Chapter 4
Sparse Cholesky Solver: The Symbolic Phase

The modern view of numerical linear algebra as being to a large extent the study and systematic use of matrix decompositions has certainly been influenced by Cholesky's posthumously published work – Benzi (2017).

This chapter focuses on the symbolic phase of a sparse Cholesky solver. The sparsity pattern $\mathcal{S}\{A\}$ of the symmetric positive definite (SPD) matrix A is used to determine the nonzero structure of the Cholesky factor L without computing the numerical values of the nonzeros. The subsequent numerical factorization is discussed in the next chapter. Because the symbolic phase works only with $\mathcal{S}\{A\}$ (the values of the entries of A are not considered), it is also used for symmetric indefinite matrices and sometimes within LU factorizations of symmetrically structured nonsymmetric problems. It is implicitly assumed that all the diagonal entries of A are included in $\mathcal{S}\{A\}$ (even if they are zero). During the factorization phase, it may be necessary to amend the data structures to allow for indefiniteness. This makes the factorization of indefinite matrices potentially more expensive and more complex; this is considered further in Chapter 7.

A fundamental difference between dense and sparse Cholesky factorizations is that, in the latter, each column of L depends on only a subset of the previous columns. The elimination tree is a data structure that describes the dependencies among the columns of A during its factorization. A key result that assists in the understanding of sparse Cholesky factorizations is that the sparsity pattern of column j of L is the union of the pattern of column j of the lower triangular part of A and the patterns of the children of j in the elimination tree; this is shown in Section 4.3. Furthermore, the fact that disjoint parts of the elimination tree can be factored independently offers the potential for high-level tree-based parallelism that does not exist for dense matrices.

J. Scott, M. Tůma, *Algorithms for Sparse Linear Systems*, Nečas Center Series,
https://doi.org/10.1007/978-3-031-25820-6_4

4.1 Column Replication Principle

We begin by looking at how the sparsity pattern of a computed column of L influences the patterns of subsequent Schur complements. From (3.2), the Schur complement $S^{(k)}$ can be written as

$$S^{(k)} = A_{k:n,k:n} - \sum_{j=1}^{k-1} \begin{pmatrix} l_{kj} \\ \vdots \\ l_{nj} \end{pmatrix} \begin{pmatrix} l_{kj} & \dots & l_{nj} \end{pmatrix}. \tag{4.1}$$

Consider column j of L ($1 \le j \le k-1$), and let $l_{ij} \neq 0$ for some $i > j$. The involvement of l_{ij} in the outer product in (4.1) allows the following observation.

Observation 4.1 *For any $i > j \ge 1$ such that $l_{ij} \neq 0$*

$$\mathcal{S}\{L_{i:n,j}\} \subseteq \mathcal{S}\{L_{i:n,i}\}. \tag{4.2}$$

This is called the **column replication principle** *because the pattern of column j of L (rows i to n) is replicated in the pattern of column i of L.*

Denote the row index of the first subdiagonal nonzero entry in column j of L by $parent(j)$, that is,

$$parent(j) = \min\{i \mid i > j \text{ and } l_{ij} \neq 0\}. \tag{4.3}$$

If there is no such entry, set $parent(j) = 0$. The row index $parent(parent(j))$ is denoted by $parent^2(j)$, and so on. Applying column replication recursively implies the sparsity pattern of column j of L is replicated in that of column $parent(j)$, which in turn is replicated in the pattern of column $parent^2(j)$, and so on. This is illustrated in Figure 4.1. Here $j = 1$, and because the first subdiagonal entry in column 1 is in row 3, $parent(1) = 3$. Likewise, $parent(3) = parent^2(1) = 5$.

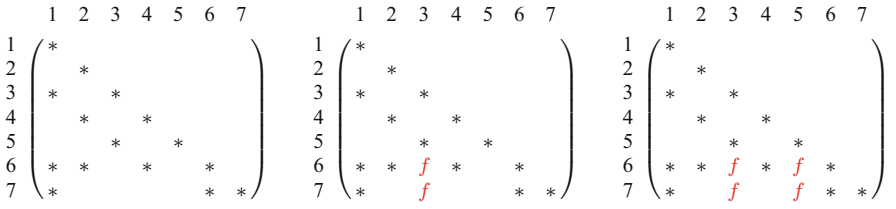

Figure 4.1 An illustration of column replication. On the left are the entries in L before step 1 of a Cholesky factorization (that is, the entries in the lower triangular part of A); in the centre, we show the replication of the nonzeros from column 1 in the pattern of column $parent(1) = 3$ (red entries f); on the right, we show the subsequent replication in column $parent^2(1) = 5$.

The following result shows that, provided A is irreducible, the mapping $parent(j)$ has nonzero values given by (4.3) for all $j < n$.

Theorem 4.1 (Liu 1986) *If A is SPD and irreducible, then in each column j ($1 \leq j < n$) of its Cholesky factor L there exists an entry $l_{ij} \neq 0$ with $i > j$.*

Proof From Parter's rule, each step of the Cholesky factorization corresponds to adding new edges into the graph of the corresponding Schur complement. If A is irreducible, then the graphs corresponding to the Schur complements are connected. Consequently, for any vertex j ($1 \leq j < n$) in any of these graphs, there is at least one vertex i with $i > j$ to which j is connected. This corresponds to the nonzero entry in column j of L. □

With the convention $parent^1(j) = parent(j)$, the next theorem shows that if entry l_{ij} of L is nonzero, then $parent^t(j) = i$ for some $t \geq 1$, and there is an entry in row i of L in each of the columns in the replication sequence $j,\ parent^1(j),\ parent^2(j),\ \ldots,\ parent^t(j)$.

Theorem 4.2 (Liu 1990; George 1998) *Let A be SPD, and let L be its Cholesky factor. If $l_{ij} \neq 0$ for some $j < i \leq n$, then there exists $t \geq 1$ such that $parent^t(j) = i$ and $l_{ik} \neq 0$ for $k = j,\ parent^1(j),\ parent^2(j),\ \ldots,\ parent^t(j)$.*

Proof If $i = parent^1(j)$, the result is immediate. Otherwise, there exists an index k, $j < k < i$ of a subdiagonal entry in column j of L such that $k = parent^1(j)$. Column replication implies $l_{ik} \neq 0$. Applying an inductive argument to l_{ik}, the result follows after a finite number of steps. □

If there is a sequence of nonzeros in a row of L, it is natural to ask where the sequence begins. It is straightforward to see if there is no $k \geq 1$ such that $a_{ik} \neq 0$, no replication of nonzeros can start in row i. The main result on the replication of nonzeros of A is summarized as Theorem 4.3.

Theorem 4.3 (Liu 1986) *Let A be SPD, and let L be its Cholesky factor. If $a_{ij} = 0$ for some $1 \leq j < i \leq n$, then there is a filled entry $l_{ij} \neq 0$ if and only if there exist $k < j$ and $t \geq 1$ such that $a_{ik} \neq 0$ and $parent^t(k) = j$.*

4.2 Elimination Trees

The discussion of column replication is significantly simplified using elimination trees. The **elimination tree** (or **etree**) $\mathcal{T}(A)$ (or simply $\mathcal{T}$) of an SPD matrix has vertices $1, 2, \ldots, n$ and an edge between each pair $(j, parent(j))$, where $parent(j)$ is given by (4.3); j is a root vertex of the tree if $parent(j) = 0$. The edges of $\mathcal{T}$ are considered to be directed from a child to its parent, that is,

$$\mathcal{E}(\mathcal{T}) = \{(j \longrightarrow i) \mid i = parent(j)\}.$$

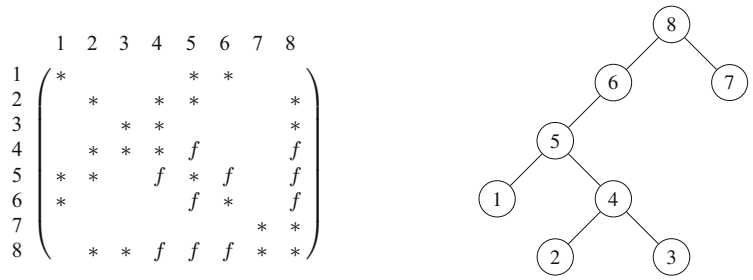

Figure 4.2 An illustration of a sparse matrix A with a symmetric sparsity pattern and its elimination tree $\mathcal{T}(A)$. The root vertex is 8. The filled entries in $\mathcal{S}\{L + L^T\}$ are denoted by f.

If $\mathcal{T}$ has a single component, then the root vertex is n. Despite the terminology, the elimination tree need not be connected and in general is a **forest**. For simplicity, in our discussions, we assume $\mathcal{T}$ has a single component, and we say that $\mathcal{T}$ is described by the vector $parent$.

An example of a matrix and its elimination tree is given in Figure 4.2. Here and elsewhere, following conventional notation, directional arrows are omitted from the tree plot.

Concepts such as child, leaf, ancestor, and descendant vertices introduced in Section 2.3 for directed rooted trees can be applied to $\mathcal{T}$. Additionally, $anc_{\mathcal{T}}\{j\}$ and $desc_{\mathcal{T}}\{j\}$ are defined to be the sets of ancestors and descendants of vertex j in $\mathcal{T}$. We denote by $\mathcal{T}(j)$ the **subtree** of $\mathcal{T}$ induced by j and $desc_{\mathcal{T}}\{j\}$; j is the root vertex of $\mathcal{T}(j)$. The **size** $|\mathcal{T}(j)|$ is the number of vertices in the subtree. A **pruned subtree** of $\mathcal{T}(j)$ is the connected subgraph induced by j and a subset of $desc_{\mathcal{T}}\{j\}$. That is, for any vertex i in a pruned subtree of $\mathcal{T}(j)$, all the ancestors of i belong to the pruned subtree. A pruned subtree of $\mathcal{T}$ shares the mapping $parent$ with $\mathcal{T}$.

The following observation is straightforward.

Observation 4.2 *If $i \in anc_{\mathcal{T}}\{j\}$ for some $j \neq i$, then $i > j$.*

The connection between the mapping $parent$ and the sets of ancestors and descendants is emphasized by the next observation.

Observation 4.3 *If i and j are vertices of the elimination tree $\mathcal{T}$ with $j < i \leq n$, then*

$$i \in anc_{\mathcal{T}}\{j\} \text{ if and only if } j \in desc_{\mathcal{T}}\{i\} \text{ if and only if } parent^t(j) = i \text{ for some } t \geq 1.$$

The results in Section 4.1 can be expressed using rooted trees. Consider, for example, Theorem 4.2. Instead of stating that there exists $t \geq 1$ such that $parent^t(j) = i$, we can write that $i \in anc_{\mathcal{T}}\{j\}$. Rewriting Theorem 4.3 as the following corollary provides a clear characterization of the sparsity patterns of the rows of L.

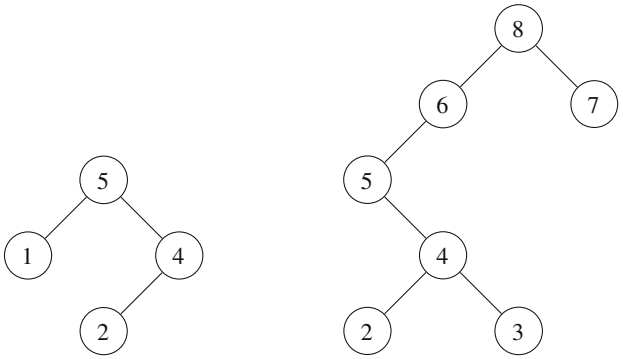

Figure 4.3 The row subtree $\mathcal{T}_r(5)$ of the elimination tree $\mathcal{T}$ from Figure 4.2 (left). Vertex 3 has been pruned because $a_{35} = 0$. The row subtree $\mathcal{T}_r(8)$ (right) differs from $\mathcal{T} = \mathcal{T}(A)$ because vertex 1 has been pruned ($a_{18} = 0$).

Corollary 4.4 (Liu 1986) *Consider the elimination tree $\mathcal{T}$ and the Cholesky factor L of A. If i and j are vertices of $\mathcal{T}$ with $j < i \leq n$ and $a_{ij} = 0$, then $l_{ij} \neq 0$ if and only if there exists $k < j$ such that $j \in anc_{\mathcal{T}}(k)$ and $a_{ik} \neq 0$.*

The subtree of $\mathcal{T}$ with vertices that correspond to the nonzeros of row i of L is called the i-th **row subtree** and is denoted by $\mathcal{T}_r(i)$. Formally, it is a pruned subtree of $\mathcal{T}$ induced by the union of the vertex set

$$\{i\} \cup \{k \mid a_{ik} \neq 0 \text{ and } k < i\}$$

with all vertices on the directed paths in $\mathcal{T}$ from k to i, that is, with all their ancestors from $\mathcal{T}_r(i)$. The root vertex is i, and the leaf vertices are a subset of the column indices in the i-th row of the lower triangular part of A. Figure 4.3 illustrates row subtrees for the matrix and elimination tree from Figure 4.2. Note that row subtrees are connected subgraphs of $\mathcal{T}$, even if $\mathcal{T}$ is not connected. If $\mathcal{T}$ can be found without determining the pattern of L, then $\mathcal{T}_r(i)$ can be used to derive the sparsity pattern of row i of L, without having to store each entry explicitly.

Theorem 4.5 characterizes the ancestors of a given vertex j using paths in $\mathcal{G}(A)$. The proof helps clarify the relationship between $\mathcal{T}$ and paths in $\mathcal{G}(A)$.

Theorem 4.5 (Schreiber 1982; Liu 1986) *If i and j are vertices in the elimination tree $\mathcal{T}$ with $j < i \leq n$, then $i \in anc_{\mathcal{T}}\{j\}$ if and only if there exists a path*

$$j \overset{\mathcal{G}(A)}{\underset{\{1,\ldots,i\}}{\Longrightarrow}} i. \tag{4.4}$$

Proof Assume $i \in anc_{\mathcal{T}}\{j\}$. Then there is a path $j \overset{\mathcal{T}}{\Longrightarrow} i$ of length $l \geq 1$. Each edge of this path belongs to $\mathcal{G}(L)$ and corresponds either to an edge in $\mathcal{G}(A)$ or to a fill-path in $\mathcal{G}(A)$. Connecting these paths together gives (4.4).

Conversely, if the path (4.4) exists, induction on its length can be used to prove the result. If the path is of length 1, then the result holds because i and j are connected in $\mathcal{G}(A)$ by an edge. Consequently, from Theorem 4.2, i is an ancestor of j. Now assume that the result is true for all paths of length less than l ($l > 1$), and consider a path of length l. Let m be the largest vertex on this path. If $m < j$, then (4.4) is a fill-path connecting i and j and, therefore, $i \in anc_{\mathcal{T}}\{j\}$. Otherwise, for $m \geq j$, the assumption implies $i \in anc_{\mathcal{T}}\{m\} \cup \{m\}$ and $m \in anc_{\mathcal{T}}\{j\} \cup \{j\}$, that is, $i \in anc_{\mathcal{T}}\{j\}$. □

Given a vertex j in $\mathcal{T}$, the following corollary indicates how to find $parent(j)$ (if it exists). If the set of ancestors of j is non-empty, then the lowest numbered one is its parent.

Corollary 4.6 (Liu 1986, 1990) *Vertex i is the parent of vertex j in $\mathcal{T}$ if and only if i is the lowest numbered vertex satisfying $j < i \leq n$ for which there is a path (4.4).*

The existence of (4.4) is equivalent to requiring i and j belong to the same component of the graph $\mathcal{G}(A_{1:i,1:i})$ corresponding to the $i \times i$ principal leading submatrix $A_{1:i,1:i}$ of A. Figure 4.4 depicts $\mathcal{G}(A)$ for the matrix A given in Figure 4.2. Consider vertex 4. Its set of ancestors for which paths from Theorem 4.5 exist comprises vertices 5, 6, and 8. Vertex 7 is not an ancestor of 4 because there is no path from 7 to 4 in the graph $\mathcal{G}(A_{1:7,1:7})$. Among the ancestors of 4, vertex 5 fulfils the condition from Corollary 4.6 and is thus the parent of 4.

$\mathcal{T} = \mathcal{T}(A)$ can be constructed by stepwise extensions of the elimination trees of the principal leading submatrices of A. Assume we have $\mathcal{T}(A_{1:i-1,1:i-1})$ and we want to construct $\mathcal{T}(A_{1:i,1:i})$. Initialize $\mathcal{T}(A_{1:i,1:i}) = \mathcal{T}(A_{1:i-1,1:i-1})$. If there are no entries in row i of A to the left of the diagonal, then there is nothing to do, and only an isolated vertex i is added. Otherwise, i is the root of the row subtree $\mathcal{T}_r(i)$ and an ancestor of some vertex j in $\mathcal{T}$. The ancestors k of j with $k < i$ are in $\mathcal{T}(A_{1:i-1,1:i-1})$. Because row subtrees are connected subgraphs of $\mathcal{T}$, a directed path in $\mathcal{T}(A_{1:i,1:i})$ with $parent^t(j) = i$ exists for some $t \geq 1$. The search for this path starts from $jroot = j$ and continues, while $parent(jroot) \neq 0$ and $parent(jroot) \neq i$, using a sequence of assignments $jroot = parent(jroot)$. It terminates once $parent(jroot) = i$ or i is found to have already been added when

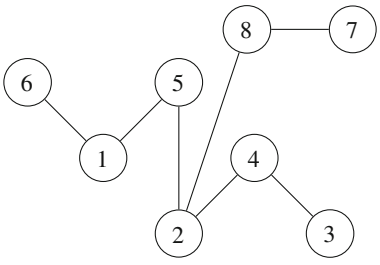

Figure 4.4 The graph $\mathcal{G}(A)$ of the matrix from Figure 4.2 illustrating Theorem 4.5 and Corollary 4.6.

tracing the path from another entry j' in row i. The construction of $\mathcal{T}$ is summarized in Algorithm 4.1.

ALGORITHM 4.1 Construction of an elimination tree
Input: A with a symmetric sparsity pattern and its undirected graph $\mathcal{G}$.
Output: Elimination tree $\mathcal{T}$ described by the vector $parent$.

```
1: for i = 1 : n do                                   ▷ Loop over the rows of A
2:     parent(i) = 0                                  ▷ Initialisation
3:     for j ∈ adj_G{i} and j < i do                  ▷ Loop over the below diagonal entries in row i
4:         jroot = j
5:         while parent(jroot) ≠ 0 and parent(jroot) ≠ i do   ▷ Find the current root
6:             jroot = parent(jroot)
7:         end while
8:         if parent(jroot) = 0 then
9:             parent(jroot) = i                      ▷ Make i the parent of jroot
10:        end if
11:    end for
12: end for
```

The most expensive part of Algorithm 4.1 is the **while** loop that searches for subtree roots. Because the directed path from j to its root $parent^t(j)$ is unique, shortcuts can be incorporated; this is called **path compression**. Having found a directed path from j to k, subsequent searches can be made more efficient by introducing a vector $ancestor$ and setting $ancestor(j) = k$. The modified algorithm is outlined in Algorithm 4.2. It maintains two structures using the current values of $parent$ and $ancestor$. The tree described by $ancestor$ is termed the **virtual tree**.

Figure 4.5 shows a matrix for which path compression makes constructing $\mathcal{T}$ significantly more efficient. For this example, $\mathcal{T}$ is determined by the mapping $parent(6) = 0$; $parent(i) = i + 1$ for $i = 1, \ldots, 5$. The complexity of Algorithm 4.1 is $O(n^2)$, but for this example the complexity of Algorithm 4.2 is $O(n)$. Formally, the complexity of Algorithm 4.2 is $O(nz(A) \log_2(n))$, where $nz(A)$ is the number of nonzeros of A, but the logarithmic factor is rarely reached. Additional modifications can reduce the theoretical complexity to $O(nz(A)\, g(nz(A), n))$, where $g(nz(A), n)$ is a very slowly increasing function called the functional inverse of Ackermann's function. This means that, in practice, the complexity of constructing $\mathcal{T}$, and hence of obtaining an implicit representation of $\mathcal{S}\{L\}$, is close to linear in $nz(A)$ (which in general is much smaller than $nz(L)$).

ALGORITHM 4.2 Construction of an elimination tree using path compression
Input: A with a symmetric sparsity pattern and its undirected graph $\mathcal{G}$.
Output: Elimination tree $\mathcal{T}$ described by the vector $parent$.

```
1:  for i = 1 : n do                                   ▷ Loop over the rows of A
2:      parent(i) = 0, ancestor(i) = 0                 ▷ Initialisation
3:      for j ∈ adj_G{i} and j < i do                  ▷ Loop over the below diagonal entries in row i
4:          jroot = j
5:          while ancestor(jroot) ≠ 0 and ancestor(jroot) ≠ i do
6:              l = ancestor(jroot)
7:              ancestor(jroot) = i                    ▷ Path compression to accelerate future searches
8:              jroot = l
9:          end while
10:         if ancestor(jroot) = 0 then
11:             ancestor(jroot) = i  and  parent(jroot) = i
12:         end if
13:     end for
14: end for
```

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

Figure 4.5 A sparse matrix for which computing the elimination tree using Algorithm 4.2 is much more efficient than using Algorithm 4.1.

The following simple theorem states that there is no edge in $\mathcal{G}(L + L^T)$ between vertices belonging to subtrees of $\mathcal{T}$ with different vertex sets. If there was such an edge (s, t), then from Theorem 4.2, one of the vertices s and t must be an ancestor of the other, which is a contradiction. The importance of this result is that it implies that for any such pairs of vertices the corresponding column sparsity patterns in L can be computed in parallel.

Theorem 4.7 (Liu 1990) *Consider the elimination tree $\mathcal{T}$ and the Cholesky factor L of A. Let $\mathcal{T}(i)$ and $\mathcal{T}(j)$ be two vertex-disjoint subtrees of $\mathcal{T}$. Then for all $s \in \mathcal{T}(i)$ and $t \in \mathcal{T}(j)$, the entry l_{st} of L is zero.*

4.3 Sparsity Pattern of L

The explicit structure of L is not always required; sometimes only the numbers of nonzeros in each row and column of L are needed. For example, when comparing the amount of fill-in in the factors for different initial orderings of A, allocating factor storage, finding relaxed supernodes (see Section 4.6), and determining load balance and synchronization events in parallel factorizations.

Let $row_L\{i\}$ denote the sparsity pattern of the off-diagonal part of row i of L, that is,

$$row_L\{i\} = \mathcal{S}\{L_{i,1:i-1}\} = \{j \mid j < i,\ l_{ij} \neq 0\}, \quad 1 \leq i \leq n.$$

The number of entries in L is

$$nz(L) = \sum_{i=1}^{n} |row_L\{i\}| + n.$$

Corollary 4.4 implies $row_L\{i\}$ is given by the vertices of the row subtree $\mathcal{T}_r(i)$. This suggests Algorithm 4.3. Here the vector $mark$ is used to flag vertices so as to avoid including them more than once within a row subtree. The complexity of the algorithm is $O(nz(L))$.

ALGORITHM 4.3 Computation of the row sparsity patterns of the Cholesky factor L
Input: A with a symmetric sparsity pattern, its undirected graph $\mathcal{G}$ and elimination tree $\mathcal{T}$ described by the vector $parent$.
Output: Row sparsity patterns $row_L\{i\}$ of the Cholesky factor L of A ($1 \leq i \leq n$).

```
1:  for i = 1 : n do                                  ▷ Loop over the rows of A
2:      row_L{i} = ∅                                  ▷ Initialisation
3:      mark(i) = i
4:      for k ∈ adj_G{i} and k < i do                 ▷ Loop over the below diagonal entries in
                                                        row i
5:          j = k
6:          while mark(j) ≠ i do                      ▷ Column j not yet encountered in row i
7:              mark(j) = i                           ▷ Flag j as encountered in row i
8:              row_L{i} = row_L{i} ∪ {j}             ▷ Add j to the sparsity pattern of row i
9:              j = parent(j)                         ▷ Move up the elimination tree
10:         end while
11:     end for
12: end for
```

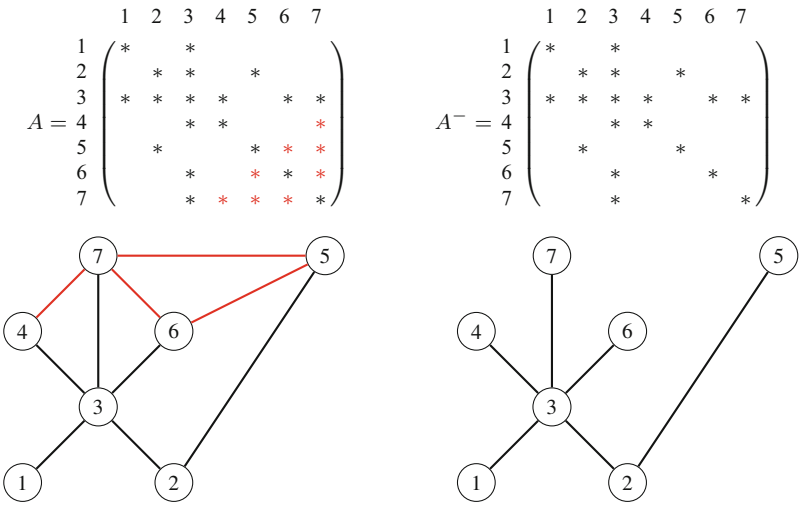

Figure 4.6 An illustration of the sparsity pattern of A and its graph $\mathcal{G}(A)$ (left) and the sparsity pattern of the corresponding skeleton matrix A^- and graph $\mathcal{G}(A^-)$ (right). The entries in A and edges of $\mathcal{G}(A)$ that do not belong to the skeleton matrix and graph are depicted in red.

Efficiency can be improved by employing the **skeleton graph** $\mathcal{G}(A^-)$ that is obtained from $\mathcal{G}(A)$ by removing every edge (i, j) for which $j < i$ and j is not a leaf vertex of $\mathcal{T}_r(i)$. $\mathcal{G}(A^-)$ is the smallest subgraph of $\mathcal{G}(A)$ with the same filled graph as $\mathcal{G}(A)$. The corresponding matrix is the **skeleton matrix**. An example is given in Figure 4.6. The complexity of constructing the elimination tree using the skeleton matrix and its graph $\mathcal{G}(A^-)$ is $O(nz(A^-)\, g(nz(A^-), n))$, where $nz(A^-)$ is the number of entries in the skeleton matrix. Because $nz(A^-)$ is often significantly smaller than $nz(A)$, an implementation that processes $\mathcal{G}(A^-)$ rather than $\mathcal{G}(A)$ can be substantially faster.

Analogously to the row sparsity patterns, let $col_L\{j\}$ denote the sparsity pattern of the off-diagonal part of column j of L, that is,

$$col_L\{j\} = \mathcal{S}(L_{j+1:n,j}) = \{i \mid i > j,\ l_{ij} \neq 0\}, \quad 1 \leq j \leq n.$$

The column replication principle can be written as

$$col_L\{j\} \subseteq col_L\{parent(j)\} \cup parent(j).$$

Theorem 4.8 describes $col_L\{j\}$ using the vertices of the subtree $\mathcal{T}(j)$.

Theorem 4.8 (George & Liu 1980c, 1981) *The column sparsity pattern $col_L\{j\}$ of the Cholesky factor L of the matrix A is equal to the adjacency set of vertices of the subtree $\mathcal{T}(j)$ in $\mathcal{G}(A)$, that is,*

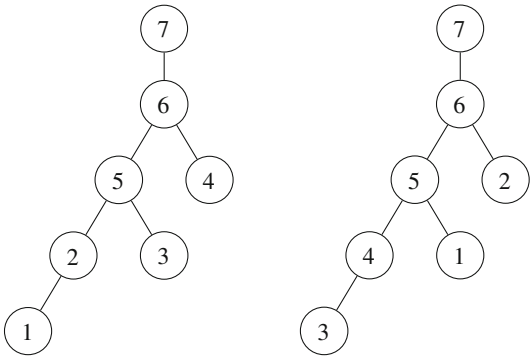

Figure 4.7 Two topological orderings of an elimination tree.

$$col_L\{j\} = adj_{\mathcal{G}(A)}\{\mathcal{T}(j)\}. \tag{4.5}$$

Proof If $i \in col_L\{j\}$, then $j \in row_L\{i\}$, and Theorem 4.3 implies $j \in anc_{\mathcal{T}}\{k\}$ for some k such that $a_{ik} \neq 0$. That is, $i \in adj_{\mathcal{G}}\{\mathcal{T}(j)\}$. Conversely, $i \in adj_{\mathcal{G}}\{\mathcal{T}(j)\}$ implies that in row i the entry in column j of L is nonzero. Thus, $j \in row_L\{i\}$, and hence, $i \in col_L\{j\}$. □

Algorithm 4.3 can be used to compute the column counts and the column sparsity patterns because when j is added to $row_L\{i\}$ at line 8, i can be added to $col_L\{j\}$. This does not generally obtain the column sparsity patterns sequentially. To derive an approach that does compute them sequentially, rewrite (4.5) as follows:

$$col_L\{j\} = \left(adj_{\mathcal{G}(A)}\{j\} \bigcup_{\{k \,|\, k \in \mathcal{T}(j) \setminus \{j\}\}} col_L\{k\} \right) \setminus \{1, \ldots, j\}.$$

Using the column replication, this can be significantly simplified

$$col_L\{j\} = \left(adj_{\mathcal{G}(A)}\{j\} \bigcup_{\{k \,|\, j = parent(k)\}} col_L\{k\} \right) \setminus \{1, \ldots, j\}. \tag{4.6}$$

This is used to obtain Algorithm 4.4, which constructs the sparsity pattern of each column j of L as the union of the sparsity pattern of column j of A ($adj_{\mathcal{G}(A)}\{j\}$) and the patterns of the children of j in $\mathcal{T}(A)$. Here $child\{j\}$ denotes the set of children of j. Because any child k of j satisfies $k < j$, the j-th outer step has the information needed to compute the sparsity pattern described by (4.6). Observe that $\mathcal{T}(A)$ does not need to be input.

ALGORITHM 4.4 Determining the sparsity patterns of each column of L
Input: A with symmetric sparsity pattern and its undirected graph $\mathcal{G}$.
Output: Column sparsity patterns $col_L\{j\}$ of the Cholesky factor L of A ($1 \leq j \leq n$).

```
1: for j = 1 : n do                                  ▷ Loop over the columns of L
2:     child{j} = ∅                                  ▷ Initialisation
3:     col_L{j} = adj_G{j} \ {1, ..., j − 1}
4:     for k ∈ child{j} do                           ▷ Unifying child structures in (4.6)
5:         col_L{j} = col_L{j} ∪ col_L{k} \ {j}
6:     end for
7:     if col_L{j} ≠ ∅ then
8:         l = min{i | i ∈ col_L{j}}
9:         child{l} = child{l} ∪ {j}                 ▷ Parent of j detected using Corollary 4.6
10:    end if
11: end for
```

4.4 Topological Orderings

The outer loop in Algorithm 4.4 does not have to be performed in the strict order $j = 1, \ldots, n$. What is necessary is that for each step j, the column sparsity pattern for each child of j has already been computed. An ordering of the vertices in a tree (and, more generally, in a DAG) is a topological ordering if, for all i and j, $j \in desc_{\mathcal{T}}\{i\}$ implies $j < i$ (Section 2.2). Observation 4.2 confirms that the ordering of vertices in the elimination tree $\mathcal{T}$ is a topological ordering. A new topological ordering of $\mathcal{T}$ defines a relabelling of its vertices corresponding to a symmetric permutation of A. This is illustrated in Figure 4.7. The sparsity patterns of the Cholesky factors of A and PAP^T can be different, but the following result shows that the amount of fill-in is the same.

Theorem 4.9 (Liu 1990) *Let $\mathcal{S}\{A\}$ be symmetric. If P is the permutation matrix corresponding to a topological ordering of the elimination tree $\mathcal{T}$ of A, then the filled graphs of A and PAP^T are isomorphic.*

There are many topological orderings of $\mathcal{T}$. One class is obtained using the depth-first search given by Algorithm 2.1. This searches all the components of $\mathcal{T}$ starting at their root vertices. In this case, once vertex i has been visited, all the vertices of the subtree $\mathcal{T}(i)$ are visited immediately after i and i is labelled as the last vertex of $\mathcal{T}(i)$. A topological ordering of $\mathcal{T}$ is a **postordering** if the vertex set of any subtree $\mathcal{T}(i)$ ($i = 1, \ldots, n$) is a contiguous sublist of $1, \ldots, n$. Unless additional rules on how vertices are selected are imposed, a postordering is generally not unique, as demonstrated in Figure 4.8. One possible postordering is defined in Algorithm 2.1. In this case, there is some freedom in the depth-first search to choose from the vertices that have not been visited, resulting in different postorderings.

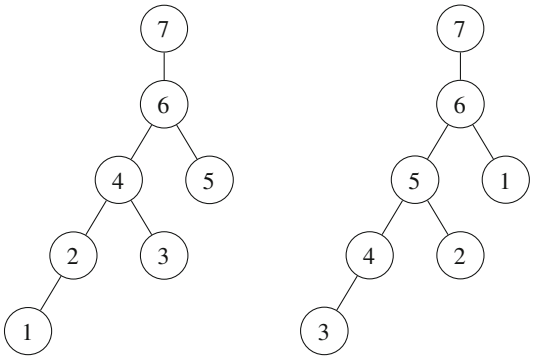

Figure 4.8 An example to illustrate the non-uniqueness of postorderings of an elimination tree.

4.5 Leaf Vertices of Row Subtrees

Leaf vertices of row subtrees play a key role in graph algorithms related to sparse Cholesky factorizations. They can be used to find the skeleton matrix described in Section 4.3, and they are important in parallel processing based on fundamental supernodes (see Section 4.6.1). Theorem 4.10 describes the relation between standard subtrees of $\mathcal{T}$ and row subtrees obtained by pruning (Section 4.2). This pruning is determined by the leaf vertices of row subtrees.

Theorem 4.10 (Liu 1986) *Let the elimination tree $\mathcal{T}$ of A be postordered. Let the column indices of the nonzeros in the strictly lower triangular part of row i of A be $c_1, \ldots, c_s$ with $s \geq 1$ and $0 < c_1 < \ldots < c_s < i$. Then c_t is a leaf vertex of the row subtree $\mathcal{T}_r(i)$ if and only if*

$$t = 1 \;\; or \;\; (1 < t \leq s \;\; and \;\; c_{t-1} \notin \mathcal{T}(c_t)).$$

Proof c_1 is always a leaf vertex of $\mathcal{T}_r(i)$. If this is not the case, then there exists a directed path from some vertex k, $k \neq c_1$ to i via c_1 such that $k \in \mathcal{T}_r(i)$ and $a_{ik} \neq 0$. Postordering of $\mathcal{T}$ implies $k < c_1$. This is a contradiction because c_1 is the index of the first nonzero in row i.

Consider now $t > 1$. Assume that $c_{t-1} \in \mathcal{T}(c_t)$ and that c_t is a leaf vertex of $\mathcal{T}_r(i)$. Row replication (Theorem 4.2) implies any $k \in anc_{\mathcal{T}}\{c_{t-1}\} \cup \{c_{t-1}\}$ such that $c_{t-1} \leq k < i$ satisfies $l_{ik} \neq 0$. Because $\mathcal{T}$ is postordered, $c_{t-1} \leq k \leq c_t$, and there is at least one $k < c_t$ satisfying this inequality. It follows that $k = c_{t-1}$. Because k belongs to $\mathcal{T}_r(i)$, c_t cannot be a leaf vertex of $\mathcal{T}_r(i)$, which is a contradiction.

Conversely, assume for $t > 1$ that $c_{t-1} \notin \mathcal{T}(c_t)$ and c_t is not a leaf vertex of $\mathcal{T}_r(i)$. From the second part of the assumption and the fact that $c_t \in \mathcal{T}_r(i)$, it follows that there is at least one leaf vertex $k < i$ of $\mathcal{T}_r(i)$ from which there is a directed path to i via c_t. Thus $k < c_t$. From the definition of the postordering of $\mathcal{T}$, all vertices l with $k < l \leq c_t$ are vertices of $\mathcal{T}(c_t)$. Vertex c_{t-1} must be among them and $c_{t-1} \in \mathcal{T}(c_t)$. This contradiction completes the proof. □

ALGORITHM 4.5 Find the sizes of subtrees $\mathcal{T}(i)$ of $\mathcal{T}$
Input: Elimination tree $\mathcal{T}$ described by the vector $parent$.
Output: Subtree sizes $|\mathcal{T}(i)|$ $(1 \le i \le n)$.

```
1: |T(1 : n)| = 1
2: for i = 1 : n − 1 do
3:     k = parent(i)
4:     |T(k)| = |T(k)| + |T(i)|
5: end for
```

Corollary 4.11 (Liu 1986) *Under the assumptions of Theorem 4.10, c_t is a leaf vertex of $\mathcal{T}_r(i)$ if and only if*

$$t = 1 \ \text{ or } \ (1 < t \le s \ \text{ and } \ c_{t-1} < c_t - |\mathcal{T}(c_t)| + 1).$$

Subtree sizes can be computed using Algorithm 4.5. Correctness of Algorithm 4.5 is guaranteed because $parent$ defines a topological ordering of $\mathcal{T}$.

Theorem 4.12 relaxes the condition that the entries in the rows of A are sorted by increasing column indices. This allows the leaf vertices of the row subtrees to be determined by columns.

Theorem 4.12 (Liu et al. 1993) *Consider the elimination tree $\mathcal{T}$ of A. Vertex j is a leaf vertex of some row subtree of $\mathcal{T}$ if and only if there exists $i \in adj_{\mathcal{G}(A)}\{j\}$, $j < i \le n$, such that $i \notin adj_{\mathcal{G}(A)}\{k\}$ for all $k \in \mathcal{T}(j) \setminus \{j\}$.*

Proof Assume that for some $i \in anc_{\mathcal{T}}\{j\}$ vertex j is a leaf vertex of $\mathcal{T}_r(i)$. That is, $i \in adj_{\mathcal{G}(A)}\{j\}, i > j$. Suppose there exists k in $\mathcal{T}(j)\setminus\{j\}$ such that $i \in adj_{\mathcal{G}(A)}\{k\}$. Then all the ancestors of k, $k \le i$, in particular j, belong to $\mathcal{T}_r(i)$ and j cannot be a leaf vertex of $\mathcal{T}_r(i)$. This is a contradiction.

Conversely, assume that j is not a leaf vertex of any row subtree of $\mathcal{T}$ and that there exists $i \in adj_{\mathcal{G}(A)}\{j\}$, $j < i \le n$, such that $i \notin adj_{\mathcal{G}(A)}\{k\}$ for all $k \in \mathcal{T}(j) \setminus \{j\}$. Because j is not a leaf vertex of any such $\mathcal{T}_r(i)$, Theorem 4.3 implies that there exists $k \in \mathcal{T}(j) \setminus \{j\}$ such that $a_{ik} \ne 0$, which gives a contradiction and completes the proof. □

To find leaf vertices of row subtrees of $\mathcal{T}$, Algorithm 4.6 uses a marking scheme based on Theorem 4.12 and exploits the postordering of $\mathcal{T}$. The auxiliary vector $prev\_nonz$ stores the column indices of the most recently encountered entries in the rows of the strictly lower triangular part of A.

4.6 Supernodes and the Assembly Tree

Because of column replication, the columns of L generally become denser as the Cholesky factorization proceeds. Exploiting this density can significantly enhance

ALGORITHM 4.6 Find leaf vertices of row subtrees of $\mathcal{T}$
Input: A with a symmetric sparsity pattern and a corresponding postordered elimination tree $\mathcal{T}$.
Output: Logical vector $isleaf$ with entries set to true for leaf vertices of row subtrees.

```
1: isleaf(1:n) = false,  prev_nonz(1:n) = 0
2: Compute |T(1:n)|                                   ▷ Use Algorithm 4.5
3: for j = 1:n do                                     ▷ Loop over the columns of A
4:     for i such that i > j and a_ij ≠ 0 do          ▷ Row index in strictly lower
                                                        triangular part of A
5:         k = prev_nonz(i)                           ▷ Column index of most recently seen entry in row i
6:         if k < j − |T(j)| + 1 then
7:             isleaf(j) = true                       ▷ j is a leaf vertex by Corollary 4.11
8:         end if
9:         prev_nonz(i) = j                           ▷ Flag j as the most recently seen entry in row i
10:    end for
11: end for
```

the performance of the numerical factorization in terms of both computation time and memory requirements. For this, we require the concept of supernodes. The idea is to group together columns with the same sparsity structure, so that they can be treated as a dense matrix for storage and computation. Let $1 \le s, t \le n$ with $s + t - 1 \le n$. A set of contiguously numbered columns of L with indices $S = \{s, s+1, \ldots, s+t-1\}$ is a **supernode** of L if

$$col_L\{s\} \cup \{s\} = col_L\{s+t-1\} \cup \{s, \ldots, s+t-1\}, \tag{4.7}$$

and S cannot be extended for $s > 1$ by adding $s-1$ or for $s+t-1 < n$ by adding $s+t$. Because S cannot be extended, it is a **maximal** subset of column indices. In graph terminology, a supernode is a **maximal clique** of contiguous vertices of $\mathcal{G}(L + L^T)$. A supernode may contain a single vertex. Figure 4.9 illustrates the supernodes in a Cholesky factor of order 8.

The **supernodal elimination** or **assembly tree** is defined to be the reduction of the elimination tree that contains only supernodes. Each vertex of the elimination tree is associated with one elimination, and a single integer (the index of its parent) is needed. Associated with each vertex of the assembly tree is an index list of the row indices of the nonzeros in the columns of the supernode. These implicitly define the sparsity pattern of L. An example that demonstrates the difference between the elimination and assembly trees is given in Figure 4.10. Here the elimination tree is postordered, and there are 5 supernodes: {1, 2}, 3, 4, 5, {6, 7, 8, 9}. For supernode 1 that comprises columns 1 and 2, the row index list is {1, 2, 8, 9}.

$$L = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array} \begin{array}{c} \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\ \begin{pmatrix} * & & & & & & & \\ * & * & & & & & & \\ & & * & & & & & \\ & & * & * & & & & \\ * & * & * & * & * & & & \\ & & & & * & * & & \\ * & * & & & * & * & * & \\ & & * & * & * & * & * & * \end{pmatrix} \end{array}$$

Figure 4.9 An example to illustrate supernodes in L. The first supernode comprises columns 1 and 2, the second columns 3 and 4, and the third columns 5–8.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array} \\ \begin{pmatrix} * & * & & & & & & * & * \\ * & * & & & & & & * & * \\ & & * & * & & & & * & \\ & & * & * & & & * & * & \\ & & & & * & * & * & & \\ & & & & * & * & f & * & * \\ & & & * & * & f & * & f & * \\ * & * & * & * & & * & f & * & f \\ * & * & & & & * & * & f & * \end{pmatrix} \end{array}$$

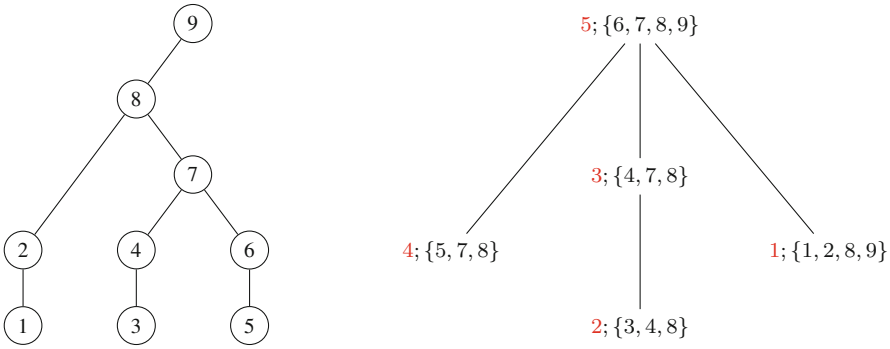

Figure 4.10 A sparse matrix and its postordered elimination tree (left) and postordered assembly tree (right). Filled entries in $\mathcal{S}\{L + L^T\}$ are denoted by f. For the assembly tree, the vertices are in red and the index lists associated with each vertex are given.

Supernodes can be characterized by the following result on the column counts of L, from which we see that supernodes can be found using column counts rather than the column sparsity patterns that appear in (4.7).

Theorem 4.13 (Liu et al. 1993) *The set of columns of L with indices $S = \{s, s + 1, \ldots, s + t - 1\}$ is a supernode of L if and only if it is a maximal set of contiguous columns such that $s + i - 1$ is a child of $s + i$ for $i = 1, \ldots, t - 1$ and*

$$|\, col_L\{s\}\,| = |\, col_L\{s + t - 1\}\,| + t - 1. \tag{4.8}$$

Proof Let S be a supernode. For $i, j \in S$ with $i > j$, we have $i \in col_L\{j\}$. This implies that in the postordered elimination tree the vertex $i = j + 1$ is the parent of j for $j = s, \ldots, s + t - 2$. Moreover, from Observation 4.2, for any $i, j \in S$ with $i > j, i \in col_L\{j\}$ implies $col_L\{j\} \setminus \{1, \ldots, i\} \subseteq col_L\{i\}$. Therefore,

$$|\, col_L\{s + i\}\,| \geq |\, col_L\{s + i - 1\}\,| - 1, \quad i = 1, \ldots, t - 1, \tag{4.9}$$

with equality if and only if

$$col_L\{s + i\} = col_L\{s + i - 1\} \setminus \{s + i\},$$

that is, if S is a supernode.

Conversely, assume S is a maximal set of contiguous columns such that, for $i = 1, \ldots, t - 1$, $s + i - 1$ is a child of $s + i$ and S satisfies (4.8). Because of column replication, such a sequence of parent and child vertices must satisfy (4.9) with equality if and only if (4.7) is satisfied. It follows that S is a supernode. □

Supernodes enhance the efficiency of sparse factorizations and sparse triangular solves because they enable floating-point operations to be performed on dense submatrices rather than on individual nonzeros, thus improving memory hierarchy utilization and allowing the use of highly efficient dense linear algebra kernels (such as Level 3 BLAS kernels). Because the rows and columns of a supernode have a common sparsity structure, this only needs to be stored once, reducing indirect addressing. Supernodes help to increase the granularity of tasks, which is useful for improving the computation to overhead ratio in a parallel implementation. Fill-in results in supernodes near the root of the assembly tree often being much larger than those close to the leaf vertices.

Observe that the columns within a supernode are numbered consecutively, but they can be numbered within the supernode in any order without changing the number of nonzeros in L (assuming the corresponding rows are permuted symmetrically). On some architectures, particularly those using GPUs, this freedom can be exploited to improve the factorization efficiency. Essentially, it is desirable to order the columns within a supernode such that the entries of L form fewer but less fragmented dense blocks.

Some applications, such as power grid analysis, in which the basis of the linear system is not a finite element or finite difference discretization of a physical domain, can lead to sparse matrices that incur very little fill-in during factorization. The supernodes can then be very small, and the costs associated with identifying them may not be offset by the increase in performance resulting from the potential for block operations. However, as supernodes can offer such significant performance gains, it can be advantageous to merge (small) supernodes that have similar (but not exactly the same) nonzero patterns, despite this increasing the overall fill-in and operation count. This process is termed **supernode amalgamation**, and the resultant nodes are often referred to as **relaxed supernode**.

4.6.1 Fundamental Supernodes

In practice, fundamental supernodes are easier to work with in the numerical factorization. Let $1 \le s, t \le n$ with $s + t - 1 \le n$. A maximal set of contiguously numbered columns of L with indices $S = \{s, s+1, \ldots, s+t-1\}$ is a **fundamental supernode** if for any successive pair $i-1$ and i in the list, $i-1$ is the only child of i in $\mathcal{T}$ and $col_L\{i\} \cup \{i\} = col_L\{i-1\}$. s is termed the starting vertex. An example is given in Figure 4.11. The difference between the sets of supernodes and fundamental supernodes is normally not large, with the latter having (slightly) more blocks in the resulting partitioning of L. Note that fundamental supernodes are independent of the choice of the postordering of $\mathcal{T}$. Theorem 4.14 describes the relationship between fundamental supernodes and the leaf vertices of row subtrees of $\mathcal{T}$. In particular, it characterizes starting vertices of the fundamental supernodes. The leaf vertices of $\mathcal{T}$ are trivially starting vertices of fundamental supernodes. But, possibly surprisingly, so too are the leaf vertices of row subtrees.

Theorem 4.14 (Liu et al. 1993) *Assume $\mathcal{T}$ is postordered. Vertex s is the starting vertex of a fundamental supernode if and only if it has at least two child vertices in $\mathcal{T}$ or it is a leaf vertex of a row subtree of $\mathcal{T}$.*

Proof If s has at least two child vertices then, from the definition of a fundamental supernode, it must be the starting vertex of a fundamental supernode. Assume that, for some $i > s$, s is a leaf vertex of $\mathcal{T}_r(i)$. If s is also a leaf vertex of $\mathcal{T}$, then s is a starting vertex of a supernode. The remaining case is s having only one child. Because $\mathcal{T}$ is postordered, this child must be $s-1$. Theorem 4.3 then implies $a_{is} \neq 0$ and $a_{i,s-1} = 0$, that is, $i \in col_L\{s\}$ and $i \notin col_L\{s-1\}$. It follows that

$$\mathcal{S}\{L_{s-1:n,s-1}\} \subsetneqq \mathcal{S}\{L_{s:n,s}\} \cup \{s-1\},$$

and vertices s and $s-1$ cannot belong to the same supernode. Hence, s is the starting vertex of a new fundamental supernode.

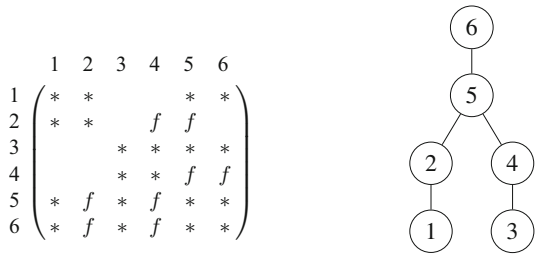

Figure 4.11 A matrix A and its postordered elimination tree $\mathcal{T}$ for which the set of supernodes $\{1, 2\}$ and $\{3, 4, 5, 6\}$ and the set of fundamental supernodes $\{1, 2\}$, $\{3, 4\}$ and $\{5, 6\}$ are different. The filled entries in $\mathcal{S}\{L + L^T\}$ are denoted by f.

Conversely, assume that s is the starting vertex of a fundamental supernode S. If s has no child vertices or at least two child vertices, the result follows. If s has exactly one child vertex, postordering implies this child is $s-1$. Because S is maximal, there exists i such that $i \notin col_L\{s-1\}$ and $i \in col_L\{s\}$ (otherwise S could be extended by adding $s-1$). Hence, s is a leaf vertex of $\mathcal{T}_r(i)$. □

Because fundamental supernodes are characterized by their starting vertices, they can be found by modifying Algorithm 4.6 to incorporate marking leaf vertices of the row subtrees and vertices with at least two child vertices. Once the elimination tree has been computed, the complexity is $O(n+nz(A))$. The computation can be made even more efficient by using the skeleton graph $\mathcal{G}(A^-)$.

4.7 Notes and References

The excellent monographs by Tewarson (1973), George & Liu (1981), and Davis (2006) represent milestones in the development of contemporary symbolic factorization algorithms and their implementation. A complementary way to follow many of the developments is by looking at the early software (and accompanying user documentation), such as YSMP (Eisenstat et al., 1982) and SPARSPAK (George & Ng, 1984). In addition, there are several influential survey articles focusing on sparse Cholesky algorithms and emphasizing the crucial role of the elimination tree, for example, Liu (1990), George (1998); see also Bollhöfer & Schenk (2006), Hogg & Scott (2013a) and the more recent comprehensive survey of Davis et al. (2016). The latter provides a general overview of much of the research related to sparse direct methods and includes pointers to many specialized references.

There are a large number of journal articles that provide a fuller understanding of the theory and algorithms employed in symbolic factorizations. Schreiber (1982) defines the elimination tree of a sparse symmetric matrix. The seminal paper of Liu (1986) describes elimination tree construction, while for an extensive overview of the roles of elimination trees and topological orderings as well as the determination of the column sparsity patterns of the factor L, we refer to Liu (1990). If only row and column counts of L are needed, the fastest known algorithms are described in Gilbert et al. (1994). This paper also refers to another admirable paper of Liu et al. (1993) that describes the efficient computation of fundamental supernodes based on the leaf vertices of row subtrees of the elimination tree.

A key driver behind research into efficient (in terms of time and memory) sparse Cholesky algorithms has always been the development of computational codes. Many currently available packages implement not only sparse Cholesky factorizations but also more general LDLT factorizations of sparse symmetric indefinite matrices. The software is necessarily highly sophisticated and is therefore generally accompanied by technical reports and/or journal publications that explain the data structures and choices that were made in the algorithm and software design as well as providing details of the different options that are offered (examples include Duff (2004), Reid & Scott (2009), Hogg et al. (2010)).