CHAPTER 3

Computing the Cholesky Factorization of Sparse Matrices

In many support preconditioners, the preconditioner B is factored before the iterations begin. The Cholesky factorization of B allows us to efficiently solve the correction equations $Bz = r$. This chapter explains the principles behind the factorization of sparse symmetric positive definite matrices.

1. The Cholesky Factorization

We first show that the Cholesky factorization $A = LL^T$ of a symmetric positive-definite (SPD) matrix A always exists.

A matrix A is *positive definite* if $x^T Ax > 0$ for all $0 \neq x \in \mathbb{R}^n$. The same definition extends to complex Hermitian matrices. The matrix is positive *semidefinite* if $x^T Ax \geq 0$ for all $0 \neq x \in \mathbb{R}^n$ and $x^T Ax = 0$ for some $0 \neq x \in \mathbb{R}^n$.

To prove the existence of the factorization, we use induction and the construction shown in Chapter XXX. If A is 1-by-1, then $x^T Ax = A_{11}x_1^2 > 0$, so $A_{11} \geq 0$, so it has a real square root. We set $L_{11} = \sqrt{A_{11}}$ and we are done. We now assume by induction that all SPD matrices of dimension $n-1$ or smaller have a Cholesky factorization. We now partition A into a 2-by-2 block matrix and use the partitioning to construct a factorization,

$$A = \begin{bmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{bmatrix} .$$

Because A is SPD, A_{11} must also be SPD. If it is not positive definite, then a vector $y \neq 0$ such that $y^T A_{11} y \leq 0$ can be extended with zeros to a vector x such that $x^T Ax \leq 0$. Therfore, A_{11} has a Choleskyh factor L_{11} by induction. The Cholesky factor of a nonsingular matrix must be nonsingular, so we can define $L_{21} = A_{21} L_{11}^{-T}$ (L_{11}^{-T} denotes the inverse of L_{11}^T). The key step in the proof is to show that the *Schur complement* $A_{22} - L_{21}L_{21}^T = A_{22} - A_{21}A_{11}^{-1}A_{21}^T$ is also positive definite. Suppose for contradition that it is not, and let $z \neq 0$ be such that

$z^T \left(A_{22} - L_{21} L_{21}^T \right) z \leq 0$. We define

$$x = \begin{bmatrix} -A_{11}^{-1} A_{21}^T z \\ z \end{bmatrix} = \begin{bmatrix} w \\ z \end{bmatrix} .$$

Now we have

$$\begin{aligned} x^T A x &= w^T A_{11} w + w^T A_{21}^T z + z^T A_{21} w + z^T A_{22} z \\ &= \left(A_{11}^{-1} A_{21}^T z \right)^T A_{11} \left(A_{11}^{-1} A_{21}^T z \right) - 2 z^T A_{21} \left(A_{11}^{-1} A_{21}^T z \right) w + z^T A_{22} z \\ &= z^T A_{21} A_{11}^{-1} A_{21}^T z - 2 z^T A_{21} A_{11}^{-1} A_{21}^T z + z^T A_{22} z \\ &= z^T \left(A_{22} - L_{21} L_{21}^T \right) z \\ &\leq 0 , \end{aligned}$$

which is impossible, so our supposition was wrong. Because $A_{22} - L_{21} L_{21}^T$ is SPD, it has a Cholesky factor L_{22} by induction. The three blocks L_{11}, L_{21}, and L_{22} form a Cholesky factor for A, since

$$\begin{aligned} A_{11} &= L_{11} L_{11}^T \\ A_{21} &= L_{21} L_{11}^T \\ A_{22} &= L_{21} L_{21}^T + L_{22} L_{22}^T . \end{aligned}$$

Symmetric positive semidefinite (SPSD) matrices also have a Cholesky factorization, but in floating-point arithmetic, it is difficult to compute a Cholesky factor that is both backward stable and has the same rank as A. To see that a factorization exists, we modify the construction as follows. If A is 1-by-1, then if it is singular than it is exactly zero, in which case we can set $L = A$. Otherwise, we partition A, selecting $A_{11}$to be 1-by-1. If $A_{11} \neq 0$, then it is invertible and we can continue with the same proof, except that we show that the Schur complement is semidefinite, not definte. If $A_{11} = 0$, then it is not hard to show that A_{21} must be zero as well. This implies that the equation $A_{21} = L_{21} L_{11}^T$ is zero on both sides for any choice of L_{21}. We set $L_{21} = 0$ and continue.

The difficulty in a floating-point implementation lies in deciding whether a computed A_{11} would be zero in exact arithmetic. In general, even if the next diagonal element $A_{11} = 0$ in exact arithmetic, in floating-point it might be computed as a small nonzero value. Should we round it (and A_{21}) to zero? Assuming that it is a nonzero when in fact is should be treated is a zero leads to an unstable factorization. The small magnitude of the computed L_{11} can cause the elements of L_{21} to be large, which leads to large inaccuracies in the Schur complement. On the other hand, rounding small values to zero always leads to a backward stable factorization, since the rounding is equivalent to

a small perturbation in A. But rounding a column to zero when the value in exact arithmetic is not zero causes the rank of L to be smaller than the rank of A. This can later cause trouble, since some vectors b that are in the range of A are not in the range of L. In such a case, there is no x such that $LL^T x = b$ even if $Ax = b$ is consistent.

2. Work and Fill in Sparse Cholesky

When A is sparse, operations on zeros can be skipped. For example, suppose that

$$A = \begin{bmatrix} 2 & 0 & \cdots \\ 0 & 3 & \\ \vdots & & \ddots \end{bmatrix} = \begin{bmatrix} \sqrt{2} & 0 & \cdots \\ 0 & \sqrt{3} & \\ \vdots & & \ddots \end{bmatrix} \begin{bmatrix} \sqrt{2} & 0 & \cdots \\ 0 & \sqrt{3} & \\ \vdots & & \ddots \end{bmatrix}^T .$$

There is no need to divide 0 by $\sqrt{2}$ to obtain $L_{2,1}$ (the element in row 2 and column 1; from here on, expressions like $L_{2,1}$ denote matrix elements, not submatrices). Similarly, there is no need to subtract 0×0 from the 3, the second diagonal element. If we represent zeros implicitly rather than explicitly, we can avoid computations that have no effect, and we can save storage. How many arithmetic operations do we need to perform in this case?

Definition 2.1. We define $\eta(A)$ to be the number of nonzero elements in a matrix A. We define $\phi_{\texttt{alg}}(A)$ to be the number of arithmetic operations that some algorithm `alg` performs on an input matrix A, excluding multiplications by zeros, divisions of zeros, additions and subtractions of zeros, and taking square roots of zeros. When the algorithm is clear from the context, we drop the subscript in the ϕ notation.

Theorem 2.2. *Let* `SparseChol` *be a sparse Cholesky factorization algorithm that does not multiply by zeros, does not divide zeros, does not add or subtract zeros, and does not compute square roots of zeros. Then for a symmetric positive-definite matrix A with a Cholesky factor L we have*

$$\begin{aligned} \phi_{\texttt{SparseChol}}(A) &= \sum_{j=1}^{n} \left(1 + \eta\left(L_{j+1:\,n}\right) + \frac{\eta\left(L_{j+1:\,n}\right)\left(\eta\left(L_{j+1:\,n}\right)+1\right)}{2} \right) \\ &= \sum_{j=1}^{n} O\left(\eta^2\left(L_{j+1:\,n}\right)\right) . \end{aligned}$$

Proof. Every arithmetic operation in the Cholesky factorization involves an element or two from a column of L: in square roots and divisions the output is an element of L, and in multiply-subtract the

two inputs that are multiplied are elements from one column of L. We count the total number of operations by summing over columns of L.

Let us count the operations in which elements of $L_{j:\,n,j}$ are involved. First, one square root. Second, divisions of $\eta\left(L_{j+1:\,n}\right)$ by that square root. We now need to count the operations in which elements from this column update the Schur complement. To count these operations, we assume that the partitioning of A is into a 2-by-2 block matrix, in which the first diagonal block consists of j rows and columns and the second of $n-j$. The computation of the Schur complement is

$$A_{j+1:\,n,j+1:\,n} - L_{j+1:\,n,1:\,j} L^T_{j+1:\,n,1:\,j} = A_{j+1:\,n,j+1:\,n} - \sum_{k=1}^{j} L_{j+1:\,n,k} L^T_{j+1:\,n,k} \ .$$

This is the only Schur-complement computation in which $L_{j:\,n,j}$ is involved under this partitioning of A. It was not yet used, because it has just been computed. It will not be used again, because the recursive factorization of the Schur complement is self contained. The column contributes one outer product $L_{j+1:\,n,j} L^T_{j+1:\,n,j}$. This outer product contains $\eta^2\left(L_{j+1:\,n}\right)$ nonzero elements, but it is symmetric, so only its lower triangle needs to be computed. For each nonzero element in this outer product, two arithmetic operations are performed: a multiplication of two elements of $L_{j+1:\,n}$ and a subtraction of the product from another number. This yields the total operation count. □

Thus, the number of arithmetic operations is asymptotically proportional to the sum of squares of the nonzero counts in columns of L. The total number of nonzeros in L is, of course, simply the sum of the nonzero counts,

$$\eta(L) = \sum_{j=1}^{n} \eta\left(L_{j+1:\,n}\right) \ .$$

This relationship between the arithmetic operation count and the nonzero counts shows two things. First, sparser factors usually (but not always) require less work to compute. Second, a factor with balanced nonzero counts requires less work to compute than a factor with some relatively dense columns, even if the two factors have the same dimension and the same total number of nonzeros.

The nonzero structure of A and L does not always reveal everything about the sparsity *during* the factorization. Consider the following

matrix and its Cholesky factor,

$$A = \begin{bmatrix} 4 & & 2 & 2 \\ & 4 & 2 & -2 \\ 2 & 2 & 6 & \\ 2 & -2 & & 6 \end{bmatrix} = \begin{bmatrix} 2 & & & \\ & 2 & & \\ 1 & 1 & 2 & \\ 1 & -1 & & 2 \end{bmatrix} \begin{bmatrix} 2 & & & \\ & 2 & & \\ 1 & 1 & 2 & \\ 1 & -1 & & 2 \end{bmatrix}^T .$$

The element in position $4, 3$ is zero in A and in L, but it might fill in one of the Schur complements. If we partition A into two 2-by-2 blocks, this element never fills, since

$$\begin{aligned} A_{3:4,3:4} - L_{3:4,1:2} L_{3:4,1:2}^T &= \begin{bmatrix} 6 & \\ & 6 \end{bmatrix} - \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \\ &= \begin{bmatrix} 6 & \\ & 6 \end{bmatrix} - \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \\ &= \begin{bmatrix} 4 & \\ & 4 \end{bmatrix} . \end{aligned}$$

However, if we first partition A into a 1-by-1 block and a 3-by-3 block, then the $4, 3$ element fills in the Schur complement,

$$\begin{aligned} A_{2:4,2:4} - L_{2:4,1} L_{2:4,1}^T &= \begin{bmatrix} 4 & 2 & -2 \\ 2 & 6 & \\ -2 & & 6 \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 4 & 2 & -2 \\ 2 & 6 & \\ -2 & & 6 \end{bmatrix} - \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 4 & 2 & -2 \\ 2 & 5 & -1 \\ -2 & -1 & 5 \end{bmatrix} . \end{aligned}$$

When we continue the factorization, the $4, 3$ element in the Schur complement must cancel exactly by a subsequent subtraction, because we know it is zero in L (The Cholesky factor is unique). This example shows that an element can fill in some of the Schur complements, even if it zero in L. Clearly, even an position that is not zero in A can become zero in L due to similar cancelation. For some analyses, it helps to define fill in a way that accounts for this possibility.

DEFINITION 2.3. A *fill* in a sparse Cholesky factorization is a row-column pair i, j such that $A_{i,j} = 0$ and which fills in at least one Schur complement. That is, $A_{i,j} = 0$ and $A_{i,j} - L_{i,1:k} L_{1:k,j}^T \neq 0$ for some $k < j$.

3. An Efficient Implemention of Sparse Cholesky

To fully exploit sparsity, we need to store the matrix and the factor in a data structure in which effect-free operations on zeros incur no computational cost at all. Testing values to determine whether they are zero before performing an arithmetic operation is a bad idea: the test takes time even if the value is zero (on most processor, a test like this takes more time than an arithmetic operation). The data structure should allow the algorithm to implicitly skip zeros. Such a data structure increases the cost of arithmetic on nonzeros. Our goal is to show that in spite of the overhead of manipulating the sparse data structure, the total number of operations in the factorization can be kept proportional to the number of arithmetic operations.

Another importan goal in the design of the data structure is memory efficiency. We would like the total amount of memory required for the sparse-matrix data structure to be proportional to the number of nonzeros that it stores.

Before we can present a data structure that achieves these goals, we need to reorder the computations that the Cholesky factorization perform. The reordered variant that we present is called *column-oriented* Cholesky. In the framework of our recursive formulations, this variant is based on repartitioning the matrix after the elimination of each column. We begin with a partitioning of A into the first row and column and the rest of the matrix,

$$A = \begin{bmatrix} A_{1,1} & A_{2:\,n,1}^T \\ A_{2:\,n,1} & A_{2:\,n,2:\,n} \end{bmatrix} .$$

We compute $L_{1,1} = \sqrt{A_{1,1}}$ and divide $A_{2:\,n,1}$ to by the root to obtain $L_{2:\,n,1}$. Now comes the key step. Instead of computing all the Schur complement $A_{2:\,n,2:\,n} - L_{2:\,n,1}L_{2:\,n,1}^T$, we compute only its first column, $A_{2:\,n,2} - L_{2:\,n,1}L_{2,1}^T$. The first column of the Schur complement allows us to compute the second column of L. At this point we have computed the first two columns of L and nothing else. We now view the partitioning of A as

$$A = \begin{bmatrix} A_{1:2,1:2} & A_{3:\,n,1:2}^T \\ A_{3:\,n,1:2} & A_{3:\,n,3:\,n} \end{bmatrix} = \begin{bmatrix} L_{1:2,1:2} & \\ L_{3:\,n,1:2} & L_{3:\,n,3:\,n} \end{bmatrix} \begin{bmatrix} L_{1:2,1:2} & \\ L_{3:\,n,1:2} & L_{3:\,n,3:\,n} \end{bmatrix}^T .$$

We have already computed $L_{1:2,1:2}$ and $L_{3:\,n,1:2}$. We still need to compute $L_{3:\,n,3:\,n}$. We do so in the same way: computing the first column of the Schur complement $A_{3:\,n,3} - L_{3:\,n,1:2}L_{3,1:\,2}^T$ and eliminating it. The algorithm, ignoring sparsity issues, is shown in Figure XXX.

for $j = 1 : n$
  $S_{j:\,n} = A_{j:\,n,j} - L_{j:\,n,1:j-1} L^T_{j,1:\,j-1}$
  $L_{j,j} = \sqrt{S_j}$
  $L_{j+1:\,n,j} = S_{j+1:\,n} / L_{j,j}$
end

FIGURE 1. Column-oriented Cholesky. The vector S is a temporary vector that stores a column of the Schur complement. By definition, operations involving a range $i : j$ of rows or columns for $i < 1$ or $j > n$ are not performed at all. (This allows us to drop the conditional that skips the computation of the Schur-complement column for $j = 1$ and the computation of the nonexisting subdiagonal part of the last column of L; in an actual algorithm, these conditionals would be present, or the processing of $j = 1$ and $j = n$ would be performed outside the loop.)

We now present a data structure for the efficient implementation of sparse column-oriented Cholesky. Our only objective in presenting this data structure is show that sparse Cholesky can be implemented in time proportional to arithmetic operations and in space proportional to the number of nonzeros in A and L. The data structure that we present is not the best possible. Using more sophisticated data structures, the number of operations in the factorization and the space required can be reduced further, but not asymptotically.

We store A using an array of n linked lists. Each linked list stores the diagonal and subdiagonal nonzeros in one column of A. Each structure in the linked-list stores the value of one nonzero and its row index. There is no need to store the elements of A above its main diagonal, because (1) A is symmetric, and (2) column-oriented Cholesky never references them. There is also no need to store the column indices in the linked-list structures, because each list stores elements from only one column. The linked lists are ordered from low row indices to high row indices.

We store the already-computed columns of L redundantly. One part of the data structure, which we refer to as `columns`, stores the columns of L in exactly the same way we store A. The contents of the two other parts of the data structure, called `cursors` and `rows`, depend on the number of columns that have already been computed. Immediately before step j begins, these parts contains the following data. `Cursors` is an array of n pointers. The first $j - 1$ pointers, if

set, point to linked-list elements of `columns`. $\texttt{Cursors}_i$ points to the first element with row index larger than j in $\texttt{columns}_i$, if there is such an element. If not, it is not set (that is, it contains a special invalid value). The other $n-j+1$ elements of `cursors` are not yet used. Like `columns`, `rows` is an array of linked list. The ith list stores the elements of $L_{i,1:\,j-1}$, but in reverse order, from high to low column indices. Each element in such a list contains a nonzero value and its column index.

The column S of the Schur complement is represented by a data structure `s` called a *sparse accumulator*. This data structure consists of an array `s.values` of n real or complex numbers, an array `s.rowind` of n row indices, and array `s.exists` of booleans, and an integer `s.nnz` that specifies how many rows indices are actually stored in `s.rowind`.

Here is how we use these data structures to compute the factorization. Step j begins by copying $A_{j:\,n,j}$ to `s`. To do so, we start by setting `s.nnz` to zero. We then traverse the list that represents $A_{j:\,n,j}$. For a list element that represents $A_{i,j}$, we increment `s.nnz`, store i in $\texttt{s.rowind}_{\texttt{s.nnz}}$ store A_{ij} in $\texttt{s.values}_i$, and set $\texttt{s.exists}_i$ to a true value.

Our next task is to subtract $L_{j:\,n,1:j-1}L^T_{j,1:\,j-1}$ from S. We traverse $\texttt{rows}_j$ to find the nonzero elements in $L^T_{j,1:\,j-1}$. For each nonzero $L_{j,k}$ that we find, we subtract $L_{j:\,n,k}L^T_{k,j} = L_{j,k}L_{j:\,n,k}$ from S. To subtract, we traverse $\texttt{columns}_k$ starting from $\texttt{cursors}_k$. Let $L_{i,k}$ be a nonzero found during this traversal. To subtract $L_{j,k}L_{i,k}$ from S_i, we first check whether $\texttt{s.exists}_i$ is true. If it is, we simply subtract $L_{j,k}L_{i,k}$ from $\texttt{s.values}_i$. If not, then S_i was zero until now (in step j). We increment `s.nnz`, store i in $\texttt{s.rowind}_{\texttt{s.nnz}}$ store $0 - L_{j,k}L_{i,k}$ in $\texttt{s.values}_i$, and set $\texttt{s.exists}_i$ to a true. During the traversal of $\texttt{columns}_k$, we many need to advance $\texttt{cursors}_k$ to prepare it for subsequent steps of the algorithm. If the first element that we find in the travelral has row index j, we advance $\texttt{cursors}_k$ to the next element in $\texttt{columns}_k$. Otherwise, we do not modify $\texttt{columns}_k$.

Finally, we compute $L_{j:\,n,j}$ and insert it into the nonzero data structure that represents L. We replace $\texttt{s.values}_j$ by its square root. For each rows index $i \neq j$ stored in one of the first `s.nnz` entries of `s.rowind`, we divide $\texttt{s.values}_i$ by $\texttt{s.values}_j$. We now sort elements 1 though `s.nnz` of `s.rowind`, to ensure that the elements $\texttt{columns}_j$ are linked in ascending row order. We traverse the row indices stored in `s.rowind`. For each index i such that $\texttt{s.values}_i \neq 0$, we allocate a new element for $\texttt{columns}_j$, link it to the previous element that we created, and store in it i and $\texttt{s.values}_i$. We also set $\texttt{s.exists}_i$ to false, to prepare it for the next iteration.

We now analyze the number of operations and the storage requirements of this algorithm.

THEOREM 3.1. *The total number of operations that the sparse-cholesky algorithm described above performs is* $\Theta(\phi(A))$. *The amount of storage that the algorithm uses, including the representation of its input and output, is* $\Theta(n + \eta(A) + \eta(A))$.

PROOF. Let's start with bounding work. Step j starts with copying column j of A into the sparse accumulator, at a total cost of $\Theta(1 + \eta(A_{j:\,n,j}))$. No aritmetic is performed, but we can charge these operations to subsequent arithmetic operations. If one of these values is modified in the accumulator, we charge the copying to the subsequent multiply-subtract. If not, we charge it to either the square root of the diagonal element or to the division of subdiagonal elements. We cannot charge all the copying operations to roots and divisions, becuase some of the copied elements might get canceled before they are divided by $L_{j,j}$.

The traversal of $\texttt{rows}_j$ and the subtractions of scaled columns of L from the accumulator are easy to account for. The processing of an element of $\texttt{rows}_j$ is charged to the modification of the diagonal, $S_j = S_j - L_{j,k}^2$. The traversal of the suffix of $\texttt{columns}_k$ performs $2\eta(L_{j:\,n,k})$ arithmetic operations and$\Theta(\eta(L_{j:\,n,k}))$ non-arithmetic operations, so all operations are accounted for.

After the column of the Schur complement is computed, the algorithm computes a square root, scales $\eta(L_{j:\,n,j}) - 1$ elements, sorts the indices of $L_{j:\,n,j}$, and creates a linked-list to hold the elements of $L_{j:\,n,j}$. The total number of operations in these computations is clearly $O(\eta^2(L_{j+1:\,n,j}))$, even if we use a quadratic sorting algorithm, so by using Theorem 2.2, we conclude that the operation-count bound holds.

Bounding space is easier. Our data structure includes a few arrays of length n and linked lists. Every linked-list element represents a nonzero in A or in L, and every nonzero is represented by at most two linked-list elements. Therefore, the total storage requirements is $\Theta(n + \eta(A) + \eta(A))$. □

The theorem shows that the only way to asymptotically reduce the total number of operations in a sparse factorization is to reduce the number of arithmetic operations. Similarly, to asymptotically reduce the total storage we must reduce the fill. There are many ways to optimize and improve both the data structure and the algorithm that

we described, but these optimizations can reduce the number of operations and the storage requirements only by a constant multiplicative constant.

Improvements in this algorithm range from simple data-structure optimizations, through sophisticated preprocessing steps, to radical changes in the representation of the Schur complement. The most common data-structure optimization, which is used by many sparse factorization codes, is the use of *compressed-column storage*. In this data structure, all the elements in all the `columns` list are stored in two contiguous arrays, one for the actual numerical values and another for the row indices. A integer third array of size $n + 1$ points to the beginning of each column in these arrays (and to the location just past column n). Preprocessing steps can upper bound the number of nonzeros in each column of L (this is necessary for exact preallocation of the compressed-column arrays) and the identity of potential nonzeros. The prediction of fill in L can can eliminate the conditional in the inner loop that updates the sparse accumulator; this can significantly speed up the computation. The preprocessing steps can also construct a compact data structured called the *elimination tree* of A that can be used for determining the nonzero structure of a row of L without maintaining the `rows` lists. This also speeds up the computation significantly. The elimination tree has many other uses. Finally, *multifrontal* factorization algorithms maintain the Schur complement in a completely different way, as a sum of sparse symmetric update matrices.

The number of basic computational operations in an algorithm is not an accurate predictor of its running time. Different algorithms have different mixtures of operations and different memory-access patterns, and these factors affect running times, not only the total number of operations. We have seen evidence for this in Chapter XXX, were we have seen cases where direct solvers run at much higher computational rates than iterative solvers. But to develop fundamental improvements in algorithms, it helps to focus mainly on operation counts, and in particular, on asymptotic operation counts. Once new algorithms are discovered, we can and should optimize them both in terms of operation counts and in terms of the ability to exploit cache memories, multiple processors, and other architectural features of modern computers. But our primary concern here is asymptotic operation counts.

4. Characterizations of Fill

If we want to reduce fill, we need to characterize fill. Definition 2.3 provides an characterization, but this characterization is not useful for predicting fill before it occurs. One reason that predicting fill using Definition 2.3 is that it for cancellations, which are difficult to predict. In this section we provide two other characterizations. They are not exact, in the sense that they characterize a superset of the fill. In many cases these characterizations are exact, but not always. On the other hand, these charactersizations can be used to predict fill before the factorization begins. Both of them are given in terms of graphs that are related to A, not in terms of matrices.

DEFINITION 4.1. The *pattern graph* of an n-by-n symmetric matrix A is an undirected graph $G_A = (\{1, 2, \ldots n\}, E)$ whose vertices are the integers 1 through n and whose edges are pairs (i, j) such that $A_{i,j} \neq 0$.

DEFINITION 4.2. Let G be an undirected graph with vertices $1, 2, \ldots, n$. A *vertex elimination step* on vertex j of G is the transformation of G into another undirected graph eliminate(G, j). The edge set of eliminate(G, j) is the union of the edge set of G with a clique on the neighbors of j in G whose indices are larger than j,

$$E(\text{eliminate}(G, j)) = E(G) \cup \{(i, k) \quad | \quad \begin{array}{l} i > j \\ k > j \\ (j, i) \in E(G) \\ (k, i) \in E(G)\} \end{array} .$$

The *fill graph* G_A^+ of A is the graph

$$G_A^+ = \text{eliminate}(\text{eliminate}(\cdots \text{eliminate}(G_A, 1) \cdots), n-1), n) .$$

The edges of the fill graph provide a bound on fill

LEMMA 4.3. *Let A be an n-by-n symmetric positive-definite matrix. Let $1 \leq j < i \leq n$. If $A_{i,j} \neq 0$ or i, j is a fill element, then (i, j) is an edge of G_A^+.*

PROOF. The elimination of a vertex only adds edges to the graph. The fill graph is produced by a chain of vertex eliminations, starting from the graph of A. If $A_{i,j} \neq 0$, then (i, j) is an edge of the graph of A, and therefore also an edge of its fill graph.

We now prove by induction on j that a fill element i, j is also an edge (i, j) in

$$\text{eliminate}(\text{eliminate}(\cdots \text{eliminate}(G_A, 1) \cdots), j-2), j-1) .$$

The claim hold for $j = 1$ because $L_{i,1} \neq 0$ if and only if $A_{i,1} \neq 0$. Therefore, there are no fill edges for $j = 1$, so the claims holds. We now prove the claim for $j > 1$. Assume that i, j is a fill element, and let $S^{(j)}$ be the Schur complement just prior to the jth elimination step,

$$S^{(j)} = A_{j\colon n,j\colon n} - L_{j\colon n,1\colon j-1} L^T_{j\colon n,1\colon j-1} = A_{j\colon n,j\colon n} - \sum_{k=1}^{j-1} L_{j\colon n,k} L^T_{j\colon n,k} \; .$$

Since i, j is a fill element, $S^{(j)}_{i,j} \neq 0$ but $A_{i,j} = 0$. Therefore, $L_{i,k} L_{j,k} \neq 0$ for some $k < j$. This implies that $L_{i,k} \neq 0$ and $L_{j,k} \neq 0$. By induction, (i, k) and (j, k) are edges of

$$\text{eliminate}(\cdots \text{eliminate}(G_A, 1) \cdots), k - 1) \; .$$

Therefore, (i, j) is an edge of

$$\text{eliminate}(\text{eliminate}(\cdots \text{eliminate}(G_A, 1) \cdots), k - 1), k) \; .$$

Since edges are never removed by vertex eliminations, (i, j) is also an edge of

$$\text{eliminate}(\text{eliminate}(\cdots \text{eliminate}(G_A, 1) \cdots), j - 2), j - 1) \; .$$

This proves the claim and the entire lemma. □

The converse is not true. An element in a Schur complement can fill and then be canceled out, but the edge in the fill graph remains. Also, fill in a Schur complement can cancel exactly an original element of A, but the fill graph still contains the corresponding edge.

The following lemma provides another characterization of fill.

LEMMA 4.4. *The pair (i, j) is an edge of the fill graph of A if and only if G_A contains a simple path from i to j whose internal vectices all have indices smaller than* $\min(i, j)$.

PROOF. Suppose that G_A contains such a path. We prove that (i, j) edge of the fill graph by induction on the length of the path. If path contains only one edge then (i, j) is an edge of G_A, so it is also an edge of G^+_A. Suppose that the claim holds for paths of length $\ell - 1$ or shorter and that a path of length ℓ connects i and j. Let k be the smallest-index vertex in the path. The elimination of vertex k adds an edge connecting its neighbors in the path, because their indices are higher than k. Now there is a path of length $\ell - 1$ between i and j; the internal vertices still have indices smaller than $\min(i, j)$. By induction, future elimination operations on the graph will create the fill edge (i, j).

To prove the other direction, suppose that (i, j) is an edge of G^+_A. If (i, j) is an edge of G_A, a path exists. If not, the edge (i, j) is created by some elimination step. Let k be the vertex whose elimination creates

this edge. We must have $k < \min(i, j)$, otherwise the kth elimination step does not create the edge. Furthermore, when k is eliminated, the edges (k, i) and (k, j) exist. If they are original edge of G_A, we are done—we have found the path. If not, we use a similar argument to show that there must be suitable paths from i to k and from k to j. The concatenation of these paths is the sought after path. □

5. Perfect and Almost-Perfect Elimination Orderings

The Cholesky factorization of symmetric positive definite matrices is numerically stable, and symmetrically permuting the rows and columns of an SPD matrix yields another SPD matrix. Therefore, we can try to symmetrically permute the rows and columns of a sparse matrix to reduce fill and work in the factorization. We do not have to worry that the permutation will numerically destabilize the factorization. In terms of the graph of the matrix, a symmetric row and column pemutation corresponds to relabeling the vertices of the graphs. In otherwords, given an undirected graph we seek an elimination ordering for its vertices.

Some graphs have elimination orderings that generate no fill, so that $G_A^+ = G_A$. Such orderings are called *perfect-elimination orderings*. The most common example are trees.

LEMMA 5.1. *If G_A is a tree or a forest, then $G_A^+ = G_A$.*

PROOF. On a tree or forest, any depth-first-search postorder is a perfect-elimination ordering.

By Lemma 4.4, all the fill edges occur within connected components of G_A. Therefore, it is enough to show that each tree in a forest has a perfect-elimination ordering. We assume from here on that G_A is a tree.

Let v be an arbitrary vertex of G_A. Perform a depth-first traversal of G_A starting from the root v, and number the vertices in postorder: give the next higher index to a vertex v immediately after the traversal returns from the last child of v, starting from index 1. Such an ordering guarantees that in the rooted tree rooted at v, a vertex u has a higher index than all the vertices in the subtree rooted at u.

Under such an ordering, the elimination of a vertex u creates no fill edges at all, because u has at most one neighbor with a higher index, its parent in the rooted tree. □

Most graphs do not have perfect-elimination orderings, but some orderings are almost as good. The elimination of a vertex with only one higher-numbered neighbor is called a perfect elimination step, because

it produces no fill edge. Eliminating a vertex with two higher-numbered neighbors is not perfect, but almost: it produces one fill edge. If G_A contains an isolated path

$$v_0 \leftrightarrow v_1 \leftrightarrow v_2 \leftrightarrow \cdots \leftrightarrow v_\ell \leftrightarrow v_{\ell+1}$$

such that the degree of $v_1, v_2, \ldots, v_\ell$ in G_A is 2, then we can eliminate $v_1, \ldots, v_\ell$ (in any order), producing only ℓ fill edges and performing only $\Theta(\ell)$ operations. This observation is useful if we try to sparsify a graph so that the sparsified graph has a good elimination ordering.

6. Symbolic Elimination and the Elimination Tree

We can use the fill-characterization technique that Lemma 4.3 describes to create an efficient algorithm for predicting fill. Predicting fill, or even predicting just the nonzero counts in rows and columns of the factor, can lead to more efficient factorization algorithms. The improvements are not asymptotic, but they can be nonetheless significant.

The elimination of vertex k adds to the graph the edges of a clique, a complete subgraph induced by the higher-numbered neighbors of k. Some of these edges may have already been part of the graph, but some may be new. If we represent the edge set of the partially-eliminated graph using a *clique-cover* data structure, we efficiently simulate the factorization process and enumerate the fill edges. An algorithm that does this is called a *symbolic elimination* algorithm. It is symbolic in the sense that it simulates the sparse factorization process, but without computing the numerical values of elements in the Schur complement or the factor.

A clique cover represents the edges of an undirected graph using an array of linked lists. Each list specifies a clique by specifying the indices of a set of vertices: each link-list element specifies one vertex index. The edge set of the graph is the union of the cliques. We can create a clique-cover data structure by creating one two-vertex clique for each edge in the graph.

A clique cover allows us to simulate elimination steps efficiently. We also maintain an array of vertices. Each element in the vertex array is a doubly-linked list of cliques that the vertex participates in. To eliminate vertex k, we need to create a new clique containing its higher-numbered neighbors. These neighbors are exactly the union of higher-than-k vertices in all the cliques that k participates in. We can find them by traversing the cliques that k participates in. For each clique q that k participates in, we traverse q's list of vertices and add the higher-than-k vertices that we find to the new clique. We add a

vertex to the new clique at most once, using a length-n bit vector to keep track of which vertices have already been added to the new clique. Before we move on to eliminate vertex $k+1$, we clear the bits that we have set in the bit vector, using the vertices in the new clique to indicate which bits must be cleared. For each vertex j in the new clique, we also add the new clique to the list of cliques that j participates in.

The vertices in the new clique, together with k, are exactly the nonzero rows in column k of a cancellation-free Cholesky factor of A. Thus, the symbolic-elimination algorithm predicts the structure of L if there are no cancellations.

We can make the process more efficient by not only creating new cliques, but merging cliques. Let q be a clique that k participates in, and let $i > k$ and $j > k$ be vertices in q. The clique q represents the edge (i, j). But the new clique that the elimination of k creates also represents (i, j), because both i and j belong to the new clique. Therefore, we can partially remove clique q from the data structure. The removal makes the elimination of higher-than-k vertices belonging to it cheaper, because we will not have to traverse it again. To remove q, we need each element of the list representing q to point to the element representing q in the appropriate vertex list. Because the vertex lists are doubly-linked, we can remove elements from them in constant time.

Because each clique either represents a single nonzero of A or the nonzeros in a single column of a cancellation-free factor L, the total size of the data structure is $\Theta(\eta(A) + \eta(L) + n)$. If we do not need to store the column structures (for example, if we are only interested in nonzero counts), we can remove completely merged cliques. Because we create at most one list element for each list element that we delete, the size of the data structure in this scheme is bounded by the size of the original data structure, which is only $\Theta(\eta(A) + n)$.

The number of operations in this algorithm is $\Theta(\eta(A) + \eta(L) + n)$, which is often much less than the cost of the actual factorization,

$$\phi(A) = \sum_{j=1}^{n} O\left(\eta^2\left(L_{j+1:\,n}\right)\right) \ .$$

To see that this is indeed the case, we observe that each linked-list element is touched by the algorithm only twice. An element of a clique list is touched once when it is first created, and another time when the clique is merged into another clique. Because the clique is removed from vertex lists when it is merged, it is never referenced again. An element of a vertex list is also touched only once after it is created.

Either the element is removed from the vertex's list before the vertex is eliminated, or it is touched during the traveral of the vertex's list.

The process of merging the cliques defines an important structure that plays an important role in many sparse-matrix factorization algorithms, the *elimination tree* of A. To define the elimination tree, we name some of the cliques. Cliques that are created when we initialize the clique cover to represent the edge set of G_A have no name. Cliques that are formed when we eliminate a vertex are named after the vertex. The elimination tree is a rooted forest on the vertices $\{1, 2, \ldots, n\}$. The parent $\pi(k)$ of vertex k is the name of the clique into which clique k is merged, if it is. If clique k is not merged in the symbolic-elimination process, k is a root of a tree in the forest. There are many alternative definitions of the elimination tree. The elimination tree has many applications. In particular, it allows us to compute the number of nonzeros in each row and column of a cancellation-free Cholesky factor in time almost linear in $\eta(A)$, even faster than using symbolic elimination.

7. Minimum-Degree Orderings

The characterization of fill in Lemma 4.3 also suggests an ordering heuristic for reducing fill. If the elimination of a yet-uneliminated vertex creates a clique whose size is the number of the uneliminated neighbors of the chosen vertex, it makes sense to eliminate the vertex with the fewer uneliminated neighbors. Choosing this vertex minimizes the size of the clique that is created in the next step and minimizes the arithmetic work in the next step. Ordering heuristics based on this idea are called minimum-degree heuristics.

There are two problems in the minimum-degree idea. First, not all the edges in the new cliques are new; some of them might be part of G_A or might have already been created by a previous elimination step. It is possible to minimize not the degree but the actual new fill, but this turns out to be more expensive algorithmically. Minimizing fill in this way also turns out to produce orderings that are not significantly better than minimum-degree orderings. Second, an optimal choice for the next vertex to eliminate may be suboptimal globally. There are families of matrices on which minimum-degree orderings generate asymptotically more fill than the optimal ordering. Minimum-degree and minimum-fill heuristics are greedy and myopic. They select vertices for elimination one at a time, and the lack of global planning can hurt them. This is why minimum fill is often not much better than minimum fill.

Even though minimum-degree algorithms are theoretically known to be suboptimal, they are very fast and often produce effective orderings. On huge matrices nested-dissection orderings, which we discuss next, are often more effective than minimum-degree orderings, but on smaller matrices, minimum-degree orderings are sometimes more effective in reducing fill and work.

Minimum-degree algorithms usually employ data structures similar to the clique cover used by the symbolic elimination algorithm. The data structure is augmented to allow vertex degrees to be determined or approximated. Maintaining exact vertex degrees is expensive, since a vertex cover does not represent vertex degrees directly. Many successful minimum-degree algorithms therefore use degree approximations that are cheaper to maintain or compute. Since the minimum-degree is only a heuristic, these approximations are not neccessarily less effective in reducing fill than exact degrees; sometimes they are more effective.

8. Nested-Dissection Orderings for Regular Meshes

Nested-dissection orderings are known to be approximately optimal, and on huge matrices that can be significantly more effective than other ordering heuristics. On large matrices, the most effective orderings are often nested-dissection/minimum-degree hybrids.

Nested-dissection orderings are defined recursively using vertex subsets called separators.

DEFINITION 8.1. Let $G = (V, E)$ be an undirected graph with $|V| = n$. Let α and β be real positive constants, and let f be a real function over the positive integers. An (α, β, f) *vertex separator* in G is a set $S \subseteq V$ of vertices that satisfies the following conditions.

Separation:: There is a partition $V_1 \cup V_2 = V \setminus S$ such that for any $v \in V_1$ and $u \in V_2$, the edge $(u, v) \notin E$.
Balance:: $|V_1|, |V_2| \leq \alpha n$.
Size:: $|S| \leq \beta f(n)$.

Given a vertex separator S in G_A, we order the rows and columns of A of the separator last, say in an arbitrary order (but if $G \setminus S$ contains many connected components then a clever ordering of S can further reduce fill and work), and the rows and columns corresponding to V_1 and V_2 first. By Lemma 4.4, this ensures that for any $v \in V_1$ and $u \in V_2$, the edge (u, v) is *not* a fill edge, so $L_{u,v} = L_{v,u} = 0$. Therefore, the interleaving of vertices of V_1 and V_2 has no effect on fill, so we can just as well order all the vertices of V_1 before all the vertices of V_2.

The function of the separator in the ordering is to ensure that $L_{u,v} = 0$ for any $v \in V_1$ and $u \in V_2$. Any ordering in which the vertices of

V_1 appear first, followed by the vertices of V_2, followed by the vertices of S, ensures that a $|V_1|$-by-$|V_2|$ rectangular block in L does not fill. A good separator is one for which this block is large. The size of S determines the circumference of this rectangular block, because half the circumference is $|V_1| + |V_2| = n - |S|$. The size imbalance between V_1 and V_2 determines the aspect ratio of this rectangle. For a given circumference, the area is maximized the rectangle is as close to square as possible. Therefore, a small balanced separator reduces fill more effectively than a large or unbalanced separator.

By using separators in the subgraphs of G_A induced by V_1 and V_2 to order the diagonal blocks of A that correspond to V_1 and V_2, we can avoid fill in additional blocks of L. These blocks are usually smaller than the top-level $|V_1|$-by-$|V_2|$ block, but they are still helpful and significant in reducing the total fill and work. If $|V_1|$ or $|V_2|$ are small, say smaller than a constant threshold, we order the corresponding subgraphs arbitrarily.

Let us see how this works on square two-dimensional meshes. To keep the analysis simple, we use a cross-shaped separator that partitions the mesh into four square or nearly-square subgraphs. We assume that G_A is an n_x-by-n_y where $n_x n_y = n$ and where $|n_x - n_y| \leq 1$. The size of a cross-shaped separator in G_A is $|S| = n_x + n_y - 1 < 2\sqrt{n}$. To see this, we note that if $n_x = n_y = \sqrt{n}$ then $|S| = 2\sqrt{n} - 1$. Otherwise, without loss of generality $n_x = n_y - 1 < n_y$, so $|S| = 2n_x = 2\sqrt{n_x n_x} < 2\sqrt{n_x n_y} = 2\sqrt{n}$. The separator breaks the mesh into four subgraphs, each of which is almost square (their x and y dimensions differ by at most 1), of size at most $n/4$. This implies that throught the recursion, each size-m subgraph that is produced by the nested-dissection ordering has a $(0.25, 0.5, \sqrt{m})$ 4-way separator that we use in the ordering.

We analyze the fill and work in the factorization by blocks of columns. Let S be the top-level separator and let $V_1, V_2, \ldots, V_4$ be the vertex sets of the separated subgraphs. We have

$$\begin{aligned}
\eta(L) &= \eta\left(L_{:,S}\right) + \eta\left(L_{:,V_1}\right) + \eta\left(L_{:,V_2}\right) + \eta\left(L_{:,V_3}\right) + \eta\left(L_{:,V_4}\right) \\
&\leq \frac{|S|\left(|S|+1\right)}{2} + \eta\left(L_{:,V_1}\right) + \eta\left(L_{:,V_2}\right) + \eta\left(L_{:,V_3}\right) + \eta\left(L_{:,V_4}\right) \ .
\end{aligned}$$

Bounding the fill in a block of columns that corresponds to one of the separated subgraphs is tricky. We cannot simply substitute a similar expression to form a reccurence. The matrix A is square, but when we analyze fill in one of these column blocks, we need to account for fill in both the square diagonal block and in the subdiagonal block. If $u \in S$ and $v \in V_1$ and $A_{u,v} \neq 0$, then elements in L_{u,V_1} can fill. A

accurate estimate of how much fill occurs in such blocks of the factor is critical to the analysis of nested dissection algorithms. If we use the trivial bound $\eta(L_{u,V_1}) \leq n/4$, we get an asymptotically loose bound $\eta(L) = O(n^{1.5})$ on fill.

To achieve an asymptotically tight bound, we set up a recurrence for fill in a nested-dissection ordering of the mesh. Let $\bar{\eta}(m_x, m_y)$ be a bound on the fill in the columns corresponding to an m_x-by-m_y submesh (with $|m_x - m_y| \leq 1$). At most $2m_x + 2m_y$ edges connect the submesh to the rest of the mesh. Therefore we have

$$\bar{\eta}(m_x, m_y) \leq \frac{(m_x + m_y - 1)(m_x + m_y)}{2} + (m_x + m_y - 1)(2m_x + 2m_y) + 4\bar{\eta}\left(\frac{m_x}{2}, \frac{m_y}{2}\right) .$$

The first term in the right-hand side bounds fill in the separator's diagonal block. The second term, which is the crucial ingredient in the analysis, bounds fill in the subdiagonal rows of the separator columns. The third term bounds fill in the four column blocks corresponding to the subgraphs of the separated submesh. If we cast the recurrence in terms of $m = m_x m_y$ we obtain

$$\bar{\eta}(m) \leq \Theta(m) + 4\bar{\eta}\left(\frac{m}{4}\right) .$$

By the Master Theorem CLR2ed_Thm_4.1, The solution of the recurrence is $\bar{\eta}(m) = O(m \log m)$. Since $\eta(L) \leq \bar{\eta}(n)$, we have $\eta(L) = O(n \log n)$. We can analyze work in the factorization in a similar way. We set up a recurrence

$$\bar{\phi}(m) \leq \Theta\left(m^{1.5}\right) + 4\phi\left(\frac{\bar{m}}{4}\right) ,$$

whose solution leads to $\phi(A) = O(n\sqrt{n})$. It is possible to show that these two bounds are tight and that under such a nested-dissection ordering for a two-dimensional mesh, $\eta(L) = \Theta(n \log n)$ and $\phi(A) = \Theta(n\sqrt{n})$.

For a three-dimensional mesh, a similar analysis yields $\eta(L) = \Theta(n^{4/3})$ and $\phi(A) = \Theta(n^{6/3}) = \Theta(n^2)$.

In practice, we can reduce the constants by stopping the recurrence at fairly large subgraphs, say around $m = 100$, and to order these subgraphs using a minimum-degree ordering. This does not change the asymptotic worst-case bounds on work and fill, but it usually improves the actual work and fill counts. There are other nested-dissection/minimum-degree hybrids that often improve the work and fill counts without hurting the asymptotic worst-case bounds.

DEFINITION 8.2. A class of graphs satisfies an (α, β, f) *vertex-separator theorem* (or a separator theorem for short) if every n-vertex graph in the class has an (α, β, f) vertex separator.

9. Generalized Nested-Dissection Orderings

When G_A is not a regular mesh, the analysis becomes much harder. When applied to general graphs, the ordering framework that we described in which a small balanced vertex separator is ordered last and the connected components are ordered recursively is called *generalized nested dissection*.

DEFINITION 9.1. A class of graphs satisfies an (α, β, f) *vertex-separator theorem* (or a separator theorem for short) if every n-vertex graph in the class has an (α, β, f) vertex separator.

For example, planar graphs satisfy an $(2/3, \sqrt{8}, \sqrt{n})$ separator theorem. Since nested dissection orders subgraphs recursively, we must ensure that subgraphs belong to the same class of graphs, so that they can be ordered effectively as well.

DEFINITION 9.2. A class of graphs is said to be *closed under subgraph* if every subgraph of a graph in the class is also in the class.

There are two ways to use small balanced vertex separators to order an arbitrary graph. The first algorithm, which we call the LRT algorithm, guarantees an effective ordering for graphs belonging to a class that satisfies a separator theorem and is closed under subgraph. The algorithm receives an input a range $[i, j]$ of indices and graph G with n vertices, ℓ of which may already have been ordered. If $\ell > 0$, then the ordered vertices have indices $j - \ell + 1, \ldots, j$. The algorithm assigns the rest of the indices, $i, \ldots, j - \ell$, to the unnumbered vertices. The algorithm works as follows.

(1) If n is small enough, $n \leq (\beta(1-\alpha))^2$, the algorithm orders the unnumbered vertices arbitrarily and returns.
(2) The algorithm finds a small balanced vertex separator S in G. The separator separates the graph into subgraphs with vertex sets V_1 and V_2. The two subgraphs may contain more than one connected componet each.
(3) The algorithm arbitrarily assigns indices $j-\ell-|S|+1, \ldots, j-\ell$ to the vertices of S.
(4) The algorithm recurses on the subgraphs induced by $V_1 \cup S$ and by $V_2 \cup S$. The unnumbered vertices in the first subgraph are assigned the middle range of $[i, j]$ and the second the first range (starting from i).

We initialize the algorithm by setting $\ell = 0$ and $[i, j] = [1, n]$.

The second algorithm, called the GT algorithm, finds a separator, orders its vertices last, and then recurses on each connected component of the graph with the separator removed; the vertices of each component are ordered consecutively. In each recursive call the algorithm finds a separator that separates the vertices of one component, ignoring the rest of the graph. This algorithm does not guarantee an effective orderings to any graph belonging to a class that satisfies a separator theorem and is closed under subgraph; additional conditions on the graphs are required, such as bounded degree or closure under edge contractions. Therefore, we analyze the first algorithm.

THEOREM 9.3. *Let G be a graph belonging to a class that satisfies an $(\alpha, \beta, \sqrt{n})$ separator theroem and closed under subgraph. Ordering the vertices of G using the LRT algorithm leades to $O(n \log n)$ fill edges.*

PROOF. We prove the theorem by induction on n. If $n \leq n_0$, the theorem holds by setting the constant c in the big-O notation high enough so that $cn \log n > n(n-1)/2$ for all $n \leq n_0$. Suppose that the theorem holds for all graphs with fewer than n vertices.

The algorithm finds a separator S that splits the graph into subgraphs induced by V_1 and by V_2. We partition the fill edges into four categories: (1) fill edges with two endpoints in S; (2) fill edges with one endpoint in S and the other in one of the ℓ already-numbered vertices; (3,4) fill edges with at least one endpoint in V_2 or in V_2. Since there are no fill edges with one endpoint in V_1 and the other in V_2, categories 3 and 4 are indeed disjoint.

The number of fill edges in Category 1 is at most $|S|(|S|-1)/2 \leq \beta^2 n/2$.

The number of fill edges in Category 2 is at most $|S|\ell \leq \beta\ell\sqrt{n}$.

Let $\bar{\eta}(n, \ell)$ be an upper bound on the number of fill edges in the ordering produced by the algorithm on a graph with n vertices, ℓ of which are already numbered.

The number of fill edges in Category 3 is at most $\bar{\eta}(|V_1| + |S|, \ell_1)$, where ℓ_1 is the number of already-numbered vertices in $V_1 \cup S$ *after the vertices of S have been numbered*. Note that ℓ_1 may be smaller than $\ell + |S|$ because some of the ℓ vertices that are initially numbered be in V_2. Similarly, the number of fill edges in Category 4 is at most $\bar{\eta}(|V_2| + |S|, \ell_2)$.

By summing the bounds for the four categories, we obtain a recurrence on $\bar{\eta}$,

$$\bar{\eta}(n, \ell) \leq \frac{\beta^2 n}{2} + \beta\ell\sqrt{n} + \bar{\eta}(|V_1| + |S|, \ell_1) + \bar{\eta}(|V_2| + |S|, \ell_2) \ . \tag{1}$$

We claim that

$$\bar{\eta}\,(n,\ell) \le c'(n+\ell)\log_2 n + c''\ell\sqrt{n} \tag{2}$$

for some constants c' and c''. Since initially $\ell = 0$, this bound implies the $O(n \log n)$ bound on fill.

To prove the bound, we denote $n_1 = |V_1| + |S|$ and $n_2 = |V_2| + |S|$ and note the following bounds:

$$\begin{array}{rcccl} & & \ell_1 + \ell_2 & \le & \ell + 2\beta\sqrt{n} \\ n & \le & n_1 + n_2 & \le & n + \beta\sqrt{n} \\ (1-\alpha)n & \le & n_1, n_2 & \le & \alpha n + \beta\sqrt{n} \end{array}$$

The first inequality follows from the fact that every alread-numbered vertex in the input to the two recursive calls is either one of the ℓ initially-numbered vertices or a vertex of S. An initially-numbered vertex that is not in S is passed to only one of the recursive calls; vertices in S are be passed as already numbered to the two calls, but there are at most $\beta\sqrt{n}$ of them. The second inequality follows from the fact that the subgraphs passed to the two recursive calls contain together all the vertices in $V = S \cup V_1 \cup V_2$ and that $|S| \le \beta\sqrt{n}$ vertices are passed to the both of the recursive calls. The third inequality follows from the guarantees on $|V_1|$, $|V_2|$, and $|S|$ under the separator theroem.

We now prove the claim (2) by induction on n. For n smaller than some constant size, we can ensure that the claim holds simply by choosing c' and c'' to be large enough. In particular, $\bar{\eta}\,(n,\ell) \le n(n-1)/2$, which is smaller than the right-hand side of (2) for small enough n and large enough c' and c''.

For larger n, we use Equation (1) and invoke the inductive assumption regarding the correctness of (2),

$$\begin{aligned} \bar{\eta}\,(n,\ell) &\le \frac{\beta^2 n}{2} + \beta\ell\sqrt{n} + \bar{\eta}\,(|V_1| + |S|, \ell_1) + \bar{\eta}\,(|V_2| + |S|, \ell_2) \\ &\le \frac{\beta^2 n}{2} + \beta\ell\sqrt{n} + c'\,(n_1 + \ell_1)\log_2 n_1 + c''\ell_1\sqrt{n_1} + c'\,(n_2 + \ell_2)\log_2 n_2 + c''\ell_2\sqrt{n_2}\,. \end{aligned}$$

The rest of the proof only involves manipulations of the expression in the second line to show that for large enough c' and c'', it is bounded by (2). We omit the details. □

A similar analysis yields an analysis on arithmetic operations.

THEOREM 9.4. *Let G be a graph belonging to a class that satisfies an $(\alpha, \beta, \sqrt{n})$ separator theroem and closed under subgraph. Ordering the vertices of G using the LRT algorithm leads to $O(n^{1.5})$ arithmetic operations in the algorithm.*

These results can be applied directly to matrices whose pattern graphs are planar, and more generally to graphs that can be embedded in surfaces with a bounded genus. Such graphs are closed under subgraph and satisfy a $(2/3, \sqrt{8}, \sqrt{n})$ separator theorem. Furthermore, the separator in an n-vertex graph of this family can be found in $O(n)$ operations, and it is possible to show that we can even find *all* the separators required in all the levels of the recursive algorithm in $O(n \log n)$ time.

For the GT algorithm, which is somewhat easier to implement and which finds separators faster (because the separator is not included in the two subgraphs to be recursively partitioned), a separator theorem is not suffiecient to guarantee comparable bounds on fill and work. To ensure that the algorithm leads to similar asymptotic bounds, one must also assume that the graphs are closed under edge contraction or that they have a bounded degree. However, the two algorithms have roughly the same applicability, because planar graphs and graphs that can be embedded on surfaces with a bounded genus are closed under edge contraction. It is not clear which of the two algorithms is better in practice.

The fill and work results for both the LRT and the GT algorithms can be extended to graphs that satisfy separator theorems with separators smaller or larger than $\sqrt{n}$. See the original articles for the asymptotic bounds.

Finally, we mention that an algorithm that finds approximately optimal balanced vertex separators can be used to find a permutation that approximately minimizes fill and work in sparse Cholesky. The algorithm is similar in principle to the LRT and GT algorithms. This result shows that up to a polylogarithmic factors, the quality of vertex separators in a graph determines sparsity that we can achieve in sparse Cholesky.

10. Notes and References

Bandwidth and envelope reduction orderings.

The idea of nested dissection, and the analysis of nested dissection on regular meshes is due to George XXX. The LRT generalized-nested-dissection algorithm and its analysis are due to Lipton, Rose, and Tarjan XXX. The GT algorithm and its analysis are due to Gilbert and Tarjan XXX.

Exercises

EXERCISE 11. The proof of Theorem 2.2 was essentially based on the fact that the following algorithm is a correct implementation of the Cholesky factorization:

```
S = A
for j = 1:n
  L_{j,j} = sqrt(S_{j,j})
  L_{j+1:n,j} = S_{j+1:n,j} / L_{j,j}
  for i = j+1:n
    for k = j+1:n
      S_{i,k} = S_{i,k} - L_{i,j} L_{k,j}
    end
  end
end
```

(1) Show that this implementation is correct.
(2) Show that in each outer iteration, the code inside the inner loop, the k loop, performs $\eta^2(L_{j+1:n,j})$ nontrivial subtractions (subtractions in which a nonzero value is subtracted).
(3) This implementation of the Cholesky factorization is often called jik Cholesky, because the ordering of the loops is j first (outermost), then j, and finally k, the inner loop. In fact, all 6 permutations of the loop indices yield correct Cholesky factorizations; the expression inside the inner loop should is the same in all the permutations. Show this by providing 6 appropriate MATLAB functions.
(4) For each of the 6 permutations, consider the middle iteration of the outer loop. Sketch the elements of A, the elements of L, and the elements of S that are referenced during this iteration of the outer loop. For the jik loop ordering shown above, A is not referenced inside outer loop, and for L and S the sketches shown above.

EXERCISE 12. In this exercise we explore fill and work in the Cholesky factorization of banded and low-profile matrices. We say that a matrix has a half-bandwidth k if for all $i < j - k$ we have $A_{j,i} = 0$.

(1) Suppose that A corresponds to an x-by-y two-dimensional mesh whose vertices are ordered as in the previous chapter. What is the half bandwidth of A? What happens when x is larger than y, and what happens when y is larger? Can you permute A to achieve a minimal bandwidth in both cases?

(2) Show that in the factorization of a banded matrix, all the fill occurs within the band. That is, L is also banded with the same bound on its half bandwidth.
(3) Compute a bound on fill and work as a function of n and k.
(4) In some cases, A is banded but with a large bandwidth, but in most of the rows and/or columns all the nonzeros are closer to the main diagonal than predicted by the half-bandwidth. Can you derive an improved bound that takes into account the local bandwidth of each row and/or column? In particular, you need to think about whether a bound on the rows or on the columns, say in the lower triangular part of A, is more useful.
(5) We ordered the vertices of our meshes in a way that matrices come out banded. In many applications, the matrices are not banded, but they can be symmetrically permuted to a banded form. Using an x-by-y mesh with $x \gg y$ as a motivating example, develop a graph algorithm that finds a permutation P that clusters the nonzeros of PAP^T near the main diagonal. Use a breadth-first-search as a starting point. Consider the previous quesion in the exercise as you refine the algorithm.
(6) Given an x-by-y mesh with $x \gg y$, derive bounds for fill and work in a generalized-nested-dissection factorization of the matrix corresponding to the mesh. Use separators that approximately bisect along the x dimention until you reach subgraphs with a square aspect ratio. Give the bounds in terms of x and y.