FAST COMPUTATION OF MINIMAL FILL INSIDE A GIVEN ELIMINATION ORDERING

PINAR HEGGERNES* AND BARRY W. PEYTON†

Abstract. Minimal elimination orderings were introduced by Rose, Tarjan, and Lueker in 1976, and during the last decade they have received increasing attention. Such orderings have important applications in several different fields, and they were first studied in connection with minimizing fill in sparse matrix computations. Rather than computing any minimal ordering, which might result in fill that is far from minimum, it is more desirable for practical applications to start from an ordering produced by a fill-reducing heuristic, and then compute a minimal fill that is a subset of the fill produced by the given heuristic. This problem has been addressed previously, and there are several algorithms for solving it. The drawback of these algorithms is that either there is no theoretical bound given on their running time although they might run fast in practice, or they have a good theoretical running time but they have never been implemented or they require a large machinery of complicated data structures to achieve the good theoretical time bound. In this paper, we present an algorithm called MCS-ETree for solving the mentioned problem in $O(nm\,\mathrm{A}(m,n))$ time, where m and n are respectively the number of edges and vertices of the graph corresponding to the input sparse matrix, and $\mathrm{A}(m,n)$ is the very slowly growing inverse of Ackerman's function. A primary strength of MCS-ETree is its simplicity and its straightforward implementation details. We present run time test results to show that our algorithm is fast in practice. Thus our algorithm is the first that both has a provably good running time with easy implementation details, and is fast in practice.

Key words. sparse matrix computations, minimal fill, elimination trees, composite tree rotations, maximum cardinality search (MCS), minimal triangulation

AMS subject classifications. 65F05, 65F50, 05C85, 05C90

1. Introduction. Consider the Cholesky factorization $A = LL^T$ of an $n \times n$ symmetric positive definite sparse matrix A. Elements $l_{ij} \neq 0$, where $a_{ij} = 0$, are called *fill* elements. It is well known that finding a good permutation matrix P and computing the Cholesky factor of PAP^T rather than the Cholesky factor of A can give much less fill, and is an essential operation in sparse matrix computations. The matrix A is conveniently interpreted as a graph G, where G has a vertex v_i for each row (or equivalently column) i of A, and $\{v_i, v_j\}$ is an edge in G if and only if $a_{ij} \neq 0$. Similarly, the *filled graph* G^+ is the graph of $L + L^T$, and fill elements of L correspond to *fill edges* of G^+. Any permutation matrix P for A corresponds to an elimination ordering α on G such that G_α is the graph of PAP^T, and the number of fill edges in G^+_α is entirely dependent on α. Thus we refer to the fill edges of G^+_α as the fill *produced by* α. (Definitions and notation are detailed in Section 2.)

For sparse matrix computations [26] and in many other fields [7, 16, 27], one would like to find orderings that produce the minimum possible fill. This problem was shown to be NP-hard by Yannakakis in 1981 [28]. Already in 1976, Rose, Tarjan, and Lueker [25] conjectured the NP-hardness of this problem. They also introduced the notion of *minimal elimination orderings* and *minimal fill*, and they presented an algorithm for computing both in $O(nm)$ time in the same paper. An ordering α is a minimal elimination ordering if there is no ordering β such that G^+_β is a strict subgraph of G^+_α. For any ordering α, the filled graph G^+_α is a chordal graph [10], and is called a *triangulation* of G. If α is a minimal elimination ordering then G^+_α is a minimal

*Department of Informatics, University of Bergen, N-5020 Bergen, Norway. Email: pinar@ii.uib.no

†Dalton State College, 650 College Drive, Dalton GA 30720, USA. Email: bpeyton@daltonstate.edu

triangulation. One reason that minimal elimination orderings are highly desirable in sparse matrix computations is that they ensure that subsequent equivalent reorderings do not change the space allocation requirements [5]. In the field of graph algorithms, minimal elimination orderings and minimal triangulations are very important and well studied [13], as they include the set of triangulations that correspond to widely studied graph parameters, like *minimum fill* and *treewidth*, and thus provide a tool to compute these by approximation algorithms [21] or exact (fast) exponential time algorithms [9].

A minimal triangulation can contain fill that is far from minimum fill. Consequently, for practical applications it is more appropriate to start with a good triangulation produced by a common heuristic algorithm, like Minimum Degree [1, 17] or Nested Dissection [11], and then compute a minimal triangulation that is a subgraph of the initial triangulation [6]. This problem, sometimes called the *minimal triangulation sandwich problem*, was first posed and solved by Blair, Heggernes, and Telle in 1996 [5], and they presented an algorithm with running time $O(mf + f^2)$, where f is the number of fill edges in the initial triangulation. For small f this algorithm is fast in practice, however its running time is heavily dependent on f, which might be $O(n^2)$, giving an $O(n^4)$ time algorithm in the worst case. Later, Dahlhaus solved the same problem with an algorithm of running time $O(nm)$ [8], but this algorithm has never been implemented to our knowledge. A more recent algorithm by Berry et al. solves the same problem in $O(nm)$ time [3]; however, a heavy machinery of complicated data structures is necessary to achieve this time bound. In addition to these, two algorithms based on iterations were given without running time analysis separately by Peyton [24], and by Berry, Heggernes, and Simonet [4]. The algorithm of Peyton is documented to run fast in practice[1], whereas the latter algorithm is of less practical and more theoretical interest [22].

In this paper, we present an algorithm called MCS-ETree that takes as input a graph G and an initial ordering β, and produces as output a minimal elimination ordering α such that G^+_α is a subgraph of G^+_β (i.e., G^+_α is *sandwiched* between G and G^+_β). The running time of our algorithm is $O(nm\ \mathrm{A}(m,n))$, where $\mathrm{A}(m,n)$ is the very slowly growing inverse of Ackerman's function. Hence our theoretical running time is very close to the best known theoretical running time $O(nm)$ for solving this problem. Compared to $O(nm)$ algorithms solving the same problem, MCS-ETree has the advantage of being both fast in practice and easy to implement, not relying on complicated data structures; it uses basic operations and data structures commonly used in practice in sparse matrix computations, with modest adaptations for use by the algorithm. In addition, in practical tests our algorithm is usually faster than the previous algorithm with fastest running time.

This paper is organized as follows. Section 2 introduces most of the background, terminology, and notation. Section 3 gives some background on composite elimination tree rotations [19], which are used by our new algorithm in a slightly modified form. Section 4 presents the new algorithm, MCS-ETree, which computes minimal orderings and solves the above mentioned sandwich problem. This section also proves that the algorithm is correct. Section 5 discusses some of the implementation issues, and shows that the running time is $O(nm\ \mathrm{A}(m,n))$. Also, Section 5 both presents a straightforward implementation and discusses how to enhance the implementation in ways that dramatically improve the performance in our tests. Section 6 reports the

[1] In fact the algorithm of Peyton [24] is the fastest in practice of all mentioned algorithms.

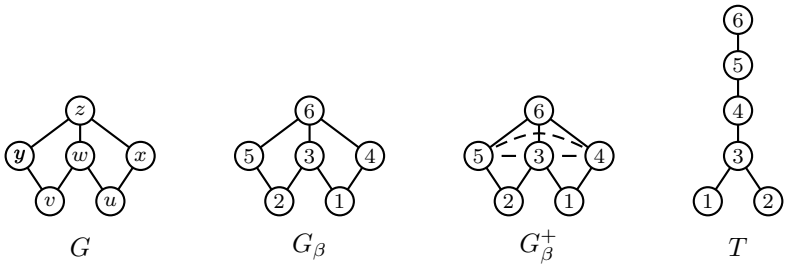

FIG. 2.1. *For the given graph G and its ordering β, the Elimination Game results in the filled graph G^+_β. The three fill edges are dashed lines. Also pictured is the elimination tree produced by the ordering.*

results of these tests. Finally, Section 7 gives some concluding remarks.

2. Background and notation. A graph $G = (V, E)$ consists of a set V of n vertices and a set E of m edges. For a given graph G, we denote the set of its vertices by $V(G)$ and the set of its edges by $E(G)$. When $\{u, v\}$ is an edge we say that u and v are *adjacent* or *neighbors*. For a vertex v of G, $\text{adj}_G(v)$ denotes the set of vertices adjacent to v, also called the *adjacency* or *neighborhood* of v, and $\text{adj}_G[v] = \text{adj}_G(v) \cup \{v\}$. For a set of vertices $X \subseteq V$, $\text{adj}_G(X) = \bigcup_{x \in X} \text{adj}_G(x) \setminus X$ and $\text{adj}_G[X] = \text{adj}_G(X) \cup X$. An *ordering* α of G is a bijective function $\alpha : V \to \{1, 2, ..., n\}$, and we will sometimes write $\alpha = (v_1, v_2, ..., v_n)$, meaning that $\alpha(v_i) = i$ for $1 \leq i \leq n$ (we will call i the *number* of v_i). When an ordering α is given, the ordered graph is denoted by G_α. In this case, $\text{hadj}_{G_\alpha}(v_i) = \text{adj}_{G_\alpha}(v_i) \cap \{v_{i+1}, v_{i+2}, ..., v_n\}$ is the set of higher numbered neighbors of v_i, and $\text{ladj}_{G_\alpha}(v_i) = \text{adj}_{G_\alpha}(v_i) \cap \{v_1, v_2, ..., v_{i-1}\}$ is the set of lower numbered neighbors of v_i.

If two graphs G and H have the same vertex set, then G is a *subgraph* of H if $E(G) \subseteq E(H)$, and G is a *proper subgraph* of H if $E(G) \subset E(H)$. A subgraph of G *induced* by a vertex set $X \subseteq V$ will be denoted by $G(X)$. An induced subgraph $G(X)$ contains every edge of G with both endpoints in X. A set $X \subseteq V$ of vertices is a *clique* if every pair of vertices in X is adjacent in G. A *maximal clique* is any clique whose vertex set is maximal with respect to subset inclusion for this property. A *path* is a sequence of distinct vertices $x_1 - x_2 - ... - x_k$ such that x_i is adjacent to x_{i+1} for $1 \leq i < k$. A *chord* on a path is an edge between two non-consecutive vertices of the path. A *cycle* is a path where the first vertex is the same as the last vertex. A graph is *chordal* if it contains no chordless cycle on 4 or more vertices.

The following simple algorithm is called the *Elimination Game* [23], and it simulates (on graphs) Cholesky factorization of matrices. With input graph G and ordering α, repeatedly pick the smallest numbered vertex, add edges to make its set of neighbors a clique, and remove this vertex and the edges incident upon it from the graph, until the graph is empty. The set of edges that are added during the algorithm is called *fill*, and the *filled graph* G^+_α is obtained by adding to G this fill. Figure 2.1 shows a graph G, an ordering β of the graph (G_β), and the filled graph G^+_β associated with the ordering. This graph and initial ordering will be used throughout the paper to illustrate algorithms and other points, as needed.

The following lemma, which we will use in our proofs, characterizes the edges in a filled graph G^+_α.

LEMMA 2.1. [25] *Given a graph G and an ordering $\alpha = (v_1, v_2, ..., v_n)$, an edge $\{v_i, v_j\}$ with $i < j$ is present in G^+_α if and only if $\{v_i, v_j\}$ is an edge of G, or there is a path between v_i and v_j in G containing only vertices from the set $\{v_1, v_2, ..., v_{i-1}\}$.*

A path between v_i and v_j containing only vertices that all have smaller numbers in α than the smaller of i and j will be called a *fill path*.

If G^+_α contains no fill edges then α is called a *perfect elimination ordering (peo)*. Fulkerson and Gross showed that chordal graphs are exactly the class of graphs that have perfect elimination orderings [10]. Thus for every graph G and ordering α, the filled graph G^+_α is chordal, and G^+_α is a triangulation of G. In a chordal graph, the vertices of any maximal clique can be ordered last by some peo in any arbitrary internal order [27].

Any ordering β of G that is a peo of G^+_α is an *equivalent reordering* of G with respect to α. An equivalent reordering introduces no new fill; that is, G^+_β is a subgraph of G^+_α. If there exists no ordering β for which G^+_β is a proper subgraph of G^+_α, then α is called a *minimal elimination ordering (meo)*, and G^+_α is a *minimal triangulation*. Computing an meo is equivalent to computing a minimal triangulation, as every peo of a minimal triangulation gives the same filled graph when applied to the original graph [5, 25]. The following is a characterization of minimal triangulations that we will use in the proof that our new algorithm is correct.

THEOREM 2.2. [25] *A given triangulation H of a graph G is a minimal triangulation if and only if every fill edge added to G to obtain H is the unique chord of a 4-cycle in H.*

Given a graph G and an ordering α, the filled graph G^+_α defines a structure called an *elimination tree* T as follows: vertex v_j is the parent of vertex v_i in T if v_j is the smallest numbered vertex in $\mathrm{hadj}_{G^+_\alpha}(v_i)$. The elimination tree associated with the filled graph in Figure 2.1 is included in Figure 2.1. Due to Lemma 2.1, the elimination tree corresponding to α can be computed directly from G and α without computing G^+_α explicitly [20]. A *topological ordering* of T is any ordering that numbers each child with a number smaller than that of its parent. The ordering in Figure 2.1 is a topological ordering of T. Any topological ordering of T is an equivalent ordering of G with respect to α. Consequently, we will talk about equivalent orderings with respect to an ordering and with respect to an elimination tree, interchangeably. Liu [20] provides a thorough examination of elimination trees. Note also that if G has more than one connected component, then one obtains an elimination forest with one tree for each connected component.

In a rooted tree, an *ancestor* of a vertex v is any vertex that is on the unique path between v and the root, including the root; and a *descendant* of v is any vertex of which v is an ancestor. Let $T[v]$ be the subtree of an elimination tree T that is rooted at v and consists of v and every descendant of v in T; such subtrees will be called *elimination subtrees*. It is well known that $\mathrm{hadj}_{G^+_\alpha}(v) = \mathrm{adj}_G(V(T[v]))$ [20], and we will make use of this fact throughout the paper. As an illustration, note that in Figure 2.1 we have

$$\begin{aligned}\mathrm{adj}_G(V(T[v_3])) &= \mathrm{adj}_G(\{v_1, v_2, v_3\}) \\ &= \{v_4, v_5, v_6\} \\ &= \mathrm{adj}_{G^+_\beta}(v_3).\end{aligned}$$

Finally, we let $\mathrm{anc}_T(v)$ be the set of ancestors of v in T, where v is *not* included in the set; we also write $\mathrm{anc}_T[v] = \mathrm{anc}_T(v) \cup \{v\}$.

Algorithm Change_Root(G, T, u)
Input: A graph $G = (V, E)$, an elimination tree T of G, and a vertex $u \in V$.
Output: A reordering γ of G that is equivalent with respect to T,
where u is numbered last.
Number u last in γ and mark u as already numbered;
$z \leftarrow u$;
while z is not the root of T **do**
Order the unnumbered vertices of $\mathrm{adj}_G(V(T[z]))$ last in γ, but
before those that are already numbered by γ;
Mark the newly-numbered vertices as already numbered;
$z \leftarrow$ the parent of z in T;
end while;
Number in γ the vertices in $V \setminus \mathrm{anc}_T[u]$ using their original
relative order in T;
end Change_Root;

FIG. 3.1. *An algorithm for changing the root of an elimination tree with an equivalent reordering (See Algorithm 3.2 Composite_Rotations in Liu [19]).*

3. Changing the root of an elimination subtree. In our new algorithm MCS-ETree, we will need to reorder an elimination subtree $T[v]$ in such a way that a particular vertex $u \in V(T[v])$ is numbered last by this reordering, and the corresponding reordering of $G(V(T[v]))$ is equivalent to any given topological ordering of $T[v]$. Since u is numbered last among the vertices in $V(T[v])$ by the reordering, it will be the root of the new elimination subtree associated with the new equivalent reordering. A trivial modification of the composite elimination tree rotations algorithm in Liu [19] will perform this task.

For a given graph G and a given ordering β, let T be the elimination tree associated with the filled graph G^+_β, and let $u \in V(G)$. Algorithm 3.2 (Composite_Rotations) from [19] reorders G with a peo γ of G^+_β such that the vertices of $\mathrm{adj}_G(V(T[u]))$ are numbered last in γ. (Recall that γ is an equivalent reordering of G with respect to β.) Notice that there might be ancestors of u in T that do not belong to $\mathrm{adj}_G(V(T[u]))$, and hence u will often become closer to the root of the resulting new elimination tree corresponding to γ, and will never be further away than it is in T. The algorithm Change_Root in Figure 3.1 adds a single first line to Liu's Composite_Rotations algorithm in order to number u last, and also modifies the last line to number the vertices in $V \setminus \mathrm{anc}_T[u]$ so that u is not also numbered there. These are the only modifications to the original algorithm. Consequently, the rest of the vertices are numbered in the same order as in Composite_Rotations; that is, the vertices of $\mathrm{adj}_G(V(T[u]))$ are ordered next-to-last, and so on. The elimination tree obtained from the ordering produced by Change_Root clearly is rooted at vertex u.

In Figure 3.2, we illustrate a run of Change_Root on the graph and elimination tree in Figure 2.1. The elimination tree in Figure 3.2 is the same as that in Figure 2.1 with the numbers replaced by the appropriate letters. The vertex u has been chosen to become the new root. The algorithm first numbers u last with the number 6. The main loop then processes the ancestors of u in ascending order. When u is processed, the unnumbered neighbors of $V(T[u])$, namely, w and x, are numbered next-to-last in front of u with numbers 4 and 5, respectively. When w is processed next, the unnumbered neighbors of $V(T[w])$, namely, z and y, are numbered last in front of

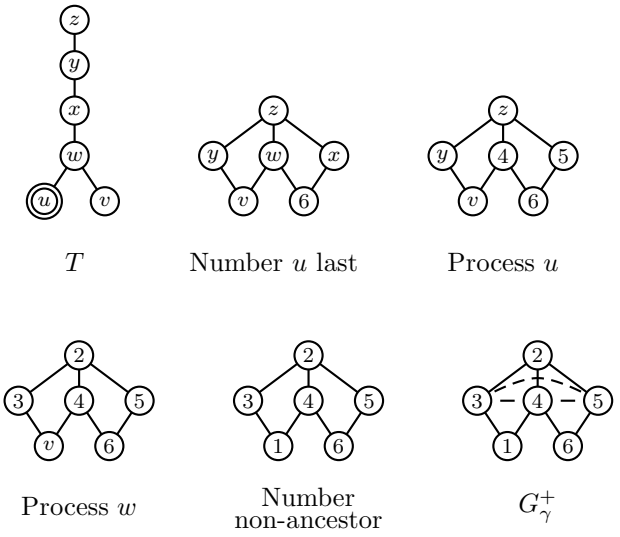

FIG. 3.2. *The algorithm Change_Root is run on the graph and elimination tree shown in Figure 2.1. The new root is to be vertex u.*

the previously numbered vertices with numbers 2 and 3, respectively. When y and z are processed, no vertices receive their numbers. Finally, when the main loop is finished, the single non-ancestor of u, namely v, receives the number 1. Note that G^+_γ in Figure 3.2 is identical to G^+_β in Figure 2.1, so that γ is an equivalent reordering with respect to β, as required. Note also that u will be the root of the new elimination tree, as desired.

The following lemma shows that Change_Root produces an ordering equivalent to the input ordering.

LEMMA 3.1. *Let β be an ordering of a graph G, and let T be the elimination tree associated with G^+_β. Choose $u \in V(G)$. Any ordering γ produced by Change_Root(G, T, u) is equivalent with respect to β.*

Proof. The desired property is inherited directly from Composite_Rotations, as we will see. Consider the ordering γ' produced by Composite_Rotations that corresponds as closely as possible to the ordering γ produced by Change_Root. It is known from Liu [19] that γ' is equivalent with respect to β. The ordering γ' matches the ordering γ except for the placement of u. The ordering γ numbers u at the end of the ordering; the ordering γ' numbers u before the vertices of $\mathrm{adj}_G(V(T[u]))$ and after the vertices of $V(T[u]) \setminus \{u\}$. The lowest numbered vertex of $\mathrm{adj}_G(V(T[u]))$ will be the parent of u in the elimination tree T' associated with γ'. Note then that the set $\{u\} \cup \mathrm{adj}_G(V(T[u]))$ is a clique in $G^+_{\gamma'}$ and forms a chain in T' from u to the root of the tree. A new topological ordering of T' can be obtained by ordering the vertices of $\{u\} \cup \mathrm{adj}_G(V(T[u]))$ last (consistent with γ') and then ordering the rest earlier, but consistent with γ'. The key observation is that when we change this ordering so that u is moved to the end (i.e., u is ordered last, but the vertices of $\mathrm{adj}_G(V(T[u]))$ remain ordered consistent with γ'), we obtain a topological ordering of the elimination tree associated with the ordering γ. Since we have merely changed the order of the the vertices in the highest-numbered clique, clearly γ is equivalent with respect to γ'. Since γ' is equivalent with respect to β, we have γ is equivalent with respect to β, as

desired. □

We will use Change_Root to process elimination subtrees. Choose a vertex $v \in V(T)$ and consider the elimination subtree $T[v]$. Observe that $T[v]$ is the elimination tree one obtains by applying a topological ordering of $T[v]$ to the induced subgraph $H = G(V(T[v]))$. This follows because any topological ordering of an elimination tree will produce the same elimination tree. Let $u \in V(T[v])$. When we execute Change_Root(H, $T[v]$, u), we obtain an equivalent reordering of H with respect to $T[v]$, where u is numbered last, and hence will become the root of the new elimination subtree. When this new subtree is glued to the old elimination tree with the old parent of v now becoming the new parent of u, a revised elimination tree for the entire graph is obtained.

4. Algorithm MCS-ETree and its proof of correctness. In this section, we present a new algorithm MCS-ETree that solves the minimal triangulation sandwich problem. Given an original ordering β and graph G, this algorithm generates a minimal ordering α such that G^+_α is a subgraph of G^+_β. The algorithm generates α by numbering the vertices from n down to 1, and the key feature is the selection at each step of a vertex of "maximum cardinality" to receive the next number. Hence the algorithm resembles a minimal triangulation algorithm called MCS-M [2], and its proof of correctness uses the same technique used there. In Section 5, we will show that it can be implemented to run in $O(nm\,\mathrm{A}(m,n))$ time, and can be implemented so that it does not compute any filled graphs explicitly.

Our new algorithm is given in Figure 4.1. First the algorithm computes the elimination tree T^* obtained when β is used as an elimination order on G. The set of elimination subtrees remaining to be processed (i.e., numbered) is $Trees$. Initially $Trees$ contains the single member $T^* = T^*[x]$, where x is the root of T^*. We have assumed that G is a connected graph so that we have an elimination tree rather than an elimination forest. If we had an elimination forest, then we would place each tree of the forest in $Trees$.

4.1. Executing MCS-ETree on an example. Figure 4.2 walks step-by-step through an execution of MCS-ETree on the example introduced in Figure 2.1. In Figure 4.2, the filled graph G^+_β is the same as that pictured in Figure 2.1. The initialization step computes the elimination tree T^* of G with respect to β. This is shown next in the figure. The elimination subtree $T^*[x]$ is placed in $Trees$, the set of elimination subtrees yet to receive their α-numbers. The set of vertices that have received their α-numbers, namely L, is initially empty. The set L is listed above the heavy horizontal line above T^*. The counter k used to assign the next α-number is initially 6.

The algorithm then enters the main **while** loop. There is only one unnumbered elimination subtree to process, which is referred to as $T = T[v]$ by the algorithm. Now, $T = T^*$ at this stage in the example. Since $L = \emptyset$, the cardinalities referred to next in the algorithm are all zero. In the figure, these zero cardinalities are written beside each vertex. A vertex of maximum cardinality, with no descendant of maximum cardinality, will have to be a leaf in this case. The algorithm chooses vertex 1 as the maximum cardinality vertex.

The algorithm next uses Change_Root to reorder the subtree and change the root to vertex 1. Precisely this operation was illustrated in Figure 3.2. The ordering shown in the first graph of Figure 4.2 labeled G^+_γ is precisely the same as the final ordering shown in Figure 3.2.

Algorithm MCS-ETree
Input: A graph G and an ordering β.
Output: An meo α of G such that G_α^+ is a subgraph of G_β^+.

```
/* Initializations */
Compute the elimination tree T* of G with respect to β;
x ← root of T*;
Trees ← {T*[x]};   L ← ∅;   k ← n;

while Trees ≠ ∅ do

    /* Get an unnumbered elimination subtree T */
    Pick an arbitrary elimination subtree T = T[v] from Trees;
    Trees ← Trees \ {T};

    /* Find a special vertex u in T of "maximum cardinality" */
    Find a vertex u ∈ V(T) for which |adj_G(V(T[u])) ∩ L| = |adj_G(V(T))|,
    and |adj_G(V(T[w])) ∩ L| < |adj_G(V(T[u])) ∩ L| for each descendant w of
    u in T;

    /* Re-order the subtree and change the root to u */
    H ← G(V(T));
    Compute a topological order γ1 of T;
    Use Change_Root(H, T, u) to compute a peo γ2 of H^+_γ1 that numbers u last;

    /* Compute the elimination subtree for the new reordering */
    Compute the elimination tree T' = T'[u] of H with respect to γ2;

    /* Number u and store the unnumbered subtrees for future processing */
    for each child c of u in T' do
        Trees ← Trees ∪ {T'[c]};
    end for;
    α(u) ← k;   L ← L ∪ {u};   k ← k − 1;

end while;

end MCS-ETree;
```

FIG. 4.1. *Algorithm MCS-ETree, which finds a minimal ordering and solves the minimal triangulation sandwich problem.*

The algorithm next computes the elimination subtree for the new reordering. This new elimination tree T' rooted at the vertex now numbered 6 by γ appears next in the figure. At the bottom of the **while** loop, the root vertex receives its α-number 6. It is removed and added to the set of α-numbered vertices L. The unnumbered elimination subtree rooted at the sole child of vertex 6 (i.e., vertex 5) is also added to $Trees$ to be processed later.

At the top of the next iteration of the **while** loop, the unnumbered subtree $T'[c]$, where c is the vertex numbered 5 by the current ordering γ, is chosen to be processed next. Note that in the previous filled graph only vertices 5 and 4 are neighbors of the

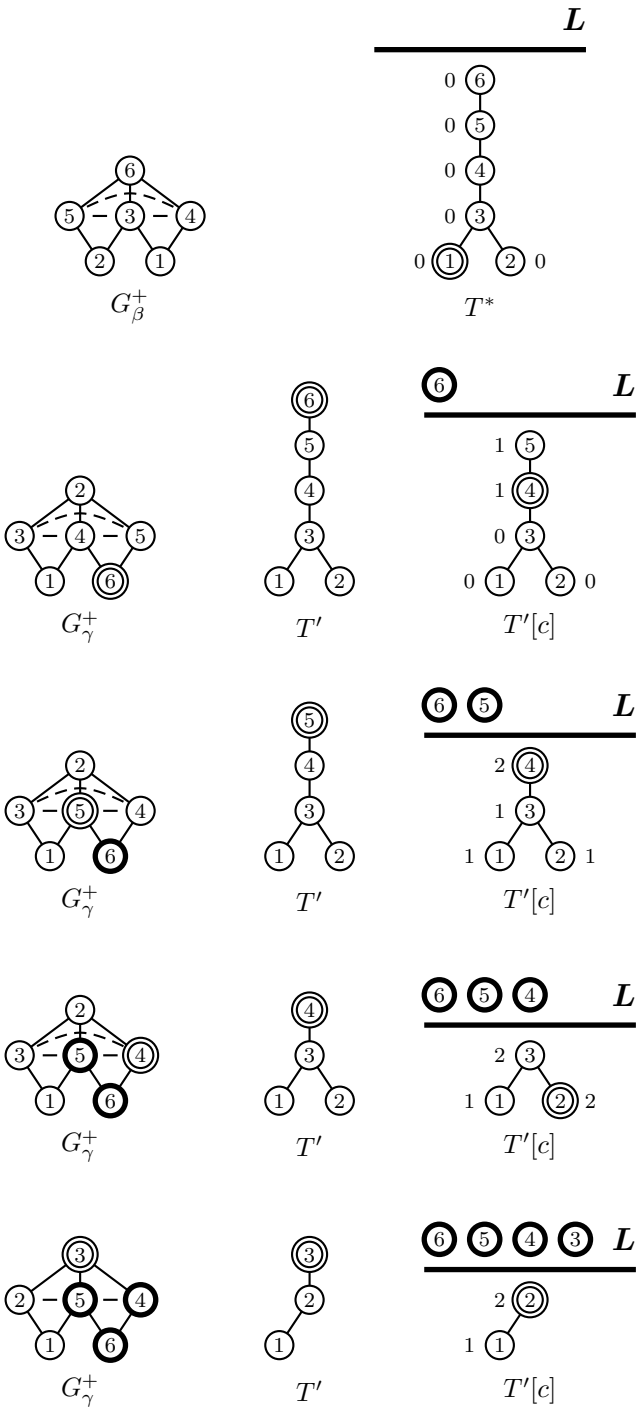

FIG. 4.2. *An execution of Algorithm MCS-ETree on the example introduced in Figure 2.1.*

vertex that received α-number 6. The cardinalities of vertices 4 and 5 then are both 1, while the cardinalities of the rest of the vertices in the subtree are 0, as indicated in the figure. The vertex with γ-number 4 is the maximum cardinality vertex with no descendants of maximum cardinality.

We leave it for the reader to walk through the remaining steps of MCS-ETree on our example shown in Figure 4.2. Observe that in the final filled graph G^+_γ, the fill edge that once joined current vertices 2 and 4 has disappeared. In the unfilled graph, vertices 1, 5, 3, and 2 form an unchorded 4-cycle, and so do vertices 6, 4, 3, and 5 (using the current γ-numbers). No single fill edge will suffice to chord both cycles, so any minimum fill set will have two fill edges — one to chord each cycle. It follows that the last filled graph pictured has minimum fill, and hence minimal fill. We leave it for the reader to verify that the final ordering α produced by the algorithm is the same as that shown in the last graph G^+_γ shown at the bottom of the figure. Hence the final ordering α is a minimal ordering.

4.2. Proving the algorithm correct. In this subsection, we will show that the algorithm is correct. The following simple loop invariant is needed to verify the integrity of the elimination subtrees.

LEMMA 4.1. *The following is a loop invariant of Algorithm MCS-ETree:*
$adj_G(V(T)) \subseteq L$ *for each elimination subtree* $T \in Trees$.

Proof. The statement is clearly true before the first iteration of the **while** loop. Suppose that it is true at the beginning of an iteration. Then for the elimination subtree $T = T[v]$ chosen to be processed and removed from $Trees$, we have $\mathrm{adj}_G(V(T)) \subseteq L$, where L is the set of vertices already numbered by the eventual meo α. The next step in the iteration chooses a vertex $u \in V(T)$ of maximum cardinality. As in the MCS-M ordering [2], the cardinality is determined by the number of neighbors in the filled graph that have received their number in α. Subsequently, the iteration reorders the connected component $H = G(V(T))$ so that u is numbered last. Next, the iteration computes the new elimination subtree $T' = T'[u]$ obtained when the computed reordering is applied to H. The vertex u is the root of this new elimination subtree. Then u is ordered next by MCS-ETree and added to L. From basic properties of elimination trees [20] and the fact that the loop invariant holds at the beginning of the iteration, we have the following for each elimination subtree $T'[c]$ added to $Trees$:

$$\begin{aligned}\mathrm{adj}_G(V(T'[c])) &\subseteq \{u\} \cup \mathrm{adj}_G(V(T'[u])) \\ &= \{u\} \cup \mathrm{adj}_G(V(T')) \\ &= \{u\} \cup \mathrm{adj}_G(V(T)) \\ &\subseteq L.\end{aligned}$$

The result then follows. □

At the beginning or end of any iteration, a *current ordering* γ is implicitly associated with the algorithm, as follows. For each vertex $x \in L$, we let $\gamma(x) = \alpha(x)$. The vertices in $V \setminus L$ are numbered from 1 to $n - |L|$ so that each child in an elimination subtree in $Trees$ receives a smaller number than that assigned to its parent. Then at the beginning or end of any iteration the *current filled graph* implicitly associated with the algorithm is G^+_γ.

Lemma 4.1 says that there are no edges in G joining two vertices from different elimination subtrees in $Trees$ at any point during the algorithm. It follows that the

elimination subtrees generated throughout the algorithm will maintain their integrity, with no pair of elimination subtrees merged into a single elimination subtree by a current ordering γ associated with the algorithm. In other words, every elimination tree in $Trees$ is an elimination subtree of the elimination tree with respect to γ.

The key step within each iteration of the algorithm selects the vertex to receive the highest α-number among the vertices in the elimination subtree $T = T[v]$. For any vertex $x \in V(T)$ the set of vertices in L adjacent to x in the current filled graph G^+_γ is $\mathrm{hadj}_{G^+_\gamma}(x) \cap L$. What MCS-ETree requires is a maximum cardinality vertex in T with no descendants that are maximum cardinality vertices. That is, choose a vertex $u \in V(T)$ for which $|\mathrm{hadj}_{G^+_\gamma}(u) \cap L| = |\mathrm{hadj}_{G^+_\gamma}(v)|$ and for which $|\mathrm{hadj}_{G^+_\gamma}(w) \cap L| < |\mathrm{hadj}_{G^+_\gamma}(u) \cap L|$ for every descendant w of u. Since $\mathrm{hadj}_{G^+_\gamma}(x) \cap L = \mathrm{adj}_G(V(T[x])) \cap L$ for every vertex $x \in V(T)$, the algorithm equivalently chooses a vertex $u \in V(T)$ for which $|\mathrm{adj}_G(V(T[u])) \cap L| = |\mathrm{adj}_G(V(T))|$ and for which $|\mathrm{adj}_G(V(T[w])) \cap L| < |\mathrm{adj}_G(V(T[u])) \cap L|$ for every descendant w of u. Notice that such a vertex u always exists, since $\mathrm{adj}_G(V(T[v])) \cap L = \mathrm{adj}_G(V(T))$, and hence v can be chosen as u if no descendant x of v satisfies $|\mathrm{adj}_G(V(T[x])) \cap L| = |\mathrm{adj}_G(V(T))|$. Furthermore, if every descendant x of v satisfies $|\mathrm{adj}_G(V(T[x])) \cap L| = |\mathrm{adj}_G(V(T))|$, then u is chosen to be a leaf of T.

A second important step within each iteration uses algorithm Change_Root to compute a new peo of the filled graph of induced subgraph $H = G(V(T))$ under a topological ordering of T. This is instrumental in causing G^+_α to be a subgraph of G^+_β.

LEMMA 4.2. *If γ is a current ordering for the algorithm at the beginning of an iteration and γ' is a current ordering for the algorithm at the end of the same iteration, then $E(G^+_{\gamma'}) \subseteq E(G^+_\gamma)$.*

Proof. Let $\{x, y\}$ be a fill edge in $E(G^+_{\gamma'})$ at the end of the iteration. Let L be the set of numbered vertices at the beginning of the iteration. (Vertex u is added to L at the end of the iteration.) If both x and y belong to L, then by Lemma 2.1 we have $\{x, y\} \in E(G^+_\gamma)$. If either x or y is a vertex in one of the unnumbered elimination subtrees other than $T = T[v]$, then $\{x, y\} \in E(G^+_\gamma)$ because the subtree is ordered topologically by both γ and γ'. If both x and y are vertices in $V(T)$, then $\{x, y\} \in E(G^+_\gamma)$ because $H = G(V(T))$ is renumbered by algorithm Change_Root with a peo of the filled graph $H^+_{\gamma_1}$, where γ_1 is a topological ordering of T. Finally, the only case remaining is where $x \in V(T)$ and $y \in L$. If $x \notin \mathrm{anc}_T[u]$, then from Algorithm Change_Root we have $T'[x] = T[x]$, so by Lemma 2.1 we have $\{x, y\} \in E(G^+_\gamma)$. If $x \in \mathrm{anc}_T[u]$, then by the choice of u in the algorithm and simple properties of elimination trees [20],

$$\begin{aligned}
\mathrm{adj}_{G^+_{\gamma'}}(x) \cap L &= \mathrm{adj}_G(V(T'[x])) \cap L \\
&\subseteq \mathrm{adj}_G(V(T')) \\
&= \mathrm{adj}_G(V(T)) \\
&= \mathrm{adj}_G(V(T[u])) \cap L \\
&\subseteq \mathrm{adj}_G(V(T[x])) \cap L \\
&= \mathrm{adj}_{G^+_\gamma}(x) \cap L.
\end{aligned}$$

This completes the proof. ∎

Observe that Lemma 4.2 holds for the sequence of filled graphs associated with the sequence of current orderings in Figure 4.2. Observe also that in the final filled graph in Figure 4.2, fill edge $\{4,5\}$ is the sole chord of a 4-cycle joining vertices 3, 5, 6, and 4, and fill edge $\{2,5\}$ is the sole chord of a 4-cycle joining vertices 1, 5, 3, and 2. In our proof of correctness, we will show that every fill edge in the final filled graph G_α^+ generated by MCS-ETree is the sole chord of a 4-cycle, and then use Theorem 2.2 to argue the result.

THEOREM 4.3. *Given a graph G and an ordering β of G, Algorithm MCS-ETree generates an meo α of G such that G_α^+ is a subgraph of G_β^+.*

Proof. From Lemma 4.2 and the definition of current orderings γ within the algorithm, it follows that G_α^+ is a subgraph of G_β^+. If there are no fill edges in G_α^+, then G is chordal and α is a peo of G. It follows that α is an meo of G, and hence we have the result in this case. Assume therefore that G_α^+ has at least one fill edge. Let $\{u,w\}$ be a fill edge in G_α^+. We will find a 4-cycle in G_α^+ for which $\{u,w\}$ is the only chord. The minimality of the ordering α will then follow by Theorem 2.2.

Without loss of generality, assume that $\alpha(u) < \alpha(w)$. Let $T = T[v]$ be the elimination subtree that the algorithm is processing when u is chosen by the algorithm to receive its α-number. Let γ be a current ordering for the algorithm at the beginning of this iteration. By Lemma 4.2, since $\{u,w\}$ is a fill edge in G_α^+, $\{u,w\}$ is also a fill edge in the current filled graph G_γ^+. By Lemma 2.1, there is a fill path $u - x_1 - \ldots - x_r - w$ ($r \geq 1$) in G through vertices x_i that are descendants of u in T. Notice that u therefore cannot be a leaf of T. The vertices x_i come from one (and only one) of the subtrees rooted at a child of u in T, say $T[c]$. Let x_t be the vertex among the x_i that is eventually numbered highest by the algorithm. Then there are fill paths (or direct edges) in G from x_t to u and from x_t to w under the final ordering α generated by the algorithm. So $\{x_t,u\}$ and $\{x_t,w\}$ are edges in the final filled graph G_α^+.

By Lemma 2.1, we know that $\mathrm{hadj}_{G_\gamma^+}(c) \cap L \subseteq \mathrm{hadj}_{G_\gamma^+}(u) \cap L$ for each child c of u in T. From the choice of u, we can conclude that $\mathrm{hadj}_{G_\gamma^+}(c) \cap L \subset \mathrm{hadj}_{G_\gamma^+}(u) \cap L$ (proper subset). Let $y \in (\mathrm{hadj}_{G_\gamma^+}(u) \cap L) \setminus (\mathrm{hadj}_{G_\gamma^+}(c) \cap L)$, which is not empty. By Lemma 2.1, none of the vertices x_i is adjacent to y in G_γ^+ (including x_t). By Lemma 4.2, $\{x_t,y\}$ is not an edge in the final filled graph G_α^+.

Finally, the set $\mathrm{hadj}_{G_\gamma^+}(u) \cap L = \mathrm{hadj}_{G_\gamma^+}(v)$ is the higher adjacency set of u in the final filled graph G_α^+, since u receives its number at this step, and all vertices that are numbered higher than u in α have already received their numbers. So $\mathrm{hadj}_{G_\gamma^+}(u) \cap L = \mathrm{hadj}_{G_\alpha^+}(u)$, and recall that $\mathrm{hadj}_{G_\alpha^+}(u) \cup \{u\}$ is a clique in G_α^+. Note that both y and w belong to $\mathrm{hadj}_{G_\alpha^+}(u)$. It follows that $\{u,y\}$ and $\{w,y\}$ are both edges in G_α^+. This completes a 4-cycle $x_t - u - y - w - x_t$ in G_α^+ for which $\{u,w\}$ is the only chord.

Since every fill edge is the only chord of such a 4-cycle, the final filled graph G_α^+ is a minimal chordal supergraph by Theorem 2.2, and the final ordering, which is a peo of G_α^+, is an meo of G. □

5. Implementation details and running time analysis. We can adapt basic tools from sparse matrix computations to obtain a time bound of $O(nm\,\mathrm{A}(m,n))$ for Algorithm MCS-ETree.

THEOREM 5.1. *The running time of Algorithm MCS-ETree is $O(nm\,A(m,n))$.*

Proof. Let us consider the three major tasks the algorithm must perform as it goes through a single iteration of the **while** loop. Let γ be a current ordering at the beginning of the iteration, and let $T = T[v]$ be the elimination subtree chosen to be processed.

First, the algorithm needs the values $|\mathrm{hadj}_{G_\gamma^+}(x) \cap L| = |\mathrm{adj}_G(V(T[x])) \cap L|$ for every vertex $x \in V(T)$. One option is to compute and work directly with the filled graph G_γ^+, but this leads to $O(nm')$ total work for this task (summed over all iterations of the **while** loop), where m' is the number of edges in the initial filled graph G_β^+. It also requires storage of filled graphs rather than just the original graph G. We have not implemented this option. Gilbert, Ng, and Peyton [12] introduced a fast algorithm for computing the number of nonzeros in each row and each column of a sparse Cholesky factor. Hence a second, and better, option is to modify the algorithm in [12] for column nonzero counts to compute the values $|\mathrm{hadj}_{G_\gamma^+}(x) \cap L|$ for every vertex $x \in V(T)$. This option leads to $O(nm\ \mathrm{A}(m,n))$ total work for this task, summed over all iterations of the **while** loop. It does not involve or require the computation of any filled graphs explicitly.

The algorithm in [12] is geared to compute $|\mathrm{ladj}_{G_\gamma^+}(x) \cup \{x\}|$ (the "row count") and $|\mathrm{hadj}_{G_\gamma^+}(x) \cup \{x\}|$ (the "column count") for every vertex $x \in V(G)$. In adapting for use by MCS-ETree, the computation is restricted in three different ways. First, none of the computation connected with row counts is carried out. Second, the computation can be restricted to the elimination subtree $T = T[v]$ processed by the current iteration of the algorithm rather than the entire elimination tree associated with a current ordering γ. And third, the counts must be restricted to compute $|\mathrm{hadj}_{G_\gamma^+}(x) \cap L|$ rather than $|\mathrm{hadj}_{G_\gamma^+}(x) \cup \{x\}|$. It is straightforward to adapt the implementation in Gilbert, Ng, and Peyton [12, page 1085] to incorporate these restrictions. Note also that a postordering of the elimination subtree T is required by our adaptation of the algorithm. This requirement is inherited from the original algorithm.

Second, MCS-ETree uses algorithm Change_Root to reorder the vertices of T so that u becomes the new root and there is no additional fill under the new current ordering. To implement Change_Root, we initially reorder the vertices of T by a postordering that numbers each vertex in $\mathrm{anc}_T[u]$ before any of its siblings. The ordering and marking process can then be performed as the vertices are visited in this postorder. The total work spent on this task over all iterations of the outer loop is $O(nm)$.

Third, the algorithm needs to recompute the elimination subtree for the new ordering of $H = G(V(T))$. The elimination subtree can be computed with a single sweep of the full adjacency lists of the vertices of T and the required disjoint set union operations. It is trivial to adapt the standard algorithm [20] for computing the entire elimination tree to compute the elimination subtrees needed here. The total work for this task over all iterations of the outer loop is $O(nm\ \mathrm{A}(m,n))$. This concludes the proof. □

5.1. Basic implementation. We have implemented these three steps in the most straightforward way possible, with no attempt at avoiding redundant work. The object with the first implementation was to make it as simple as possible. We have called this first implementation the *basic* implementation.

With the basic implementation established, we sought to enhance the implementation by avoiding redundant work. There is much redundant work to be avoided in all three of the major steps within each iteration. Getting rid of this redundant work does not reduce the overall provable time bound of the algorithm, but it results in a much faster implementation in practice, as the test results will show in the next section.

5.2. Enhanced implementation. Consider again the computation of the values $|\text{hadj}_{G_\gamma^+}(x) \cap L|$ for every vertex $x \in V(T[v])$. A key part of the algorithm in [12] is the recognition of and reduction to the so-called *skeleton adjacency sets* [18] associated with the current ordering. If these sets are known and stored ahead of the computation, then they can be traversed rather than full adjacency sets. Let $z \in L$ and let $T[v]$ be the current elimination subtree. Let $T_r[z, v]$ denote the *row subtree* of z in $T[v]$. That is, $V(T_r[z, v])$ is the set of vertices of $T[v]$ that are adjacent to z in the current filled graph G_γ^+. We say that z is in the *skeleton adjacency set* of $x \in V(T[v])$ if x is a leaf of $T_r[z, v]$. Note that our skeleton adjacency set of $x \in V(T[v])$ is limited to vertices in L. To ultimately improve efficiency, we store the skeleton adjacency sets of all vertices $x \in V(T[v])$ as the values $|\text{hadj}_{G_\gamma^+}(x) \cap L|$ are computed. The skeleton adjacency sets come as a natural by-product of the computation.

In Figure 4.2, consider the point where the vertex of maximum cardinality is chosen from the unnumbered elimination subtree $T'[c]$, whose root has current γ-number 4. The skeleton adjacency sets of vertices 1, 2, 3, and 4 are $\{5\}$, $\{5\}$, $\emptyset$, and $\{6\}$, respectively. These skeleton adjacency sets suffice to compute the needed cardinalities.

Again, let $T = T[v]$. For each vertex $x \in \text{anc}_T[u]$, it is possible that $T'[x] \neq T[x]$ because of the reordering obtained from algorithm Change_Root (if $u \neq v$). So the vertices in the skeleton adjacency set of x cannot safely be used during the next step that processes the subtree containing x. The entire adjacency set of x must be used during the next step that processes the subtree containing x. But in the case where $x \in V(T) \setminus \text{anc}_T[u]$, we have $T'[x] = T[x]$ because of the reordering obtained from algorithm Change_Root. The descendants of x remain precisely the same, so the skeleton adjacency set of x does not change, except for the possible addition of the new root u. We take care of the update with u, and process the old abbreviated skeleton adjacency set during the next step that processes the subtree containing x.

So in summary, we process abbreviated skeleton adjacency sets, many of which are in practice empty, for vertices that at the most recent relevant step were in $V(T) \setminus \text{anc}_T[u]$; we process full adjacency sets for vertices that at the most recent relevant step were in $\text{anc}_T[u]$. To store the skeleton adjacency sets requires another vector large enough to store the full adjacency structure of G. But this technique promises to improve run times appreciably.

Consider again how to implement algorithm Change_Root for computing a reordering of an elimination tree T so that $u \in V(T)$ becomes the new root and no new fill is introduced. As before, we reorder the vertices of T with a postordering for which every vertex in $\text{anc}_T[u]$ is numbered before any of its siblings. The procedure Change_Root2 in Figure 5.1 can then be used to perform the reordering. The prescribed postordering is input as γ_1, which is of course a topological ordering of T. Unlike our earlier implementation of algorithm Change_Root, the only adjacency sets that algorithm Change_Root2 traverses are those for vertices in $\text{anc}_T[u]$. This also promises to improve run times appreciably.

Consider the recomputation of the elimination subtree, replacing $T = T[v]$ with $T' = T'[u]$. Again, the subtrees rooted at vertices in $V(T) \setminus \text{anc}_T[u]$ remain unchanged as MCS-ETree goes forward to the next iteration. So there is no need to recompute these portions of the elimination subtree. The new subtree can be patched together with an enhanced implementation that traverses the adjacency sets of the vertices of $\text{anc}_T[u]$ only.

These enhancements do not change the time complexity of the algorithm; it re-

```
procedure Change_Root2(G, T, γ1, u, γ2)
Input: A graph G, an elimination tree T of G with respect to γ1,
       a prescribed postordering γ1 of T, and a vertex u ∈ V(T).
Output: An equivalent reordering γ2 of G with respect to γ1,
        where u is numbered last.
  for i ∈ [0, 1, ..., |V(T)|]; B(i) ← ∅; end for;
  j ← 0;
  for x ∈ V(T) in the prescribed postorder γ1
    if x ∈ V(T) \ anc_T[u] then
      j ← j + 1; γ2(x) ← j;
    end if;
  end for;
  B(0) ← B(0) ∪ {u};
  for x ∈ anc_T(u)
    j ← |V(T)|;
    for y ∈ adj_G(x)
      j ← min(j, γ1(y));
    end for;
    B(j) ← B(j) ∪ {x};
  end for;
  j ← |V(T)|;
  for i ∈ [0, 1, ..., |V(T)|] in order
    for x ∈ B(i)
      γ2(x) ← j;
      j ← j − 1;
    end for;
  end for;
end Change_Root2;
```

FIG. 5.1. *An enhanced variant of algorithm Change_Root for the second step in the main loop of MCS-ETree.*

mains $O(nm\,\mathrm{A}(m,n))$. We call the improved implementation of the algorithm the *enhanced* implementation. Because components of the work of lower time complexity have greater relative influence on performance after these enhancements are incorporated, there are other improvements implemented in marking processes and initializations. These are not described here.

5.3. Blocked implementation. Finally there is one further enhancement of a completely different sort to incorporate into the code. For this last version we first include all the enhancements described thus far, then we add the following. When $T = T[v]$ is processed and vertex u is to be numbered, we can often detect other vertices among the vertices of $\mathrm{anc}_T(u)$ that can be numbered in a block along with u and removed with no further processing. There are two cases to consider. First, consider the case where u has descendants in T. Let $c_1, \ldots, c_r$ be the children of u in T. If a vertex $x \in \mathrm{anc}_T(u)$ is adjacent to each subtree $T[c_1], \ldots, T[c_r]$ and the adjacency set of x contains every vertex that is in the skeleton adjacency set of u (again, limited to skeleton neighbors in L), then x can be ordered in a block along with u (see Lemma 5.2). Having possession of the skeleton adjacency sets is crucial here for implementing detection of this condition. These are available only after our

enhancement for the computation of cardinalities.

In Figure 4.2, consider the unnumbered elimination subtree $T'[c]$ whose root has γ-number 5. There, the maximum cardinality vertex chosen is vertex 4. The sole member of the skeleton adjacency set of vertex 4 is vertex 6. Since vertex 5 is adjacent in G to vertex 6 and also adjacent in G to the subtree rooted at vertex 3, it follows that vertex 5 can be ordered in a block along with vertex 4.

Second, consider the case where u has no descendants in T. If a vertex $x \in \text{anc}_T(u)$ is adjacent to u and every vertex in $\text{adj}_G(u)$ (except x of course), then x can be ordered in a block along with u (see Lemma 5.3).

LEMMA 5.2. *Let u be a vertex of maximum cardinality chosen at some iteration of Algorithm MCS-ETree such that u has descendants in $T = T[v]$. Let X be the set comprised of u and any vertex $x \in anc_T(u)$ adjacent to all the subtrees rooted at children of u and adjacent to all the members of u's skeleton adjacency set. Our algorithm can be modified so that it numbers next as a block the vertices in X in the current iteration.*

Proof. Let the algorithm be modified so that it numbers the vertices of X next as a block in the current iteration. Let L be the set of numbered vertices before the vertices of X are numbered. Note first that by the choice of u, the definition of X, and Lemma 2.1, every vertex of X will be adjacent to every vertex of $\text{adj}_G(V(T))$ in the final filled graph. Choose $x \in X$, and let $\{x, w\}$ be a fill edge where w is numbered higher than x by the ordering. (Note that x may be u.) For our first case, suppose that $w \in L$. Note that w is not in u's skeleton adjacency set, otherwise w would be in x's adjacency set, and hence we would not have a fill edge. This means that w is adjacent to one of the subtrees rooted at a child c of u. Since x is adjacent to every vertex of $\text{adj}_G(V(T))$ in the final fill graph and x is also adjacent to $T[c]$, this means that we can argue, just as in the proof of correctness, the existence of a 4-cycle that has the fill edge $\{x, w\}$ as its sole chord.

For our second case, suppose that $w \in X$. Since both x and w are adjacent to all subtrees rooted at children of u, we can again argue, as above and in the proof of correctness, the existence of a 4-cycle that has the fill edge $\{x, w\}$ as its sole chord. □

LEMMA 5.3. *Let u be a vertex of maximum cardinality chosen at some iteration of MCS-ETree such that u has no descendants in $T = T[v]$. Any vertex $x \in anc_T(u)$ that is adjacent to u and every vertex in $adj_G(u)$ (except x), can be ordered in a block along with u.*

Proof. In this case there is no fill edge incident to x and a higher numbered vertex so the result follows. □

Based on Lemma 5.2, we modified MCS-ETree to number all vertices of any block X described by the lemma at the end of the current iteration. Based on Lemma 5.3, we also modified MCS-ETree to number all vertices of any block described by the lemma at the end of the current iteration. We call our implementation that includes all the previous enhancements and this capability to number blocks of vertices the *blocked* implementation. Detection of the blocks is implemented by additional code within the Change_Root2 procedure that does not require any further traversal of adjacency sets. The vertices of a block are placed in the set $B(0)$, where they are labeled last by ordering γ_2 among the vertices of the current elimination subtree.

6. Test results. We have coded the *basic*, *enhanced*, and *blocked* implementations discussed in Section 5. For test results in an earlier technical report [15], we ran these implementations on a set of test problems taken from the Harwell-Boeing collection of sparse matrices. The initial orderings used in [15] were Approximate

Minimum Degree (AMD) [1] orderings and random orderings. As reported in earlier work [5, 24], minimum degree orderings are so close to minimal in practice that there is very little extraneous fill to remove. Consequently, the practical impact of MCS-ETree is extremely limited when AMD initial orderings are used. But our timing results in [15] indicate that the best implementation of MCS-ETree is very efficient on AMD initial orderings. A full set of tables and discussion of results on AMD and random initial orderings can be found in our technical report [15].

In this paper, we run our three implementations of MCS-ETree on a set of test problems taken from the sparse matrix collection of Tim Davis. A greater variety of stuctural analysis problems are included, along with a number of problems from optimization and other application areas. The structural analysis problems include BCSSTK17, BCSSTK25, SRBEDDY, CRYSTK02, CRYSTK03, nasasrb, pkustk01, pwt, shuttle_eddy, skirt, tandem_dual, and pli. The optimization problems include ex3sta1, fxm4_6, gupta1, minfurfo, and pfinan512. Also included are Helmholtz equations on a unit cube (helm3d01), a CFD problem (Pres_Poisson), a quantum chemistry problem (nmeth02), a thermoelasticity problem (ted_b), and an acoustics problem (vibrobox).

Also, we look at nested dissection (ND) initial orderings only. For ND initial orderings there is sometimes a significant amount of extraneous fill to remove; hence MCS-ETree is tested in a more demanding setting and gives results of more practical consequence. We also run a code that implements the algorithm from Peyton [24] for solving the same problem, and compare our algorithm with this algorithm, since it has the fastest documented practical running time. We would remind that the theoretical running time bound of the algorithm of [24] is not known.

Table 6.1 reports the number of vertices in each graph and the number of edges in the filled graphs for the ND orderings and the minimal orderings obtained from the *blocked* implementation of MCS-ETree. Also shown are the number of factorization operations that result from the ND orderings and the minimal orderings. The percent decrease in edges is less than 2% for 12 of the 22 problems. It is greater than 5% for 6 of the 22 problems. For matrix ted_B, which comes from coupled linear thermoelasticity equations, the decrease is 17%. For nemeth02, which comes from a Newton-Schultz iteration for a chemistry problem, the decrease is 52%. The latter is a long "path-like" problem for which nested dissection is inappropriate unless one seeks to exploit parallelism during the factorization.

Looking at the reductions in factorization operations gives us more matrices where MCS-ETree has some practical impact. Nonetheless, the reduction is less than 5% for 12 of the 22 problems. For three of the structural analysis matrices used also in our technical report [15], namely BCSSTK17, BCSSTK25, and SRBEDDY, the decreases are roughly 15%, 18%, and 19.5%, respectively. Two structural analysis matrices added to our problem set for this paper, namely nasasrb and shuttle_eddy, have decreases of roughly 14% and 19%, respectively. The two matrices with the largest fill reductions, namely ted_B and nemeth02, have decreases of roughly 29% and 76% respectively.

Table 6.2 reports the CPU time in seconds for the ND orderings, for each of the three implementations of algorithm MCS-ETree, and for the algorithm of [24]. The tests were run on a PC with a Pentium 4 processor running at 2.66 GHz with 0.99 GB RAM available. The code was written in Fortran and executed under the Linux operating system. We used the Metis software package available from the University of Minnesota to compute nested dissection orderings.

Matrix	$\|V\|$	Edges in filled graph			Factorization operations		
		ND ($\times 10^3$)	MCS-ETree ($\times 10^3$)	% decr.	ND ($\times 10^6$)	MCS-ETree ($\times 10^6$)	% decr.
BCSSTK17	10974	1126	1061	5.81	191.2	162.6	14.95
BCSSTK25	15439	1541	1434	6.89	350.4	286.1	18.35
SRBEDDY	46772	7527	7057	6.25	2190.1	1764.6	19.43
CRYSTK02	13965	4248	4205	1.02	1923.5	1863.2	3.13
CRYSTK03	24696	9508	9390	1.24	5631.4	5388.9	4.31
nasasrb	54870	10505	10011	4.70	3559.9	3055.4	14.17
pkustk01	22044	2075	2053	1.04	421.9	414.9	1.66
pwt	36519	1346	1341	0.39	110.9	110.3	0.52
shuttle_eddy	10429	352	329	6.58	22.2	17.9	19.00
skirt	12598	466	453	2.65	31.1	29.5	5.08
tandem_dual	94069	6481	6466	0.23	2265.0	2260.7	0.19
helm3d01	32226	4914	4900	0.29	2783.9	2773.5	0.37
pli	22695	13842	13799	0.31	15845.2	15813.4	0.20
Pres_Poisson	14822	2387	2338	2.05	553.2	522.1	5.63
ex3sta1	16782	7748	7658	1.16	7440.5	7276.8	2.20
fxm4_6	18892	435	424	2.65	23.5	22.5	4.53
gupta1	31802	2014	1987	1.33	297.1	270.9	8.83
minsurfo	40806	954	952	0.23	97.5	97.3	0.22
nemeth02	9506	460	220	52.05	24.5	5.8	76.35
pfinan512	74752	1748	1723	1.45	163.0	156.4	4.08
ted_B	10605	87	72	17.05	1.5	1.1	29.23
vibrobox	12328	2483	2466	0.68	1318.3	1310.4	0.60

TABLE 6.1

Number of vertices in each graph, the number of edges in each filled graph, and the number of factorization operations when the initial ordering is ND.

Matrix	ND time	MCS-ETree			Peyton [24] time
		(*basic*) time	(*enhanced*) time	(*blocked*) time	
BCSSTK17	0.124	9.769	0.820	0.052	0.496
BCSSTK25	0.256	10.141	1.688	0.088	0.604
SRBEDDY	0.188	129.492	10.225	0.280	0.648
CRYSTK02	0.144	38.538	3.016	0.088	0.144
CRYSTK03	0.272	98.350	7.800	0.152	0.276
nasasrb	1.056	225.106	16.173	0.368	4.044
pkustk01	0.076	24.914	2.332	0.156	1.316
pwt	0.420	12.029	2.160	0.144	0.304
shuttle_eddy	0.096	2.356	0.400	0.040	0.088
skirt	0.148	1.952	0.316	0.052	0.556
tandem_dual	1.236	72.601	22.221	0.472	2.772
helm3d01	0.580	62.580	12.829	0.324	1.988
pli	0.972	94.538	11.277	0.148	1.776
Pres_Poisson	0.148	22.333	1.680	0.080	0.184
ex3sta1	0.336	64.776	10.205	0.240	3.056
fxm4_6	0.248	5.064	0.660	0.104	0.768
gupta1	1.680	38.178	17.565	6.876	17148.713
minsurfo	0.355	12.854	3.172	0.188	0.369
nemeth02	0.219	44.676	2.082	1.859	2.553
pfinan512	1.188	20.822	3.971	0.324	4.281
ted_B	0.057	2.057	0.168	0.076	0.148
vibrobox	0.293	17.010	2.592	0.080	0.867

TABLE 6.2

CPU seconds to compute the ND initial orderings and the minimal orderings using the algorithm of [24] and the three implementations of MCS-ETree.

The basic implementation of MCS-ETree has much larger run times than the ND code, and is clearly too inefficient for practical sparse matrix computations. The enhanced implementation of MCS-ETree is much faster than the basic implementation in every case. Often it is ten times faster, or close to ten times faster, than the basic implementation. But comparing run times for the enhanced implementation with the ND ordering times, it is obvious that the enhanced implementation is also too inefficient for practical sparse matrix computations, despite its improvements.

Our timings, however, improve dramatically for most problems as we move to the blocked implementation, which includes the blocking technique along with all the enhancements employed by the enhanced implementation. For 17 of the 22 problems, the blocked implementation runs more than ten times faster than the enhanced implementation. For 13 of the 22 problems, the blocked implementation runs more than 100 times faster than the basic implemenation.

The three problems for which the reduction in time from the basic implementation to the blocked implementation is smallest are gupta1, nemeth02, and ted_B. For each of these problems the number of blocks detected by the blocked implementation is unusually large relative to the number of vertices in the graph. For gupta1, there are 31,198 blocks and 31,802 vertices; for nemeth02, there are 7,619 blocks and 9506 vertices; and for ted_B, there are 8,750 blocks and 10,605 vertices. For nemeth02, the reduction from the enhanced implementation to the blocked implementation is very small — from 2.082 seconds to 1.859 seconds. The matrix gupta1, which arises in an LP optimization problem, presents the greatest difficulties to all the algorithms we have looked at. For gupta1, the reduction from the basic implementation to the blocked implementation is from 38.178 seconds to 6.876 seconds — a reduction of only 82%. The time for the ND ordering of gupta1 is also greater than the ND ordering time for any other matrix. The matrices nemeth02 and ted_B have by far the greatest percent reduction in edges, and it may be natural to pay more in time to remove a greater percentage of edges.

The algorithm of Peyton [24] runs reasonably fast for ND initial orderings (except on the matrix gupta1). It is not as fast as the blocked implementation of MCS-ETree for any test matrix when ND initial orderings are used; however there are a few instances where it is faster when AMD initial orderings are used [15]. In fairness, the implementation of the algorithm of [24] has not been improved to the extent that the blocked implementation of MCS-ETree has been improved. It would be interesting to see if implementation of the algorithm of [24] could be improved to the extent that it would prove more competitive than the blocked implementation of MCS-ETree for ND initial orderings.

The algorithm of [24] takes an exhorbitant amount of time (4.76 hours) to compute the minimal ordering for the matrix gupta1. It became clear to us what is going on when we saw that the AMD ordering time for gupta1 is 52.2 seconds, which is extremely large. It was shown in Heggernes et al. [14] that AMD is an $O(nm)$ algorithm and that there exist examples to which this worst-case time complexity applies. The algorithm of [24] relies on iterations of a restricted version of the minimum degree algorithm with exact degrees. The time complexity of minimum degree with exact degrees is $O(n^2m)$ [14]. This interesting test matrix fully reveals the vulnerability of the algorithm of [24].

Finally, run times for the blocked implementation are reduced to the point that MCS-ETree is fast enough to be considered for sparse matrix computations. For 16 of 22 problems, the time required by the blocked implementation of MCS-ETree is less

than that required by the ND algorithm. For only two of the 22 problems, the ratio of the time for the blocked implementation to the time for the ND ordering is greater than three; for gupta1 the ratio is 4.09 and for nemeth02 the ratio is 8.50.

7. Concluding remarks. We have introduced a new algorithm MCS-ETree for computing a minimal ordering whose minimal fill lies inside the fill of any given initial ordering. The $O(nm\,\mathrm{A}(m,n))$ running time complexity is virtually as good as the best known time complexity of $O(nm)$. In practical tests, our algorithm performs better than the previous fastest algorithm of [24], and has the advantage of having a provably good theoretical running time as well. Algorithm MCS-ETree explicitly deals with a current ordering and the structure associated with that ordering, at the cost of disjoint set union operations that lead to the extremely slowly growing $\mathrm{A}(m,n)$ term in its running time complexity. By explicitly computing and exploiting elimination subtrees and partial Cholesky column nonzero counts, one obtains a relatively simple algorithm whose proof of correctness is also relatively simple. The new algorithm is based on selecting a special vertex of maximum cardinality at each step and resembles in this regard the algorithm MCS-M introduced in [2].

The algorithm can be implemented in $O(nm\ \mathrm{A}(m,n))$ time by adapting three commonly-used sparse matrix algorithms that date from the mid-1980's to the mid-1990's:

1. An $O(m\ \mathrm{A}(m,n))$ algorithm for computing the number of nonzeros in each column of a Cholesky factor [12],
2. An $O(m)$ algorithm for computing equivalent reorderings [19], and
3. An $O(m\ \mathrm{A}(m,n))$ algorithm for computing an elimination tree [20].

We were able to improve the *basic* implementation to obtain much faster implementations. The first set of enhancements are straightforward programming-level improvements that greatly limit the number of times adjacency lists are traversed or shorten those lists to abbreviated skeleton adjacency lists. The other improvement allows blocks of vertices to be numbered by a single iteration of the algorithm, and this is based closely on the idea of indistinguishable vertex sets in elimination graphs exploited so successfully by implementations of the minimum degree algorithm [17].

We coded in Fortran the *basic*, *enhanced*, and *blocked* implementations and our timing results show that the blocked implementation is fast enough to be considered for use in sparse matrix computations. The best implementation of the algorithm could prove useful when the initial orderings are nested dissection orderings, because sometimes the fill and factorization operations can be significantly reduced by removing extraneous fill.

Acknowledgements. The authors thank the referees for their many helpful suggestions. We also thank Vince Postell for his help in gaining access to Linux and PSTricks.

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] A. Berry, J. R. S. Blair, P. Heggernes, and B. W. Peyton, *Maximum cardinality search for computing minimal triangulations of graphs*, Algorithmica, 39 (2004), pp. 287 – 298.

[3] A. Berry, J.-P. Bordat, P. Heggernes, G. Simonet, and Y. Villanger, *A wide-range algorithm for minimal triangulation from an arbitrary ordering*, J. Algorithms, 58 (2006), pp. 33–66.

[4] A. Berry, P. Heggernes, and G. Simonet, *The minimum degree heuristic and the minimal triangulation process*, Proceedings of the 29^{th} Workshop on Graph Theoretic Concepts in Computer Science, (2003), pp. 58 – 70. LNCS 2880.

[5] J. R. S. Blair, P. Heggernes, and J. A. Telle, *A practical algorithm for making filled graphs minimal*, Theor. Comput. Sci., 250 (2001), pp. 125–141.

[6] H. L. Bodlaender and A. M. C. A. Koster, *Safe separators for treewidth*, Tech. Rep. UU-CS-2003-027, Institute of information and computing sciences, Utrecht University, Netherlands, 2003.

[7] F. R. K. Chung and D. Mumford, *Chordal completions of planar graphs*, J. Comb. Theory, 31 (1994), pp. 96–106.

[8] E. Dahlhaus, *Minimal elimination ordering inside a given chordal graph*, in Graph Theoretical Concepts in Computer Science - WG '97, Springer Verlag, 1997, pp. 132–143. LNCS 1335.

[9] F. V. Fomin, D. Kratsch, and I. Todinca, *Exact (exponential) algorithms for treewidth and minimum fill-in*, in Automata, Languages and Programming - ICALP 2004, Springer Verlag, 2004, pp. 568 – 580. LNCS 3142.

[10] D. Fulkerson and O. Gross, *Incidence matrices and interval graphs*, Pacific Journal of Math., 15 (1965), pp. 835–855.

[11] J. A. George, *Nested dissection of a regular finite element mesh*, SIAM J. Numer. Anal., 10 (1973), pp. 345–363.

[12] J. R. Gilbert, E. G. Ng, and B. W. Peyton, *An efficient algorithm to compute row and column counts for sparse Cholesky factorization*, SIAM J. Matrix Anal. Appl., 15 (1994), pp. 1075–1091.

[13] P. Heggernes, *Minimal triangulations of graphs: A survey*, Disc. Math., 306 (2006), pp. 297–317.

[14] P. Heggernes, S. Eisenstat, G. Kumfert, and A. Pothen, *The computational complexity of the Minimum Degree algorithm*, in Proceeding of 14th Norwegian Computer Science Conference, NIK 2001, University of Tromso, Norway.

[15] P. Heggerenes and B. W. Peyton, *Fast computation of minimal fill inside a given elimination ordering*, Reports in Informatics 343, University of Bergen, January, 2007.

[16] S. L. Lauritzen and D. J. Spiegelhalter, *Local computations with probabilities on graphical structures and their applications to expert systems*, J. Royal Statist. Soc., ser B, 50 (1988), pp. 157–224.

[17] J. W. H. Liu, *Modification of the minimum degree algorithm by multiple elimination*, ACM Trans. Math. Software, 11 (1985), pp. 141–153.

[18] ———, *A compact row storage scheme for Cholesky factors using elimination trees*, ACM Trans. Math. Software, 12 (1986), pp. 127–148.

[19] ———, *Equivalent sparse matrix reorderings by elimination tree rotations*, SIAM J. Sci. Stat. Comput., 9 (1988), pp. 424–444.

[20] ———, *The role of elimination trees in sparse factorization*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 134–172.

[21] A. Natanzon, R. Shamir, and R. Sharan, *A polynomial approximation algorithm for the minimum fill-in problem*, SIAM J. Computing, 30 (2000), pp. 1067–1079.

[22] T. Nedretvedt, *Implementation of a minimal triangulation algorithm for studying properties of the Minimum Degree heuristic*, Master's thesis, University of Bergen, Norway, 2005.

[23] S. Parter, *The use of linear graphs in Gauss elimination*, SIAM Review, 3 (1961), pp. 119–130.

[24] B. W. Peyton, *Minimal orderings revisited*, SIAM J. Matrix Anal. Appl., 23 (2001), pp. 271–294.

[25] D. J. Rose, R. E. Tarjan, and G. Lueker, *Algorithmic aspects of vertex elimination on graphs*, SIAM J. Comput., 5 (1976), pp. 146–160.

[26] D. J. Rose, *A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations*, in Graph Theory and Computing, R. C. Read, ed., Academic Press, New York, 1972, pp. 183–217.

[27] R. E. Tarjan and M. Yannakakis, *Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs*, SIAM J. Comput., 13 (1984), pp. 566–579.

[28] M. Yannakakis, *Computing the minimum fill-in is NP-complete*, SIAM J. Alg. Disc. Meth., 2 (1981), pp. 77–79.