Chapter 4
Cholesky factorization

The *Cholesky factorization* of a sparse symmetric positive definite matrix A is the product $A = LL^T$, where L is a lower triangular matrix with positive entries on its diagonal. Entries in L that do not appear in A are called *fill-in*. Let G_{L+L^T} be the undirected graph of $L + L^T$; it is called the *filled graph* of A. The structure of G_{L+L^T} is given by the following theorem.

Theorem 4.1 (Rose, Tarjan, and Lueker [175]). *The edge (i, j) is in the undirected graph G_{L+L^T} of $L+L^T$ if and only if there exists a path $i \rightsquigarrow j$ in the undirected graph of A where all nodes in the path except i and j are numbered less than* $\min(i, j)$.

Numeric Cholesky factorization is typically preceded by a *symbolic analysis* step that determines either G_{L+L^T} or some of its key properties. The goal is to keep the numeric factorization as simple as possible in terms of time complexity, memory usage, and clarity of code. The analysis step presented here finds the *elimination tree*, computes its *postordering*, and then computes the *column counts*, which are the number of nonzeros in each column of L. Some numeric Cholesky factorization algorithms also need the nonzero pattern of L.

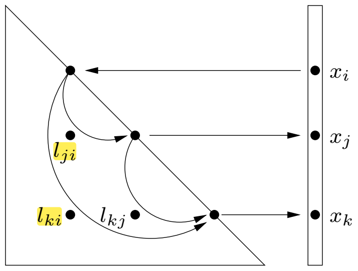
There are many ways to compute the Cholesky factorization $A = LL^T$ and the graph G_{L+L^T}. The sparse triangular solve, $Lx = b$, forms a common thread throughout this book; it is used as the basis of an *up-looking* sparse Cholesky factorization described here. Consider a 2-by-2 block decomposition $LL^T = A$,

1

$$\begin{bmatrix} L_{11} & \\ l_{12}^T & l_{22} \end{bmatrix} \begin{bmatrix} L_{11}^T & l_{12} \\ & l_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & a_{12} \\ a_{12}^T & a_{22} \end{bmatrix}, \quad (4.1)$$

L11 L11T, L11 l12
l12T L11T l12T l12 + l22 l22

where L_{11} and A_{11} are $(n-1)$-by-$(n-1)$. The three equations that lead to the up-looking Cholesky factorization algorithm are $L_{11}L_{11}^T = A_{11}$, $L_{11}l_{12} = a_{12}$, and $l_{12}^T l_{12} + l_{22}^2 = a_{22}$. The first equation can be solved recursively to obtain L_{11}, followed by a sparse triangular solve using the second equation to compute l_{12}. Finally, a sparse dot product and scalar square root, $l_{22} = \sqrt{a_{22} - l_{12}^T l_{12}}$, result in l_{22}. If the matrix is positive definite, then $a_{22} > l_{12}^T l_{12}$ holds, and the Cholesky



Thm. 4.2: $a_{ik} \neq 0$ implies $l_{ki} \neq 0$
Thm. 4.3: $l_{ji} \neq 0$ and $l_{ki} \neq 0$ implies $l_{kj} \neq 0$
Thus l_{ki} redundant for $\mathcal{X} = \text{Reach}_L(i)$

2

Figure 4.1. *Pruning the directed graph G_L yields the elimination tree $\mathcal{T}$*

factorization exists. The nonrecursive version of this algorithm is demonstrated by `chol_up` below. It is called up-looking because it looks up (accessing L_{11} or rows 1 to $k-1$ of L) to construct the kth row of L.

```
function L = chol_up (A)
n = size (A) ;
L = zeros (n) ;
for k = 1:n
    L (k,1:k-1) = (L (1:k-1,1:k-1) \ A (1:k-1,k))' ;
    L (k,k) = sqrt (A (k,k) - L (k,1:k-1) * L (k,1:k-1)') ;
end
```

4.1 Elimination tree

The *elimination tree* is even more important to sparse matrix algorithms than the sparse triangular solve. It appears in many algorithms and many theorems, both in this book and elsewhere. One motivation for deriving the tree is to reduce the time required to compute the graph reachability (Theorem 3.1) for the sparse triangular solve presented in Section 3.2. This triangular solve is required by the up-looking sparse Cholesky factorization algorithm just described in the previous section. **Goal!**

Consider (4.1). The vector l_{12} is computed with a sparse triangular solve, $L_{11}l_{12} = a_{12}$, and its transpose becomes the kth row of L. Its nonzero pattern is thus $\mathcal{L}_k = \text{Reach}_{G_{k-1}}(\mathcal{A}_k)$, where G_{k-1} is the directed graph of L_{11}, $\mathcal{L}_k$ denotes the nonzero pattern of the kth row of L, and $\mathcal{A}_k$ denotes the nonzero pattern of the upper triangular part of the kth column of A. **A_k should contain k!** **Save some works!**

The depth-first search of G_{k-1} is sufficient for computing $\mathcal{L}_k$, but a simpler method exists, taking only $O(|\mathcal{L}_k|)$ time. Consider any $i < j < k$, where $l_{ji} \neq 0$ and $a_{ik} \neq 0$, as shown in Figure 4.1. A graph traversal of G_{k-1} will start at node i. Thus, $i \in \mathcal{L}_k$, the nonzero pattern of the solution to the triangular system. This becomes the kth row of L, and thus $l_{ki} \neq 0$. The traversal will visit node j because of the edge (i,j) (corresponding to the nonzero l_{ji}). Thus, $j \in \mathcal{L}_k$, and $l_{kj} \neq 0$; two nonzeros in column i (l_{ji} and l_{ki}) imply that l_{kj} is nonzero also. In terms of the directed graph G_L (not just the graph of L_{11}), edges (i,j) and (i,k) imply edge (j,k). These facts are summarized in the following theorems that completely define the nonzero pattern of the Cholesky factor L. **L_k should contain k!** 1

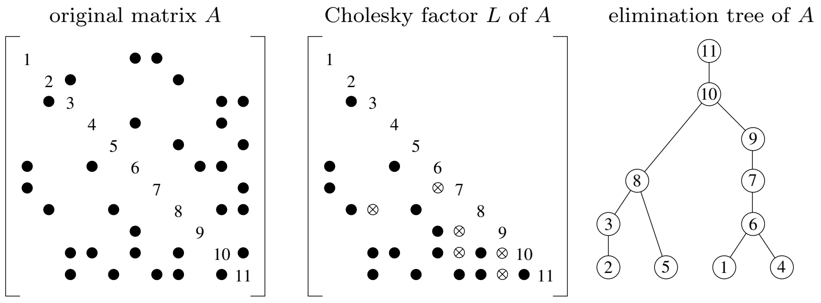


Figure 4.2. *Example matrix A, factor L, and elimination tree*

Theorem 4.2. *For a Cholesky factorization $LL^T = A$, and neglecting numerical cancellation, $a_{ij} \neq 0 \Rightarrow l_{ij} \neq 0$. That is, if a_{ij} is nonzero, then l_{ij} will be nonzero as well.*

Theorem 4.3 (Parter [165])**.** *For a Cholesky factorization $LL^T = A$, and neglecting numerical cancellation, $i < j < k \wedge l_{ji} \neq 0 \wedge l_{ki} \neq 0 \Rightarrow l_{kj} \neq 0$. That is, if both l_{ji} and l_{ki} are nonzero where $i < j < k$, then l_{kj} will be nonzero as well.* 2

why? Through up-looking Cholesky

Since there is a path from i to k via j that does not traverse the edge (i, k), the edge (i, k) is not needed to compute Reach(i). The set Reach(t) for any other node $t < i$ with a path $t \rightsquigarrow i$ is also not affected if (i, k) is removed from the directed graph G_L. This removal of edges leaves at most one outgoing edge from node i in the pruned graph, all the while not affecting Reach(i). If $j > i$ is the least numbered node for which $l_{ji} \neq 0$, all other nonzeros l_{ki} where $k > j$ are redundant.

The result is the elimination tree. The parent of node i in the tree is j, where the first off-diagonal nonzero in column i has row index j (the smallest $j > i$ for which $l_{ji} \neq 0$). Node i is a root of the tree if column i has no off-diagonal nonzero entries; it has no parent. The tree may actually be a forest, with multiple roots, if the graph of A consists of multiple connected components (there will be one tree per component of A). By convention, it is still called a tree. Assume the edges of the tree are directed, from a child to its parent. Let $\mathcal{T}$ denote the elimination tree of L, and let $\mathcal{T}_k$ denote the elimination tree of submatrix $L_{1...k,1...k}$, the first k rows and columns of L. An example matrix A, its Cholesky factor L, and its elimination tree $\mathcal{T}$ are shown in Figure 4.2. In the factor L, fill-in (entries that appear in L but not in A) are shown as circled x's. Diagonal entries are numbered for easy reference.

The existence of the elimination tree has been shown; it is now necessary to compute it. A few more theorems are required.

Theorem 4.4 (Schreiber [181])**.** *For a Cholesky factorization $LL^T = A$, and neglecting numerical cancellation, $l_{ki} \neq 0$ and $k > i$ imply that i is a descendant of k in the elimination tree $\mathcal{T}$; equivalently, $i \rightsquigarrow k$ is a path in $\mathcal{T}$.*

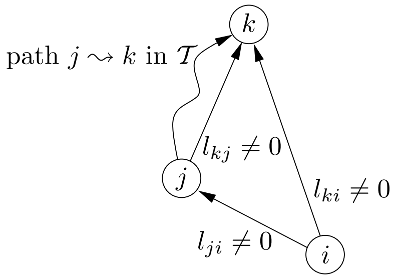


Figure 4.3. *Illustration of Theorem* 4.4

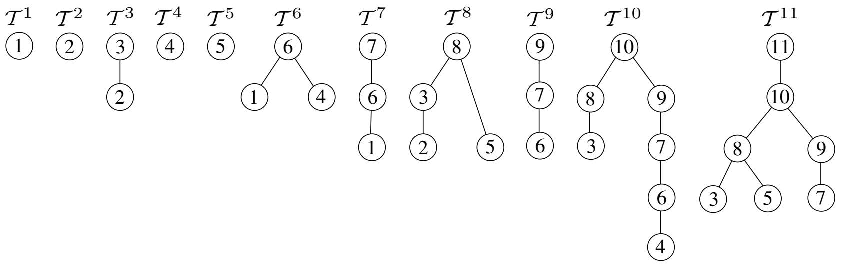


Figure 4.4. *Row subtrees of the example in Figure* 4.2

Proof. Refer to Figure 4.3. The proof is by induction on i for a fixed k. Let $j = \min\{j \mid l_{ji} \neq 0 \wedge j > i\}$ be the parent of i. The parent $j > i$ must exist because $l_{ki} \neq 0$. For the base case, if $k = j$, then k is the parent of i and thus $i \rightsquigarrow k$ is a path in $\mathcal{T}$. For the inductive step, $k > j > i$ must hold, and there are thus two nonzero entries l_{ki} and l_{ji}. From Theorem 4.3, $l_{kj} \neq 0$. By induction, l_{kj} implies the path $j \rightsquigarrow k$ exists in $\mathcal{T}$. Combined with the edge (i, j), this means there is a path $i \rightsquigarrow k$ in $\mathcal{T}$. $\square$

Removing edges from the directed graph G_L to obtain the elimination tree $\mathcal{T}$ does not affect the Reach(i) of any node. The result is the following theorem.

Theorem 4.5 (Liu [148]). *The nonzero pattern $\mathcal{L}_k$ of the kth row of L is given by*

$$\mathcal{L}_k = Reach_{G_{k-1}}(\mathcal{A}_k) = Reach_{\mathcal{T}_{k-1}}(\mathcal{A}_k). \tag{4.2}$$

3

T_k, T^k different notations!

Theorem 4.5 defines $\mathcal{L}_k$. The kth row subtree, denoted $\mathcal{T}^k$, is the subtree of $\mathcal{T}$ induced by the nodes in $\mathcal{L}_k$. The 11 row subtrees $\mathcal{T}^1, \ldots, \mathcal{T}^{11}$ of the matrix shown in Figure 4.2 are shown in Figure 4.4. Each row subtree is characterized by its leaves, which correspond to entries in A. This fact is summarized by the following theorem.

L1={1}, L2={2}, L3={3,2}, L4={4}, L5={5}, L6={6, 4, 1}

Theorem 4.6 (Liu [148]). *Node j is a leaf of $\mathcal{T}^k$ if and only if both $a_{jk} \neq 0$ and $a_{ik} = 0$ for every descendant i of j in the elimination tree $\mathcal{T}$.* 3

A corollary to Theorem 4.4 fully characterizes the elimination tree $\mathcal{T}$.

Corollary 4.7 (Liu [148]). *For a Cholesky factorization $LL^T = A$, and neglecting numerical cancellation, $a_{ki} \neq 0$ and $k > i$ imply that i is a descendant of k in the elimination tree $\mathcal{T}$; equivalently, $i \rightsquigarrow k$ is a path in $\mathcal{T}$.* important! 3

Theorem 4.4 and Corollary 4.7 lead to an algorithm that computes the elimination tree $\mathcal{T}$ in almost $O(|A|)$ time. Suppose $\mathcal{T}_{k-1}$ is known. This tree is a subset of $\mathcal{T}_k$. To compute $\mathcal{T}_k$ from $\mathcal{T}_{k-1}$, the children of k (which are root nodes in $\mathcal{T}_{k-1}$) must be found. Since $a_{ki} \neq 0$ implies the path $i \rightsquigarrow k$ exists in $\mathcal{T}$, this path can be traversed in $\mathcal{T}_{k-1}$ until reaching a root node. This node must be a child of k, since the path $i \rightsquigarrow k$ must exist. How to implement?

To speed up the traversal of the partially constructed elimination tree $\mathcal{T}_{k-1}$, a set of *ancestors* is kept. The ancestor of i, ideally, would simply be the root r of the partially constructed tree $\mathcal{T}_{k-1}$ that contains i. Traversing the path $i \rightsquigarrow r$ would take $O(1)$ time, simply by finding the ancestor r of i. This goal can nearly be met by a *disjoint-set-union* data structure. An optimal one would result in a total time complexity of $O(|A|\alpha(|A|, n))$ for the $|A|$ path traversals that need to be made, where $\alpha(|A|, n)$ is the inverse Ackermann function, a very slowly growing function. However, a simpler method is used that leads to an $O(|A| \log n)$ time algorithm. The $\log n$ upper bound is never reached in practice, however, and the resulting algorithm takes practically $O(|A|)$ time and is faster (in practice, not asymptotically) than the $O(|A|\alpha(|A|, n))$-time disjoint-set-union algorithm. The time complexity of `cs_etree` is called *nearly* $O(|A|)$ time. 4

The `cs_etree` function computes the elimination tree of the Cholesky factorization of A (assuming `ata` is false), using just `A` and returning the `int` array `parent`, of size `n`. It iterates over each column `k` and considers every entry a_{ik} in the upper triangular part of A. It updates the tree, following the path from `i` to the root of the tree. Rather than following the path via the `parent` array, an array `ancestor` is kept, where `ancestor[i]` is the highest known ancestor of `i`, not necessarily the root of the tree in $\mathcal{T}_{k-1}$ containing `i`. If `r` is a root, it has no ancestor (`ancestor[r]` is `-1`). Since the path is guaranteed to lead to node `k` in $\mathcal{T}_k$, the ancestors of all nodes along this path are set to `k` (*path compression*). If a root node is reached in $\mathcal{T}_{k-1}$ that is not `k`, it must be a child of `k` in $\mathcal{T}_k$; `parent` is updated to reflect this. upper here!

If the input parameter `ata` is true, `cs_etree` computes the elimination tree of A^TA without forming A^TA. This is the *column elimination tree*. It will be used in the QR and LU factorization algorithms in Chapters 5 and 6. Row i of A creates a dense submatrix, or clique, in the graph of A^TA. Rather than using the graph of A^TA (with one node corresponding to each column of A), a new graph is constructed dynamically (also with one node per column of A). If the nonzero pattern of row i contains column indices $j_1, j_2, j_3, j_4, \ldots$, the new graph is given edges (j_1, j_2), (j_2, j_3), (j_3, j_4), and so on. Each row i creates a path in this new graph. In the tree, these edges ensure $j_1 \rightsquigarrow j_2 \rightsquigarrow j_3 \rightsquigarrow j_4 \ldots$ is a path in $\mathcal{T}$.

The clique in A^TA has edges between all nodes $j_1, j_2, j_3, j_4, \ldots$ and will have the same ancestor/descendant relationship. Thus, the elimination tree of A^TA and this new graph will be the same. The path is constructed dynamically as the algorithm progresses, using the `prev` array. `prev[i]` starts out equal to `-1` for all `i`. Let $\mathcal{A}_k$ be the nonzero pattern of `A(:,k)`. When column `k` is considered, the edge (`prev[i]`,`k`) is created for each $i \in \mathcal{A}_k$. This edge is used to update the elimination tree, traversing from `prev[i]` up to `k` in the tree. After this traversal, `prev[i]` is set to `k`, to prepare for the next edge in this row `i`, for a subsequent column in the outer `for k` loop.

```
int *cs_etree (const cs *A, int ata)
{                                                                    ata=False!
    int i, k, p, m, n, inext, *Ap, *Ai, *w, *parent, *ancestor, *prev ;
    if (!CS_CSC (A)) return (NULL) ;        /* check inputs */
    m = A->m ; n = A->n ; Ap = A->p ; Ai = A->i ;
    parent = cs_malloc (n, sizeof (int)) ;              /* allocate result */
    w = cs_malloc (n + (ata ? m : 0), sizeof (int)) ;   /* get workspace */
    if (!w || !parent) return (cs_idone (parent, NULL, w, 0)) ;
    ancestor = w ; prev = w + n ;
    if (ata) for (i = 0 ; i < m ; i++) prev [i] = -1 ;          How to represent a tree?
    for (k = 0 ; k < n ; k++)     column
    {
        parent [k] = -1 ;                   /* node k has no parent yet */
        ancestor [k] = -1 ;                 /* nor does k have an ancestor */
        for (p = Ap [k] ; p < Ap [k+1] ; p++)    get current column
        {
            i = ata ? (prev [Ai [p]]) : (Ai [p]) ;
            for ( ; i != -1 && i < k ; i = inext)   /* traverse from i to k */
            {
                inext = ancestor [i] ;              /* inext = ancestor of i */
                ancestor [i] = k ;                  /* path compression */
                if (inext == -1) parent [i] = k ;   /* no anc., parent is k */
            }
            if (ata) prev [Ai [p]] = k ;
        }
    }
    return (cs_idone (parent, NULL, w, 1)) ;
}
```

The `cs_idone` function used by `cs_etree` returns an `int` array and frees any workspace.

```
int *cs_idone (int *p, cs *C, void *w, int ok)
{
    cs_spfree (C) ;                         /* free temporary matrix */
    cs_free (w) ;                           /* free workspace */
    return (ok ? p : cs_free (p)) ;         /* return result if OK, else free it */
}
```

The MATLAB statement `parent=etree(A)` computes the elimination tree of a symmetric matrix `A`, represented as a size-`n` array; `parent(i)=k` if `k` is the parent of `i`. To compute the column elimination tree, a second parameter is added: `parent=etree(A,'col')`. Both algorithms in MATLAB use the same method as in `cs_etree` (using `cholmod_etree`).

4.2 Sparse triangular solve

Solving $Lx = b$ when L, x, and b are sparse has already been discussed in the general case in Section 3.2. Here, L is a more specific lower triangular matrix, arising from a Cholesky factorization. The set $\text{Reach}_L(i)$ for a Cholesky factor L can be computed in time proportional to the size of the set via a simple tree traversal that starts at node i and traverses the path to the root of the elimination tree. This is much less than the time to compute $\text{Reach}(i)$ in the general case.

The `cs_ereach` function computes the nonzero pattern of the kth row of L, $\mathcal{L}_k = \text{Reach}_{k-1}(\mathcal{A}_k)$, where Reach_{k-1} denotes the graph of `L(1:k-1,1:k-1)`. It uses the elimination tree $\mathcal{T}$ of L to traverse the kth row subtree, $\mathcal{T}^k$. The set $\mathcal{A}_k$ denotes the nonzero pattern of the kth column of the upper triangular part of A. This is the same as the kth row of the lower triangular part of A, which is how the kth row subtree $\mathcal{T}^k$ is traversed. The first part of the code iterates for each i in this set $\mathcal{A}_k$. Next, the path from i towards the root of the tree is traversed. The traversal marks the nodes and stops if it encounters a marked node. Nodes are marked using the `w` array of size `n`; all entries in `w` must be greater than or equal to zero on input, and the contents of `w` are restored on output. This subpath is then pushed onto the output stack. On output, the set $\mathcal{L}_k$ is contained in `s[top]` through `s[n-1]` (except for the diagonal entry). It is created in topological order.

```
int cs_ereach (const cs *A, int k, const int *parent, int *s, int *w)
{
    int i, p, n, len, top, *Ap, *Ai ;
    if (!CS_CSC (A) || !parent || !s || !w) return (-1) ;   /* check inputs */
    top = n = A->n ; Ap = A->p ; Ai = A->i ;
    CS_MARK (w, k) ;                /* mark node k as visited */
    for (p = Ap [k] ; p < Ap [k+1] ; p++)
    {
        i = Ai [p] ;                /* A(i,k) is nonzero */
        if (i > k) continue ;       /* only use upper triangular part of A */
        for (len = 0 ; !CS_MARKED (w,i) ; i = parent [i]) /* traverse up etree*/
        {
            s [len++] = i ;         /* L(k,i) is nonzero */
            CS_MARK (w, i) ;        /* mark i as visited */
        }
        while (len > 0) s [--top] = s [--len] ; /* push path onto stack */
    }
    for (p = top ; p < n ; p++) CS_MARK (w, s [p]) ;    /* unmark all nodes */
    CS_MARK (w, k) ;                /* unmark node k */
    return (top) ;                  /* s [top..n-1] contains pattern of L(k,:)*/
}
```

The total time taken by the algorithm is $O(|\mathcal{L}_k|)$, the number of nonzeros in row k of L. This is much faster in general than the `cs_reach` function that computes $\text{Reach}_L(\mathcal{B})$ for an arbitrary lower triangular L. Solving $Lx = b$ is an integrated part of the up-looking Cholesky factorization; a stand-alone $Lx = b$ solver when L is a Cholesky factor is left as an exercise.

The `cs_ereach` function can be used to construct the elimination tree itself by extending the tree one node at a time. The time taken would be $O(|L|)$. As a by-product, the entries in L are created one at a time. They can be kept to obtain the nonzero pattern of L, or they can be counted and discarded to obtain a count of nonzeros in each row and column of L. The latter function is done more efficiently with `cs_post` and `cs_counts`, discussed in the next three sections.

MATLAB uses a code similar to `cs_ereach` in its sparse Cholesky factorization methods (an up-looking sparse Cholesky `cholmod_rowfac` and a supernodal symbolic factorization, `cholmod_super_symbolic`). However, it cannot use the tree when computing `x=L\b`, for several reasons. The elimination tree is discarded after `L` is computed. MATLAB does not keep track of how `L` was computed, and `L` may be modified prior to using it in `x=L\b`. It may be an arbitrary sparse lower triangular system, whose nonzero pattern is not governed by the tree. Numerically zero entries are dropped from `L`, so even if `L` is not modified by the application, the tree cannot be determined from the first off-diagonal entry in each column of `L`. For these reasons, the MATLAB statement `x=L\b` determines only that `L` is sparse and lower triangular (see Section 8.5) and uses an algorithm much like `cs_lsolve`.

4.3 Postordering a tree

Once the elimination tree is found, its *postordering* can be found. A postorder of the tree is required for computing the number of nonzeros in each column of L in the algorithm presented in the next section. It is also useful in its own right. If A is permuted according to the postordering P (`C=A(p,p)` in MATLAB notation), then the number of nonzeros in $LL^T = C$ is unchanged, but the nonzero pattern of L will be more structured and the numerical factorization will often be faster as a result, even with no change to the factorization code.

Theorem 4.8 (Liu [150]). *The filled graphs of A and PAP^T are isomorphic if P is a postordering of the elimination tree of A. Likewise, the elimination trees of A and PAP^T are isomorphic.*

traverse the tree!

In a postordered tree, the d proper descendants of any node k are numbered $k-d$ through $k-1$. If `post` represents the postordering permutation, `post[k]=i` means node `i` of the original tree is node `k` of the postordered tree. The most natural postordering preserves the relative ordering of the children of each node; if nodes $c_1 < c_2 < \cdots < c_t$ are the t children of node k, then `post`$[c_1] <$ `post`$[c_2] < \cdots <$ `post`$[c_t]$. A recursive depth-first search of the tree can compute the postordering:

```
function postorder ($\mathcal{T}$)
    $k = 0$
    for each root node $j$ of $\mathcal{T}$ do
        dfstree ($j$)

function dfstree ($j$)
    for each child $i$ of $j$ do
        dfstree ($i$)
    post[$k$] = $j$
    $k = k + 1$
```

5

However, the depth of the elimination tree can easily be $O(n)$, causing stack overflow for large matrices. A nonrecursive implementation is better, as shown in the cs_post function below.

```
int *cs_post (const int *parent, int n)
{
    int j, k = 0, *post, *w, *head, *next, *stack ;
    if (!parent) return (NULL) ;                        /* check inputs */
    post = cs_malloc (n, sizeof (int)) ;                /* allocate result */
    w = cs_malloc (3*n, sizeof (int)) ;                 /* get workspace */
    if (!w || !post) return (cs_idone (post, NULL, w, 0)) ;
    head = w ; next = w + n ; stack = w + 2*n ;
    for (j = 0 ; j < n ; j++) head [j] = -1 ;           /* empty linked lists */
    for (j = n-1 ; j >= 0 ; j--)            /* traverse nodes in reverse order*/
    {
        if (parent [j] == -1) continue ;    /* j is a root */
        next [j] = head [parent [j]] ;      /* add j to list of its parent */
        head [parent [j]] = j ;
    }
    for (j = 0 ; j < n ; j++)
    {
        if (parent [j] != -1) continue ;    /* skip j if it is not a root */
        k = cs_tdfs (j, k, head, next, post, stack) ;
    }
    return (cs_idone (post, NULL, w, 1)) ;  /* success; free w, return post */
}

int cs_tdfs (int j, int k, int *head, const int *next, int *post, int *stack)
{
    int i, p, top = 0 ;
    if (!head || !next || !post || !stack) return (-1) ;    /* check inputs */
    stack [0] = j ;                 /* place j on the stack */
    while (top >= 0)                /* while (stack is not empty) */
    {
        p = stack [top] ;           /* p = top of stack */
        i = head [p] ;              /* i = youngest child of p */
        if (i == -1)
        {
            top-- ;                 /* p has no unordered children left */
            post [k++] = p ;        /* node p is the kth postordered node */
        }
        else
        {
            head [p] = next [i] ;   /* remove i from children of p */
            stack [++top] = i ;     /* start dfs on child node i */
        }
    }
    return (k) ;
}
```

First, workspace is allocated, and a set of n linked lists is initialized. The jth linked list contains a list of all the children of node j in ascending order. Next, nodes j from 0 to n-1 are traversed, corresponding to the **for** j loop in the *postorder* pseudocode. If j is a root, a depth-first search of the tree is performed, using cs_tdfs. The cs_tdfs function places the root j on a stack. Each iteration of the while loop considers the node p at the top of the stack. If it has no unordered children

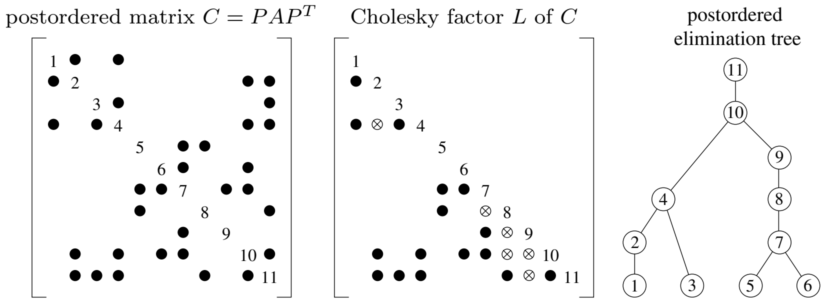

postordered matrix $C = PAP^T$
Cholesky factor L of C
postordered elimination tree

Figure 4.5. *After elimination tree postordering*

left, it is removed from the stack and ordered as the kth node in the postordering. Otherwise, its youngest unordered child `i` is removed from the head of the pth linked list and placed on the stack. The next iteration of the `while` loop will commence the depth-first search at this node `i`.

The `cs_post` function takes as input the elimination tree $\mathcal{T}$ represented as the `parent` array of size `n`. The `parent` array is not modified. The function returns a pointer to an integer vector `post`, of size `n`, that contains the postordering. The total time taken by the postordering is $O(n)$.

Figure 4.5 illustrates the matrix PAP^T, its Cholesky factor, and its elimination tree, where P is the postordering of the elimination tree in Figure 4.2.

The MATLAB statement `[parent,post] = etree(A)` computes the elimination tree and its postordering, using the same algorithms (`cholmod_etree` and `cholmod_postorder`). Node `i` is the `k`th node in the postordered tree if `post(k)=i`. A matrix can be permuted with this postordering via `C=A(post,post)`; the number of nonzeros in `chol(A)` and `chol(C)` will be the same. Looking ahead, Chapter 7 discusses how to find a fill-reducing ordering, `p`, where the permuted matrix is `A(p,p)`. This permutation vector `p` can be combined with the postordering of `A(p,p)`, using the following MATLAB code:

```
[parent, post] = etree (A (p,p)) ;
p = p (post) ;
```

4.4 Row counts

The *row counts* of a Cholesky factorization are the numbers of nonzeros in each row of L. The numeric factorization presented in Section 4.7 requires the *column counts* (the number of nonzeros in each column of L), not the row counts. However, the row count algorithm is simpler, and many of the ideas and theorems needed to understand the simpler row count algorithm are used in the column count algorithm.

A simple (yet nonoptimal) method of computing both the row and column counts is to traverse each row subtree. For each row i, consider each nonzero a_{ij}. Traverse up the tree from node j marking each node and stop if node i or a marked node is found (the marks must be cleared for each row i). The row count for

original idea

6

row i is the number of nodes in the row subtree $\mathcal{T}^i$, and the column counts can be accumulated while traversing each node of the ith row subtree. The number of nonzeros in column j of L is the number of row subtrees that contain node j. However, this method requires $O(|L|)$ time. The goal of this section and the following one is to show how to compute the row and column counts in nearly $O(|A|)$ time.

To reduce the time complexity to nearly $O(|A|)$, five concepts must be introduced: (1) the *least common ancestor* of two nodes, (2) *path decomposition*, (3) the *first descendant* of each node, (4) the *level* of a node in the elimination tree, and (5) the *skeleton matrix*. The basic idea is to decompose each row subtree into a set of disjoint paths, each starting with a leaf node and terminating at the least common ancestor of the current leaf and the prior leaf node. The paths are not traversed one node at a time. Instead, the length of these paths are found via the difference in the levels of their starting and ending nodes, where the level of a node is its distance from the root. The row count algorithm exploits the fact that all subtrees are related to each other; they are all subtrees of the elimination tree.

tree only knows his parent!

The first step in the row count algorithm is to find the level and first descendant of each node of the elimination tree. The *first descendant* of a node j is the smallest postordering of any descendant of j. The first descendant and level of each node of the tree can be easily computed in $O(n)$ time by the `firstdesc` function below.

one node may have multiple descendants!

level=distance from the root

```
void firstdesc (int n, int *parent, int *post, int *first, int *level)
{
    int len, i, k, r, s ;
    for (i = 0 ; i < n ; i++) first [i] = -1 ;
    for (k = 0 ; k < n ; k++)
    {
        i = post [k] ;      /* node i of etree is kth postordered node */
        len = 0 ;           /* traverse from i towards the root */
        for (r = i ; r != -1 && first [r] == -1 ; r = parent [r], len++)
            first [r] = k ;
        len += (r == -1) ? (-1) : level [r] ;   /* root node or end of path */
        for (s = i ; s != r ; s = parent [s]) level [s] = len-- ;
    }
}
```

6

How to implement?

A node `i` whose first descendant has not yet been computed has `first[i]` equal to `-1`. The function starts at the first node (`k=0`) in the postordered elimination tree and traverses up towards the root. All nodes `r` along this path from node zero to the root have a first descendant of zero, and `first[r]=k` is set accordingly. For `k>0`, the traversal can also terminate at a node `r` whose `first[r]` and `level[r]` have already been determined. Once the path has been found, it is retraversed to set the levels of each node along this path.

Once the first descendant and level of each node are found, the row subtree is decomposed into disjoint paths. To do this, the leaves of the row subtrees must be found. The entries corresponding to these leaves form the *skeleton matrix* $\widehat{A}$; an entry a_{ij} is defined to be in the skeleton matrix $\widehat{A}$ of A if node j is a leaf of the ith row subtree. The nonzero patterns of the Cholesky factorization of the skeleton matrix of A and the original matrix A are identical. If node j is a leaf of the ith row

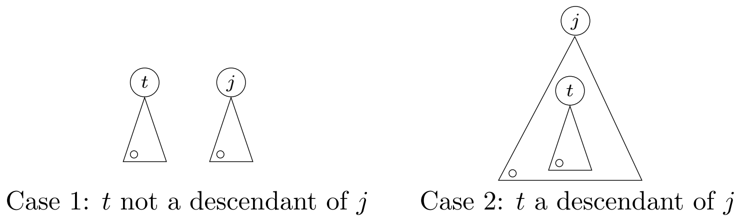


Figure 4.6. *Descendants in a postordered tree*

subtree, a_{ij} must be nonzero, but the converse is not true. For example, consider row 11 of the matrix A in Figure 4.2 and its corresponding row subtree $\mathcal{T}^{11}$ in Figure 4.4. The nonzero entries in row 11 of A are in columns 3, 5, 7, 8, 10, and 11, but only the first three are leaves of the 11th row subtree.

Suppose the matrix and the elimination tree are postordered. The first descendant of each node determines the leaves of the row subtrees, using the following *skeleton* function.

```
function skeleton
    maxfirst[$0 \dots n-1$] = $-1$
    for $j = 0$ to $n-1$ do
        for each $i > j$ for which $a_{ij} \neq 0$
            if first[$j$] > maxfirst[$i$]
                node $j$ is a leaf in the ith subtree
                maxfirst[$i$] = first[$j$]
```

The algorithm considers node j in all row subtrees i that contain node j, where j iterates from 0 to $n-1$. Let `first[j]` be the first descendant of node j in the elimination tree. Let `maxfirst[i]` be the largest `first[j]` seen so far for any nonzero a_{ij} in the ith subtree. If `first[j]` is less than or equal to `maxfirst[i]`, then node j must have a descendant $d < j$ in the ith row subtree, for which `first[d]=maxfirst[i]` will equal or exceed `first[j]`. Node j is thus not a leaf of the ith row subtree. If `first[j]` exceeds `maxfirst[i]`, then node j has no descendant in the ith row subtree, and node j is a leaf. The correctness of *skeleton* depends on Corollary 4.11 below.

Lemma 4.9. *Let $f_j \leq j$ denote the first descendant of j in a postordered tree. The descendants of j are all nodes $f_j, f_j+1, \dots, j-1, j$.*

Theorem 4.10. *Consider two nodes $t < j$ in a postordered tree. Then either (1) $f_t \leq t < f_j \leq j$ and t is not a descendant of j, or (2) $f_j \leq f_t \leq t < j$ and t is a descendant of j.*

Proof. The two cases of Theorem 4.10 are illustrated in Figure 4.6. A triangle represents the subtree rooted at a node j, and a small circle represents f_j. Case

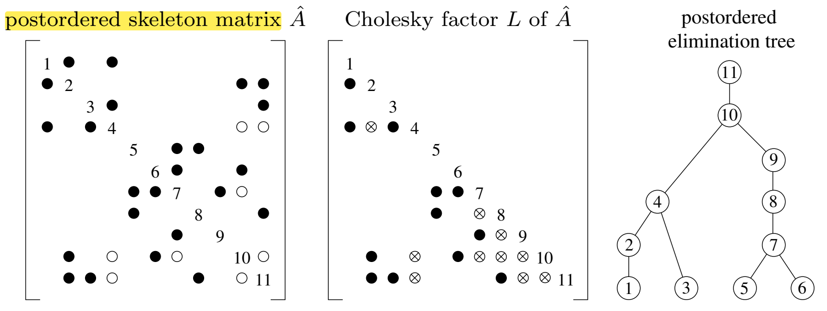

postordered skeleton matrix $\hat{A}$
Cholesky factor L of $\hat{A}$
postordered elimination tree

Figure 4.7. *Postordered skeleton matrix, its factor, and its elimination tree*

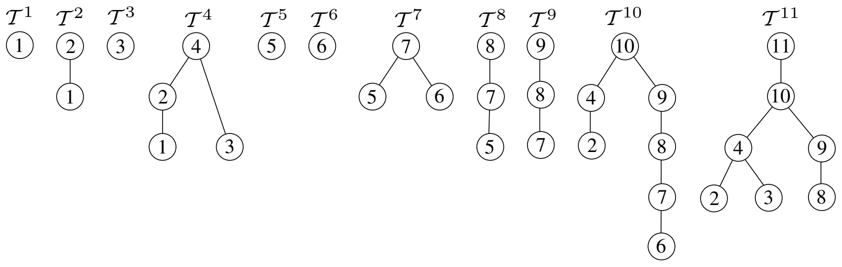


Figure 4.8. *Postordered row subtrees*

(1): Node t is not a descendant of j if and only if $t < f_j$, because of Lemma 4.9. Case (2): If t is a descendant of j, then f_t is also a descendant of j, and thus $f_j \le f_t$. If $f_j \le f_t$, then all nodes f_t through t must be descendants of j (Lemma 4.9). ☐

Corollary 4.11. *Consider a node j in a postordered tree and any set of nodes $\mathcal{S}$ where all nodes $s \in \mathcal{S}$ are numbered less than j. Let t be the node in $\mathcal{S}$ with the largest first descendant f_t. Node j has a descendant in $\mathcal{S}$ if and only if $f_t \ge f_j$.*

Figure 4.7 shows the postordered skeleton matrix of A, denoted $\widehat{A}$, its factor L, and its elimination tree (compare with Figure 4.5). Figure 4.8 shows the postordered row subtrees (compare with Figure 4.4). Entry a_{ij} (where $i > j$) is present in the skeleton matrix if and only if j is a leaf of the (postordered) ith subtree; they are shown as dark circles (the entry a_{ji} is also shown in the upper triangular part of $\widehat{A}$). A white circle denotes an entry in A that is not in the skeleton matrix $\widehat{A}$.

The leaves of the row subtree can be used to decompose the row subtree into a set of disjoint paths in a process called path decomposition. Consider the first (least numbered) leaf of a row subtree. The first disjoint path starts at this node and leads all the way to the root. Consider two consecutive leaves of a row subtree, $j_{\text{prev}} < j$. The next disjoint path starts at j and ends at the child of the least

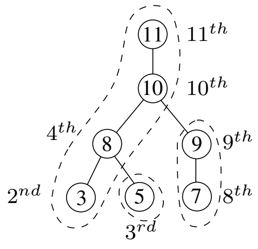


Figure 4.9. *Path decomposition of $\mathcal{T}^{11}$ from Figure* 4.4

common ancestor of j_{prev} and j. The *least common ancestor* of two nodes a and b is the least numbered node that is an ancestor of both a and b and is denoted as $q = lca(a, b)$. In $\mathcal{T}^{11}$, shown in Figure 4.9, the first path is from node 3 (the 2nd node in the postordered tree) to the root node 11. The next path starts at node 5 (the 3rd node in the postordered tree) and terminates at node 5 (node 8 is the least common ancestor of the two leaves 3 and 5). The third and last path starts at node 7 and terminates at the child node 9 of the least common ancestor (node 10) of nodes 5 and 7. Figure 4.9 shows the path decomposition of $\mathcal{T}^{11}$ into these three disjoint paths. Each node is labeled with its corresponding column index in A and its postordering (node 8 is the 4th node in the postordered tree, for example).

Once the kth row subtree is decomposed into its disjoint paths, the kth row count is computed as the sum of the lengths of these paths. An efficient method for finding the least common ancestors of consecutive leaves j_{prev} and j of the row subtree is needed. Given the least common ancestor q of these two leaves, the length of the path from j to q can be added to the row count (excluding q itself). The lengths of the paths can be found by taking the difference of the starting and ending nodes of each path.

Theorem 4.12. *Assume that the elimination tree $\mathcal{T}$ is postordered. The least common ancestor of two nodes a and b, where $a < b$, can be found by traversing the path from a towards the root. The first node $q \geq b$ found along this path is the least common ancestor of a and b.*

The `rowcnt` function takes as input the matrix A, its elimination tree, and a postordering of the elimination tree. It uses a disjoint-set-union data structure to efficiently compute the least common ancestors of successive pairs of leaves of the row subtrees. Since it is not actually part of CSparse, it does not check any out-of-memory error conditions. Unlike `cs_etree`, the function uses the lower triangular part of A only and omits an option for computing the row counts of the Cholesky factor of A^TA. The `cs_leaf` function determines if j is a leaf of the ith row subtree, $\mathcal{T}^i$. If it is, it computes the lca of j_{prev} (the previous leaf found in $\mathcal{T}^i$) and node j.

To compute $q = lca(j_{\text{prev}}, j)$ efficiently in `cs_leaf`, an ancestor of each node is maintained, using a disjoint-set-union data structure. Initially, each node is in its own set, and ancestor$[i] = i$ for all nodes i. If a node i is the root of a set, it is its own ancestor. For all other nodes i, traversing the ancestor tree and hitting a root q determines the representative (q) of the set containing node i.

```
int *rowcnt (cs *A, int *parent, int *post) /* return rowcount [0..n-1] */
{
    int i, j, k, len, s, p, jprev, q, n, sparent, jleaf, *Ap, *Ai, *maxfirst,
        *ancestor, *prevleaf, *w, *first, *level, *rowcount ;
    n = A->n ; Ap = A->p ; Ai = A->i ;                  /* get A */
    w = cs_malloc (5*n, sizeof (int)) ;                 /* get workspace */
    ancestor = w ; maxfirst = w+n ; prevleaf = w+2*n ; first = w+3*n ;
    level = w+4*n ;
    rowcount = cs_malloc (n, sizeof (int)) ;    /* allocate result */
    firstdesc (n, parent, post, first, level) ; /* find first and level */
    for (i = 0 ; i < n ; i++)
    {
        rowcount [i] = 1 ;      /* count the diagonal of L */
        prevleaf [i] = -1 ;     /* no previous leaf of the ith row subtree */
        maxfirst [i] = -1 ;     /* max first[j] for node j in ith subtree */
        ancestor [i] = i ;      /* every node is in its own set, by itself */
    }
    for (k = 0 ; k < n ; k++)
    {
        j = post [k] ;          /* j is the kth node in the postordered etree */
        for (p = Ap [j] ; p < Ap [j+1] ; p++)
        {
            i = Ai [p] ;
            q = cs_leaf (i, j, first, maxfirst, prevleaf, ancestor, &jleaf) ;
            if (jleaf) rowcount [i] += (level [j] - level [q]) ;
        }
        if (parent [j] != -1) ancestor [j] = parent [j] ;
    }
    cs_free (w) ;
    return (rowcount) ;
}

int cs_leaf (int i, int j, const int *first, int *maxfirst, int *prevleaf,
    int *ancestor, int *jleaf)
{
    int q, s, sparent, jprev ;
    if (!first || !maxfirst || !prevleaf || !ancestor || !jleaf) return (-1) ;
    *jleaf = 0 ;
    if (i <= j || first [j] <= maxfirst [i]) return (-1) ;  /* j not a leaf */
    maxfirst [i] = first [j] ;      /* update max first[j] seen so far */
    jprev = prevleaf [i] ;          /* jprev = previous leaf of ith subtree */
    prevleaf [i] = j ;
    *jleaf = (jprev == -1) ? 1: 2 ; /* j is first or subsequent leaf */
    if (*jleaf == 1) return (i) ;   /* if 1st leaf, q = root of ith subtree */
    for (q = jprev ; q != ancestor [q] ; q = ancestor [q]) ;
    for (s = jprev ; s != q ; s = sparent)
    {
        sparent = ancestor [s] ;    /* path compression */
        ancestor [s] = q ;
    }
    return (q) ;                    /* q = least common ancester (jprev,j) */
}
```

Assuming the matrix is already postordered, rowcnt considers each node j

one at a time, where j iterates from 0 to $n-1$. For each node j, all row subtrees i that contain it are considered (all row indices i corresponding to nonzero entries a_{ij}, where $i > j$). Since j_{prev} and j are leaves of the same row subtree, they will have a least common ancestor q that is greater than j and which will be the representative of the set containing j_{prev}. Traversing from node j_{prev} towards the root determines node q. After this path is traversed, it is compressed to speed up any remaining path traversals. After all row subtrees containing node j are considered, it is merged into the set corresponding to its parent. Assuming the elimination tree is connected, no nodes 0 to j are now root nodes of any set. This ensures that traversing a path in the ancestor tree will find the least common ancestor for subsequent nodes j (see Theorem 4.12).

4.5 Column counts

Computing the number of nonzeros in each column of L can be done in nearly $O(|A|)$ time, similar to the row counts. A characterization of the nonzero pattern of each column of L is required. Let $\mathcal{A}_j$ denote the nonzero pattern of the jth column of the strictly lower triangular part of A, and let $\widehat{\mathcal{A}}_j$ denote entries in the same part of the skeleton matrix $\widehat{A}$. Let $\mathcal{L}_j$ denote the nonzero pattern of column j of L, and let $c_j = |\mathcal{L}_j|$ denote the number of entries in that column (including the diagonal).

Theorem 4.13 and its corollary show how to compute the nonzero pattern of L in a left-looking manner. It states that the nonzero pattern of `L(:,j)` is the union of the nonzero patterns of the children of `j` in the elimination tree and the nonzero pattern of `A(:,j)`.

Theorem 4.13 (George and Liu [89]). *If $\mathcal{L}_j$ denotes the nonzero pattern of the jth column of L, and $\mathcal{A}_j$ denotes the nonzero pattern of the strictly lower triangular part of the jth column of A, then*

$$\mathcal{L}_j = \mathcal{A}_j \cup \{j\} \cup \left(\bigcup_{j=parent(s)} \mathcal{L}_s \setminus \{s\} \right). \tag{4.3}$$

Proof. Refer to Figure 4.10. Consider any descendant d of j and any row $i \in \mathcal{L}_d$. That is, $l_{id} \neq 0$ and the path $d \rightsquigarrow j$ exists in $\mathcal{T}$. Theorem 4.5 states that the nonzero pattern of row i is given by the ith row subtree, $\mathcal{T}^i$. Thus, the path $d \rightsquigarrow s \rightarrow j$ exists in $\mathcal{T}^i$ for some s, and row index i is present in $\mathcal{L}_j$ and in $\mathcal{L}_s$ of the child s of j (also true if $d = s$). To construct the nonzero pattern of column j ($\mathcal{L}_j$), only $\mathcal{L}_s$ of the children s of j need to be considered. Likewise, there can be no $i \in \mathcal{L}_j$ not accounted for by (4.3). If $i \in \mathcal{L}_j$, then j must be in $\mathcal{T}^i$. Either j is a leaf (and thus $i \in \widehat{\mathcal{A}}_j \subseteq \mathcal{A}_j$), or it is not a leaf (and thus j has a child s in $\mathcal{T}^i$, and $i \in \mathcal{L}_s$). $\square$

Corollary 4.14 (Schreiber [181]). *The nonzero pattern of the jth column of L is a subset of the path $j \rightsquigarrow r$ from j to the root of the elimination tree $\mathcal{T}$.*

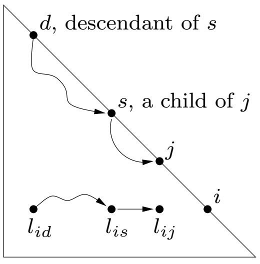


Figure 4.10. *The nonzero pattern of* `L(:,j)` *is the union of its children*

Computing the column counts $c_j = |\mathcal{L}_j|$ can be done in $O(|L|)$ time by using Theorem 4.13 or by traversing each row subtree explicitly and keeping track of how many times j appears in any row subtree. Using the least common ancestors of successive pairs of leaves of the row subtree reduces the time to nearly $O(|A|)$.

Consider the case where j is a leaf of the elimination tree $\mathcal{T}$. The column count c_j is simply $c_j = |\mathcal{A}_j| + 1 = |\widehat{\mathcal{A}}_j| + 1$, since j has no children and each entry in column j of A is also in the skeleton matrix $\widehat{A}$. Consider the case where j is not a leaf of the elimination tree. Theorem 4.13 states that $\mathcal{L}_j$ is the union of $\mathcal{A}_j \cup \{j\}$ and the nonzero patterns of its children, $\mathcal{L}_s \setminus \{s\}$. Since $\widehat{\mathcal{A}}_j$ is disjoint from each child $\mathcal{L}_s$, and since $s \in \mathcal{L}_s$,

$$c_j = |\widehat{\mathcal{A}}_j| + \left| \bigcup_{j=\text{parent}(s)} \mathcal{L}_s \setminus \{s\} \right| = |\widehat{\mathcal{A}}_j| - e_j + \left| \bigcup_{j=\text{parent}(s)} \mathcal{L}_s \right|,$$

where e_j is the number of children of j in the elimination tree $\mathcal{T}$. Suppose there was an efficient method for computing the *overlap* between the nonzero patterns of the children of j. That is, let o_j be the term required so that

$$c_j = |\widehat{\mathcal{A}}_j| - e_j - o_j + \sum_{j=\text{parent}(s)} c_s. \tag{4.4}$$

As an example, consider column $j = 4$ of Figure 4.7. Its two children are $\mathcal{L}_2 = \{2, 4, 10, 11\}$ and $\mathcal{L}_3 = \{3, 4, 11\}$. The skeleton matrix is empty in this column, so

$$\mathcal{L}_4 = \widehat{\mathcal{A}}_4 \cup \mathcal{L}_2 \setminus \{2\} \cup \mathcal{L}_3 \setminus \{3\} = \emptyset \cup \{4, 10, 11\} \cup \{4, 11\} = \{4, 10, 11\}.$$

The overlap $o_4 = 2$, because rows 4 and 11 each appear twice in the children. The number of children is $e_4 = 2$. Thus, $c_4 = 0 - 2 - 2 + 4 + 3 = 3$. If the overlap and the skeleton matrix are known for each column j, (4.4) can be used to compute the column counts. Note that the diagonal entry $j = 4$ does not appear in $\widehat{\mathcal{A}}_4$. Instead, $4 \in \mathcal{L}_4$ appears in each child, and the overlap accounts for all but one of these entries.

The overlap can be computed by considering the row subtrees. There are three cases to consider. Recall that the ith row subtree $\mathcal{T}^i$ determines the nonzero pattern of the ith row of L. Node j is present in the ith subtree if and only if $i \in \mathcal{L}_j$.

1. If $j \notin \mathcal{T}^i$, then $i \notin \mathcal{L}_j$ and row i does not contribute to the overlap o_j.
2. If j is a leaf of $\mathcal{T}^i$, then by definition a_{ij} is in the skeleton matrix. Row i does not contribute to the overlap o_j, because it appears in none of the children of j. Row i contributes exactly one to c_j, since $i \in \widehat{\mathcal{A}}_j$.
3. If j is not a leaf of $\mathcal{T}^i$, let d_{ij} denote the number of children of j that are in $\mathcal{T}^i$. These children are a subset of the children of j in the elimination tree $\mathcal{T}$. Row i is present in the nonzero patterns of each of these d_{ij} children. Thus, row i contributes $d_{ij} - 1$ to the overlap o_j. If j has just one child, row i appears only in that one child and there is no overlap.

Case 3, above, is the only place where the overlap needs to be considered: nodes in the row subtrees with more than one child. Refer to Figure 4.8, and consider column 4. Node 4 has two children in subtrees $\mathcal{T}^4$ and $\mathcal{T}^{11}$.

The key observation is to see that if j has d children in a row subtree, then it will be the least common ancestor of exactly $d-1$ successive pairs of leaves in that row subtree. For example, node 4 is the least common ancestor of leaves 1 and 3 in $\mathcal{T}^4$ and the least common ancestor of leaves 2 and 3 in $\mathcal{T}^{11}$. Thus, each time column j becomes a least common ancestor of any successive pair of leaves in the row count algorithm, the overlap o_j can be incremented.

These modifications can be folded into a single correction term, Δ_j. If j is a leaf of the elimination tree, $c_j = \Delta_j = |\widehat{\mathcal{A}}_j| + 1$. Otherwise, $\Delta_j = |\widehat{\mathcal{A}}_j| - e_j - o_j$. Equation (4.4) becomes

$$c_j = \Delta_j + \sum_{j=\mathrm{parent}(s)} c_s \tag{4.5}$$

for both cases. The term Δ_j is initialized as 1 if j is a leaf or 0 otherwise. It is incremented once for each entry a_{ij} in the skeleton matrix, decremented once for each child of j, and decremented once each time j becomes the least common ancestor of a successive pair of leaves in a row subtree. Once Δ_j is computed, (4.5) can be used to compute the column counts.

The column count algorithm `cs_counts` is given below. It takes as input the matrix `A`, the elimination tree (`parent`), and its postordering (`post`). It also takes a parameter `ata` that determines whether the Cholesky factor of A or A^TA is to be computed. If `ata` is nonzero, the column counts of the Cholesky factor of A^TA are computed. To compute the column counts for the Cholesky factorization of A, `ata` is zero, the `init_ata` function is not used, and the `for J` loop iterates just once with `J = j`.

Unlike `rowcnt`, the `cs_counts` function uses the upper triangular part of `A` to compute the column counts; it transposes `A` internally. None of the inputs are modified. The function returns an array of size `n` of the column counts or `NULL` on error. Compare `cs_counts` with `rowcnt`. It is quite simple to combine these two functions into a single function that computes both the row and column counts. They are split into two functions here, to introduce the concepts more gradually and because CSparse does not require the row counts.

```
#define HEAD(k,j) (ata ? head [k] : j)
#define NEXT(J)   (ata ? next [J] : -1)
```

```
static void init_ata (cs *AT, const int *post, int *w, int **head, int **next)
{
    int i, k, p, m = AT->n, n = AT->m, *ATp = AT->p, *ATi = AT->i ;
    *head = w+4*n, *next = w+5*n+1 ;
    for (k = 0 ; k < n ; k++) w [post [k]] = k ;    /* invert post */
    for (i = 0 ; i < m ; i++)
    {
        for (k = n, p = ATp[i] ; p < ATp[i+1] ; p++) k = CS_MIN (k, w [ATi[p]]);
        (*next) [i] = (*head) [k] ;     /* place row i in linked list k */
        (*head) [k] = i ;
    }
}
int *cs_counts (const cs *A, const int *parent, const int *post, int ata)
{
    int i, j, k, n, m, J, s, p, q, jleaf, *ATp, *ATi, *maxfirst, *prevleaf,
        *ancestor, *head = NULL, *next = NULL, *colcount, *w, *first, *delta ;
    cs *AT ;
    if (!CS_CSC (A) || !parent || !post) return (NULL) ;    /* check inputs */
    m = A->m ; n = A->n ;
    s = 4*n + (ata ? (n+m+1) : 0) ;
    delta = colcount = cs_malloc (n, sizeof (int)) ;    /* allocate result */
    w = cs_malloc (s, sizeof (int)) ;                   /* get workspace */
    AT = cs_transpose (A, 0) ;                          /* AT = A' */
    if (!AT || !colcount || !w) return (cs_idone (colcount, AT, w, 0)) ;
    ancestor = w ; maxfirst = w+n ; prevleaf = w+2*n ; first = w+3*n ;
    for (k = 0 ; k < s ; k++) w [k] = -1 ;      /* clear workspace w [0..s-1] */
    for (k = 0 ; k < n ; k++)                   /* find first [j] */
    {
        j = post [k] ;
        delta [j] = (first [j] == -1) ? 1 : 0 ;  /* delta[j]=1 if j is a leaf */
        for ( ; j != -1 && first [j] == -1 ; j = parent [j]) first [j] = k ;
    }
    ATp = AT->p ; ATi = AT->i ;
    if (ata) init_ata (AT, post, w, &head, &next) ;
    for (i = 0 ; i < n ; i++) ancestor [i] = i ; /* each node in its own set */
    for (k = 0 ; k < n ; k++)
    {
        j = post [k] ;          /* j is the kth node in postordered etree */
        if (parent [j] != -1) delta [parent [j]]-- ;    /* j is not a root */
        for (J = HEAD (k,j) ; J != -1 ; J = NEXT (J))   /* J=j for LL'=A case */
        {
            for (p = ATp [J] ; p < ATp [J+1] ; p++)
            {
                i = ATi [p] ;
                q = cs_leaf (i, j, first, maxfirst, prevleaf, ancestor, &jleaf);
                if (jleaf >= 1) delta [j]++ ;   /* A(i,j) is in skeleton */
                if (jleaf == 2) delta [q]-- ;   /* account for overlap in q */
            }
        }
        if (parent [j] != -1) ancestor [j] = parent [j] ;
    }
    for (j = 0 ; j < n ; j++)           /* sum up delta's of each child */
    {
        if (parent [j] != -1) colcount [parent [j]] += colcount [j] ;
    }
    return (cs_idone (colcount, AT, w, 1)) ;    /* success: free workspace */
}
```

The column count algorithm presented here can also be used for the QR and LU factorization of a square or rectangular matrix A. For QR factorization, the nonzero pattern of R is identical to L^T in the Cholesky factorization $LL^T = A^TA$ (assuming no numerical cancellation and mild assumptions discussed in Chapter 5). This same matrix R provides an upper bound on the nonzero pattern of U for an LU factorization of A. Details are presented in Chapters 5 and 6.

One method for finding the row counts of R is to compute A^TA explicitly and then find the column counts of its Cholesky factorization. This can be expensive both in time and memory. A better method taking nearly $O(|A|+n+m)$ time is to find a symmetric matrix with fewer nonzeros than A^TA but whose Cholesky factor has the same nonzero pattern as A^TA. One matrix that satisfies this property is the *star* matrix. It has $O(|A|)$ entries and can be found in $O(|A|+n+m)$ time. Each row of A defines a clique in the graph of A^TA. Let $\mathcal{A}_i$ denote the nonzero pattern of the ith row of A. Consider the lowest numbered column index k of nonzeros in row i of A; that is, $k = \min \mathcal{A}_i$. The clique in A^TA corresponding to row i of A is the set of entries $\mathcal{A}_i \times \mathcal{A}_i$. Consider an entry $(A^TA)_{ab}$, where $a \in \mathcal{A}_i$ and $b \in \mathcal{A}_i$. If both $a > k$ and $b > k$, then this entry is not needed. It can be removed without changing the nonzero pattern of the Cholesky factor of A^TA. Without loss of generality, assume $a > b$. The entries $(A^TA)_{bk}$ and $(A^TA)_{ak}$ will both be nonzero. Theorems 4.2 and 4.3 imply that l_{ab} is nonzero, regardless of whether or not $(A^TA)_{ab}$ is nonzero.

The nonzero pattern of the kth row and column of the star matrix is thus the union of all $\mathcal{A}_i$, where $k = \min \mathcal{A}_i$. Fortunately, this union need not be formed explicitly, since the row and column count algorithms (and specifically the skeleton function) implicitly ignore duplicate entries. To traverse all entries in the kth column of the star matrix, all rows $\mathcal{A}_i$, where $k = \min \mathcal{A}_i$, are considered. In `cs_counts`, this is implemented by placing each row in a linked list corresponding to its least numbered nonzero column index (using a `head` array of size `n+1` and a `next` array of size `n`).

In MATLAB, `c = symbfact(A)` uses the same algorithms given here, returning the column counts of the Cholesky factorization of A. The column counts of the Cholesky factorization of A^TA are given by `c = symbfact(A,'col')`. Both forms use the CHOLMOD function `cholmod_rowcolcounts`.

4.6 Symbolic analysis

The *symbolic analysis* of a matrix is a precursor to its numerical factorization. It includes computations that typically depend only on the nonzero pattern, not the numerical values. This allows the numerical factorization to be repeated for a sequence of matrices with identical nonzero pattern (a situation that often arises when solving nonlinear equations). *Symbolic factorization* includes the computation of an explicit representation of the nonzero pattern of the factorization; some sparse Cholesky algorithms require this. Permuting a matrix has a large impact on the amount of fill-in. Typically a fill-reducing permutation P is found so that the factorization of PAP^T is sparser than that of A. The `cs_schol` function performs the symbolic analysis for the up-looking sparse Cholesky factorization presented in the next section.

```
typedef struct cs_symbolic  /* symbolic Cholesky, LU, or QR analysis */
{
    int *pinv ;     /* inverse row perm. for QR, fill red. perm for Chol */
    int *q ;        /* fill-reducing column permutation for LU and QR */
    int *parent ;   /* elimination tree for Cholesky and QR */
    int *cp ;       /* column pointers for Cholesky, row counts for QR */
    int *leftmost ; /* leftmost[i] = min(find(A(i,:))), for QR */
    int m2 ;        /* # of rows for QR, after adding fictitious rows */
    double lnz ;    /* # entries in L for LU or Cholesky; in V for QR */
    double unz ;    /* # entries in U for LU; in R for QR */
} css ;

css *cs_schol (int order, const cs *A)
{
    int n, *c, *post, *P ;
    cs *C ;
    css *S ;
    if (!CS_CSC (A)) return (NULL) ;        /* check inputs */
    n = A->n ;
    S = cs_calloc (1, sizeof (css)) ;       /* allocate result S */
    if (!S) return (NULL) ;                 /* out of memory */
    P = cs_amd (order, A) ;                 /* P = amd(A+A'), or natural */
    S->pinv = cs_pinv (P, n) ;              /* find inverse permutation */
    cs_free (P) ;
    if (order && !S->pinv) return (cs_sfree (S)) ;
    C = cs_symperm (A, S->pinv, 0) ;        /* C = spones(triu(A(P,P))) */
    S->parent = cs_etree (C, 0) ;           /* find etree of C */
    post = cs_post (S->parent, n) ;         /* postorder the etree */
    c = cs_counts (C, S->parent, post, 0) ; /* find column counts of chol(C) */
    cs_free (post) ;
    cs_spfree (C) ;
    S->cp = cs_malloc (n+1, sizeof (int)) ; /* allocate result S->cp */
    S->unz = S->lnz = cs_cumsum (S->cp, c, n) ; /* find column pointers for L */
    cs_free (c) ;
    return ((S->lnz >= 0) ? S : cs_sfree (S)) ;
}

css *cs_sfree (css *S)
{
    if (!S) return (NULL) ;     /* do nothing if S already NULL */
    cs_free (S->pinv) ;
    cs_free (S->q) ;
    cs_free (S->parent) ;
    cs_free (S->cp) ;
    cs_free (S->leftmost) ;
    return (cs_free (S)) ;      /* free the css struct and return NULL */
}
```

cs_schol does not compute the nonzero pattern of L. First, a css structure S is allocated. For a sparse Cholesky factorization, S->pinv is the fill-reducing permutation (stored as an inverse permutation vector), S->parent is the elimination tree, S->cp is the column pointer of L, and S->lnz $= |L|$. This symbolic structure will also be used for sparse LU and QR factorizations. Next, p is found via a minimum degree ordering of $A + A^T$ (see Chapter 7) if `order` is `1`. No permutation is used if `order` is `0` (`p` is `NULL` to denote the identity permutation). This vector is inverted to obtain `pinv`. The upper triangular part of `A` is permuted to obtain `C` (`C=A(p,p)` in MATLAB notation). The elimination tree of `C` is found and postordered, and the column counts of L are found. A cumulative sum of the column counts gives both `S->cp`, the column pointers of `L`, and `S->lnz`$= |L|$. `cs_free` frees a symbolic analysis.

4.7 Up-looking Cholesky

A great deal of theory and algorithms have been covered to reach this point: a simple, concise sparse Cholesky factorization algorithm. The `cs_chol` function presented below implements the up-looking algorithm described in the preface to this chapter. To clarify the discussion, the bold paragraph headings are tied to comments in the code starting with `---`. This style is also used elsewhere in this book.

`cs_chol` computes the Cholesky factorization of `C=A(p,p)`. The input matrix `A` is assumed to be symmetric; only the upper triangular part is accessed. The function requires the symbolic analysis from `cs_schol`. It returns a numeric factorization, consisting of just the `N->L` matrix. First, workspace is allocated, the contents of the symbolic analysis are retrieved, `L` is allocated, and `A` is permuted (`E` is `A(p,p)` if `A` is permuted, or `NULL` otherwise, so that `A(p,p)` can be freed by `cs_ndone`).

Nonzero pattern of `L(k,:)`: The `k`th iteration of the `for` loop computes the `k`th row of `L`. It starts by computing the nonzero pattern of this `k`th row, placing the result in `s[top]` through `s[n-1]` in topological order. The entry `x[k]` is cleared to ensure that `x[0]` through `x[k]` are all zero. The `k`th column of `C` is scattered into the dense vector `x`. The diagonal entry of `C` is retrieved from `x`, and `x[k]` is cleared to prepare for iteration `k+1` of the outer loop.

Triangular solve: The numerical solution to $L_{0\ldots k-1,0\ldots k-1}x = C_{0\ldots k-1,k}$ is found, which defines the numerical values in the row `k` of `L`. As each entry l_{ki} is computed, the workspace `x[i]` is cleared, the numerical value l_{ki} and row index `k` are placed in column `i` of `L`, and the dot product $L_{k,0\ldots k-1}L^T_{k,0\ldots k-1}$ is added to `d`.

Compute `L(k,k)`: The `k`th diagonal entry of `L` is computed. The function frees `N` and returns if the matrix `A` is not positive definite. When the factorization finishes successfully, the workspace is freed and `N` is returned.

The time taken by `cs_chol` to compute the Cholesky factorization of a symmetric positive definite matrix is $O(f)$, the number of floating-point operations performed; $f = \sum |\mathtt{L(:,k)}|^2$. To perform a complete sparse Cholesky factorization, including a fill-reducing preordering and symbolic analysis, use `S=cs_schol(order,A)` followed by `N=cs_chol(A,S)`, where `order` is `0` to use the natural ordering or `1` to use a fill-reducing minimum degree ordering of $A + A^T$. **key figure!**

In MATLAB, the sparse Cholesky factorization `R=chol(A)` returns `R=L'`. If `A` is very sparse, it uses an up-looking algorithm (`cholmod_rowfac`, much like [29]). Otherwise, it uses a left-looking supernodal factorization (`cholmod_super_numeric`), discussed in the next section.

```
csn *cs_chol (const cs *A, const css *S)
```

```
{
    double d, lki, *Lx, *x, *Cx ;
    int top, i, p, k, n, *Li, *Lp, *cp, *pinv, *s, *c, *parent, *Cp, *Ci ;
    cs *L, *C, *E ;
    csn *N ;
    if (!CS_CSC (A) || !S || !S->cp || !S->parent) return (NULL) ;
    n = A->n ;
    N = cs_calloc (1, sizeof (csn)) ;       /* allocate result */
    c = cs_malloc (2*n, sizeof (int)) ;     /* get int workspace */
    x = cs_malloc (n, sizeof (double)) ;    /* get double workspace */
    cp = S->cp ; pinv = S->pinv ; parent = S->parent ;
    C = pinv ? cs_symperm (A, pinv, 1) : ((cs *) A) ;
    E = pinv ? C : NULL ;           /* E is alias for A, or a copy E=A(p,p) */
    if (!N || !c || !x || !C) return (cs_ndone (N, E, c, x, 0)) ;
    s = c + n ;
    Cp = C->p ; Ci = C->i ; Cx = C->x ;
    N->L = L = cs_spalloc (n, n, cp [n], 1, 0) ;    /* allocate result */
    if (!L) return (cs_ndone (N, E, c, x, 0)) ;
    Lp = L->p ; Li = L->i ; Lx = L->x ;
    for (k = 0 ; k < n ; k++) Lp [k] = c [k] = cp [k] ;
    for (k = 0 ; k < n ; k++)       /* compute L(:,k) for L*L' = C */
    {
        /* --- Nonzero pattern of L(k,:) ------------------------------------ */
        top = cs_ereach (C, k, parent, s, c) ;      /* find pattern of L(k,:) */
        x [k] = 0 ;                                 /* x (0:k) is now zero */
        for (p = Cp [k] ; p < Cp [k+1] ; p++)       /* x = full(triu(C(:,k))) */
        {
            if (Ci [p] <= k) x [Ci [p]] = Cx [p] ;
        }
        d = x [k] ;                     /* d = C(k,k) */
        x [k] = 0 ;                     /* clear x for k+1st iteration */
        /* --- Triangular solve --------------------------------------------- */
        for ( ; top < n ; top++)    /* solve L(0:k-1,0:k-1) * x = C(:,k) */
        {
            i = s [top] ;               /* s [top..n-1] is pattern of L(k,:) */
            lki = x [i] / Lx [Lp [i]] ; /* L(k,i) = x (i) / L(i,i) */
            x [i] = 0 ;                 /* clear x for k+1st iteration */
            for (p = Lp [i] + 1 ; p < c [i] ; p++)
            {
                x [Li [p]] -= Lx [p] * lki ;
            }
            d -= lki * lki ;            /* d = d - L(k,i)*L(k,i) */
            p = c [i]++ ;
            Li [p] = k ;                /* store L(k,i) in column i */
            Lx [p] = lki ;
        }
        /* --- Compute L(k,k) ----------------------------------------------- */
        if (d <= 0) return (cs_ndone (N, E, c, x, 0)) ; /* not pos def */
        p = c [k]++ ;
        Li [p] = k ;                /* store L(k,k) = sqrt (d) in column k */
        Lx [p] = sqrt (d) ;
    }
    Lp [n] = cp [n] ;               /* finalize L */
    return (cs_ndone (N, E, c, x, 1)) ; /* success: free E,s,x; return N */
}
```

The `csn` structure contains a numeric Cholesky, LU, or QR factorization.

For a sparse Cholesky factorization, only `N->L` is used. `cs_nfree` frees a numeric factorization. `cs_ndone` frees any workspace and returns a numeric factorization.

```
typedef struct cs_numeric   /* numeric Cholesky, LU, or QR factorization */
{
    cs *L ;         /* L for LU and Cholesky, V for QR */
    cs *U ;         /* U for LU, R for QR, not used for Cholesky */
    int *pinv ;     /* partial pivoting for LU */
    double *B ;     /* beta [0..n-1] for QR */
} csn ;

csn *cs_nfree (csn *N)
{
    if (!N) return (NULL) ;     /* do nothing if N already NULL */
    cs_spfree (N->L) ;
    cs_spfree (N->U) ;
    cs_free (N->pinv) ;
    cs_free (N->B) ;
    return (cs_free (N)) ;      /* free the csn struct and return NULL */
}

csn *cs_ndone (csn *N, cs *C, void *w, void *x, int ok)
{
    cs_spfree (C) ;                     /* free temporary matrix */
    cs_free (w) ;                       /* free workspace */
    cs_free (x) ;
    return (ok ? N : cs_nfree (N)) ;    /* return result if OK, else free it */
}
```

4.8 Left-looking and supernodal Cholesky

The *left-looking* Cholesky factorization algorithm is more commonly used than the up-looking algorithm. The MATLAB function `chol_left` implements this algorithm.

```
function L = chol_left (A)
n = size (A,1) ;
L = zeros (n) ;
for k = 1:n
    L (k,k) = sqrt (A (k,k) - L (k,1:k-1) * L (k,1:k-1)') ;
    L (k+1:n,k) = (A (k+1:n,k) - L (k+1:n,1:k-1) * L (k,1:k-1)') / L (k,k) ;
end
```

It computes L one column at a time and can be derived from the expression

$$\begin{bmatrix} L_{11} & & \\ l_{12}^T & l_{22} & \\ L_{31} & l_{32} & L_{33} \end{bmatrix} \begin{bmatrix} L_{11}^T & l_{12} & L_{31}^T \\ & l_{22} & l_{32}^T \\ & & L_{33}^T \end{bmatrix} = \begin{bmatrix} A_{11} & a_{12} & A_{31}^T \\ a_{12}^T & a_{22} & a_{32}^T \\ A_{31} & a_{32} & A_{33} \end{bmatrix}, \qquad (4.6)$$

where the middle row and column of each matrix are the kth row and column of L, L^T, and A, respectively. If the first $k-1$ columns of L are known, $l_{22} = \sqrt{a_{22} - l_{12}^T l_{12}}$ can be computed first, followed by $l_{32} = (a_{32} - L_{31}l_{12})/l_{22}$. For the sparse case, an amplified version is given below.

All left parts are updated!

```
function L = chol_left (A)
n = size (A,1) ;
L = sparse (n,n) ;
a = sparse (n,1) ;
for k = 1:n
    a (k:n) = A (k:n,k) ;
    for j = find (L (k,:))
        a (k:n) = a (k:n) - L (k:n,j) * L (k,j) ;
    end
    L (k,k) = sqrt (a (k)) ;
    L (k+1:n,k) = a (k+1:n) / L (k,k) ;
end
```

This method requires the nonzero pattern of the columns of `L` to be computed (in sorted order) prior to numerical factorization. In contrast, the up-looking method requires only the cumulative sum of the column counts (`Lp`) prior to numerical factorization and computes both the pattern (in sorted order) and numerical values in a single pass. Computing the pattern of `L` can be done in $O(|L|)$ time, using `cs_ereach` (see Problem 4.10).

The left-looking numerical factorization needs access to row `k` of `L`. The nonzero pattern of `L(k,:)` is given by the kth row subtree (4.2), $\mathcal{T}^k$. This is enough information for the up-looking method, but the left-looking method also needs access to the numerical values of `L(k,:)` and the submatrix `L(k:n,1:k-1)`. To access the numerical values of the `k`th row, an array `c` of size `n` is maintained in the same way as the `c` array in the up-looking algorithm `cs_chol`. The row indices and numerical values in `L(k:n,j)` are given by `Li[c[j] ... Lp[j+1]-1]` and `Lx[c[j] ... Lp[j+1]-1]`, respectively. A left-looking sparse Cholesky factorization is left as an exercise (see Problem 4.11).

The left-looking algorithm forms the basis for the *supernodal* method. The `chol_super` function is a working prototype for a supernodal left-looking Cholesky factorization.

```
function L = chol_super (A,s)
n = size (A) ;
L = zeros (n) ;
ss = cumsum ([1 s]) ;
for j = 1:length (s)
    k1 = ss (j) ;
    k2 = ss (j+1) ;
    k = k1:(k2-1) ;
    L (k,k) = chol (A (k,k) - L (k,1:k1-1) * L (k,1:k1-1)')' ;
    L (k2:n,k) = (A (k2:n,k) - L (k2:n,1:k1-1) * L (k,1:k1-1)') / L (k,k)' ;
end
```

Consider (4.6), and let the middle row and column of the three matrices represent a block of $s_j \geq 1$ rows and columns. This block of columns is selected so that the nonzero patterns of these s_j columns are all identical, except for the diagonal block L_{22}, which is dense. In MATLAB notation, `s` is an integer vector where `all(s>0)` is true, and `sum(s)=n`. The `j`th supernode consists of `s(j)` columns of `L` which can be stored as a dense matrix of dimension $|\mathcal{L}_f|$ by s_j, where f is the column of L represented as the leftmost column in the jth supernode. `chol_super` relies on four key operations, all of which can exploit dense matrix kernels:

1. A symmetric update, `A(k,k)-L(k,1:k1-1)*L(k,1:k1-1)'`. In the sparse case, `A(k,k)` is a dense matrix. `L(k,1:k1-1)` represents the rows in a subset of the descendants of the `j`th supernode. The update from each descendant can be done with a single dense matrix multiplication.

2. A dense Cholesky factorization, `chol`.

3. A sparse matrix product, `A(k2:n,k)-L(k2:n,1:k1-1)*L(k,1:k1-1)'`, where the two `L` terms come from the descendants of the `j`th supernode.

4. A dense triangular solve `(...)/L(k,k)'` using the `k`th diagonal block of `L`.

A supernodal Cholesky factorization based on dense matrix kernels (the BLAS) can achieve a substantial fraction of a computer's theoretical peak performance. MATLAB uses this method (`cholmod_super_numeric`) for a sparse symmetric positive definite A, except when A is very sparse, in which case it uses the up-looking algorithm described in Section 4.7.

4.9 Right-looking and multifrontal Cholesky

The *right-looking* Cholesky factorization is based on the following matrix expression, where l_{11} is a scalar:

$$\begin{bmatrix} l_{11} & \\ l_{21} & L_{22} \end{bmatrix} \begin{bmatrix} l_{11} & l_{21}^T \\ & L_{22}^T \end{bmatrix} = \begin{bmatrix} a_{11} & a_{21}^T \\ a_{21} & A_{22} \end{bmatrix}. \tag{4.7}$$

The first equation, $l_{11}^2 = a_{11}$, is solved for l_{11}, followed by $l_{21} = a_{21}/l_{11}$. Next, the Cholesky factorization $L_{22}L_{22}^T = A_{22} - l_{21}l_{21}^T$ is computed. The `chol_right` function is the MATLAB expression of this algorithm.

```
function L = chol_right (A)
n = size (A) ;
L = zeros (n) ;
for k = 1:n
    L (k,k) = sqrt (A (k,k)) ;
    L (k+1:n,k) = A (k+1:n,k) / L (k,k) ;
    A (k+1:n,k+1:n) = A (k+1:n,k+1:n) - L (k+1:n,k) * L (k+1:n,k)' ;
end
```

It forms the basis for the *multifrontal* method, which is similar to `chol_right`, except that the summation of the outer product $l_{21}l_{21}^T$ is postponed. Only a brief overview of the multifrontal method is given here. See Section 6.3 for more details. Just as in the supernodal method, the columns of L are grouped together; each group is represented by a dense *frontal matrix*. Let $\mathcal{L}_f$ be the nonzero pattern of the first column in a frontal matrix. The frontal matrix has dimension $|\mathcal{L}_f|$-by-$|\mathcal{L}_f|$. Within this frontal matrix, $k \geq 1$ steps of factorization are computed, and a rank-k outer product is computed. These steps can use the dense matrix BLAS, and thus they too can obtain very high performance.

Unlike chol_right, the outer product computed in the frontal matrix is not immediately added into the sparse matrix A. Let column e be the last pivot column of L represented by a frontal matrix ($e =$ k2-1 in chol_super). Its contribution is held in the frontal matrix until its parent is factorized. Its parent is that frontal matrix whose first column is the parent of e in the elimination tree of L. When a frontal matrix is factorized, the contribution blocks of its children are first added together.

MATLAB does not use a multifrontal sparse Cholesky method. It does use the multifrontal method for its sparse LU factorization (see Section 6.3).

4.10 Modifying a Cholesky factorization

Given the sparse Cholesky factorization $LL^T = A$, some applications require the factorization of A after a low-rank change, $A \pm WW^T$, where W is n-by-k with $k \ll n$. Computing the factorization $\overline{LL}^T = A + WW^T$ is a rank-k *update*, and computing $\overline{LL}^T = A - WW^T$ is a rank-k *downdate*. If A is positive definite, $A + WW^T$ is always positive definite, but $A - WW^T$ may not be.

Only the rank-1 case is considered here. There are many methods for computing $\overline{L}$ from L and w. The chol_update function below performs a rank-1 update and is used as the basis for the sparse rank-1 update. For details of its derivation, see the references discussed in Section 4.11. The function returns a modified L̄ that is the Cholesky factor of the matrix L*L'+w*w'. It also computes one additional result, w=L\w, which reveals a key feature exploited in the sparse case.

```
function [L, w] = chol_update (L, w)
beta = 1 ;
n = size (L,1) ;
for j = 1:n
    alpha = w (j) / L (j,j) ;
    beta2 = sqrt (beta^2 + alpha^2) ;
    gamma = alpha / (beta2 * beta) ;
    delta = beta / beta2 ;
    L (j,j) = delta * L (j,j) + gamma * w (j) ;
    w (j) = alpha ;
    beta = beta2 ;
    if (j == n) return, end
    w1 = w (j+1:n) ;
    w (j+1:n) = w (j+1:n) - alpha * L (j+1:n,j) ;
    L (j+1:n,j) = delta * L (j+1:n,j) + gamma * w1 ;
end
```

Consider the sparse case. A key observation is to note that the columns of L that are modified correspond to the nonzero pattern of the solution to the triangular system $Lx = w$. At the jth step, the variable alpha is equal to x_j. This can be seen by removing everything from the algorithm except the modifications to w; all that is left is just a lower triangular solve. If alpha is zero, beta2 and beta are identical, gamma is zero, and delta is one. No change is made to the jth column of L in this case. Thus, the jth step can be skipped if $x_j = 0$.

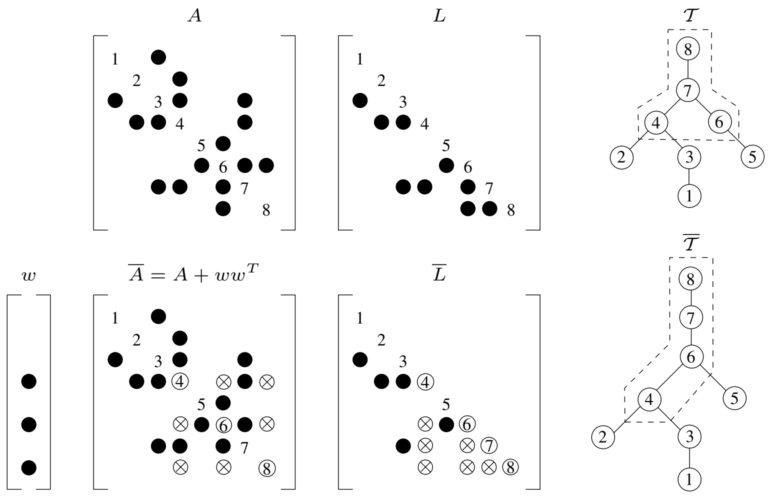


Figure 4.11. *Rank*-1 *update*

Let $\mathcal{X}$ be the nonzero pattern of the solution to $Lx = w$. Let $f = \min \mathcal{W}$, where $\mathcal{W} = \{i \,|, w_i \neq 0\}$ is the set of row indices of nonzero entries in w. Applying Theorem 4.5 to the system $Lx = w$ reveals that $\mathcal{X} = \text{Reach}_{\mathcal{T}}(\mathcal{W})$. That is, $\mathcal{X}$ is a set of paths in the elimination tree, starting at nodes $i \in \mathcal{W}$ and walking up the tree to the root.

Assume that numerical cancellation is ignored. If the nonzero pattern of L changes as a result of an update or downdate, the set $\mathcal{X}$ becomes a single path in the elimination tree $\overline{\mathcal{T}}$ of $\overline{L}$. This is because the first nontrivial step $f = \min \mathcal{W}$ adds the nonzero pattern of $\mathcal{W}$ to the nonzero pattern of column f of L, and thus $\mathcal{W} \subseteq \overline{\mathcal{L}}_f$. Corollary 4.14 states that $\overline{\mathcal{L}}_f$ is a subset of the path from f to the root r in the elimination tree of $\overline{L}$. Thus, $\mathcal{W}$ is a subset of this path $f \rightsquigarrow r$, and computing a rank-1 update requires the traversal of a single path $f \rightsquigarrow r$ in the elimination tree $\overline{\mathcal{T}}$ of $\overline{\mathcal{L}}$. An example rank-1 update is shown in Figure 4.11, where $\mathcal{W} = \{4, 6, 8\}$. Entries in $\overline{A} = A + ww^T$ that differ from A are circled x's or circled numbers on the diagonal, as are entries in $\overline{L}$. The columns that are modified in $\overline{L}$ correspond to the path $\{4, 6, 7, 8\}$; these nodes are highlighted in both elimination trees.

Modifying the nonzero pattern of L is not trivial, since adding entries to individual columns of the `cs` data structure requires all subsequent columns to be shifted. Modifying both the pattern and the values of L can be done in time proportional to the number of entries in L that change, but the full algorithm is beyond the concise scope of this book. The simpler case assumes the pattern of L does not change. This case occurs if and only if $\mathcal{W} \subseteq \mathcal{L}_f$, in which case the elimination tree of L and $\overline{L}$ are the same, and a rank-1 update traverses the path $f \rightsquigarrow r$ in $\mathcal{T}$.

The MATLAB `chol_downdate` function below performs the rank-1 downdate, returning a new $\overline{\texttt{L}}$ that is the Cholesky factor of `L*L'-w*w'`.

```
function [L, w] = chol_downdate (L, w)
beta = 1 ;
n = size (L,1) ;
for j = 1:n
    alpha = w (j) / L (j,j) ;
    beta2 = sqrt (beta^2 - alpha^2) ;
    if (~isreal (beta2)) error ('not positive definite') , end
    gamma = alpha / (beta2 * beta) ;
    delta = beta2 / beta ;
    L (j,j) = delta * L (j,j) ;
    w (j) = alpha ;
    beta = beta2 ;
    if (j == n) return, end
    w (j+1:n) = w (j+1:n) - alpha * L (j+1:n,j) ;
    L (j+1:n,j) = delta * L (j+1:n,j) - gamma * w (j+1:n) ;
end
```

The `cs_updown` function computes a rank-1 update if `sigma=1` or a rank-1 downdate if `sigma=-1`. It assumes that $\mathcal{W}$ is a subset of the first column to be modified, $\mathcal{L}_f$; it does not modify the nonzero pattern of `L`. The sparse column w is passed in as the first column of the `C` matrix. The elimination tree, `parent`, is also required on input.

```
int cs_updown (cs *L, int sigma, const cs *C, const int *parent)
{
    int p, f, j, *Lp, *Li, *Cp, *Ci ;
    double *Lx, *Cx, alpha, beta = 1, delta, gamma, w1, w2, *w, n,  beta2 = 1 ;
    if (!CS_CSC (L) || !CS_CSC (C) || !parent) return (0) ;  /* check inputs */
    Lp = L->p ; Li = L->i ; Lx = L->x ; n = L->n ;
    Cp = C->p ; Ci = C->i ; Cx = C->x ;
    if ((p = Cp [0]) >= Cp [1]) return (1) ;        /* return if C empty */
    w = cs_malloc (n, sizeof (double)) ;            /* get workspace */
    if (!w) return (0) ;                            /* out of memory */
    f = Ci [p] ;
    for ( ; p < Cp [1] ; p++) f = CS_MIN (f, Ci [p]) ;  /* f = min (find (C)) */
    for (j = f ; j != -1 ; j = parent [j]) w [j] = 0 ;  /* clear workspace w */
    for (p = Cp [0] ; p < Cp [1] ; p++) w [Ci [p]] = Cx [p] ; /* w = C */
    for (j = f ; j != -1 ; j = parent [j])          /* walk path f up to root */
    {
        p = Lp [j] ;
        alpha = w [j] / Lx [p] ;                    /* alpha = w(j) / L(j,j) */
        beta2 = beta*beta + sigma*alpha*alpha ;
        if (beta2 <= 0) break ;                     /* not positive definite */
        beta2 = sqrt (beta2) ;
        delta = (sigma > 0) ? (beta / beta2) : (beta2 / beta) ;
        gamma = sigma * alpha / (beta2 * beta) ;
        Lx [p] = delta * Lx [p] + ((sigma > 0) ? (gamma * w [j]) : 0) ;
        beta = beta2 ;
        for (p++ ; p < Lp [j+1] ; p++)
        {
            w1 = w [Li [p]] ;
            w [Li [p]] = w2 = w1 - alpha * Lx [p] ;
            Lx [p] = delta * Lx [p] + gamma * ((sigma > 0) ? w1 : w2) ;
        }
    }
    cs_free (w) ;
    return (beta2 > 0) ;
}
```

`cs_updown` returns `0` if it runs out of memory or if the downdated matrix is not positive definite, and `1` otherwise. To keep the user interface as simple as possible, the function allocates its own workspace `w` of size `n`. Not all of this will be used, and care is taken to initialize only the part of the workspace `w` that will be needed. The time taken by the function is proportional to the number of entries in L that change; this can be much less than `n` (consider the case where `f=n`, for example). Since all columns along the path $f \rightsquigarrow r$ are a subset of this path, only those entries `w[i]` where $i \in \{f \rightsquigarrow r\}$ will be used. An alternative approach (used by CHOLMOD) is to allocate and initialize `w` only once and to set it to zero as the path is traversed. In this case, the traversal of the path to clear `w` can be skipped.

The update/downdate algorithms in CHOLMOD can modify the nonzero pattern of `L`, but this requires a more complex data structure than the `cs` sparse matrix. CHOLMOD can also perform a multiple-rank update/downdate of a supernodal Cholesky factorization and can add or delete rows and columns from L.

The MATLAB interface for `cs_updown` adds a substantial amount of overhead, since a MATLAB mexFunction should not modify its inputs. The interface makes a copy of `L` (taking $O(n + |L|)$ time) and then modifies it with a call to `cs_updown` (taking time proportional to the number of entries in L that change). The latter can be as small as $\Omega(1)$ and as high as $O(|L|)$, so the time to make the copy can be substantially larger than the time to update L. The `cs_updown` function also requires a matrix L for which no entries have been dropped due to numerical cancellation. The MATLAB statement `L=chol(A)'` drops zeros due to numerical cancellation, but `L=cs_chol(A,0)` leaves them in the matrix. The MATLAB function `cholupdate` computes a rank-1 update or downdate for full matrices only.

4.11 Further reading

George and Liu [88, 89] cover sparse Cholesky factorization in depth, prior to the development of elimination trees, supernodal factorization, or many of the algorithms in this chapter. They include a full description of SPARSPAK, which includes the left-looking method described in Section 4.8. SPARSPAK uses linked lists, first used in YSMP (Eisenstat et al.[73] and Eisenstat, Schultz, and Sherman [76]), rather than the $\mathcal{T}^k$ traversal. Gilbert [101] catalogs methods for computing nonzero patterns in sparse matrix computations. Schreiber [181] provides the first formal definition of the elimination tree and also introduces the row subtree $\mathcal{T}^k$. Liu describes the many uses of the elimination tree and how to compute it [148, 150], discusses the row-oriented sparse Cholesky factorization [151], and gives an overview of the multifrontal method [152]. Gilbert et al. [102] and Gilbert, Ng, and Peyton [107] discuss how to compute the row and column counts for Cholesky, QR, and LU factorization and how to compute the elimination tree of A^TA. Davis [29] presents an up-looking LDL^T factorization algorithm with an $O(|L|)$-time column count algorithm that traverses each row subtree explicitly.

The row and column count algorithms `rowcnt` and `cs_counts` are due to Gilbert, Ng, and Peyton [107], except for a few minor modifications. The algorithms described here use a slightly different method for determining the skeleton matrix. Tarjan [195] discusses how the disjoint-set-union data structure can be used efficiently to compute a sequence of least common ancestors.

Many software packages are available for factorizing sparse symmetric positive definite or symmetric indefinite matrices. Details of these packages are summarized in Section 8.6, including references to papers that discuss the supernodal and multifrontal methods. Gould, Hu, and Scott [116] compare many of these packages.

The BLAS (Dongarra et al. [46]) and LAPACK (Anderson et al. [8]) are two of the many software packages that provide dense matrix operations and factorization methods. Optimized BLAS can obtain near-peak performance on many computers (Goto and van de Geijn [115]).[9]

Applications that require the update or downdate of a sparse Cholesky factorization include optimization algorithms, least squares problems in statistics, the analysis of electrical circuits and power systems, structural mechanics, boundary condition changes in partial differential equations, domain decomposition methods, and boundary element methods, as discussed by Hager [124]. Gill et al. [110] and Stewart [190] provide an extensive summary of update/downdate methods. Stewart [189] introduced the term *downdating* and analyzed its error properties. LINPACK includes a rank-1 dense update/downdate [45]; it is used in the MATLAB `cholupdate` function. The `chol_update` function above is Carlson's algorithm [20], and `chol_downdate` is from Pan [163]. The `cs_updown` function is based on Bischof, Pan, and Tang's combination of Carlson's update and Pan's downdate [16]. Davis and Hager developed an optimal sparse multiple-rank supernodal update/downdate method, including a method to add and delete rows from A (CHOLMOD [35, 36, 37]).

Exercises

4.1. Use `cs_ereach` to implement an $O(|L|)$-time algorithm for computing the elimination tree and the number of entries in each row and column of L. It should operate using only the matrix `A` and $O(n)$ additional workspace. The matrix `A` should not be modified.

4.2. Compare and contrast `cs_chol` with the LDL package [29] and with `cholmod_rowfac`. Both can be downloaded from www.siam.org/books/fa02.

4.3. Write a function that solves $Lx = b$ when L, x, and b are sparse and L comes from a Cholesky factorization, using the elimination tree. Assume the elimination tree is already available; it can be passed in a `parent` array, or it can be found by examining `L` directly, since `L` has sorted columns.

4.4. Repeat Problem 4.3, but solve $L^T x = b$ instead.

4.5. The `cs_ereach` function can be simplified if `A` is known to be permuted according to a postordering of its elimination tree and if the row indices in each column of `A` are known to be sorted. Consider two successive row indices i_1 and i_2 in a column of A. When traversing up the elimination tree from node

[9] www.tacc.utexas.edu/resources/software

i_1, the least common ancestor of i_1 and i_2 is the first node $a > i_2$. Let p be the next-to-the-last node along the path $i_1 \rightsquigarrow a$ (where $p < i_2 < a$). Include the path $i_1 \rightsquigarrow p$ in an output queue (not a stack). Continue traversing the tree, starting at node i_2. The resulting queue will be in perfectly sorted order. The `while(len>0)` loop in `cs_ereach` can then be removed.

4.6. Compute the *height* of the elimination tree, which is the length of the longest path from a root to any leaf. The time taken should be $O(n)$. The result should be the same as the second output of the MATLAB `symbfact` function.

4.7. Why is `head` of size `n+1` in `cs_counts`?

4.8. How does the *skeleton* function implicitly ignore duplicate entries?

4.9. The `cs_schol` function computes a postordering, but does not combine it with the fill-reducing ordering, because the ordering from `cs_amd` includes an approximate postordering of the elimination tree. However, `cs_amd` might not be called. Add an option to `cs_schol` to combine the fill-reducing order (or the natural ordering) with the postordering.

4.10. Write a function that computes the symbolic Cholesky factorization of A (the nonzero pattern of L). Hint: start with `cs_chol` and remove any numerical computations. The algorithm should compute the pattern of L in $O(|L|)$ time and return a matrix L with sorted columns. The `s` array can be removed, since the row indices can be stored immediately into `L` in any order. It should allocate both `N->L->i` and `N->L->x` for use in Problem 4.11. Allocating `N->L->x` can be postponed, but allocating it here makes it simpler to write a MATLAB mexFunction interface for this problem.

4.11. Write a sparse left-looking Cholesky factorization algorithm with prototype `int cs_leftchol(cs *A, css *S, csn *N)`. It should assume the nonzero pattern of `L` has already been computed (see Problem 4.10). Compare its performance with `cs_chol` and `cs_rechol` in Problem 4.12. The algorithm is very similar to `cs_chol`. The initializations are identical, except that `x` should be created with `cs_calloc`, not `cs_malloc`. The `N` structure should be passed in with all of `N->L` preallocated. The `s` array is not needed if `cs_ereach` is merged with `cs_leftchol` (the topological order is not required).

4.12. Write a function with prototype `int cs_rechol(cs *A, css *S, csn *N)` that computes the Cholesky factorization of A using the up-looking method. It should assume that the nonzero pattern of `L` has already been computed in a prior call to `cs_chol` (or by Problem 4.10). The nonzero pattern of `A` should be the same as in the prior call to `cs_chol`.

4.13. An *incomplete Cholesky factorization* computes an approximation to L with fewer nonzeros. It is useful as a preconditioner for iterative methods, as discussed in detail by Saad [178]. One method for computing it is to drop small entries from L as they are computed. Another is to use a fixed nonzero pattern (typically the nonzero entries in A) and keep only entries in L within that pattern. Write an incomplete Cholesky factorization based on `cs_chol` or `cs_leftchol` (Problem 4.11). See Problem 6.13 for more details. See also the MATLAB `cholinc` function.