University of Maryland College Park

Institute for Advanced Computer Studies TR–2003–95
Department of Computer Science TR–4527

Building an Old-Fashioned Sparse Solver*

G. W. Stewart†

August 2003

ABSTRACT

A sparse matrix is a matrix with very few nonzero elements. Many applications in diverse fields give rise to linear systems of the form $Ax = b$, where A is sparse. The problem in solving these systems is to take advantage of the preponderance of zero elements to reduce both memory use and computation time. The purpose of this paper is to introduce students (and perhaps their teachers) to sparse matrix technology. It is impossible to treat all the techniques developed since the subject started in the 1960's. Instead, this paper constructs a sparse solver for positive definite systems that would have been state of the art around 1980, emphasizing equally theory and computational practice. It is hoped that a mastery of this material will allow the reader to study the subject independently.

*This report is available by anonymous ftp from thales.cs.umd.edu in the directory pub/reports or on the web at http://www.cs.umd.edu/∼stewart/.

†Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742 (stewart@cs.umd.edu).

Building an Old-Fashioned Sparse Solver

G. W. Stewart

ABSTRACT

A sparse matrix is a matrix with very few nonzero elements. Many applications in diverse fields give rise to linear systems of the form $Ax = b$, where A is sparse. The problem in solving these systems is to take advantage of the preponderance of zero elements to reduce both memory use and comutation time. The purpose of this paper is to introduce students (and perhaps their teachers) to sparse matrix technology. It is impossible to treat all the techniques developed since the subject started in the 1960's. Instead, this paper constructs a sparse solver for positive definite systems that would have been state of the art around 1980, emphasizing equally theory and computational practice. It is hoped that a mastery of this material will allow the reader to study the subject independently.

1. Introduction

A matrix A of order n is said to be sparse if it has a very small number of nonzero elements. In this paper we will be concerned with solving sparse linear systems of the form

$$Ax = b. \tag{1.1}$$

Sparse systems arise in many connections — fluid dynamics, structural engineering, linear programming, economic models, electrical circuits, just to name just a few. The car you drive was designed in part by solving large sparse systems.

Sparsity is a desirable property. If A is dense — if most of its elements are nonzero — then solving the system requires work proportional to n^3. On my PC, I can solve a system of order one thousand in about eight seconds. The n-cubed law says that it would take over two hours to solve a system of order ten thousand, and ten thousand is not especially large in many applications. Sparsity represents a hope of getting out of the n-cubed trap by taking advantage of the large number of zero elements.

Storage is also a problem. The memory needed to store a dense matrix increases as the square of its order. To store a dense matrix of order one thousand requires a million words of eight-byte floating-point words or about eight megabytes — well within the range of a garden variety PC. On the other hand a matrix of order ten thousand requires eight hundred megabytes, which is not found on your typical PC or workstation. But if the same matrix has only, say, ten elements in a row, the storage requirement for the nonzero elements is eight hundred kilobytes. The savings are obvious, although as we shall see later bookkeeping overhead raises the storage count somewhat.

We will be concerned with direct as opposed to iterative methods for solving (1.1). Most direct sparse matrix solvers are based on some variant of Gaussian elimination. The broad outline is the same as for dense solvers. The matrix A in question is factored into the product

$$A = LU, \tag{1.2}$$

where L is lower triangular and U is upper triangular. The system $Ax = b$ can then be solved in two stages.

$$\begin{array}{ll} 1. & \text{Solve } Ly = b \\ 2. & \text{Solve } Ux = y \end{array} \tag{1.3}$$

Since L and U are triangular, the systems in (1.3) can be solved efficiently by standard algorithms that require no further reduction of L or U.

The purpose of this paper is to provide a look at the technology people use to take advantage of sparsity. We will do this by building a sparse solver for symmetric positive definite systems. Now the sparse matrix ball started rolling in the 1960s, and it is impossible to include all the sophisticated techniques developed over the past thirty five years in a single expository paper. Instead we will build an old-fashioned solver — one that would have been considered state-of-the-art around 1980. The situation is analogous to describing a car of the late 1920s, cars that first exhibited the standard features of today's cars. Cars and solvers have come a long way from their beginnings. They've been streamlined and supercharged. Their infrastructures — the roads and computers they run on — have improved immeasurably. But in a modern car there is the soul of a Model-A Ford. Likewise, the heart of most sparse solvers is a relative of the old-fashioned solver we describe here.

This paper is organized as follows. In the next section we will discuss generalities about sparse matrices and solvers and introduce a specific class of sparse matrices to use as an example later. In Section 3, we will treat the Cholesky algorithm, a variant of Gaussian elimination that factors a symmetric positive definite matrix A into a product LL^{T}, where L is lower triangular (L is called the *Cholesky factor* of A). We will also show how this algorithm generates *fill-in* — nonzero elements in L where A had zero elements. In Section 4 we will describe a useful pictorial technique for tracking fill-in as it occurs. In Section 5, we will introduce the data structure we will use to represent a sparse matrix and illustrate its manipulation by two algorithms. The next two sections are the mathematical heart of the paper. Section 6 gives a brief treatment of the graph theory required to derive our algorithm. Section 7 is devoted to the definition and properties of a particular graph, called the elimination tree, that is the basis of our subsequent algorithms. In Section 8, we will show how to compute the structure of the Cholesky factor L so that we can set up a data structure to hold it. This process, called symbolic factorization, is followed by the actual numerical factorization, which is treated in Section 9. Finally, in Section 10 we briefly discuss modern additions to our sparse solver. The paper concludes with some bibliographic notes.

In the course of the exposition, we will present several algorithms. They will be written in a pseudocode that should be readable by anyone reasonably familiar with one of the standard high level languages; e.g., Fortran 95 or C. It is a hodge-podge that tilts toward Fortran. In particular, all arrays begin indexing at one, and subprogram parameters are passed by reference, so that modifications made to them in the subprogram are passed back to the calling program.

A word on notation. Matrices will be written with upper-case letters and vectors with lower-case letters. Scalars will be written with lower-case Roman or Greek letters. In particular, elements of A, L, and U will always be denoted by the Greek letters α, λ, and υ.

The reader is assumed to be familiar with the basics of matrix computations: the use of partitioned matrices to derive algorithms and especially Gaussian elimination. From this standard background, however, the paper quickly moves into uncharted territory, and the reader should be prepared to pore over passages until understanding comes. A note pad and a pencil equipped with a good eraser are essential tools. I hope that when the reader comes to the end of the journey he or she will feel that the effort was well spent.

2. Sparse Matrices and Sparse Solvers

In this section we will consider generalities about sparse matrices and sparse solvers. We begin with a discussion of what constitutes a sparse matrix.

2.1. Sparse matrices

The notion of a *sparse matrix* is one of those concepts that is most useful if it is not pinned down too tightly. The reason is that matrices come in so many varieties that the attempt to give a formal definition of sparseness is likely to exclude matrices that someone would naturally consider sparse. Nonetheless, there are some guidelines.

First, the number of nonzero elements must be small enough. Most people would not consider a triangular matrix sparse, since only about half its elements are zero. When the positions of the nonzero elements of a matrix — its *structure* we call it — depends on its order, the matrix is commonly called sparse if the number of nonzero elements is $O(n)$. But many matrices cannot be treated this way — models of electric circuits fall in this category. In that case the most useful definition is operational: a matrix is sparse if its manipulation can benefit from sparse technology.

Second, many people would exclude matrices that can be treated by minor extensions of dense matrix technology. For example, a *tridiagonal matrix* is one whose nonzero elements lie only on the diagonal, the superdiagonal, and the subdiagonal of the matrix. It is about as sparse as you can get. But a tridiagonal system can be solved by an obvious variant of Gaussian elimination that simply ignores the zero elements. Band matrices, where nonzero elements cluster in a band about the diagonal, constitute an intermediate case. If the band is dense, then a variant of ordinary Gaussian elimination applies. (It is significant that algorithms for dense band matrices are generally found in dense matrix packages such as LAPACK.) On the other hand, if the band is sparse, it may pay to use an appropriate sparse solver.

The discussion in the last paragraph suggests that the structure of the matrix plays an important role in the construction of a sparse solver. For example, symmetric positive definite matrices have special properties that distinguish them from general nonsymmetric matrices, and the best solver for one is not suitable for the other. Thus sparse matrices fall into classes that require different algorithms. However, each class occurs frequently enough in applications to justify the design and implementation of a general algorithm for the class in question. In this paper, as we said earlier, we will be concerned with a general sparse solver for symmetric positive definite systems.

2.2. Sparse solvers

Gaussian elimination is at once the simplest and most complicated of algorithms. It is so simple that it can be taught to undergraduates — even high schoolers. But it is so flexible that it yields many different algorithms that are not obviously related. This is as true of dense systems as sparse ones, although the useful variants are not necessarily the same for each category. Thus the first task in the design of a sparse solver is to choose an appropriate form of Gaussian elimination.

Having decided on a variant of Gaussian elimination, the designer of a sparse matrix solver faces some additional decisions.

1. How can the matrix A be represented so that only nonzero elements are stored?

2. The process of computing L and U from A will generate additional nonzero elements, called *fill-in*. Fill-in creates two problems.

    1. Interchanging rows and columns of A and the corresponding components of b, simply interchanges the same components of x, so that solution is essentially undisturbed. But it also affects the course of Gaussian elimination and hence the amount of fill-in. It is therefore natural to ask if we can *order* A to reduce fill-in.

    2. An efficient algorithm will need to know in advance where fill-in occurs so that it can allocate storage and set up data structures for L and U. This process is called *symbolic factorization* (or *analysis*).

3. Once the symbolic factorization has been accomplished, one must perform the actual *numerical factorization.* In general, the algorithms for symbolic and numerical factorization are quite different, with the symbolic factorization being the cheaper and, paradoxically, the more complicated.

4. Finally, one must solve the triangular systems in (1.3). Since L and U are not represented as arrays, the sparse algorithms are different from their textbook counterparts.

Thus a typical sparse solver proceeds through stages of ordering, symbolic factorization, numerical factorization, and triangular solution. It should be stressed that these stages are often combined or omitted. But you will not go far wrong in understanding a sparse solver if you ask if and how it implements each of the above steps.

Ordering is something of an exception in the above list. More than the other steps, it depends on the details of the application generating the matrix. For example, problems associated with two-dimensional manifolds generate matrices for which a good ordering can often be found by a process called nested dissection. Because of the specificity of ordering algorithms, we will not treat them in this paper.

2.3. Grid-graph matrices

In this subsection we will introduce a class of sparse matrices associated with certain elliptic partial differential equations defined on a square, say on the interval $\Omega = [0,1]\times[0,1]$. Without going into details, the problem is turned into a matrix problem by placing an $(N+1)\times(N+1)$ grid on the square as shown in Figure 2.1. Each interior grid point (j,k) is associated with an approximation to the solution u_{jk} at that point. (Note that the indexing is not the same as for a matrix: the j is the column index, k is the row index, and indexing starts from the southwest corner.) From the differential equation we can derive a linear relation that involves u_{jk} and its neighboring approximations $u_{j-1,k}$, $u_{j+1,k}$, $u_{j,k-1}$ and $u_{j,k+1}$. Thus we have $n = N^2$ linear equations in n unknows, which can be solved for the u_{jk}. The corresponding matrix is sparse because each row involves only the unknown U_{jk} and its four neighbors.

The matrix is also structured. If we order the unknowns u_{jk} rowwise thus

$$u_{11}, u_{21}, \ldots, u_{N1}, u_{12}, u_{22}, \ldots, u_{N2}, \ldots, \tag{2.1}$$

then the matrix of the system has the form

$$A = \begin{pmatrix} T_1 & D_1 & & & \\ D_1 & T_2 & D_2 & & \\ & \ddots & \ddots & \ddots & \\ & & D_{N-2} & T_{N-1} & D_{N-1} \\ & & & D_{N-1} & T_N \end{pmatrix},$$

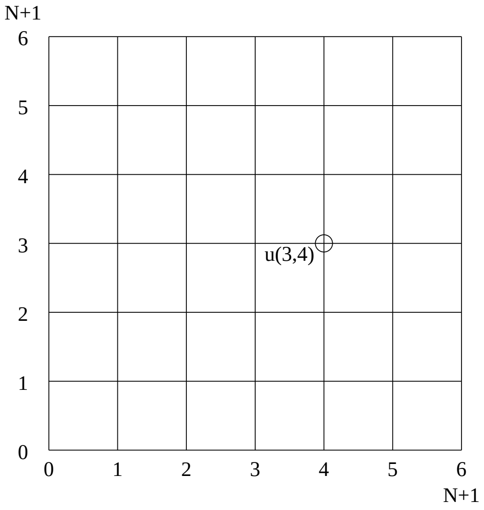


Figure 2.1: A Discretization Grid

where the T_i are tridiagonal of order N and the D_i are diagonal. This matrix, which we will call a *grid-graph matrix,* can be shown to be symmetric positive definite. It is also a band matrix with a sparse band and therefore a candidate for a sparse solver. We have introduced it because we know a great deal about its Cholesky factorization. Specifically:

> *For n large, there is an ordering of A (called a nested dissection ordering) such that A can be factored in approximately*
>
> $$10n^{\frac{3}{2}} \textit{ floating-point additions and multiplications.}$$
>
> *Moreover, the Cholesky factor L of A has approximately* (2.2)
>
> $$4n \log_2 n \textit{ nonzero elements.}$$
>
> *Up to order constants these results are optimal.*

We will use these results later in assessing the effects of overhead in our algorithms.

The result (2.2) shows the importance of ordering in sparse factorization. For the natural ordering (2.1) the band of the Cholesky factor is essentially full and has about $n\sqrt{n}$ nonzero elements. Thus the ratio of nonzeros of the natural ordering to the nested dissection ordering is $\frac{\sqrt{n}}{4\log_2 n}$. When $n = 90,000$, corresponding to a 300×300 grid, the natural ordering requires about 4.5 times the storage as the nested dissection ordering.

3. The Cholesky Decomposition and Fill-in

Our sparse solve will be based on the Cholesky algorithm, a variant of Gaussian elimination that factors a symmetric positive definite matrix into the product LL^{T} of a lower triangular matrix and its transpose. In this section we will first sketch a proof the existence of such a factorization. We will then describe the variant of Cholesky's algorithm that we will use here. The remainder of the section is devoted to a discussion of fill-in.

3.1. Existence

It might be expected that the LU decomposition $A = LU$ of a nonsingular, symmetric matrix should itself be symmetric; i.e., that we can write it in the form

$$A = LL^{\mathrm{T}}, \tag{3.1}$$

where L is lower triangular. Unfortunately, this is not always the case. For suppose that x is nonzero. Then because L is nonsingular, $y = L^{\mathrm{T}}x \neq 0$. It follows that

$$x^{\mathrm{T}}Ax = x^{\mathrm{T}}LL^{\mathrm{T}}x = y^{\mathrm{T}}y = \textstyle\sum_i y_i^2 > 0.$$

Thus only matrices satisfying

$$x \neq 0 \implies x^{\mathrm{T}}Ax > 0 \tag{3.2}$$

can have a *Cholesky decomposition* of the form (3.1). We call any such matrix a *symmetric positive definite matrix.*

Being symmetric positive definite is not only necessary for a matrix to have a Cholesky decomposition, it is also sufficient.

> *Let A be symmetric positive definite. Then there is a unique lower triangular matrix L with positive diagonal elements such that $A = LL^{\mathrm{T}}$.*

The various proofs of this result lead to variants of Gaussian elimination. For example, one proof begins by partitioning the factorization $A = LL^{\mathrm{T}}$ in the form

$$\begin{pmatrix} \alpha & a^{\mathrm{T}} \\ a & \hat{A} \end{pmatrix} = \begin{pmatrix} \lambda & 0 \\ \ell & \hat{L} \end{pmatrix} \begin{pmatrix} \lambda & \ell^{\mathrm{T}} \\ 0 & \hat{L}^{\mathrm{T}} \end{pmatrix}.$$

Then computing the $(1,1)$-element of the partition, we find that $\alpha = \lambda^2$, so that that $\lambda = \sqrt{\alpha}$. Thus we have computed the $(1,1)$-element of L. Similarly by computing the $(2,1)$-block of the partition, we get $\ell = \lambda^{-1}a$. Finally from the $(2,2)$-block of the partition we find that

$$\hat{L}\hat{L}^{\mathrm{T}} = \hat{A} - \ell\ell^{\mathrm{T}} \equiv S,$$

so that $\hat{L}$ is the Cholesky factor of S (which is called the Schur complement of α). The matrix S can be shown to be positive definite (that is the tricky part), so that $\hat{L}$ exists by an obvious induction.

This proof leads naturally to an algorithm in which the first column of L is computed, the matrix S is formed, and the process is repeated recursively on S. This algorithm, which corresponds to classical Gaussian elimination, is widely used in sparse solvers. However we will base our solver on an algorithm that builds up L column by column. This algorithm is also widely used, and for our purposes it has the advantage that it provides insight into the dynamics of fill-in.

3.2. A columnwise algorithm

The columnwise algorithm can be derived as follows. Suppose we have computed the first $k-1$ columns of L and wish to compute the kth. Consider the partitioned decomposition

$$\begin{pmatrix} A_{11} & a_{21} & A_{31}^{\mathrm{T}} \\ a_{21}^{\mathrm{T}} & \alpha_{22} & a_{32}^{\mathrm{T}} \\ A_{31} & a_{32} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 & 0 \\ \ell_{21}^{\mathrm{T}} & \lambda_{22} & 0 \\ L_{31} & \ell_{32} & L_{33} \end{pmatrix} \begin{pmatrix} L_{11}^{\mathrm{T}} & \ell_{21} & L_{31}^{\mathrm{T}} \\ 0 & \lambda_{22} & \ell_{32}^{\mathrm{T}} \\ 0 & 0 & L_{33}^{\mathrm{T}} \end{pmatrix},$$

in which A_{11} is of order $k-1$. Computing the kth column of this partition, we find that

$$\begin{pmatrix} a_{21} \\ \alpha_{22} \\ a_{32} \end{pmatrix} = \begin{pmatrix} L_{11}\ell_{21} \\ \ell_{21}^{\mathrm{T}}\ell_{21} + \lambda_{22}^2 \\ L_{31}\ell_{21} + \lambda_{22\ell_{32}} \end{pmatrix}$$

From this we see that

$$\lambda_{22} = \sqrt{\alpha_{22} - \ell_{21}^{\mathrm{T}}\ell_{21}} \quad \text{and} \quad \ell_{32} = \lambda_{22}^{-1}(a_{32} - L_{31}\ell_{21}), \tag{3.3}$$

which gives the kth column of L.

Algorithm 3.1 implements this columnwise scheme. Here we use colon notation to designate a range. For example, `L[k:n,k]` represents the vector formed from the elements `k, k+1,..., n` of column `k` of `L`. We have also used the convention that inconsistent loops are not executed; e.g., the loop in statement 4 when `k` is equal to one. Finally, we have computed the quantities $\alpha_{22} - \ell_{21}^{\mathrm{T}}\ell_{21}$ and $a_{32} - L_{31}\ell_{21}$ in (3.3) together in the loop on `j`, and then adjusted them in statements 7 and 8.

This algorithm and the classical variant sketched above are numerically stable. The computed Cholesky factor satisfies $L^{\mathrm{T}}L = A+E$, where E of of the order of the rounding unit compared with A. It is worth noting that in the nonsymmetric case one must pivot (i.e., interchange rows and columns of A) to achieve similar stability. Thus an important distinction between symmetric positive definite and nonsymmetric sparse solvers is that the former can reorder solely to minimize fill-in, whereas the latter must balance fill-in and numerical stability in its ordering schemes.

`Colchol` computes the Cholesky factor of the symmetric positive definite matrix A.

```
 1. Colchol(A, L)
 2.    Move the lower half of A to L
 3.    for k=1 to n
 4.       for j=1 to k-1
 5.          L[k:n,k] = L[k:n,k] - L[k,j]*L[k:n,j]
 6.       end for j
 7.       L[k,k] = sqrt(L[k,k])
 8.       L[k+1:n,k]  = L[k+1:n,k]/L[k,k]
 9.    end for k
10. end Colchol
```

Algorithm 3.1: The columnwise Cholesky algorithm

3.3. The columnwise algorithm and fill-in

The algorithm `Colchol` allows us to understand how fill-in occurs in the Cholesky factor of a sparse matrix. To see this, let us rewrite the loop 4 in a form that better reflects the realities of sparse computation — namely, that only some of the columns of L are actually accumulated inside the loop.

```
1.    for j=1,k-1
2.       if (L[k,j] .ne.  0)
3.          L[k:n,k] = L[k:n,k] - L[k,j]*L[k:n,j]            (3.4)
4.       end if
5.    end for j
```

Recall that at the outset `L[k:n,k]` is initialized to `A[k:n,k]`. Except for some final adjustment — corresponding to statements 7 and 8 in Algorithm 3.1 — the vector `L[k:n,k]` is computed by subtracting multiples of a subset of the truncated columns `L[k:n,j]` of L. This subset is precisely those columns for which `L[k,j]` is nonzero. Thus the columns we subtract are determined by the nonzero structure of the row `L[k,1:k-1]` of L.

The loop shows how fill-in occurs in sparse elimination. Suppose in statement 3 the truncated column `L[k:n,j]` has a nonzero entry `L[i,j]` and `A[i,j]` is zero. Then the calculation will put a nonzero element in `L[i,j]`; i.e., the originally zero element `A[i,j]` will be filled in by the elimination process. Actually, we must be a little careful here. There is always the possibility that fortuitous cancellation will produce a zero element where a nonzero is expected. However, this situation is unstable — a small change in an appropriate element of A will cause the nonzero to reappear. Consequently, we will ignore this possibility and assume that fill-in occurs wherever our formulas lead us to expect it.

To examine the properties of fill-in more carefully, it will be convenient to drop our programming notation. Let $a_k^{(k)}$ be the truncated kth column of A beginning with α_{kk} — i.e., `A[k:n,k]` — and let $\ell_j^{(k)}$ be the truncated jth column of L beginning with λ_{kj} — i.e., `L[k{:}n,j]`. Then the fragment (3.4) computes the vector

$$\hat{\ell}_k^{(k)} = a_k^{(k)} - \sum_{\substack{j=1 \\ \lambda_{kj} \neq 0}}^{k-1} \lambda_{kj} \ell_j^{(k)}. \tag{3.5}$$

which has the same pattern of nonzero elements as $\ell_k^{(k)}$, since it differs from $\ell_k^{(k)}$ only by a nonzero scaling factor. From the discussion above, it follows that the pattern of nonzeros in $\ell_k^{(k)}$ is composed of the pattern of $a_k^{(k)}$ and the patterns of the truncated columns of L that begin with a nonzero component.

We can write this fact more succinctly by introducing some notation. Define the *structure* of $\ell_j^{(k)}$ to be

$$\operatorname{str}(\ell_j^{(k)}) = \{i \geq k \colon \lambda_{ij} \neq 0\}.$$

In other words the structure of $\ell_j^{(k)}$ is the set of all row indices i for which the corresponding component is nonzero. Define the structure of the truncated columns of A analogously. Then

$$\operatorname{str}(\ell_k^{(k)}) = \operatorname{str}(a_k^{(k)}) \cup \bigcup_{\substack{j=1 \\ \lambda_{kj} \neq 0}}^{k-1} \operatorname{str}(\ell_j^{(k)}). \tag{3.6}$$

In Section 2 we used the term *symbolic factorization* to refer to the process of determining the structure of L so that we could set up a data structure to hold it. Equation (3.6) would seem to provide a way of performing symbolic elimination, since it furnishes the wherewithal to determine the structure of successive columns of L in terms of their predecessors. Unfortunately, this procedure mimics Gaussian elimination too closely. To determine the structure of the current column we need the structures of the preceding columns — and storing and manipulating those structures is no easier than storing and manipulating the elements of L itself.

A cure for this problem is to recognize the fact that it may not be necessary to work with all columns for which $\lambda_{kj} \neq 0$. As an extreme example, suppose that column $\ell_k^{(k)}$ fills in completely. Then every subsequent column also fills in, and the computation in (3.6) becomes unnecessary. In practice, it turns out that as the elimination progresses, only a few of the columns in (3.6) are needed to determine the structure of the kth column of L. The problem is to determine which columns are needed — or equivalently which columns to prune from the union in (3.6). The answer is provided by an auxiliary structure called the elimination tree, which will be introduced in Section 7.

3.4. Alpha-precursors

Fill-in cannot occur unless there is something to do the filling in. If, for example, the elements $\alpha_{i1}, \ldots, \alpha_{ik}$ $(i > k)$ are all zero, it follows by an induction on (3.5) that the corresponding elements of L are also zero. Consequently, an element $\lambda_{ik} \neq 0$ of L for which $\alpha_{ik} = 0$ must depend on some nonzero element of A proceeding it in row i. We call such elements *α-precursors* of λ_{ik}. Since α-precursors will prove important later, we will now show how to construct them.

Suppose that $\lambda_{ik} \neq 0$—i.e., i belongs to $\mathrm{str}(\ell_k^{(k)})$. Now if $\alpha_{ik} \neq 0$, then α_{ik} itself is an α-precursor of λ_{ik}. If not, from (3.5) we have

$$\hat{\lambda}_{ik} = -\sum_{\substack{j=1 \\ \lambda_{kj} \neq 0}}^{k-1} \lambda_{kj}\lambda_{ij}.$$

Since there must be at least one nonzero term in this sum, there is a $k_1 < k$ such that $\lambda_{ik_1} \neq 0$ and $\lambda_{kk_1} \neq 0$. Now if $a_{ik_1} \neq 0$, it is an α-precursor. Otherwise we have,

$$\hat{\lambda}_{ik_1} = -\sum_{\substack{j=1 \\ \lambda_{k_1 j} \neq 0}}^{k_1-1} \lambda_{k_1 j}\lambda_{ij}.$$

Thus there is a $k_2 < k_1$ such that $\lambda_{ik_2} \neq 0$ and $\lambda_{k_2 k_1} \neq 0$. If $\alpha_{ik_2} \neq 0$ we have our α-precursor. Otherwise we continue backtracking as above. The result is a decreasing sequence of indices $k > k_1 > k_2 > \cdots$ such that $\hat{\lambda}_{ik_r} \neq 0$ $(r = 0, 1, \ldots)$. The sequence either terminates with a nonzero element of A or with $\hat{\lambda}_{i1} \neq 0$, in which case $\alpha_{i1} = \hat{\lambda}_{i1} \neq 0$ is an α-precursor.

We have arrived at the following result.

> *If i lies in the structure of the kth column of L, then λ_{ik} has an α-precursor $\alpha_{iq} \neq 0$. Specifically, there is a sequence of indices $q = k_p < k_{p-1} < \ldots, < k_1 < k_0 = k$ such that* (3.7)
>
> 1. $\alpha_{i,k_p} \neq 0$,
> 2. $\lambda_{k_r,k_{r+1}} \neq 0, \quad r = p-1, \ldots, 1.$

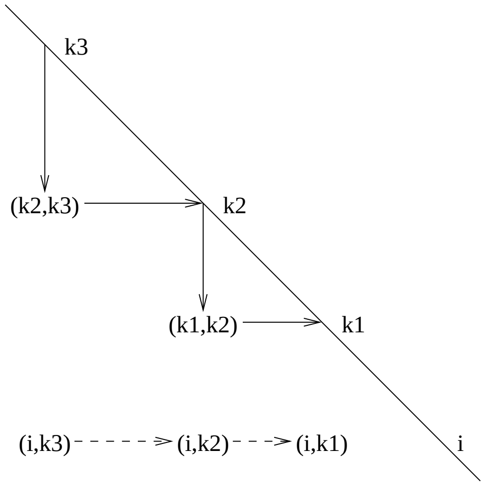


Figure 4.1: Reflection diagram for (3.7)

4. Reflection diagrams

In view of our construction of α_{iq}, it would seem that statement 1 in (3.7) should read $\alpha_{i,k_p} \neq 0$ *and* $\lambda_{i,k_r} \neq 0$ $(r = p-1, \ldots, 1)$. However, this extended statement is implied by the original. To see this we will introduce a pictorial method for tracking fill-in as one moves around in a matrix. The diagram in Figure 4.1 represents (3.7) for the case $p = 3$. It is to be understood as follows. The diagram represents a grid corresponding to the elements in the lower half of a matrix, although we do not explicitly draw the grid. Points on the diagonal are identified with diagonal elements; e.g., `k3`, `k2`, `k1`, and `i` in the diagram. Points in the interior represent subdiagonal elements of the matrix. The row index of the element is the index of the diagonal to the east of it; the column index is the index of the diagonal north of it — e.g. `(k2,k3)` in the above diagram. Thus each pair of distinct diagonal elements subsumes a unique element of the matrix.

One is permitted to pass between two diagonals provided the element they subsume is nonzero. Such a transition is always shown as proceeding through the subsumed element, as in the path from `k3` to `k2`. The diagram is called a reflection diagram because if we place a mirror pointing northeast at, say, `(k2,k3)` a southward beam of light beginning at `k3` will be reflected to `k2`.

The connection with fill-in is illustrated by the element `(i,k3)` in the figure. Here we suppose that the underlying matrix is the Cholesky factor L. Because the element `(k2,k3)` is nonzero, (3.6) implies that the structure of `L(k2:n,k3)` is contained in the structure of `L(k2:n,k2)`. In particular, if element `(i,k3)` is nonzero, a fill-in must occur at `(i,k2)`, as shown by the dashed arrow. Similarly, the passage from `k2` to `k1` reveals a fill-in at `(i,k1)`. Thus the connection shows that only the statement $\alpha_{ik_3} \neq 0$ (which implies $\lambda_{ik_3} \neq 0$) is necessary in (3.7).

One can also move backward in a reflection diagram — and we will later on — but backward movement does not reveal fill-in. It is an instructive exercise to figure out why.

5. Representing and manipulating sparse matrices

Having decided on a numerical algorithm, we must now decide how to represent a sparse matrix. The conventional representation as a square array of numbers is untennable. For example, a grid-graph matrix has roughly $5n$ nonzero elements, whereas the conventional representation would take n^2 words of memory — the overwhelming majority of them zero. In this section we introduce a structure that only stores the nonzero elements of a matrix, but at the cost of some additional bookkeeping arrays. We will then illustrate the use of this structure by developing two algorithms, one for computing a matrix vector product and the other for traversing a matrix by rows.

5.1. A data structure

To represent a sparse matrix by storing only its nonzero elements we must provide additional information that enables us to determine where an element lies. For example, we could represent a sparse matrix as a collection of triplets

$$(\mathtt{val}, \mathtt{rx}, \mathtt{cx})$$

where `val` represents the value of the element and `rx` and `cx` are its row and column indices. In other words if the matrix in question is `A`, then $\mathtt{a}_{\mathtt{rx},\mathtt{cx}} = \mathtt{val}$.

This *coordinate representation* is simple and natural. It has the advantage that it makes it very easy for a user to generate a sparse matrix on a computer. For example, the matrix could be represented by a structure of the form

```
coordmat structure
   int  nrow     ! Number of rows
   int  ncol     ! Number of columns
   int  nnz      ! Number of nonzero elements
   int  rx[]     ! Array of row indices
   int  cx[]     ! Array of column indices
   real val[]    ! Array of values
end structure
```

(5.1)

Then to initialize a sparse matrix, the user could write a program like the following.

```
coordmat A
A.nrow = number of rows
A.ncol = number of columns
A.nnz  = number nonzero elements
for k = 1 to A.nnz
   generate i, j, and aij
   A.rx[k] = i; A.cx[k] = j
   A.val[k] = aij
end for k
```

Unfortunately, the representation (5.1) is good for little other than entering a sparse matrix. For example, there is no convenient way to pass along a row or down a column of a sparse matrix so represented. This illustrates an important point about the representation of sparse matrices: the representation must not only be economical in storage, but it must allow the efficient implementation of whatever operations must be performed on the matrix. Since there are many conceivable operations that one might want to perform, we are left with the possibility that no one structure can serve to represent a sparse matrix in all capacities.

Fortunately, when it comes to solving symmetric positive definite systems we basically want to do two things: compute a Cholesky factorization of the matrix in question and solve sparse triangular systems involving the Cholesky factor. Although the former will require that a number of operations in addition to those of Gaussian elimination be performed on the matrix, it turns out that there is a representation for a symmetric positive definite matrix that permits all these operations to be performed with reasonable efficiency.

The idea behind the structure is to store the nonzero elements a linear array (`val`) in column-major order — that is, in a linear array with the nonzero elements of the first column in their natural order follow by those of the second column, and so on. Because of symmetry we need only store the entries of a column from the diagonal downward. A parallel array of integers (`rx`) gives the row index of each element. To distinguish the columns, we have another array (`colp`) pointing to the beginning of each column. We call this structure *packed column representation.*

```
define pcmat structure
   int  n          ! The order of the matrix
   int  nnz        ! Number of nonzero elements
   int  colp[]     ! Array of start of column pointers
   int  rx[]       ! Array of row indices
   real val[]      ! Array of off-diagonal values
end structure
```

To illustrate the structure consider the matrix

$$\begin{pmatrix} 4.6 & 0.0 & 1.3 & 0.0 & 0.0 & 2.5 \\ 0.0 & 6.4 & 1.7 & 0.0 & 0.0 & 3.9 \\ 1.3 & 1.7 & 7.3 & 2.1 & 0.0 & 3.1 \\ 0.0 & 0.0 & 2.1 & 6.9 & 2.8 & 0.0 \\ 0.0 & 0.0 & 0.0 & 2.8 & 4.7 & 0.0 \\ 2.5 & 3.9 & 3.1 & 0.0 & 0.0 & 9.9 \end{pmatrix} \tag{5.2}$$

The corresponding `pcmat` is

`n`	6													
`nnz`	13													
`colp`	1			4			7			10		12	13	14
`rx`	1	3	6	2	3	6	3	4	6	4	5	5	6	
`val`	4.6	1.3	2.5	6.4	1.7	3.9	7.3	2.1	3.1	6.9	2.8	4.7	9.9	

(5.3)

Note that `colp` has `n+1` entries, with the last pointing to the nonexistent entry `val[nnz+1]`. The reason is that it allows us to loop through the elements of a column of `A`. For example, the following fragment prints the lower half of `A` by columns.

```
for j=1 to A.n
   for ii=A.colp[j] to A.colp[j+1]-1
      print(A.rx[ii], j, A.val[ii])
   end for ii
end for j
```

(5.4)

When `k = n`, the loop correctly prints only the value `val[nnz]`.

At this point we must say something about memory management. Originally, sparse matrix code was written largely in Fortran 66, and later in Fortran 77. These languages had no mechanisms for allocating storage. Thus the user had to hard-wire the necessary storage into the main program and pass it to the various components of the solver through their argument lists. In order to do this, the user had to know or estimate the amount of memory needed. Nothing could be done about a bad guess but return with an error flag and let the user recompile the program with a larger amount of storage.

At present Fortran 95 and the C family of languages have methods for allocating storage, and memory management can be relegated to the sparse solver. However, to allocate memory, the solver needs to know how much is needed. This is particularly important in the symbolic factorization step, where we must know the nonzero count for L so that the arrays `rx` and `val` can be allocated. This problem will be treated in Section 7.

We must also say something about auxiliary arrays. In many of our algorithms we will need to allocate extra storage to hold intermediate quantities. Such allocations usually come in two sizes: arrays of length `n` and arrays of length `nnz`. Since memory is limited, any additional arrays will reduce the size of the problems we can solve; but clearly an array of length `n` will do less harm than an array of length `nnz`. Even here we must distinguish between arrays of length `A.nnz` for the original matrix and arrays of length `L.nnz` for the Cholesky factor. Owing to fill-in, the latter will be larger than the former; hence the allocation of an array of length `A.nzz` will have relatively less effect. In our solver, the only auxiliary arrays will be of length `n`.

5.2. Matrix-vector multiplication

We turn now to two examples of programs that use the `pcmat` data structure. The first example is matrix-vector multiplication. Although we will not actually use this algorithm here, it illustrates some important points about manipulating sparse matrices. Moreover, matrix-vector multiplication is important in its own right, especially in iterative methods for solving large linear systems.

To derive an algorithm we begin with the usual definition of the product $y = Ax$:

$$y_i = \textstyle\sum_j \alpha_{ij} x_j. \tag{5.5}$$

In a naive implementation of this formula, to compute y_i we must access the elements

$$\alpha_{i1}, \alpha_{i2}, \ldots, \alpha_{i,i-1}, \alpha_{ii}, \alpha_{i,i+1} \cdots \alpha_{in}$$

of A. But in a `pcmat`, we store only the lower half of A. Hence we must access the elements

$$\alpha_{i1}, \alpha_{i2}, \ldots, \alpha_{i,i-1} \tag{5.6}$$

followed by

$$\alpha_{ii}, \alpha_{i+1,i}, \ldots, \alpha_{ni}; \tag{5.7}$$

i.e., we must go across row i of A until we reach the diagonal and then down column i. In a `pcmat` the references in (5.7) are easy to do — see (5.4). But it is not clear how to implement the passage along a row required by (5.6).

A cure for this problem is to reinterpret the formula (5.5). If we start with $y = 0$ and if $\alpha_{ij} \neq 0$, we must update y by adding $\alpha_{ij} x_j$ to y_i. For $i \geq j$, we can do the updates by traversing the `pcmat` by columns. But whenever we encounter an nonzero element α_{ij} $(i > j)$ in the structure, by symmetry we also have the value of $\alpha_{ji} = \alpha_{ij}$. Thus at that time we can also update y_j by $\alpha_{ij} x_i$, which takes care of the case $i < j$.

Algorithm 5.1 implements this strategy. There are three comments to make about it.

- The diagonal elements must be treated specially, since they cause only one update.

- The inner loop of `Mvmult` illustrates a convention we will use in the remainder of the paper. A double index like `ii` will refer to a position in the arrays `rx` and `val`. The

Given a pcmat A and two vectors x and y, Mvprod computes y = A*x.

```
Mvmult(A, x, y)
   y = 0
   for j=1 to A.n
      y[j] = y[j] + x[j]*A.val[A.colp[j]]
      for ii=A.colp[j]+1 to A.colp[j+1]-1
         i =A.rx[ii]
         y[i] = y[i] + x[j]*A.val[ii]
         y[j] = y[j] + x[i]*A.val[ii]
      end for ii
   end for j
end Mvmult
```

Algorithm 5.1: Computation of $y = Ax$

corresponding single index like `i = rx[ii]` will refer to the row index of the element pointed to by `ii`.

- The number of floating-point additions and multiplications is about `nnz`. For a full matrix the count is about to n^2 which can be much greater than `nnz`. For example, for a grid-graph matrix the operation count is approximately $4n\log_2 n$ [see (2.2)]. The ratio of the dense count to the sparse count is $n/4\log_2 n$. When $n = 90{,}000$, this factor is about 1,367. It pays to take advantage of sparsity!

5.3. Traversing a `pcmat` by rows

The trick used in the matrix-vector multiply algorithm to avoid accessing a `pcmat` by rows serves its purpose well, but there are times when we actually need the elements of a row of a sparse matrix. This is not an easy task to perform efficiently. For example, consider the following code to output the values of the nonzero elements in the lower half of a `pcmat` in row order.

```
for k=1 to A.n
   for j=1 to k
      for ii=A.colp[j] to A.colp[j+1]-1
         i = A.rx[ii]
         if (i > k) leave ii; fi
         if (i = k) print(i, j, val[ii]); leave ii; fi
      end for ii
   end for j
end for k
```

(5.8)

To find the nonzero `(j,k)`-element, if it exists, the code searches down column `j`. This loop could be improved by storing the most recent values of `ii` (one for each column) and restarting the search from that value when `k` changes. But in point of fact, no amount of optimization could render this code acceptable. For the first two loops imply that the work is at least $O(n^2)$. For many sparse matrices this is much greater than the work required to factor the matrix. For example, the work required to factor the grid-graph matrix is $O(n^{\frac{3}{2}})$.

The problem with (5.8) is that it treats each row independently. It turns out that if we take a peek ahead each time we process an element in a row, we can accumulate enough information to traverse subsequent rows without searching. To see this consider the following Wilkinson diagram of lower half of the matrix (5.2):

$$\begin{pmatrix} \mathtt{X} & & & & & \\ \mathtt{0} & \mathtt{X} & & & & \\ \mathtt{X} & \mathtt{X} & \mathtt{X} & & & \\ \mathtt{0} & \mathtt{0} & \mathtt{X} & \mathtt{X} & & \\ \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{X} & \mathtt{X} & \\ \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{0} & \mathtt{0} & \mathtt{X} \end{pmatrix} \tag{5.9}$$

Here an `X` represents an element that is presumed to be nonzero, and `0` an element that is exactly zero. Now we can traverse the first row immediately, since we know the position of α_{11} in `val` and `rx`. As we do, we can learn that the position of α_{31}: it is simply the position of α_{11} plus 1. We also learn that α_{21} is zero. This means that we can traverse the second row. As we do, we learn the position of α_{32}. This means that we can traverse the third row, at the same time learning the positions of α_{43}, α_{61}, α_{62}. The following table shows this process carried to its conclusion.

After traversing row	we know the positions of
0	$\alpha_{11}\ \alpha_{22}\ \alpha_{33}\ \alpha_{44}\ \alpha_{55}\ \alpha_{66}$
1	$\alpha_{22}\ \alpha_{31}\ \alpha_{33}\ \alpha_{44}\ \alpha_{55}\ \alpha_{66}$
2	$\alpha_{31}\ \alpha_{32},\ \alpha_{33}\ \alpha_{44}\ \alpha_{55}\ \alpha_{66}$
3	$\alpha_{43}\ \alpha_{44}\ \alpha_{55}\ \alpha_{61}\ \alpha_{62}\ \alpha_{66}$
4	$\alpha_{54}\ \alpha_{55}\ \alpha_{61}\ \alpha_{62}\ \alpha_{63}\ \alpha_{66}$
5	$\alpha_{61}\ \alpha_{62}\ \alpha_{63}\ \alpha_{66}$

Note that after we have processed row i we have all the information we need to process row $i+1$.

We have to decide how to encode this information. We will keep it in an array `link` of length `n`, whose contents may be described as follows. As suggested above, the traversal of row `i` begins with the `i`th element. Then `i1 = link[i]` is the column index of another element in the row. Similarly, `i2 = link[i1]` locates yet another element. The list ends when for some `ip` the value of `link[ip]` is zero. A second array, `pos`, gives the positions of the elements in the arrays `rx` and `val`. This method is feasible for two reasons. First, we never need to store more than n links, provided we discard the links associated with a row as it is traversed. Second, the links for different rows cannot overlap, so that two untraversed rows can live together in `link`. We will use this linking technique again when we implement symbolic factorization.

We will package this algorithm in a routine `RowTrav` that produces elements of the matrix row by row. Each call to the routine gives a new element. After a row has been processed, `RowTrav` returns an end of row indication to allow the calling code to take any action required when passing from one row to the next. A drawback of `RowTrav` that it does not return the elements of a row in their natural order; but in many applications — ours in particular — that is not necessary.

More specifically, the program has the calling sequence `RowTrav(A, i, j, posij)`. Here is an illustration of how it traverses the lower part of the `dmat A` row by row.

```
i = -1
RowTrav(A, i, j, posij)
for ix=1 to A.n
   while (RowTrav(A, i, j, posij) != 0)
      process element (i, j)
   end while
   process row i
end for
```

(5.10)

The first call, with `i` negative initializes the routine. Subsequent calls traverse the rows of `A`, producing the row subscript `i`, the column subscript `j` and the position `posij` of the element in the arrays `rx` and `val`. The `(i,i)`-element of row `i` is produced first, but the order of the other elements of the row has no useful pattern. After the `i`th row has been traversed, the routine returns a zero as an end-of-row indication. Under no circumstances should the user change the values of `i` and `j` while the `pcmat` is being traversed.

Algorithm 5.2 contains the code for `RowTrav`. Here are some comments.

- The best way to see what is going on is to work through a small example — say for the matrix (5.9) — tracking the entries in `link` and `pos` as the algorithm proceeds. An interesting feature is that the program must squirrel away the next value of `j` — i.e., `link[j]` — in `nextj`, since the value of `link[j]` may change when `link` is updated.

- `RowTrav` depends heavily on variables like `link`, `pos`, and `nextj` that must retain their values between calls to `RowTrav`. Such variables are said to be *static*, and most programming languages provide them.

- Since the links have to be updated, the row traversal is more expensive than a straightforward column traversal. However, there is only one update per nonzero element of A, so that the algorithm runs in time proportional to `A.nnz`.

- The algorithm carries a storage overhead of $2n$ integers for the arrays `link` and `pos`. In light of the comments above, this does not appear to be excessive.

6. Graphs

We have decided on a numerical algorithm for factoring a matrix and a data structure for representing a sparse matrix. Our job now is to bring them together in harmonious wedlock. It is no easy task.

Symbolic factorization is the key. If we know, the structure of L, we can place the lower half of A in it and implement Algorithm 3.1 using the techniques developed in the last section. But efficient algorithms for symbolic factorization require considerable mathematical support, which is customarily couched in the language of graph theory. This section is devoted to a review of the fundamentals. The next section will treat a particular graph associated with a sparse matrix — the elimination tree. These two sections are the heaviest going in this paper. But don't despair. When you emerge from them you will have arrived at the point where you can read much of the sparse matrix literature on your own.

An *undirected graph* consists of a set of *nodes* (also called *vertices*) and a set of *edges* connecting the nodes. The graph $G(A)$ of a symmetric matrix A of order n has nodes $\{1, 2, \ldots, n\}$ (which may conveniently be identified with the diagonals of A). The set of edges is the set of pairs $\{i, j\}$ for which $i \neq j$ and $\alpha_{ij} \neq 0$. Traditionally, an edge is represented by drawing a line between the two nodes. For example, the matrix (5.2)

`RowTrav` traverses rowwise the lower part of a `pcmat` as shown in (5.10).

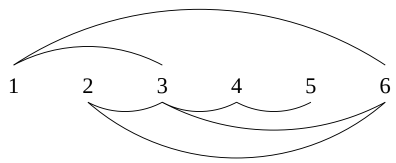
```
RowTrav(A, i, j, posij)
   if (i < 0)
      ! Initialize.
      link[1:A.n] = 0
      pos[1:A.n] = 0
      j = 0; i = 0
      return j
   end if
   if (j = 0)
      ! Set up for row i.
      i = i+1; j = i
      posij = A.colp[i]
   else
      ! Get the next element of row i.
      j = nextj;
      if (j = 0) return j; fi  ! End of row
      posij = pos[j]
   end if
   nextj = link[j]
   link[j] = 0
   nextdown = posij + 1
   if (nextdown < A.colp[j+1])
      ! There is an element in column j. Link it up.
      pos[j] = nextdown
      id = rx[nextdown]
      link[j] = link[id]
      link[id] = j;
   end
   return j
```

Algorithm 5.2: Row traversal

has a structure described by the following Wilkinson diagram:

$$\begin{pmatrix} \mathtt{X} & \mathtt{0} & \mathtt{X} & \mathtt{0} & \mathtt{0} & \mathtt{X} \\ \mathtt{0} & \mathtt{X} & \mathtt{X} & \mathtt{0} & \mathtt{0} & \mathtt{X} \\ \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{0} & \mathtt{X} \\ \mathtt{0} & \mathtt{0} & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{0} \\ \mathtt{0} & \mathtt{0} & \mathtt{0} & \mathtt{X} & \mathtt{X} & \mathtt{0} \\ \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{0} & \mathtt{0} & \mathtt{X} \end{pmatrix}. \tag{6.1}$$

(Here an `X` stands for an element presumed to be nonzero a `0` stands for a zero element.) The graph of this matrix is



(6.2)

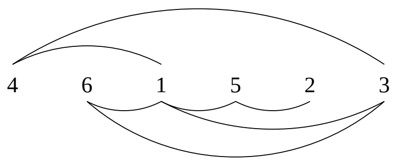
It is a useful exercise to convince yourself that the graph of a grid-graph matrix looks like the grid in Figure 2.1.

In addition to our symmetric positive definite matrix A, we will be interested in a graph associated with its Cholesky factor L. Now L is not a symmetric matrix; but $L+L^{\mathrm{T}}$ is, and from that matrix we can form a graph. For brevity we will abuse notation and write $G(L)$ for $G(L+L^{\mathrm{T}})$. The graph $G(A)$ can be regarded as a subgraph of $G(L)$ by the expedient of dropping all the edges in $G(L)$ that are not in $G(A)$. These edges correspond to filled-in elements.

Changing the numbering of the nodes in the graph of a matrix affects the structure of the matrix. Specifically, let $p_1, p_2, \ldots, p_n$ be a permutation of the integers $1, 2, \ldots, n$. We can get a new graph from from $G(A)$ by the following procedure. If $\{p_i, p_j\}$ is an edge of $G(A)$ replace it by $\{i, j\}$. This graph has exactly the same structure as the original graph since all we have done is move around the numbers of the nodes. For example, under the permutation

$$p_1 = 3 \quad p_2 = 5 \quad p_3 = 6 \quad p_4 = 1 \quad p_5 = 4 \quad p_6 = 2$$

the graph (6.2) becomes



However, the Wilkinson diagram of the corresponding matrix is

$$\begin{pmatrix} \mathtt{X} & \mathtt{0} & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} \\ \mathtt{0} & \mathtt{X} & \mathtt{0} & \mathtt{0} & \mathtt{X} & \mathtt{0} \\ \mathtt{X} & \mathtt{0} & \mathtt{X} & \mathtt{X} & \mathtt{0} & \mathtt{X} \\ \mathtt{X} & \mathtt{0} & \mathtt{X} & \mathtt{X} & \mathtt{0} & \mathtt{0} \\ \mathtt{X} & \mathtt{X} & \mathtt{0} & \mathtt{0} & \mathtt{X} & \mathtt{0} \\ \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{0} & \mathtt{0} & \mathtt{X} \end{pmatrix}. \tag{6.3}$$

We have already observed that reordering a matrix will change the fill-in in its Cholesky factor. The matrices (6.1) and (6.3) are examples of this. The Cholesky factor of (6.3) fills in completely after two elimination steps, leaving only four zero elements. On the other hand, the factor of (6.1) has fill in at only λ_{64} and λ_{65}, leaving six zero elements. Graphs are especially well-suited for finding orderings that reduce fill-in, since renumbering nodes is easier than manipulating the structure of the original matrix.

A *path* in an undirected graph is a sequence of nodes $i_1, i_2, \ldots, i_k$ such that i_j is connected to i_{j+1} by an edge. If there is a path between i and j, we say that i and j are *connected*. A subset of a graph is said to be connected if all its nodes are connected. By convention, each node in a graph is connected to itself. Transitions in a reflection diagram, like the transition from `k3` to `k1` in Figure 4.1, are paths in the graph of the matrix in question.

The property of being connected is an equivalence relation between nodes, and hence the nodes of any graph can be partitioned into disjoint, connected sets that are not connected to one another. They are called the *connected components* of the graph. When the graph is associated with a matrix, this partition has an important interpretation. Suppose, for example, that $G(A)$ has two connected components $\mathcal{C}_1$ and $\mathcal{C}_2$. Suppose further that $\mathcal{C}_1$ has m nodes, and renumber the nodes of $G(A)$ so that nodes $1, 2, \ldots, m$ belong to $\mathcal{C}_1$. Then the matrix corresponding to the renumbered graph has the form

$$\begin{pmatrix} A_{11} & 0 \\ 0 & A_{22} \end{pmatrix}.$$

Thus the matrix reduces to a block diagonal form.

In applications, this means that the sparse system decomposes into two unconnected systems that can be treated separately. For this reason, sparse matrix solvers often try to find the connected components of $G(A)$ as part of the ordering step. We will assume that this has been done and that $G(A)$ is connected. In this case, we also say that the matrix A is *irreducible.* Throughout the remainder of this paper we will assume that A is irreducible.

A *cycle* is a path $i_1, i_2, \ldots, i_{n-1}, i_1$ in which the nodes $i_1, \ldots, i_{n-1}$ are distinct. In other words, a cycle is a nontrivial path that starts and ends at the same node without intersecting itself. A graph without cycles is said to be *acyclic*. A connected acyclic graph is called a *tree*. Given a tree, we can choose any node as a *root* of the tree. Since the graph is connected, there is a path from the root to any other node, and because the graph is acyclic this path is unique. Paths in a tree cannot be extended indefinitely, and their terminal nodes are called *leaves* of the tree. Paradoxically, trees are usually drawn upside down with the root at the top. For an example see Figure 8.1. Trees can be described by a very simple data structure. Let T be a tree, and let the node r be its root. Let i be a node. Then, as noted above, there is a unique path $r, j_1, \ldots, j_k, i$ from r to i. We will call the node j_k the *parent* of i and write $j_k = \text{parent}(i)$. The parent relation uniquely determines the tree. More generally a parent relation specifies a graph whose edges are $\{i, \text{parent}(i)\}$; however, this graph need not be a tree. The following result gives conditions under which a parent relation produces a tree.

> *Let a parent relation defined on the nodes* $1, 2, \ldots, n$ *have following properties.*
>
> 1. *The node* n *does not have a parent.*
> 2. *For* $i \neq n$, $\text{parent}(i) > i$. (6.4)
>
> *Then the graph* T *whose edges are* $\{i, \text{parent}(i)\}$ $(i = 1, \ldots, n-1)$ *is a tree with root* n.

To see this, we must show that T is connected and acyclic. To show the former, we will show that all the nodes are connected to n. Let $i \neq n$ be a node in T. Then the sequence $\text{parent}(i), \text{parent}[\text{parent}(i)], \ldots$ is strictly increasing and bound by n. It follows that it must terminate with the integer n.

To show that T is acyclic, suppose there is a cycle in T, and let i be the smallest node in the cycle. Then i must be connected to two distinct nodes $j > i$ and $k > i$. But then j and k must both be parents of i. The contradiction establishes the result.

The term "parent" suggests a natural nomenclature for expressing relations among the elements of a tree. If k is the parent of j, we call j a *child* of k. If $j < k$ and there is a path from j to k we say that k is an *ancestor* of j and j is a *descendent* of k. We will use this nomenclature freely in what follows.

7. The Elimination Tree

In Section 3.3 we derived the following relation for the structure of the kth column of the Cholesky factor L:

$$\text{str}(\ell_k^{(k)}) = \text{str}(a_k^{(k)}) \cup \bigcup_{\substack{j=1 \\ \lambda_{kj} \neq 0}}^{k-1} \text{str}(\ell_j^{(k)}).$$

Although the union over j has too many terms to make this formula suitable for a symbolic factorization algorithm, we suggested that it might possible to prune terms from union. The device for doing this is a tree called the *elimination tree.* Not only will the elimination tree lead naturally to a symbolic factorization algorithm, but it will provide us with an algorithm for determining the number of nonzero elements in L.

7.1. Definition and basic properties

There seems to be no good way of motivating the definition of the elimination tree. Instead we simply define it to be the tree generated by the following parent relation:

$$\text{parent}(k) = \min\{j > k\colon \lambda_{jk} \neq 0\}.$$

Otherwise put, if we take the Cholesky factor L and retain only the first element below the subdiagonal in each column, then the graph of the resulting matrix is the elimination tree.

This construction assumes that each column of L has a subdiagonal element, in which case (6.4) implies that the parent relation defines a tree. However, it is not trivial to show that the necessary subdiagonal elements exist. We begin with the following technical result.

Let $j > k$. If there is a path $k, k_1, \ldots, k_p, j$ from k to j in $G(L)$ with $k_1, \ldots, k_p < k$, then $\lambda_{jk} \neq 0$ (7.1)

We will use reflection diagrams of L to prove this assertion. We first show that we can assume that $k_1 > \ldots > k_p$ — i.e., that we move backward in the reflection diagram until very last. Suppose to the contrary that there are forward jumps, and consider the first one. If it is the very first jump from k to k_1, we must have $k_1 = j$ and $\lambda_{jk} \neq 0$. Thus we can assume the first forward jump is preceded by a backward jump.

The two reflection diagrams in Figure 7.1 illustrates what can happen. In diagram A we backtrack along `abc` and then move forward along `dce`. However that transition shows that there must be a nonzero at `f`. Hence we can get from `a` to `e` by the transition `afe`, which is backward. Thus we can eliminate the forward jump from the path. This also includes the degenerate case where `a` = `e` and the backward and forward jumps simply cancel one another.

Consider now the diagram B. Here the net jump `abc` and `cde` is forward. However, the presence of `f` shows that it can be replaced by the smaller forward jump `afe`. If we continue this process (note the preceding backward jump changes with each step), one of two things must happen.

1. We find ourselves in case A and can eliminate the forward jump.

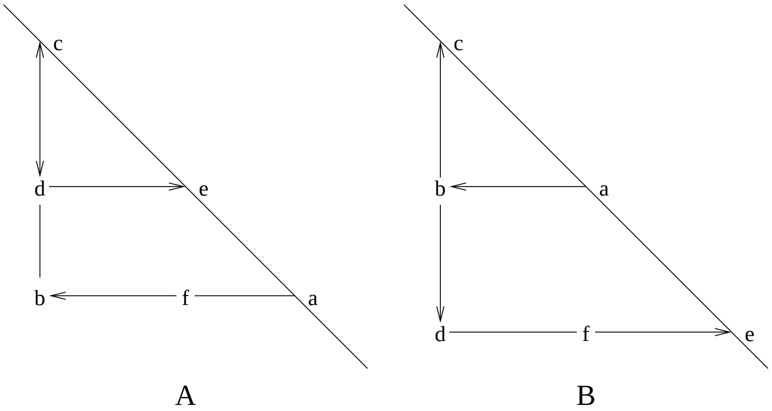


Figure 7.1: No forward jumps

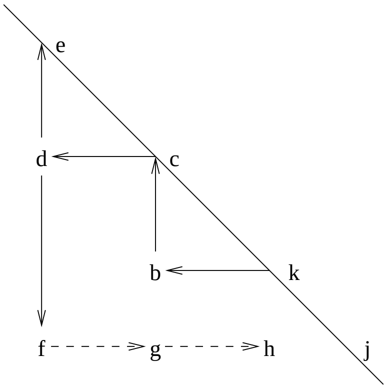


Figure 7.2: Illustrating $\lambda_{jk} \neq 0$

2. The length of the forward jump becomes one, and we again find ourselves in the case A.

Thus we can eventually eliminate the forward jump. Continuing this process with subsequent forward jumps, we end up with only backward transitions.

To complete the proof of (7.1) consider the reflection diagram in Figure 7.2. The path makes two backward transitions `kbc` and `cde` followed by the forward transition `efj` to `j`. But the transition `edc` (a legal transition even if it goes against the arrows) insures that `g` will be nonzero, and hence the transition `cbk` insures that `h` will be nonzero. But `h` occupies the position of λ_{jk}, which is therefore nozero. Except for the specific number of backward jumps, this argument is perfectly general and establishes

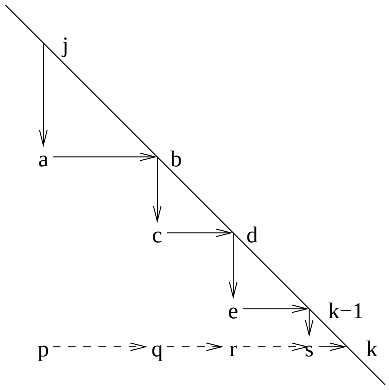


Figure 7.3: $\lambda_{kj} \neq 0 \implies j \in T[k]$

(7.1).

Returning now to the elimination tree, to show that it is well defined we must show that each column of L has a subdiagonal element. As stated above, we will assume that A is irreducible. Let $k < n$ be given and let $j > k$. Then there is a path in $G(L)$ from k to j. Let i be the first node in the path with $i > k$. Then by (7.1), $\lambda_{ik} \neq 0$. In other words, L has an element in column k below the subdiagonal.

We will denote the elimination tree of the Cholesky factor L by T. Let k be given. The graph consisting of all the descendents of k in T along with k itself is obviously a tree. We will denote it by $T[k]$. These trees will play an important role in what follows, and it will be useful to know what elements lie in them. The following result shows that $T[k]$ contains the structure of row k of L.

If $\lambda_{kj} \neq 0$ *then* $j \in T[k]$. (7.2)

To see this, consider the reflection diagram in Figure 7.3. Since $\lambda_{kj} \neq 0$, the element `p` is nonzero. If follows that `b` = parent(`j`) $\leq$ `k`. If it is equal, then we are through: there is a path from `j` to `k` in T. If not, then there is a fill-in at `q` under `b`. Hence `d` = parent(`b`) $\leq$ `k`. If `d` is equal to `k`, we are once again through. Otherwise, there is a fill-in at `r`. Proceeding in this manner, we must either generate a path in T from `j` to `k`, or we must eventually arrive at the node `k-1`. Since there is a fill-in at `s`, we must have parent(`k`−`1`) = `k`, which completes the path.

It is worth pointing out that $T[k]$ can be bigger than $\mathrm{str}(L[k, 1{:}k])$. For example, the structure the kth row of a tridiagonal matrix is $\{k, k{-}1\}$. But $T[k] = \{1, \ldots, k\}$.

7.2. Constructing the elimination tree

Although we have defined elimination trees and shown them to exist, we have not shown how to construct them. In this subsection we will remedy this deficiency and in the process obtain a count of the number of nonzero elements in the Cholesky factor L — a count we will need to implement the symbolic factorization.

We first observe that if we can build up the structure of L row by row in its natural order, we can determine the parent relation that defines the elimination tree. Specifically, we can initialize an array `parent` of length to zero. Now for each index j, parent(j) is the row index i of the first nonzero element of L below λ_{jj}. Thus when we find a nonzero element λ_{ij} in the course of traversing row i, we check `parent(j)` to see if the latter is nonzero. If it is then we have already determined its value while traversing a previous row. If not, we can set `parent(j)=i`. At the end of the process we have the elimination tree of L.

On the other hand, if we know the parent relation we can determine the structure of any row. Specifically, consider the reflection diagram in Figure 4.1. Here $\lambda_{i,k_1} \neq 0$ and α_{i,k_3} is an α-precursor of λ_{i,k_1}. But by (7.2), $k_3 \in T[k_2]$ and $k_2 \in T[k_1]$. Consequently we can find k_1 — the column index of λ_{i,k_1} — by starting at k_1 — a row index of an α-precursor of λ_{i,k_1} — and using the parent relation to move up the elimination tree to k_1. Since every nonzero element in row i has an α-precursor, we can determine the row structure of the ith row of L by following the elimination tree up from the nonzero elements of row i of A.

At this point it looks like we have a vicious circle. If we know the row structure, we can compute the elimination tree; if we know the elimination tree we can compute the row structure. But where to start? Surprisingly, we can start with the structure of the first row, which we know, and build up both the elimination tree and the row structure simultaneously.

Specifically, suppose we have determined the row structures of rows $1, \ldots, i-1$ and have determined the parent relation insofar as is possible with this information. We will say that an index $j < i$ is untouched if

1. it is not known whether j belongs to the structure of row i or
2. it belongs to the structure of row i but the parent relation as so far determined does not define a path from j to i.

Before we start searching the ith row, we mark the indices $1, \ldots, i$ as untouched.

Let α_{ij} be an element of row i of A. We now use the parent relation to move from j up the elimination tree. Eventually one of two things must happen.

1. As we move up the tree, we encounter a node we have previously touched. This case is illustrated by the reflection diagram in Figure 7.4, in which `j2` is the

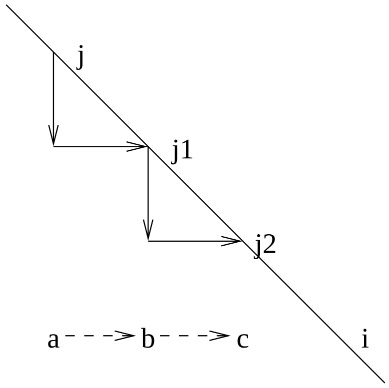


Figure 7.4: Searching a row

touched node. Since a and b are nonzero, `j` and `j1` are in the structure of row `i`. Moreover, since `j2` is touched, there is a path from `j2` to `i`. Thus `j` and `j1` become touched.

2. We reach a node whose parent has not yet been defined. This case is again illustrated by the diagram in Figure 7.4, where it is now assumed that `j2` has no parent. But since we have already search the proceeding rows, there can be no element of L between `c` and `j2`, so that `i` is the parent of `j2`. Thus `j`, `j1`, and `j2` become touched, and we have added `parent(j2)=i` to the parent relation.

Since every element of row i of L has an α-precursor and we start searching at every element of A in row i the result is the complete row structure of row i and an updated parent relation.

There is one technical point that we must dispose of before we can write down code. A natural way of indicating if an node has been touched is to initialize an array `touched` of length n to zero and set `touched[j]` to one when node j has been touched. This works well enough row by row. But when we finish a row and go on to the next, we must reinitialize the array `touched`. If this is done for all n rows, the result is an $O(n^2)$ algorithm, which is forbidden [see the comments after (5.8)]. The cure to the problem is to set `touched[j]` to `i` when we reach node `j` in the search of row `i`. At the start of the search all components of `touched` are strictly less than `i`, so that this procedure marks the touched elements without any reinitialization whatsoever.

Algorithm 7.1 contains a routine to construct the elimination tree of a `pcmat` and count the nonzeros in its Cholesky factor. The routine uses `RowTrav` (Algorithm 5.2) to produce elements from the rows of `A`. `Etgen` requires $4n$ units of auxiliary storage for the arrays `parent` and `touched` as well and the arrays `link` and `pos` in `RowTrav`

Given a `pcmat A`, `Etgen` returns the `parent` structure of its elimination tree and the the nonzero count `nnz` for its Cholesky factor.

```
Etgen(parent, nnz)
   ! Initialize.
   nnz = 0
   touched[1:A.n] = 0
   parent[1:A.n] = 0
   ! Traverse the rows of A.
   i = -1
   RowTrav(A, i, j, posij)
   for ix=1 to A.n
      while (RowTrav(A, i, j, posij) != 0)
         if (i = j)
            ! Process diagonal element.
            nnz = nnz + 1
            touched[j] = i
         else
            ! Off diagonal element. Search the tree.
            js = j
            while (touched[js] != i)
               touched[js] = i
               nnz = nnz + 1
               if (parent[js] = 0)
                  parent[js] = i
                  leave while
               end if
               js = parent[js]
            end while
         end if
      end while
   end while
end Etgen
```

Algorithm 7.1: Constructing an elimination tree

(Algorithm 5.2). Since we have taken care to minimize retouching, the algorithm runs in time proportional to the nonzero count of the Cholesky factor. (To see this, note that the each iteration in the while loop beginning at statement 8 increases `nnz` by one.)

8. Symbolic Factorization

Now that we have a nonzero count for L, we can allocate storage for the symbolic factorization. To complete the factorization we need to be able to determine the structure of the columns of L. We now turn to that task.

8.1. The column structure of L

We have already observed (twice) that the formula

$$\mathrm{str}(\ell_k^{(k)}) = \mathrm{str}(a_k^{(k)}) \cup \bigcup_{\substack{j=1 \\ \lambda_{kj}\neq 0}}^{k-1} \mathrm{str}(\ell_j^{(k)}). \tag{8.1}$$

is not suitable for determining the structure of $\ell_k^{(k)}$ because the union of j in general has too many terms. To prune the range of the union, we begin by observing that if j is a child of k in the elimination tree, then $\mathrm{str}(\ell_j^{(k)}) \subset \mathrm{str}(\ell_k^{(k)})$. This fact follows from the fact that $\lambda_{kj} \neq 0$, so that $\ell_j^{(k)}$ is in the union (8.1). This implies that:

If j is a descendent of k in the elimination tree, then $\mathrm{str}(\ell_j^{(k)}) \subset \mathrm{str}(\ell_k^{(k)})$. (8.2)

Now note that by (7.2) all the terms of the union (8.1) have indices in $T[k]$. Consider column k of L and assume that j is one of its children. Then $\mathrm{str}(\ell_j^{(k)})$ is in the union (8.1). Moreover, by (8.2) columns with index $i \in T[j]$ with $i < j$ can be omitted from the union. Thus all nodes in $T[k]$ that are descendents of a child of k can be pruned, and we are left with only the structure of $a_k^{(k)}$ and the structures of the columns corresponding to the children of k. Hence

$$\mathrm{str}(\ell_k^{(k)}) = \mathrm{str}(a_k^{(k)}) \cup \bigcup_{\substack{j=1 \\ j \text{ a child of } k}}^{n} \mathrm{str}(\ell_j^{(k)}). \tag{8.3}$$

We have thus reduced the set of columns of L that we must merge from those for which $\lambda_{kj} \neq 0$ to those for which j is a child of k in the elimination tree. In general, the latter set is far smaller than the former.

A simple example may make this point clearer. Consider the node 11 in the elimination tree in Figure 8.1. The union in (8.1) may range over as many as all the nodes

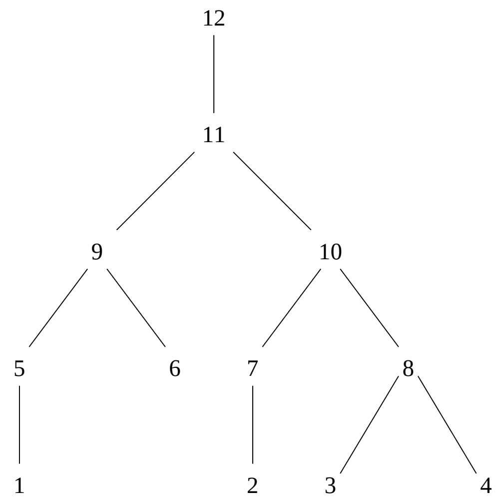


Figure 8.1: An elimination tree

1 through 10. But by (8.2), the column structures of nodes 1, 5, and 6 are a subset of that of 9, and the column structures of the nodes 2, 3, 4, 7, and 8 are subsets of the that of 10. Thus we can prune the structures of columns 1–8 from the union.

A difficulty with this approach is that we need to know the children of the nodes of the elimination tree. Although we have shown how to compute the parent relation that defines the tree, it is not of much help in finding children. (Strange parents that don't know their own children!) It turns out, however, that we can compute the children as we compute the structure of L. Specifically, suppose we have computed the structure of $\ell_k^{(k)}$, so we can find the first nonzero element below the diagonal of $\ell_k^{(k)}$ — call it λ_{jk}. Then k is a child of j. If we store this information, by the time the process reaches column j we will have a list of all the children of j.

8.2. Implementation

Having the characterization (8.3) of the structures of the columns of L, we are now ready to implement the symbolic factorization phase of our old-fashioned solver. We have five problems to tackle.

1. How do we represent the matrix L?

2. How do we determine the storage needed to represent L?

3. How can we keep track of the children of a node?

4. How can we merge the sets in (8.3)? The problem here is to keep the row indices in order.

5. Having computed the structure of the kth column of L, how do we update the structure?

The representation of the matrix L is simple. We can put it in a `pcmat L`. The components of the structure will have slightly different meanings — e.g., `L.colp(k)` points to the first nonzero entry in the `k`th column — but that causes no problems. We have already done something like this in referring to the graph $G(L+L^{\mathrm{T}})$ as $G(L)$.

The second problem stems from the fact that we must allocate storage to contain the arrays `val` and `rx` in the `pcmat` structure. The length of these arrays is `nnz` — the number of nonzero elements in L — which is initially unknown. Fortunately, we can use `Etgen` (Algorithm 7.1) to compute `nnz`. The auxiliary storage required for `Etgen` is proportional to the order of the original matrix, which is known at the outset.

There is an elegant way of keeping track of the children. We create an array `bs` (for baby sitter) of length n and initialize it to zero. When we find a child of, say, node j we put its number in `bs[j]`. If we find another child, we place it in `bs[bs[j]]`, and so on. After we have computed the structure of column j, we zero out the corresponding components of `bs`. Note that if j has, say, two children then the contents of `bs[bs[bs[j]]]` will always be zero, because node `bs[bs[j]]`, being a child of `j`, will have already been processed. Thus when we encounter a zero component in the sequence `bs[j]`, `bs[bs[j]]`, $\ldots$, we will have processed all the children of `j`.

By way of illustration, Figure 8.2 exhibits the contents of the baby-sitter array as we proceed through the tree in Figure 8.1. The number to the side in a row is the node that has just been processed. The number at the top is the position in the baby-sitter array.

The merging problem arises from the fact that the columns structures of the children of a node will not all be the same. For example, if the column structure of node j_1 is $\{4,7,9\}$ and that of j_2 is $\{5,7,10\}$, then we must merge these structures to get $\{4,7,9,10\}$. A natural way to proceed is to initialize and array of n integers to zero. When we encounter a new element of the structure, we set the corresponding entry of the array to one. The trouble with this approach is that at the end of determining the structure of column k, we must search entries $k,\ldots,n$ of the array to recover the structure. Repeated n times, this gives an $O(n^2)$ algorithm.

The alternative we will use also requires an auxiliary array, `ma` (for merge array), of length n. Its use is best seen through an example. Suppose that we are accumulating the structure of column 3, of a matrix of order 10, and suppose the current state of the structure set is $\{3,5,6,9\}$. Then the merge array contains the following entries.

```
11 11  5 11  6  9 11 11 11 11
```

	1	2	3	4	5	6	7	8	9	10	11	12
1 :	0	0	0	0	1	0	0	0	0	0	0	0
2 :	0	0	0	0	1	0	2	0	0	0	0	0
3 :	0	0	0	0	1	0	2	3	0	0	0	0
4 :	0	0	4	0	1	0	2	3	0	0	0	0
5 :	0	0	4	0	0	0	2	3	5	0	0	0
6 :	0	0	4	0	6	0	2	3	5	0	0	0
7 :	0	0	4	0	6	0	0	3	5	7	0	0
8 :	0	0	0	0	6	0	8	0	5	7	0	0
9 :	0	0	0	0	0	0	8	0	0	7	9	0
10 :	0	0	0	0	0	0	0	0	10	0	9	0
11 :	0	0	0	0	0	0	0	0	0	0	0	11
12 :	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8.2: A baby-sitter array

Thus the first element of the structure is `3` (the number of the column under consideration), the second element is `ma[3] = 5`, the third is `ma[5] = 6`, and so on. Note that unused members of `ma` are set to `n+1`, so that if `m[i]=n+1` we are at the end of the merge list.

To keep things simple, we will assume that we have at hand a routine

```
Merge(B, j, k, ma)
```

that merges the structure of column `j` of `B` into the current structure for column `k`. We will later give code for `Merge`.

The problem of filling in the columns of the `pcmat` for L is relatively simple, but again for convenience we relegate this computation to a function

```
Makecol(k, ma, L)
```

that takes the output of `Merg` and transfers it to the `k`th column of the `pcmat L`. This routine also reinitializes the merge array.

Algorithm 8.1 performs symbolic factorization. It is relatively straightforward. It uses `Merge` to initialize `ma` to the `k`th column of `A`, after which it folds in the structures of the children of node `k`. It then generates the `k`th column of `L` from `ma` and uses it to update the baby sitter. The loop beginning with statement 11 is the heart of the algorithm. Since a child can have only one parent, the call to `Merge` is executed only `n-1` times.

Algorithm 8.2, merges column structures. It depends on the facts that the structure of column `k` starts at `k`, that the row indices from column `j` of `B` are strictly increasing, `Symbolfac` computes the symbolic factorization of the matrix in the `pcmat A` and places it in the `pcmat L`, which is assumed to be suitably initialized.

```
Symbolfac(A, L)
   pcmat A, L
   int bs[n], ma[n]
   ! Initialize.
   for i=1 to n
      bs[i] = 0
      ma[i] = A.n + 1
   end for i
   ! Main loop on columns of A.
   for k=1 to n
      ! Compute the structure of the kth column.
      Merge(A, k, k, ma)
      j = bs[k]
      while (j != 0)
         Merge(L, j, k, ma)
         jt = bs[j]; bs[j] = 0; j = jt
      end
      ! Set up the kth column of L.
      Makekcol(k, ma, L)
      ! Update the baby sitter.
      if (k != n)
         j = L.rx[L.colp[k] + 1]  ! j is the parent of k
         while (j != 0) jt = j; j = bs[j]; end
         bs[jt] = k
      end if
  end for k
end Symbolfac
```

Algorithm 8.1: Symbolic factorization

`Merge` merges the structure of the jth column of B into the current structure of column `k` as represented by `ma`.

```
Merge(B, j, k, ma)
   m = k
   ! Loop over elements in column j of B.
   for ii=B.colp[j]+2 to B.colp[j+1]-1
       i = B.rx[ii]
      ! Search for m and m1 with m < i <= m1.
      m1 = m
      while (i > m1)
         m = m1;
         m1 = ma[m]
      end while
      if (i != m1)
         ! Insert i in ma.
         ma[m] = i
         ma[i] = m1
      end if
      m = i
   end for ii
end Merge
```

Algorithm 8.2: Merging structures

`Makecol` takes the structure for column `k` contained in `ma` and transfers it to the `k`th column of the `pcmat` `L`. It also reinitializes the merge array `ma`.

```
Makecol(k, ma, L)
   if (k = 1) L.colp[1] = 1; fi
   ii = L.colp[k]
   m = k
   while (m < L.n+1)
      L.rx[ii] = m
      ii = ii + 1
      mt = ma[m]
      ma[m] = L.n+1
      m = mt
   end while
   L.colp[k+1] = ii
end Makecol
```

Algorithm 8.3: Generate a column of L

and that the unused parts of `ma` are set to `n+1`. By initializing `m` to `k` and resetting it to `i = B.rx[ii]` after element `i` has been processed, we can be assured that at statement 6 we have `m < i`. Thus we have a starting point to search for a bracket `[m,m1]` satisfying

`m < i <= ma[m] = m1.`

Once this bracket has been established, we can easily incorporate `i` into `ma`. The fact that the unused parts of `ma` are set to `n+1` makes the algorithm work when we are appending an element to the end of the list.

The merging starts with the third element in column `j`. The reason is that the first element is the `j`th, which cannot be in the structure of column `k`, since `j<k`. The second element of column `j` has the row index `k`, since `j` is a child of `k`, and therefore `j` was entered into the structure of column `k` when the `k`th column of `A` was processed.

At worst any call to merge involves passing through the number of elements in `k`, and merge must be called for each child of `k`. if `CMAX` is the maximum number of children any node has, then total time spent merging will be bounded by `CMAX*L.nnz`. In practice, nodes in an elimination tree are an infertile lot and tend to have only one or two children.

Algorithm 8.3 takes the merge array and generates a column of `L`. Essentially it traverses the array `ma` and transfers the row indexes to `L.rx`. The reinitialization of the merge array `ma` illustrates a point about the economics of sparse elimination — one that we have already encountered in connection with the generation of the elimination tree. In the program `Symbolfac` we initialized the array `ma` in statement 6, and it would seem more natural to reinitialize it by moving the initialization inside the loop on `k`. However, that would result in $O(n^2)$ operations, which we have seen is unacceptable. On the other hand, `Makecol` only reinitializes the components of `ma` that have actually been changed, so that the total work in maintaining `ma` is proportional to the number of nonzero elements of L.

9. The Numerical Coda

The object of the long development above is to put us in a position where we can solve the sparse system $Ax = b$. As we have seen earlier we can do this by computing the Cholesky factor L of A and solving the systems $Ly = b$ and $L^{\mathrm{T}}x = y$. Because we have predetermined the structure of L, the numerical factorization and triangular solves are something of an anticlimax — a straightforward translation of standard algorithms into the language of `pcmats`. We will begin with the factorization.

9.1. Numerical factorization

The numerical factorization is an implementation of the algorithm `Colchol` (Algorithm 3.1). It heart is the computation of the sum

$$a_k^{(k)} - \sum_{\substack{j=1 \\ \lambda_{jk} \neq 0}}^{k-1} \lambda_{kj} \ell_j^{(k)},$$

where $a_k^{(k)} = A[k{:}n, k]$ and $\ell_j^{(k)} = L[k{:}n, j]$. There are two problems associated with this computation.

The first problem is how to locate the columns j for which $\lambda_{kj} \neq 0$. But these columns correspond to the nonzero elements in row k of L. Thus we can locate them using the routine `RowTrav` (Algorithm 5.2). In fact `RowTrav` will turn out to be the driver of our algorithm.

The second problem is where to accumulate the sum. One possibility is to accumulate it in the kth column of the `pcmat L`. This can certainly be done, but the indexing is complicated, since $\ell_j^{(k)}$ will generally have fewer nonzeros that $\ell_k^{(k)}$. An alternative, is to use an *accumulator array* `accum` of length n. We zero out the components of `accum` corresponding to the nonzero elements of $\ell_k^{(k)}$ and then load the nonzero components of $a_k^{(k)}$ into their natural positions. After the sum has been accumulated in the array, it is modified to give the kth column of L and returned to the kth column of the `pcmat`.

Algorithm 9.1 performs the numerical factorization. Note the nice way `RowTrav`

`Numfac` overwrites the `pcmat` `L` with the Cholesky factor of `A`.

```
Numfac(A, L)
   k = -1
   RowTrav(L, k, j, poskj)
   for kx = 1 to L.n ! Process column k
      while (RowTrav(L, k, j, poskj) != 0)
         if (j = k) ! Initialize accum.
            for ii=L.colp[k] to L.colp[k+1]-1
               accum[L.rx[ii]] = 0
            end for ii
            for ii=A.colp[k] to A.colp[k+1]-1
               accum[A.rx[ii]] = A.val[ii]
            end for ii
         else ! Subtract L[k:n,j] from L[k:n,k]
            Lkj = L.val[poskj];
            for ii=poskj to L.colp[j+1]-1
               i = L.rx[ii];
               accum[i] = accum[i] - Lkj*L.val[ii]
            end for ii
         end if
      end while
      Move L[k:n,k] from accum to L, adjusting its components.
      for ii=L.colp[k] to L.colp[j+1]-1
         i = L.rx[ii]
         if (i = k)
            L.val[ii] = sqrt(accum[i])
            Lkkinv = 1/L.val[ii]
         else
            L.val[ii] = Lkkinv*accum[i]
         end if
      end for ii
   end for kx
end Numfac
```

Algorithm 9.1: Numerical factorization supports the algorithm. The first element it returns in row `k` is the `k`th, which is just what we need to initialize `accum`. Moreover, when we get the `j`th element, `poskj` points to the top of the vector $\ell_j^{(k)}$.

There are four comments to be made about this algorithm.

- The use of an accumulator has the disadvantage that references are spread out unsystematically across an array of memory consisting of n words. Such references are known to reduce cache performance — i.e., to slow the rate at which items are read from or written to memory. If we perform the elimination within the `pcmat L`, the references are less separated in memory, which improves cache performance at the cost of additional indexing.

- The only part of the array `accum` that needs to be initialized at each stage are the components corresponding to the nonzeros of $\ell_k^{(k)}$. Thus the initialization costs are proportional to `L.nnz`.

- The elements of A are automatically transferred to L in the process of initializing `accum`.

- In some applications we must repeatedly solve systems of the same structure but with different numerical values. Because the Cholesky factors will also have a common structure, we can reuse `L` when we perform the numerical factorizations.

9.2. Triangular solves

As we have seen earlier, we can solve the system $Ax = b$ by solving the two systems $Ly = b$ and $L^{\mathrm{T}}x = y$. We will now show how to solve these systems when L is represented by a `pcmat`.

Since in a `pcmat` column traversals are more efficient that row traversals, we should use a column oriented algorithm to solve the system $Ly = x$. We can derive one as follows. Partition the system in the form

$$\begin{pmatrix} \lambda & 0 \\ \ell & \hat{L} \end{pmatrix} \begin{pmatrix} \eta \\ \hat{y} \end{pmatrix} = \begin{pmatrix} \beta \\ \hat{b} \end{pmatrix}.$$

Then from the first row of the partition, we get

$$\lambda\eta = \beta,$$

from which we find

$$\eta = \lambda/\beta.$$

From the second row, we get

$$\eta\ell + \hat{L}\hat{y} = \hat{b},$$

Let the lower triangular matrix L be contained in the `pcmat L` and let b be contained in an array `b`. `Lsolve` overwrites `b` with the solution of the system $Ly = b$.

```
Lsolve(L, b)
   for j=1 to L.n
      b[j] = b[j]/L.val[L.colp[j]]
      for ii=L.colp[j]+1 to L.colp[j+1]-1
         i = L.rx[ii]
         b[i] = b[i] - b[j]*L.val[ii]
      end for ii
   end for j
end Ltsolve
```

Algorithm 9.2: Solution of $Ly = b$

from which we find

$$\hat{L}\hat{y} = b - \eta\ell.$$

This is a linear system of order one less than the original, which can be solved by a recursive application of the above process.

All this leads to the following algorithm.

```
y = b
For j=1 to n
   y[j] = b[j]/L[j,j]
   for i=j+1 to n
      y[i] = y[i] - y[j]*L[i,j]
   end for i
end for j
```

(9.1)

The algorithm destroys the original right-hand side b, which in many applications is not needed. In fact, we can arrange for the algorithm to overwrite b with the solution y by replacing all references to y with references to b.

Algorithm 9.2 overwrites b with the solution of $Ly = b$. It is a straightforward implementation of (9.1) for a `pcmat L`. Note that it touches each nonzero element of L only once. Hence it runs in time proportional to `L.nnz`.

Turning now to the solution of $L^{\mathrm{T}}x = b$, we first note that if we set $U = L^{\mathrm{T}}$, then U is upper triangular. Since the columns of L correspond to the rows of U, we must now find a row oriented algorithm for solving $Ux = b$.

The algorithm, which is the classical back-substitution algorithm taught in connection with Gaussian elimination, can be derived as follows. Partition the system $Ux = b$

as follows.

$$\begin{pmatrix} \upsilon & u^{\mathrm{T}} \\ 0 & \hat{U} \end{pmatrix} \begin{pmatrix} \xi \\ \hat{x} \end{pmatrix} = \begin{pmatrix} \beta \\ \hat{b} \end{pmatrix}.$$

Then from the last row we have

$$\hat{U}\hat{x} = \hat{b}$$

and from the first row

$$\upsilon\xi = u^{\mathrm{T}}\hat{x}$$

Thus if we have already solved for $\hat{x}$ (by a recursive application of our algorithm), we can solve for ξ in the form

$$\xi = \upsilon^{-1}u^{\mathrm{T}}\hat{x}.$$

The following algorithm, in which x overwrites b, implements this scheme.

```
for i=n to 1 by -1
   for j=i+1 to n
      b[i] = b[i] - b[j]*U[i,j]
   end for j
   b[i] = b[i]/U[i,i]
end for i
```

When we write the algorithm in terms of L, we get

```
for j=n to 1 by -1
   for i=j+1 to n
      b[j] = b[j] - b[i]*L[i,j]
   end for i
   b[j] = b[j]/L[j,j]
end for j
```

Algorithm 9.3 overwrites b with the solution of $L^{\mathrm{T}}x = b$. Like its counterpart for $Ly = b$, it runs in time proportional to `L.nnz`.

10. Back to the future

We have completed the construction of our old-fashioned sparse solver. It is not a toy. Around 1975, highly skilled researchers were working hard to perfect a solver like ours. But neither is it a state-of-the-art, twenty-first century solver. To give you a feel for what came after, we will look at two ideas that have played an increasingly important role in sparse matrix technology: supernodes and multifrontal elimination.

Both these ideas address a problem that we have mentioned in connection with Algorithm 9.1 for numerical factorization: namely references to elements in the accumulator

Let the lower triangular matrix L be contained in the `pcmat L` and let b be contained in an array `b`. `Ltsolve` overwrites `b` with the solution of the system $L^{\mathrm{T}}x = b$.

```
Ltsolve(L, b)
   for j=n to 1 by -1
      for ii = L.colp[j]+1 to L.colp[j+1]-1
         i = L.rx[ii]
         b[j] = b[j] - b[i]*L.val[ii]
      end for ii
      b[j] = b[j]/L.val[L.colp[j]]
   end for j
end Ltsolve
```

Algorithm 9.3: Solution of $L^{\mathrm{T}}x = b$

jump around irregularly over n memory locations. This not only can slow down memory access, but it also makes it difficult to vectorize the computations. Both approaches mitigate this problem by concentrating at least some of the memory references into a compact, fully utilized region of memory.

The purpose of this section is to sketch in outline, and we will not present things in detail as in the earlier sections. If you like, look on the statements here as postgraduate exercises, where you have the opportunity to test your mastery of the subject.

10.1. Supernodes

A supernode is a maximal sequence of consecutive of column indices of L, whose columns have essentially the same structure. Specifically, the sequence $s, \ldots, s{+}t{-}1$ form a supernode if

$$\mathrm{str}(\ell_s) = \mathrm{str}(\ell_{s+t-1}) \cup \{s, s{+}1, \ldots, s{+}t{-}2\}.$$

Since $s, \ldots, s{+}t{-}1$ are in the structure of ℓ_s, the the lower triangle of the matrix $L(s{:}s{+}t{-}1, s{:}s{+}t{-}1)$ must be full. Moreover, the structure of the columns below this triangle — i.e., columns $L(s{+}t{:}n, j)$ $(j = s, \ldots, s{+}t{-}1)$ — must have the same structure. The structure of a supernode is illustrated in Figure 10.1. It might be thought that the supernode structure is so special that it is unlikely to arise in practice. On the contrary, many problems give rise to matrices with a rich supply of supernodes.

In our packed column representation, all the nonzero elements of a supernode end up stored columnwise in the contiguous region of memory from `L.val(L.colp(s))` to `L.val(L.colp(s+t)-1)`. This has an important implication for the numerical factorization phase of our old-fashioned solver. Suppose that the columns of a supernode have

```
X 0 0 0
X X 0 0
X X X 0
X X X X
0 0 0 0
0 0 0 0
0 0 0 0
X X X X
X X X X
0 0 0 0
X X X X
X X X X
X X X X
```

Figure 10.1: A typical supernode

been generated, and they need to be used to generate a subsequent column. Ordinarily, a multiple each column of the supernode would be subtracted from the accumulator in Algorithm 9.1. But alternatively, we can compute the sum of each contribution directly from the array array `L.val`, which can be done quite efficiently because of the supernode structure. This sum can then be added into the accumulator as usual.

A less significant savings must be had when the supernode itself must be factored. Namely, one can apply columns $1, \ldots, s{-}1$ to all the columns of the supernode in the usual way, and the factorization can be completed in the array `L.val`.

There are many applications of supernodes that cannot be illustrated by our old-fashioned solver. For example, they can be used to reduce effective size of the graph of L. This is done by regarding the set $\{s, \ldots, s{+}t{-}1\}$ as a single node (whence the name supernode) and adjusting the edges so that any edge involving one of the nodes, is now associated with the supernode. This trick can save considerable time in manipulations with the graph of L.

Supernodes can be calculated directly from the packed column structure of L. In fact, one only needs to know the number $\eta(j)$ of nonzero elements in column j of L. Specifically, $\{s, \ldots, s{+}t{-}1\}$ is a supernode if and only if it is a maximal set of nodes such that $s{+}i{-}1$ is a child of $s{+}i$ in the elimination tree and

$$\eta(s) = \eta(s{+}t{-}1) + t - 1. \tag{10.1}$$

However, there are other ways of detecting supernodes, and which one is most suitable will depend on the details of the solver.

10.2. The multifrontal method

We will introduce the multifrontal method by considering the classical Gaussian elimination algorithm sketched Section 3.1. The matrices in this method are associated with rows of A and L, and will be convenient to extend our previous notation. Recall that we used $\ell_j^{(k)}$ to represent the part of the jth column of L extending from λ_{jk} downward — i.e., $L(k{:}n, j)$. In what follows we use the superscript $^{(k)}$ for the part of a vector associated with rows k through n, or the trailing principal submatrix of a matrix that begins with its (k,k)-element.

Let the equation $A = LL^{\mathrm{T}}$ be partitioned in the form

$$\begin{pmatrix} \alpha_{11} & a_1^{(2)\mathrm{T}} \\ a_1^{(2)} & A^{(2)} \end{pmatrix} = \begin{pmatrix} \lambda_{11} & 0 \\ \ell_1^{(2)} & L^{(2)} \end{pmatrix} \begin{pmatrix} \lambda_{11} & \ell_1^{(2)\mathrm{T}} \\ 0 & L^{(2)\mathrm{T}} \end{pmatrix}.$$

Then as in Section 3.1, we find that

1. $\lambda_{11} = \sqrt{\alpha_{11}}$,
2. $\ell_1^{(2)} = \lambda_{11}^{-1} a_{11}^{(2)}$
3. $L^{(2)} L^{(2)\mathrm{T}} = A^{(2)} - \ell_1^{(2)} \ell_1^{(2)\mathrm{T}} \equiv A^{(2)} + U_1^{(2)}$,

where $U_1^{(2)}$ is called an *update matrix.*

Conventional Gaussian elimination would continue the process with the matrix $A^{(2)} - U_1^{(2)}$. But the computations can be arranged differently. Instead of incorporating the updates in $U_1^{(2)}$ and its successors into the current matrix, we can accumulate them in update matrices and use them to generate L column by column. Specifically, let

$$U_{k-1}^{(k)} = -\sum_{i=1}^{k-1} \ell_i^{(k)} \ell_i^{(k)\mathrm{T}}, \tag{10.2}$$

be the $(k-1)$th update matrix. We now partition

$$U_{k-1}^{(k)} = \begin{pmatrix} v_{k-1}^{(k)} & u_{k-1}^{(k+1)\mathrm{T}} \\ u_{k-1}^{(k+1)} & U_{k-1}^{(k+1)} \end{pmatrix}$$

and form the *frontal matrix*

$$F_k = \begin{pmatrix} \alpha_{kk} & 0 \\ a_k^{(k+1)} & 0 \end{pmatrix} + \begin{pmatrix} v_{k-1}^{(k)} & u_{k-1}^{(k+1)\mathrm{T}} \\ u_{k-1}^{(k+1)} & U_{k-1}^{(k+1)} \end{pmatrix} = \begin{pmatrix} \hat{\lambda}_{kk} & u_{k-1}^{(k+1)\mathrm{T}} \\ \hat{\ell}_k^{(k+1)} & U_{k-1}^{(k+1)} \end{pmatrix}.$$

From (10.2) it follows that the first column of F_k is $a_k^{(k)}$ with updates and is therefore the vector $\hat{\ell}_k^{(k)}$ in equation (3.5). Consequently $\lambda_{kk} = \sqrt{\hat{\lambda}_{11}}$, and $\ell_k^{(k+1)} = \lambda_{11}^{-1} \hat{\ell}_k^{(k+1)}$.

Thus we have computed the kth column of L. The next update matrix is given by

$$U_k^{(k+1)} = U_{k-1}^{(k+1)} - \ell_k^{(k+1)} \ell_k^{(k+1)^T}.$$

Proceeding in this manner we can compute all the columns of L.

On the face of it, this is a perfectly silly way to implement Gaussian elimination. Computing a column of ℓ by adding an update matrix into a mostly empty matrix is plainly inefficient. But this is only because for a general dense matrix the order of elimination is fixed — a fact reflected in its elimination tree, which is a straight line going from its root at node n to its single leaf at node 1.

Things are otherwise for a sparse matrix whose elimination tree has many branches. In particular, because the kth column of L depends only on the columns of A corresponding to $T[k]$, we can compute it without having to to compute any columns corresponding to the set complementary to $T[k]$. To see this, let j be a child of k, and define the update matrix $U_j^{(k)}$ by

$$U_j^{(k)} = -\sum_{i\in T[j]} \ell_i^{(k)} \ell_i^{(k)\mathrm{T}}. \tag{10.3}$$

If we sum the $U_j^{(k)}$ over the children of k, it can be verified (this is your final exam in elimination trees) that the first column of the sum is precisely the vector that must be added to $a_k^{(k)}$ to get $\hat{\ell}_k^{(k)}$.

All this leads to the following algorithm for computing $\ell_k^{(k)}$ and $U_k^{(p)}$, where $p =$ parent(k).

1. Assemble the frontal matrix

$$F_k = \begin{pmatrix} \alpha_{kk} & 0 \\ a_k^{(k+1)} & 0 \end{pmatrix} + \sum_{j \text{ a child of } k} \begin{pmatrix} v_j^{(k)} & u_j^{(k+1)\mathrm{T}} \\ u_j^{(k+1)} & U_j^{(k+1)} \end{pmatrix} = \begin{pmatrix} \hat{\lambda}_{kk} & f_k^{(k+1)\mathrm{T}} \\ \hat{\ell}_k^{(k+1)} & F_k^{(k+1)} \end{pmatrix}. \tag{10.4}$$

2. Compute

$$\lambda_{kk} = \sqrt{\hat{\lambda}_{kk}} \quad \text{and} \quad \ell_k^{(k+1)} = \lambda_{kk}^{-1} \hat{\ell}_k^{(k+1)}$$

3. Let $p = \text{parent}(k)$ and compute

$$U_k^p = F_k^{(p)} - \ell_k^{(p)} \ell_k^{(p)^{\mathrm{T}}}.$$

By executing this algorithm for $k = 1, \ldots, n$, we can compute the Cholesky factor of A.

This is still not a working algorithm, since it consumes too much storage. In the first place, the update matrices are symmetric. This problem may be solved by storing only the lower half of of these matrices, and likewise for the frontal matrices.

More important, the update and frontal matrices are sparse. For from (10.3) and the fact that

$$i \in T[j] \implies \mathrm{str}(\ell_i^{(k)}) \subset \mathrm{str}(\ell_j^{(k)}),$$

it follows that if $i \notin \mathrm{str}(\ell_j^{(k)})$ then the row and column of $U_j^{(k)}$ corresponding to i are zero. A similar statement holds for the frontal matrices. The cure is to remove these empty rows and columns to give full dense matrices. When we do this, however, the assembly of the frontal matrix becomes more difficult, since the update matrices $U_j^{(k)}$ in (10.4) are no longer of the same size. What one has to do is to calculate where each element in the update matrices goes in the frontal matrix and add it in. This creates additional overhead for the algorithm. But at least we are working with dense matrices.

A final adjustment of the algorithm is necessary. We can form update matrices in any order as long as we form the update matrix for the children of j before we form the update matrix for j. However, no update matrix can be discarded until it has been used to compute the update matrix of its parent. For example, if in the elimination tree of Figure 8.1 we generate elimination trees in the natural order, at one point we will have to store the five update matrices corresponding to columns 2, 3, 4, 5, and 6. On the other hand, if we generate update matrices in the order 1, 5, 6, 9, 3, 4, 8, 2, 7, 10, 11, 12 we never have to store more that two update matrices at any one time. This latter is an example of a *postordering* of a tree, and it is not surprising that consumers of multifrontal algorithms are keenly interested in finding optimal postorderings.

Supernodes mix well with the multifrontal approach. With proper organization, the method requires only one update matrix per supernode. If there are many nice fat supernodes the savings will be proportionately great.

11. Bibliographical notes

Just as it was impossible to present a fully modern sparse solver in this paper, it is equally impossible to give a full bibliographical survey of the subject. The following notes contain some primary references along with more recent references containing surveys and bibliographies.

11.1. Sparse matrices and solvers

In 1968 Ralph Willoughby organized a meeting on sparse matrices at the IBM Research Center at Yorktown Heights and edited its proceedings [27]. This meeting marks the emergence of the subject as a coherent field. It was followed by a sequence of meetings, whose proceedings give a history of the development of the subject over a little more than a decade [2, 4, 6, 17, 19].

It is only fair, however, to note that many of the techniques that would prove important after the first sparse matrix meeting were in place before it began. In a 1963 paper Sato and Tinney [22] describe a compressed row storage scheme for the sparse factors and the use of an accumulator in the numerical factorization. In addition, they propose a primitive ordering scheme, which today we should call a minimum degree ordering based on the original rows. In 1967 Tinney and Walker [26] described the classical minimum degree ordering. Although they do not say how they computed it, they comment

> At the completion of the optimal ordering algorithm [scheme 2) or 3)], the exact form of the table of factors is established and this information is recorded in various tables to guide the actual elimination.

In other words, in a combined ordering and symbolic factorization, they set up the structure for the subsequent numerical factorization — just like subsequent sparse solvers. The terms symbolic factorization, numerical factorization, and solve, along with a compressed row storage scheme, were introduced by Chang [3] at the 1968 sparse matrix meeting.

There are not a large number of textbooks on the subject of sparse matrices. George and Liu's *Computer Solution of Large Sparse Positive Definite Systems* [10], although somewhat dated, is still valuable, and I have drawn heavily on it for this paper. Duff, Erisman, and Reid's *Direct Methods for Sparse Matrices* [5] is an excellent introduction to the basics with a hands-on flavor. Unfortunately, both books are out of print.

Grid-graph matrices arises from elliptic partial differential equations discretized on a square. They had traditionally served as model problems for the solution of linear systems by iterative methods, and their factorization naturally became an important problem in direct sparse algorithms. The nested-dissection ordering is due to George [8] as are the operation and fill-in counts given here. For the optimality of nested dissection see [11].

There is a large literature on ordering, which we cannot survey here. The texts cited above contain much useful material. Saad's *Iterative Methods for Sparse Linear Systems* [21] contains a brief survey of ordering methods with pointers to the more recent literature.

A happy practice of the sparse community is that they implement their algorithms in high quality software. The solver of this paper is a cousin of two excellent packages produced in the 1970's: The Yale Sparse Matrix Package [7] and SPARSEPACK [10, Appendix A], developed at the University of Waterloo.

11.2. The Cholesky decomposition and fill-in

For the basic variants of Gaussian eliminate see [24, Ch. 3]. The columnwise algorithm, which we use here, has a rowwise analogue, which can also been used to implement sparse solvers.

The result (3.7) on α-precursors is an example of a general class of theorems that go under the rubric of path theorems. In terms of graph theory (3.7) says that if $\lambda_{ik} \neq 0$ then there is a path $k, k_1, \ldots, k_p, i$ in $G(L)$ with the $k_i < k$ and $\alpha_{i,k_p} \neq 0$. The granddaddy of path theorems is the elegant result, due to Rose, Tarjan, and Lueker [18], that $\lambda_{ik} \neq 0$ if and only if there is a path $k, k_1, \ldots, k_p, i$ in $G(A)$ with the $k_i < k$.

I devised reflection diagrams in an attempt to simplify the proofs in the literature. However Iain Duff has told me that he has used such diagrams informally. They also appear in an unpublished manuscript by Gibert and Lui.

11.3. Representing and manipulating sparse matrices

There are many other schemes for representing sparse matrices than packed column format. Saad [20] gives descriptions of the most important ones along with programs for converting from one to the other.

The row traversal algorithm was designed specifically for this paper, but it was inspired by the numerical factorization code in George and Liu [10].

11.4. Graphs

Parter [16] was the first to relate graphs and Gaussian elimination applied to sparse matrices. As George [9] points out, however, the graph-theoretic results most useful in sparse applications have been developed independently of classical graph theory. On the other hand, algorithms for manipulating graphs, developed primarily by computer scientists, are widely used in sparse matrix technology. Two standard references are [1, 25].

11.5. The elimination tree

The elimination tree is so useful that it or its near equivalents were invented and reinvented by several people (for a list of references see [12, p. 130]). Liu [13] gives a magisterial survey of the elimination tree and its applications, which has greatly influenced this paper.

Algorithm 7.1 for generating the elimination tree is due to Liu [12, 13]. It is less efficient than it might be, taking time proportional to `L.nnz`. We can improve it by a process known a path compression. Specifically, in a separate array we record the most distant ancestor currently found for each node. When it comes time to start searching from node `i` we can use this information to jump over already touched nodes. The process reduces the time to $O(\texttt{A.nnz} \log \texttt{n})$. (This algorithm is also due to Liu [12].) For our solver there is not much to choose between the two since the work in the numerical factorization is generally much greater than $O(\texttt{L.nnz})$.

11.6. The numerical coda

Schrieber [23] gives an algorithm for numerical factorization that avoids using an accumulator. It is based on two observations.

First, the elimination tree can be used to guide the computation of the correction in column k. To illustrate this, consider the node 10 in the elimination tree in Figure 8.1. Assuming that all the nodes below 10 form the row structure of row 10 of L, we can combine column 2 with 7, columns 3 and 4 with 8, and finally columns 7 and 8 with 10. This can be done quite efficiently with a stack of auxiliary storage. Get storage for 10, then 7. Combine 2 and 7, then 7 and 10, popping the storage for 7. Get storage for 8, combine 3, 4, and 8, then 8 and 10, popping the storage for 8.

The advantage of this scheme is that the storage for a top node of a combination is exactly the size of the column structure of the that node — generally much smaller than the size n of an accumulator. Consequently, there is less jumping around of memory references and better cache performance.

The second observation is that instead of storing row indices for a column of `L`, we can store the relative indices of where the elements will end up when the column is combined with its parent column. This not only makes combining columns easy, but it also reduces storage overhead, since the relative indices are smaller than row indices and can be packed into a smaller part of memory.

11.7. Back to the future

A good source of references for supernodes is [15], which contains the condition (10.1). Liu [14] gives a survey of multifrontal methods, including references for the use of supernodes and relative indices.

12. Acknowledgements

I grateful to Joeseph W. Liu for detailed comments on the first half of this paper. I am indebted to the Mathematical and Computational Sciences Division of the National Institute of Standards and Technology for the use of their research facilities.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison–Wesley, Reading, MA, 1974.

[2] J. R. Bunch and D. J. Rose, editors. *Sparse Matrix Computations*, New York, 1976. Academic Press.

[3] A. Chang. Application of sparse matrix methods in electric power system analysis. In R. A. Willoughby, editor, *Sparse Matrix Proceedings*, pages 113–122, Yorktown Heights, 1969. IBM. Technical Report RQ 1 (#11707).

[4] I. S. Duff, editor. *Sparse Matrices and their Uses*, New York, 1981. IMA Numerical Analysis Group Conference, Reading, 1980, Academic Press. 1981.

[5] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, Oxford, 1986.

[6] I. S. Duff and G. W. Stewart, editors. *Sparse Matrix Proceedings 1978*, Philadelphia, 1979. SIAM.

[7] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman. Yale sparse matrix package I: The symmetric codes. *International Journal for Numerical Methods in Engineering*, 18:1145–1151, 1982.

[8] J. A. George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10:345–363, 1973.

[9] J. A. George. Direct solution of sparse positive definite systems: Some baisic ideas and open problems. In I. S. Duff, editor, *Sparse Matrices and Their Uses*, pages 283–306, New York, 1981. Academic Press.

[10] J. A. George and J. W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice–Hall, Englewood Cliffs, NJ, 1981.

[11] A. J. Hoffman, M. S. Martin, and D. J. Rose. Complexity bounds for regular finite difference and finite element grids. *SIAM Journal on Numerical Analysis*, 10:364–369, 1973.

[12] J. H. W. Liu. A compact row storage scheme for cholesky factors. *ACM Transactions on Mathematical Software*, 12:127–148, 1986.

[13] J. H. W. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11:134–172, 1990.

[14] J. H. W. Liu. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM Review*, 34:82–109, 1992.

[15] J. W. H. Liu, E. Ng, and B. W. Peyton. On finding supernoes for sparse matrix computations. *SIAM Journal on Matrix Analysis and Applications*, 14:242–252, 1993.

[16] S. V. Parter. The use of linear graphs in Gauss elimination. *SIAM Review*, 3:119–130, 1961.

[17] J. K. Reid, editor. *Large Sparse Sets of Linear Equations*, New York, 1971. Academic Press.

[18] D. J. Rose, R. E. Tarjan, and G. S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, 5:266–283, 1976.

[19] D. J. Rose and R. A. Willoughby, editors. *Sparse Matrices and Their Applications*, New York, 1972. Plenum Press.

[20] Y. Saad. SPARSEKIT: A basic tool kit for sparse matrix computations. Available at `http://www-users.cs.umn.edu/ saad/software/SPARSKIT/sparskit.html`, 1994.

[21] Y. Saad. *Iterative Methods for Sparse Linear Systems.* SIAM, Philadelphia, 2003.

[22] N. Sato and W. F. Tinney. Techniques for exploiting the sparsity of the network admittance matrix. *IEEE Transactions on Power Apparatus and Systems*, 82:944–950, 1963.

[23] R. Schreiber. A new implementation of sparse gaussian elimination. *ACM Transactions on Mathematical Software*, 8:256f–276, 1982.

[24] G. W. Stewart. *Matrix Algorithms I: Basic Decompositions.* SIAM, Philadelphia, 1998.

[25] R. E. Tarjan. *Network Structures and Network Algorithms.* SIAM, Philadelphia, 1983.

[26] W. F. Tinney and J. W. Walker. Direct solution of sparse network equations by optimally ordered triangular factorization. *Proc. IEEE*, 55:1801–1809, 1967.

[27] R. A. Willoughby (Editor). Sparse matrix proceedings. Report RA1 (#11707), IBM Research, Yorktown Heights., 1968.