Constructing elimination trees for sparse unsymmetric matrices

Kamer Kaya, Bora Uçar

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Constructing elimination trees for sparse unsymmetric matrices

Kamer Kaya — Bora Uçar

N° 7549

February 2011

Distributed and High Performance Computing

ISSN 0249-6399 ISRN INRIA/RR--7549--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

Constructing elimination trees for sparse unsymmetric matrices

Kamer Kaya* , Bora Uçar†

Theme : Distributed and High Performance Computing
Équipe-Projet GRAAL

Rapport de recherche n° 7549 — February 2011 — 13 pages

Abstract: The elimination tree model for sparse unsymmetric matrices and an algorithm for constructing it have been recently proposed [Eisenstat and Liu, SIAM J. Matrix Anal. Appl., 26 (2005) and 29 (2008)]. The construction algorithm has a worst case time complexity $\mathcal{O}(mn)$ for an $n \times n$ unsymmetric matrix having m nonzeros. We propose another algorithm that has a worst case time complexity of $\mathcal{O}(m \log n)$.

Key-words: Elimination tree, sparse matrix factorization

Revised: October 2011

* CERFACS, Toulouse, France
† CNRS and ENS Lyon, France

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Construire les arbes d'élimination pour des matrices creuses non symétriques

Résumé : Eisenstat et Liu ont récemment décrit l'arbre d'élimination pour des matrices creuses non symétriques et ont proposé un algorithme pour le construire en temps $\mathcal{O}(mn)$ pour une matrice de taille n ayant m éléments non nuls [Eisenstat and Liu, SIAM J. Matrix Anal. Appl., 26 (2005) and 29 (2008)]. Nous décrivons un algorithme dont la complexité en temps est $\mathcal{O}(m \log n)$.

Mots-clés : Arbre d'élimination, factorisation des matrices creuses

1 Introduction

Arguably, the elimination tree is the single most important data structure in sparse matrix factorization methods. It is used to estimate and optimize the storage and computational requirements during symbolic and numerical factorizations [9]. Although the elimination tree for symmetric (positive definite) systems goes back to 80s [10] and even before (see [3, Section 3.3] and [9]), the elimination tree for unsymmetric matrices is recent. Eisenstat and Liu [5] define the the elimination tree for unsymmetric matrices and discuss its properties. In a follow up paper [6], they describe algorithms to construct those trees, to reorder matrices into a special form, and discuss the use of the elimination trees to improve the performance of a symbolic factorization algorithm. In this paper, we present an $\mathcal{O}(m \log n)$ time algorithm to construct the elimination tree of an $n \times n$ unsymmetric matrix with m nonzeros. The algorithm by Eisenstat and Liu [6] has a time complexity $\mathcal{O}(mn)$.

A directed graph $G = (V, E)$ consists of a finite set of vertices V and a set of edges E, where each edge $(u, v) \in E$ is a binary relation on V. An edge of the form (u, u) is called a *self-loop*. Let $\mathbf{A} = [a_{ij}]$ be a sparse $n \times n$ matrix with m nonzeros. The directed graph $G(\mathbf{A}) = (V, E)$ of $\mathbf{A}$ contains n vertices, $V = \{1, 2, \ldots, n\}$, and m edges, $E = \{(i, j) : a_{ij} \neq 0, \text{ for } 1 \leq i, j \leq n\}$. We assume that $\mathbf{A}$ has a zero-free diagonal and therefore $(u, u) \in E$ for all $u \in V$ (this simplifies some of the definitions and guarantees that $|E| \geq |V|$).

A *path* of length k in a directed graph $G = (V, E)$ is a sequence of vertices $v_0, v_1, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $i = 0, \ldots, k-1$. For two vertices $u, v \in V$, we say that u is *connected* to v in G, denoted $u \overset{G}{\to} v$, if there is path from u to v. A cycle is a path which starts and ends at the same vertex. A simple cycle is a path $v_0, v_1, \ldots, v_k, v_0$ where $k \geq 1$ and $v_i \neq v_j$ for $0 \leq i < j \leq k$.

A directed graph $G' = (V', E')$ is a *subgraph* of a larger digraph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E \cap (V' \times V')$. A subgraph is said to be a *induced subgraph* if $E' = E \cap (V' \times V')$. Let $V = \{v_1, v_2, \ldots, v_n\}$ and we have a total order on the vertices. Then, G_k denotes a special induced subgraph containing the first k vertices in V. That is,

$$G_k = (\{1, \ldots, k\}, E \cap \{v_1, \ldots, v_k\} \times \{v_1, \ldots, v_k\}) .$$

If $u \overset{G}{\to} v$ for all $u, v \in V$, the digraph G is called *strongly connected*. A maximal, strongly connected, induced subgraph C of G is called a strong component of G. Due to the zero-free diagonal assumption, the vertices have self-loops. Hence, if G contains a single vertex, it is strongly connected. A directed graph without simple cycles is called a dag (otherwise put, all cycles of a dag are self-loops). The strong components of a dag are its vertices, i.e., each vertex forms a trivial strong component.

Let $G = (V, E)$ be a directed graph and $\Pi = \{V_1, V_2, \ldots, V_\ell\}$ be a partition of the vertex set V. The *quotient graph* $G^\Pi = (V^\Pi, E^\Pi)$ induced by Π has ℓ super-vertices $V_1, \ldots, V_\ell$ and a directed edge $(V_i, V_j) \in E^\Pi$ iff $(u, v) \in E$ for some $u \in V_i$ and $v \in V_j$.

Let $\mathbf{A}$ be a nonsingular, sparse, unsymmetric $n \times n$ matrix with a nonzero diagonal, and assume that the LU-factorization $\mathbf{A} = \mathbf{LU}$ with unit lower triangular $\mathbf{L}$ and upper triangular exist. Then, the *elimination tree* $T(\mathbf{A})$ of an

unsymmetric $\mathbf{A}$ is defined by Eisenstat and Liu [5, 6] as follows.

$$\textsc{Parent}(i) = \min\{j : j > i \text{ and } j \overset{G(\mathbf{L})}{\rightarrow} i \overset{G(\mathbf{U})}{\rightarrow} j\} \tag{1}$$

where $\textsc{Parent}(i)$ denotes the parent of vertex i in the elimination tree if i is not a root. Otherwise, $\textsc{Parent}(i) = \infty$. Eisenstat and Liu prove the following theorem in order to develop an algorithm for constructing $T(\mathbf{A})$ without knowing $G(\mathbf{L})$ and $G(\mathbf{U})$.

Theorem 1.1 (see Theorem 3.3 of [5]) *Vertex k is the parent of vertex i in the elimination tree $T(\mathbf{A})$ if and only if k is the first vertex after i such that k and i belong to the same strong component of the subgraph $G_k(\mathbf{A})$ of $G(\mathbf{A})$.*

Given this theorem, Eisenstat and Liu [6] propose the following algorithm to compute the parent pointers (1) and construct the elimination tree.

Algorithm 1 eTree

```
1: for vertex k = 1 to n do
2:    Find the component C of G_k(A) that contains k
3:    for each vertex i ∈ C \ {k} do
4:       if Parent(i) = ∞ then
5:          Parent(i) = k
6:    Parent(k) = ∞
```

The strong components of a digraph with m edges and n vertices can be found in $\mathcal{O}(n + m)$ time [11]. Hence, the worst case time complexity of the algorithm eTree is $\mathcal{O}(mn)$ for a graph containing n vertices and m edges. Eisenstat and Liu reduce the practical running time of the algorithm by using quotient graphs [6] and specialized algorithms to find the strong component containing a given vertex, but report that the worst case time complexity of the improved algorithm is still $\mathcal{O}(mn)$.

2 An $\mathcal{O}(m \log n)$ time algorithm

The essence of the proposed algorithm lies in the following theorem by Eisenstat and Liu [6, Theorem 3.2].

Theorem 2.1 (see [6, Theorem 3.2]) *Let $C = (V_c, E_c)$ be a strong component of $G_k(\mathbf{A})$ and let i_t be the highest numbered vertex in V_c. Then the vertex set V_c corresponds to the vertex set of the subtree of the elimination tree $T(\mathbf{A})$ rooted at i_t.*

Eisenstat and Liu make use of this theorem in a bottom-up approach in the improved version of Algorithm 1. Each time k is incremented, the newly formed connected component (a union of a set of strong components) is detected using quotient graphs and the vertex k is set to be the root of the new component whose children are the roots of the strong components that form the component of k. The algorithm that we propose makes use of Theorem 2.1 in a top-down approach. Assume we have detected the strong components of G_k. Then, we

can build the subtrees of $T(\mathbf{A})$ rooted at the highest numbered vertex in each component recursively. Furthermore, we can build the quotient graph induced by the partition formed by the components of G_k and the vertices of G that are not in G_k (each as a single vertex) and compute the elimination tree associated with this quotient graph recursively. Once this is done, the whole tree $T(\mathbf{A})$ can be obtained by replacing each super-vertex with the associated subtree. The algorithm that we propose below implements this recursion. The algorithm is based on an algorithm of Tarjan [12].

2.1 The algorithm

Let $\mathbf{A}$ be an irreducible, unsymmetric, $n \times n$ matrix with a nonzero diagonal. Algorithm 2 displays the proposed algorithm `UET` which finds the elimination tree $T(\mathbf{A})$. There are two inputs for the algorithm: a strongly connected digraph G with η vertices and an integer $s < \eta$ such that G_s is acyclic. That is, all the strong components of G_s are trivial. The initial call is $\texttt{UET}(G(\mathbf{A}) = (\{1, 2, \ldots, n\}, E), 0)$. We assume that the parent pointers are initialized to ∞ before the execution of the algorithm.

Algorithm 2 $r = \texttt{UET}(G = (\{i_1, i_2, \ldots, i_\eta\}, E), s)$

Input A graph $G = (\{i_1, i_2, \ldots, i_\eta\}, E)$ and a nonnegative integer $s < \eta$ such that

(i) G_s is acyclic,

(ii) the remaining vertices satisfy $i_{s+1} < i_{s+2} < \cdots < i_\eta$ whenever $s < \eta - 1$.

Output the root r (an index from 1 to n corresponding to a vertex of the initial call) of the elimination tree of G is returned and PARENT pointers are set.

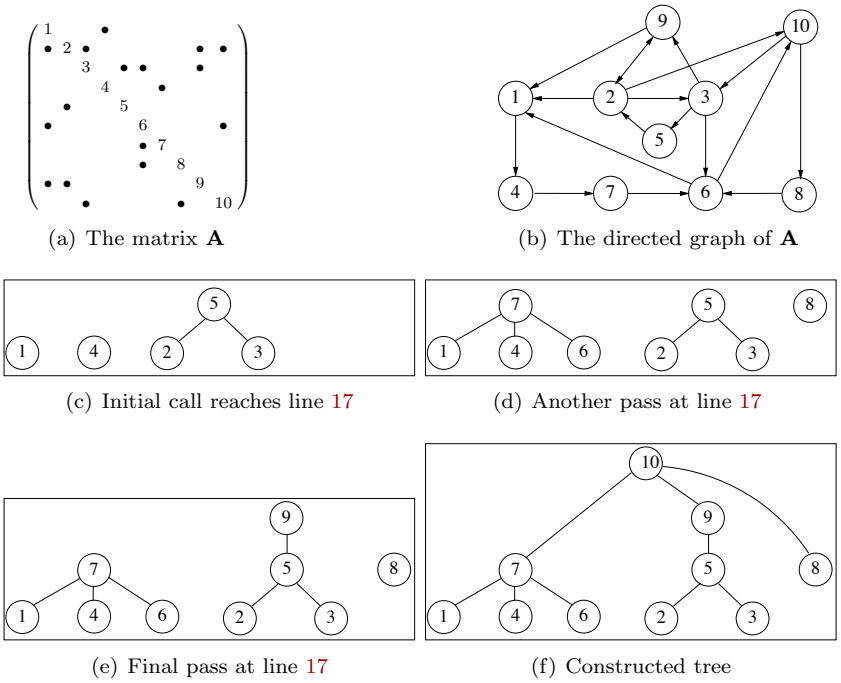
```
1:  if s = η − 1 then
2:    for j = 1 to η − 1 do
3:      PARENT(i_j) = i_η
4:    return i_η        ▶ i_η is the root
5:  else
6:    k = ⌈(s + η)/2⌉
7:    Let p be the number of strong components of G_k
8:    for ℓ = 1 to p do
9:      Let C_ℓ = (V_ℓ, E_ℓ) be the ℓth strong component of G_k
10:     if |V_ℓ| > 1 then
11:       s_ℓ = |{i_j : i_j ∈ V_ℓ, j ≤ s}|     ▶ the first s vertices form a dag
12:       r_ℓ ← UET(C_ℓ, s_ℓ)                  ▶ find the root of each str. comp.
13:     else
14:       r_ℓ ← the vertex in V_ℓ
15:   Let Π = {V_1, ..., V_p, i_{k+1}, i_{k+2}, ..., i_η} be a partition of V
16:   Let G^Π = (V^Π = {r_1, ..., r_p, i_{k+1}, i_{k+2}, ..., i_η}, E^Π) be the quotient
      graph induced by Π where V_ℓ is represented by r_ℓ for ℓ = 1, ..., p
17:   return UET(G^Π = (V^Π, E^Π), p)     ▶ G^Π_p is acyclic
```

Since G is strongly connected and G_s is known to be acyclic, if $s = \eta - 1$, the vertex i_η must be the vertex that makes G strongly connected. If this is the case, we set the parent pointers appropriately; vertex i_η become the parent of all vertices $i_1, \ldots, i_{\eta-1}$. If this is not the case, the algorithm examines G_k where $k = \lceil (s+\eta)/2 \rceil$. Assume that G_k has p strong components $C_\ell = (V_\ell, E_\ell)$ for $\ell = 1, \ldots, p$. By Theorem 2.1, the vertex set of each strong component V_ℓ corresponds to the vertices of the subtree of the elimination tree rooted at the highest numbered vertex in V_ℓ. In this case, each recursive call at line 12 is responsible for setting the parent pointers of the vertices in these subtrees and returns the root of each such subtree (essentially the highest numbered vertex in the associated component). When we have computed the subtrees associated with each strong component of G, `UET` is called on the quotient graph with partition $\Pi = \{V_1, \ldots, V_p, i_{k+1}, i_{k+2}, \ldots, i_\eta\}$, where $V_1, \ldots, V_p$ correspond to the p strong components of G_k, to set the parents of the roots of these subtrees. As we need a vertex to represent V_ℓ for $\ell = 1, \ldots, p$, we find it convenient to return the root of the subtree corresponding to a strong component (which is the highest numbered vertex in that component).

In a recursive call with a strong component C_ℓ, s_ℓ is equal to the number of vertices in V_ℓ at an index smaller than or equal to s. Hence, the subgraph of C_ℓ containing the first s_ℓ vertices is acyclic (has only trivial strong components). On the other hand, for the graph G^Π, we know that the first p vertices form a directed acyclic graph since they correspond to the strong components of G_k. These two observations show that at an invocation of `UET` each vertex $i_j \in V$ such that $j \leq s$ represents a subtree in $T(\mathbf{A})$ with root i_j (may also be a leaf node).

We now analyze the worst case running time complexity of Algorithm 2. Consider the call tree for `UET` where the root node is the initial call and the children of a node are the recursive calls it makes. The complexity of the body of the algorithm without the recursive calls is $\mathcal{O}(|E|)$, as the running time of the body is dominated by the computation of the strong components (in $\mathcal{O}(|E|)$ time) of a graph and the construction of the associated quotient graph (which is done in $\mathcal{O}(|E|)$ time as we also discuss in the next subsection). That is, at each node of the call tree, the complexity is $\mathcal{O}(|E|)$. We now proceed in two steps to bound the running time. In the first step, we obtain the bound $\mathcal{O}(m)$ on the sum of the nonrecursive work of all nodes of the call tree at a given depth. In the second step, we obtain the bound $\mathcal{O}(\log n)$ on the height of the call tree.

Consider the algorithm `UET` making the recursive calls for a given graph with $|E|$ edges and p strong components. Each edge in E is included either in one recursive call at line 12, or in the call at line 17 on the quotient graph, or discarded while building the quotient graph. This implies that $\sum_{\ell=1}^{p} |E_\ell| + |E^\Pi| \leq |E|$. Therefore the total number of edges in all calls at a given depth is no larger than the number of edges m in the initial call, and hence the complexity of nonrecursive work at a given depth is $\mathcal{O}(m)$. Let $\eta - s$ be the *rank* of the problem, i.e., the number of vertices to be examined for strong connectivity in G. Among the vertices of C_ℓ, there are s_ℓ of them in the acyclic graph G_s. The remaining vertices all should be between s and k. Therefore, for a strong component C_ℓ, we have $|V_\ell| - s_\ell \leq k - s$. The problem containing the quotient graph has rank $\eta - k$, as among a total of $\eta - k + p$ vertices, the first p of them form a dag. By the definition $k = \lceil (n+s)/2 \rceil$, we have $\max\{k-s, \eta-k\} \leq 2(\eta-k)/3$; notice that the inequality is tight only when $n - s = 3$. Therefore, the rank of a



(a) The matrix **A**

(b) The directed graph of **A**

(c) Initial call reaches line 17

(d) Another pass at line 17

(e) Final pass at line 17

(f) Constructed tree

Figure 1: Tracing the algorithm on the sample matrix.

node of the call tree is at most 2/3 as large as its parent. Since the rank of the initial problem is n, the height of the call tree is $\mathcal{O}(\log n)$, and the worst case time complexity of the algorithm is $\mathcal{O}(m \log n)$.

Consider the sample matrix from [6] shown in Fig. 1(a) whose graph is shown in Fig. 1(b). The first call with the whole graph finds $k = 5$, $p = 3$ with $V_1 = \{1\}$, $V_2 = \{4\}$, $V_3 = \{2, 3, 5\}$, and then recursively calls the algorithm in V_3; then the algorithm reaches to line 17, yielding the subtree shown in Fig. 1(c) with $V^{\Pi} = \{r_1 = 1; r_2 = 4; r_3 = 5; 6; 7; 8; 9; 10\}$. The recursive call is made with $s = 3$ and then finds $k = 6$ (pointing to vertex 8). Three strong components $V_1 = \{1, 4, 6, 7\}$, $V_2 = \{2, 3, 5\}$, and $V_3 = \{8\}$ are found; 7 becomes the root of V_1, and 5 becomes the root of V_1 after the recursive calls at line 12, yielding the subtrees shown in Fig. 1(d). Then the algorithm reaches again to line 17 with $V^{\Pi} = \{r_1 = 7; r_2 = 5; r_3 = 8; 9; 10\}$ and $s = 3$. This time $k = 4$ (points at vertex 9). Again, three strong components are found and 9 becomes the root of the component containing 5, yielding the tree shown in Fig. 1(e). The last call results in the elimination tree shown in Fig. 1(f).

2.2 Implementation details

Our implementation allocates $10n + m$ space for a strongly connected graph before calling the recursive function (this does not include the parent pointers). If the graph is not strongly connected, but has p strong components, then another $2p$ space is allocated. The recursive function itself allocates $2n + p$ extra space before the recursive calls on the p strong components (at line 12) and frees up

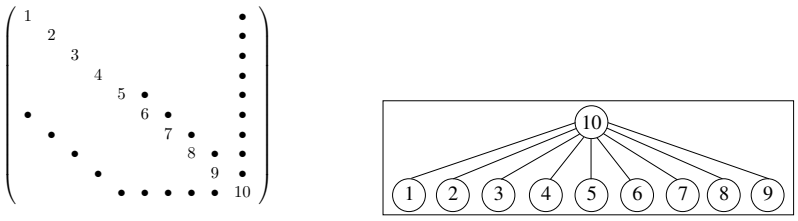
this space just before the recursive call on G^{Π}. By following an analysis similar to the one in the previous subsection, we can see that the recursive calls allocate and free a total of $\mathcal{O}(n \log n)$ space throughout the whole process. Notice that this is not the peak memory requirement at a given time. In comparison, `EL` uses $9n + 2m$ space (this does not include the parent pointers).

The most involved part of the proposed `UET` algorithm is the construction of the subgraphs C_ℓ and the graph quotient G^{Π} while discarding the duplicate edges. In short, we do this operation as follows: we visit the edges of G once and construct the graphs of C_ℓ for $\ell = 1, \ldots, p$ and the graph G^{Π} during this visit. We implemented this operation in two different for-loops. In the first for-loop, we visit the vertices that are not in G_k to add some edges to G^{Π}. For each such vertex v, we visit its neighbors in order. If a neighbor u of v is not in G_k, then the edge is copied into G^{Π}. Otherwise, we put an edge from v to the vertex representing u's strong component, if such an edge is not put before (an array of markers is used to do this without searching). In the second for-loop, we visit the vertices of a strong component before visiting the vertices of another one. When we visit a vertex v in a strong component C_ℓ, the neighbors of v that are in C_ℓ are copied into the graph of C_ℓ. For a neighbor u of v which is either in another strong component C_q or not in G_k , we add an edge from the vertex representing C_ℓ to the vertex representing C_q (or to u itself if u is not in G_k) if such an edge was not added before (we again use an array of markers to avoid searches). Note that the vertices representing the strong components are the roots, and their ids along with the ids of vertices not in G_k are passed to the recursive call on G^{Π} to set their parent pointers.

One further detail we need to mention is that the body of the **else** statement in at line 5 handles the case where $k = p$ differently. If we have $k = p$, then G_k is acyclic and the strong components of the graph G_k are all trivial. Therefore, instead of doing recursive calls on the components of G_k, the variable s is modified and another k between the new s and η is searched in a while loop, where at each iteration the strong components of G_k are found. If during this search we find a G_k with a non-trivial strong component, then the algorithm continues with the body of the **else** statement as displayed in Algorithm 2. If s is found to be $\eta - 1$, then the parent pointers of all vertices are set to the last vertex of G and the control returns to the caller.

3 Experimental results

We have implemented the proposed elimination tree construction algorithm in C and compared it with a Fortran implementation of the algorithm by Eisenstat and Liu [6]. In the sequel, `UET` refers to the proposed algorithm with the modification discussed in the last paragraph of Section 2.2, `UET'` is the proposed algorithm without the modification, and `EL` refers to the algorithm by Eisenstat and Liu. We compiled the C and Fortran codes with `mex` of Matlab which uses `gcc` and `gfortran` versions `4.4.2` with the optimization flag `-O`. We run the compiled codes on a machine with a 2.4Ghz AMD Opteron 250 processor and 8Gbytes of RAM.



(a) Sample 10×10 matrix (b) Elimination tree

Figure 2: A matrix on which EL runs in quadratic time and the corresponding elimination tree.

3.1 Constructed examples

There are matrices on which the algorithm EL runs in time $\mathcal{O}(mn)$. For a given positive integer k, let A be a matrix of size $n = 2k$ whose nonzeros are at the following positions:

$$\{a_{ii}|i = 1, \ldots, n\} \cup \{a_{i,i+1}|i = k, \ldots, n-1\} \cup \{a_{i,i-k}|i = k+1, \ldots, n\} \\ \cup \{a_{in}|i = 1, \ldots, n\} \cup \{a_{ni}|i = k+1, \ldots, n-1\} \,. \quad (2)$$

A sample matrix of size 10×10 (with $k = 5$) from this family is shown in Fig. 2(a).

The matrices as described above are irreducible and have zero fill-in when the natural order is used during the elimination process. For these matrices, using the quotient graphs does not have any effect (each component is of size one until the nth vertex) and therefore EL runs in $\mathcal{O}(mn) = \mathcal{O}(n^2)$ time. The proposed UET' algorithm starts with the initial call with $s = 0$ and all of the 10 vertices. Then, the graph G_5 is found to be acyclic, and a recursive call is made with $s = 5$ and again with 10 vertices. The graph G_8 is found to be acyclic, and a recursive call is made with $s = 8$ and again with 10 vertices. The graph G_9 is found to be acyclic, and a recursive call is made with $s = 9$ and all the 10 vertices. This last call founds $s = 9$ and $\eta = 10$ (the condition holds at the first **if** statement) and sets all parent pointers to 10 resulting in the elimination tree shown in Fig 2(b). Note that UET sets the parent pointers without any recursive call (but computes the strong components of G_5, G_8 and G_9).

In order to assess the difference in the execution times of both algorithms, we run them on matrices described above with parameter $k = 5 \times 10^4, 10 \times 10^4$, and 15×10^4. The execution times are reported in seconds in Table 1. In order to put the running times into perspective, we also report the running time of the standard elimination tree algorithm (`etree` in Matlab) in Table 1, where `etree` was run on the symmetrized matrices.

As seen in the table, EL has a much higher execution time than UET' and UET. The increases in the running time of the algorithms from one row to the other follow the increases in $m \times n$ and $m \log n$, although not very closely. For example, for a 10-folds increase in $m \times n$ from the first row to the third one, the execution time of EL increases by about 9, and for a 3.29-folds increase in $m \log n$ from the first row to the third one, the execution time of UET' and UET increase, respectively, by 3.47 and 3.50. The algorithms UET' and UET are much slower

n	m	`etree`	`EL`	`UET'`	`UET`
100000	349997	0.005	27.648	0.211	0.068
200000	699997	0.009	124.087	0.459	0.148
300000	1049997	0.014	276.854	0.732	0.238

Table 1: Running time of the algorithms on instances on which `EL` runs in quadratic time.

	`dmperm`	`MC64`
AMD	85	88
MeTiS	90	90

Table 2: Number of instances on which `EL` had a smaller running time than `UET`. Each cell can be at most 102.

than `etree` as expected. The algorithm `etree` runs linearly and for a 3-folds increase in m from the first row to the third one, its execution time increases by only 2.8.

3.2 Real life matrices

We have compared the execution times of the algorithms `EL` and `UET` on a set of matrices from the UFL sparse matrix collection (`http://www.cise.ufl.edu/research/sparse/matrices/`). The matrices are listed with the properties: square, $50000 < n \leq 5000000$, numerical symmetry is less than 1.0, $2 \times n \leq m \leq 15000000$, real, and not of type `Graph`. There were a total of 102 matrices having these properties at the time of experimentation.

We preprocessed these matrices as follows. First, we ensured that the diagonal is zero-free. We have two alternatives for this purpose. The first one is to use `MC64` [2] to find a maximum product matching and to permute the entries in the matching to the diagonal. The second one is to use an arbitrary zero-free diagonal using a maximum cardinality matching algorithm (such as those resulting from `dmperm` of Matlab). Next, we obtained the Dulmage-Mendelsohn decomposition [4] and permute the matrix accordingly (using the output of `dmperm`), and deleted all entries that are in the off-diagonal blocks. The reason in deleting those entries is that it is advisable to factor only the diagonal blocks of a reducible matrix. As the last step, we used AMD [1] (the command `amd` in Matlab) and MeTiS [8] (using MeshPart toolbox [7]) to reorder the matrices using their symmetrized versions. We use the term "instance"to refer to a matrix preprocessed in one of the four possible alternative ways. Therefore, we have $4 \times 102 = 408$ instances. We run `EL` and `UET` ten times for each instance resulting from this preprocessing step in order to be able to report reproducible running time results. We did not use randomization for AMD in Matlab, and we did not turn the randomization off in MeTiS. As we expect the same running time when AMD was used in ordering, we report the minimum of ten running time for each algorithm. On the other hand, we report the median of ten running time for each algorithm when MeTiS was used, since the expected behavior in this case is the median one.

	dmperm diagonal				MC64 diagonal			
	AMD		MeTiS		AMD		MeTiS	
matrix	EL	UET	EL	UET	EL	UET	EL	UET
pre2	0.393	0.998	0.370	1.247	0.481	0.997	0.314	1.093
Hamrle3	11.218	2.262	6.195	2.726	17.418	2.487	12.842	2.633
rajat30	0.361	0.977	0.361	1.063	0.290	0.988	0.300	1.028
largebasis	0.215	1.064	0.230	1.149	0.195	1.086	0.235	1.120
t2em	0.243	1.230	0.287	1.525	0.243	1.231	0.287	1.572
tmt_unsym	0.255	1.273	0.290	1.510	0.255	1.274	0.288	1.511
ch7-8-b5	81.328	0.369	2.572	0.350	89.785	0.397	3.352	0.357
m133-b3	143.754	0.550	17.777	0.473	286.641	0.581	11.823	0.481
shar_te2-b3	134.641	0.541	15.703	0.470	299.741	0.553	4.286	0.459
atmosmodd	0.471	2.227	0.544	3.211	0.470	2.212	0.544	3.249
atmosmodj	0.471	2.248	0.540	3.195	0.504	2.242	0.543	3.190
atmosmodl	0.553	2.726	0.634	3.791	0.551	2.684	0.636	3.863
atmosmodm	0.555	2.718	0.635	3.796	0.551	2.668	0.636	3.811
memchip	0.827	4.082	0.844	4.850	0.830	4.086	0.843	4.855
circuit5M_dc	0.976	5.233	0.994	5.496	0.973	5.235	0.986	5.429

Table 3: Running time of EL and UET (in seconds) on real-life matrices where at least one of the algorithms run in more than one second.

We first give the number of instances on which EL had a smaller running time than UET in Table 2. As we have 102 matrices in total, the table shows that EL's practical running time is faster than that of UET in a significant number of cases. We highlight that both of the algorithms are efficient; in only 15 matrices one of the algorithms had a running time larger than one second with any of the four preprocessing alternatives. In Table 3, we present the matrices on which this latter case occurred. In this table, in only 16 out of 60 instances, running time of EL is larger than that of UET. These instances correspond to the matrices Hamrle3, ch7-8-b5, m133-b3, and shar_te2-b3. As seen in these instances, the difference in the running time can be significant in favor of UET. We note that the three matrices for which there are two orders of difference in the running times are from the same family in the UFL collection. Combined with the previous subsection results, we conclude that there are matrices for which the running time of EL can be significantly larger than that of UET.

4 Conclusion

We have proposed an algorithm to compute the elimination tree of an $n \times n$ unsymmetric matrix with m nonzeros in $\mathcal{O}(m \log n)$ time. The previously known algorithm [6] is of the worst case time complexity $\mathcal{O}(mn)$. The proposed algorithm is based on an algorithm of Tarjan [12]. Tarjan's algorithm constructs a tree whose leaves correspond to the vertices of a given graph. Whereas Tarjan's algorithm applies the recursion to the edge set, the proposed algorithm applies the recursion to the vertex set.

We have implemented the proposed algorithm and compared it with the algorithm of Eisenstat and Liu [6]. Both algorithms are found to be fast in general. On 87 among 102 real life matrices satisfying the properties $50000 <$

$n \leq 5000000$ and $2 \times n \leq m \leq 15000000$, both of the algorithms run in less than one second on current desktop computers. The algorithm by Eisenstat and Liu is found, in a significantly large number of cases, to be faster than the proposed one. However, there are both real life and constructed matrices where the proposed algorithm is significantly faster.

Acknowledgments

We thank an anonymous referee whose suggestions improved the paper. We also thank Stanley Eisenstat for his editorial and technical comments, for sharing the code for their algorithm, for a simplified analysis of the running time complexity, and for the worst case examples presented in the experiments. We are appreciative of his help in designing the experimental set up as well. The proposed algorithm is available at authors' web pages.

References

[1] Patrick R. Amestoy, Timothy A. Davis, and Iain S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, 1996.

[2] Iain S. Duff and Jacko Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications*, 22:973–996, 2001.

[3] Iain S. Duff and Bora Uçar. Combinatorial problems in solving linear systems. Technical Report TR/PA/09/60, CERFACS, Toulouse, France, 2009.

[4] A. L. Dulmage and N. S. Mendelsohn. Coverings of bipartite graphs. *Canadian Journal of Mathematics*, 10:517–534, 1958.

[5] Stanley C. Eisenstat and Joseph W. H. Liu. The theory of elimination trees for sparse unsymmetric matrices. *SIAM J. Matrix Anal. Appl.*, 26:686–705, January 2005.

[6] Stanley C. Eisenstat and Joseph W. H. Liu. Algorithmic aspects of elimination trees for sparse unsymmetric matrices. *SIAM J. Matrix Anal. Appl.*, 29:1363–1381, January 2008.

[7] John R. Gilbert, Gary L. Miller, and Shang-Hua Teng. Geometric mesh partitioning: Implementation and experiments. *SIAM Journal on Scientific Computing*, 19(6):2091–2110, 1998.

[8] G. Karypis and V. Kumar. *MeTiS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 4.0*. University of Minnesota, Department of Comp. Sci. and Eng., Army HPC Research Center, Minneapolis, 1998.

[9] Joseph W. H. Liu. The role of elimination trees in sparse factorization. *SIAM J. Matrix Anal. Appl.*, 11(1):134–172, 1990.

[10] Rob Schreiber. A new implementation of sparse Gaussian elimination. *ACM T. Math. Software*, 8(3):256–276, 1982.

[11] Robert E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[12] Robert E. Tarjan. An improved algorithm for hierarchical clustering using strong components. *Inform. Process. Lett.*, 17(1):37–41, 1983.

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399