Chapter 3

SPARSE MATRICES

As described in the previous chapter, standard discretizations of Partial Differential Equations typically lead to large and sparse *matrices. A sparse matrix is defined, somewhat vaguely, as a matrix which has very few nonzero elements. But, in fact, a matrix can be termed sparse whenever special techniques can be utilized to take advantage of the large number of zero elements and their locations. These sparse matrix techniques begin with the idea that the zero elements need not be stored. One of the key issues is to define data structures for these matrices that are well suited for efficient implementation of standard solution methods, whether direct or iterative. This chapter gives an overview of sparse matrices, their properties, their representations, and the data structures used to store them.*

3.1 Introduction

The natural idea to take advantage of the zeros of a matrix and their location was initiated by engineers in various disciplines. In the simplest case involving banded matrices, special techniques are straightforward to develop. Electrical engineers dealing with electrical networks in the 1960s were the first to exploit sparsity to solve general sparse linear systems for matrices with irregular structure. The main issue, and the first addressed by sparse matrix technology, was to devise direct solution methods for linear systems. These had to be economical, both in terms of storage and computational effort. Sparse direct solvers can handle very large problems that cannot be tackled by the usual "dense" solvers.

Essentially, there are two broad types of sparse matrices: *structured* and *unstructured*. A structured matrix is one whose nonzero entries form a regular pattern, often along a small number of diagonals. Alternatively, the nonzero elements may lie in blocks (dense submatrices) of the same size, which form a regular pattern, typically along a small number of (block) diagonals. A matrix with irregularly located entries is said to be irregularly structured. The best example of a regularly structured matrix is a matrix that consists of only a few diagonals. Finite difference matrices on rectangular grids, such as the ones seen in the previous chapter, are typical examples of matrices with regular structure. Most finite element or finite volume techniques

applied to complex geometries lead to irregularly structured matrices. Figure 3.2 shows a small irregularly structured sparse matrix associated with the finite element grid problem shown in Figure 3.1.

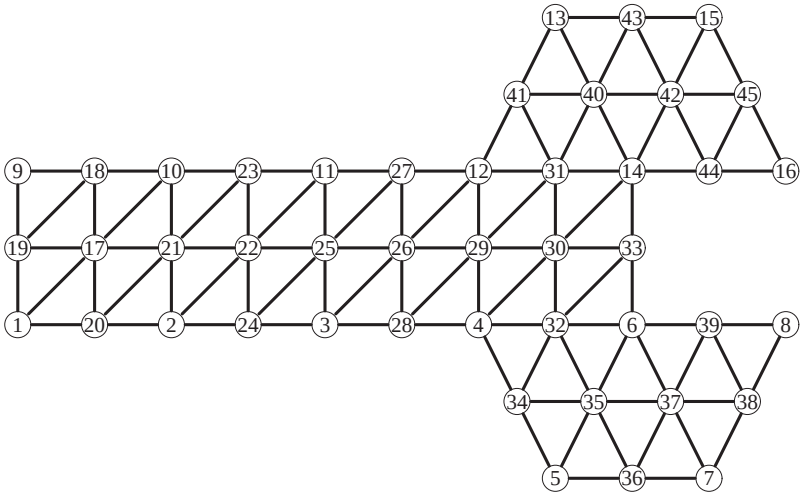


Figure 3.1: A small finite element grid model.

The distinction between the two types of matrices may not noticeably affect direct solution techniques, and it has not received much attention in the past. However, this distinction can be important for iterative solution methods. In these methods, one of the essential operations is matrix-by-vector products. The performance of these operations can differ significantly on high performance computers, depending on whether they are regularly structured or not. For example, on vector computers, storing the matrix by diagonals is ideal, but the more general schemes may suffer because they require indirect addressing.

The next section discusses graph representations of sparse matrices. This is followed by an overview of some of the storage schemes used for sparse matrices and an explanation of how some of the simplest operations with sparse matrices can be performed. Then sparse linear system solution methods will be covered. Finally, Section 3.7 discusses test matrices.

3.2 Graph Representations

Graph theory is an ideal tool for representing the structure of sparse matrices and for this reason it plays a major role in sparse matrix techniques. For example, graph theory is the key ingredient used in unraveling parallelism in sparse Gaussian elimination or in preconditioning techniques. In the following section, graphs are discussed in general terms and then their applications to finite element or finite difference matrices are discussed.

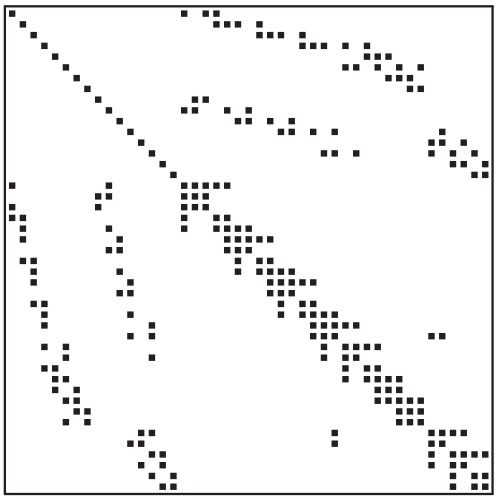


Figure 3.2: Sparse matrix associated with the finite element grid of Figure 3.1.

3.2.1 Graphs and Adjacency Graphs

Remember that a graph is defined by two sets, a set of vertices

$$V = \{v_1, v_2, \ldots, v_n\},$$

and a set of edges E which consists of pairs (v_i, v_j), where v_i, v_j are elements of V, i.e.,

$$E \subseteq V \times V.$$

This graph $G = (V, E)$ is often represented by a set of points in the plane linked by a directed line between the points that are connected by an edge. A graph is a way of representing a binary relation between objects of a set V. For example, V can represent the major cities of the world. A line is drawn between any two cities that are linked by a nonstop airline connection. Such a graph will represent the relation "there is a nonstop flight from city (A) to city (B)." In this particular example, the binary relation is likely to be symmetric, i.e., when there is a nonstop flight from (A) to (B) there is also a nonstop flight from (B) to (A). In such situations, the graph is said to be undirected, as opposed to a general graph which is directed.

Going back to sparse matrices, the *adjacency graph* of a sparse matrix is a graph $G = (V, E)$, whose n vertices in V represent the n unknowns. Its edges represent the binary relations established by the equations in the following manner: There is an edge from node i to node j when $a_{ij} \neq 0$. This edge will therefore represent the binary relation *equation i involves unknown j*. Note that the adjacency graph is an undirected graph when the matrix pattern is symmetric, i.e., when $a_{ij} \neq 0$ iff $a_{ji} \neq 0$ for all $1 \leq i, j \leq n$).

When a matrix has a symmetric nonzero pattern, i.e., when a_{ij} and a_{ji} are always nonzero at the same time, then the graph is *undirected*. Thus, for undirected

graphs, every edge points in both directions. As a result, undirected graphs can be represented with nonoriented edges.

As an example of the use of graph models, parallelism in Gaussian elimination can be extracted by finding unknowns that are independent at a given stage of the elimination. These are unknowns which do not depend on each other according to the above binary relation. The rows corresponding to such unknowns can then be used as pivots simultaneously. Thus, in one extreme, when the matrix is diagonal, then all unknowns are independent. Conversely, when a matrix is dense, each unknown will depend on all other unknowns. Sparse matrices lie somewhere between these two extremes.

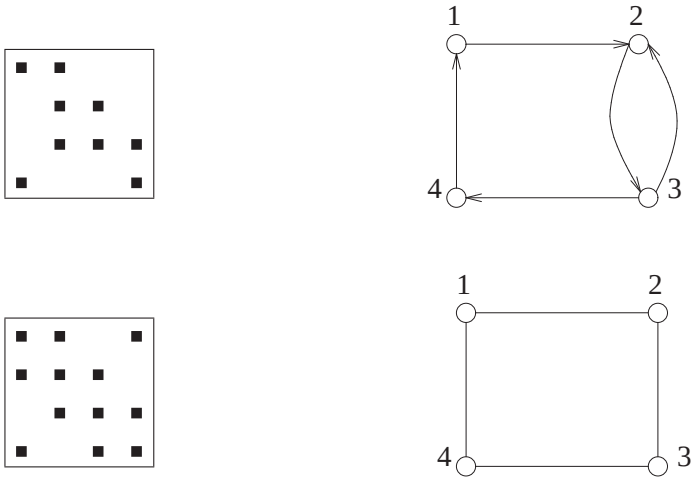


Figure 3.3: Graphs of two 4×4 sparse matrices.

There are a few interesting simple properties of adjacency graphs. The graph of A^2 can be interpreted as an n-vertex graph whose edges are the pairs (i, j) for which there exists at least one path of length exactly two from node i to node j in the original graph of A. Similarly, the graph of A^k consists of edges which represent the binary relation "there is at least one path of length k from node i to node j." For details, see Exercise 4.

3.2.2 Graphs of PDE Matrices

For Partial Differential Equations involving only one physical unknown per mesh point, the adjacency graph of the matrix arising from the discretization is often the graph represented by the mesh itself. However, it is common to have several unknowns per mesh point. For example, the equations modeling fluid flow may involve the two velocity components of the fluid (in two dimensions) as well as energy and momentum at each mesh point.

In such situations, there are two choices when labeling the unknowns. They can be labeled contiguously at each mesh point. Thus, for the example just men-

tioned, we can label all four variables (two velocities followed by momentum and then pressure) at a given mesh point as $u(k)$, ..., $u(k+3)$. Alternatively, all unknowns associated with one type of variable can be labeled first (e.g., first velocity components), followed by those associated with the second type of variables (e.g., second velocity components), etc. In either case, it is clear that there is redundant information in the graph of the adjacency matrix.

The *quotient* graph corresponding to the *physical mesh* can be used instead. This results in substantial savings in storage and computation. In the fluid flow example mentioned above, the storage can be reduced by a factor of almost 16 for the integer arrays needed to represent the graph. This is because the number of edges has been reduced by this much, while the number of vertices, which is usually much smaller, remains the same.

3.3 Permutations and Reorderings

Permuting the rows or the columns, or both the rows and columns, of a sparse matrix is a common operation. In fact, *reordering* rows and columns is one of the most important ingredients used in *parallel* implementations of both direct and iterative solution techniques. This section introduces the ideas related to these reordering techniques and their relations to the adjacency graphs of the matrices. Recall the notation introduced in Chapter 1 that the j-th column of a matrix is denoted by a_{*j} and the i-th row by a_{i*}.

3.3.1 Basic Concepts

We begin with a definition and new notation.

Definition 3.1 *Let A be a matrix and $\pi = \{i_1, i_2, \ldots, i_n\}$ a permutation of the set $\{1, 2, \ldots, n\}$. Then the matrices*

$$\begin{aligned} A_{\pi,*} &= \{a_{\pi(i),j}\}_{i=1,\ldots,n;j=1,\ldots,m}, \\ A_{*,\pi} &= \{a_{i,\pi(j)}\}_{i=1,\ldots,n;j=1,\ldots,m} \end{aligned}$$

are called row π-permutation and column π-permutation of A, respectively.

It is well known that any permutation of the set $\{1, 2, \ldots, n\}$ results from at most n interchanges, i.e., elementary permutations in which only two entries have been interchanged. An *interchange matrix* is the identity matrix with two of its rows interchanged. Denote by X_{ij} such matrices, with i and j being the numbers of the interchanged rows. Note that in order to interchange rows i and j of a matrix A, we only need to premultiply it by the matrix X_{ij}. Let $\pi = \{i_1, i_2, \ldots, i_n\}$ be an arbitrary permutation. This permutation is the product of a sequence of n consecutive interchanges $\sigma(i_k, j_k), k = 1, \ldots, n$. Then the rows of a matrix can be permuted by interchanging rows i_1, j_1, then rows i_2, j_2 of the resulting matrix, etc., and finally by interchanging i_n, j_n of the resulting matrix. Each of these operations can be achieved

by a premultiplication by X_{i_k,j_k}. The same observation can be made regarding the columns of a matrix: In order to interchange columns i and j of a matrix, postmultiply it by X_{ij}. The following proposition follows from these observations.

Proposition 3.2 *Let π be a permutation resulting from the product of the interchanges $\sigma(i_k, j_k)$, $k = 1, \ldots, n$. Then,*

$$A_{\pi,*} = P_\pi A, \quad A_{*,\pi} = AQ_\pi,$$

where

$$P_\pi = X_{i_n,j_n} X_{i_{n-1},j_{n-1}} \ldots X_{i_1,j_1}, \tag{3.1}$$
$$Q_\pi = X_{i_1,j_1} X_{i_2,j_2} \ldots X_{i_n,j_n}. \tag{3.2}$$

Products of interchange matrices are called *permutation matrices*. Clearly, a permutation matrix is nothing but the identity matrix with its rows (or columns) permuted.

Observe that $X_{i,j}^2 = I$, i.e., the square of an interchange matrix is the identity, or equivalently, the inverse of an interchange matrix is equal to itself, a property which is intuitively clear. It is easy to see that the matrices (3.1) and (3.2) satisfy

$$P_\pi Q_\pi = X_{i_n,j_n} X_{i_{n-1},j_{n-1}} \ldots X_{i_1,j_1} \times X_{i_1,j_1} X_{i_2,j_2} \ldots X_{i_n,j_n} = I,$$

which shows that the two matrices Q_π and P_π are nonsingular and that they are the inverse of one another. In other words, permuting the rows and the columns of a matrix, *using the same permutation*, actually performs a similarity transformation. Another important consequence arises because the products involved in the definitions (3.1) and (3.2) of P_π and Q_π occur in reverse order. Since each of the elementary matrices X_{i_k,j_k} is symmetric, the matrix Q_π is the transpose of P_π. Therefore,

$$Q_\pi = P_\pi^T = P_\pi^{-1}.$$

Since the inverse of the matrix P_π is its own transpose, permutation matrices are unitary.

Another way of deriving the above relationships is to express the permutation matrices P_π and P_π^T in terms of the identity matrix, whose columns or rows are permuted. It can easily be seen (See Exercise 3) that

$$P_\pi = I_{\pi,*}, \quad P_\pi^T = I_{*,\pi}.$$

It is then possible to verify directly that

$$A_{\pi,*} = I_{\pi,*}A = P_\pi A, \quad A_{*,\pi} = AI_{*,\pi} = AP_\pi^T.$$

It is important to interpret permutation operations for the linear systems to be solved. When the rows of a matrix are permuted, the order in which the equations are written is changed. On the other hand, when the columns are permuted, the unknowns are in effect *relabeled*, or *reordered*.

Example 3.1. Consider, for example, the linear system $Ax = b$ where

$$A = \begin{pmatrix} a_{11} & 0 & a_{13} & 0 \\ 0 & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & 0 \\ 0 & a_{42} & 0 & a_{44} \end{pmatrix}$$

and $\pi = \{1, 3, 2, 4\}$, then the (column-) permuted linear system is

$$\begin{pmatrix} a_{11} & a_{13} & 0 & 0 \\ 0 & a_{23} & a_{22} & a_{24} \\ a_{31} & a_{33} & a_{32} & 0 \\ 0 & 0 & a_{42} & a_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_3 \\ x_2 \\ x_4 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix}.$$

Note that only the unknowns have been permuted, not the equations, and in particular, the right-hand side has not changed. □

In the above example, only the columns of A have been permuted. Such one-sided permutations are not as common as two-sided permutations in sparse matrix techniques. In reality, this is often related to the fact that the diagonal elements in linear systems play a distinct and important role. For instance, diagonal elements are typically large in PDE applications and it may be desirable to preserve this important property in the permuted matrix. In order to do so, it is typical to apply the same permutation to both the columns and the rows of A. Such operations are called *symmetric permutations*, and if denoted by $A_{\pi,\pi}$, then the result of such symmetric permutations satisfies the relation

$$A_{\pi,\pi} = P_\pi A P_\pi^T.$$

The interpretation of the symmetric permutation is quite simple. The resulting matrix corresponds to renaming, or relabeling, or reordering the unknowns and then reordering the equations in the same manner.

Example 3.2. For the previous example, if the rows are permuted with the same permutation as the columns, the linear system obtained is

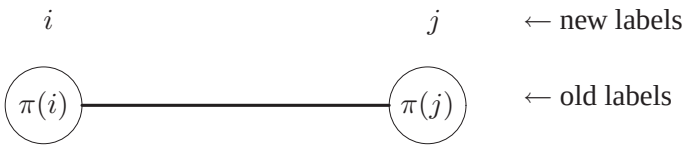
$$\begin{pmatrix} a_{11} & a_{13} & 0 & 0 \\ a_{31} & a_{33} & a_{32} & 0 \\ 0 & a_{23} & a_{22} & a_{24} \\ 0 & 0 & a_{42} & a_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_3 \\ x_2 \\ x_4 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_3 \\ b_2 \\ b_4 \end{pmatrix}.$$

Observe that the diagonal elements are now diagonal elements from the original matrix, placed in a different order on the main diagonal. □

3.3.2 Relations with the Adjacency Graph

From the point of view of graph theory, another important interpretation of a symmetric permutation is that *it is equivalent to relabeling the vertices of the graph* without

altering the edges. Indeed, let (i, j) be an edge in the adjacency graph of the original matrix A and let A' be the permuted matrix. Then $a'_{ij} = a_{\pi(i),\pi(j)}$ and as a result (i, j) is an edge in the adjacency graph of the permuted matrix A', if and only if $(\pi(i), \pi(j))$ is an edge in the graph of the original matrix A. In essence, it is as if we simply relabel each node with the "old" label $\pi(i)$ with the "new" label i. This is pictured in the following diagram:



Thus, the graph of the permuted matrix has not changed; rather, the labeling of the vertices has. In contrast, nonsymmetric permutations do not preserve the graph. In fact, they can transform an indirected graph into a directed one. Symmetric permutations change the order in which the nodes are considered in a given algorithm (such as Gaussian elimination) and this may have a tremendous impact on the performance of the algorithm.

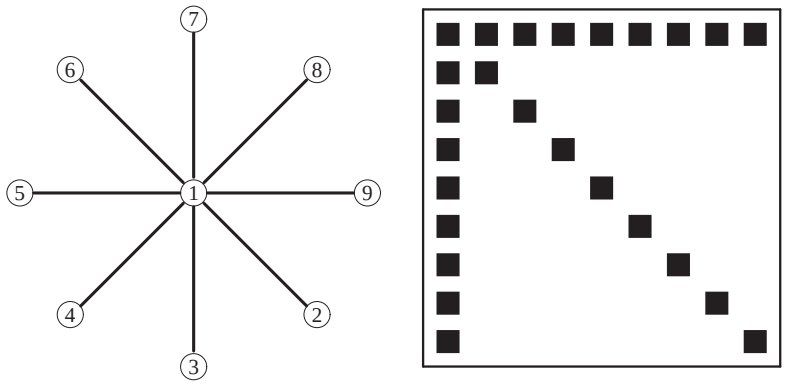


Figure 3.4: Pattern of a 9×9 arrow matrix and its adjacency graph.

Example 3.3. Consider the matrix illustrated in Figure 3.4 together with its adjacency graph. Such matrices are sometimes called "arrow" matrices because of their shape, but it would probably be more accurate to term them "star" matrices because of the structure of their graphs. If the equations are reordered using the permutation $9, 8, \ldots, 1$, the matrix and graph shown in Figure 3.5 are obtained.

Although the difference between the two graphs may seem slight, the matrices have a completely different structure, which may have a significant impact on the algorithms. As an example, if Gaussian elimination is used on the reordered matrix, no fill-in will occur, i.e., the L and U parts of the LU factorization will have the same structure as the lower and upper parts of A, respectively.

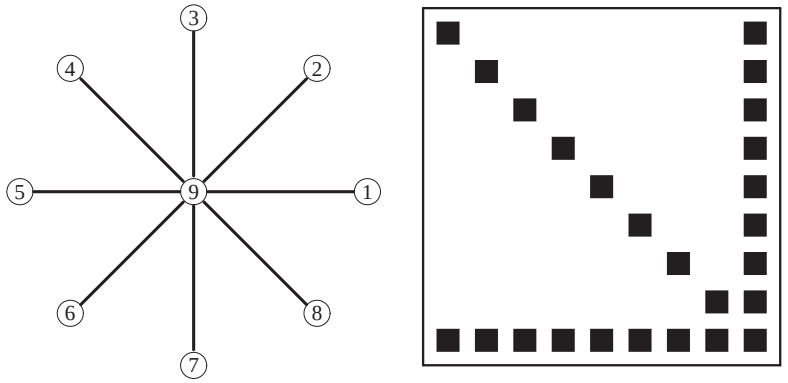


Figure 3.5: Adjacency graph and matrix obtained from above figure after permuting the nodes in reverse order.

On the other hand, Gaussian elimination on the original matrix results in disastrous fill-ins. Specifically, the L and U parts of the LU factorization are now dense matrices after the first step of Gaussian elimination. With direct sparse matrix techniques, it is important to find permutations of the matrix that will have the effect of reducing fill-ins during the Gaussian elimination process. □

To conclude this section, it should be mentioned that two-sided nonsymmetric permutations may also arise in practice. However, they are more common in the context of direct methods.

3.3.3 Common Reorderings

The type of reordering, or permutations, used in applications depends on whether a direct or an iterative method is being considered. The following is a sample of such reorderings which are more useful for iterative methods.

Level-set orderings. This class of orderings contains a number of techniques that are based on traversing the graph by *level sets*. A level set is defined recursively as the set of all unmarked neighbors of all the nodes of a previous level set. Initially, a level set consists of one node, although strategies with several starting nodes are also important and will be considered later. As soon as a level set is traversed, its nodes are marked and numbered. They can, for example, be numbered in the order in which they are traversed. In addition, the order in which each level itself is traversed gives rise to different orderings. For instance, the nodes of a certain level can be visited in the natural order in which they are listed. The neighbors of each of these nodes are then inspected. Each time, a neighbor of a visited vertex that is not numbered is encountered, it is added to the list and labeled as the next element of the next level set. This simple strategy is called *Breadth First Search* (BFS) traversal in graph theory. The ordering will depend on the way in which the nodes are traversed in each

level set. In BFS the elements of a level set are always traversed in the natural order in which they are listed. In the *Cuthill-McKee ordering* the nodes adjacent a a visited node are always traversed from lowest to highest degree.

ALGORITHM **3.1** *BFS*(G, v)

1. *Initialize* $S = \{v\}$, $seen = 1$, $\pi(seen) = v$; *Mark* v;
2. *While* $seen < n$ *Do*
3. $S_{new} = \emptyset$;
4. *For each node* v *in* S *do*
5. *For each unmarked* w *in adj*(v) *do*
6. *Add* w *to* S_{new};
7. *Mark* w;
8. $\pi(++seen) = w$;
9. *EndDo*
10. $S := S_{new}$
11. *EndDo*
12. *EndWhile*

In the above algorithm, the notation $\pi(++seen) = w$ in Line 8, uses a style borrowed from the C/C++ language. It states that $seen$ should be first incremented by one, and then $\pi(seen)$ is assigned w. Two important modifications will be made to this algorithm to obtain the Cuthill Mc Kee ordering. The first concerns the selection of the first node to begin the traversal. The second, mentioned above, is the orde in which the nearest neighbors of a given node are traversed.

ALGORITHM **3.2** *Cuthill-McKee (* G*)*

0. *Find an intial node* v *for the traversal*
1. *Initialize* $S = \{v\}$, $seen = 1$, $\pi(seen) = v$; *Mark* v;
2. *While* $seen < n$ *Do*
3. $S_{new} = \emptyset$;
4. *For each node* v *Do:*
5. $\pi(++seen) = v$;
6. *For each unmarked* w *in adj*(v)*, going from lowest to highest degree Do:*
7. *Add* w *to* S_{new};
8. *Mark* w;
9. *EndDo*
10. $S := S_{new}$
11. *EndDo*
12. *EndWhile*

The π array obtained from the procedure lists the nodes in the order in which they are visited and can, in a practical implementation, be used to store the level sets in succession. A pointer is needed to indicate where each set starts.

The main property of level sets is that, at the exception of the first and the last levels, they are are *graph separators*. A graph separator is a set of vertices, the removal of which separates the graph in two disjoint components. In fact if there are l levels and $V_1 = S_1 U S_2 \ldots S_{i-1}$, $V_2 = S_{i+1} U \ldots S_l$, then the nodes of V_1 are V_2 are not coupled. This is easy to prove by contradiction. A major consequence of this property is that the matrix resulting from the Cuthill-McKee (or BFS) ordering is block-tridiagonal, with the i-th block being of size $|S_i|$.

I order to explain the concept of level sets, the previous two algorithms were described with the explicit use of level sets. A more common, and somewhat simpler, implementation relies on *queues*. The queue implementation is as follows.

ALGORITHM **3.3** *Cuthill-McKee (G) – Queue implementation*

0. *Find an intial node v for the traversal*
1. *Initialize $Q = \{v\}$, Mark v;*
2. *While $|Q| < n$ Do*
3. $head++$ *;*
4. *For each unmarked w in adj(h), going from lowest to highest degree Do:*
5. *Append w to Q;*
6. *Mark w;*
7. *EndDo*
8. *EndWhile*

The final array Q will give the desired permutation π. Clearly, this implementation can also be applied to BFS. As an example, consider the finite element mesh problem illustrated in Figure 2.10 of Chapter 2, and assume that $v = 3$ is the initial node of the traversal. The state of the Q array after each step along with the head vertex $head$ and its adjacency list are shown in the following table. Note that the adjacency lists in the third column are listed by increasing degrees.

Q	$head$	$adj(head)$
3	3	7, 10, 8
3, 7, 10, 8	7	1, 9
3, 7, 10, 8, 1, 9	10	5, 11
3, 7, 10, 8, 1, 9, 5, 11	8	2
3, 7, 10, 8, 1, 9, 5, 11, 2	1	-
3, 7, 10, 8, 1, 9, 5, 11, 2	9	-
3, 7, 10, 8, 1, 9, 5, 11, 2	5	14, 12
3, 7, 10, 8, 1, 9, 5, 11, 2, 14, 12	11	13
3, 7, 10, 8, 1, 9, 5, 11, 2, 14, 12, 13	2	-
3, 7, 10, 8, 1, 9, 5, 11, 2, 14, 12, 13	14	6, 15
3, 7, 10, 8, 1, 9, 5, 11, 2, 14, 12, 13, 6, 15	12	4
3, 7, 10, 8, 1, 9, 5, 11, 2, 14, 12, 13, 6, 15, 4		

An implementation using explicit levels, would find the sets $S_1 = \{3\}$, $S_2 = \{7, 8, 10\}$, $S_3 = \{1, 9, 5, 11, 2\}$, $S_4 = \{14, 12, 13\}$, and $S_5 = \{6, 15, 4\}$. The

new labeling of the graph along with the corresponding matrix pattern are shown in Figure 3.6. The partitionning of the matrix pattern corresponds to the levels.

In 1971, George [142] observed that *reversing* the Cuthill-McKee ordering yields a better scheme for sparse Gaussian elimination. The simplest way to understand this is to look at the two graphs produced by these orderings. The results of the standard and reversed Cuthill-McKee orderings on the sample finite element mesh problem seen earlier are shown in Figures 3.6 and 3.7, when the initial node is $i_1 = 3$ (relative to the labeling of the original ordering of Figure 2.10). The case of the figure, corresponds to a variant of CMK in which the traversals in Line 6, is done in a random order instead of according to the degree. A large part of the structure of the two matrices consists of little "arrow" submatrices, similar to the ones seen in Example 3.3. In the case of the regular CMK ordering, these arrows point upward, as in Figure 3.4, a consequence of the level set labeling. These blocks are similar the star matrices of Figure 3.4. As a result, Gaussian elimination will essentially fill in the square blocks which they span. As was indicated in Example 3.3, a remedy is to reorder the nodes backward, as is done globally in the reverse Cuthill-McKee strategy. For the reverse CMK ordering, the arrows are pointing downward, as in Figure 3.5, and Gaussian elimination yields much less fill-in.

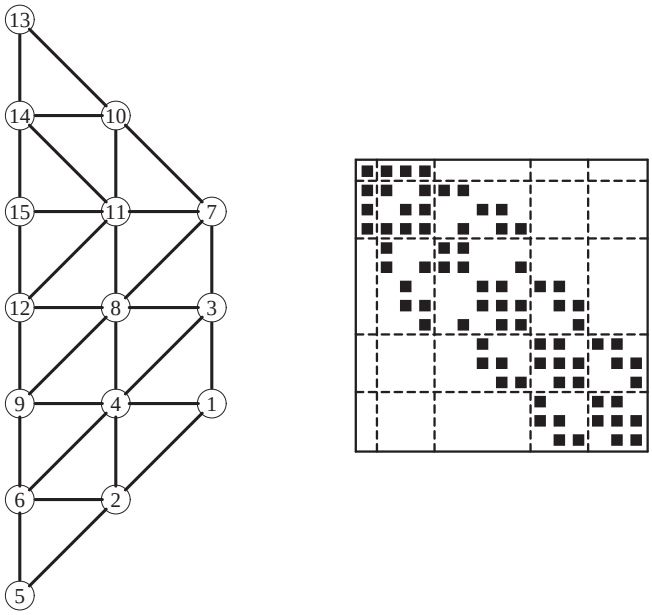


Figure 3.6: Graph and matrix pattern for example pf Figure 2.10 after Cuthill-McKee ordering.

Example 3.4. The choice of the initial node in the CMK and RCMK orderings may be important. Referring to the original ordering of Figure 2.10, the previous illustration used $i_1 = 3$. However, it is clearly a poor choice if matrices with small bandwidth or *profile* are desired. If $i_1 = 1$ is selected instead, then the reverse

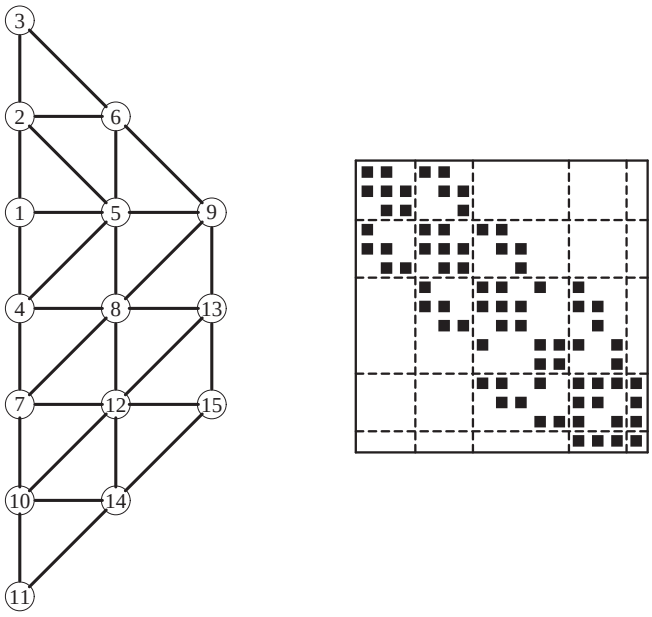


Figure 3.7: Reverse Cuthill-McKee ordering.

Cuthill-McKee algorithm produces the matrix in Figure 3.8, which is more suitable for banded or *skyline* solvers. □

Independent set orderings. The matrices that arise in the model finite element problems seen in Figures 2.7, 2.10, and 3.2 are all characterized by an upper-left block that is diagonal, i.e., they have the structure

$$A = \begin{pmatrix} D & E \\ F & C \end{pmatrix}, \tag{3.3}$$

in which D is diagonal and C, E, and F are sparse matrices. The upper-diagonal block corresponds to unknowns from the previous levels of refinement and its presence is due to the ordering of the equations in use. As new vertices are created in the refined grid, they are given new numbers and the initial numbering of the vertices is unchanged. Since the old connected vertices are "cut" by new ones, they are no longer related by equations. Sets such as these are called *independent sets*. Independent sets are especially useful in parallel computing, for implementing both direct and iterative methods.

Referring to the adjacency graph $G = (V, E)$ of the matrix, and denoting by (x, y) the edge from vertex x to vertex y, an *independent set* S is a subset of the vertex set V such that

$$\text{if } x \in S, \quad \text{then} \quad \{(x, y) \in E \text{ or } (y, x) \in E\} \rightarrow y \notin S.$$

To explain this in words: Elements of S are not allowed to be connected to other elements of S either by incoming or outgoing edges. An independent set is

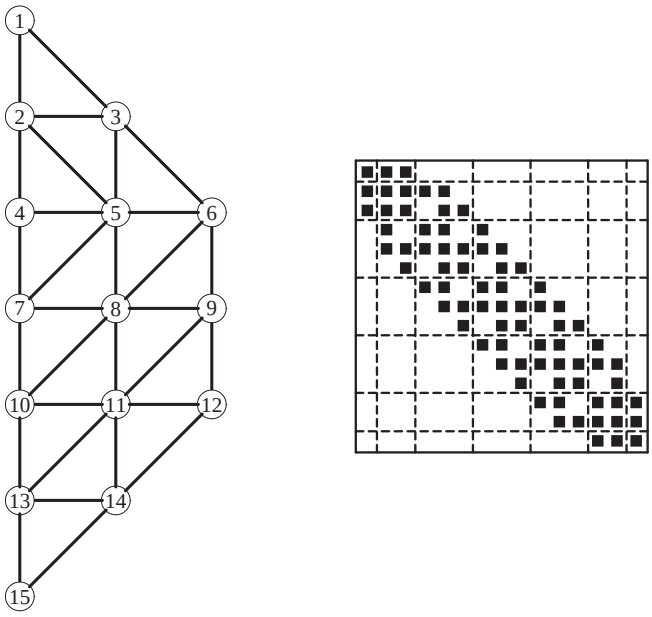


Figure 3.8: Reverse Cuthill-McKee starting with $i_1 = 1$.

maximal if it cannot be augmented by elements in its complement to form a larger independent set. Note that a maximal independent set is by no means the largest possible independent set that can be found. In fact, finding the independent set of maximum cardinal is NP-hard [183]. In the following, the term *independent set* always refers to *maximal independent set*.

There are a number of simple and inexpensive heuristics for finding large maximal independent sets. A greedy heuristic traverses the nodes in a given order, and if a node is not already marked, it selects the node as a new member of S. Then this node is marked along with its nearest neighbors. Here, a nearest neighbor of a node x means any node linked to x by an incoming or an outgoing edge.

ALGORITHM **3.4** *Greedy Algorithm for ISO*

1. *Set* $S = \emptyset$.
2. *For* $j = 1, 2, \ldots, n$ *Do:*
3. *If node* j *is not marked then*
4. $S = S \cup \{j\}$
5. *Mark* j *and all its nearest neighbors*
6. *EndIf*
7. *EndDo*

In the above algorithm, the nodes are traversed in the natural order $1, 2, \ldots, n$, but they can also be traversed in any permutation $\{i_1, \ldots, i_n\}$ of $\{1, 2, \ldots, n\}$. Since the size of the reduced system is $n - |S|$, it is reasonable to try to maximize the size of S in order to obtain a small reduced system. It is possible to give a rough

idea of the size of S. Assume that the maximum degree of each node does not exceed ν. Whenever the above algorithm accepts a node as a new member of S, it potentially puts all its nearest neighbors, i.e., at most ν nodes, in the complement of S. Therefore, if s is the size of S, the size of its complement, $n - s$, is such that $n - s \leq \nu s$, and as a result,

$$s \geq \frac{n}{1+\nu}.$$

This lower bound can be improved slightly by replacing ν with the maximum degree ν_S of *all the vertices that constitute* S. This results in the inequality

$$s \geq \frac{n}{1+\nu_S},$$

which suggests that it may be a good idea to first visit the nodes with smaller degrees. In fact, this observation leads to a general heuristic regarding a good order of traversal. The algorithm can be viewed as follows: Each time a node is visited, remove it and its nearest neighbors from the graph, and then visit a node from the remaining graph. Continue in the same manner until all nodes are exhausted. Every node that is visited is a member of S and its nearest neighbors are members of $\bar{S}$. As result, if ν_i is the degree of the node visited at step i, adjusted for all the edge deletions resulting from the previous visitation steps, then the number n_i of nodes that are left at step i satisfies the relation

$$n_i = n_{i-1} - \nu_i - 1.$$

The process adds a new element to the set S at each step and stops when $n_i = 0$. In order to maximize $|S|$, the number of steps in the procedure must be maximized. The difficulty in the analysis arises from the fact that the degrees are updated at each step i because of the removal of the edges associated with the removed nodes. If the process is to be lengthened, a rule of thumb would be to visit the nodes that have the smallest degrees first.

ALGORITHM **3.5** *Increasing Degree Traversal for ISO*

1. *Set $S = \emptyset$. Find an ordering $i_1, \ldots, i_n$ of the nodes by increasing degree.*
2. *For $j = 1, 2, \ldots n$, Do:*
3. *If node i_j is not marked then*
4. *$S = S \cup \{i_j\}$*
5. *Mark i_j and all its nearest neighbors*
6. *EndIf*
7. *EndDo*

A refinement to the above algorithm would be to update the degrees of all nodes involved in a removal, and dynamically select the one with the smallest degree as the next node to be visited. This can be implemented efficiently using a min-heap data structure. A different heuristic is to attempt to maximize the number of elements in S by a form of local optimization which determines the order of traversal dynamically. In the following, removing a vertex from a graph means deleting the vertex and all edges incident to/from this vertex.

Example 3.5. The algorithms described in this section were tested on the same example used before, namely, the finite element mesh problem of Figure 2.10. Here, all strategies used yield the initial independent set in the matrix itself, which corresponds to the nodes of all the previous levels of refinement. This may well be optimal in this case, i.e., a larger independent set may not exist. □

Multicolor orderings. Graph coloring is a familiar problem in computer science which refers to the process of labeling (coloring) the nodes of a graph in such a way that no two adjacent nodes have the same label (color). The goal of graph coloring is to obtain a colored graph which uses the smallest possible number of colors. However, optimality in the context of numerical linear algebra is a secondary issue and simple heuristics do provide adequate colorings. Basic methods for obtaining a multicoloring of an arbitrary grid are quite simple. They rely on greedy techniques, a simple version of which is as follows.

ALGORITHM **3.6** *Greedy Multicoloring Algorithm*

1. *For* $i = 1, \ldots, n$ *Do: set* $Color(i) = 0$.
2. *For* $i = 1, 2, \ldots, n$ *Do:*
3. *Set Color*$(i) = \min \{k > 0 \mid k \neq$ *Color*$(j), \forall\, j \in \mathrm{Adj}(i))\}$
4. *EndDo*

Line 3 assigns the smallest *allowable* color number to node i. Allowable means a positive number that is different from the colors of the neighbors of node i. The procedure is illustrated in Figure 3.9. The node being colored in the figure is indicated by an arrow. It will be assigned color number 3, the smallest positive integer different from 1, 2, 4, 5.

In the above algorithm, the order $1, 2, \ldots, n$ has been arbitrarily selected for traversing the nodes and coloring them. Instead, the nodes can be traversed in any order $\{i_1, i_2, \ldots, i_n\}$. If a graph is *bipartite*, i.e., if it can be colored with two colors, then the algorithm will find the optimal two-color (Red-Black) ordering for *Breadth-First* traversals. In addition, if a graph is bipartite, it is easy to show that the algorithm will find two colors for any traversal which, at a given step, visits an unmarked node that is adjacent to at least one visited node. In general, the number of colors needed does not exceed the maximum degree of each node +1. These properties are the subject of Exercises 11 and 10.

Example 3.6. Figure 3.10 illustrates the algorithm for the same example used earlier, i.e., the finite element mesh problem of Figure 2.10. The dashed lines separate the different color sets found. Four colors are found in this example. □

Once the colors have been found, the matrix can be permuted to have a block structure in which the diagonal blocks are diagonal. Alternatively, the color sets $S_j = [i_1^{(j)}, \ldots, i_{n_j}^{(j)}]$ and the permutation array in the algorithms can be used.

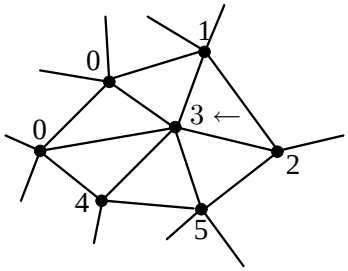


Figure 3.9: The greedy multicoloring algorithm.

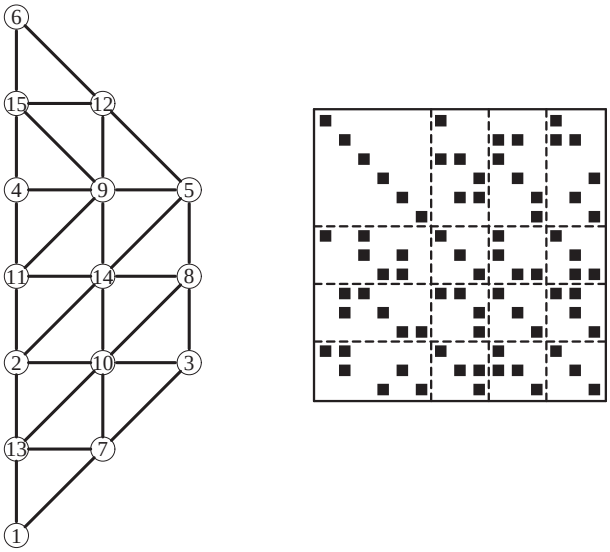


Figure 3.10: Graph and matrix corresponding to mesh of Figure 2.10 after multicolor ordering.

3.3.4 Irreducibility

Remember that a *path* in a graph is a sequence of vertices $v_1, v_2, \ldots, v_k$, which are such that (v_i, v_{i+1}) is an edge for $i = 1, \ldots, k-1$. Also, a graph is said to be *connected* if there is a path between any pair of two vertices in V. A *connected component* in a graph is a *maximal subset* of vertices which all can be connected to one another by paths in the graph. Now consider matrices whose graphs may be *directed*. A matrix is *reducible* if its graph is not connected, and *irreducible* otherwise. When a matrix is reducible, then it can be permuted by means of *symmetric* permutations into a block upper triangular matrix of the form

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} & \ldots \\ & A_{22} & A_{23} & \ldots \\ & & \ddots & \vdots \\ & & & A_{pp} \end{pmatrix},$$

where each partition corresponds to a connected component. It is clear that linear systems with the above matrix can be solved through a sequence of subsystems with the matrices A_{ii}, $i = p, p-1, \ldots, 1$.

3.4 Storage Schemes

In order to take advantage of the large number of zero elements, special schemes are required to store sparse matrices. The main goal is to represent only the nonzero elements, and to be able to perform the common matrix operations. In the following, Nz denotes the total number of nonzero elements.

The simplest storage scheme for sparse matrices is the so-called coordinate format. The data structure consists of three arrays: (1) a real array containing all the real (or complex) values of the nonzero elements of A in any order; (2) an integer array containing their row indices; and (3) a second integer array containing their column indices. All three arrays are of length Nz, the number of nonzero elements.

Example 3.7. The matrix

$$A = \begin{pmatrix} 1. & 0. & 0. & 2. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 6. & 0. & 7. & 8. & 9. \\ 0. & 0. & 10. & 11. & 0. \\ 0. & 0. & 0. & 0. & 12. \end{pmatrix}$$

will be represented (for example) by

AA	12.	9.	7.	5.	1.	2.	11.	3.	6.	4.	8.	10.
JR	5	3	3	2	1	1	4	2	3	2	3	4
JC	5	5	3	4	1	4	4	1	1	2	4	3

□

In the above example, the elements are listed in an arbitrary order. In fact, they are usually listed by row or columns. If the elements were listed by row, the array JC which contains redundant information might be replaced by an array which points to the beginning of each row instead. This would involve nonnegligible savings in storage. The new data structure has three arrays with the following functions:

- A real array AA contains the real values a_{ij} stored row by row, from row 1 to n. The length of AA is Nz.
- An integer array JA contains the column indices of the elements a_{ij} as stored in the array AA. The length of JA is Nz.
- An integer array IA contains the pointers to the beginning of each row in the arrays AA and JA. Thus, the content of $IA(i)$ is the position in arrays AA and JA where the i-th row starts. The length of IA is $n+1$ with $IA(n+1)$

containing the number $IA(1) + Nz$, i.e., the address in A and JA of the beginning of a fictitious row number $n + 1$.

Thus, the above matrix may be stored as follows:

AA	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.
JA	1	4	1	2	4	1	3	4	5	3	4	5
IA	1	3	6	10	12	13						

This format is probably the most popular for storing general sparse matrices. It is called the *Compressed Sparse Row* (CSR) format. This scheme is preferred over the coordinate scheme because it is often more useful for performing typical computations. On the other hand, the coordinate scheme is advantageous for its simplicity and its flexibility. It is often used as an "entry" format in sparse matrix software packages.

There are a number of variations for the Compressed Sparse Row format. The most obvious variation is storing the columns instead of the rows. The corresponding scheme is known as the *Compressed Sparse Column* (CSC) scheme.

Another common variation exploits the fact that the diagonal elements of many matrices are all usually nonzero and/or that they are accessed more often than the rest of the elements. As a result, they can be stored separately. The *Modified Sparse Row* (MSR) format has only two arrays: a real array AA and an integer array JA. The first n positions in AA contain the diagonal elements of the matrix in order. The unused position $n + 1$ of the array AA may sometimes carry some information concerning the matrix.

Starting at position $n + 2$, the nonzero entries of AA, excluding its diagonal elements, are stored by row. For each element $AA(k)$, the integer $JA(k)$ represents its column index on the matrix. The $n + 1$ first positions of JA contain the pointer to the beginning of each row in AA and JA. Thus, for the above example, the two arrays will be as follows:

AA	1.	4.	7.	11.	12.	*	2.	3.	5.	6.	8.	9.	10.
JA	7	8	10	13	14	14	4	1	4	1	4	5	3

The star denotes an unused location. Notice that $JA(n) = JA(n + 1) = 14$, indicating that the last row is a zero row, once the diagonal element has been removed.

Diagonally structured matrices are matrices whose nonzero elements are located along a small number of diagonals. These diagonals can be stored in a rectangular array `DIAG(1:n,1:Nd)`, where `Nd` is the number of diagonals. The offsets of each of the diagonals with respect to the main diagonal must be known. These will be stored in an array `IOFF(1:Nd)`. Thus, the element $a_{i,i+\mathrm{ioff}(j)}$ of the original matrix is located in position (i, j) of the array `DIAG`, i.e.,

$$\mathtt{DIAG}(i, j) \leftarrow a_{i,i+\mathrm{ioff}(j)}.$$

The order in which the diagonals are stored in the columns of DIAG is generally unimportant, though if several more operations are performed with the main diagonal, storing it in the first column may be slightly advantageous. Note also that all the diagonals except the main diagonal have fewer than n elements, so there are positions in DIAG that will not be used.

Example 3.8. For example, the following matrix which has three diagonals

$$A = \begin{pmatrix} 1. & 0. & 2. & 0. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 0. & 6. & 7. & 0. & 8. \\ 0. & 0. & 9. & 10. & 0. \\ 0. & 0. & 0. & 11. & 12. \end{pmatrix}$$

will be represented by the two arrays

$$\texttt{DIAG} = \begin{array}{|ccc|} \hline * & 1. & 2. \\ 3. & 4. & 5. \\ 6. & 7. & 8. \\ 9. & 10. & * \\ 11 & 12. & * \\ \hline \end{array} \qquad \texttt{IOFF} = \boxed{\texttt{-1 0 2}}\,.$$

□

A more general scheme which is popular on vector machines is the so-called Ellpack-Itpack format. The assumption in this scheme is that there are at most Nd nonzero elements per row, where Nd is small. Then two rectangular arrays of dimension n $\times$ Nd each are required (one real and one integer). The first, COEF, is similar to DIAG and contains the nonzero elements of A. The nonzero elements of each row of the matrix can be stored in a row of the array COEF(1:n,1:Nd), completing the row by zeros as necessary. Together with COEF, an integer array JCOEF(1:n,1:Nd) must be stored which contains the column positions of each entry in COEF.

Example 3.9. Thus, for the matrix of the previous example, the Ellpack-Itpack storage scheme is

$$\texttt{COEF} = \begin{array}{|ccc|} \hline 1. & 2. & 0. \\ 3. & 4. & 5. \\ 6. & 7. & 8. \\ 9. & 10. & 0. \\ 11 & 12. & 0. \\ \hline \end{array} \qquad \texttt{JCOEF} = \begin{array}{|ccc|} \hline 1 & 3 & 1 \\ 1 & 2 & 4 \\ 2 & 3 & 5 \\ 3 & 4 & 4 \\ 4 & 5 & 5 \\ \hline \end{array}\,.$$

A certain column number must be chosen for each of the zero elements that must be added to pad the shorter rows of A, i.e., rows 1, 4, and 5. In this example, those integers are selected to be equal to the row numbers, as can be seen in the JCOEF array. This is somewhat arbitrary, and in fact, any integer between 1 and n would be

acceptable. However, there may be good reasons for not inserting the same integers too often, e.g. a constant number, for performance considerations. □

3.5 Basic Sparse Matrix Operations

The matrix-by-vector product is an important operation which is required in most of the iterative solution algorithms for solving sparse linear systems. This section shows how these can be implemented for a small subset of the storage schemes considered earlier.

The following FORTRAN 90 segment shows the main loop of the matrix-by-vector operation for matrices stored in the Compressed Sparse Row stored format.

```
DO I=1, N
   K1 = IA(I)
   K2 = IA(I+1)-1
   Y(I) = DOTPRODUCT(A(K1:K2),X(JA(K1:K2)))
ENDDO
```

Notice that each iteration of the loop computes a different component of the resulting vector. This is advantageous because each of these components can be computed independently. If the matrix is stored by columns, then the following code could be used instead:

```
DO J=1, N
   K1 = IA(J)
   K2 = IA(J+1)-1
   Y(JA(K1:K2)) = Y(JA(K1:K2))+X(J)*A(K1:K2)
ENDDO
```

In each iteration of the loop, a multiple of the j-th column is added to the result, which is assumed to have been initially set to zero. Notice now that the outer loop is no longer parallelizable. An alternative to improve parallelization is to try to split the vector operation in each inner loop. The inner loop has few operations, in general, so this is unlikely to be a sound approach. This comparison demonstrates that data structures may have to change to improve performance when dealing with high performance computers.

Now consider the matrix-by-vector product in diagonal storage.

```
DO J=1, NDIAG
   JOFF = IOFF(J)
   DO I=1, N
      Y(I) = Y(I) +DIAG(I,J)*X(JOFF+I)
   ENDDO
ENDDO
```

Here, each of the diagonals is multiplied by the vector x and the result added to the vector y. It is again assumed that the vector y has been filled with zeros at the start of the loop. From the point of view of parallelization and/or vectorization, the above code is probably the better to use. On the other hand, it is not general enough.

Solving a lower or upper triangular system is another important "kernel" in sparse matrix computations. The following segment of code shows a simple routine for solving a unit lower triangular system $Lx = y$ for the CSR storage format.

```
X(1) = Y(1)
DO I = 2, N
   K1 = IAL(I)
   K2 = IAL(I+1)-1
   X(I)=Y(I)-DOTPRODUCT(AL(K1:K2),X(JAL(K1:K2)))
ENDDO
```

At each step, the inner product of the current solution x with the i-th row is computed and subtracted from $y(i)$. This gives the value of $x(i)$. The `dotproduct` function computes the dot product of two arbitrary vectors `u(k1:k2)` and `v(k1:k2)`. The vector `AL(K1:K2)` is the i-th row of the matrix L in sparse format and `X(JAL(K1:K2))` is the vector of the components of `X` *gathered* into a short vector which is consistent with the column indices of the elements in the row `AL(K1:K2)`.

3.6 Sparse Direct Solution Methods

Most direct methods for sparse linear systems perform an LU factorization of the original matrix and try to reduce cost by minimizing fill-ins, i.e., nonzero elements introduced during the elimination process in positions which were initially zeros. The data structures employed are rather complicated. The early codes relied heavily on *linked lists* which are convenient for inserting new nonzero elements. Linked-list data structures were dropped in favor of other more dynamic schemes that leave some initial elbow room in each row for the insertions, and then adjust the structure as more fill-ins are introduced.

A typical sparse direct solution solver for positive definite matrices consists of four phases. First, preordering is applied to reduce fill-in. Two popular methods are used: minimum degree ordering and nested-dissection ordering. Second, a symbolic factorization is performed. This means that the factorization is processed only symbolically, i.e., without numerical values. Third, the numerical factorization, in which the actual factors L and U are formed, is processed. Finally, the forward and backward triangular sweeps are executed for each different right-hand side. In a code where numerical pivoting is necessary, the symbolic phase cannot be separated from the numerical factorization.

3.6.1 Minimum degree ordering

The minimum degree (MD) algorithm is perhaps the most popular strategy for minimizeing fill-in in sparse Gaussian elimination, specifically for SPD matrices. At a given step of Gaussian elimination, this strategy selects the node with the smallest degree as the next pivot row. This will tend to reduce fill-in. To be exact, it will minimize (locally) an upper bound for the number of fill-ins that will be introduced at the corrresponding step of Gaussian Elimination.

In contrast with the Cuthill McKee ordering, minimum degree ordering does not have, nor does it attempt to have, a banded structure. While the algorithm is excellent for sparse direct solvers, it has been observed that it does not perform as the RCM ordering when used in conjunction with preconditioning (Chapter 10).

The Multiple Minimum Degree algorithm is a variation due to Liu [204, 143] which exploits independent sets of pivots at each step. Degrees of nodes adjacent to any vertex in the independent set are updated only after all vertices in the set are processed.

3.6.2 Nested Dissection ordering

Nested dissection is used primarily to reduce fill-in in sparse direct solvers for Symmetric Positive Definite matrices. The technique is easily described with the help of recursivity and by exploiting the concept of 'separators'. A set S of vertices in a graph is called a separator if the removal of S results in the graph being split in two disjoint subgraphs. For example, each of the intermediate levels in the BFS algorithm is in fact a separator. The nested dissection algorithm can be succinctly described by the following algorithm

ALGORITHM **3.7** *ND*$(G, nmin)$

1. *If* $|V| \leq nmin$
2. *label nodes of* V
3. *Else*
4. *Find a separator* S *for* V
5. *Label the nodes of* S
6. *Split* V *into* G_L, G_R *by removing* S
7. *ND*$(G_L, nmin)$
8. *ND*$(G_R, nmin)$
9. *End*

The labeling of the nodes in Lines 2 and 5, usually proceeds in sequence, so for example, in Line 5, the nodes of S are labeled in a certain order, starting from the last labeled node so far in the procedure. The main step of the ND procedure is to separate the graph in three parts, two of which have no coupling between each other. The third set has couplings with vertices from both of the first sets and is referred to as a sepator. The key idea is to separate the graph in this way and then repeat the

process recursively in each subgraph. The nodes of the separator are numbered last. An illustration is shown in 3.11.

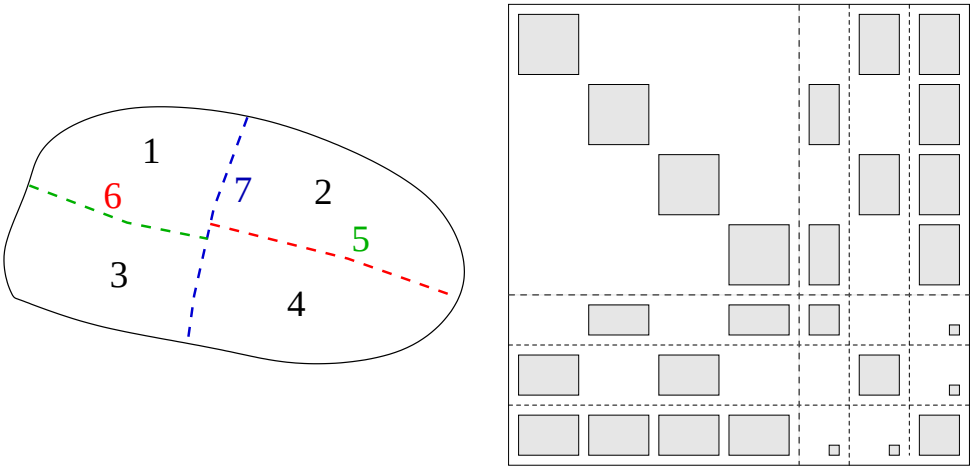


Figure 3.11: Nested dissection ordering and corresponding reordered matrix

3.7 Test Problems

For comparison purposes it is important to use a common set of test matrices that represent a wide spectrum of applications. There are two distinct ways of providing such data sets. The first approach is to collect sparse matrices in a well-specified standard format from various applications. This approach is used in the Harwell-Boeing collection of test matrices. The second approach is to generate these matrices with a few sample programs such as those provided in the SPARSKIT library [245]. The coming chapters will use examples from these two sources. In particular, five test problems will be emphasized for their varying degrees of difficulty.

The SPARSKIT package can generate matrices arising from the discretization of the two- or three-dimensional Partial Differential Equations

$$-\frac{\partial}{\partial x}\left(a\frac{\partial u}{\partial x}\right)-\frac{\partial}{\partial y}\left(b\frac{\partial u}{\partial y}\right)-\frac{\partial}{\partial z}\left(c\frac{\partial u}{\partial z}\right) + \frac{\partial\left(du\right)}{\partial x}+\frac{\partial\left(eu\right)}{\partial y}+\frac{\partial\left(fu\right)}{\partial z}+gu = h$$

on rectangular regions with general mixed-type boundary conditions. In the test problems, the regions are the square $\Omega = (0,1)^2$, or the cube $\Omega = (0,1)^3$; the Dirichlet condition $u = 0$ is always used on the boundary. Only the discretized matrix is of importance, since the right-hand side will be created artificially. Therefore, the right-hand side, h, is not relevant.

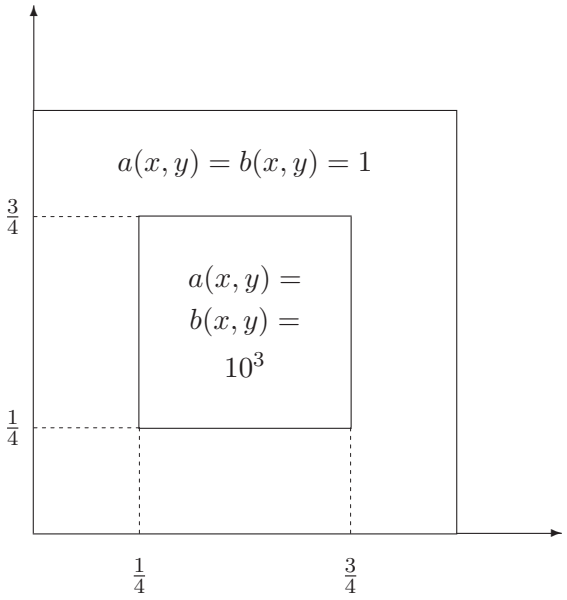


Figure 3.12: Physical domain and coefficients for Problem 2.

Problem 1: F2DA. In the first test problem which will be labeled F2DA, the domain is two-dimensional, with

$$a(x,y) = b(x,y) = 1.0$$

and

$$d(x,y) = \gamma(x+y), \quad e(x,y) = \gamma(x-y), \quad f(x,y) = g(x,y) = 0.0, \qquad (3.4)$$

where the constant γ is equal to 10. If the number of points in each direction is 34, then there are $n_x = n_y = 32$ interior points in each direction and a matrix of size $n = n_x \times n_y = 32^2 = 1024$ is obtained. In this test example, as well as the other ones described below, the right-hand side is generated as

$$b = Ae,$$

in which $e = (1, 1, \ldots, 1)^T$. The initial guess is always taken to be a vector of pseudo-random values.

Problem 2: F2DB. The second test problem is similar to the previous one but involves discontinuous coefficient functions a and b. Here, $n_x = n_y = 32$ and the functions d, e, f, g are also defined by (3.4). However, the functions a and b now both take the value 1,000 inside the subsquare of width $\frac{1}{2}$ centered at $(\frac{1}{2}, \frac{1}{2})$, and one elsewhere in the domain, i.e.,

$$a(x,y) = b(x,y) = \begin{cases} 10^3 & \text{if } \frac{1}{4} < x, y < \frac{3}{4} \\ 1 & \text{otherwise} \end{cases}.$$

The domain and coefficients for this problem are shown is Figure 3.12.

Problem 3: F3D. The third test problem is three-dimensional with $n_x = n_y = n_z = 16$ internal mesh points in each direction leading to a problem of size $n = 4096$. In this case, we take

$$a(x,y,z) = b(x,y,z) = c(x,y,z) = 1$$

$$d(x,y,z) = \gamma e^{xy}, \quad e(x,y,z) = \gamma e^{-xy},$$

and

$$f(x,y,z) = g(x,y,z) = 0.0.$$

The constant γ is taken to be equal to 10.0 as before.

The Harwell-Boeing collection is a large data set consisting of test matrices which have been contributed by researchers and engineers from many different disciplines. These have often been used for test purposes in the literature [108]. The collection provides a data structure which constitutes an excellent medium for exchanging matrices. The matrices are stored as ASCII files with a very specific format consisting of a four- or five-line header. Then, the data containing the matrix is stored in CSC format together with any right-hand sides, initial guesses, or exact solutions when available. The SPARSKIT library also provides routines for reading and generating matrices in this format.

Only one matrix from the collection was selected for testing the algorithms described in the coming chapters. The matrices in the last two test examples are both irregularly structured.

Problem 4: ORS The matrix selected from the Harwell-Boeing collection is ORSIRR1. This matrix arises from a reservoir engineering problem. Its size is $n = 1030$ and it has a total of Nz =6,858 nonzero elements. The original problem is based on a $21 \times 21 \times 5$ irregular grid. In this case and the next one, the matrices are preprocessed by scaling their rows and columns.

Problem 5: FID This test matrix is extracted from the well known fluid flow simulation package FIDAP [120]. It is actually the test example number 36 from this package and features a two-dimensional Chemical Vapor Deposition in a Horizontal Reactor. The matrix has a size of $n = 3079$ and has $Nz = 53843$ nonzero elements. It has a symmetric pattern and few diagonally dominant rows or columns. The rows and columns are prescaled in the same way as in the previous example. Figure 3.13 shows the patterns of the matrices ORS and FID.

PROBLEMS

P-3.1 Consider the mesh of a discretized PDE. In which situations is the graph representing this mesh the same as the adjacency graph of the matrix? Give examples from both Finite Difference and Finite Element discretizations.

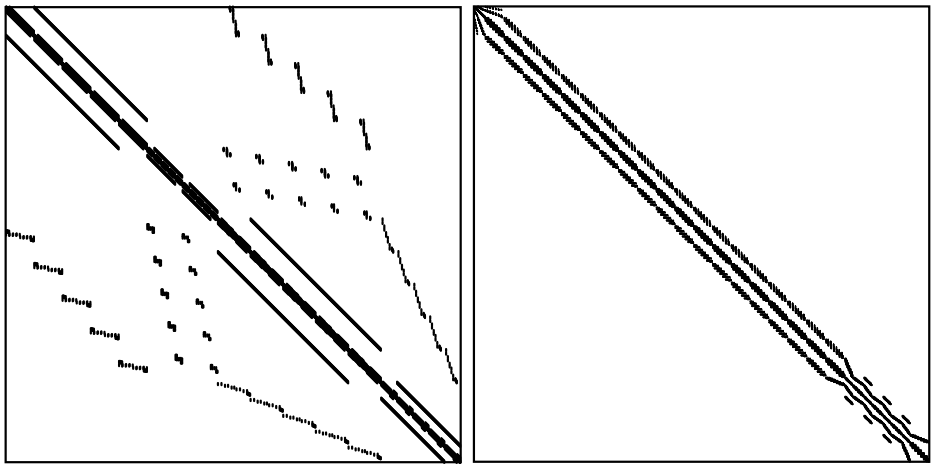


Figure 3.13: Patterns of the matrices ORS (left) and FID (right).

P-3.2 Let A and B be two sparse (square) matrices of the same dimension. How can the graph of $C = A + B$ be characterized with respect to the graphs of A and B?

P-3.3 Consider the matrix defined as

$$P_\pi = I_{\pi,*}.$$

Show directly (without using Proposition 3.2 or interchange matrices) that the following three relations hold

$$\begin{aligned} A_{\pi,*} &= I_{\pi,*}A \\ I_{*,\pi} &= P_\pi^T \\ AP_\pi^T &= A_{*,\pi}. \end{aligned}$$

P-3.4 Consider the two matrices

$$A = \begin{pmatrix} \star & \star & 0 & \star & 0 & 0 \\ 0 & \star & 0 & 0 & 0 & \star \\ 0 & \star & \star & 0 & 0 & 0 \\ 0 & \star & 0 & 0 & \star & 0 \\ 0 & 0 & 0 & 0 & \star & 0 \\ 0 & 0 & 0 & 0 & 0 & \star \end{pmatrix} \quad B = \begin{pmatrix} \star & 0 & 0 & 0 & 0 & 0 \\ \star & 0 & \star & 0 & \star & 0 \\ 0 & \star & 0 & 0 & 0 & 0 \\ \star & \star & 0 & 0 & 0 & 0 \\ 0 & \star & 0 & \star & \star & 0 \\ 0 & 0 & \star & 0 & 0 & \star \end{pmatrix}$$

where a $\star$ represents an arbitrary nonzero element.

a. Show the adjacency graphs of the matrices A, B, AB, and BA. (Assume that there are no numerical cancellations in computing the products AB and BA). Since there are zero diagonal elements, represent explicitly the cycles corresponding to the (i, i) edges when they are present.

b. Consider the matrix $C = AB$. Give an interpretation of an edge in the graph of C in terms of edges in the graph of A and B. Verify this answer using the above matrices.

c. Consider the particular case in which $B = A$. Give an interpretation of an edge in the graph of C in terms of paths of length two in the graph of A. The paths must take into account the cycles corresponding to nonzero diagonal elements of A.

d. Now consider the case where $B = A^2$. Give an interpretation of an edge in the graph of $C = A^3$ in terms of paths of length three in the graph of A. Generalize the result to arbitrary powers of A.

P-3.5 Consider two matrices A and B of dimension $n \times n$, whose diagonal elements are all nonzeros. Let E_X denote the set of edges in the adjacency graph of a matrix X (i.e., the set of pairs (i, j) such $X_{ij} \neq 0$), then show that

$$E_{AB} \supset E_A \cup E_B \ .$$

Give extreme examples when $|E_{AB}| = n^2$ while $E_A \cup E_B$ is of order n. What practical implications does this have on ways to store products of sparse matrices (Is it better so store the product AB or the pairs A, B separately? Consider both the computational cost for performing matrix-vector products and the cost of memory)

P-3.6 Consider a 6×6 matrix which has the pattern

$$A = \begin{pmatrix} \star & \star & & & \star & \\ \star & \star & \star & & & \star \\ & \star & \star & & & \\ & & & \star & \star & \\ \star & & & \star & \star & \star \\ & \star & & & \star & \star \end{pmatrix}.$$

a. Show the adjacency graph of A.

b. Consider the permutation $\pi = \{1, 3, 4, 2, 5, 6\}$. Show the adjacency graph and new pattern for the matrix obtained from a symmetric permutation of A based on the permutation array π.

P-3.7 You are given an 8 matrix which has the following pattern:

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

a. Show the adjacency graph of A ;

b. Find the Cuthill Mc Kee ordering for the matrix (break ties by giving priority to the node with lowest index). Show the graph of the matrix permuted according to the Cuthill-Mc Kee ordering.

c. What is the Reverse Cuthill Mc Kee ordering for this case? Show the matrix reordered according to the reverse Cuthill Mc Kee ordering.

d. Find a multicoloring of the graph using the greedy multicolor algorithm. What is the minimum number of colors required for multicoloring the graph?

e. Consider the variation of the Cuthill Mc-Kee ordering in which the first level consists L_0 several vertices instead on only one vertex. Find the Cuthill Mc Kee ordering with this variant with the starting level $L_0 = \{1, 8\}$.

P-3.8 Consider a matrix which has the pattern

$$A = \begin{pmatrix} \star & \star & & & \star & & & \star \\ \star & \star & \star & & & \star & & \\ & \star & \star & \star & & & \star & \\ & & \star & \star & \star & & & \star \\ \star & & & \star & \star & \star & & \\ & \star & & & \star & \star & \star & \\ & & \star & & & \star & \star & \star \\ \star & & & \star & & & \star & \star \end{pmatrix}.$$

a. Show the adjacency graph of A. (Place the 8 vertices on a circle.)

b. Consider the permutation $\pi = \{1, 3, 5, 7, 2, 4, 6, 8\}$. Show the adjacency graph and new pattern for the matrix obtained from a symmetric permutation of A based on the permutation array π.

c. Show the adjacency graph and new pattern for the matrix obtained from a reverse Cuthill-McKee ordering of A starting with the node 1. (Assume the vertices adjacent to a given vertex are always listed in increasing order in the data structure that describes the graph.)

d. Find a multicolor ordering for A (give the vertex labels color 1, followed by those for color 2, etc.).

P-3.9 Given a five-point finite difference graph, show that the greedy algorithm will always find a coloring of the graph with two colors.

P-3.10 Prove that the total number of colors found by the greedy multicoloring algorithm does not exceed $\nu_{max} + 1$, where ν_{max} is the maximum degree of all the vertices of a graph (not counting the cycles (i, i) associated with diagonal elements).

P-3.11 Consider a graph that is bipartite, i.e., 2-colorable. Assume that the vertices of the graph are colored by a variant of Algorithm (3.6), in which the nodes are traversed in a certain order $i_1, i_2, \ldots, i_n$.

a. Is it true that for any permutation $i_1, \ldots, i_n$ the number of colors found will be two?

b. Consider now a permutation satisfying the following property: for each j at least one of the nodes $i_1, i_2, \ldots, i_{j-1}$ is adjacent to i_j. Show that the algorithm will find a 2-coloring of the graph.

c. Among the following traversals indicate which ones satisfy the property of the previous question: (1) Breadth-First Search, (2) random traversal, (3) traversal defined by i_j = any node adjacent to i_{j-1}.

P-3.12 Given a matrix that is irreducible and with a symmetric pattern, show that its structural inverse is dense. Structural inverse means the pattern of the inverse, regardless of the values, or otherwise stated, is the union of all patterns of the inverses for all possible values. [Hint: Use Cayley Hamilton's theorem and a well known result on powers of adjacency matrices mentioned at the end of Section 3.2.1.]

P-3.13 The most economical storage scheme in terms of memory usage is the following variation on the coordinate format: Store all nonzero values a_{ij} in a real array $AA[1 : Nz]$ and the corresponding "linear array address" $(i-1)*n+j$ in an integer array $JA[1 : Nz]$. The order in which these corresponding entries are stored is unimportant as long as they are both in the same position in their respective arrays. What are the advantages and disadvantages of

this data structure? Write a short routine for performing a matrix-by-vector product in this format.

P-3.14 Write a FORTRAN-90 or C code segment to perform the matrix-by-vector product for matrices stored in Ellpack-Itpack format.

P-3.15 Write a small subroutine to perform the following operations on a sparse matrix in coordinate format, diagonal format, and CSR format:

a. Count the number of nonzero elements in the main diagonal;

b. Extract the diagonal whose offset is k;

c. Add a nonzero element in position (i, j) of the matrix (this position may initially contain a zero or a nonzero element);

d. Add a given diagonal to the matrix. What is the most convenient storage scheme for each of these operations?

P-3.16 Linked lists is another popular scheme often used for storing sparse matrices. These allow to link together k data items (e.g., elements of a given row) in a large linear array. A starting position is given in the array which contains the first element of the set. Then, a link to the next element in the array is provided from a LINK array.

a. Show how to implement this scheme. A linked list is to be used for each row.

b. What are the main advantages and disadvantages of linked lists?

c. Write an algorithm to perform a matrix-by-vector product in this format.

NOTES AND REFERENCES. Two good references on sparse matrix computations are the book by George and Liu [144] and the more recent volume by Duff, Erisman, and Reid [107]. These are geared toward direct solution methods and the first specializes in symmetric positive definite problems. Also of interest are [221] and [227] and the early survey by Duff [106].

Sparse matrix techniques have traditionally been associated with direct solution methods. This has changed in the last decade because of the increased need to solve three-dimensional problems. The SPARSKIT library, a package for sparse matrix computations [245] is available from the author at: `http://www.cs.umn.edu/ saad/software` .
Another available software package which emphasizes object-oriented design with the goal of hiding complex data structures from users is PETSc [24].

The idea of the greedy multicoloring algorithm is known in Finite Element techniques (to color elements); see, e.g., Benantar and Flaherty [31]. Wu [319] presents the greedy algorithm for multicoloring vertices and uses it for SOR type iterations, see also [248]. The effect of multicoloring has been extensively studied by Adams [2, 3] and Poole and Ortega [228]. Interesting results regarding multicoloring in the context of finite elements based on quad-tree structures have been obtained by Benantar and Flaherty [31] who show, in particular, that with this structure a maximum of six colors is required.
■