SIAM J. MATRIX ANAL. APPL.
Vol. 22, No. 4, pp. 997–1013

MULTIPLE-RANK MODIFICATIONS OF A SPARSE CHOLESKY FACTORIZATION*

TIMOTHY A. DAVIS† AND WILLIAM W. HAGER‡

Abstract. Given a sparse symmetric positive definite matrix $\mathbf{AA}^\mathsf{T}$ and an associated sparse Cholesky factorization $\mathbf{LDL}^\mathsf{T}$ or $\mathbf{LL}^\mathsf{T}$, we develop sparse techniques for updating the factorization after either adding a collection of columns to $\mathbf{A}$ or deleting a collection of columns from $\mathbf{A}$. Our techniques are based on an analysis and manipulation of the underlying graph structure, using the framework developed in an earlier paper on rank-1 modifications [T. A. Davis and W. W. Hager, *SIAM J. Matrix Anal. Appl.*, 20 (1999), pp. 606–627]. Computationally, the multiple-rank update has better memory traffic and executes much faster than an equivalent series of rank-1 updates since the multiple-rank update makes one pass through $\mathbf{L}$ computing the new entries, while a series of rank-1 updates requires multiple passes through $\mathbf{L}$.

Key words. numerical linear algebra, direct methods, Cholesky factorization, sparse matrices, mathematical software, matrix updates

AMS subject classifications. 65F05, 65F50, 65-04

PII. S0895479899357346

1. Introduction.

This paper presents a method for evaluating a multiple-rank update or downdate of the sparse Cholesky factorization $\mathbf{LDL}^\mathsf{T}$ or $\mathbf{LL}^\mathsf{T}$ of the matrix $\mathbf{AA}^\mathsf{T}$, where $\mathbf{A}$ is $m \times n$. More precisely, given an $m \times r$ matrix $\mathbf{W}$, we evaluate the Cholesky factorization of $\mathbf{AA}^\mathsf{T} + \sigma\mathbf{WW}^\mathsf{T}$ where either σ is $+1$ (corresponding to an update) and $\mathbf{W}$ is arbitrary, or σ is -1 (corresponding to a downdate) and $\mathbf{W}$ consists of columns of $\mathbf{A}$. Both $\mathbf{AA}^\mathsf{T}$ and $\mathbf{AA}^\mathsf{T} + \sigma\mathbf{WW}^\mathsf{T}$ must be positive definite. It follows that $n \geq m$ in the case of an update, and $n - r \geq m$ in the case of a downdate.

One approach to the multiple-rank update is to express it as a series of rank-1 updates and use the theory developed in [10] for updating a sparse factorization after a rank-1 change. This approach, however, requires multiple passes through $\mathbf{L}$ as it is updated after each rank-1 change. In this paper, we develop a sparse factorization algorithm that makes only one pass through $\mathbf{L}$.

For a dense Cholesky factorization, a one-pass algorithm to update a factorization is obtained from Method C1 in [18] by making all the changes associated with one column of $\mathbf{L}$ before moving to the next column, as is done in the following algorithm that overwrites $\mathbf{L}$ and $\mathbf{D}$ with the new factors of $\mathbf{AA}^\mathsf{T} + \sigma\mathbf{WW}^\mathsf{T}$. Algorithm 1 performs $2rm^2 + 4rm$ floating-point operations.

ALGORITHM 1 (dense rank-r update/downdate).

for $i = 1$ **to** r **do**
  $\alpha_i = 1$
end for
for $j = 1$ **to** m **do**
  for $i = 1$ **to** r **do**

*Received by the editors June 17, 1999; accepted for publication (in revised form) by S. Vavasis August 16, 2000; published electronically January 31, 2001. This work was supported by the National Science Foundation.

http://www.siam.org/journals/simax/22-4/35734.html

†Department of Computer and Information Science and Engineering, University of Florida, P.O. Box 116120, Gainesville, FL 32611-6120 (davis@cise.ufl.edu, http://www.cise.ufl.edu/~davis).

‡Department of Mathematics, University of Florida, P.O. Box 118105, Gainesville, FL 32611-8105 (hager@math.ufl.edu, http://www.math.ufl.edu/~hager).

```
            ᾱ = α_i + σ w_ji^2 / d_j  (σ = +1 for update or −1 for downdate)
            d_j = d_j ᾱ
            γ_i = w_ji / d_j
            d_j = d_j / α_i
            α_i = ᾱ
        end for
        for p = j + 1 to m do
            for i = 1 to r do
                w_pi = w_pi − w_ji l_pj
                l_pj = l_pj + σ γ_i w_pi
            end for
        end for
    end for
```

We develop a sparse version of this algorithm that only accesses and modifies those entries in $\mathbf{L}$ and $\mathbf{D}$ which can change. For $r = 1$, the theory in our rank-1 paper [10] shows that those columns which can change correspond to the nodes in an elimination tree on a path starting from the node k associated with the first nonzero element w_{k1} in $\mathbf{W}$. For $r > 1$ we show that the columns of $\mathbf{L}$ which can change correspond to the nodes in a subtree of the elimination tree, and we express this subtree as a modification of the elimination tree of $\mathbf{AA}^{\mathsf{T}}$. Also, we show that with a reordering of the columns of $\mathbf{W}$, it can be arranged so that in the inner loop where elements in row p of $\mathbf{W}$ are updated, the elements that change are adjacent to each other. The sparse techniques that we develop lead to sequential access of matrix elements and to efficient computer memory traffic. These techniques to modify a sparse factorization have many applications, including the linear program dual active set algorithm (LPDASA) [20], least-squares problems in statistics, the analysis of electrical circuits and power systems, structural mechanics, sensitivity analysis in linear programming, boundary condition changes in partial differential equations, domain decomposition methods, and boundary element methods (see [19]).

Section 2 describes our notation. In section 3, we present an algorithm for computing the *symbolic factorization* of $\mathbf{AA}^{\mathsf{T}}$ using multisets, which determines the location of nonzero entries in $\mathbf{L}$. Sections 4 and 5 describe our multiple-rank symbolic update and downdate algorithms for finding the nonzero pattern of the new factors. Section 6 describes our algorithm for computing the new numerical values of $\mathbf{L}$ and $\mathbf{D}$, for either an update or downdate. Our experimental results are presented in section 7.

2. Notation and background. Given the location of the nonzero elements of $\mathbf{AA}^{\mathsf{T}}$, we can perform a *symbolic factorization* (this terminology is introduced by George and Liu in [15]) of the matrix to predict the location of the nonzero elements of the Cholesky factor $\mathbf{L}$. In actuality, some of these predicted nonzeros may be zero due to numerical cancellation during the factorization process. The statement "$l_{ij} \neq 0$" will mean that l_{ij} is *symbolically* nonzero. The main diagonals of $\mathbf{L}$ and $\mathbf{D}$ are always nonzero since the matrices that we factor are positive definite (see [26, p. 253]). The nonzero pattern of column j of $\mathbf{L}$ is denoted $\mathcal{L}_j$,

$$\mathcal{L}_j = \{i \,:\, l_{ij} \neq 0\},$$

while $\mathcal{L}$ denotes the collection of patterns

$$\mathcal{L} = \{\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_m\}.$$

Similarly, $\mathcal{A}_j$ denotes the nonzero pattern of column j of $\mathbf{A}$,

$$\mathcal{A}_j = \{i \,:\, a_{ij} \neq 0\},$$

while $\mathcal{A}$ is the collection of patterns

$$\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n\}.$$

The *elimination tree* can be defined in terms of a *parent map* π (see [22]). For any node j, $\pi(j)$ is the row index of the first nonzero element in column j of $\mathbf{L}$ beneath the diagonal element

$$\pi(j) = \min\ \mathcal{L}_j \setminus \{j\},$$

where "min $\mathcal{X}$" denotes the smallest element of $\mathcal{X}$:

$$\min\ \mathcal{X} = \min_{i \in \mathcal{X}}\ i.$$

Our convention is that the min of the empty set is zero. Note that $j < \pi(j)$ except in the case where the diagonal element in column j is the only nonzero element. The children of node j is the set of nodes whose parent is j:

$$\{c : j = \pi(c)\}.$$

The *ancestors* of a node j, denoted $\mathcal{P}(j)$, is the set of successive parents:

$$\mathcal{P}(j) = \{j, \pi(j), \pi(\pi(j)), \ldots\}.$$

Since $\pi(j) > j$ for each j, the ancestor sequence is finite. The sequence of nodes $j, \pi(j), \pi(\pi(j)), \ldots$, forming $\mathcal{P}(j)$, is called the *path* from j to the associated tree root, the final node on the path. The collection of paths leading to a root form an *elimination tree*. The set of all trees is the *elimination forest*. Typically, there is a single tree whose root is m; however, if column j of $\mathbf{L}$ has only one nonzero element, the diagonal element, then j will be the root of a separate tree.

The number of elements (or size) of a set $\mathcal{X}$ is denoted $|\mathcal{X}|$, while $|\mathcal{A}|$ or $|\mathcal{L}|$ denote the sum of the sizes of the sets they contain.

3. Symbolic factorization. For a matrix of the form $\mathbf{A}\mathbf{A}^{\mathsf{T}}$, the pattern $\mathcal{L}_j$ of column j is the union of the patterns of each column of $\mathbf{L}$ whose parent is j and each column of $\mathbf{A}$ whose smallest row index of its nonzero entries is j (see [16, 22]):

$$\mathcal{L}_j = \{j\} \cup \left(\bigcup_{\{c : j = \pi(c)\}} \mathcal{L}_c \setminus \{c\} \right) \cup \left(\bigcup_{\min\ \mathcal{A}_k = j} \mathcal{A}_k \right). \tag{3.1}$$

To modify (3.1) during an update or downdate, without recomputing it from scratch, we need to keep track of how each entry i entered into $\mathcal{L}_j$ [10]. For example, if $\pi(c)$ changes, we may need to remove a term $\mathcal{L}_c \setminus \{c\}$. We cannot simply perform a set subtraction, since we may remove entries that appear in other terms. To keep track of how entries enter and leave the set $\mathcal{L}_j$, we maintain a multiset associated with column j. It has the form

$$\mathcal{L}_j^{\sharp} = \{(i, m(i,j)) : i \in \mathcal{L}_j\},$$

where the multiplicity $m(i,j)$ is the number of children of j that contain row index i in their pattern plus the number of columns of $\mathcal{A}$ whose smallest entry is j and that contain row index i. Equivalently, for $i \neq j$,

$$m(i,j) = |\{c : j = \pi(c) \text{ and } i \in \mathcal{L}_c\}| + |\{k : \min\ \mathcal{A}_k = j \text{ and } i \in \mathcal{A}_k\}|\,.$$

For $i = j$, we increment the above equation by one to ensure that the diagonal entries never disappear during a downdate. The set $\mathcal{L}_j$ is obtained from $\mathcal{L}_j^\sharp$ by removing the multiplicities.

We define the addition of a multiset $\mathcal{X}^\sharp$ and a set $\mathcal{Y}$ in the following way:

$$\mathcal{X}^\sharp + \mathcal{Y} = \{(i, m'(i))\ :\ i \in \mathcal{X} \text{ or } i \in \mathcal{Y}\},$$

where

$$m'(i) = \begin{cases} 1 & \text{if } i \notin \mathcal{X} \text{ and } i \in \mathcal{Y}, \\ m(i) & \text{if } i \in \mathcal{X} \text{ and } i \notin \mathcal{Y}, \\ m(i)+1 & \text{if } i \in \mathcal{X} \text{ and } i \in \mathcal{Y}. \end{cases}$$

Similarly, the subtraction of a set $\mathcal{Y}$ from a multiset $\mathcal{X}^\sharp$ is defined by

$$\mathcal{X}^\sharp - \mathcal{Y} = \{(i, m'(i))\ :\ i \in \mathcal{X} \text{ and } m'(i) > 0\},$$

where

$$m'(i) = \begin{cases} m(i) & \text{if } i \notin \mathcal{Y}, \\ m(i)-1 & \text{if } i \in \mathcal{Y}. \end{cases}$$

The multiset subtraction of $\mathcal{Y}$ from $\mathcal{X}^\sharp$ undoes a prior addition. That is, for any multiset $\mathcal{X}^\sharp$ and any set $\mathcal{Y}$, we have

$$((\mathcal{X}^\sharp + \mathcal{Y}) - \mathcal{Y}) = \mathcal{X}^\sharp.$$

In contrast $((\mathcal{X} \cup \mathcal{Y}) \setminus \mathcal{Y})$ is equal to $\mathcal{X}$ if and only if $\mathcal{X}$ and $\mathcal{Y}$ are disjoint sets.

Using multiset addition instead of set union, (3.1) leads to the following algorithm for computing the symbolic factorization of $\mathbf{A}\mathbf{A}^\mathsf{T}$.

ALGORITHM 2 (symbolic factorization of $\mathbf{A}\mathbf{A}^\mathsf{T}$, using multisets).

```
for j = 1 to m do
    L_j^# = {(j, 1)}
    for each c such that j = π(c) do
        L_j^# = L_j^# + (L_c \ {c})
    end for
    for each k where min A_k = j do
        L_j^# = L_j^# + A_k
    end for
    π(j) = min L_j \ {j}
end for
```

4. Multiple-rank symbolic update. We consider how the pattern $\mathcal{L}$ changes when $\mathbf{A}\mathbf{A}^\mathsf{T}$ is replaced by $\mathbf{A}\mathbf{A}^\mathsf{T} + \mathbf{W}\mathbf{W}^\mathsf{T}$. Since

$$\mathbf{A}\mathbf{A}^\mathsf{T} + \mathbf{W}\mathbf{W}^\mathsf{T} = [\mathbf{A}|\mathbf{W}][\mathbf{A}|\mathbf{W}]^\mathsf{T},$$

we can in essence augment $\mathbf{A}$ by $\mathbf{W}$ in order to evaluate the new pattern of column j in $\mathbf{L}$. According to (3.1), the new pattern $\overline{\mathcal{L}}_j$ of column j of $\mathbf{L}$ after the update is

$$\overline{\mathcal{L}}_j = \{j\} \cup \left(\bigcup_{\{c: j=\overline{\pi}(c)\}} \overline{\mathcal{L}}_c \setminus \{c\} \right) \cup \left(\bigcup_{\min\, \mathcal{A}_k = j} \mathcal{A}_k \right) \cup \left(\bigcup_{\min\, \mathcal{W}_i = j} \mathcal{W}_i \right), \tag{4.1}$$

where $\mathcal{W}_i$ is the pattern of column i in $\mathbf{W}$. Throughout, we put a bar over a matrix or a set to denote its new value after the update or downdate.

In the following theorem, we consider a column j of the matrix $\mathbf{L}$ and how its pattern is modified by the sets $\mathcal{W}_i$. Let $\overline{\mathcal{L}}_j^\sharp$ denote the multiset for column j after the rank-r update or downdate has been applied.

THEOREM 4.1. *To compute the new multiset $\overline{\mathcal{L}}_j^\sharp$, initialize $\overline{\mathcal{L}}_j^\sharp = \mathcal{L}_j^\sharp$ and perform the following modifications.*

- *Case A: For each i such that $j = \min \mathcal{W}_i$, add $\mathcal{W}_i$ to the pattern for column j,*

$$\overline{\mathcal{L}}_j^\sharp = \overline{\mathcal{L}}_j^\sharp + \mathcal{W}_i.$$

- *Case B: For each c such that $j = \pi(c) = \overline{\pi}(c)$, compute*

$$\overline{\mathcal{L}}_j^\sharp = \overline{\mathcal{L}}_j^\sharp + (\overline{\mathcal{L}}_c \setminus \mathcal{L}_c)$$

  (c is a child of j in both the old and new elimination tree).
- *Case C: For each c such that $j = \overline{\pi}(c) \neq \pi(c)$, compute*

$$\overline{\mathcal{L}}_j^\sharp = \overline{\mathcal{L}}_j^\sharp + (\overline{\mathcal{L}}_c \setminus \{c\})$$

  (c is a child of j in the new tree, but not the old one).
- *Case D: For each c such that $j = \pi(c) \neq \overline{\pi}(c)$, compute*

$$\overline{\mathcal{L}}_j^\sharp = \overline{\mathcal{L}}_j^\sharp - (\mathcal{L}_c \setminus \{c\})$$

  (c is a child of j in the old tree, but not the new one).

Proof. Cases A–D account for all the adjustments we need to make in $\mathcal{L}_j$ in order to obtain $\overline{\mathcal{L}}_j$. These adjustments are deduced from a comparison of (3.1) with (4.1). In case A, we simply add in the $\mathcal{W}_i$ multisets of (4.1) that do not appear in (3.1). In case B, node c is a child of node j both before and after the update. In this case, we must adjust for the deviation between $\overline{\mathcal{L}}_c$ and $\mathcal{L}_c$. By [10, Prop. 3.2], after a rank-1 update, $\mathcal{L}_c \subseteq \overline{\mathcal{L}}_c$. If $\mathbf{w}_i$ denotes the ith column of $\mathbf{W}$, then

$$\mathbf{W}\mathbf{W}^\mathsf{T} = \mathbf{w}_1\mathbf{w}_1^\mathsf{T} + \mathbf{w}_2\mathbf{w}_2^\mathsf{T} + \cdots + \mathbf{w}_r\mathbf{w}_r^\mathsf{T}.$$

Hence, updating $\mathbf{A}\mathbf{A}^\mathsf{T}$ by $\mathbf{W}\mathbf{W}^\mathsf{T}$ is equivalent to r successive rank-1 updates of $\mathbf{A}\mathbf{A}^\mathsf{T}$. By repeated application of [10, Prop. 3.2], $\mathcal{L}_c \subseteq \overline{\mathcal{L}}_c$ after a rank-r update of $\mathbf{A}\mathbf{A}^\mathsf{T}$. It follows that $\overline{\mathcal{L}}_c$ and $\mathcal{L}_c$ deviate from each other by the set $\overline{\mathcal{L}}_c \setminus \mathcal{L}_c$. Consequently, in case B we simply add in $\overline{\mathcal{L}}_c \setminus \mathcal{L}_c$.

In case C, node c is a child of j in the new elimination tree, but not in the old tree. In this case we need to add in the entire set $\overline{\mathcal{L}}_c \setminus \{c\}$ since the corresponding term does not appear in (3.1). Similarly, in case D, node c is a child of j in the old elimination tree, but not in the new tree. In this case, the entire set $\mathcal{L}_c \setminus \{c\}$ should be deleted. The case where c is not a child of j in either the old or the new elimination

tree does not result in any adjustment since the corresponding $\mathcal{L}_c$ term is absent from both (3.1) and (4.1). ∎

An algorithm for updating a Cholesky factorization that is based only on this theorem would have to visit all nodes j from 1 to m, and consider all possible children $c < j$. On the other hand, not all nodes j from 1 to m need to be considered since not all columns of $\mathbf{L}$ change when $\mathbf{A}\mathbf{A}^\mathsf{T}$ is modified. In [10, Thm. 4.1] we show that for $r = 1$, the nodes whose patterns can change are contained in $\overline{\mathcal{P}}(k_1)$, where we define $k_i = \min \mathcal{W}_i$. For a rank-r update, let $\mathcal{P}^{(i)}$ be the ancestor map associated with the elimination tree for the Cholesky factorization of the matrix

$$\mathbf{A}\mathbf{A}^\mathsf{T} + \sum_{j=1}^{i} \mathbf{w}_j \mathbf{w}_j^\mathsf{T}. \tag{4.2}$$

Again, by [10, Thm. 4.1], the nodes whose patterns can change during the rank-r update are contained in the union of the patterns $\mathcal{P}^{(i)}(k_i)$, $1 \le i \le r$. Although we could evaluate $\mathcal{P}^{(i)}(k_i)$ for each i, it is difficult to do this efficiently since we need to perform a series of rank-1 updates and evaluate the ancestor map after each of these. On the other hand, by [10, Prop. 3.1] and [10, Prop. 3.2], $\mathcal{P}^{(i)}(j) \subseteq \mathcal{P}^{(i+1)}(j)$ for each i and j, from which it follows that $\mathcal{P}^{(i)}(k_i) \subseteq \overline{\mathcal{P}}(k_i)$ for each i. Consequently, the nodes whose patterns change during a rank-r update are contained in the set

$$\overline{\mathcal{T}} = \bigcup_{1 \le i \le r} \overline{\mathcal{P}}(k_i).$$

Theorem 4.2, below, shows that any node in $\overline{\mathcal{T}}$ is also contained in one or more of the sets $\mathcal{P}^{(i)}(k_i)$. From this it follows that the nodes in $\overline{\mathcal{T}}$ are precisely those nodes for which entries in the associated columns of $\mathbf{L}$ can change during a rank-r update. Before presenting the theorem, we illustrate this with a simple example shown in Figure 4.1. The left of Figure 4.1 shows the sparsity pattern of original matrix $\mathbf{A}\mathbf{A}^\mathsf{T}$, its Cholesky factor $\mathbf{L}$, and the corresponding elimination tree. The nonzero pattern of the first column of $\mathbf{W}$ is $\mathcal{W}_1 = \{1, 2\}$. If performed as a single rank-1 update, this causes a modification of columns 1, 2, 6, and 8 of $\mathbf{L}$. The corresponding nodes in the original tree are encircled; these nodes form the path $\mathcal{P}^{(1)}(1) = \{1, 2, 6, 8\}$ from node 1 to the root (node 8) in the second tree. The middle of Figure 4.1 shows the matrix after this rank-1 update, and its factor and elimination tree. The entries in the second matrix $\mathbf{A}\mathbf{A}^\mathsf{T} + \mathbf{w}_1\mathbf{w}_1^\mathsf{T}$ that differ from the original matrix $\mathbf{A}\mathbf{A}^\mathsf{T}$ are shown as small pluses. The second column of $\mathbf{W}$ has the nonzero pattern $\mathcal{W}_2 = \{3, 4, 7\}$. As a rank-1 update, this affects columns $\mathcal{P}^{(2)}(3) = \overline{\mathcal{P}}(3) = \{3, 4, 5, 6, 7, 8\}$ of $\mathbf{L}$. These columns form a single path in the final elimination tree shown in the right of the figure.

For the first rank-1 update, the set of columns that actually change are $\mathcal{P}^{(1)}(1) = \{1, 2, 6, 8\}$. This is a subset of the path $\overline{\mathcal{P}}(1) = \{1, 2, 6, 7, 8\}$ in the final tree. If we use $\overline{\mathcal{P}}(1)$ to guide the work associated with column 1 of $\mathbf{W}$, we visit all the columns that need to be modified, plus column 7. Node 7 is in the set of nodes $\overline{\mathcal{P}}(3)$ affected by the second rank-1 update, however, as shown in the following theorem.

THEOREM 4.2. *Each of the paths $\mathcal{P}^{(i)}(k_i)$ is contained in $\overline{\mathcal{T}}$ and conversely, if $j \in \overline{\mathcal{T}}$, then j is contained in $\mathcal{P}^{(i)}(k_i)$ for some i.*

Proof. Before the theorem, we observe that each of the paths $\mathcal{P}^{(i)}(k_i)$ is contained in $\overline{\mathcal{T}}$. Now suppose that some node j lies in the tree $\overline{\mathcal{T}}$. We need to prove that it is contained in $\mathcal{P}^{(i)}(k_i)$ for some i. Let s be the largest integer such that $\overline{\mathcal{P}}(k_s)$ contains j, and let c be any child of j in $\overline{\mathcal{T}}$. If c lies on the path $\overline{\mathcal{P}}(k_i)$ for some i, then j lies

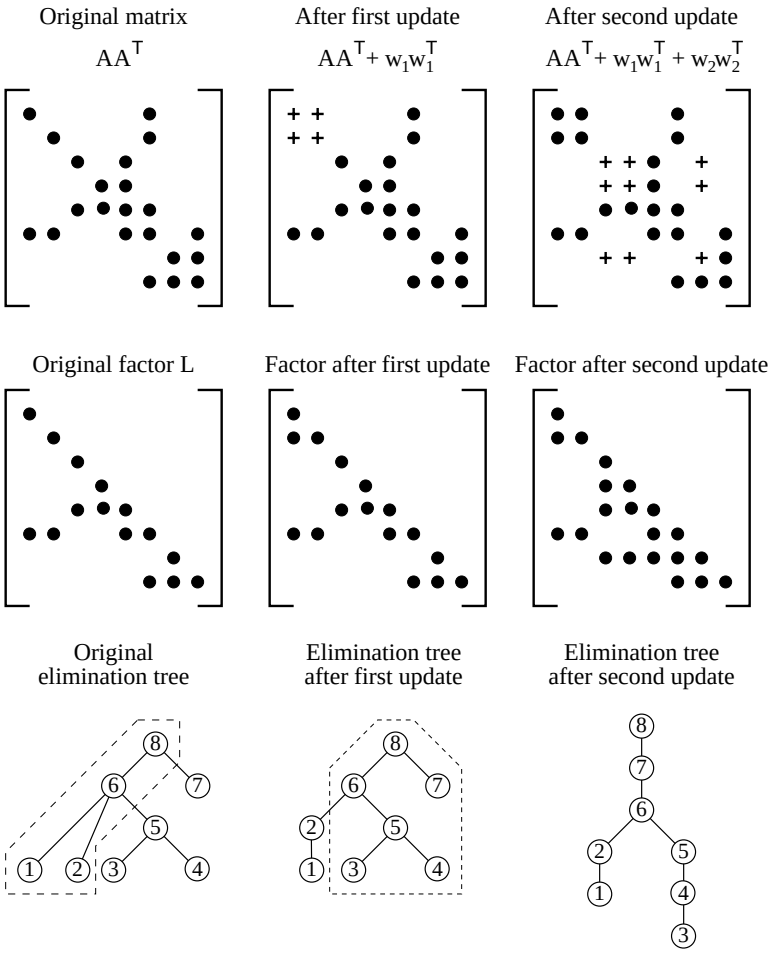

FIG. 4.1. *Example rank-2 update.*

on the path $\overline{\mathcal{P}}(k_i)$ since j is the parent of c. Since j does not lie on the path $\overline{\mathcal{P}}(k_i)$ for any $i > s$, it follows that c does not lie on the path $\overline{\mathcal{P}}(k_i)$ for any $i > s$. Applying this same argument recursively, we conclude that none of the nodes on the subtree of $\overline{\mathcal{T}}$ rooted at j lie on the path $\overline{\mathcal{P}}(k_i)$ for any $i > s$. Let $\overline{\mathcal{T}_j}$ denote the subtree of $\overline{\mathcal{T}}$ rooted at j. Since $\mathcal{P}^{(i)}(k_i)$ is contained in $\overline{\mathcal{P}}(k_i)$ for each i, none of the nodes of $\overline{\mathcal{T}_j}$ lie on any of the paths $\mathcal{P}^{(i)}(k_i)$ for $i > s$. By [10, Thm. 4.1], the patterns of all nodes outside the path $\mathcal{P}^{(i)}(k_i)$ are unchanged for each i. Let $\mathcal{L}_c^{(i)}$ be the pattern of column c in the Cholesky factorization of (4.2). Since any node c contained in $\overline{\mathcal{T}_j}$ does not lie on any of the paths $\mathcal{P}^{(i)}(k_i)$ for $i > s$, $\mathcal{L}_c^{(i)} = \mathcal{L}_c^{(l)}$ for all $i, l \geq s$. Since k_s is a node of $\overline{\mathcal{T}_j}$, the path $\mathcal{P}^{(s)}(k_s)$ must include j. □

Figure 4.2 depicts a subtree $\overline{\mathcal{T}}$ for an example rank-8 update. The subtree consists of all those nodes and edges in one or more of the paths $\overline{\mathcal{P}}(k_1), \overline{\mathcal{P}}(k_2), \ldots, \overline{\mathcal{P}}(k_8)$. These paths form a subtree, and not a general graph, since they are all paths from an initial node to the root of the elimination tree of the matrix $\overline{\mathbf{L}}$. The subtree $\overline{\mathcal{T}}$ might actually be a forest, if $\overline{\mathbf{L}}$ has an elimination forest rather than an elimination tree. The first nonzero positions in $\mathbf{w}_1$ through $\mathbf{w}_8$ correspond to nodes k_1 through k_8. For this

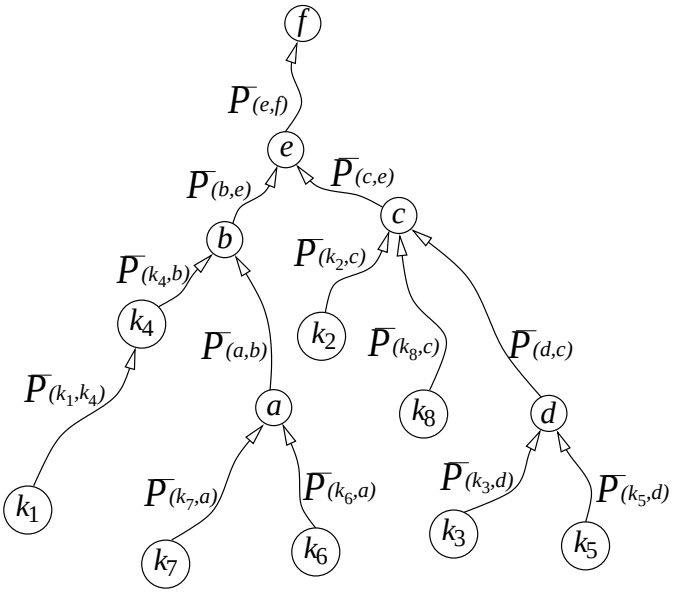

FIG. 4.2. *Example rank-*8 *symbolic update and subtree* $\overline{\mathcal{T}}$.

example, node k_4 happens to lie on the path $\mathcal{P}^{(1)}(k_1)$. Nodes at which paths first intersect are shown as smaller circles and are labeled a through f. Other nodes along the paths are not shown. Each curved arrow denotes a single subpath. For example, the arrow from nodes b to e denotes the subpath from b to e in $\overline{\mathcal{P}}(b)$. This subpath is denoted as $\overline{\mathcal{P}}(b,e)$ in Figure 4.2.

The following algorithm computes the rank-r symbolic update. It keeps track of an array of m "path-queues," one for each column of $\mathbf{L}$. Each queue contains a set of path-markers in the range 1 to r, which denote which of the paths $\overline{\mathcal{P}}(k_1)$ through $\overline{\mathcal{P}}(k_r)$ will modify column j next. If two paths have merged, only one of the paths needs to be considered. (We arbitrarily select the higher-numbered path to represent the merged paths.) This set of path-queues requires $O(m+r)$ space. Removing and inserting a path-marker in a path-queue takes $O(1)$ time. The only outputs of the algorithm are the new pattern of $\overline{\mathbf{L}}$ and its elimination tree, namely, $\overline{\mathcal{L}}_j^\sharp$ and $\overline{\pi}(j)$ for all $j \in [1,m]$. Not all columns are affected by the rank-r update. We define $\overline{\mathcal{L}}_j^\sharp = \mathcal{L}_j^\sharp$ and $\overline{\pi}(j) = \pi(j)$ for any node j not in $\overline{\mathcal{T}}$.

Case C will occur for c and j prior to visiting column $\pi(c)$, since $j = \overline{\pi}(c) < \pi(c)$. Thus we place c in the lost-child-queue of column $\pi(c)$ when encountering case C for nodes c and j. When the algorithm visits node $\pi(c)$, its lost-child-queue will contain all those nodes for which case D holds. This set of lost-child-queues is not the same as the set of path-queues (although there is exactly one lost-child-queue and one path-queue for each column j of $\mathbf{L}$).

ALGORITHM 3 (symbolic rank-r update; add new matrix $\mathbf{W}$).

Find the starting nodes of each path
for $i = 1$ **to** r **do**
  $\mathcal{W}_i = \{k : w_{ki} \neq 0\}$
  $k_i = \min\ \mathcal{W}_i$
  place path-marker i in path-queue of column k_i

end for
Consider all columns corresponding to nodes in the paths $\overline{\mathcal{P}}(k_1)$ *through* $\overline{\mathcal{P}}(k_r)$
for $j = \min_{i\in[1,r]} k_i$ **to** m **do**
  if path-queue of column j is nonempty **do**
    $\overline{\mathcal{L}}_j^\sharp = \mathcal{L}_j^\sharp$
    for each path-marker i on path-queue of column j **do**
      Path $\overline{\mathcal{P}}(k_i)$ *includes column* j
      Let c be the prior column on this path (if any), where $\overline{\pi}(c) = j$
      if $j = k_i$ **do**
        Case A: j *is the first node on the path* $\overline{\mathcal{P}}(k_i)$*, no prior* c
        $\overline{\mathcal{L}}_j^\sharp = \overline{\mathcal{L}}_j^\sharp + \mathcal{W}_i$
      else if $j = \pi(c)$ **then**
        Case B: c *is an old child of* j*, possibly changed*
        $\overline{\mathcal{L}}_j^\sharp = \overline{\mathcal{L}}_j^\sharp + (\overline{\mathcal{L}}_c \setminus \mathcal{L}_c)$
      else
        Case C: c *is a new child of* j *and a lost child of* $\pi(c)$
        $\overline{\mathcal{L}}_j^\sharp = \overline{\mathcal{L}}_j^\sharp + (\overline{\mathcal{L}}_c \setminus \{c\})$
        place c in lost-child-queue of column $\pi(c)$
      end if
    end for
    Case D: consider each lost child of j
    for each c in lost-child-queue of column j **do**
      $\overline{\mathcal{L}}_j^\sharp = \overline{\mathcal{L}}_j^\sharp - (\mathcal{L}_c \setminus \{c\})$
    end for
    Move up one step in the path(s)
    $\overline{\pi}(j) = \min\ \overline{\mathcal{L}}_j \setminus \{j\}$
    if $\overline{\mathcal{L}}_j \setminus \{j\} \neq \emptyset$ **then**
      Let i be the largest path-marker in path-queue of column j
      Place path-marker i in path-queue of column $\overline{\pi}(j)$
    end if
  end if *path-queue of column* j *nonempty*
end for

The optimal time for a general rank-r update is

$$O\left(\sum_{j\in\overline{\mathcal{T}}} |\overline{\mathcal{L}}_j|\right).$$

The actual time taken by Algorithm 3 is only slightly higher, namely,

$$O\left(m + \sum_{j\in\overline{\mathcal{T}}} |\overline{\mathcal{L}}_j|\right),$$

because of the $O(m)$ bookkeeping required for the path-queues. In most practical cases, the $O(m)$ term will not be the dominant term in the run time.

Algorithm 3 can be used to compute an entire symbolic factorization. We start by factorizing the identity matrix $\mathbf{I} = \mathbf{I}\mathbf{I}^\mathsf{T}$ into $\mathbf{L}\mathbf{D}\mathbf{L}^\mathsf{T} = \mathbf{I}\mathbf{I}\mathbf{I}$. In this case, we have $\mathcal{L}_j^\sharp = \{(j, 1)\}$ for all j. The initial elimination tree is a forest of m nodes and no

edges. We can now determine the symbolic factorization of $\mathbf{I}+\mathbf{A}\mathbf{A}^{\mathsf{T}}$ using the rank-r symbolic update algorithm above, with $r = m$. This matrix has identical symbolic factors as $\mathbf{A}\mathbf{A}^{\mathsf{T}}$. Case A will apply for each column in $\mathbf{A}$, corresponding to the

$$\bigcup_{\min\ \mathcal{A}_k=j} \mathcal{A}_k$$

term in (3.1). Since $\pi(c) = 0$ for each c, cases B and D will not apply. At column j, case C will apply for all children in the elimination tree, corresponding to the

$$\bigcup_{\{c:j=\pi(c)\}} \mathcal{L}_c \setminus \{c\}$$

term in (3.1). Since duplicate paths are discarded when they merge, we modify each column j once, for each child c in the elimination tree. This is the same work performed by the symbolic factorization algorithm, Algorithm 2, which is $O(|\mathcal{L}|)$. Hence, Algorithm 3 is equivalent to Algorithm 2 when we apply it to the update $\mathbf{I}+\mathbf{A}\mathbf{A}^{\mathsf{T}}$. Its run time is optimal in this case.

5. Multiple-rank symbolic downdate. The downdate algorithm is analogous. The downdated matrix is $\mathbf{A}\mathbf{A}^{\mathsf{T}} - \mathbf{W}\mathbf{W}^{\mathsf{T}}$, where $\mathbf{W}$ is a subset of the columns of $\mathbf{A}$. In a downdate, $\overline{\mathcal{P}}(k) \subseteq \mathcal{P}(k)$, and thus rather than following the paths $\overline{\mathcal{P}}(k_i)$, we follow the paths $\mathcal{P}(k_i)$. Entries are dropped during a downdate, and thus $\overline{\mathcal{L}}_j \subseteq \mathcal{L}_j$ and $\pi(j) \le \overline{\pi}(j)$. We start with $\overline{\mathcal{L}}_j^{\sharp} = \mathcal{L}_j^{\sharp}$ and make the following changes.

- Case A: If $j = \min \mathcal{W}_i$ for some i, then the pattern $\mathcal{W}_i$ is removed from column j,

$$\overline{\mathcal{L}}_j^{\sharp} = \overline{\mathcal{L}}_j^{\sharp} - \mathcal{W}_i.$$

- Case B: If $j = \pi(c) = \overline{\pi}(c)$ for some node c, then c is a child of j in both the old and new tree. We need to remove from $\overline{\mathcal{L}}_j^{\sharp}$ entries in the old pattern $\mathcal{L}_c$ but not in the new pattern $\overline{\mathcal{L}}_c$,

$$\overline{\mathcal{L}}_j^{\sharp} = \overline{\mathcal{L}}_j^{\sharp} - (\mathcal{L}_c \setminus \overline{\mathcal{L}}_c).$$

- Case C: If $j = \pi(c) \ne \overline{\pi}(c)$ for some node c, then c is a child of j in the old elimination tree, but not the new tree. We compute

$$\overline{\mathcal{L}}_j^{\sharp} = \overline{\mathcal{L}}_j^{\sharp} - (\mathcal{L}_c \setminus \{c\}).$$

- Case D: If $j = \overline{\pi}(c) \ne \pi(c)$ for some node c, then c is a child of j in the new tree, but not the old one. We compute

$$\overline{\mathcal{L}}_j^{\sharp} = \overline{\mathcal{L}}_j^{\sharp} + (\overline{\mathcal{L}}_c \setminus \{c\}).$$

Case C will occur for c and j prior to visiting column $\overline{\pi}(c)$, since $j = \pi(c) < \overline{\pi}(c)$. Thus we place c in the new-child-queue of $\overline{\pi}(c)$ when encountering case C for nodes c and j. When the algorithm visits node $\overline{\pi}(c)$, its new-child-queue will contain all those nodes for which case D holds.

ALGORITHM 4 (symbolic rank-r downdate; remove matrix $\mathbf{W}$).

Find the starting nodes of each path
for $i = 1$ **to** r **do**
  $\mathcal{W}_i = \{k : w_{ki} \neq 0\}$
  $k_i = \min\ \mathcal{W}_i$
  place path-marker i in path-queue of column k_i
end for
Consider all columns corresponding to nodes in the paths $\mathcal{P}(k_1)$ through $\mathcal{P}(k_r)$
for $j = \min_{i\in[1,r]} k_i$ **to** m **do**
  if path-queue of column j is nonempty **do**
    $\overline{\mathcal{L}}_j^\sharp = \mathcal{L}_j^\sharp$
    for each path-marker i on path-queue of column j **do**
      Path $\mathcal{P}(k_i)$ includes column j
      Let c be the prior column on this path (if any), where $\pi(c) = j$
      if $j = k_i$ **do**
        Case A: j is the first node on the path $\mathcal{P}(k_i)$, no prior c
        $\overline{\mathcal{L}}_j^\sharp = \overline{\mathcal{L}}_j^\sharp - \mathcal{W}_i$
      else if $j = \overline{\pi}(c)$ **then**
        Case B: c is an old child of j, possibly changed
        $\overline{\mathcal{L}}_j^\sharp = \overline{\mathcal{L}}_j^\sharp - (\mathcal{L}_c \setminus \overline{\mathcal{L}}_c)$
      else
        Case C: c is a lost child of j and a new child of $\overline{\pi}(c)$
        $\overline{\mathcal{L}}_j^\sharp = \overline{\mathcal{L}}_j^\sharp - (\mathcal{L}_c \setminus \{c\})$
        place c in new-child-queue of column $\overline{\pi}(c)$
      end if
    end for
    Case D: consider each new child of j
    for each c in new-child-queue of j **do**
      $\overline{\mathcal{L}}_j^\sharp = \overline{\mathcal{L}}_j^\sharp + (\overline{\mathcal{L}}_c \setminus \{c\})$
    end for
    Move up one step in the path(s)
    $\overline{\pi}(j) = \min\ \overline{\mathcal{L}}_j \setminus \{j\}$
    if $\mathcal{L}_j \setminus \{j\} \neq \emptyset$ **then**
      Let i be the largest path-marker in path-queue of column j
      Place path-marker i in path-queue of column $\pi(j)$
    end if
  end if *path-queue of column j nonempty*
end for

The time taken by Algorithm 4 is

$$O\left(m + \sum_{j\in\mathcal{T}} |\mathcal{L}_j|\right),$$

which is slightly higher than the optimal time,

$$O\left(\sum_{j\in\mathcal{T}} |\mathcal{L}_j|\right).$$

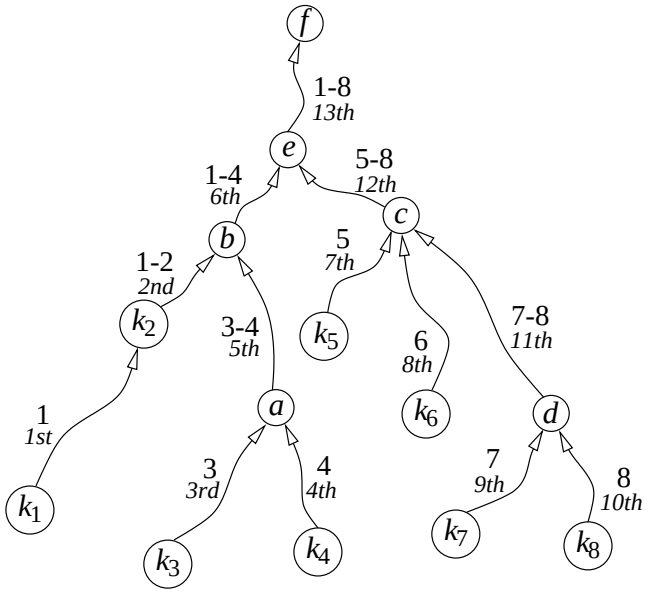

FIG. 6.1. *Example rank-8 update after depth-first-search reordering.*

In most practical cases, the $O(m)$ term in the asymptotic run time for Algorithm 4 will not be the dominant term.

6. Multiple-rank numerical update and downdate. The following numerical rank-r update/downdate algorithm, Algorithm 5, overwrites $\mathbf{L}$ and $\mathbf{D}$ with the updated or downdated factors. The algorithm is based on Algorithm 1, the one-pass version of Method C1 in [18] presented in section 1. The algorithm is used after the symbolic update algorithm (Algorithm 3) has found the subtree $\overline{\mathcal{T}}$ corresponding to the nodes whose patterns can change, or after the symbolic downdate algorithm (Algorithm 4) has found $\mathcal{T}$. Since the columns of the matrix $\mathbf{W}$ can be reordered without affecting the product $\mathbf{W}\mathbf{W}^{\mathsf{T}}$, we reorder the columns of $\mathbf{W}$ using a depth-first search [6] of $\overline{\mathcal{T}}$ (or $\mathcal{T}$) so that as we march through the tree, consecutive columns of $\mathbf{W}$ are utilized in the computations. This reordering improves the numerical update/downdate algorithm by placing all columns of $\mathbf{W}$ that affect any given subpath next to each other, eliminating an indexing operation. Reordering the columns of a sparse matrix prior to Cholesky factorization is very common [3, 22, 23, 25]. It improves data locality and simplifies the algorithm, just as it does for reordering $\mathbf{W}$ in a multiple-rank update/downdate. The depth-first ordering of the tree changes as the elimination tree changes, so columns of $\mathbf{W}$ must be ordered for each update or downdate.

To illustrate this reordering, consider the subtree $\overline{\mathcal{T}}$ in Figure 4.2 for a rank-8 update. If the depth-first-search algorithm visits child subtrees from left to right, the resulting reordering is as shown in Figure 6.1. Each subpath in Figure 6.1 is labeled with the range of columns of $\mathbf{W}$ that affect that subpath, and with the order in which the subpath is processed by Algorithm 5. Consider the path from node c to e. In Figure 4.2, the columns of $\mathbf{L}$ corresponding to nodes on this subpath are updated by columns 2, 8, 3, and 5 of $\mathbf{W}$, in that order. In the reordered subtree (Figure 6.1), the columns on this subpath are updated by columns 5 through 8 of the reordered $\mathbf{W}$.

ALGORITHM 5 (sparse numeric rank-r modification; add $\sigma \mathbf{W}\mathbf{W}^{\mathsf{T}}$).
The columns of $\mathbf{W}$ *have been reordered.*
for $i = 1$ **to** r **do**
  $\alpha_i = 1$
end for
for each subpath in depth-first-search order in $\overline{\mathcal{T}}$ ($\sigma = 1$) or $\mathcal{T}$ ($\sigma = -1$) **do**
  Let c_1 through c_2 be the columns of $\mathbf{W}$ that affect this subpath
  for each column j in the subpath **do**
    for $i = c_1$ **to** c_2 **do**
      $\overline{\alpha} = \alpha_i + \sigma w_{ji}^2 / d_j$
      $d_j = d_j \overline{\alpha}$
      $\gamma_i = w_{ji} / d_j$
      $d_j = d_j / \alpha_i$
      $\alpha_i = \overline{\alpha}$
    end for
    for all $p \in \overline{\mathcal{L}}_j \setminus \{j\}$ ($\sigma = 1$) or $p \in \mathcal{L}_j \setminus \{j\}$ ($\sigma = -1$) **do**
      for $i = c_1$ **to** c_2 **do**
        $w_{pi} = w_{pi} - w_{ji} l_{pj}$
        $l_{pj} = l_{pj} + \sigma \gamma_i w_{pi}$
      end for
    end for
  end for
end for

The time taken by r rank-1 updates [10] is

$$O\left(\sum_{i=1}^{r} \sum_{j \in \mathcal{P}^{(i)}(k_i)} |\mathcal{L}_j^{(i)}| \right), \tag{6.1}$$

where $\mathcal{L}_j^{(i)}$ is the pattern of column j after the ith rank-1 update. This time is asymptotically optimal. A single rank-r update cannot determine the paths $\mathcal{P}^{(i)}(k_i)$, but uses $\overline{\mathcal{P}}(k_i)$ instead. Thus, the time taken by Algorithm 5 for a rank-r update is

$$O\left(\sum_{i=1}^{r} \sum_{j \in \overline{\mathcal{P}}(k_i)} |\overline{\mathcal{L}}_j| \right).$$

This is slightly higher than (6.1), because $\mathcal{P}^{(i)}(k_i) \subseteq \overline{\mathcal{P}}(k_i)$ and $\mathcal{L}_j^{(i)} \subseteq \overline{\mathcal{L}}_j$. Since $\mathcal{P}^{(i)}(k_i) \subseteq \overline{\mathcal{P}}(k_i)$, the ith column of $\mathbf{W}$ does not necessarily affect all of the columns in the path $\overline{\mathcal{P}}(k_i)$. If $\mathbf{w}_i$ does not affect column j, then w_{ji} and γ_i will both be zero in the inner loop in Algorithm 5. An example of this occurs in Figure 4.1, where column 1 of $\mathbf{W}$ does not affect column 7 of $\mathbf{L}$. We could check this condition, and reduce the asymptotic run time to

$$O\left(\sum_{i=1}^{r} \sum_{j \in \mathcal{P}^{(i)}(k_i)} |\overline{\mathcal{L}}_j| \right).$$

In practice, however, we found that the paths $\mathcal{P}^{(i)}(k_i)$ and $\overline{\mathcal{P}}(k_i)$ did not differ much. Including this test did not improve the overall performance of our algorithm. The

time taken by Algorithm 5 for a rank-r downdate is similar, namely,

$$O\left(\sum_{i=1}^{r}\sum_{j\in\mathcal{P}(k_i)}|\mathcal{L}_j|\right).$$

The numerical algorithm for updating and downdating $\mathbf{LL}^\mathsf{T}$ is essentially the same as that for $\mathbf{LDL}^\mathsf{T}$ [4, 24]; the only difference is a diagonal scaling. For either $\mathbf{LL}^\mathsf{T}$ or $\mathbf{LDL}^\mathsf{T}$, the symbolic algorithms are identical.

7. Experimental results. To test our methods, we selected the same experiment as in our earlier paper on the single-rank update and downdate [10], which mimics the behavior of the LPDASA [20]. The first matrix is $10^{-6}\mathbf{I}+\mathbf{A}_0\mathbf{A}_0{}^\mathsf{T}$, where $\mathbf{A}_0$ consists of 5446 columns from a larger 6071-by-12,230 matrix $\mathbf{B}$ with 35,632 nonzeros arising in an airline scheduling problem (DFL001) [13]. The 5446 columns correspond to the optimal solution of the linear programming problem. Starting with an initial $\mathbf{LDL}^\mathsf{T}$ factorization of the matrix $10^{-6}\mathbf{I}+\mathbf{A}_0\mathbf{A}_0{}^\mathsf{T}$, we added columns from $\mathbf{B}$ (corresponding to an update) until we obtained the factors of $10^{-6}\mathbf{I}+\mathbf{BB}^\mathsf{T}$. We then removed columns in a first-in-first-out order (corresponding to a downdate) until we obtained the original factors. The LPDASA algorithm would not perform this much work (6784 updates and 6784 downdates) to solve this linear programming problem.

Our experiment took place on a Sun Ultra Enterprise running the Solaris 2.6 operating system, with eight 248 MHz UltraSparc-II processors (only one processor was used) and 2 GB of main memory. The dense matrix performance in millions of floating-point operations per second (Mflops) of the BLAS [12] is shown in Table 7.1. All results presented below are for our own codes (except for `colmmd`, `spooles`, and the BLAS) written in the C programming language and using double precision floating-point arithmetic.

TABLE 7.1
Dense matrix performance for 64*-by-*64 *matrices and* 64*-by-*1 *vectors.*

BLAS operation	Mflops
DGEMM (matrix-matrix multiply)	171.6
DGEMV (matrix-vector multiply)	130.0
DTRSV (solve $\mathbf{Lx}=\mathbf{b}$)	81.5
DAXPY (the vector computation $\mathbf{y}=\alpha\mathbf{x}+\mathbf{y}$)	78.5
DDOT (the dot product $\alpha=\mathbf{x}^\mathsf{T}\mathbf{y}$)	68.7

We first permuted the rows of $\mathbf{B}$ to preserve sparsity in the Cholesky factors of $\mathbf{BB}^\mathsf{T}$. This can be done efficiently with `colamd` [7, 8, 9, 21], which is based on an approximate minimum degree ordering algorithm [1]. However, to keep our results consistent with our prior rank-1 update/downdate paper [10], we used the same permutation as in those experiments (from `colmmd` [17]). Both `colamd` and MATLAB's `colmmd` compute the ordering without forming $\mathbf{BB}^\mathsf{T}$ explicitly. A symbolic factorization of $\mathbf{BB}^\mathsf{T}$ finds the nonzero counts of each column of the factors. This step takes an amount of space that is proportional to the number of nonzero entries in $\mathbf{B}$. It gives us the size of a static data structure to hold the factors during the updating and downdating process. The numerical factorization of $\mathbf{BB}^\mathsf{T}$ is not required. A second symbolic factorization finds the first nonzero pattern $\mathcal{L}$. An initial numerical factorization computes the first factors $\mathbf{L}$ and $\mathbf{D}$. We used our own nonsupernodal factorization code (similar to SPARSPAK [5, 15]), since the update/downdate algorithms do not use supernodes. A supernodal factorization code such as `spooles` [3] or

TABLE 7.2
Average update and downdate performance results.

rank r	rank-r time / r in seconds		Mflops	
	Update	Downdate	Update	Downdate
1	0.0840	0.0880	30.3	29.6
2	0.0656	0.0668	38.9	39.0
3	0.0589	0.0597	43.3	43.6
4	0.0513	0.0549	49.7	47.5
5	0.0500	0.0519	51.0	50.2
6	0.0469	0.0487	54.4	53.5
7	0.0451	0.0468	56.6	55.7
8	0.0434	0.0448	58.8	58.2
9	0.0431	0.0458	59.1	57.0
10	0.0426	0.0447	60.0	58.3
11	0.0415	0.0437	61.5	59.6
12	0.0413	0.0432	61.8	60.3
13	0.0403	0.0424	63.2	61.4
14	0.0402	0.0420	63.6	62.1
15	0.0395	0.0413	64.6	63.1
16	0.0392	0.0408	65.1	63.9

TABLE 7.3
Dense matrix performance for 64*-by-*64 *matrices and* 64*-by-*1 *vectors.*

Operation	Time (sec)	Mflops	Notes
`colamd` ordering	0.45	-	
Symbolic factorization (of $\mathbf{BB}^\mathsf{T}$)	0.07	-	1.49 million nonzeros
Symbolic factorization for first $\mathcal{L}$	0.46	-	831 thousand nonzeros
Numeric factorization for first $\mathbf{L}$ (our code)	20.07	24.0	
Numeric factorization for first $\mathbf{L}$ (`spooles`)	18.10	26.6	
Numeric factorization of $\mathbf{BB}^\mathsf{T}$ (our code)	61.04	18.5	not required
Numeric factorization of $\mathbf{BB}^\mathsf{T}$ (`spooles`)	17.80	63.3	not required
Average rank-16 update	0.63	65.1	compare with rank-1
Average rank-5 update	0.25	51.0	compare with solve step
Average rank-1 update	0.084	30.3	
Average solve $\mathbf{LDL}^\mathsf{T}\mathbf{x} = \mathbf{b}$	0.27	18.2	

a multifrontal method [2, 14] can get better performance. The factorization method used has no impact on the performance of the update and downdate algorithms.

We ran 16 different experiments, each one using a different rank-r update and downdate, where r varied from 1 to 16. After each rank-r update, we solved the sparse linear system $\mathbf{LDL}^\mathsf{T}\mathbf{x} = \mathbf{b}$ using a dense right-hand side $\mathbf{b}$. To compare the performance of a rank-1 update with a rank-r update ($r > 1$), we divided the run time of the rank-r update by r. This gives us a normalized time for a single rank-1 update. The average time and Mflops rate for a normalized rank-1 update and downdate for the entire experiment is shown in Table 7.2. The time for the update, downdate, or solve increases as the factors become denser, but the performance in terms of Mflops is fairly constant for all three operations. The first rank-16 update when the factor $\mathbf{L}$ is sparsest takes 0.47 seconds (0.0294 seconds normalized) and runs at 65.5 Mflops compared to 65.1 Mflops in Table 7.2 for the average speed of all the rank-16 updates.

The performance of each step is summarized in Table 7.3. A rank-5 update takes about the same time as using the updated factors to solve the sparse linear system $\mathbf{LDL}^\mathsf{T}\mathbf{x} = \mathbf{b}$, even though the rank-5 update performs 2.6 times the work.

The work, in terms of floating-point operations, varies only slightly as r changes.

With rank-1 updates, the total work for all the updates is 17.293 billion floating-point operations, or 2.55 million per rank-1 update. With rank-16 updates (the worst case), the total work increases to 17.318 billion floating-point operations. The rank-1 downdates take a total of 17.679 billion floating-point operations (2.61 million per rank-1 downdate), while the rank-16 downdates take a total of 17.691 billion operations. This confirms the near-optimal operation count of the multiple-rank update/downdate, as compared to the optimal rank-1 update/downdate.

Solving $\mathbf{Lx} = \mathbf{b}$ when $\mathbf{L}$ is sparse and $\mathbf{b}$ is dense, and computing the sparse $\mathbf{LDL}^\mathsf{T}$ factorization using a nonsupernodal method, both give a rather poor computation-to-memory-reference ratio of only 2/3. We tried the same loop unrolling technique used in our update/downdate code for our sparse solve and sparse $\mathbf{LDL}^\mathsf{T}$ factorization codes, but this resulted in no improvement in performance.

A sparse rank-r update or downdate can be implemented in a one-pass algorithm that has much better memory traffic than that of a series of r rank-1 modifications. In our numerical experimentation with the DFL001 linear programming test problem, the rank-r modification was more than twice as fast as r rank-1 modifications for $r \geq 11$. The superior performance of the multiple-rank algorithm can be explained using the computation-to-memory-reference ratio. If $c_1 = c_2$ in Algorithm 5 (a subpath affected by only one column of $\mathbf{W}$), it can be shown that this ratio is about 4/5 when $\overline{\mathcal{L}}_j$ is large. The ratio when $c_2 = c_1 + 15$ (a subpath affected by 16 columns of $\mathbf{W}$) is about 64/35 when $\overline{\mathcal{L}}_j$ is large. Hence, going from a rank-1 to a rank-16 update improves the computation-to-memory-reference ratio by a factor of about 2.3 when column j of $\mathbf{L}$ has many nonzeros. By comparison, the level-1 BLAS routines for dense matrix computations (vector computations such as DAXPY and DDOT) [11] have computation-to-memory-reference ratios between 2/3 and 1. The level-2 BLAS (DGEMV and DTRSV, for example) have a ratio of 2.

8. Summary. Because of improved memory locality, our multiple-rank sparse update/downdate method is over twice as fast as our prior rank-1 update/downdate method. The performance of our new method (65.1 Mflops for a sparse rank-16 update) compares favorably with both the dense matrix performance (81.5 Mflops to solve the dense system $\mathbf{Lx} = \mathbf{b}$) and the sparse matrix performance (18.0 Mflops to solve the sparse system $\mathbf{Lx} = \mathbf{b}$ and an observed peak numerical factorization of 63.3 Mflops in `spooles`) on the computer used in our experiments. Although not strictly optimal, the multiple-rank update/downdate method has nearly the same operation count as the rank-1 update/downdate method, which has an optimal operation count.

REFERENCES

[1] P. R. Amestoy, T. A. Davis, and I. S. Duff, *An approximate minimum degree ordering algorithm*, SIAM J. Matrix Anal. Appl., 17 (1996), pp. 886–905.

[2] P. R. Amestoy and I. S. Duff, *Vectorization of a multiprocessor multifrontal code*, Internat. J. Supercomputer Appl., 3 (1989), pp. 41–59.

[3] C. Ashcraft and R. G. Grimes, *SPOOLES: An object-oriented sparse matrix library*, in Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing, San Antonio, TX, 1999, CD-ROM, SIAM, Philadelphia, 1999.

[4] C. H. Bischof, C.-T. Pan, and P. T. P. Tang, *A Cholesky up- and downdating algorithm for systolic and SIMD architectures*, SIAM J. Sci. Comput., 14 (1993), pp. 670–676.

[5] E. Chu, A. George, J. W. H. Liu, and E. Ng, *SPARSPAK: Waterloo Sparse Matrix Package, User's Guide for SPARSPAK-A*, Technical report, Department of Computer Science, University of Waterloo, Waterloo, ON, Canada, 1984.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, and McGraw–Hill, New York, 1990.

[7] T. A. Davis, J. R. Gilbert, S. I. Larimore, E. Ng, and B. Peyton, *A column approximate minimum degree ordering algorithm*, in Proceedings of the Sixth SIAM Conference on Applied Linear Algebra, Snowbird, UT, 1997, p. 29.
[8] T. A. Davis, J. R. Gilbert, E. Ng, and B. Peyton, *A column approximate minimum degree ordering algorithm*, in Abstracts of the Second SIAM Conference on Sparse Matrices, Snowbird, UT, 1996.
[9] T. A. Davis, J. R. Gilbert, E. Ng, and B. Peyton, *A column approximate minimum degree ordering algorithm*, presented at the 13th Householder Symposium on Numerical Linear Algebra, Pontresina, Switzerland, 1996.
[10] T. A. Davis and W. W. Hager, *Modifying a sparse Cholesky factorization*, SIAM J. Matrix Anal. Appl., 20 (1999), pp. 606–627.
[11] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979.
[12] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, *A set of level-3 basic linear algebra subprograms*, ACM Trans. Math. Software, 16 (1990), pp. 1–17.
[13] J. J. Dongarra and E. Grosse, *Distribution of mathematical software via electronic mail*, Comm. ACM, 30 (1987), pp. 403–407.
[14] I. S. Duff and J. K. Reid, *The multifrontal solution of indefinite sparse symmetric linear equations*, ACM Trans. Math. Software, 9 (1983), pp. 302–325.
[15] A. George and J. W. H. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice–Hall, Englewood Cliffs, NJ, 1981.
[16] A. George, J. Liu, and E. Ng, *A data structure for sparse QR and LU factorizations*, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 100–121.
[17] J. R. Gilbert, C. Moler, and R. Schreiber, *Sparse matrices in MATLAB: Design and implementation*, SIAM J. Matrix Anal. Appl., 13 (1992), pp. 333–356.
[18] P. E. Gill, G. H. Golub, W. Murray, and M. A. Saunders, *Methods for modifying matrix factorizations*, Math. Comp., 28 (1974), pp. 505–535.
[19] W. W. Hager, *Updating the inverse of a matrix*, SIAM Rev., 31 (1989), pp. 221–239.
[20] W. W. Hager, *The LP dual active set algorithm*, in High Performance Algorithms and Software in Nonlinear Optimization, R. D. Leone, A. Murli, P. M. Pardalos, and G. Toraldo, eds., Kluwer, Dordrecht, The Netherlands, 1998, pp. 243–254.
[21] S. I. Larimore, *An Approximate Minimum Degree Column Ordering Algorithm*, Technical Report TR-98-016, University of Florida, Gainesville, FL, 1998; also available online at http://www.cise.ufl.edu/tech-reports/.
[22] J. W. H. Liu, *The role of elimination trees in sparse factorization*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 134–172.
[23] E. G. Ng and B. W. Peyton, *A supernodal Cholesky factorization algorithm for shared-memory multiprocessors*, SIAM J. Sci. Comput., 14 (1993), pp. 761–769.
[24] C.-T. Pan, *A modification to the LINPACK downdating algorithm*, BIT, 30 (1990), pp. 707–722.
[25] E. Rothberg, A. Gupta, E. G. Ng, and B. W. Peyton, *Parallel sparse Cholesky factorization algorithms for shared-memory multiprocessor systems*, in Advances in Computer Methods for Partial Differential Equations - VII, R. Vichnevetsky, D. Knight, and G. Richter, eds., IMACS, 1992, pp. 622–628.
[26] G. Strang, *Linear Algebra and Its Applications*, Academic Press, New York, 1980.