REPORTS IN INFORMATICS

ISSN 0333-3590

Fast computation of minimal fill inside a given elimination ordering

Pinar Heggernes **Barry W. Peyton**

REPORT NO 343 **January 2007**

Department of Informatics

UNIVERSITY OF BERGEN

Bergen, Norway

This report has URL http://www.ii.uib.no/publikasjoner/texrap/pdf/2007-343.pdf

Reports in Informatics from Department of Informatics, University of Bergen, Norway, is available at http://www.ii.uib.no/publikasjoner/texrap/.

Requests for paper copies of this report can be sent to:

Department of Informatics, University of Bergen, Høyteknologisenteret,
P.O. Box 7800, N-5020 Bergen, Norway

Fast computation of minimal fill inside a given elimination ordering

Pinar Heggernes* 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.

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. 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.)

*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

For sparse matrix computations [24] and in many other fields [7, 14, 25], one would like to find orderings that produce the minimum possible fill. This problem was shown to be NP-hard by Yannakakis in 1981 [26]. Already in 1976, Rose, Tarjan, and Lueker [23] 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 triangulation. For sparse matrix computations, minimal elimination orderings are highly desirable because 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 [19] 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, 15] 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 time $O(nm)$ [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 [22], 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 [20].

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 practical 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 [17], which are used by our new algorithm in a slightly modified form. Section 4

[1] In fact the algorithm of Peyton [22] is the fastest in practice of all mentioned algorithms.

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 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, $\mathrm{adj}_G(v)$ denotes the set of vertices adjacent to v, also called the *adjacency* or *neighborhood* of v, and $\mathrm{adj}_G[v] = \mathrm{adj}_G(v) \cup \{v\}$. For a set of vertices $X \subseteq V$, $\mathrm{adj}_G(X) = \bigcup_{x \in X} \mathrm{adj}_G(x) \setminus X$ and $\mathrm{adj}_G[X] = \mathrm{adj}_G(X) \cup X$. An *ordering* (also called a *linear layout*) 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, $\mathrm{hadj}_{G_\alpha}(v_i) = \mathrm{adj}_{G_\alpha}(v_i) \cap \{v_{i+1}, v_{i+2}, ..., v_n\}$ is the set of higher numbered neighbors of v_i, and $\mathrm{ladj}_{G_\alpha}(v_i) = \mathrm{adj}_{G_\alpha}(v_i) \cap \{v_1, v_2, ..., v_{i-1}\}$ 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 $E(G) \subset E(H)$. (Relations $\subseteq$ and $\subset$ are referred to as the *subset* and the *proper subset* relations, respectively.) 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 are adjacent in G. A *path* is a sequence of vertices $x_1 - x_2 - ... - x_k$ such that x_i is adjacent to x_{i+1} for $1 \leq i < k$. The *length* of a path is the number of vertices it contains, and we will refer to a path on k vertices as a *k-path*. 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* [21], 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. The following lemma characterizes the edges in G^+_α.

Lemma 1 [23] *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 [25].

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, 23]. The following is a characterization of minimal triangulations that we will use in the proof that our new algorithm is correct.

Theorem 1 [23] *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 $\text{hadj}_{G_\alpha^+}(v_i)$. Due to Lemma 1, the elimination tree corresponding to α can be computed directly from G and α without computing G_α^+ explicitly [18]. A *topological ordering* of T is any ordering that numbers each child with a number smaller than that of its parent. 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 gives a thorough examination of elimination trees in [18]. 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 simple 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 $\text{hadj}_{G_\alpha^+}(v) = \text{adj}_G(V(T[v]))$ [18], and we will make use of this fact throughout the paper. We let $\text{anc}_T(v)$ be the set of ancestors of v in T, where v is *not* included in the set. We write $\text{anc}_T[v] = \text{anc}_T(v) \cup \{v\}$.

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 with vertices $V(T[v])$ and associated with the new equivalent reordering. A trivial modification of the composite elimination tree rotations algorithm in Liu [17] 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 [17] reorders G with a peo γ of G_β^+ such that the vertices of $\text{adj}_G(V(T[u]))$ 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 $\text{adj}_G(V(T[u]))$, and hence u will often become closer to the root of the resulting new elimination tree corresponding to γ, and will never

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 ← u;
while z is not the root of T do
    Order the unnumbered vertices of 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 ← the parent of z in T;
end while;
Number in γ the vertices in V \ anc_T[u] using their original
relative order in T;
```

end Change_Root;

Figure 1: An algorithm for changing the root of an elimination tree with an equivalent reordering (See Algorithm 3.2 Composite_Rotations in Liu [17]).

be further away than it is in T. The algorithm Change_Root in Figure 1 adds a single first line to Liu's Composite_Rotations algorithm in order to number u last, and this is the only modification 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.

Observe that for every vertex u, $\mathrm{adj}_G(V(T[u])) \cup \{u\} = \mathrm{adj}_{G^+_\beta}(u) \cup \{u\}$ is a clique in G^+_β. Hence it follows by the correctness of Algorithm Composite_Rotations from [17] that the ordering γ produced by Change_Root is also a peo of G^+_β, and thus an equivalent reordering of G with respect to β.

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 $H = G(V(T[v]))$. Let $u \in V(T[v])$. When we execute Change_Root($T[v]$, u, H), 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 give the next α-number to. Hence the algorithm resembles a minimal triangulation algorithm called MCS-M [2], and its proof of correctness

uses the same technique used there. Ultimately, 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 2. 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 $Trees$.

The overall structure of the algorithm is viewed in the proof of the following lemma.

Lemma 2 *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 $\mathrm{adj}_G(V(T)) \subseteq L$. The next step in the iteration chooses a vertex $u \in V(T)$. The next step reorders the connected component $H = G(V(T))$ so that u is numbered last. The next step 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 [18] 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 2 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 subtree in $Trees$ is an elimination subtree of the elimination tree with respect to a current ordering 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

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.

Compute the elimination tree T^* of G with respect to β;
$x \leftarrow$ root of T^*;
$Trees \leftarrow \{T^*[x]\}$; $L \leftarrow \emptyset$; $k \leftarrow n$;

while $Trees \neq \emptyset$ **do**

Pick an arbitrary elimination subtree $T = T[v]$ from $Trees$;
$Trees \leftarrow Trees \setminus \{T\}$;

Find a vertex $u \in V(T)$ for which $|\text{adj}_G(V(T[u])) \cap L| = |\text{adj}_G(V(T))|$, and $|\text{adj}_G(V(T[w])) \cap L| < |\text{adj}_G(V(T[u])) \cap L|$ for each descendant w of u in T;

$H \leftarrow G(V(T))$;
Compute a topological order γ_1 of T;
Use Change_Root(T, u, H) to compute a peo γ_2 of $H^+_{\gamma_1}$ that numbers u last;

Compute the elimination tree $T' = T'[u]$ of H with respect to γ_2;

for each child c of u in T' **do**
$Trees \leftarrow Trees \cup \{T'[c]\}$;
end for;
$\alpha(u) \leftarrow k$; $L \leftarrow L \cup \{u\}$; $k \leftarrow k - 1$;

end while;

end MCS-ETree;

Figure 2: Algorithm MCS-ETree, which finds a minimal ordering and solves the minimal triangulation sandwich problem.

descendant w of u. Since $\text{hadj}_{G^+_\gamma}(x) \cap L = \text{adj}_G(V(T[x])) \cap L$ for every vertex $x \in V(T)$, the algorithm equivalently chooses a vertex $u \in V(T)$ for which $|\text{adj}_G(V(T[u])) \cap L| = |\text{adj}_G(V(T))|$ and for which $|\text{adj}_G(V(T[w])) \cap L| < |\text{adj}_G(V(T[u])) \cap L|$ for every descendant w of u. Notice that such a vertex u always exists, since $\text{adj}_G(V(T[v])) \cap L = \text{adj}_G(V(T))$, and hence v can be chosen as u if no descendant x of v satisfies $|\text{adj}_G(V(T[x])) \cap L| = |\text{adj}_G(V(T))|$. Furthermore, if every descendant x of v satisfies $|\text{adj}_G(V(T[x])) \cap L| = |\text{adj}_G(V(T))|$, then u is chosen to be a leaf of T.

A second important step within each iteration uses a call to 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 3 *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 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 $x \notin \text{anc}_T[u]$, then from Algorithm Change_Root we have $T'[x] = T[x]$, so by Lemma 1 we have $\{x, y\} \in E(G^+_\gamma)$. If $x \in \text{anc}_T[u]$, then by the choice of u in the algorithm and simple properties of elimination trees [18],

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

This completes the proof. ∎

Theorem 2 *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 3 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 1.

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 3, 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 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 1, we know that $\text{hadj}_{G^+_\gamma}(c) \cap L \subseteq \text{hadj}_{G^+_\gamma}(u) \cap L$ for each child c of u in T. From the choice of u, we can conclude that $\text{hadj}_{G^+_\gamma}(c) \cap L \subset \text{hadj}_{G^+_\gamma}(u) \cap L$ (proper subset). Let $y \in (\text{hadj}_{G^+_\gamma}(u) \cap L) \setminus (\text{hadj}_{G^+_\gamma}(c) \cap L)$, which is not empty. By Lemma 1, none of the vertices x_i is adjacent to y in G^+_γ (including x_t). By Lemma 3, $\{x_t, y\}$ is not an edge in the final filled graph G^+_α.

Finally, the set $\text{hadj}_{G^+_\gamma}(u) \cap L = \text{hadj}_{G^+_\gamma}(v)$ 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 $\text{hadj}_{G^+_\gamma}(u) \cap L = \text{hadj}_{G^+_\alpha}(u)$, and recall that $\text{hadj}_{G^+_\alpha}(u) \cup \{u\}$ is a clique in G^+_α. Note that both y and w belong to $\text{hadj}_{G^+_\alpha}(u)$. 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 1, 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 running time bound of $O(nm\,\text{A}(m, n))$ for Algorithm MCS-ETree.

Theorem 3 *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 $|\text{hadj}_{G^+_\gamma}(x) \cap L| = |\text{adj}_G(T[x]) \cap L|$ 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 $|\text{hadj}_{G^+_\gamma}(x) \cap L|$ for every vertex $x \in V(T)$. This option leads to $O(nm\,\text{A}(m, n))$ 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 $|\text{ladj}_{G_\gamma^+}(x) \cup \{x\}|$ (the "row count") and $|\text{hadj}_{G_\gamma^+}(x) \cup \{x\}|$ (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 $|\text{hadj}_{G_\gamma^+}(x) \cap L|$ rather than $|\text{hadj}_{G_\gamma^+}(x) \cup \{x\}|$. 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 $\text{anc}_T[u]$ 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 [18] 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 $O(nm\,\text{A}(m,n))$. 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* [16] 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.

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 3 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 remains $O(nm\,\text{A}(m,n))$. 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 $\text{anc}_T(u)$ 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 $x \in \text{anc}_T(u)$ 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

```
procedure Change_Root2(T, γ1, u, G, 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;
```

Figure 3: An enhanced variant of algorithm Change_Root for the second step in the main loop of MCS-ETree.

skeleton neighbors in L), then x can be ordered in a block along with u (see Lemma 4). 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. 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).

Lemma 4 *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 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 *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 4, 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, 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. We have run these implementations on a set of test problems taken from the Harwell-Boeing

Matrix	$\|V\|$	AMD	MCS-ETree			Peyton [22]
			(*basic*)	(*enhanced*)	(*blocked*)	
		$\|E(G_\beta^+)\|$	$\|E(G_\alpha^+)\|$	$\|E(G_\alpha^+)\|$	$\|E(G_\alpha^+)\|$	$\|E(G_\alpha^+)\|$
BCSSTK13	2003	263939	263876	263876	263876	263876
BCSSTK15	3948	623820	623815	623815	623815	623815
BCSSTK16	4884	807299	807299	807299	807299	807299
BCSSTK17	10974	1044953	1042116	1042116	1042116	1042116
BCSSTK18	11948	624786	624248	624248	624248	624248
BCSSTK23	3134	428210	428210	428210	428210	428210
BCSSTK25	15439	1463677	1457094	1457094	1457094	1457094
BCSSTK28	4410	335965	335965	335965	335965	335965
BCSSTK29	13992	1757897	1755088	1755083	1755083	1755082
BCSSTK30	28924	3824804	3824720	3824720	3824720	3824720
BCSSTK31	35588	5521612	5521594	5521594	5521594	5521594
BCSSTK32	44609	4941894	4938563	4938562	4938562	4938562
BCSSTK33	8738	2562558	2561456	2561456	2561456	2561456
BCSSTK35	30237	2701793	2695630	2695630	2695630	2695594
BCSSTK36	23052	2709459	2709459	2709459	2709459	2709459
BCSSTK37	25503	2773697	2773562	2773562	2773562	2773562
BCSSTK38	8032	743524	741003	741003	741003	741003
SRBEDDY	46772	6641014	6638704	6638704	6638704	6638704
CRYSTK01	4875	955936	955934	955934	955934	955934
CRYSTK02	13965	6104963	6104961	6104961	6104961	6104961
CRYSTK03	24696	12778474	12778471	12778471	12778471	12778471

Table 1: Number of vertices in each graph and the number of edges in each filled graph when the initial ordering is AMD.

collection of sparse matrices. Two sets of experiments were performed. For the first set, each matrix was initially ordered using the Approximate Minimum Degree algorithm (AMD) of Amestoy, Davis, and Duff [1]. The three different implementations of the MCS-ETree algorithm were then applied to the initial AMD orderings to obtain minimal orderings where the final fill is a subset of the fill obtained under the original AMD ordering. We also ran a code that implements the algorithm from Peyton [22] for solving the same problem, and compared our algorithm to this algorithm, since it has the fastest documented practical running time. We would like to remind that the theoretical running time bound of the algorithm of [22] is not known.

Table 1 reports the number of vertices in each graph and the number of edges in the filled graphs for the AMD orderings and the minimal orderings obtained from the algorithm of [22] and all three implementations of MCS-ETree. As reported in earlier work [5, 22], the minimum degree algorithm applied to problems encountered in practice produces orderings that are very close to minimal; very few edges are removed to obtain the minimal chordal supergraphs. Also note that the different implementations of MCS-ETree and the algorithm of [22] are apparently removing the same set of fill edges in most cases; the only matrices for which the fill size varies among the implementations of MCS-ETree and the algorithm of [22] are BCSSTK29, BCSSTK32, and BCSSTK35.

Table 2 reports the CPU time in seconds for the AMD orderings, for each of the three

Matrix	AMD time	MCS-ETree (*basic*) time	MCS-ETree (*enhanced*) time	MCS-ETree (*blocked*) time	Peyton [22] time
BCSSTK13	0.009	1.088	0.234	0.013	0.023
BCSSTK15	0.017	2.895	0.437	0.019	0.040
BCSSTK16	0.012	7.510	0.633	0.032	0.028
BCSSTK17	0.031	14.981	1.132	0.060	0.177
BCSSTK18	0.053	6.399	1.178	0.070	0.112
BCSSTK23	0.014	1.258	0.341	0.014	0.012
BCSSTK25	0.079	15.824	2.455	0.099	0.277
BCSSTK28	0.005	2.385	0.244	0.027	0.017
BCSSTK29	0.037	25.348	2.401	0.111	0.155
BCSSTK30	0.104	150.948	8.722	0.304	0.438
BCSSTK31	0.194	102.897	13.515	0.322	0.529
BCSSTK32	0.161	99.396	9.745	0.316	0.585
BCSSTK33	0.033	28.457	2.259	0.069	0.135
BCSSTK35	0.064	62.213	5.245	0.235	0.349
BCSSTK36	0.047	45.433	4.390	0.175	0.122
BCSSTK37	0.064	52.467	4.965	0.182	0.282
BCSSTK38	0.023	10.341	0.831	0.060	0.144
SRBEDDY	0.078	171.956	13.366	0.371	0.501
CRYSTK01	0.013	8.985	0.714	0.031	0.063
CRYSTK02	0.045	73.403	6.185	0.128	0.212
CRYSTK03	0.088	214.216	20.190	0.231	0.390

Table 2: CPU seconds to compute the AMD initial orderings and the minimal orderings using the algorithm of [22] and the three implementations of MCS-ETree.

implementations of algorithm MCS-ETree, and for the algorithm of [22]. The tests were run on a PC with a Pentium 4 processor running at 2.20 GHz with 768 MB RAM available. The code was written in Fortran using Salford FTN95, which is a Fortran95 compiler and code development environment under the Windows operating system. The basic implementation of MCS-ETree has much larger run times than the AMD 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 AMD 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 as we move to the blocked implementation, which includes the blocking technique along with all the enhancements employed by the enhanced implementation. In every case but one, the blocked implementation runs more than ten times faster than the enhanced implementation. (For the exception, BCSSTK28, the reduction is from 0.244 seconds to 0.027 seconds.) Often the blocked implementation is much better than ten times faster than the enhanced implementation. The poorest reduction going from the basic implementation to the blocked implementation is for BCSSTK13, where the run time is reduced from 1.088 seconds to 0.013 seconds. Here the blocked implementa-

tion runs roughly 84 times faster than the basic implementation. The best reduction going from the basic implementation to the blocked implementation is for CRYSTK03, where the run time is reduced from 214.216 seconds to 0.231 seconds. Here the blocked implementation runs roughly 927 times faster than the basic implementation. In every case but four (BCSSTK13, BCSSTK18, BCSSTK23, and BCSSTK28), the blocked implementation runs over 100 times faster than the basic implementation.

The algorithm of Peyton [22] runs fast for AMD initial orderings, but generally not as fast the the blocked implementation of MCS-ETree. In fairness, the implementation of the algorithm of [22] 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 [22] could be improved to the extent that it would prove more efficient than the blocked implementation of MCS-ETree for AMD initial orderings. Note that there are four problems for which the algorithm of [22] runs faster than the blocked implementation of MCS-ETree.

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 every case but two, the ratio of the time for the blocked implementation to the time for the AMD ordering is less than four. For BCSSTK28 the ratio is 5.25 and for SRBEDDY the ratio is 4.77. In every case, the ratio of the time for the blocked implementation to the time for the AMD ordering is less than six. In no case, however, is the blocked implementation of MCS-ETree actually faster than the AMD ordering time.

The second set of experiments is exactly the same as the first set, except the initial ordering is changed. For each problem we compute a random ordering, which produces very poor ordering quality. We do not recommend this as a way to compute minimal orderings; the resulting minimal fill graphs have many more fill edges than those obtained using AMD as the initial ordering. We run the experiment to show how the algorithm of [22] and the three implementations of MCS-ETree perform when there is much fill to remove to achieve minimality. The results of these experiments are reported in Tables 3 and 4.

The first point to make is that the improved implementations still improve performance dramatically even though the initial orderings are so poor. But the overall improvements are not as dramatic as we observed with AMD initial orderings. In only one case does the blocked implementation run more than 100 times faster than the basic implementation (CRYSTK03). In two cases the blocked implementation runs less than ten times faster than the basic implementation (BCSSTK18 and BCSSTK23). For BCSSTK25 the blocked implementation runs approximately 15 times faster than the basic implementation. For every other case but these four mentioned above, the blocked implementation runs better than 20 times faster, but not more than 100 times faster, than the basic implementation.

The second point to make is that the algorithm of [22] has extremely high run times when the initial order is random. In most cases it runs slower than the basic implementation of MCS-ETree. On problem BCSSTK31 it required approximately 63 minutes of CPU time. Again, the implementation of the algorithm of [22] could be improved; but it could not be improved enough to be competitive in this setting. All the run times for the blocked implementation of MCS-ETree are less than 10 seconds. We view this experiment as strong evidence that the blocked implementation of MCS-ETree can handle very effectively challenging problems where the initial ordering is far from minimal.

Matrix	$\|V\|$	Random	MCS-ETree			Peyton [22]
			(*basic*)	(*enhanced*)	(*blocked*)	
		$\|E(G_\beta^+)\|$	$\|E(G_\alpha^+)\|$	$\|E(G_\alpha^+)\|$	$\|E(G_\alpha^+)\|$	$\|E(G_\alpha^+)\|$
BCSSTK13	2003	1633578	546144	540516	541501	541512
BCSSTK15	3948	6077701	1325381	1325222	1325222	1337114
BCSSTK16	4884	10179273	1127807	1127112	1127112	1172678
BCSSTK17	10974	43219330	2990858	2987686	2987872	3064417
BCSSTK18	11948	29454753	3496976	3494655	3494819	3481613
BCSSTK23	3134	3094857	1454041	1455144	1455144	1429839
BCSSTK25	15439	71225932	7908015	7908618	7909345	7821653
BCSSTK28	4410	7791171	609597	607191	607191	614413
BCSSTK29	13992	76801577	4427957	4429850	4429850	4168807
BCSSTK30	28924	354859808	13670277	13669708	13669738	14195060
BCSSTK31	35588	466567346	23419653	23402473	23402343	23057216
BCSSTK32	44609	794739747	24084167	24091646	24091783	22772317
BCSSTK33	8738	33968641	8522284	8526351	8526351	8216924
BCSSTK35	30237	368375438	8112005	8107592	8107223	7756561
BCSSTK36	23052	218922773	13510689	13510786	13510786	12838673
BCSSTK37	25503	257642682	8311035	8308697	8308697	8102911
BCSSTK38	8032	25823466	1594218	1591484	1591890	1570685
SRBEDDY	46772	866571820	18913142	18892317	18892886	18542626
CRYSTK01	4875	10503691	2262935	2257689	2257689	2014703
CRYSTK02	13965	86714036	12738450	12723152	12723152	13946811
CRYSTK03	24696	275748152	53027237	52976945	52977377	51139144

Table 3: Number of vertices in each graph and the number of edges in each filled graph when the initial ordering is random.

Matrix	MCS-ETree			Peyton
	(*basic*) time	(*enhanced*) time	(*blocked*) time	[22] time
BCSSTK13	1.523	0.506	0.054	3.449
BCSSTK15	6.232	1.274	0.219	9.635
BCSSTK16	15.297	2.249	0.434	7.583
BCSSTK17	32.211	6.507	1.244	68.457
BCSSTK18	18.466	6.241	1.966	353.526
BCSSTK23	2.541	1.058	0.414	17.537
BCSSTK25	49.044	15.733	3.221	899.952
BCSSTK28	6.978	0.961	0.101	1.866
BCSSTK29	43.666	8.982	1.000	136.593
BCSSTK30	279.191	53.497	7.707	421.552
BCSSTK31	315.942	72.428	7.784	3773.641
BCSSTK32	401.715	78.585	7.498	3046.106
BCSSTK33	62.520	11.770	1.919	29.004
BCSSTK35	120.340	29.282	3.139	441.933
BCSSTK36	153.582	36.252	3.337	172.391
BCSSTK37	157.057	43.108	6.066	355.907
BCSSTK38	20.294	2.939	0.319	28.779
SRBEDDY	425.434	78.068	5.689	811.697
CRYSTK01	15.793	3.647	0.407	6.070
CRYSTK02	131.571	20.741	2.033	74.701
CRYSTK03	481.491	104.865	3.338	671.777

Table 4: CPU seconds to solve the minimal triangulation sandwich problem when the initial ordering input into the algorithm of [22] and the three MCS-ETree implementations is random.

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 the best known time complexity of $O(nm)$. In practical tests, our algorithm performs at least as good as the previous fastest algorithm of [22], 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 $O(nm\,\mathrm{A}(m,n))$ algorithm for computing the number of nonzeros in each column of a Cholesky factor [12],

2. An $O(nm)$ algorithm for computing equivalent reorderings [17], and

3. An $O(nm\,\mathrm{A}(m,n))$ algorithm for computing an elimination tree [18].

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 [15].

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 if one is concerned with squeezing out any remaining extraneous fill or one is concerned that the space requirement does not change when an equivalent reordering is computed. When Approximate Minimum Degree (AMD) was used as the initial ordering, in no case did the blocked implementation of MCS-ETree take more than six times as much time as the original AMD ordering. It is well documented that AMD is very fast on such problems as we tested, and our timings bear that out.

Finally, we close with an open question. The blocked implementation of MCS-ETree is very fast when the initial ordering is an AMD ordering; it is even quite fast when the initial ordering is random. Are there special initial orderings, such as Lexicographic Breadth First Search orderings or Maximum Cardinality Search orderings, that can be obtained in linear time, and might provide MCS-ETree the opportunity to compute minimal orderings with a running time complexity better than $O(nm\,\mathrm{A}(m,n))$?

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] 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.

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

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

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

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

[19] 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.

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

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

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

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

[24] 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.

[25] 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.

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