equality structure of a *sparmat* or *spvec*. As for the recursive partition we have mentioned a number of problems, but left open how it could be determined (see V.5.1.3).
Operators of type (5.23) applied to *sparmat*s, *spvec*s or *scalel*s that are slices of *sympat*s or *spsym*s do not disturb the symmetry in the *sympat*s or *spsym*s of which they are slices. Thus, it is possible that elements of a *sympat* or *spsym* that are not element of the slice, are changed by an application of these operators.
In the descriptions of the operators we will often state that an application of an operator is equivalent to a short ALGOL 68 routine-text. This is only an equivalence with regard to total-arrays, equality structure and bounds, but certainly not with regard to the recursive partition. Thus, we describe only what happens with the total-array, the bounds, the equality structure and the slices. This is very important for operators of type (5.23) with two *spvec*- or *sparmat*-operands, because the operands can be slices of the same *sparmat* and have elements in common. For example, what happens with the *spmat* *a* in the elaboration of

(a?(1//10))plusabdel(a?(3//11)newlwb 1)

Describing the meaning of an operator by means of a short ALGOL 68 routine-text does not mean that in an implementation of TORRIX-SPARSE this operator

is coded by this routine-text. Often, much more efficient codes are possible. If in the description of an operator we use the concept "generates a new *sparmat*" (or *spvec*) we mean that a new *sparmat*-value and all its *scal*-locations are allocated with each elaboration of an application of this operator. Only *[]int*s of its operands can be used for the row-partition and colum-partition of this new *sparmat* (or *spvec*).

We reserve specific modes for several identifiers in order to shorten the description:

i, *j*, *h* and *k*	are *int*s,
s	is a *scal* or *scalel*,
r	is a *scalel*,
tr1 and *tr2*	are *trimmer*s,
mb, *mb1* and *mb2*	are *matbnds*',
u	is a *vec* or an *spvec*,
v	is an *spvec*,
a	is a *sparmat* or a *diagmat*,
b	is a *sparmat*.
x and *y*	are *sparmat*s or *spvec*s,

mb consists of four integers: $[m_1:n_1,m_2:n_2]$,

a has bounds $[m_1:n_1,m_2:n_2]$ (if *a* is an *spsym* or *sympat*, then $m_1=m_2$ and $n_1=n_2$),

tr1 and *tr2* have values $[m_1:n_1]$ and $[m_2:n_2]$, respectively

The following identifiers will be declared in TORRIX-SPARSE:

trimmer zerotrim = *c* (t,-t) (see I.2.2) *c*,
matbnds zerobnds = *(zerotrim,zerotrim)*,
spvec zerospvec = *c* the zerovector with bounds *zerotrim* *c*,
spsym zerospsym = *c* the zeromatrix with bounds *zerobnds* *c*,
sympat zerosympat = *c* the zeromatrix with bounds *zerobnds* *c*,
spmat zerospmat = *c* the zeromatrix with bounds *zerobnds* *c*;

The identifier *zerosparmat* will sometimes be used in this appendix to denote one of them. Which one it actually is, can be determined from the result mode of an operator or its operand.

procedure	parameter	result
emptyt	*trimmer*	*bool*
emptym	*matbnds*	*bool*

operator	prio	left operand	right operand	result
?	*9*	*int*	*int*	*pair*
meet	*8*	*trimmer*	*trimmer*	*trimmer*
span	*8*	*trimmer* *matbnds*	*trimmer* *matbnds*	*trimmer* *matbnds*
fitsin	*5*	*trimmer* *matbnds*	*trimmer* *matbnds*	*bool* *bool*
//	*5*	*int*	*int*	*trimmer*
lwb *upb*	*10*		*trimmer*	*int*
lwb *upb*	*8*	*int*	*matbnds*	*int*
rbnds *cbnds*	*10*		*matbnds*	*trimmer*
bnds	*10*		*spvec* *vec* *sparmat*	*trimmer* *trimmer* *matbnds*

emptyt(t1) returns *true* if and only if $m_1 > n_1$.

emptym(mb) returns *true* if and only if $m_1 > n_1$ or $m_2 > n_2$.

i?j returns the *pair (i,j)*.

tr1 meet tr2 returns $[\max(m_1,m_2) : \min(n_1,n_2)]$. However, if this is an empty *trimmer*, then *zerotrim* will be returned.

tr1 span tr2 returns $[\min(m_1,m_2) : \max(n_1,n_2)]$.

tr1 fitsin tr2 returns *true* if and only if $m_1 \geq m_2$ and $n_1 \leq n_2$.

h//k returns *zerotrim* if $h>k$, otherwise $[h:k]$ will be returned.

lwb t1 returns m_1.
upb t1 returns n_1.

k lwb mb returns m_k $(1 \leq k \leq 2)$.
k upb mb returns n_k $(1 \leq k \leq 2)$.

rbnds mb returns $[m_1:n_1]$.
cbnds mb returns $[m_2:n_2]$.

mb1 span mb2 returns [*rbnds mb1 span rbnds mb2* , *cbnds mb1 span cbnds mb2*].
mb1 fitsin mb2 returns *true* if and only if
rbnds mb1 fitsin rbnds mb2 and *cbnds mb1 fitsin cbnds mb2*.

bnds x returns the bounds of *x*.

operator	prio	left operand	right operand	result
x	7	sparmat	vec	vec
		sparmat	spvec	spvec
		sparmat	sparmat	spmat
		diagmat	vec	vec
+ , -	6	spmat	sparmat	spmat
		sparmat	spmat	spmat
		spsym	spsym	spsym
		spsym	sympat	sympat
		sympat	spsym	sympat
		sympat	sympat	sympat
copy, concrcopy	10		spsym	spsym
			sympat	sympat
			spmat	spmat
unshift	10		spsym	spsym
			sympat	sympat
			spmat	spmat
clear, cleargen	10		spsym	spsym
			sympat	sympat
			spmat	spmat
sigmabs	10		sparmat	scal
?	9	spvec	int	scal
		sparmat	pair	scal
?	9	spvec	trimmer	spvec
		sparmat	trimmer	spmat
		trimmer	sparmat	spmat

a×u returns a newly generated *spvec* or *vec* *v* containing the result of the matrix-vector product. Moreover, *bnds v fitsin rbnds bnds a*.

a+b
a-b returns a newly generated *sparmat* *c* containing the sum (difference) of *a* and *b*. The bounds of *c* are *bnds a span bnds b*.

copy a returns a newly generated *sparmat* with the same bounds, total-array2, recursive partition and set of concrete elements as *a*.

concrcopy a returns a newly generated *sparmat* with the same bounds, recursive partition and set of concrete elements as *a*, but its total-array2 is the zero total-array2.

unshift a sets all shifts of all subblocks of *a* (except *a* itself and sub-blocks of *a* that are structurally equal to any other block) to zero by incorporating these shifts in *mat*s and *diagmat*s. *a* will be returned. The recursive partition, the bounds, the total-array2, the equality structure and the set of concrete elements of *a* are not changed.

clear a deletes all partition lines of *a* that are redundant according to (5.40). If possible, partitioned blocks will be replaced by non-partitioned blocks. There is no change in the bounds, the total-array2, the equality structure and the set of concrete elements of *a*. *a* will be returned.

cleargen a deletes all partition lines of *a* that are redundant according to (5.40) or (5.41). If possible, partitioned blocks will be replaced by non-partitioned blocks. There is no change in the bounds, the total-array2, the equality structure and the set of concrete elements of *a*. *a* will be returned.

sigmabs a returns the sum of the absolute values of all *scal*-elements of *a*.

u?i
a?(i?j) returns the *scal*-value of the i^{th} element of *u* ($(i,j)^{th}$ element of *a*). This is not a slice operator. If *i* (or *(i,j)*) is out of the bounds of *u* (or *a*), 0 will be returned.

u?(h//k) returns the slice *v* of *u* containing all elements of *u* with index i ($h \leq i \leq k$). *bnds v = (h//k)meet bnds u*. The i^{th} element of *v* is the i^{th} element of *u* ($i \in$ *bnds v*).

a?(h//k) returns the slice *b* of *a* containing all elements (i,j) of *a* with $h \leq i \leq k$ and $j \in$ *cbnds bnds a*.
bnds b = [(h//k)meet rbnds bnds a, cbnds bnds a]. If *b* has empty bounds, then *b* is *zerospmat*. The $(i,j)^{th}$ element of *a* is the $(i,j)^{th}$ element of *b* ($(i,j) \in$ *bnds b*).

(h//k)?a returns the slice *b* of *a* containing all elements (i,j) of *a* with $h \leq j \leq k$ and $i \in$ *rbnds bnds a*.
bnds b = [rbnds bnds a, (h//k)meet cbnds bnds a]. If *b* has empty bounds, then *b* is *zerospmat*. The $(i,j)^{th}$ element of *a* is the $(i,j)^{th}$ element of *b* ($(i,j) \in$ *bnds b*).

operator	prio	left operand	right operand	result
?	*9*	*sparmat*	*int*	*spvec*
		int	*sparmat*	*spvec*
diag	*8*	*int*	*sparmat*	*spvec*
!	*9*	*spvec*	*int*	*scalel*
		sparmat	*pair*	*scalel*
newlwb	*9*	*spvec*	*int*	*spvec*
		spmat	*int*	*spmat*
		int	*spmat*	*spmat*
symslice	*9*	*spsym*	*trimmer*	*spsym*
		sympat	*trimmer*	*sympat*
symnewlwb	*9*	*spsym*	*int*	*spsym*
		sympat	*int*	*sympat*

a?i returns the slice *v* of *a* containing all elements in row *i* of *a*; *bnds v = cbnds bnds a*.

j?a returns the slice *v* of *a* containing all elements in column *j* of *a*; *bnds v = rbnds bnds a*.

k diag a returns the slice of *a* containing all elements in the k^{th} diagonal of *a*.

u!i returns the slice of *u* that is element *i* of *u*. This element will be concrete after the elaboration of *u!i*. Therefore, a change in the recursive partition of *u* can be made.

a!(i?j) is equivalent with *(a?i)!j*.

u newlwb k returns a slice *v* of *u* containing all elements of *u* and *bnds v = [k : upb bnds u - lwb bnds u +k]* and the i^{th} element of *u* is the $(i+k-lwb\ bnds\ u)^{th}$ element of *v*; if *u* is *zerospvec* then *v* is *u*.

a newlwb k returns a slice *b* of *a* containing all elements of *a* and *bnds b = [k : 1 upb bnds a - 1 lwb bnds a+k, cbnds bnds a]*; the $(i,j)^{th}$ element of *a* is the $(i+k - 1\ lwb\ bnds\ a\ ,j)^{th}$ element of *b*; if *a* is *zerospmat* then *b* is *zerospmat*.

k newlwb a returns a slice *b* of *a* containing all elements of *a* and *bnds b = [rbnds bnds a , k : 2 upb bnds a - 2 lwb bnds a +k]*; the $(i,j)^{th}$ element of *a* is the $(i\ ,\ j+k - 2\ lwb\ bnds\ a)^{th}$ element of *b*; if *a* is *zerospmat* then *b* is *zerospmat*.

a symslice(h//k) returns the slice *(h//k)?a?(h//k)* with this difference that the result is of the same mode as *a*.

a symnewlwb k returns the slice *k newlwb a newlwb k* with this difference that the result is of the same mode as *a*.

operator	prio	left operand	right operand	result
slalias	4	*scalel*	*scalel*	*bool*
		scalel	*spvec*	*bool*
		scalel	*sparmat*	*bool*
		spvec	*scalel*	*spvec*
		spvec	*spvec*	*spvec*
		spvec	*sparmat*	*spvec*
		sparmat	*scalel*	*spmat*
		sparmat	*spvec*	*spmat*
		spsym	*spsym*	*spsym*
		sympat	*sympat*	*sympat*
		sparmat	*spmat*	*spmat*
		spmat	*sparmat*	*spmat*
delete	10		*spvec*	*spvec*
			spsym	*spsym*
			sympat	*sympat*
			spmat	*spmat*
<:=	1	*scalel*	*scal*	*scalel*
		scalel	*scalel*	*scalel*
		spvec	*spvec*	*spvec*
		spsym	*spsym*	*spsym*
		sympat	*spsym*	*sympat*
		sympat	*sympat*	*sympat*
		spmat	*sparmat*	*spmat*
assigndel	1	*spvec*	*spvec*	*spvec*
		spsym	*spsym*	*spsym*
		sympat	*spsym*	*sympat*
		sympat	*sympat*	*sympat*
		spmat	*sparmat*	*spmat*

r slalias s

(i) if *r* and *s* are not slices, *true* will be returned if and only if *r* and *s* are the same *scalels*,

(ii) if exactly one of *r* and *s* is a slice, *false* will be returned,

(iii) if *r* and *s* are slices and there is an *spvec*, *spmat* or *sympat* *x* and integers i or i and j such that *r* and *s* are both the i^{th} or $(i,j)^{th}$ element of *x*, then *true* will be returned,

(iv) if *r* and *s* are slices and there is an *spsym* *a* and integers i and j such that *r* and *s* are the $(i,j)^{th}$ of $(j,i)^{th}$ element of *a*, then *true* will be returned.

Observe, that *r slalias s* is not dependent of the equality structure of a *sparmat*.

If *r slalias s* returns *true*, we say that *r* (*s*) has a slalias with *s* (resp. *r*).

r slalias x returns *true* if *r* has a slalias with an element of *x*.

x slalias s returns the smallest slice *x?(h//k)* (if *x* is an *spvec*) or *(h2//k2)?x?(h1//k1)* (if *x* is a *sparmat*) containing all elements of *x* (in the lower triangular part of *x* if *x* is an *spsym*) that have a slalias with *s*.

x slalias y returns the smallest slice *x?(h//k)* (if *x* is an *spvec*) or *(h2//k2)?x?(h1//k1)* (if *x* is a *sparmat*) containing all elements of *x* (in the lower triangular part of *x* if *x* is an *spsym*) that have a slalias with an element of *y*.

delete x returns an *spvec* of *sparmat y* with the same bounds, total-array, set of concrete elements and equality structure as *x* before the elaboration of *delete x*. All elements of *x* are made virtual or zero. As for the equality structure of *ax* if *x* is a slice of *ax*, *delete x* satisfies (5.48)-(5.51) with active domain all elements of *ax* selected by *x*. The result of *delete x* is not a slice.

r<:=s assigns the *scal*-value of *s* to *r*.

x<:=y requires that *bnds y fitsin bnds x*; if *x slalias y* returns *zerosparmat*, then *y* is assigned to *x*, otherwise *x<:=y* is equivalent with *x <:= copy y*. Structurally equal blocks in *y* will lead to structurally equal blocks in *x*. Elements in *x* with indices outside the bounds of *y* are made virtual or zero. If *x* is a slice of *ax*, the active domain of *x<:=y* consists of all elements of *ax* selected by *x*. *x<:=y* satisfies (5.48)-(5.51).

x assigndel b is equivalent to *(x <:= delete y)*.

operator	prio	left operand	right operand	result
=:=	1	spvec	spvec	spvec
blockexch	1	spsym sympat spmat	spsym sympat sparmat	spsym sympat spmat
x<	1	spmat sympat spsym	scal scal scal	spmat sympat spsym
+ - plusabdel	1	spsym sympat sympat spmat	spsym spsym sympat sparmat	spsym sympat sympat spmat
unshare	10		spmat sympat spsym	spmat sympat spsym

u=:=v is equivalent to *(spvec w = delete v; v<:=u; u<:=w).*

a blockexch b requires that *a* and *b* are slices of the same *sparmat c* and that there are blocks A and B of *c* that have bounds equal to the bounds of *a* and *b*, respectively. It exchanges A and B and returns *a*. As for structurally equal blocks C and C' of *c* we have:

(i) if neither C nor C' is a subblock of A or B, then C and C' will remain structurally equal.

(ii) if exactly one of C and C' is a subblock of A or B (suppose C), then C' will be structurally equal to the block obtained by shifting C in the same way as A (resp. B) is shifted.

(iii) if both C and C' are subblocks of A and B, then C and C' will be shifted and there shifted blocks will be structurally equal.

a×<s performs *r×:=s* for each *scal*-element (virtual or concrete) *r* of *a*.

x+<y adds the value of the i^{th} (or $(i,j)^{th}$) element of *y* to the i^{th} (or $(i,j)^{th}$) element of *x* and returns *x*. It is required that *bnds y fitsin bnds x*. As for the equality structure of *ax* (if *x* is a slice of *ax*), *x+<y* satisfies (5.48)-(5.51) with active domain the elements of *ax* selected by *x*. If C and C' are structurally equal subblocks of *x* and are both outside *(cbnds bnds y)?x?(rbnds bnds y)*, then C and C' will remain structurally equal. It is left to the implementor what happens with other structurally equal subblocks of *x*.

x-<y as *x+<y* but now with subtraction.

x plusabdel y is equivalent with
(sparmat z = delete y; x+<z) or *(spvec z = delete y; x+<z).*

unshare a disconnects all structurally equal blocks of which only one is a subblock of *a*; structurally equal blocks that are both subblocks of *a*, will not be disconnected; *a* will be returned.

CHAPTER VI

OPTIMAL PARTITIONS OF A SPARSE MATRIX IN TORRIX-SPARSE

In chapter V we have developed a data structure for sparse matrices based on recursive partitions. We assumed that the recursive partition of the matrix was determined by the user or dynamically by the supporting system. However, in the latter case we left open how the recursive partition could be determined. In this chapter we will deal with the problem of determining an optimal sparse matrix storage tree T (a recursive partition) for a matrix A in case its sparsity pattern is given as a set of *mat*s and *diagmat*s S. We will consider two criteria for optimization:

(i) the sparse matrix storage tree T for A should have a minimum number of leaves,

(ii) the sparse matrix storage tree T for A should be of minimum height.

With (i) the administration overhead in the storage scheme for A is minimized and with (ii) we have a minimum upperbound for the time required to access the $(i,j)^{th}$ element of A. We will see that there are matrices A for which a sparse matrix storage tree T does not exist unless *mat*s and *diagmat*s of S are splitted into smaller ones. There is a set S such that each sparse matrix storage tree T for the sparse matrix represented by S has at least $\frac{3}{2}.|S|$ leaves.

In this chapter we will present several algorithms to determine a sparse matrix storage tree of minimum size or height and we will prove that criteria (i) and (ii) cannot always be satisfied simultaneously. The chapter contains three sections. In the first section we will introduce the concept of a storage tree for a set of concrete domains. In VI.2 we will consider storage trees with a minimum number of leaves (i.e., the optimization criterion (i)). We will deal briefly with the problem of finding a recursive partition of a sparse matrix, given a family of sets of concrete domains of which the corresponding submatrices must be structurally equal (cf. chapter V). In VI.3 we will deal with storage trees of minimum height (i.e., the optimization cri-

terion (ii)) and prove that both criteria cannot be satisfied simultaneously for each sparse matrix.
Throughout this chapter we will assume that the sparsity pattern of A is given as a set of rectangular concrete domains. We will not deal with the problem to find a minimum number of concrete domains that represent the sparsity pattern of a matrix. For this we refer to [54] and [53]. In this chapter the concept of a concrete domain can often be replaced by the concept of a rectangle as used in literature (cf. e.g. [08]).

VI.1 STORAGE TREES AND SETS OF CONCRETE DOMAINS.

DEFINITION 6.1. A concrete domain B is a 4-tuple $(l_1(B),u_1(B),l_2(B),u_2(B))$ of integer numbers. The index set ind(B) of B is $\{(i,j)\in \mathbb{Z}^2 : l_1(B)\leq i\leq u_1(B)$ and $l_2(B)\leq j\leq u_2(B)\}$. B is non-empty if ind(B) is non-empty. Two concrete domains B_1 and B_2 are disjoint if their index sets are disjoint.

DEFINITION 6.2. Given a concrete domain B, we define short hand notations for the following subdomains. Let h, k, p, q$\in \mathbb{Z}$.

$B[h:k,p:q] = (\max\{h,l_1(B)\}, \min\{k,u_1(B)\}, \max\{p,l_2(B)\}, \min\{q,u_2(B)\})$,
$B[h:k, \ ] = B[h:k, l_2(B):u_2(B)]$,
$B[\ ,p:q] = B[l_1(B):u_1(B), p:q]$,
$B[h: , \ ] = B[h:u_1(B), \ ]$,
$B[\ :k, \ ] = B[l_1(B):k, \ ]$,
$B[\ ,p: \ ] = B[\ , p:u_2(B)]$,
$B[\ , :q] = B[\ , l_2(B):q]$.

Each sparse matrix A can be considered a set of *mat*s with mutually disjoint concrete domains. These *mat*s may consist of only one *scal*-element. As for the sparsity patterns and the recursive partitions the values of the elements of *mat*s are not important. It suffices to know the concrete domains. Thus, we restrict ourselves to sets of *mat*s, though the matrix can contain also *diagmat*s. However, all results we will present, still hold if *diagmat*s are incorporated. In this chapter we will deal with finite sets of concrete domains. We assume that all concrete domains in such a set are non-empty and mutually disjoint unless it is explicitly stated otherwise.

DEFINITION 6.3. Let S_1 and S_2 be sets of concrete domains. S_1 and S_2 are equivalent if

$$\bigcup_{B\in S_1} \text{ind}(B) = \bigcup_{B\in S_2} \text{ind}(B).$$

Thus, S_1 and S_2 are equivalent if they represent the same sparsity pattern.

DEFINITION 6.4. Let S_1 and S_2 be sets of concrete domains. S_1 is splittable into S_2 ($S_2 \succ S_1$) if the two following conditions are satisfied:

(i) S_1 and S_2 are equivalent,

(ii) for each $B\in S_2$ there is a $B'\in S_1$ such that $\text{ind}(B) \subseteq \text{ind}(B')$.

Obviously, if $S_2 \succ S_1$ then $|S_2| \geq |S_1|$ and the concrete domains of S_1 can actually be split by finitely many cuts to obtain the concrete domains of S_2.

DEFINITION 6.5. Let S be a set of concrete domains. Then

$$l_i(S) = \min\{l_i(B) : B\in S\} \quad (\min(\emptyset)=\infty) \quad 1\leq i\leq 2,$$

$$u_i(S) = \max\{u_i(B) : B\in S\} \quad (\max(\emptyset)=-\infty) \quad 1\leq i\leq 2,$$

S[r: ,] = {B[r: ,] : B∈S and B[r: ,] is non-empty},

S[:r,] = {B[:r,] : B∈S and B[:r,] is non-empty},

S[,c:] = {B[,c:] : B∈S and B[,c:] is non-empty},

S[, :c] = {B[, :c] : B∈S and B[, :c] is non-empty},

S[,r:c] = S[,r:][, :c],

S[r:c,] = S[r: ,][:c,].

DEFINITION 6.6. Let S_1 and S_2 be sets of concrete domains. S_1 includes S_2 if S_1 and S_2 satisfy the following two conditions:

(i) for each $B_2\in S_2$ there is a $B_1\in S_1$ with $\text{ind}(B_2) \subseteq \text{ind}(B_1)$,

(ii) for each $B_1\in S_1$ there is at most one $B_2\in S_2$ with $\text{ind}(B_2) \subseteq \text{ind}(B_1)$.

Proposition 6.1. *Let S be a set of concrete domains and r and c∈ ℤ. Then*

(i) S includes S[:r,], S[r+1: ,], S[, :c] *and* S[,c+1:]

(ii) S[:r,] ∪ S[r+1: ,] ≻ S, S[, :c] ∪ S[,c+1:] ≻ S.

Proof: trivial. □

A set of concrete domains will be stored at the leaves of a rooted tree T. There may be more leaves resulting from the construction of T, but they will be limited in number (see proposition 6.2) and are easily interpreted as

"empty" concrete domains. For each vertex v of T, T(v) is the subtree of T with root v and leaves(T(v)) is the set of non-empty leaves of T(v).

DEFINITION 6.7. Let S be a set of concrete domains and T a tree with leaves that are (possibly empty) concrete domains. T is a storage tree for S if the following four conditions are satisfied:

(i) each non-leaf v of T has at least two sons and at least two sons of v have non-empty concrete domains as descendants,

(ii) the leaves of T are mutually disjoint concrete domains,

(iii) for each vertex v of T there are m, n, h and k in $\mathbb{Z}$ such that

$$\text{leaves}(T(v)) = \{B \in S : \text{ind}(B) \subseteq [m:n]\times[h:k]\}.$$

(iv) for each non-leaf v of T with

$$\text{leaves}(T(v)) = \{B \in S : \text{ind}(B) \subseteq [m:n]\times[h:k]\}$$

there are $m-1=r_0<r_1<\ldots<r_p=n$ and $h-1=c_0<c_1<\ldots<c_q=k$ (6.1

such that v has $p\times q$ sons w_{ij} $(1\leq i\leq p, 1\leq j\leq q)$ with

$$\text{leaves}(T(w_{ij})) = \{B \in S : \text{ind}(B) \subseteq [r_{i-1}+1:r_i]\times[c_{j-1}+1:c_j]\}.$$

Fact 6.1. There is a set S of concrete domains for which no storage tree exists.

Proof: see fig. 6.1. □

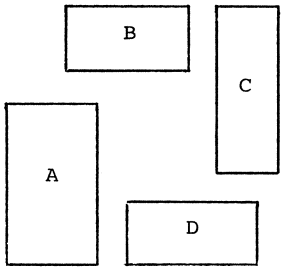


fig. 6.1. A set {A,B,C,D} of concrete domains for which no storage tree exists.

Observe the difference between a sparse matrix storage tree and a storage tree for a set of concrete domains. The latter determines a recursive partition whereas the first is determined by a recursive partition.

Proposition 6.2. *For each set S of concrete domains there is an S'⊃S for which a storage tree exists.*

Proof:
Let $S' = \{(i,i,j,j) : i,j \in \mathbb{Z}$ and there is a $B \in S$ such that $(i,j) \in ind(B)\}$.
Clearly S' is finite, S'⊃S and a storage tree for S' exists.

□

It is obvious that, if a storage tree exists for a set S of concrete domains, then a storage tree exists for each $S' \subseteq S$ and each S' included in S.
If a storage tree T is stored in computer memory, the numbers r_i $(0 \leq i \leq p)$ and c_j $(0 \leq j \leq q)$ in (6.1) will be stored for each non-leaf. The leaves are stored as four integers. With each non-leaf v a rectangular array with the sons of v will be maintained. Thus, storing T requires $O(|T|)$ space with $|T|$ the number of vertices of T.

Remark 6.1. In the literature several other data structures for storing configurations of objects in the plane are studied. For example, quad trees and k-d trees (cf. e.g. [24], [07]). Each quad tree has leaves that correspond to concrete domains and thus is a storage tree with p=q=2 in (6.1) for each non-leaf. In a similar way each k-d tree is a storage tree. Thus, the set of quad trees and k-d trees with leaves that are concrete domains is a subset of the set of storage trees. As a consequence, a number of results presented in this chapter also hold for the cases in which we would restrict ourselves to quad trees or k-d trees to represent sets of concrete domains.

Fact 6.2. For each $k \geq 4$ there is a set S of concrete domains with $|S|=k$ such that no storage tree exists for S, but for each $B \in S$ a storage tree exists for $S \setminus \{B\}$.

Proof: For k=4, let S be the set as denoted in fig. 6.1.
Let $k \in \mathbb{N}$, $k \geq 5$. Then $S=\{A_{-1}, A_0, A_1, \ldots, A_{k-2}\}$ with

$$A_i = (i,i+1,i,i) \qquad 1 \leq i \leq k-2$$
$$A_0 = (k-1,k-1,1,2)$$
$$A_{-1} = (1,1,2,k-2) \qquad \text{(see fig. 6.2).}$$

It is easy to see that there is no $r \in \mathbb{Z}$ and no $c \in \mathbb{Z}$ such that

$$S = S[:r,]US[r+1:,] \text{ with } S[:r,] \neq \emptyset \text{ and } S[r+1:,] \neq \emptyset$$
$$\text{or } S = S[,:c]US[,c+1:] \text{ with } S[,:c] \neq \emptyset \text{ and } S[,c+1:] \neq \emptyset.$$

Thus no storage tree for S exists.

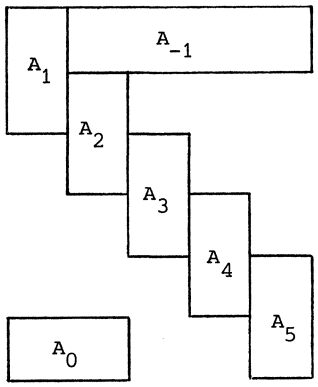


fig. 6.2. A set of 7 concrete domains for which no storage tree exists, but whose proper subsets all allow a storage tree.

Let $S_i = S \setminus A_i$ $(-1 \leq i \leq k-2)$. If $i \geq 1$ then $S_i = S_i[:i,] \cup S_i[i+1:,]$ with $A_{-1} \in S_i[:i,]$ and $A_0 \in S_i[i+1:,]$. It is easy to see that storage trees exist for $S_i[:i,]$ and $S_i[i+1:,]$.
$S_0 = S_0[,:1] \cup S[,2:]$ with $S_0[,:1] \neq \emptyset$ and $S_0[,2:] \neq \emptyset$. Obviously $S_0[,:1]$ and $S_0[,2:]$ have storage trees.
$S_{-1} = S_{-1}[,:k-3] \cup S_{-1}[,k-2:]$. Storage trees exists for $S_{-1}[,:k-3]$ and $S_{-1}[,k-2:]$.

□

VI.2 FINDING A STORAGE TREE WITH A MINIMUM NUMBER OF LEAVES.

In the previous section we have seen that a storage tree does not exist for each set of concrete domains S, but, on the other hand, that for each S there is an $S' \supset S$ with a storage tree. In this section we will concentrate on finding an S' for which a storage tree exists with a minimum number of leaves. We will present lower and upperbounds for the number of leaves in a storage tree for S'. Moreover, we will present a polynomial time algorithm that, given S, finds an $S' \supset S$ that has a storage tree with a minimum number of leaves.

Proposition 6.3. *There is a sequence* $(S_p)_{p \geq 2}$ *of sets of concrete domains with* $|S_p| = p$ *such that for every storage tree* T_p *with* $\text{leaves}(T_p) \succ S_p$

$$|\text{leaves}(T_p)| \geq \lfloor \tfrac{3}{2} . |S_p| \rfloor - 1. \qquad (6.2)$$

Proof: We define the following sets $(S_i)_{i\geq 2}$ of concrete domains. Let $i\geq 1$; then S_{2i} and S_{2i+1} are defined with:

$S_{2i}=\{A_j : 1\leq j\leq 2i\}$

$A_{2j} = (1,2j-1,2j-1,2j) \quad 1\leq j\leq i-1$

$A_{2i} = (1,2i-1,2i-1,2i-1)$

$A_1 = (3,2i+1,1,1)$

$A_{2j-1} = (2j+1,2i+1,2j-2,2j-1) \quad 2\leq j\leq i$

$S_{2i+1}=\{A_j : 1\leq j\leq 2i+1\}$

$A_{2j} = (1,2j-1,2j-1,2j) \quad 1\leq j\leq i$

$A_1 = (3,2i+3,1,1)$

$A_{2j-1} = (2j+1,2i+3,2j-2,2j-1) \quad 2\leq j\leq i$

$A_{2i+1} = (2i+3,2i+3,2i,2i)$

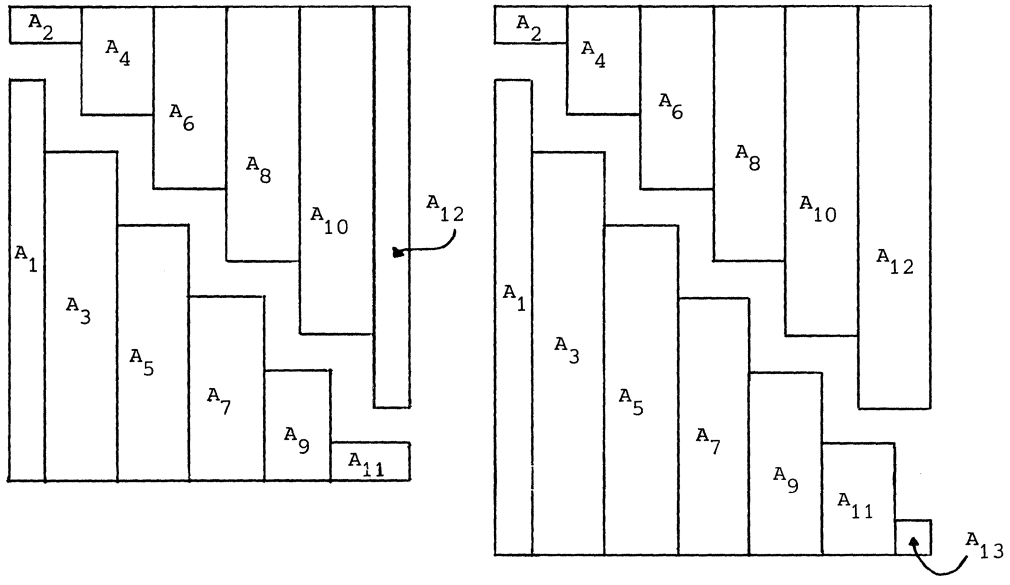


fig. 6.3. S_{12} and S_{13} as used in the proof of proposition 6.3.

Without proof we state the following properties concerning S_{2i} and S_{2i+1} ($i\in\mathbb{N}$, $i\geq 1$):

(i) $|S_{2i}|=2i$ and $|S_{2i+1}|=2i+1$.

(ii) For each $r\in\mathbb{N}$ $(1\leq r\leq 2i+1)$:

$S_{2i}[:r,]\cup S_{2i}[r+1:,]$ contains $3i-1$ concrete domains with $r\leq 2i$.

$S_{2i+1}[:r,]\cup S_{2i+1}[r+1:,]$ contains at least $3i$ concrete domains. (6.3)

(iii) For each even $c \in \mathbb{N}$ $(2 \leq c \leq 2i-2)$:

$S_{2i}[\ ,:c]$ includes S_{c+1},

$S_{2i}[\ ,c+1:]$ includes S_{2i-c} (if S_{2i-c} is shifted right and downwards in the plane over c positions),

$S_{2i+1}[\ ,:c]$ includes S_{c+1}, (6.

$S_{2i+1}[\ ,c+1:]$ includes S_{2i+1-c} (if S_{2i+1-c} is shifted right and downwards in the plane over c positions). (6.

(iv) For each odd $c \in \mathbb{N}$ $(1 \leq c \leq 2i-1)$:

$S_{2i}[\ ,:c]$ includes S_{c+1},

$S_{2i}[\ ,c+1:] \backslash A_{c+1}[\ ,c+1:]$ includes S_{2i-c-1} (if S_{2i-c-1} is shifted right and downwards in the plane over c positions),

$S_{2i+1}[\ ,:c]$ includes S_{c+1}, (6.

$S_{2i+1}[\ ,c+1:] \backslash A_{c+1}[\ ,c+1:]$ includes S_{2i-c} (if S_{2i-c} is shifted right and downwards in the plane over c positions). (6.

Let T_p be a storage tree for some $S_p' \succ S_p$. We will prove by induction on p that $|\text{leaves}(T_p)| \geq \lfloor \frac{3}{2} |S_p| \rfloor - 1$. The proposition holds for p=2 and p=3 (S_2 and S_3 have storage trees). It is easy to see that there is an $S_4' \succ S_4$ with $|S_4'| = 5$ and S_4' has a storage tree. Suppose (6.2) holds for p=2,3,...,2i. Let T_{2i+1} be a storage tree for some $S_{2i+1}' \succ S_{2i+1}$. Without loss of generality we may assume that the root of T_{2i+1} has exactly two sons t_1 and t_2. There are two cases:

Case 1: There is an $r \in \mathbb{N}$ $(1 \leq r \leq 2i+2)$ such that $T_{2i+1}(t_1)$ is a storage tree for $S_{2i+1}'[:r,\ ] \succ S_{2i+1}[:r,\ ]$ and $T_{2i+1}(t_2)$ is a storage tree for $S_{2i+1}'[r+1:,\ ] \succ S_{2i+1}[r+1:,\ ]$. Then with (6.3):

$$|\text{leaves}(T_{2i+1})| \geq 3i = \lfloor \tfrac{3}{2} |S_{2i+1}| \rfloor - 1.$$

Case 2: There is a $c \in \mathbb{N}$ $(1 \leq c \leq 2i)$ such that $T_{2i+1}(t_1)$ is a storage tree for $S_{2i+1}'[\ ,:c] \succ S_{2i+1}[\ ,:c]$ and $T_{2i+1}(t_2)$ is a storage tree for $S_{2i+1}'[\ ,c+1:] \succ S_{2i+1}[\ ,c+1:]$. For odd c we obtain with (6.6), (6.7) and the induction hypothesis:

$$|\text{leaves}(T_{2i+1})| = |\text{leaves}(T_{2i+1}(t_1))| + |\text{leaves}(T_{2i+1}(t_2))| \geq$$

$$\geq \tfrac{3}{2}(c+1) - 1 + 1 + \lfloor \tfrac{3}{2}(2i-c) \rfloor - 1 = \lfloor \tfrac{3}{2} \cdot |S_{2i+1}| \rfloor - 1.$$

For even c we obtain with (6.4), (6.5) and the induction hypothesis:

$$|\text{leaves}(T_{2i+1})| = |\text{leaves}(T_{2i+1}(t_1))| + |\text{leaves}(T_{2i+1}(t_2))| \geq$$
$$\lfloor\tfrac{3}{2}(c+1)\rfloor-1+\lfloor\tfrac{3}{2}(2i+1-c)\rfloor-1 = \lfloor\tfrac{3}{2}.|S_{2i+1}|\rfloor-1.$$

Thus, if (6.2) holds for p=2,3,...,2i, then it holds for p=2i+1. In a similar way it can be proven that (6.2) holds for p=2i+2, if it holds for p=2,3, ...,2i+1.

□

DEFINITION 6.8. Let S and S' be sets of concrete domains. S' is a minimum split of S if it has a storage tree, S'≻S and for every S"≻S that has a storage tree, $|S'|\leq|S''|$. The function minsplit is defined as:

$$\text{minsplit}(S)=|S'|, \text{ with } S' \text{ a minimum split of } S.$$

Proposition 6.4. *For each set S of concrete domains* $\text{minsplit}(S)\leq|S|^2$.

Proof: Let S be a set of concrete domains. Let $(x_i)_{i\geq1}$ be the ordered sequence of numbers of the set $\{u_1(B):B\in S\}$ and let $x_0=l_1(S)-1$. Let $(y_j)_{j\geq1}$ be the ordered sequence of numbers of the set $\{u_2(B):B\in S\}$ and let $y_0=l_2(S)-1$. Let $(x_i)_{i\geq0}$ and $(y_j)_{j\geq0}$ have p and q elements respectively, $p,q\leq|S|$. For each i,j $(1\leq i\leq p, 1\leq j\leq q)$ we define

$$S_{ij} = \{B[x_{i-1}+1:x_i, y_{j-1}+1:y_j] : B\in S \text{ and } B[x_{i-1}+1:x_i, y_{j-1}+1:y_j] \text{ is non-empty}\},$$

and $S'=\bigcup_{i=1}^{p}\bigcup_{j=1}^{q}S_{ij}$.

Obviously $|S_{ij}|\leq1$ $(1\leq i\leq p, 1\leq j\leq q)$. With this choice of S' we have that S'≻S and S' has a storage tree.

$$\text{minsplit}(S) \leq |S'| \leq \sum_{i=1}^{p}\sum_{j=1}^{q}|S_{ij}| \leq p.q \leq |S|^2.$$

□

A better upperbound can be shown.

Theorem 6.1. *For each* $\varepsilon\in\mathbb{R}$ $(0<\varepsilon<1)$ *there is a* $d\in\mathbb{N}$ *such that for each set S of concrete domains* $\text{minsplit}(S)\leq d.|S|^{1+\varepsilon}$.

Proof: Let $g(n)=\max\{\text{minsplit}(S) : S \text{ is a set of concrete domains, } |S|=n\}$.

Let $\varepsilon\in\mathbb{R}$ $(0<\varepsilon<1)$. Let $c\in\mathbb{R}$, $c>0$, $2c^{\varepsilon}+c^{1+\varepsilon}<1$.

Let $f(k)=\lceil c.k^{1-\varepsilon}\rceil$ $(k\in\mathbb{N})$.

Let S be a set of concrete domains and $|S|=n$. Let $i_{max}=\left\lceil\frac{2n-1}{f(n)}\right\rceil$.

Without loss of generality we can assume that for all

$$B,B' \in S \ (B \neq B'): \quad l_2(B) \neq l_2(B'),\ u_2(B) \neq u_2(B') \text{ and } l_2(B)-1 \neq u_2(B'). \tag{6.8}$$

Let $x_0,\ldots,x_{2n-1}$ be the ordered set of elements of

$$\{l_2(B)-1 : B \in S\} \cup \{u_2(B) : B \in S\}.$$

Now we split S into smaller sets S_i $(1 \leq i \leq i_{max})$:

$$S_i = S[\ ,1+x_{(i-1).f(n)} : x_{i.f(n)}] \quad \text{if } (1 \leq i \leq i_{max}-1)$$

and $S_i = S[,1+x_{(i-1).f(n)} : x_{2n-1}]$ if $i=i_{max}$.

If $S_i' > S_i$ has a storage tree for each i $(1 \leq i \leq i_{max})$, then $\bigcup_{i=1}^{i_{max}} S_i' > S$ has a storage tree.

Let $A_i = \{B \in S : 1+x_{(i-1)f(n)} < l_2(B) \leq u_2(B) < x_{i.f(n)}\}$,

$B_i = \{B \in S \setminus A_i : 1+x_{(i-1).f(n)} < l_2(B) \leq x_{i.f(n)}$ or

$1+x_{(i-1)f(n)} \leq u_2(B) < x_{i.f(n)}\}$,

$C_i = \{B \in S : l_2(B) \leq 1+x_{(i-1).f(n)}$ and $u_2(B) \geq x_{i.f(n)}\}$,

$D_i = S \setminus (A_i \cup B_i \cup C_i)$.

Observe that $S_i = A_i \cup B_i[\ ,l_2(S_i):u_2(S_i)] \cup C_i[\ ,l_2(S_i):u_2(S_i)]$ and

$$|S_i| = |A_i| + |B_i| + |C_i|. \tag{6.9}$$

All blocks $B \in C_i[\ ,l_2(S_i):u_2(S_i)]$ satisfy $l_2(B)=l_2(S_i)$ and $u_2(B)=u_2(S_i)$.

Thus:

$$\text{minsplit}(S_i) = \text{minsplit}(S_i \setminus C_i[,l_2(S_i):u_2(S_i)]) + |C_i[,l_2(S_i):u_2(S_i)]|. \tag{6.10}$$

For each $B \in S$ there are at most $i_{max}-2$ i's with $2 \leq i \leq i_{max}-1$ such that $B \in C_i$ and hence

$$\sum_{i=2}^{i_{max}-1} |C_i[\ ,l_2(S_i):u_2(S_i)]| \leq (i_{max}-2).|S| = n(i_{max}-2).$$

Moreover we have $|C_1|, |C_{i_{max}}| \leq 1$ and therefore

$$\sum_{i=1}^{i_{max}} |C_i[\ ,l_2(S_i):u_2(S_i)]| \leq n(i_{max}-2)+2 \leq \frac{2n^2}{f(n)}. \tag{6.11}$$

For each $B \in A_i \cup B_i$ there is a j $((i-1).f(n)+1 \leq j \leq i.f(n)-1)$ with

$$1+x_j = l_2(B) \text{ or } x_j = u_2(B).$$

With (6.8) and the definition of S_i we conclude that

$$|A_i \cup B_i| \leq i.f(n)-1-((i-1).f(n)+1)+1 = f(n)-1 \text{ and hence}$$

$$|A_i \cup B_i[\ ,l_2(S_i):u_2(S_i)]| \leq f(n)-1 \quad (1 \leq i \leq i_{max})$$

By the definition of g, this leads to

$$\text{minsplit}(A_i \cup B_i[\ ,l_2(S_i):u_2(S_i)]) \leq g(f(n)-1). \tag{6.12}$$

If we substitute (6.9), (6.11) and (6.12) in (6.10), we obtain

$$\text{minsplit}(S) \leq \frac{2n^2}{f(n)} + \sum_{i=1}^{i_{max}} g(f(n)-1) \leq \frac{2n^2}{f(n)} + \left(\frac{2n}{f(n)}+1\right).g(f(n)-1).$$

This holds for every set S and hence,

$$g(n) \leq \frac{2n^2}{f(n)} + \left(\frac{2n}{f(n)}+1\right).g(f(n)-1) \leq \frac{2n^2}{c.n^{1-\varepsilon}} + \left(\frac{2n}{c.n^{1-\varepsilon}}+1\right)g(\lfloor c.n^{1-\varepsilon} \rfloor).$$

If we have a monotone function $h: \mathbb{R} \to \mathbb{R}$ such that

(i) $g(n) \leq h(n)$ for all $n \in \mathbb{N}$, $n \leq N$ for some $N \in \mathbb{N}$,

(ii) $$\frac{2y^2}{c.y^{1-\varepsilon}} + \left(\frac{2y}{c.y^{1-\varepsilon}}+1\right).h(c.y^{1-\varepsilon}) \leq h(y) \tag{6.13}$$

then $g(n) \leq h(n)$ for all $n \in \mathbb{N}$. Let $h(y)=dy^{1+\varepsilon}$ with d large enough such that $g(n) \leq h(n)$ for small n and $d \geq \frac{2}{c}.\frac{1}{1-2c^{\varepsilon}-c^{1+\varepsilon}}$. With $2c^{\varepsilon}+c^{1+\varepsilon}<1$ this leads to $\frac{2}{c}+2c^{\varepsilon}d+c^{1+\varepsilon}.d \leq d$ and for all y $\frac{2}{c}y^{1+\varepsilon}+2c^{\varepsilon}dy^{1+\varepsilon}+c^{1+\varepsilon}dy^{1+\varepsilon} \leq d.y^{1+\varepsilon}$. Hence

$\frac{2}{c}.y^{1+\varepsilon} + \frac{2}{c}y^{\varepsilon}.d.c^{1+\varepsilon}.y^{1-\varepsilon^2} + dc^{1+\varepsilon}.y^{1-\varepsilon^2} \leq d.y^{1+\varepsilon}$ and

$$\frac{2y^2}{c.y^{1-\varepsilon}} + \left(\frac{2y}{c.y^{1-\varepsilon}}+1\right)d.(c.y^{1-\varepsilon})^{1+\varepsilon} \leq d.y^{1+\varepsilon}.$$

Thus $h(y)=d.y^{1+\varepsilon}$ satisfies (6.13) and $g(n) \leq d.n^{1+\varepsilon}$ for all $n \in \mathbb{N}$.

□

In order to test whether a set of concrete domains has a storage tree, the following lemma is important.

Lemma 6.1. *Let S be a set of concrete domains with* $|S| \geq 2$.

(i) If S has a storage tree then there is an $r \in \mathbb{Z}$ $(l_1(S) \leq r < u_1(S))$ *such that for all* $B \in S$ *either* $l_1(B) \geq r+1$ *or* $u_1(B) \leq r$ *or there is a* $c \in \mathbb{Z}$ $(l_2(S) \leq c < u_2(S))$ *such that for all* $B \in S$ *either* $l_2(B) \geq c+1$ *or* $u_2(B) \leq c$.

(ii) Let $r \in \mathbb{Z}$ $(l_1(S) \leq r < u_1(S))$ *such that for all* $B \in S$ *either* $l_1(B) \geq r+1$ *or* $u_1(B) \leq r$. *Then* S *has a storage tree if and only if* S[:r,] *and* S[r+1:,] *both have storage trees.*

(iii) Let $c \in \mathbb{Z}$ $(l_2(S) \leq c < u_2(S))$ *such that for all* $B \in S$ *either* $l_2(B) \geq c+1$ *or* $u_2(B) \leq c$. *Then* S *has a storage tree if and only if* S[,:c] *and* S[,c+1:] *both have storage trees.*

Proof:

(i) Let T be a storage tree for S. Without loss of generality we may assume that the root of T has exactly two sons t_1 and t_2 and for some $r \in \mathbb{Z}$ $(l_1(S) \leq r < u_1(S))$.

$$\text{leaves}(T(t_1)) = \{B \in S : u_1(B) \leq r\}, \qquad \text{leaves}(T(t_2)) = \{B \in S : l_1(B) \geq r+1\}.$$

Because $S = \text{leaves}(T(t_1)) \cup \text{leaves}(T(t_2))$, either $u_1(B) \leq r$ or $l_1(B) \geq r+1$ for each $B \in S$.

(ii) Let $r \in \mathbb{Z}$ $(l_1(S) \leq r < u_1(S))$ such that for all $B \in S$ either $l_1(B) \geq r+1$ or $u_1(B) \leq r$.

⇒: Suppose S has a storage tree. S[:r,] and S[r+1:,] are subsets of S and hence have both storage trees.

⇐: Obvious.

(iii) Similar to the proof of (ii).

□

Using lemma 6.1 an algorithm can be designed that, given a set of concrete domains, determines whether it has a storage tree.

algorithm 6.1.

```
proc TREETEST = (set S)bool:
co returns true if and only if S has a storage tree co
  if |S|≤3   then true
elif there is an r (l1(S)≤r<u1(S)) such that for all B∈S
     either u1(B)≤r or l1(B)≥r+1
then if  TREETEST(S[:r, ])  then TREETEST(S[r+1:, ]) else false fi
elif there is a c (l2(S)≤c<u2(S)) such that for all B∈S
     either u2(B)≤c or l2(B)≥c+1
then if  TREETEST(S[ ,:c])  then TREETEST(S[ ,c+1:]) else false fi
else false
  fi
```

Proposition 6.5. *With a suitable data structure for S, algorithm 6.1 will run in time* $O(|S|^2)$.

Proof: We assume that there are four doubly linked lists containing the concrete domains of S ordered with respect to l_1, u_1, l_2 and u_2 respectively. Then in each recursive call r (or c) can be determined in linear time and four lists for S[:r,] and S[r+1:,] (or S[,:c] and S[,c+1:]) can be built in time linear in $|S|$. Thus, each call (without its recursive calls) can be performed in linear time. We can charge a constant time portion of the costs to each concrete domain in the parameter of this call. For each concrete domain $B \in S$ there are at most $|S|-2$ subsets $S_{|S|-2} \subseteq S_{|S|-3} \subseteq \ldots \subseteq S_1 = S$ containing B that will be a parameter of a (recursive) call. Thus, the total time needed per concrete domain is at most $c_1.(|S|-2)$ for some $c_1 \in \mathbb{N}$. Hence, the total time used by the algorithm is $c_1.|S|.(|S|-2) = O(|S|^2)$. □

The algorithm TREETEST can be used as a subroutine in an algorithm to test whether a recursive partition of a sparse matrix exists such that the concrete domains of a user specified family of sets of concrete domains are equal to the bounds of blocks obtained by the recursive partition. Then blocks corresponding to the concrete domains of one set of the family can become structurally equal (see chapter V). Suppose sets $S_1,\ldots,S_n$ of non-empty concrete domains are given. Whether the concrete domains of $S_0 = \bigcup_{i=1}^{n} S_i$ satisfy the conditions (5.44)-(5.47) must be checked first. If they do, then for all $B_1, B_2 \in S_0$ $(B_1 \neq B_2)$ we have either $\text{ind}(B_1) \cap \text{ind}(B_2) = \emptyset$, $\text{ind}(B_1) \subsetneq \text{ind}(B_2)$ or $\text{ind}(B_2) \subsetneq \text{ind}(B_1)$. Then S_0 must be partitioned into non-empty subsets $T_1,\ldots,T_p$ such that the following three conditions are satisfied:

(i) T_i $(1 \leq i \leq p)$ is a set of mutually disjoint concrete domains.

(ii) T_1 is the maximum subset of S_0 such that for each $B \in S_0 \setminus T_1$ there is a $B' \in T_1$ with $\text{ind}(B) \subseteq \text{ind}(B')$.

(iii) For each $i \geq 2$ there is a $B \in T_j$ $(j<i)$ such that for all $B' \in T_i$, $\text{ind}(B') \subseteq \text{ind}(B)$. Moreover, if for some $B'' \in T_k$, $\text{ind}(B'') \subseteq \text{ind}(B)$, then either $k>i$ and $\text{ind}(B'') \subseteq \text{ind}(B')$ for some $B' \in T_i$, or $k=i$.

A suitable partition of the sparse matrix A exists if an only if TREETEST(T_i) returns *true* for all i $(1 \leq i \leq p)$.

Given a set S of concrete domains, we will now present an algorithm to find the cardinality of the smallest set $S' \supset S$ that has a storage tree. The algo-

rithm can easily be modified such that an actual storage tree for S' can be constructed. Thus, this algorithm finds the first of the two optimizations mentioned in the introduction of this chapter.

Proposition 6.6. *Let S and S' be sets of concrete domains and S' a minimum split of S.*

(i) There is an $r_0 \in \mathbb{Z}$ $(l_1(S) \leq r_0 < u_1(S))$ *or a* $c_0 \in \mathbb{Z}$ $(l_2(S) \leq c_0 < u_2(S))$ *such that*

$$\text{minsplit}(S) = \text{minsplit}(S[:r_0,]) + \text{minsplit}(S[r_0+1:,]) \quad \textit{or}$$
$$\text{minsplit}(S) = \text{minsplit}(S[,:c_0]) + \text{minsplit}(S[,c_0+1:]).$$

(ii) For all $r \in \mathbb{Z}$ *and all* $c \in \mathbb{Z}$

$$\text{minsplit}(S) \leq \text{minsplit}(S[:r,]) + \text{minsplit}(S[r+1:,]) \quad \textit{and}$$
$$\text{minsplit}(S) \leq \text{minsplit}(S[,:c]) + \text{minsplit}(S[,c+1:]).$$

(iii) r_0 *(or* c_0*) as defined in (i) can be chosen such that* $r_0 = u_1(B)$ *(or* $c_0 = u_2(B)$*) for some* $B \in S$.

Proof: (i) and (ii) are consequences of the definition of minsplit and lemma 6.1. The proof of (iii) is as follows. Let $S' > S$ be a minimum split of S. Without loss of generality we assume that

$$\text{minsplit}(S) = |S'| = |S'[:r,]| + |S'[r+1:,]|$$

for some $r \in \mathbb{Z}$ $(l_1(S) \leq r < u_1(S))$. Let $B \in S$ such that $u_1(B) \leq r$ and for all $B' \in S$, either $u_1(B') \leq u_1(B)$ or $u_1(B') \geq r+1$. Such a B will certainly exist. Let $r_0 = u_1(B)$. If $r_0 = r$, we do not need to prove anything. If $r_0 < r$, then

$$\text{minsplit}(S[:r_0,]) \leq \text{minsplit}(S[:r,]). \tag{6.14}$$

Moreover, $|S[r+1:,]| = |S[r_0+1:,]|$ and $S[r+1:,]$ is included in $S[r_0+1:,]$. Let T be a storage tree for $S'[r+1:,]$. Replace each leaf D of T with $l_1(D) = r+1$ by $B'[:u_1(D), l_2(D):u_2(D)]$ where $B' \in S[r_0+1:,]$ and $\text{ind}(D) \subseteq \text{ind}(B')$ It is easy to see that after this modification T is a storage tree for $\text{leaves}(T) > S[r_0+1:,]$. Thus

$$\text{minsplit}(S[r_0+1:,]) \leq \text{minsplit}(S[r+1:,]).$$

With (6.14) this yields

$$\text{minsplit}(S) = \text{minsplit}(S[:r_0,]) + \text{minsplit}(S[r_0+1:,]).$$

□

Proposition 6.6. can be used to design an algorithm to compute minsplit(S) by means of a dynamic programming approach (cf. [45]). If the values of

$$\left.\begin{array}{l}\text{minsplit(S[:r,]), minsplit(S[r+1:,]), minsplit(S[,:c]) and}\\ \text{minsplit(S[,c+1:])}\end{array}\right\} \quad (6.15)$$

are known for each $r=u_1(B)$ $(B\in S)$ and each $c=u_2(B)$ $(B\in S)$, then minsplit(S) can be computed (using proposition 6.6). In order to compute all values (6.15) with proposition 6.6, many more minsplit-values are needed. However, a recursive algorithm would consume too much time because many computations will often be repeated. Therefore, we will store the required minsplit-values in an array with four subscripts. This array will be initialized with zeros and ones.

algorithm 6.2.

```
proc MINSPLIT = (set S)int:
co returns the value minsplit(S) co
  if S=∅    then 0
else let X = {l1(S)-1}∪{u1(B):B∈S} and X=(x0<x1<x2<...<xp);
     let Y = {l2(S)-1}∪{u2(B):B∈S} and Y=(y0<y1<y2<...<yq);
     let ms be an array with 4 subscripts and bounds [0:p-1,1:p,0:q-1,1:q];
     co ms[i,j,h,k] will become minsplit(S_ijhk) with
        S_ijhk = S[x_i+1:x_j, ][ ,y_h+1:y_k]                          co
     initialize ms with 0;
     for all B∈S
      do let i_B=max{i : x_i≤l1(B)-1},  j_B=u1(B),
            h_B=max{j : y_j≤l2(B)-1},  k_B=u2(B);
         for all i,j,h,k with i_B≤i<j≤j_B and h_B≤h<k≤k_B
          do  ms[i,j,h,k]:=1  od
      od;
     for s1 from 1 to p
      do for s2 from  (if s1=1 then 2 else 1 fi)  to q
         do for i from 0 to p-s1
            do for h from 0 to q-s2
               do  if ms[i,i+s1,h,h+s2]=0
                   then co compute minsplit(S_i,i+s1,h,h+s2)          co
                        int minim:=+∞;
```

```
                    for r from i+1 to i+s1-1
                     do int submin=ms[i,r,h,h+s2]+ms[r,i+s1,h,h+s2];
                        if submin<minim then minim := submin fi
                     od;
                    for c from h+1 to h+s2-1
                     do int submin=ms[i,i+s1,h,c]+ms[i,i+s1,c,h+s2];
                        if submin<minim then minim := submin fi
                     od;
                    ms[i,i+s1,h,h+s2] := minim
                fi
              od
           od
        od
     od;
     ms[0,p,0,q]
fi
```

Proposition 6.7. *The time used by algorithm 6.2 is bounded by* $O(|S|^5)$.

Proof: Let S, p, q and *ms* as in the algorithm. The initialization of *ms* requires at most $O(|S|^4)$ time. For each i ($0 \leq i \leq p-1$), j ($i<j \leq p$), h ($0 \leq h \leq q-1$) and k ($h<k \leq q$) *ms*[i,j,h,k] must be computed. Thus the value of $\frac{pq(p+1)(q+1)}{4}$ elements of *ms* must be computed. The computation of one of these values takes at most time $O(p+q)$. Thus the time used by the algorithm is bounded by $O(p.p^2q^2+q.p^2q^2)=O(|S|^5)$.

□

Remark 6.2. Algorithm 6.2 is a general application of the dynamic programming method. For a number of dynamic programming problems a more efficient algorithm is possible (cf. [48] and [81]). The accelaration is based on a property of the function to be minimized. Minsplit would need to satisfy the following property (as well as the similar property for "vertical slices"), in order that the acceleration as proposed in [79] could be incorporated in algorithm 6.2:

Let S be a non-empty set of concrete domains and $u'=u_1(S)-1$. Let r_0 be the smallest number such that

$$\text{minsplit}(S[:r_0,]) + \text{minsplit}(S[r_0+1:,]) \leq \leq \text{minsplit}(S[:r,]) + \text{minsplit}(S[r+1:,])$$

for all r ($l_1(S) \leq r < u_1(S)$). Let s_0 be the smallest number such that for all s ($l_1(S) \leq s < u_1(S)$ [:u',])

minsplit(S[:s_0,]) + minsplit(S[s_0+1:u',]) $\leq$

$\leq$ minsplit(S[:s,]) + minsplit(S[s+1:u',]).

Then $s_0 \leq r_0$.

However, the function minsplit does not have this property. For example, let S be the set of concrete domains as given in fig. 6.4, with $u_1(S)=10$ and $l_1(S)=1$. Then $r_0=3$ and $s_0=4$.

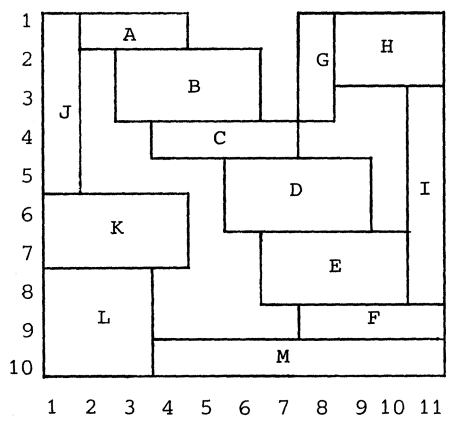


fig. 6.4. The set used in remark 6.2.

VI.3 SPARSE MATRIX STORAGE TREES OF MINIMUM HEIGHT.

In the preceding section we saw that an optimum recursive partition of a sparse matrix A can be obtained in polynomial time. The criterion of optimality used here concerns the number of *mat*s (*diagmat*s) in a sparse matrix storage tree for A as given in V.3. However, we will see that the "optimality" of a recursive partition of A in this sense does not mean that each (virtual or concrete) element of A can be retrieved efficiently. Because the number of direct subblocks of a block of a sparse matrix storage tree for A can be arbitrarily large (i.e., the degree of branching in a sparse matrix storage tree is not bounded by a fixed constant), the retrieval time

of the $(i,j)^{th}$ element α_{ij} of A is not really linearly dependent on the length of the search path of α_{ij}. Therefore, it is more realistic to measure the retrieval time of the leaf containing α_{ij} in a sparse matrix storage tree for A in which each vertex has at most two sons (a binary tree). Obviously, if there is a sparse matrix storage tree T for A, then there is a binary sparse matrix storage tree for A with at most as many leaves as T. The retrieval time of α_{ij} is then linearly dependent on the length of its search path in such a binary sparse matrix storage tree T and therefore, is bounded by the height of T.
In this section we will deal with the following problem:

> Given a set S of concrete domains. Find an S'⊃S and a binary storage tree T for S' such that for each S"⊃S and for each binary storage tree T" for S" we have height(T)≦height(T").

We will see in theorem 6.2 that there are sets S for which a tradeoff exists between the height of a storage tree for an S'⊃S and the cardinality of S' and hence a tradeoff between the maximal retrieval time and storage requirements of sparse matrices with a sparsity pattern represented by S.

DEFINITION 6.9. Let S be a set of concrete domains and T a storage tree for S. T is a binary storage tree for S if each vertex of T has either two or zero sons.

DEFINITION 6.10. Let S be a set of concrete domains. Then the functions minheight and mmh are defined as follows:

$$\text{minheight}(S) = \min\{\text{height}(T) : T \text{ is a binary storage tree for } S\},$$

$$\text{mmh}(S) = \min\{\text{minheight}(S') : S' \supset S\}$$

with $\text{minheight}(\emptyset)=-1$ and $\min(\emptyset)=+\infty$.

Note that mmh(S) may well be smaller than minheight(S). Observe that for an S'⊃S with mmh(S)=minheight(S') a storage tree T for S' with minheight(S')= height(T) determines for a sparse matrix A with a sparsity pattern represented by S a recursive partition P that minimizes the length of the longest search path in A over all recursive partitions of A. Thus, P yields a sparse matrix storage tree for A that is optimal according to the second criterion of optimality mentioned in the introduction of this chapter.

Proposition 6.8. *Let S be a non-empty set of concrete domains. Then* $mmh(S) \leq 2\lceil {}^2\log|S|\rceil$.

Proof: Let $S' \succ S$ be as defined in the proof of proposition 6.4. Obviously S' can be stored in a binary storage tree of height at most $\lceil {}^2\log|S|\rceil + \lceil {}^2\log|S|\rceil$. Therefore:

$$mmh(S) \leq minheight(S') \leq 2\lceil {}^2\log|S|\rceil.$$

□

Now we will develop an algorithm that, given S, computes mmh(S). Without proof we state:

Proposition 6.9. *Let S be a set of concrete domains.*

(i) For each $S' \subseteq S$ $minheight(S') \leq minheight(S)$, $mmh(S') \leq mmh(S)$.

(ii) For each S' included in S $minheight(S') \leq minheight(S)$, $mmh(S') \leq mmh(S)$.

DEFINITION 6.11. Let S be a set of concrete domains.

(i) $r_0 \in \mathbb{Z}$ is a minimal horizontal split line of S if for all $r \in \mathbb{Z}$
$\max\{mmh(S[:r_0,]), mmh(S[r_0+1:,])\} \leq \max\{mmh(S[:r,]), mmh(S[r+1:,])\}$.

(ii) $c_0 \in \mathbb{Z}$ is a minimal vertical split line of S if for all $c \in \mathbb{Z}$
$\max\{mmh(S[,:c_0]), mmh(S[,c_0+1:])\} \leq \max\{mmh(S[,:c]), mmh(S[,c+1:])\}$.

DEFINITION 6.12. Let S be a set of concrete domains. For i=1,2 we define

$$glb_i(S) = \max\{l_i(B):B \in S\} \qquad \max(\emptyset) = -\infty$$
$$lub_i(S) = \min\{u_i(B):B \in S\} \qquad \min(\emptyset) = +\infty.$$

For minimal horizontal and vertical split lines one can prove a number of completely similar propositions. We will state and prove them only for minimal horizontal split lines.

Proposition 6.10. *Let S be a set of concrete domains. Let* r_0 *and* $r_1 \geq r_0$ *be minimal horizontal split lines of S. Then each* $r \in \mathbb{Z}$ $(r_0 \leq r \leq r_1)$ *is a minimal horizontal split line of S.*

Proof: Let $r \in \mathbb{Z}$ with $r_0 \leq r \leq r_1$.

Let $h = \max\{mmh(S[:r_0,]), mmh(S[r_0+1:,])\}$.

Then $h = \max\{mmh(S[:r_1,]), mmh(S[r_1+1:,])\}$. $S[:r,]$ and $S[r+1:,]$ are included in $S[:r_1,]$ and $S[r_0+1:,]$ respectively. Then

$$\max\{mmh(S[:r,]), mmh(S[r+1:,])\} \leq$$
$$\leq \max\{mmh(S[:r_1,]), mmh(S[r_0+1:,])\} \leq \max\{h,h\} = h.$$

□

Proposition 6.11. *Let S be a set of concrete domains. Let* r_0 *be the smallest integer that is a minimal horizontal split line of S. If such an* r_0 *exists, then there is a* $B \in S$ *such that* $r_0 = u_1(B)$.

Proof: Analogous to the proof of proposition 6.6(iii). □

In the same way the largest integer r_1 that is a minimal horizontal split line equals $r_1 = l_1(B)-1$ for some $B \in S$, if r_1 exists.

Proposition 6.12. *Let S be a set of concrete domains.*
Let $S_1 = S[\,:glb_1(S)-1,\,]$, $S_2 = S[lub_1(S)+1:\,,\,]$. *Let* r_0 *and* r_1 *be the smallest and largest integers that are minimal horizontal split lines of* S_1, *and let* s_0 *and* s_1 *be the smallest and largest integers that are minimal horizontal split lines of* S_2, *respectively.* $r_0 = -\infty$ *and* $r_1 = +\infty$ *if each* $r \in \mathbb{Z}$ *is a minimal horizontal split line of* S_1, *and* $s_0 = -\infty$ *and* $s_1 = +\infty$ *if each* $s \in \mathbb{Z}$ *is a minimal horizontal split line of* S_2.
Let $h_1 = \max\{mmh(S_1[:r_0,\,]),\ mmh(S_1[r_0+1:,\,])\} \geq 0$ *and*
$h_2 = \max\{mmh(S_2[:s_0,\,]),\ mmh(S_2[s_0+1:,\,])\} \geq 0$.

(i) Suppose $h_1 = h_2$ *and* $-\infty < s_0 \leq r_1 < \infty$. t *is a minimal horizontal split line of S if and only if* $s_0 \leq t \leq r_1$.

(ii) Suppose $h_1 \geq h_2$, $r_0 > -\infty$ *and* $mmh(S_2) \leq h_1$. t *is a minimal horizontal split line of S if and only if* $r_0 \leq t \leq r_1$.

(iii) Suppose $h_1 \geq h_2$ *and* $mmh(S_2) \leq mmh(S_1) = h_1$. *If there is a* $c \in \mathbb{Z}$ *such that*

$$\max\{mmh(S[\,,:c]),\ mmh(S[\,,c+1:])\} = h_1 - 1$$

then each $t \in \mathbb{Z}$ *is a minimal horizontal split line of S, otherwise* t *is a minimal horizontal split line of S if and only if* $lub_1(S) \leq t \leq glb_1(S)-1$

(iv) Suppose $h_1 = h_2$ *and* $s_0 > r_1$. *If there is a* $c \in \mathbb{Z}$ *such that*

$$\max\{mmh(S[\,,:c]),\ mmh(S[\,,c+1:])\} = h_1$$

then each $t \in \mathbb{Z}$ *is a minimal horizontal split line of S, otherwise* t *is a minimal horizontal split line of S if and only if* $lub_1(S) \leq t \leq glb_1(S)-1$

Proof:

(i) $\Leftarrow$: Let $t \in \mathbb{Z}$ with $s_0 \leq t \leq r_1$. Then

$$mmh(S[:t,\,]) \leq mmh(S[:r_1,\,]) = mmh(S_1[:r_1,\,]) \leq h_1,$$

$$mmh(S[t+1:,\,]) \leq mmh(S[s_0+1:,\,]) = mmh(S_2[s_0+1:,\,]) \leq h_1,$$

and thus $\max\{mmh(S[:t,\,]),\ mmh(S[t+1:,\,])\} \leq h_1$.

With proposition 6.9, t is a minimal horizontal split line of S.

⇒: Suppose t is a minimal horizontal split line of S. Then

$$\max\{mmh(S[:t,\ ]),\ mmh(S[t+1:,\ ])\} = h_1$$

and therefore, t is a minimal horizontal split line of both S_1 and S_2. Thus $s_0 \leq t \leq r_1$.

(ii) $r_0 > -\infty$, thus $lub_1(S_2) \leq r_0 \leq r_1 \leq glb_1(S)-1$. With $mmh(S_2) \leq h_1$, $lub_1(S)$ is a minimal horizontal split line of S and therefore $r_0 = lub_1(S)$ and $mmh(S_1) = h_1+1$.

⇐: Suppose $r_0 \leq t \leq r_1$. Then

$$\max\{mmh(S[:t,\ ]),\ mmh(S[t+1:,\ ])\} \leq$$
$$\leq \max\{mmh(S_1[:t,\ ]),\ mmh(S_2)\} \leq \max\{h_1, h_1\} = h_1.$$

Thus t is a minimal horizontal split line of S.

⇒: Suppose $t < r_0$. Then $mmh(S[t+1:,\ ]) \geq mmh(S_1[t+1:,\ ])$. Because $t < lub_1(S_1)$, $mmh(S_1[t+1:,\ ]) = mmh(S_1) = h_1+1$. Thus t is not a minimal horizontal split line of S.

Suppose $t > r_1$. Then $mmh(S[:t,\ ]) > h_1$. Thus t is not a minimal horizontal split line of S.

(iii) Let $h_1 \geq h_2$ and $mmh(S_2) \leq mmh(S_1) = h_1$. If there is a $c \in \mathbb{Z}$ with

$$\max\{mmh(S[\ ,:c]),\ mmh(S[\ ,c+1:])\} = h_1-1,$$

then $mmh(S) = h_1$ and clearly each $t \in \mathbb{Z}$ is a minimal horizontal split line of S.

If such a c does not exist, $mmh(S) = h_1+1$ and t is a minimal horizontal split line of S if and only if $lub_1(S) \leq t \leq glb_1(S)-1$.

(iv) $h_1 = h_2$ and $s_0 > r_1$. Then $r_1 < +\infty$ and $s_0 > -\infty$. Thus $mmh(S_1) = mmh(S_2) = h_1+1$ and clearly $h_1+1 \leq mmh(S) \leq h_1+2$.

Suppose there is a $t \in \mathbb{Z}$ with

$$\max\{mmh(S[:t,\ ]),\ mmh(S[t+1:,\ ])\} \leq h_1.$$

Then $s_0 \leq t \leq s_1$ and $r_0 \leq t \leq r_1$. Contradiction. Then for each $t \in \mathbb{Z}$ with $lub_1(S) \leq t \leq glb_1(S)-1$ we have $\max\{mmh(S[:t,\ ]),\ mmh(S[t+1:,\ ])\} = h_1+1$. If a c exists with $\max\{mmh(S[\ ,:c]),\ mmh(S[\ ,c+1:])\} = h_1$, then $mmh(S) = h_1+1$ and (iv) is proved. Otherwise $mmh(S) = h_1+2$ and t is a minimal horizontal split line of S implies $t > -\infty$ and $t < +\infty$ thus $lub_1(S) \leq t \leq glb_1(S)-1$.

□

Corollary 6.1. *Let* S, S_i $(1 \leq i \leq 2)$, r_0, r_1, s_0, s_1, h_1, h_2 *as in proposition 6.12. In the four cases equal to those of proposition 6.12* mmh(S) *has the following value:*

(i) $\text{mmh}(S) = \text{mmh}(S_1) = \text{mmh}(S_2)$,

(ii) $\text{mmh}(S) = \text{mmh}(S_1)$,

(iii) if each $r \in \mathbb{Z}$ *is a minimal horizontal split line of* S, *then* $\text{mmh}(S) = \text{mmh}(S_1)$, *otherwise* $\text{mmh}(S) = \text{mmh}(S_1)+1$,

(iv) equal to (iii).

Now we will sketch an algorithm to compute mmh(S).

Let $\{x_0 < x_1 < \ldots < x_p\} = \{l_1(B)-1 : B \in S\} \cup \{u_1(B) : B \in S\}$,

$\{y_0 < y_1 < \ldots < y_q\} = \{l_2(B)-1 : B \in S\} \cup \{u_2(B) : B \in S\}$.

Let $S_{ijhk} = S[x_i+1 : x_j, \;][\; , y_h+1 : y_k]$.

For each S_{ijhk} $(0 \leq i < j \leq p, \; 0 \leq h < k \leq q)$ we will compute and store the following labels:

(i) iglb1, ilub1, iglb2, ilub2 defined as

$$x_{iglb1}+1 = \text{glb}_1(S_{ijhk}), \quad x_{ilub1} = \text{lub}_1(S_{ijhk}),$$

$$y_{iglb2}+1 = \text{glb}_2(S_{ijhk}), \quad y_{ilub2} = \text{lub}_2(S_{ijhk}),$$

(ii) the sets of integers that are minimal horizontal and vertical split lines of S_{ijhk}. These can be stored by means of four integers r_0, r_1, c_0 and c_1,

(iii) $\text{mmh}(S_{ijhk})$,

(iv) a boolean EMPTY to denote whether S_{ijhk} is empty or not.

Initialize all labels with a proper value. For each $B \in S$ give proper values to the labels of S_{ijhk} with

$$x_i \geq l_1(B)-1, \quad x_j \leq u_1(B), \quad y_h \geq l_2(B)-1 \quad \text{and} \quad y_k \leq u_2(B).$$

Then assign the right values to the labels of $S_{i,i+1,h,k}$ and $S_{i,j,h,h+1}$ for all i, j, h, k. Observe that the concrete domains of these sets have the natural linear ordering because they are mutually disjoint.

Then, using the same order as in algorithm 6.2, the labels of all other S_{ijhk} can be computed from the labels of $S_{i,\text{iglb1}(S_{ijhk}),h,k}$, $S_{\text{ilub1}(S_{ijhk}),j,h,k}$, $S_{i,j,h,\text{iglb2}(S_{ijhk})}$ and $S_{i,j,\text{ilub2}(S_{ijhk}),k}$ by means of proposition 6.12, corollary 6.1 and (6.16).

$$\left.\begin{aligned}
\text{iglb1}(S_{ijhk}) &= \max\{\text{iglb1}(S_{i,j,h,h+1}),\ \text{iglb1}(S_{i,j,h+1,k})\},\\
\text{iglb2}(S_{ijhk}) &= \max\{\text{iglb2}(S_{i,i+1,h,k}),\ \text{iglb2}(S_{i+1,j,h,k})\},\\
\text{ilub1}(S_{ijhk}) &= \min\{\text{ilub1}(S_{i,j,h,h+1}),\ \text{ilub1}(S_{i,j,h+1,k})\},\\
\text{ilub2}(S_{ijhk}) &= \min\{\text{ilub2}(S_{i,i+1,h,k}),\ \text{ilub2}(S_{i+1,j,h,k})\}.
\end{aligned}\right\} \quad (6.16)$$

Proposition 6.13. *Let S be a set of concrete domains. Then* mmh(S) *can be computed in time* $O(|S|^4)$.

Proof: The initialization takes time at most $O(|S|^4)$. To compute the labels of one S_{ijhk} using proposition 6.12, corollary 6.1 and (6.16) requires constant time. There are at most $O(|S|^4)$ S_{ijhk}'s for which this computation will be performed. Thus, the total time used by this algorithm is bounded by $O(|S|^4)$.

□

In proposition 6.8 we saw that $\text{mmh}(S) \leq 2\lceil {}^2\log|S|\rceil$. The question arises how large the cardinality can be of some $S' \rhd S$ with minheight(S')=mmh(S). As already stated (see VI.1) the storage requirements of a storage tree for S' are linear in the cardinality of S'. The following theorem states that there are sets S such that there is a small difference in the cardinalities of two sets $S', S'' \rhd S$ if and only if there is a small difference of minheight(S') and minheight(S"). A consequence of this theorem is that the cardinality of an $S' \rhd S$ with minheigth(S')=mmh(S) can be rather large compared with the cardinality of S.

Theorem 6.2. *There is a sequence* $(S_k)_{k\geq 2}$ *of sets of concrete domains with* $|S_k|=3k$ $(k\geq 2)$ *such that for each* $k\geq 2$ *and* $s\geq 0$

$$\min\{\text{minheight}(S') : S' \rhd S_k \text{ and } |S'| \leq |S_k|+s\} \geq k+2-s.$$

Proof: Let $A_i = (2k-2i+1,\ 2k-2i+2,\ 2k-2i+1,\ 2k-2i+2)$ $\quad 1\leq i\leq k$

$B_i = (2k-2i+2,\ 2k-2i+2,\ 2k-2i+4,\ 2k+2)$ $\quad 1\leq i\leq k$

$C_i = (2k-2i+4,\ 2k+2\ ,\ 2k-2i+2,\ 2k-2i+2)$ $\quad 1\leq i\leq k$

Let $S_k = \{A_1, A_2, \ldots, A_k, B_1, B_2, \ldots, B_k, C_1, C_2, \ldots, C_k\}$ $\quad k\geq 1$.

Let $S_0=\emptyset$. For S_7, see fig. 6.5.

We define

$$\text{mh}(S_k,s) = \min\{\text{minheight}(S') : S' \rhd S_k \text{ and } |S'| \leq |S_k|+s\}.$$

Claim 1: $\text{mh}(S_k,s) \geq \text{mh}(S_p,s)$ for all $1\leq p\leq k$.

Proof of claim 1: Let $1 \leq p \leq k$. S_p is included in S_k. From $S_k' \succ S_k$ with $|S_k'| \leq |S_k|+s$ an $S_p' \succ S_p$ can be derived such that

(i) $\text{minheight}(S_p') \leq \text{minheight}(S_k')$,

(ii) $|S_p'| \leq |S_p|+s$.

Then $mh(S_k,s) \geq mh(S_p,s)$.

Claim 2: $mh(S_k,s) \geq mh(S_k,s+1)$ for each $k \geq 1$, $s \geq 0$.

Proof of claim 2:

Trivial because $\{\text{minheight}(S') : S' \succ S_k \text{ and } |S'| \leq |S_k|+s\}$ is a subset of $\{\text{minheight}(S') : S' \succ S_k \text{ and } |S'| \leq |S_k|+s+1\}$.

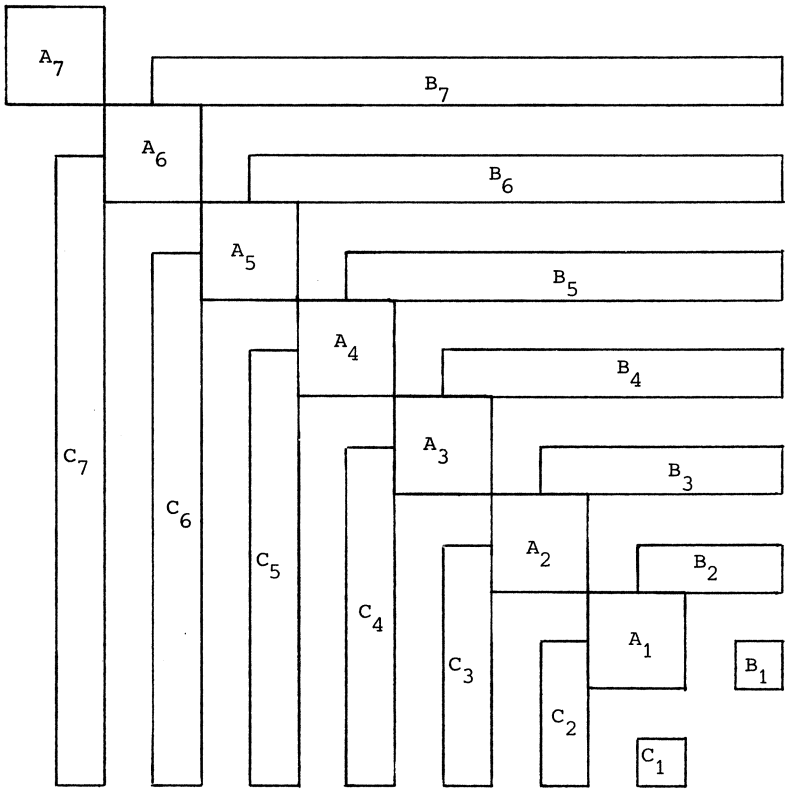


fig. 6.5. S_k for k=7 as used in the proof of theorem 6.2.

Claim 3: $mh(S_k,0)=k+2$ for each $k\geq 2$. (6.17)

Proof of claim 3: Clearly $mh(S_k,0)=\text{minheight}(S_k)$. We will prove this claim by induction. For k=1, $mh(S_k,0)=2$ whereas $mh(S_k,0)=4$ for k=2. Suppose (6.17) holds for k=2,3,...,m. Let T_{m+1} be a binary storage tree for S_{m+1}. Obviously, such a T_{m+1} exists. Let the root of T_{m+1} have two sons t_1 and t_2. Because of symmetry we can assume that there is a $c\in\mathbb{Z}$ such that $T_{m+1}(t_1)$ is a binary storage tree for $S_{m+1}[\ ,:c]$ and $T_{m+1}(t_2)$ for $S_{m+1}[\ ,c+1:]$.
Because $\text{leaves}(T_{m+1})=S_{m+1}$, c must be equal to c=2. Then $\text{leaves}(T_{m+1}(t_2))$ includes S_m (if S_m is shifted right and downwards in the plane) and therefore we have:

$$\text{height}(T_{m+1})\geq 1+mh(S_m,0) \text{ for each binary storage tree } T_{m+1} \text{ for } S_{m+1}.$$

Thus $mh(S_{m+1},0)\geq 1+mh(S_m,0)$ and by induction $mh(S_{m+1},0)\geq(m+1)+2$. Moreover, there is a binary storage tree for S_{m+1} with height m+3. Hence claim 3 is proved.

To prove the theorem we will induct on k and s. The theorem holds for $k\geq 2$ and s=0 (claim 3). Clearly it holds for k=2 and $s\geq 0$. Suppose it holds for s=0,...,q-1 ($q\geq 1$) and k=2,...,m ($m\geq 2$). It suffices to prove now that $mh(S_m,q)\geq m+2-q$.
Let T be a binary storage tree for some $S_m'>S_m$ with $|S_m'|\leq|S_m|+q$. If such a T does not exist, $mh(S_m,q)=+\infty$. Thus we assume that T exists. Let the root of T have two sons t_1 and t_2; let $T_1=T(t_1)$ and $T_2=T(t_2)$. Without loss of generality we can assume that there is a $c\in\mathbb{Z}$ such that $\text{leaves}(T_1)>S_m[\ ,:c]$ and $\text{leaves}(T_2)>S_m[\ ,c+1:]$. With a reasoning as in the proof of proposition 6.11 we can assume that c=2,4,6,8,..., i.e., $c=u_2(A_j)$ for some j ($1\leq j\leq m$). Then $S_m[\ ,:c]$ includes S_{m-j} and $S_m[\ ,c+1:]$ includes S_{j-1}. Because $|\text{leaves}(T)|\leq|S_m|+q$ we have $|\text{leaves}(T_1)|+|\text{leaves}(T_2)|\leq|S_m|+q$. Moreover, $|\text{leaves}(T_2)|\geq m+2(j-1)$ and $|\text{leaves}(T_1)|\geq 3m-3(j-1)-1$. Thus $m+2(j-1)+3m-3(j-1)-1\leq 3m+q$ and we have

$$m-j\leq q.$$

If for some p ($0\leq p\leq q-m+j$), $|\text{leaves}(T_2)|=|S_m[\ ,c+1:]|+p$ then $|\text{leaves}(T_1)|\leq|S_m[\ ,:c]|+q-m+j-p$. Because $S_m[\ ,c+1:]$ includes S_{j-1}, we have that

$$\text{height}(T_2)\geq mh(S_{j-1},p). \tag{6.18}$$

$S_m[\ ,:c]$ includes S_{m-j} and hence

$$\text{height}(T_1) \geq mh(S_{m-j}, q-m+j-p). \tag{6.19}$$

(6.18) and (6.19) lead to

$$\text{height}(T) = 1+\max\{\text{height}(T_2), \text{height}(T_1)\} \geq$$

$$\geq 1+\min\{\min\{\max\{mh(S_{j-1},p), mh(S_{m-j},q-m+j-p)\}: 0\leq p\leq q-m+j\}: 1\leq j\leq m \text{ and } q-m+j\geq 0\}.$$

This holds for every binary storage tree T with at most $|S_m|+q$ leaves and leaves(T)>S_m. Thus:

$$mh(S_m,q) \geq$$

$$\geq 1+\min\{\min\{\max\{mh(S_{j-1},p), mh(S_{m-j},q-m+j-p)\}: 0\leq p\leq q-m+j\}: 1\leq j\leq m \text{ and } q-m+j\geq 0\}$$

and by induction:

$$mh(S_m,q) \geq 1+\min\{\min\{\max\{j+1-p, 2m+2-q+p-2j\}: 0\leq p\leq q-m+j\}: 1\leq j\leq m \text{ and } j\geq m-q\}.$$

Now we will find a lowerbound for

$$V_1 = \min\{\min\{\max\{j+1-p, 2m+2-q+p-2j\}: 0\leq p\leq q-m+j\}: 1\leq j\leq m \text{ and } j\geq q+1 \text{ and } j\geq m-q\}$$

With $j\geq q+1$ we have $3j+q-2m-1\geq 2q-2m+2j$. With $2p\leq 2q-2m+2j$ this leads to $3j+q-2m-1\geq 2p$ and thus $j+1-p\geq 2m-q+p-2j+2$. Then

$$V_1 = \min\{\min\{j+1-p : 0\leq p\leq q-m+j\} : 1\leq j\leq m \text{ and } j\geq q+1 \text{ and } j\geq m-q\}$$

$$= \min\{j+1-q+m-j : 1\leq j\leq m \text{ and } j\geq q+1 \text{ and } j\geq m-q\}$$

$$= \min\{m+1-q : 1\leq j\leq m \text{ and } j\geq q+1 \text{ and } j\geq m-q\}$$

$$\geq m+1-q \quad (\text{if } \min(\emptyset) = +\infty).$$

Next we will find a lowerbound for

$$V_2 = \min\{\min\{\max\{j+1-p, 2m+2-q+p-2j\} : 0\leq p\leq q-m+j\} : 1\leq j\leq m \text{ and } m-q\leq j\leq q\}.$$

With $j<q+1$, $3j+q-2m-1<2(q+j-m)$.

If $2p\leq 3j+q-2m-1$ then $j+1-p\geq 2m+2-q+p-2j$.

If $2p>3j+q-2m-1$ then $j+1-p<2m+2-q+p-2j$. Substituting this in V_2 yields

$$V_2 = \min\{\min\{\min\{j+1-p : 0\leq 2p\leq 3j+q-2m-1\}$$
$$,\min\{2m+2-q+p-2j : 3j+q-2m\leq 2p\leq 2q-2m+2j\}$$
$$\} : 1\leq j\leq m \text{ and } m-q\leq j\leq q\}$$
$$= \min\{\min\{j+1-\lfloor\frac{3j+q-2m-1}{2}\rfloor,\ 2m+2-q-2j+\lceil\frac{3j+q-2m}{2}\rceil\} : 1\leq j\leq m \text{ and } m-q\leq j\leq q\}$$
$$= \min\{\min\{m+1-\lfloor\frac{j+q-1}{2}\rfloor,\ m-q+2+\lceil\frac{q-j}{2}\rceil\} : 1\leq j\leq m \text{ and } m-q\leq j\leq q\}$$
$$\geq \min\{\min\{m+1-\lfloor\frac{2q-1}{2}\rfloor,\ m-q+2\} : 1\leq j\leq m \text{ and } m-q\leq j\leq q\}$$
$$= \min\{m+1-(q-1),\ m-q+2\} = m+2-q.$$

Then $mh(S_m,q) \geq 1+\min\{V_1,V_2\} \geq m+2-q$.
Thus the theorem holds for $s=q$ and $k=m$ and by induction it holds for every $s\geq 0$ and $k\geq 2$.

□

Corollary 6.2. Let $(S_k)_{k\geq 2}$ be as in theorem 6.2. Let T_k be a binary storage tree for $S_k'>S_k$ such that height $(T_k)\leq 2\lceil{}^2\log|S_k|\rceil$.
Then $|S_k'|\geq|S_k|+\frac{|S_k|}{3}+2-2\lceil{}^2\log|S_k|\rceil$.

Proof: Suppose there is a $k\geq 2$ and a binary storage tree T_k' for $S_k'>S_k$ such that $\text{height}(T_k')\leq 2\lceil{}^2\log|S_k|\rceil$ and $|S_k'|\leq|S_k|+\frac{|S_k|}{3}+2-2\lceil{}^2\log|S_k|\rceil-1$.
Then by theorem 6.2:

$$\text{minheight}(S_k')\geq\frac{|S_k|}{3}+2-\frac{|S_k|}{3}-2+1+2\lceil{}^2\log|S_k|\rceil$$

and by definition

$\text{height}(T_k')\geq 1+2\lceil{}^2\log|S_k|\rceil$. Contradiction.

□

Conclusion. In this chapter we have considered several problems concerning optimal sparse matrix storage trees. We have shown that there are sparse matrices A of n *mat*s of which each sparse matrix storage tree has $\frac{3}{2}$n *mat*s.
We have presented three polynomial time algorithms:

(i) one to test whether an arbitrary sparse matrix A of n *mat*s has a sparse matrix storage tree of n *mat*s,

(ii) one to find for an arbitrary sparse matrix a sparse matrix storage tree with a minimum number of leaves,

(iii) one to find for an arbitrary sparse matrix a binary sparse matrix storage tree of minimum height.

These algorithms run in time $O(n^2)$, $O(n^5)$ and $O(n^4)$, respectively, for sparse matrices consisting of n *mat*s. The exponents of these polynomial time bounds are rather high. It should be interesting to design more efficient algorithms or to prove that no better time bounds can really be achieved.

In theorem 6.2 we have shown that there is a tradeoff between the number of vertices and the height of a binary sparse matrix storage tree: there are matrices A for which a decrease in the height of a sparse matrix storage tree for A can only be obtained by an increase in the number of vertices of the tree.

The results presented in this chapter will also appear in [79].

CHAPTER VII

SUITABILITY OF TORRIX-SPARSE FOR SYMBOLIC LU DECOMPOSITION

In chapter V we have designed a system TORRIX-SPARSE for manipulating sparse matrices. In this chapter we will test the suitability of TORRIX-SPARSE in case it is used to manipulate sparse matrices in the typical numerical application of LU decomposition. In chapter II we explained the LU decomposition of a matrix A as a direct method to solve a linear system

$$Ax = b \qquad A=(\alpha_{ij})_{1\leq i,j\leq n}. \tag{7.1}$$

For additional details concerning the LU decomposition we refer to [78] or any other treatise on numerical linear algebra.

In case the matrix A is sparse the process of solving (7.1) can be built up in five stages (cf. [63]):

(i) Find permutation matrices P and Q such that PAQ has a zero-free main diagonal.

(ii) Find a permutation matrix R such that $RPAQR^{-1}$ is of block lower triangular form.

(iii) Perform a symbolic LU decomposition of $RPAQR^{-1}$, i.e., create matrices L' and U' in computer memory such that L' and U' have the following properties

$l_{ij}\neq 0$ if and only if the $(i,j)^{th}$ element of L' is concrete,

$u_{ij}\neq 0$ if and only if the $(i,j)^{th}$ element of U' is concrete.

Thus the fill (as defined in II.3.1) is determined and matrices are built up in computer memory containing locations for all non-zero elements of the factors L and U. In order to lower the size of the fill the matrix $RPAQR^{-1}$ may be permuted. Then permutation matrices S and T can be found such that $SRPAQR^{-1}T$ and $RPAQR^{-1}$ have the same block lower triangular form. The symbolic LU decomposition will be performed on $SRPAQR^{-1}T$.

(iv) Perform a numerical LU decomposition of $SRPAQR^{-1}T$ and assign values to the concrete elements of L' and U' such that these matrices will become L and U. If this LU decomposition is numerically unstable, additional row- and column-permutations may have to be performed and it may be necessary to perform the preceding stage once again. As for the problem how to determine and to control numerical unstability we refer to [80] and [63].

(v) Backsolve the systems

$$Ly = b \quad \text{and} \quad Ux = y.$$

In case A is symmetric and positive definite, each main diagonal element of A is positive and P=Q=I in stage (i).
Observe that in the stages (i), (ii) and (iii) one does not compute with the scalar values of A; in these stages only the sparsity pattern of A is involved. The symbolic LU decomposition and the determination of the permutation matrices S and T in stage (iii) may be done simultaneously, depending on the algorithm used to solve the latter problem.
This division into stages is suited well for the case that (7.1) must be solved for several matrices with the same sparsity pattern: P, Q, R, S and T can be used in these three stages for all matrices. Often stages (iii) and (iv) are performed simultaneously for the first matrix. Thus, permutation matrices V and W are obtained such that for this matrix A the LU decomposition of $VRPAQR^{-1}W$ is numerically stable. For the subsequent matrices these same five permutation matrices are used and the stages (i) and (ii) will be skipped.

With the data structure provided by TORRIX-SPARSE we are interested in algorithms for the stages (i) to (v) that exploit the block-structured storage scheme as well as the fact that non-zero elements are stored in *mat*s and *diagmat*s. As for the stages (i) to (iii) this means that we are interested in algorithms that are block oriented and in which the sparsity pattern of the matrix is considered a set of concrete domains rather than a set of non-zero matrix elements. Hopefully the time complexity of such algorithms does not depend on the number of non-zero elements but on the number of *mat*s, *diagmat*s and the size of the recursive partition of the matrix A. In VII.1 we will deal with algorithms for stage (i), in VII.2 with algorithms for stage (ii) and in VII.3 with algorithms for the symbolic LU decomposition

in stage (iii). We will not deal with the other stages. Block methods are straightforward from the programming point of view and are already studied well in the literature (cf. [13] and [35]). In all sections we assume that A is non-singular and stored in an *spmat* without structurally equal blocks and that all concrete elements of A are non-zero. It is not necessary to take structurally equal blocks into account because in general they cannot be retained during an LU decomposition.

VII.1 APPLICABILITY OF TORRIX-SPARSE FOR PERMUTING A MATRIX TO ZERO-FREE MAIN DIAGONAL FORM.

In stage (i) (see the introduction of this chapter) permutation matrices P and Q must be found for the given matrix A such that PAQ has a zero-free main diagonal. This problem reduces to the well-known maximum matching problem in bipartite graphs, which has already been studied in literature (cf. e.g. [19], [44] and [51]). Although each algorithm on matrices can be implemented in TORRIX-SPARSE, the implementations are not necessarily efficient (see V.1.2). Therefore, we will present modifications of the algorithm explained in [51] ch. 5 sect. 5 to fit it to the TORRIX-SPARSE system, i.e., the modified algorithm uses the block-structured storage scheme as well as the fact that non-zero elements are stored in *mat*s and *diagmat*s. First we will review the necessary preliminaries and explain the algorithm of [51] ch. 5 sect. 5, then we will present the modifications and finally we will analyze the modified algorithm.

DEFINITION 7.1 (cf. [63]). Let $A=(\alpha_{ij})_{1\leq i,j\leq n}$ be a matrix of scalars. A transversal of A is a set of elements $\{\alpha_{i_1,j_1},\ldots,\alpha_{i_p,j_p}\}$ such that

(i) $\alpha_{i_q,j_q}\neq 0$ $(1\leq q\leq p)$ and

(ii) $i_q\neq i_r$ and $j_q\neq j_r$ $(1\leq q<r\leq p)$.

Every two elements of a transversal of A are in different rows and different columns of A. Thus:

Lemma 7.1. *Let* A *have a transversal of* p *elements. Then there are permutation matrices* P *and* Q *such that the first* p *elements of the main diagonal of* PAQ *are non-zero.*

Lemma 7.2(cf. [22]). *Let* A *be an* n×n *matrix of real numbers. If* A *is non-singular, then* A *contains a transversal of* n *elements.*

Assuming that A is non-singular, the problem of finding a transversal of n elements can also be formulated as a matching problem for bipartite graphs.

DEFINITION 7.2(cf. [19] and [51]). Let $G=(V,E)$ be a graph. A set $F \subseteq E$ is a matching if for all $f,g \in F$ $(f \neq g)$ there is no $x \in V$ such that f and g are both incident to x. F is a maximum matching if for all matchings F' in G, $|F'| \leq |F|$

The maximum matching problem for a bipartite graph $G=(V_R,V_C,E)$ is to find a maximum matching of G (cf. [51]). If G is the bipartite graph for which M(G) has the same sparsity pattern as A (for the definition of M(G) we refer to III.1.1), then a (maximum) transversal of A determines a (maximum) matchin in G and vice versa. There are several algorithms to find a maximum matching in a bipartite graph (see e.g. [19], [44] and [51]). We will concentrate on the algorithm in [51] ch. 5 sect. 5 that reduces the maximum matching problem for bipartite graphs to the maximum network flow problem.

DEFINITION 7.3(cf. [51]). Let $G=(V_R,V_C,E)$ be a bipartite graph and $M \subseteq E$ a matching in G. A path $\pi=(x_0,\ldots,x_{2m+1})$ in G is an augmenting path relative to M if $(x_{2i-1},x_{2i}) \in M$ $(1 \leq i \leq m)$ and no edge of M is incident to x_0 or x_{2m+1}.

THEOREM 7.1(cf. [51]). *Let* $G=(V_R,V_C,E)$ *be a bipartite graph and* $M \subseteq E$ *a matching.* M *is a maximum matching if and only if* G *does not contain an augmenting path relative to* M.

The algorithm presented in [51] ch. 5 sect. 5 consists of a number of steps such that in step k a matching M_k with $|M_k|=k$ will be found. The algorithm starts with the empty matching. In each step k three cases can be distinguished:

Case 1. There is an edge $e \in E$ such that $M_{k-1} \cup \{e\}$ is a matching.

Case 2. There is an augmenting path $\pi=(x_0,x_1,\ldots,x_{2m+1})$ $(m \geq 1)$ relative to M_{k-1}. Then:

$$M_k = (M_{k-1} \setminus \{(x_{2i-1},x_{2i}) : 1 \leq i \leq m\}) \cup \{(x_{2i},x_{2i+1}) : 0 \leq i \leq m\}$$

is a matching and $|M_k|=|M_{k-1}|+1$.

Case 3. Neither case 1 nor case 2 is satisfied. Then M_{k-1} is a maximum matching and the algorithm terminates.

With proper algorithms for the first two cases this algorithm is a polynomial time algorithm (cf. [51]).

To find a maximum transversal of a matrix A we shall reformulate the matching algorithm for the associated bipartite graph in terms of A.

DEFINITION 7.4. Let $A=(\alpha_{ij})_{1\leq i,j\leq n}$ be a matrix of scalars, $p\in\mathbb{N}$ $(0\leq p\leq n)$ and $\alpha_{ii}\neq 0$ $(1\leq i\leq p)$. A p-augmenting path π of A of length m $(m\geq 1)$ is a sequence $(\alpha_{i_0,j_0},\alpha_{i_1,j_1},\ldots,\alpha_{i_{2m-2},j_{2m-2}})$ such that

(i) $\alpha_{i_q,j_q}\neq 0$ $(0\leq q\leq 2m-2)$

(ii) $i_0\leq p<j_0$ and $i_{2m-2}>p\geq j_{2m-2}$

(iii) $i_{2q+1}=j_{2q+1}$ $(0\leq q\leq m-2)$

(iv) $i_q=i_r$ and $q<r$ imply $q=r-1$ and
$j_q=j_r$ and $q<r$ imply $q=r-1$

(v) $i_{2q}=i_{2q+1}$ $(0\leq q\leq m-2)$ and $(j_{2q-1}=j_{2q}$ $(1\leq q\leq m-1)$.

For an example of an 11-augmenting path of length 5, see fig. 7.1. If $\alpha_{ii}\neq 0$ $(1\leq i\leq p)$ then A has a transversal of p elements. If in addition, A has

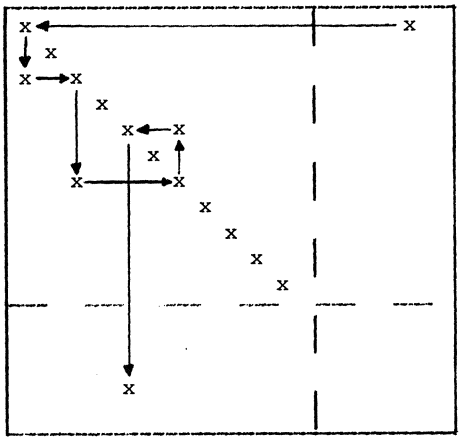


fig. 7.1. An 11-augmenting path of length 5 in a sparse matrix.

a p-augmenting path then A contains a transversal of p+1 elements and (with lemma 7.1) A can be permuted so as to have non-zero values in the first p+1 positions of the main diagonal.

Now we will reformulate the matching algorithm in terms of the matrix A

(which we assume to be non-singular). The matrix obtained in step k ($1 \leq k \leq n$) will be denoted as $A^{(k)}$. It is a permutation of A and it has non-zero elements in the first k positions of its main diagonal. Let $A^{(0)}=A$ and $A^{(k)}=(\alpha_{ij}^{k})_{1 \leq i,j \leq n}$ with $\alpha_{ii}^{k} \neq 0$ $(1 \leq i \leq k)$. In step k $(1 \leq k \leq n)$ we distinguish the following two cases:
Case 1. There is an (i,j) with $i,j \geq k$ and $\alpha_{ij}^{k-1} \neq 0$. Then this element can be permuted to position (k,k), obtaining $A^{(k)}$.
Case 2. If $A^{(k-1)}$ does not satisfy case 1, then there is a (k-1)-augmenting path in $A^{(k-1)}$. Thus A contains a transversal of k elements and therefore can be permuted (obtaining $A^{(k)}$) such that the main diagonal of $A^{(k)}$ has non-zero values in its first k positions.
This algorithm is a polynomial time algorithm because it is a reformulation of the polynomial time matching algorithm.

In [40] several heuristics are proposed to postpone the necessity of exploring case 2. These heuristics involve the choice of a non-zero element in $A^{(k-1)}$ $(1 \leq k \leq n)$ if $A^{(k-1)}$ contains several non-zero elements with both indices greater than k-1. The heuristics were designed with the purpose of yielding an $A^{(k)}$ with many non-zero elements with both indices greater than k. With A stored in an *spmat* *a* as defined in TORRIX-SPARSE we propose the following two heuristics that may be more suitable for this storage scheme:

(i) Heuristic H. If $A^{(k-1)}$ contains an element with both indices greater than k-1, then this element is a concrete element of a *mat* *m* or *diagmat* *d*. If *m* contains r rows and c columns and all elements of *m* have both indices (with respect to $A^{(k-1)}$) greater than k, then it is worthwhile to permute *m* to the main diagonal of $A^{(k-1)}$. Thus, if the first p elements of the main diagonal of $A^{(k-1)}$ are non-zero, then the first p+min(r,c) elements of the main diagonal of $A^{(k)}$ are non-zero. Permuting $A^{(k-1)}$ such that *m* will be in the right position, hardly requires more computation time than exchanging two columns and two rows, because block exchanges can be applied. A similar action can be performed if $A^{(k-1)}$ contains a *diagmat* in its right lower part.

(ii) Weight heuristic. Suppose the *spmat* *a* is a block B partitioned into $(B_{ij})_{1 \leq i \leq p,1 \leq j \leq q}$. Each block B_{ij} can be considered a sparse matrix by itself and can be searched for transversals. If each block B_{ij} $(1 \leq i \leq p,1 \leq j \leq q)$ contains a transversal of at least w_{ij} elements, then we can try to find a transversal of the matrix $(w_{ij})_{1 \leq i \leq p,1 \leq j \leq q}$ with maximum sum. This problem

is called the weighted bipartite matching problem and a polynomial time algorithm for it has been given in [51].

Lemma 7.3. *Let* A *be a matrix partitioned into* $(A_{ij})_{1\leq i\leq p,1\leq j\leq q}$. *Let* A_{ij} $(1\leq i\leq p,1\leq j\leq q)$ *have a transversal of* w_{ij} *elements. Let* $W=(w_{ij})_{1\leq i\leq p,1\leq j\leq q}$ *have a weighted matching (or transversal) with sum* w. *Then* A *has a transversal of* w *elements.*

Proof: Let $\{w_{i_1,j_1},\ldots,w_{i_r,j_r}\}$ be the transversal of W with maximum sum w. Then the union of the transversals of A_{i_h,j_h} $(1\leq h\leq r)$ is a transversal of A and contains w elements.

□

The algorithm for the weight heuristic can now easily be explained. After step k $(k\geq 0)$ the first N_k elements of the main diagonal of the *spmat* *a* are non-zero (with $N_k>N_{k-1}$ $(k\geq 1)$). Suppose after step k *a* contains a *block* B with bounds $[N_k+1:n,N_k+1:n]$. Then compute the weight w_B of B bottom-up using the weighted matching algorithm. The weight of a *mat* *m* equals 1+min(*1 upb m - 1 lwb m, 2 upb m - 2 lwb m*); the weight of a *diagmat* *d* is at least max{*upb d[i] - lwb d[i] +1 : lwb d* ≤i≤ *upb d*}, but a better approximation of the cardinality of a maximum transversal could be used also (provided for example by the matching algorithm); the weight of a partitioned *block* can be computed with the weighted matching algorithm applied to the matrix of weights of its direct subblocks.

Let w_B be the weight of B. If $w_B=0$, then B does not contain any non-zero element and one must search for an augmenting path in *a*. If $w_B\neq 0$, then with a number of cyclic row-permutations, cyclic column-permutations and block exchanges, w_B non-zero elements of B can be permuted to the main diagonal of B such that $N_{k+1}=N_k+w_B$. With this heuristic we may obtain the following:

(i) The search for an augmenting path in *a* is postponed until absolutely necessary. Of course the weighted matching algorithm uses augmenting paths, but in the way we applied this algorithm, these are "paths" of direct subblocks. Given a *block*, its direct subblocks can efficiently be accessed.

(ii) Hopefully not too many *mat*s and *diagmat*s are split into smaller ones (i.e., replaced by slices). This may also be of importance for finding the block lower triangular form of an *spmat* (see VII.2).

How large can the subsequent numbers N_k of non-zero elements on the main diagonal be? In this heuristic algorithm the computation time of the weight of a *block* B depends only on the size of the recursive partition of B and of the number of *vec*s in *diagmat*s contained in B. Given a recursive partition of the matrix the total running time of the weight heuristic will be low if the sequence $(N_k)_{k\geq 1}$ increases fast.

THEOREM 7.2. *For each* $\varepsilon>0$ *there is an* $n\in\mathbb{N}$ *and a non-singular* n×n *matrix* A *which can be stored in an* *spmat* *a in such a way that* $N_1<\varepsilon n$.

Proof: Let $\varepsilon>0$, $c\in\mathbb{N}\setminus\{0\}$ and $p\in\mathbb{N}$ with $\frac{1}{2^{p-1}}<\varepsilon$. Let M be any non-singular c×c matrix with non-zero main diagonal elements. Let A be the matrix defined by (see also fig. 7.2):

- A is of size $c.2^p \times c.2^p$,
- A is partitioned into $A=(A_{ij})_{1\leq i,j\leq 2^p}$ with each A_{ij} $(1\leq i,j\leq 2^p)$ of size c×c,
- A_{ij} =M if $1\leq i\leq 2^{p-1}$ and $j=2(i-1)+1$,
  =M if $2^{p-1}<i\leq 2^p$ and $j=2(i-2^{p-1})$,
  =0 otherwise.

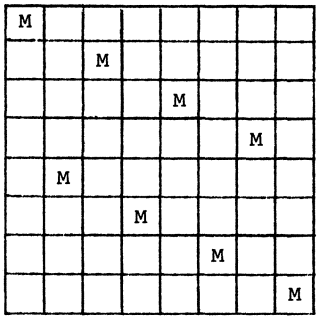


fig. 7.2. A as used in the proof of theorem 7.2 with p=3.

A can be stored in an *spmat* *a* as follows: each non-partitioned *block* of *a* corresponds to one A_{ij} and each partitioned *block* B of *a* has four direct subblocks and the number of rows and columns in B are equal. Without proof we state:

Let B be a subblock of *a* or *a* itself; then there are p_1, p_2 and $q\in\mathbb{N}$ such that the bounds of B are:

$$\text{bnds}(B) = [c.p_1.2^q+1 : c(p_1+1).2^q \ , \ c.p_2.2^q+1 : c(p_2+1).2^q] \qquad (7.2)$$

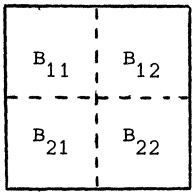
with $0 \leq p_1 < 2^{p-q}$, $0 \leq p_2 < 2^{p-q}$ and $0 \leq q \leq p$.

Claim: Let B be a subblock of α. If B is unequal to α, then weight(B)$\leq$c.

Proof of claim: We will prove this claim by induction on the size of B. Let B have bounds as given in (7.2). Because B is unequal to α, $q<p$.

For p=0, B is not partitioned and is either M or does not contain any non-zero elements. Thus, weight(B)$\leq$c. Suppose (by induction) that the claim holds for all subblocks of α with size at most $c.2^{q-1} \times c.2^{q-1}$ ($q \geq 1$). We have to prove the claim for B with size $c.2^q \times c.2^q$. There are two cases: $p_1.2^q < 2^{p-1}$ and $p_1.2^q \geq 2^{p-1}$. We will only deal with the first case. The second case can be dealt with in a similar way. Thus, let $p_1.2^q < 2^{p-1}$. Then B contains the submatrices A_{ij} ($p_1.2^q+1 \leq i \leq (p_1+1).2^q$, $p_2.2^q+1 \leq j \leq (p_2+1).2^q$). Let B be partitioned into $(B_{ij})_{1 \leq i,j \leq 2}$

B=



We will prove now:

If B_{11} or B_{12} contains an M then B_{21} and B_{22} do not contain any concrete element. Assume B_{11} contains an M. Thus there are

$$i_0, j_0 \ (1 \leq i_0, j_0 \leq 2^{q-1}) \text{ with } 2(p_1.2^q+i_0-1)+1 = p_2.2^q+j_0. \qquad (7.3)$$

With (7.3): $1-2^q \leq j_0-2i_0 = (2.p_1-p_2)2^q-1 \leq 2^{q-1}-2$ and therefore $-1+\frac{2}{2^q} \leq 2.p_1-p_2 \leq \frac{1}{2}-\frac{1}{2^q}$. Because $p_1, p_2 \in \mathbb{N}$ and $q \geq 1$, we have $0 \leq 2p_1-p_2 \leq 0$ and thus $j_0-2i_0=-1$. Assume B_{21} or B_{22} contains an M; thus, there are

$$(i,j) \text{ with } 1 \leq j \leq 2^q, \ 2^{q-1}+1 \leq i \leq 2^q \text{ and } 2(p_1.2^q+i-1)+1 = p_2.2^q+j.$$

Then (with $2p_1=p_2$) we have $2i-1=j$. (7.4)

On the other hand $j < 2^q+1 = 2.2^{q-1}+2-1 \leq 2i-1$ and hence $2i-1 \neq j$, contradicting (7.4). Therefore, neither B_{21} not B_{22} contains an M.

In a similar way it can be proven that if B_{12} contains an M, then neither B_{21} nor B_{22} contains an M. Then

$$\text{weight}(B) = \max\{\text{weight}(B_{ij}) : 1 \leq i,j \leq 2\} \leq c.$$

This ends the proof of the claim.

If each subblock of a has a weight less than or equal to c, then the weight of a itself is at most 2c. The submatrices A_{11} and $A_{2^p,2^p}$ can be chosen to be diagonal blocks of the permuted a. Thus after the permutation a contains 2c non-zero elements on its main diagonal. The values of c and p where chosen in such a way that

$$\frac{2c}{\text{number of rows of A}} < \varepsilon.$$

□

For the matrix A of the proof of theorem 7.2 we have only found an upperbound for N_1. Whether k.2c is an upperbound for N_k ($k \geq 2$) for this matrix A depends on the change in the recursive partition of a after the permutation by which a obtains N_1 non-zero elements on its main diagonal. The required permutation is not unique and, moreover, we have not indicated in which way a permutation changes the recursive partition.

In the weight heuristic we have separated clearly the execution of the permutations and the part in which the weight of a *block* will be computed. If during a permutation the recursive partition of a *block* changes, it may be worthwhile to perform once again the weighted matching algorithm for this *block*. If this leads to an increase in its weight and it was chosen as one of the *block*s in the "block"-transversal, then the weight of the *spmat* will increase. This, for example, holds for the *spmat* used in the proof of theorem 7.2.

The matrix A and its representation in the *spmat* a as used in the proof of theorem 7.2, is possible in the TORRIX-SPARSE system. However, it would be in bad taste to search for a transversal of an *spmat* with so many redundant partition lines. We can make each partition line to a useful one if each submatrix A_{ij} (except those that equal M) contains a 1×1 *mat*. Then the weight of a will be 2^p-2+2c and with p and c large enough we have $\frac{N_1}{c.2^p} = \frac{2^p-2+2c}{c.2^p} < \varepsilon$.

Conclusion. In this section we have proposed two heuristics to find a maximum transversal of a sparse matrix. Heuristic H is only a minor modification of the maximum matching algorithm of [51] ch. 5 sect. 5. The weight heuristic is block oriented. There are matrices A for which only an arbitrarily small portion of the main diagonal becomes zero free after the first part of the weight heuristic, even if A contains no redundant partition lines. For the

later part the results may depend severely on how the recursive partition will be changed as the result of row- and column-permutations.

VII.2 APPLICABILITY OF TORRIX-SPARSE FOR PERMUTING A MATRIX TO SQUARE BLOCK TRIANGULAR FORM.

Given a matrix $A=(\alpha_{ij})_{1\leq i,j\leq n}$ with $\alpha_{ii}\neq 0$ $(1\leq i\leq n)$ we will test the suitability of TORRIX-SPARSE for finding a permutation matrix R such that RAR^{-1} is of square block triangular form with as many main diagonal blocks as possible. For this purpose we will present in this section a block-oriented algorithm to find R.

For the explanation of this algorithm we have to review some definitions concerning directed graphs (see also V.3.2).
Let G=(V,E) be a directed graph. G is <u>strongly connected</u> if there is a path from v to w for all $v,w\in V$. Let $V'\subseteq V$. The <u>subgraph</u> G(V') of G is the graph (V',E(V')) with $E(V')=\{(v,w) : v,w\in V' \text{ and } (v,w)\in E\}$. G(V') is a <u>strongly connected component</u> of G if and only if G(V') is strongly connected and G(V") is not strongly connected for each $V''\subseteq V$ with $V'\subsetneq V''$. Let $P=\{V_1,\ldots,V_p\}$ be a partition of V. The <u>quotient graph</u> (cf. [35]) $G_{/P}=(W,E_P)$ consists of p vertices $w_1,\ldots,w_p$ and edges $E_P=\{(w_i,w_j) : i\neq j \text{ and there are } v,w\ (v\in V_i, w\in V_j) \text{ with } (v,w)\in E\}$.
As for (undirected) graphs (see III.1.1) one can define the <u>adjacency matrix</u> M(G) of a directed graph G=(V,E). With $V=\{v_1,\ldots,v_n\}$ M(G) is the $n\times n$ $\{0,1\}$-matrix $(m_{ij})_{1\leq i,j\leq n}$ with

$$\begin{aligned} m_{ii}&=1 \quad (1\leq i\leq n),\\ m_{ij}&=1 \quad \text{if } (v_i,v_j)\in E,\\ &=0 \quad \text{if } (v_i,v_j)\notin E \text{ and } i\neq j. \end{aligned}$$

In [68] an algorithm is given to determine the strongly connected components of an arbitrary directed graph G=(V,E) in time $O(|V|+|E|)$. Observe that this bound is the complexity of this problem: in order to explore every edge and every vertex of G at least $O(|V|+|E|)$ time is required.

Let $A=(\alpha_{ij})_{1\leq i,j\leq n}$ be a matrix with $\alpha_{ii}\neq 0$ $(1\leq i\leq n)$. If we do not take into account the numerical values of the elements of A but only its sparsity pattern, A can be considered the adjacency matrix of a directed graph G_A.

Moreover, if A is represented in an *spmat* a of which all main diagonal *block* are square, then each main diagonal *block* b of a represents the adjacency matrix of a subgraph G_b of G_A. The problem of finding a permutation matrix R such that RAR^{-1} is of square block triangular form with a maximum number of main diagonal blocks, is equivalent to the problem of determining the strongly connected components of G_A (cf. [69]). The main diagonal blocks of RAR^{-1} are the adjacency matrices of the strongly connected components of G_A. If we formulate the algorithm of [68] in terms of the representation a of G_A we do not obtain a block-oriented algorithm. For such an algorithm the following two lemmas can be used. Without proof we state:

Lemma 7.4. *Let* $G=(V,E)$ *be a directed graph. Let* $\{V_1,\ldots,V_n\}$ *be a partition of* V. *Let each* V_i $(1\leq i\leq n)$ *be partitioned into* $\{V_i^1,\ldots,V_i^{p_i}\}$ *such that* $G(V_i^j)$ *is a strongly connected component of* $G(V_i)$. *If* $W \subseteq V$ *and* G(W) *is a strongly connected component of* G *then for all* i $(1\leq i\leq n)$ *and* j $(1\leq j\leq p_i)$ *either* $V_i^j \subseteq W$ *or* $V_i^j \cap W=\emptyset$.

Lemma 7.5. *Let* G, V_i, V_i^j *be as in lemma 7.4. Let* $P=\{V_1^1,\ldots,V_1^{p_1},V_2^1,\ldots,V_n^1,\ldots,V_n^{p_n}\}$. *Let* $G_{/P}$ *have vertices* $(w_i^j)_{1\leq i\leq n,1\leq j\leq p_i}$ *corresponding to* $(V_i^j)_{1\leq i\leq n,1\leq j\leq p_i}$. *A strongly connected component of* G *consists of the vertices of* $V_{i_1}^{j_1},\ldots,V_{i_k}^{j_k}$ *if and only if* $w_{i_1}^{j_1},\ldots,w_{i_k}^{j_k}$ *form a strongly connected component of* $G_{/P}$.

These lemmas lead to the following recursive algorithm to permute the *spmat* a to square block triangular form:

algorithm 7.1. (Square block triangular form)
algorithm BTF = *(block b):*
co it is assumed that b has bounds [h:k,h:k] and that all main diagonal elements of a are non-zero; b is a *block* of a and each partitioned square *block* of a has a square partition. *co*

```
case b
in (ref mat): co all elements of b are non-zero and therefore b is already
                of square block triangular form. co
 , (ref diagmat):
    c apply some algorithm to permute rows and columns of a such that b is
      in square block triangular form
    c
 , (ref rectblock bb):
    (for i to 1 upb(subblocks of bb)
     do BTF((subblocks of bb)[i,i]); let the partition of
        (subblocks of bb)[i,i] correspond to the partition of
        (subblocks of bb)[i,i] in square block triangular form.
     od;
     c form the quotient graph G according to the partitions of the main
       diagonal direct subblocks of b;
       perform the algorithm of [68] to G;
       permute a such that the partition of b gives a square block
       triangular form to b
     c
    )
 , (ref bandblock):
    c similar to the case that b is partitioned rectangularly c
esac
```

In order to have an efficient implementation of this algorithm we introduce the following concept:

DEFINITION 7.5. Let *a* be an *spmat*. The bounds tree of *a* consists of a copy of the tree of *a* in which the leaves contain the concrete domains (and not the concrete-*array*s) of the *mat*s and *diagmat*s of the corresponding leaves in the tree of *a*.

Thus a bounds tree represents the sparsity pattern of an *spmat*.
Using bounds trees quotient graphs can be created efficiently. All changes in the recursive partition of a bounds tree can be performed as efficiently as on the corresponding *spmat* and even more efficiently if deletion of (redundant) partition lines are involved by which *mat*s and *diagmat*s are

replaced by their sum(s). Thus algorithm BTF can be implemented using bounds trees. The permutations obtained with each recursive call must be retained and when the application of BTF has finished, the overall permutation can be applied to the *spmat*.
Actually the weight heuristic (see VII.1) can also be implemented for a bounds tree.

Given a graph G and a partition P of its vertices the time used to determine the quotient graphs $G_{/P}$ in algorithm 7.1 is at most quadratic in the number of vertices of $G_{/P}$. A reduction in time used by BTF (compared with the time used by the algorithm of [68] can be obtained if the subsequent quotient graphs $G_{/P}$ as they occur in BTF have far less vertices than G. Unfortunately there are *spmat*s a of which the subsequent quotient graphs $G_{/P}$ do not differ from G. For example, if all main diagonal subblocks of a are lower triangular matrices.

Conclusion. In this section we have presented a block-oriented algorithm to permute a matrix to square block triangular form. The worst-case time analysis is somewhat disappointing, though for many matrices this proposed heuristic will not behave so badly as in the case of (permuted) lower triangular matrices.

VII.3 APPLICABILITY OF TORRIX-SPARSE FOR COMPUTING THE FILL IN LU DECOMPOSITION.

Let $A=(\alpha_{ij})_{1\leq i,j\leq n}$ be a non-singular matrix with $\alpha_{ii}\neq 0$ $(1\leq i\leq n)$ and assume A cannot be permuted to square block lower triangular form with more than one main diagonal block. Then A can be factored into

$$A=L.U \tag{7.5}$$

with $L=(l_{ij})_{1\leq i,j\leq n}$ and $U=(u_{ij})_{1\leq i,j\leq n}$ of lower and upper triangular form, respectively. Row- and column-permutations may be needed in order to obtain a numerically stable factorization or to decrease the number of non-zero elements in L and U. We assume that the permutations to decrease the number of non-zero elements in L and U have already been performed (cf. e.g. [35], [63] and [69]). Usually the data structure for A is overwritten with the elements of L and U while decomposing A (cf. [35] and [63]). Thus it may be appropriate first to make concrete those elements α_{ij} of A for which either

l_{ij} or u_{ij} will be non-zero (the fill). In this section we will show how advantage can be taken from the specific storage schemes of TORRIX-SPARSE in case the fill must be computed. Numerical cancellation cannot be taken into account except in case the numerical factorization is performed at once. We assume that all stored elements of A are non-zero.

We have the following equations for the elements of L and U (see also II.3.1):

$$l_{ij} = \alpha_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj} \quad (1 \leq j \leq i \leq n) \tag{7.6}$$

$$u_{ij} = l_{ii}^{-1} \cdot (\alpha_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj} \quad (1 \leq i < j \leq n). \tag{7.7}$$

Without loss of generality we assumed $u_{ii}=1$ $(1 \leq i \leq n)$. (7.6) and (7.7) lead to the following sparsity pattern of L and U:

$l_{ij} \neq 0$ if and only if $\alpha_{ij} \neq 0$ or $l_{ik} \neq 0$ and $u_{kj} \neq 0$ for some k with $1 \leq k \leq j-1$,

$u_{ij} \neq 0$ if and only if $\alpha_{ij} \neq 0$ or $l_{ik} \neq 0$ and $u_{kj} \neq 0$ for some k with $1 \leq k \leq i-1$.

and for the *spmat* a:

if the $(i,k)^{th}$ and $(k,j)^{th}$ element of a $(1 \leq k \leq \min(i,j))$ are concrete
then the $(i,j)^{th}$ element of a must be concrete } (7.8)

We say that in (7.8) the elements (i,k) and (k,j) of a are combined to element (i,j) of a. Obviously this combination does not need to be performed if element (i,j) of a is already concrete. Thus, as a first design, we have the following method to generate the fill:

```
for i to n
do for k to i-1
   do if a_ik is concrete
      then for j from k+1 to n
           do if a_kj is concrete
              then perform the combination of a_ik and a_kj
              fi
           od
      fi
   od
od
```

Now assume that a is partitioned into *blocks* $(b_{ij})_{1\leq i,j\leq p}$ and that the *blocks* b_{ii} $(1\leq i\leq p)$ are square. Then we can make the above algorithm suitable for *blocks*:

algorithm 7.2. (generating the fill in a square *block* b)

```
for i to p
do for k to i-1
   do if b_ik contains a concrete element
      then if b_ik contains a virtual element and
              b_kk contains a concrete element in its strictly upper trian-
              gular part
           then perform all combinations of elements of b_ik and elements
                of the strictly upper triangular part of b_kk
           fi;
           for j from k+1 to n
           do if b_kj contains a concrete element and b_ij contains a virtual
                element
              then perform all combinations of elements of b_ik with elements
                   of b_kj
              fi
           od
      fi
   od;
   perform all combinations of two concrete elements of b_ii;
   if b_ii has concrete elements in its strictly lower triangular part
   then for j from i+1 to n
        do if b_ij contains virtual and concrete elements
           then perform all combinations of elements of b_ij and elements
                of the strictly lower triangular part of b_ii
           fi
        od
   fi
od;
```

If the *spmat* a does not contain (during the elaboration of this algorithm) redundant partition lines, the question whether a *block* contains concrete and/or virtual elements can be answered in constant time: if a *block* is

partitioned, then it contains both concrete and virtual elements; if it is not partitioned the question can be decided from the bounds of the *mat* (*diagmat*) and the *block*.

Besides this, we can also take advantage from the fact that *scal*s are organized in *mat*s and *diagmat*s. An implementation of this algorithm will of course be organized recursively according to the possible levels of partitions of the *spmat* a. Deep in the recursion the algorithm will encounter non-partitioned *block*s. One can arrive at two *mat*s, a *mat* and a *diagmat* and at two *diagmat*s. The combination of all elements of two *mat*s causes that all elements of a rectangular subdomain of the *spmat* a must be (made) concrete. The bounds of this subdomain can be determined solely from the bounds of the two *mat*s. The combination of all elements of a *mat* m and one *vec* v of a *diagmat* causes that all elements of a rectangular subdomain must be (made) concrete. The bounds of this subdomain can be determined solely from the bounds of m and v the index of v in the *diagmat*.

A diagonal subdomain of an *spmat* a with index p and bounds [h:k] is the set

$$\{(i,i+p) : h \leq i \leq k\} \cap \{(i,j) : i \text{ and } j \text{ within the bounds of } a\}.$$

The combination of all elements of two *vec*s of two *diagmat*s causes that all elements of a diagonal subdomain must be (made) concrete. The index and the bounds of this subdomain can be determined solely from the bounds and the indices of the two *vec*s.

Thus, algorithm 7.2 can be implemented in such a way that it will only deal with bounds of *mat*s, *diagmat*s, *vec*s of *diagmat*s and *block*s. This means that its time complexity is not dependent on the number of non-zero elements in a but primarily on the number of *mat*s, *diagmat*s, *vec*s of *diagmat*s and the size of the recursive partition of a.

Conclusion. In this section we have given a formulation of symbolic LU decomposition (without additional row- and column-permutations) that is appropriate for the TORRIX-SPARSE data structure. This formulation results in a block-oriented algorithm which can be implemented in such a way that it deals only with rectangular and diagonal subdomains and not with the separate non-zero elements of the *spmat*.

REFERENCES

[01] AHO, A.V., J.E. HOPCROFT and J.D. ULLMAN, *The design and analysis of computer algorithms*, Addison Wesley, Reading, Mass., 1974.

[02] AMES, W.F., *Numerical methods for partial differential equations*, 2nd edition, Academic Press, New York, 1977.

[03] AXELSSON, O., *Solution of linear systems of equations: iterative methods*, in [05], pp. 1-51.

[04] BANK, R.E., *Marching algorithms and block Gaussian elimination*, in [15], pp. 293-308.

[05] BARKER, V.A. (ed.), *Sparse matrix techniques*, Lecture Notes in Mathematics 572, Springer Verlag, Berlin, 1977.

[06] BARTELS, R.H. and G.H. GOLUB, *The simplex method for linear programming using LU decomposition*, Comm. ACM 12 (1969), pp. 266-268.

[07] BENTLEY, J.L., *Multidimensional binary search trees used for associative searching*, Comm. ACM 18 (1975), pp. 509-517.

[08] BENTLEY, J.L. and D. WOOD, *An optimal worst case algorithm for reporting intersections of rectangles*, IEEE Transactions on Computers C-29 (1980), pp. 571-577.

[09] BJÖRCK, Å., *Methods for sparse linear least square problems*, in [15], pp. 177-200.

[10] BOOTH, K.S., *PQ-tree algorithms*, Ph.D thesis, Univ. of California, Berkeley, California, 1975.

[11] BOOTH, K.S. and G.S. LUEKER, *Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms*, J. Comput. System Sci. 13 (1976), pp. 335-379.

[12] BRØNLUND, O.E. and Th. LUNDE JOHNSEN, *QR-factorization of partitioned matrices*, Comp. Meth. Appl. Mech. Eng. 3 (1974), pp. 153-172.

[13] BUNCH, J.R., *Block methods for solving sparse linear systems*, in [15], pp. 39-58.

[14] BUNCH, J.R. and D.J. ROSE, *Partitioning, tearing and modification of sparse linear systems*, J. Math. Anal. Appl. 48 (1974), pp. 574-593.

[15] BUNCH, J.R. and D.J. ROSE (eds.), *Sparse matrix computations*, Academic Press, New York, 1976.

[16] BUZBEE, B.L., *A capacitance matrix technique*, in [15], pp. 365-374.

[17] BUZBEE, B.L. and F.W. DORR, *The direct solution of the biharmonic equation on rectangular regions and the Poisson equation on irregular regions*, SIAM J. Numer. Anal. 11 (1974), pp. 753-763.

[18] BUZBEE, B.L., G.H. GOLUB and C.W. NIELSON, *On direct methods for solving Poisson's equations*, SIAM J. Numer. Anal. 7 (1970), pp. 627-655.

[19] CHRISTOFIDES, N., *Graph theory, an algorithmic approach*, Academic Press, New York, 1975.

[20] COHN, P.M., *Algebra, vol. 1*, John Wiley & Sons, New York, 1974.

[21] DAHLQUIST, G. and Å. BJÖRCK, *Numerical methods*, Prentice Hall, Englewood Cliffs, New Jersey, 1974.

[22] DUFF, I.S., *On permutations to block triangular form*, J. Inst. Maths. Applic. 19 (1977), pp. 339-342.

[23] DUFF, I.S. and J.K. REID, *Some design features of a sparse matrix code*, ACM Transact. Math. Softw. 5 (1979), pp. 18-35.

[24] FINKEL, R.A. and J.L. BENTLEY, *Quad trees - a data structure for retrieval on composite keys*, Acta Inf. 4 (1974), pp. 1-9.

[25] FORREST, J.J.H. and J.A. TOMLIN, *Updating triangular factors of the basis to maintain sparsity in the product form simplex method*, Math. Prog. 2 (1972), pp. 263-278.

[26] FUCHS, G. von, J.R. ROY and E. SCHREM, *Hypermatrix solution of large sets of symmetric positive-definite linear equations*, Comp. Meth. Appl. Mech. Eng. 1 (1972), pp. 197-216.

[27] FULKERSON, D.R. and O.A. GROSS, *Incidence matrices and interval graphs*, Pacif. J. Math. 15 (1965), pp. 835-855.

[28] GAREY, M.R., R.L. GRAHAM, D.S. JOHNSON and D.E. KNUTH, *Complexity results for bandwidth minimization*, SIAM J. Appl. Math. 34 (1978), pp. 477-495.

[29] GAREY, M.R. and D.S. JOHNSON, *Computers and intractability, a guide to the theory of NP-completeness*, W.H. Freeman and Co., San Francisco, California, 1979.

[30] GENTLEMAN, W.M. and J.A. GEORGE, *Sparse matrix software*, in [15], pp. 243-262.

[31] GEORGE, J.A., *On block elimination for sparse linear systems*, SIAM J. Numer. Anal. 11 (1974), pp. 585-603.

[32] GEORGE, J.A., *Solution of linear systems of equations: direct methods for finite element problems*, in [05], pp. 52-101.

[33] GEORGE, J.A., *Numerical experiments using dissection methods to solve n by n grid problems*, SIAM J. Numer. Anal. 14 (1977), pp. 161-179.

[34] GEORGE, J.A. and J.W.H. LIU, *The design of a user interface for a sparse matrix package*, ACM Transact. Math. Softw. 5 (1979), pp. 139-162.

[35] GEORGE, J.A. and J.W.H. LIU, *Computer solution of large sparse positive definite systems*, Prentice Hall, Englewood Cliffs, New Jersey, 1981.

[36] GHOSH, S.P., *Data base organization for data management*, Academic Press, New York, 1977.

[37] GIBBS, N.E., 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 (1976), pp. 236-250.

[38] GILL, P.E. and W. MURRAY, *The orthogonal factorization of a large sparse matrix*, in [15], pp. 201-212.

[39] GOLUMBIC, M.C., *Algorithmic graph theory and perfect graphs*, Academic Press, New York, 1980.

[40] GUSTAVSON, F., *Finding the block lower triangular form of a sparse matrix*, in [15], pp. 275-289.

[41] HALMOS, P.R., *Finite dimensional vector spaces*, 2nd edition, D. van Nostrand Company, inc., New York, 1958.

[42] HELLERMAN, E. and D. RARICK, *Reinversion with the preassigned pivot procedure*, Math. Prog. 1 (1971), pp. 195-216.

[43] HELLERMAN, E. and D. RARICK, *The partitioned preassigned pivot procedure (P^4)*, in [65], pp. 67-76.

[44] HOPCROFT, J.E. and R.M. KARP, *An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs*, SIAM J. Comput. 2 (1973), pp. 225-231.

[45] HOROWITZ, E. and S. SAHNI, *Fundamentals of computer algorithms*, Pitman, San Francisco, California, 1978.

[46] JENNINGS, A., *A compact storage scheme for the solution of simultaneous equations*, Computer J. 9 (1966), pp. 281-285.

[47] KNUTH, D.E., *The art of computer programming, vol. 1, fundamental algorithms*, Addison-Wesley, Reading, Mass., 1973.

[48] KNUTH, D.E., *Optimum binary search trees*, Acta Inf. 1 (1971), pp. 14-25.

[49] KOU, L.T., *Polynomial complete consecutive information retrieval problems*, SIAM J. Comput. 6 (1977), pp. 67-75.

[50] LASDON, L.S., *Optimization theory for large systems*, The MacMillan Company, London, 1970.

[51] LAWLER, E.L., *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart & Winston, New York, 1976.

[52] LINDSEY, C.H. and S.G. VAN DER MEULEN, *Informal introduction to ALGOL 68, revised edition*, North Holland Publ. Comp., Amsterdam, 1977.

[53] LIPSKI, W. Jr., E. LODI, F. LUCCIO, C. MUGNAI and L. PAGLI, *On two-dimensional data organization II*, Fundamenta Informaticae 2 (1979), pp. 245-260.

[54] LODI, E., F. LUCCIO, C. MUGNAI and L. PAGLI, *On two-dimensional data organization I*, Fundamenta Informaticae 2 (1979), pp. 211-226.

[55] MAGNANTI, T.L., *Optimization for sparse matrices*, in [15], pp. 147-176.

[56] MEIJERINK, J.A. and H.A. VAN DER VORST, *An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix*, Math. of Comp. 31 (1977), pp. 148-162.

[57] NORRIE, D.H. and G. DE VRIES, *An introduction to finite element analysis*, Academic Press, New York, 1977.

[58] ODEN, J.T. and J.N. REDDY, *An introduction to the mathematical theory of finite elements*, J. Wiley & Sons, New York, 1976.

[59] PAGAN, F.G., *A practical guide to ALGOL 68*, J. Wiley & Sons, London, 1976.

[60] PEROLD, A.F. and G.B. DANTZIG, *A basis factorization method for block triangular linear programs*, in DUFF, I.S. and G.W. STEWART (eds.), *Sparse matrix proceedings*, SIAM, Phiadelphia, 1978, pp. 283-312.

[61] REID, J.K. (ed.), *Large sparse sets of linear equations*, Academic Press, New York, 1971.

[62] REID, J.K., *Sparse matrices*, C.S.S. 31, Computer Science and Systems Division, A.E.R.E., Harwell, U.K., 1976.

[63] REID, J.K., *Solutions of linear systems of equations: direct methods (general)*, in [05], pp. 102-129.

[64] RODRIGUE, G.H., N.K. MADSEN and J.I. KARUSH, *Odd-even reduction for banded linear equations*, J. ACM 26 (1979), pp. 72-81.

[65] ROSE, D.J. and R.A. WILLOUGHBY (eds.), *Sparse matrices and their applications*, Plenum Press, New York, 1972.

[66] SAUNDERS, M.A., *A fast stable implementation of the simplex method using Bartels-Golub updating*, in [15], pp. 213-222.

[67] STEWART, G.W., *A bibliographical tour on the large, sparse generalized eigenvalue problem*, in [15], pp. 113-130.

[68] TARJAN, R.E., *Depth first search and linear graph algorithms*, SIAM J. Comput. 1 (1972), pp. 146-160.

[69] TARJAN, R.E., *Graph theory and Gaussian elimination*, in [15], pp. 3-22.

[70] TEWARSON, R.P., *On the orthonormalization of sparse vectors*, Computing 3 (1968), pp. 268-279.

[71] TEWARSON, R.P., *Sorting and ordering sparse linear systems*, in [61], pp. 151-168.

[72] TUCKER, A., *A structure theorem for the consecutive ones property*, J. Combin. Th. 12(B) (1972), pp. 153-162.

[73] VALIANT, L.G., *The complexity of enumeration and reliability problems*, SIAM J. Comput. 8 (1979), pp. 410-421.

[74] VAN DER BLIJ, F., Personal communications, 1981.

[75] VAN DER MEULEN, S.G. and M. VELDHORST, *TORRIX, a programming system for operations on vectors and matrices over arbitrary fields and of variable size, vol. 1*, Mathematical Centre Tract 86, Mathematical Centre, Amsterdam, 1978.

[76] VAN DER MEULEN, S.G. and M. VELDHORST, *TORRIX, a programming system for operations on vectors and matrices over arbitrary fields and of variable size, vol. 2*, Mathematical Centre Tract 87, Mathematical Centre, Amsterdam, to be published.

[77] VAN WIJNGAARDEN, et al , *Revised report on the algorithmic language ALGOL 68*, Mathematical Centre Tract 50, Mathematical Centre, Amsterdam, 1976.

[78] VELDHORST, M., *Approximation of the consecutive ones matrix augmentation problem*, SIAM J. Comput. 14 (1985), pp. 709-728.

[79] VELDHORST, M., *The optimal representation of disjoint iso-oriented rectangles in two-dimensional trees*, to appear in J. Algor.

[80] WILKINSON, J.H., *The algebraic eigenvalue problem*, Clarendon Press, Oxford, 1965.

[81] YAO, F.F., *Efficient dynamic programming using quadrangle inequalities*, Proc. 12th Annual ACM Symposium on Theory of Computing, 1980, pp. 429-435.

[82] ZLATEV, Z., J. WASNIEWSKI and K. SCHAUMBURG, *Y12M, solution of large and sparse systems of linear algebraic equations*, Lecture Notes in Computer Science 121, Springer Verlag, Berlin, 1981.

INDEX

active domain 143
acyclic directed graph 115
  root of an ~ 115
  leaf of an ~ 115
adjacency matrix 51, 217
adjacent 50
administration 103
approximation algorithm 77, 79
array 15
  domain of an ~ 15
array1 15
array2 15
augmentation 74
backsolving 36
band matrix 24
bandwidth 48
basis (of a vector space) 10
basis (in a linear optimization problem) 44
BESTFIT 80
BESTFITDECR 83
bipartite graph 51
block 112
  bounds of a ~ 112
  disconnected ~s 144
  equal ~s 103, 115
  structurally equal ~s 116
  square ~ 123
block diagonal matrix 48
  block i of a ~ 82
  permuted ~ 82
block factorization 38
bounds tree 219
bump 45
chord 51
clique 51
colstrip 82
column (of a matrix) 10
column-partition 82
column-permutation 78
  optimum ~ 78
combination (of matrix elements) 221
complexification 23
concat 80
concrete-*array1* 19
concrete-*array2* 19
concrete domain 19, 180
  empty ~ 180
  disjoint ~s 180
  index set of a ~ 180
  lowerbound of a ~ 19
  upperbound of a ~ 19
concrete domains (sets of ~)
  equivalent ~ 181
  splittable ~ 181
  inclusion of ~ 181
concrete element 114
connected component 51
  strongly ~ 217
connected graph 51
connected order 96
consecutive arrangement 56
CONSECUTIVE ONES MATRIX AUGMENTATION problem 74
consecutive ones property for rows 50

COR-property 50
cycle 50, 115
  chordless ~ 51
  length of a ~ 50, 115
  simple ~ 51
decision problem 52
deterministic algorithm 54
diagarray 113
diagmat 25
diagonal (of a matrix) 25
dimension 10
directed graph 115
  acyclic ~ 115
  strongly connected ~ 217
direct method 36
edge 50, 115
elementary matrix 44
ENUMCHORDLESSPATHS 69
envelope 37
FDA 29
field 8
fill 37
finite difference approximation 29
garbage collection 106
Givens transformation 43
glb(S) 197
graph 50
  bipartite ~ 51
  connected ~ 51
  quotient ~ 217
group 7
guess 54
G_I, G_{II}, G_{III}, G_{IV}, G_V 58
hermitian form 23
heuristic H 212
Hom(V,W) 11
hypermatrix 101
incident 50
index 124
INDUCED CYCLE problem 59
INDUCED PATH problem 59
inner product 13
input 51
  length of an ~ 52
iterative method 36
k-d trees 183
leaf change 149
linear combination 9
linear independent vectors 9
linear mapping 11
  sum of ~s 11
  product of ~s 11
LONGEST CHORDLESS CYCLE problem 59
LONGEST CHORDLESS PATH problem 52, 59
LONGEST PATH problem 55
lub(S) 197
LU decomposition 36
  symbolic ~ 207
Markowitz criterion 37
mat 22
matbnds 124
matching 210
  maximum ~ 210
matrix 11
  clean ~ 79
  insertion ~ 85
  s-mix1 insertion ~ 98
  s-mix2 insertion ~ 98
MAXIMUM G SUBGRAPH problems 60

MAXIMUM FORBIDDEN SUBGRAPH problem 60
maximum weighted matching problem 213
mesh 29
meshpoint 29
minheight(S) 196
minimal horizontal split line 197
minimal vertical split line 197
minimum degree 37
minimum split 187
minsplit(S) 187
MINSPLIT(S) 193
mmh(S) 196
M_I, M_{II}, M_{III}, M_{IV}, M_V 58
neighbor 50
nested dissection 32
node see vertex
nondeterministic algorithm 54
nondeterministic polynomial time algorithm 54
NP-complete 54
ones(A) 78
on-line column insertion algorithm 85
on-line column s-mix1 insertion algorithm 98
on-line column s-mix2 insertion algorithm 98
operator 11, 17
optimization problem 52
  linear ~ 35
optstore(A) 78
original (matrix, vector or data structure) 119
orthogonalization 42
out of date 146
pair 124
partition (of a matrix or block) 47, 112
  size of a ~ 112
  square ~ 47, 123
partition line 123
  direct horizontal ~ 123
  direct vertical ~ 123
  endpoints of a ~ 123
  horizontal ~ 123
  redundant ~ 130
  vertical ~ 123
path 50, 115
  augmenting ~ 210
  chordless ~ 51
  length of a ~ 50, 115
  p-augmenting ~ 211
  simple ~ 51
P-enumerable 67
polynomial space 53
polynomial time 53
polynomial transformable 54
problem 51
  instance of a ~ 51
profile 37
quad tree 183
query 118
RAM 52
rectangle 180
recursive partition 112, 179
  depth of a ~ 112
  optimal ~ 179
  size of a ~ 112
  symmetric ~ 123

ref-change 147
ring 8
  ordered ~ 12
row (of a matrix) 11
rowmat 24, 49
row-partition 82
rowstrip 82
scal 14
scalar 9
scalar product 13
scalel 120
search path 115
separator 96
shift 133
side effect 105, 106
simplex method 44
slalias 175
slicing 118
  column ~ 119, 120
  diagonal ~ 119, 120
  element ~ 119, 120
  ~ mechanism 105, 118
  revised lowerbound ~ 119, 120
  row ~ 119, 120
  submatrix ~ 119, 120
  subvector ~ 119, 120
  vector ~ 119, 120
space complexity 53
space lowerbound 53
space upperbound 52, 53
spanning subspace 10
sparmat 117
sparse matrix 25
sparse matrix storage tree 114
SPARSEI 122, 124
SPARSEII 122, 132
SPARSEIII 122, 137
SPARSEIV 122, 145
SPARSEV (=TORRIX-SPARSE) 122, 159
sparsity pattern 27
  symmetric ~ 47
spikes 45
spmat 122
spsym 122
spvec 120
state 54
storage tree 182
  binary ~ 196
store(A) 78
subblock 113
  direct ~ 112
subdomain
  diagonal ~ 223
  rectangular ~ 124
  transpose of a ~ 139
subgraph 51, 217
submatrix 55
  permuted ~ 55
subspace 10
subtree change 149
sympat 122
system manager 102
t 16
time complexity 53
time lowerbound 53
time upperbound 52, 53
TORRIX 7
  design objectives of ~ 14
TORRIX-BASIS 18
TORRIX-COMPLEX 23
TORRIX-SPARSE 101
  design objectives of ~ 102

total-array1 16
total-array2 16
transformation 11
transversal 209
TREETEST 190
triangular matrix 24
trimmer 124
up to date 146
vec 22
vector 9
vector space 9
vertex 50, 115
  degree of a ~ 50
  descendant of a ~ 115
  indegree of a ~ 115
  outdegree of a ~ 115
  son of a ~ 115
virtual elements 114
weight heuristic 212

SUMMARY

In this book a study has been made of the storage optimality of data structures for sparse matrices. We have considered the *rowmat* data structure in which a sparse matrix is stored as a sequence of rows and each row is stored in such a way that all zero elements at both ends will not be stored. We have investigated algorithmic problems that arise if not the sparse matrix A itself but some column-permutation B of A that will lead to "fewer zeros", will be stored in a *rowmat* data structure. The NP-completeness has been proven of the problem of finding the largest permuted submatrix A' of A for which no column-permutation exists that can be stored in a *rowmat* data structure without storing any zero element, while for each proper submatrix of A' such a column-permutation does exist. In [10] the NP-completeness was proven of the problem to find an optimum column-permutation of A, i.e., a column-permutation B of A that can be stored in a *rowmat* data structure with no more elements than any other column-permutation of A. In chapter IV we have proven that even near optimum column-permutations of A cannot be found for each A by such simple algorithms like the on-line column insertion algorithms. In fact a much wider class of algorithms was characterized of which no algorithm can find near optimum column-permutations for all sparse matrices. Therefore, we have provided an indication that the *rowmat* data structure is not well suited for arbitrary sparse matrices if full storage optimality is the ultimate goal. In chapters V, VI and VII another approach was followed. Sparse matrices are partitioned recursively and stored in a tree data structure according to their recursive partitions. We have designed a sparse matrix package (TORRIX-SPARSE) for general use based on this tree data structure. A reduction in storage could be obtained in case of equal blocks. Moreover, a slicing mechanism as in ALGOL 68 was incorporated. With such a slicing mechanism operations induce specific side effects if applied to a matrix or one if its slices. In chapter VI we have designed algorithms for the problem of finding for an arbitrary sparse

matrix A recursive partitions that lead to a binary tree for A with a minimum number of vertices or minimum height, respectively. It was shown that sparse matrices exists for which no recursive partitions are possible that lead to a binary tree with a minimum number of vertices and minimum height simultaneously. In chapter VII we have tested the suitability of TORRIX-SPARSE in case it is used to solve a linear system of equations by means of LU decomposition. For the following three problems we designed algorithms that take advantage of the specific data structures of TORRIX-SPARSE: permuting a sparse matrix to zero-free diagonal form, permuting a sparse matrix to block lower triangular form and performing a symbolic LU decomposition.

MATHEMATICAL CENTRE TRACTS

1 T. van der Walt. *Fixed and almost fixed points.* 1963.

2 A.R. Bloemena. *Sampling from a graph.* 1964.

3 G. de Leve. *Generalized Markovian decision processes, part 1: model and method.* 1964.

4 G. de Leve. *Generalized Markovian decision processes, part II: probabilistic background.* 1964.

5 G. de Leve, H.C. Tijms, P.J. Weeda. *Generalized Markovian decision processes, applications.* 1970.

6 M.A. Maurice. *Compact ordered spaces.* 1964.

7 W.R. van Zwet. *Convex transformations of random variables.* 1964.

8 J.A. Zonneveld. *Automatic numerical integration.* 1964.

9 P.C. Baayen. *Universal morphisms.* 1964.

10 E.M. de Jager. *Applications of distributions in mathematical physics.* 1964.

11 A.B. Paalman-de Miranda. *Topological semigroups.* 1964.

12 J.A.Th.M. van Berckel, H. Brandt Corstius, R.J. Mokken, A. van Wijngaarden. *Formal properties of newspaper Dutch.* 1965.

13 H.A. Lauwerier. *Asymptotic expansions.* 1966, out of print; replaced by MCT 54.

14 H.A. Lauwerier. *Calculus of variations in mathematical physics.* 1966.

15 R. Doornbos. *Slippage tests.* 1966.

16 J.W. de Bakker. *Formal definition of programming languages with an application to the definition of ALGOL 60.* 1967.

17 R.P. van de Riet. *Formula manipulation in ALGOL 60, part 1.* 1968.

18 R.P. van de Riet. *Formula manipulation in ALGOL 60, part 2.* 1968.

19 J. van der Slot. *Some properties related to compactness.* 1968.

20 P.J. van der Houwen. *Finite difference methods for solving partial differential equations.* 1968.

21 E. Wattel. *The compactness operator in set theory and topology.* 1968.

22 T.J. Dekker. *ALGOL 60 procedures in numerical algebra, part 1.* 1968.

23 T.J. Dekker, W. Hoffmann. *ALGOL 60 procedures in numerical algebra, part 2.* 1968.

24 J.W. de Bakker. *Recursive procedures.* 1971.

25 E.R. Paërl. *Representations of the Lorentz group and projective geometry.* 1969.

26 European Meeting 1968. *Selected statistical papers, part I.* 1968.

27 European Meeting 1968. *Selected statistical papers, part II.* 1968.

28 J. Oosterhoff. *Combination of one-sided statistical tests.* 1969.

29 J. Verhoeff. *Error detecting decimal codes.* 1969.

30 H. Brandt Corstius. *Exercises in computational linguistics.* 1970.

31 W. Molenaar. *Approximations to the Poisson, binomial and hypergeometric distribution functions.* 1970.

32 L. de Haan. *On regular variation and its application to the weak convergence of sample extremes.* 1970.

33 F.W. Steutel. *Preservation of infinite divisibility under mixing and related topics.* 1970.

34 I. Juhász, A. Verbeek, N.S. Kroonenberg. *Cardinal functions in topology.* 1971.

35 M.H. van Emden. *An analysis of complexity.* 1971.

36 J. Grasman. *On the birth of boundary layers.* 1971.

37 J.W. de Bakker, G.A. Blaauw, A.J.W. Duijvestijn, E.W. Dijkstra, P.J. van der Houwen, G.A.M. Kamsteeg-Kemper, F.E.J. Kruseman Aretz, W.L. van der Poel, J.P. Schaap-Kruseman, M.V. Wilkes, G. Zoutendijk. *MC-25 Informatica Symposium.* 1971.

38 W.A. Verloren van Themaat. *Automatic analysis of Dutch compound words.* 1972.

39 H. Bavinck. *Jacobi series and approximation.* 1972.

40 H.C. Tijms. *Analysis of (s,S) inventory models.* 1972.

41 A. Verbeek. *Superextensions of topological spaces.* 1972.

42 W. Vervaat. *Success epochs in Bernoulli trials (with applications in number theory).* 1972.

43 F.H. Ruymgaart. *Asymptotic theory of rank tests for independence.* 1973.

44 H. Bart. *Meromorphic operator valued functions.* 1973.

45 A.A. Balkema. *Monotone transformations and limit laws.* 1973.

46 R.P. van de Riet. *ABC ALGOL, a portable language for formula manipulation systems, part 1: the language.* 1973.

47 R.P. van de Riet. *ABC ALGOL, a portable language for formula manipulation systems, part 2: the compiler.* 1973.

48 F.E.J. Kruseman Aretz, P.J.W. ten Hagen, H.L. Oudshoorn. *An ALGOL 60 compiler in ALGOL 60, text of the MC-compiler for the EL-X8.* 1973.

49 H. Kok. *Connected orderable spaces.* 1974.

50 A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens, R.G. Fisker (eds.). *Revised report on the algorithmic language ALGOL 68.* 1976.

51 A. Hordijk. *Dynamic programming and Markov potential theory.* 1974.

52 P.C. Baayen (ed.). *Topological structures.* 1974.

53 M.J. Faber. *Metrizability in generalized ordered spaces.* 1974.

54 H.A. Lauwerier. *Asymptotic analysis, part 1.* 1974.

55 M. Hall, Jr., J.H. van Lint (eds.). *Combinatorics, part 1: theory of designs, finite geometry and coding theory.* 1974.

56 M. Hall, Jr., J.H. van Lint (eds.). *Combinatorics, part 2: graph theory, foundations, partitions and combinatorial geometry.* 1974.

57 M. Hall, Jr., J.H. van Lint (eds.). *Combinatorics, part 3: combinatorial group theory.* 1974.

58 W. Albers. *Asymptotic expansions and the deficiency concept in statistics.* 1975.

59 J.L. Mijnheer. *Sample path properties of stable processes.* 1975.

60 F. Göbel. *Queueing models involving buffers.* 1975.

63 J.W. de Bakker (ed.). *Foundations of computer science.* 1975.

64 W.J. de Schipper. *Symmetric closed categories.* 1975.

65 J. de Vries. *Topological transformation groups, 1: a categorical approach.* 1975.

66 H.G.J. Pijls. *Logically convex algebras in spectral theory and eigenfunction expansions.* 1976.

68 P.P.N. de Groen. *Singularly perturbed differential operators of second order.* 1976.

69 J.K. Lenstra. *Sequencing by enumerative methods.* 1977.

70 W.P. de Roever, Jr. *Recursive program schemes: semantics and proof theory.* 1976.

71 J.A.E.E. van Nunen. *Contracting Markov decision processes.* 1976.

72 J.K.M. Jansen. *Simple periodic and non-periodic Lamé functions and their applications in the theory of conical waveguides.* 1977.

73 D.M.R. Leivant. *Absoluteness of intuitionistic logic.* 1979.

74 H.J.J. te Riele. *A theoretical and computational study of generalized aliquot sequences.* 1976.

75 A.E. Brouwer. *Treelike spaces and related connected topological spaces.* 1977.

76 M. Rem. *Associons and the closure statement.* 1976.

77 W.C.M. Kallenberg. *Asymptotic optimality of likelihood ratio tests in exponential families.* 1978.

78 E. de Jonge, A.C.M. van Rooij. *Introduction to Riesz spaces.* 1977.

79 M.C.A. van Zuijlen. *Emperical distributions and rank statistics.* 1977.

80 P.W. Hemker. *A numerical study of stiff two-point boundary problems.* 1977.

81 K.R. Apt, J.W. de Bakker (eds.). *Foundations of computer science II, part 1.* 1976.

82 K.R. Apt, J.W. de Bakker (eds.). *Foundations of computer science II, part 2.* 1976.

83 L.S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the AUTOMATH system.* 1979.

84 H.L.L. Busard. *The translation of the elements of Euclid from the Arabic into Latin by Hermann of Carinthia (?), books vii-xii.* 1977.

85 J. van Mill. *Supercompactness and Wallman spaces.* 1977.

86 S.G. van der Meulen, M. Veldhorst. *Torrix I, a programming system for operations on vectors and matrices over arbitrary fields and of variable size.* 1978.

88 A. Schrijver. *Matroids and linking systems.* 1977.

89 J.W. de Roever. *Complex Fourier transformation and analytic functionals with unbounded carriers.* 1978.

90 L.P.J. Groenewegen. *Characterization of optimal strategies in dynamic games.* 1981.

91 J.M. Geysel. *Transcendence in fields of positive characteristic.* 1979.

92 P.J. Weeda. *Finite generalized Markov programming.* 1979.

93 H.C. Tijms, J. Wessels (eds.). *Markov decision theory.* 1977.

94 A. Bijlsma. *Simultaneous approximations in transcendental number theory.* 1978.

95 K.M. van Hee. *Bayesian control of Markov chains.* 1978.

96 P.M.B. Vitányi. *Lindenmayer systems: structure, languages, and growth functions.* 1980.

97 A. Federgruen. *Markovian control problems; functional equations and algorithms.* 1984.

98 R. Geel. *Singular perturbations of hyperbolic type.* 1978.

99 J.K. Lenstra, A.H.G. Rinnooy Kan, P. van Emde Boas (eds.). *Interfaces between computer science and operations research.* 1978.

100 P.C. Baayen, D. van Dulst, J. Oosterhoff (eds.). *Proceedings bicentennial congress of the Wiskundig Genootschap, part 1.* 1979.

101 P.C. Baayen, D. van Dulst, J. Oosterhoff (eds.). *Proceedings bicentennial congress of the Wiskundig Genootschap, part 2.* 1979.

102 D. van Dulst. *Reflexive and superreflexive Banach spaces.* 1978.

103 K. van Harn. *Classifying infinitely divisible distributions by functional equations.* 1978.

104 J.M. van Wouwe. *Go-spaces and generalizations of metrizability.* 1979.

105 R. Helmers. *Edgeworth expansions for linear combinations of order statistics.* 1982.

106 A. Schrijver (ed.). *Packing and covering in combinatorics.* 1979.

107 C. den Heijer. *The numerical solution of nonlinear operator equations by imbedding methods.* 1979.

108 J.W. de Bakker, J. van Leeuwen (eds.). *Foundations of computer science III, part 1.* 1979.

109 J.W. de Bakker, J. van Leeuwen (eds.). *Foundations of computer science III, part 2.* 1979.

110 J.C. van Vliet. *ALGOL 68 transput, part I: historical review and discussion of the implementation model.* 1979.

111 J.C. van Vliet. *ALGOL 68 transput, part II: an implementation model.* 1979.

112 H.C.P. Berbee. *Random walks with stationary increments and renewal theory.* 1979.

113 T.A.B. Snijders. *Asymptotic optimality theory for testing problems with restricted alternatives.* 1979.

114 A.J.E.M. Janssen. *Application of the Wigner distribution to harmonic analysis of generalized stochastic processes.* 1979.

115 P.C. Baayen, J. van Mill (eds.). *Topological structures II, part 1.* 1979.

116 P.C. Baayen, J. van Mill (eds.). *Topological structures II, part 2.* 1979.

117 P.J.M. Kallenberg. *Branching processes with continuous state space.* 1979.

118 P. Groeneboom. *Large deviations and asymptotic efficiencies.* 1980.

119 F.J. Peters. *Sparse matrices and substructures, with a novel implementation of finite element algorithms.* 1980.

120 W.P.M. de Ruyter. *On the asymptotic analysis of large-scale ocean circulation.* 1980.

121 W.H. Haemers. *Eigenvalue techniques in design and graph theory.* 1980.

122 J.C.P. Bus. *Numerical solution of systems of nonlinear equations.* 1980.

123 I. Yuhász. *Cardinal functions in topology - ten years later.* 1980.

124 R.D. Gill. *Censoring and stochastic integrals.* 1980.

125 R. Eising. *2-D systems, an algebraic approach.* 1980.

126 G. van der Hoek. *Reduction methods in nonlinear programming.* 1980.

127 J.W. Klop. *Combinatory reduction systems.* 1980.

128 A.J.J. Talman. *Variable dimension fixed point algorithms and triangulations.* 1980.

129 G. van der Laan. *Simplicial fixed point algorithms.* 1980.

130 P.J.W. ten Hagen, T. Hagen, P. Klint, H. Noot, H.J. Sint, A.H. Veen. *ILP: intermediate language for pictures.* 1980.

131 R.J.R. Back. *Correctness preserving program refinements: proof theory and applications.* 1980.

132 H.M. Mulder. *The interval function of a graph.* 1980.

133 C.A.J. Klaassen. *Statistical performance of location estimators.* 1981.

134 J.C. van Vliet, H. Wupper (eds.). *Proceedings international conference on ALGOL 68.* 1981.

135 J.A.G. Groenendijk, T.M.V. Janssen, M.J.B. Stokhof (eds.). *Formal methods in the study of language, part I.* 1981.

136 J.A.G. Groenendijk, T.M.V. Janssen, M.J.B. Stokhof (eds.). *Formal methods in the study of language, part II.* 1981.

137 J. Telgen. *Redundancy and linear programs.* 1981.

138 H.A. Lauwerier. *Mathematical models of epidemics.* 1981.

139 J. van der Wal. *Stochastic dynamic programming, successive approximations and nearly optimal strategies for Markov decision processes and Markov games.* 1981.

140 J.H. van Geldrop. *A mathematical theory of pure exchange economies without the no-critical-point hypothesis.* 1981.

141 G.E. Welters. *Abel-Jacobi isogenies for certain types of Fano threefolds.* 1981.

142 H.R. Bennett, D.J. Lutzer (eds.). *Topology and order structures, part 1.* 1981.

143 J.M. Schumacher. *Dynamic feedback in finite- and infinite-dimensional linear systems.* 1981.

144 P. Eijgenraam. *The solution of initial value problems using interval arithmetic; formulation and analysis of an algorithm.* 1981.

145 A.J. Brentjes. *Multi-dimensional continued fraction algorithms.* 1981.

146 C.V.M. van der Mee. *Semigroup and factorization methods in transport theory.* 1981.

147 H.H. Tigelaar. *Identification and informative sample size.* 1982.

148 L.C.M. Kallenberg. *Linear programming and finite Markovian control problems.* 1983.

149 C.B. Huijsmans, M.A. Kaashoek, W.A.J. Luxemburg, W.K. Vietsch (eds.). *From A to Z, proceedings of a symposium in honour of A.C. Zaanen.* 1982.

150 M. Veldhorst. *An analysis of sparse matrix storage schemes.* 1982.

151 R.J.M.M. Does. *Higher order asymptotics for simple linear rank statistics.* 1982.

152 G.F. van der Hoeven. *Projections of lawless sequences.* 1982.

153 J.P.C. Blanc. *Application of the theory of boundary value problems in the analysis of a queueing model with paired services.* 1982.

154 H.W. Lenstra, Jr., R. Tijdeman (eds.). *Computational methods in number theory, part I.* 1982.

155 H.W. Lenstra, Jr., R. Tijdeman (eds.). *Computational methods in number theory, part II.* 1982.

156 P.M.G. Apers. *Query processing and data allocation in distributed database systems.* 1983.

157 H.A.W.M. Kneppers. *The covariant classification of two-dimensional smooth commutative formal groups over an algebraically closed field of positive characteristic.* 1983.

158 J.W. de Bakker, J. van Leeuwen (eds.). *Foundations of computer science IV, distributed systems, part 1.* 1983.

159 J.W. de Bakker, J. van Leeuwen (eds.). *Foundations of computer science IV, distributed systems, part 2.* 1983.

160 A. Rezus. *Abstract AUTOMATH.* 1983.

161 G.F. Helminck. *Eisenstein series on the metaplectic group, an algebraic approach.* 1983.

162 J.J. Dik. *Tests for preference.* 1983.

163 H. Schippers. *Multiple grid methods for equations of the second kind with applications in fluid mechanics.* 1983.

164 F.A. van der Duyn Schouten. *Markov decision processes with continuous time parameter.* 1983.

165 P.C.T. van der Hoeven. *On point processes.* 1983.

166 H.B.M. Jonkers. *Abstraction, specification and implementation techniques, with an application to garbage collection.* 1983.

167 W.H.M. Zijm. *Nonnegative matrices in dynamic programming.* 1983.

168 J.H. Evertse. *Upper bounds for the numbers of solutions of diophantine equations.* 1983.

169 H.R. Bennett, D.J. Lutzer (eds.). *Topology and order structures, part 2.* 1983.

CWI TRACTS

1 D.H.J. Epema. *Surfaces with canonical hyperplane sections.* 1984.

2 J.J. Dijkstra. *Fake topological Hilbert spaces and characterizations of dimension in terms of negligibility.* 1984.

3 A.J. van der Schaft. *System theoretic descriptions of physical systems.* 1984.

4 J. Koene. *Minimal cost flow in processing networks, a primal approach.* 1984.

5 B. Hoogenboom. *Intertwining functions on compact Lie groups.* 1984.

6 A.P.W. Böhm. *Dataflow computation.* 1984.

7 A. Blokhuis. *Few-distance sets.* 1984.

8 M.H. van Hoorn. *Algorithms and approximations for queueing systems.* 1984.

9 C.P.J. Koymans. *Models of the lambda calculus.* 1984.

10 C.G. van der Laan, N.M. Temme. *Calculation of special functions: the gamma function, the exponential integrals and error-like functions.* 1984.

11 N.M. van Dijk. *Controlled Markov processes; time-discretization.* 1984.

12 W.H. Hundsdorfer. *The numerical solution of nonlinear stiff initial value problems: an analysis of one step methods.* 1985.

13 D. Grune. *On the design of ALEPH.* 1985.

14 J.G.F. Thiemann. *Analytic spaces and dynamic programming: a measure theoretic approach.* 1985.

15 F.J. van der Linden. *Euclidean rings with two infinite primes.* 1985.

16 R.J.P. Groothuizen. *Mixed elliptic-hyperbolic partial differential operators: a case-study in Fourier integral operators.* 1985.

17 H.M.M. ten Eikelder. *Symmetries for dynamical and Hamiltonian systems.* 1985.

18 A.D.M. Kester. *Some large deviation results in statistics.* 1985.

19 T.M.V. Janssen. *Foundations and applications of Montague grammer, part 1: Philosophy, framework, computer science.* 1986.

20 B.F. Schriever. *Order dependence.* 1986.

21 D.P. van der Vecht. *Inequalities for stopped Brownian motion.* 1986.