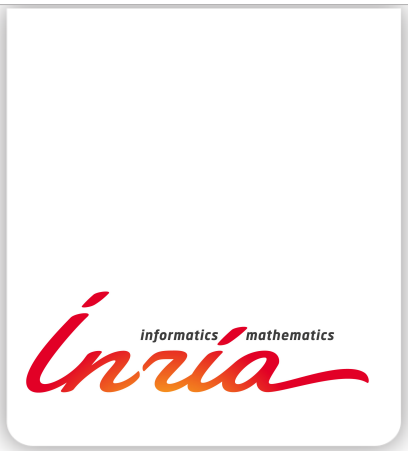
Constructing elimination trees for sparse unsymmetric matrices

Kamer Kaya, Bora Uçar

RESEARCH REPORT
N° 7549
February 2011
Project-Team ROMA

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



Constructing elimination trees for sparse unsymmetric matrices

Kamer Kaya, Bora Uçar

Project-Team ROMA

Research Report n° 7549 — February 2011 — 13 pages

Abstract: [Eisenstat and Liu, SIAM J. Matrix Anal. Appl., 26 (2005) and 29 (2008)]. The construction algorithm has a worst-case time complexity of $\Theta(mn)$ for an $n \times n$ unsymmetric matrix having m off-diagonal nonzeros. We propose another algorithm that has a worst-case time complexity of $\mathcal{O}(m \log n)$. We compare the two algorithms experimentally and show that both algorithms are efficient in general. The algorithm of Eisenstat and Liu is faster in many practical cases, yet there are instances in which there is a significant difference between the running time of the two algorithms in favor of the proposed one.

Key-words: Elimination tree, sparse matrix factorization

Revised: October 2012

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

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 hors diagonaux [Eisenstat and Liu, SIAM J. Matrix Anal. Appl., 26 (2005) and 29 (2008)]. Nous décrivons un algorithme dont la complexité en temps est $\mathcal{O}(m \log n)$. Nous comparons les deux algorithmes expérimentalement et montrent que les deux sont efficaces en général. L'algorithme de Eisenstat et Liu est plus rapide dans de nombreux cas pratiques, mais il y a des cas dans lesquels la différence entre le temps d'exécution de deux algorithmes est significative en faveur de le nôtre.

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 [10]. The elimination tree for symmetric (positive definite) systems dates from 80s [11] and even before (see [4, Section 3.3] and [10]), whereas the elimination tree for unsymmetric matrices is a recent development. Eisenstat and Liu [6] define the elimination tree for unsymmetric matrices and discuss its properties. In a follow up paper [7], they describe algorithms to construct those trees and discuss the use of the elimination trees to improve the performance of a symbolic factorization algorithm. The algorithm of Eisenstat and Liu [7] for constructing the elimination tree of an $n \times n$ unsymmetric matrix with m off-diagonal nonzeros has a worst-case time complexity of $\Theta(mn)$. We present an $\mathcal{O}(m \log n)$ time algorithm for the same purpose.

A directed graph (digraph) $G = (V, E)$ consists of a finite set of vertices V and a set of edges E, where each edge $(u, v) \in E$ is an ordered pair in V. The directed graph $G(\mathbf{A}) = (V, E)$ of an $n \times n$ matrix $\mathbf{A}$ with m off-diagonal nonzeros contains n vertices, $V = \{v_1, v_2, \ldots, v_n\}$, and m edges, $E = \{(v_i, v_j) : a_{ij} \neq 0, \text{ for } 1 \leq i \neq j \leq n\}$. We assume that $\mathbf{A}$ has a zero-free diagonal. However, as the definition of $G(\mathbf{A})$ shows, we will omit the self loops implied by the diagonal entries. We also assume that $m \geq n$ to simplify some definitions and complexity analysis.

A *path* of length k in a directed graph $G = (V, E)$ is a sequence of vertices $u_0, u_1, \ldots, u_k$ such that $(u_i, u_{i+1}) \in E$ for $i = 0, \ldots, k-1$. For two vertices $u, v \in V$, we say that u is *connected* to v in G, denoted $u \xrightarrow{G} v$, if there is path $u = u_0, u_1, \ldots, u_k = v$ from u to v. A cycle is a path which starts and ends at the same vertex. A simple cycle is a path $u_0, u_1, \ldots, u_k, u_0$ where $k \geq 1$ and $u_i \neq u_j$ 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,

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

If $u \xrightarrow{G} v$ 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. A directed graph without simple cycles is called a *directed acyclic graph* (dag). The strong components of a dag are its vertices, i.e., each vertex forms a *trivial* strong component.

We say that a matrix $\mathbf{A}$ with a zero-free diagonal is *irreducible* if $G(\mathbf{A})$ is strongly connected, i.e., it only has one strong component.

Let $\Pi = \{V_1, V_2, \ldots, V_\ell\}$ be a partition of the vertex set V of a directed graph $G = (V, E)$. 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 trian-

gular $\mathbf{L}$ and upper triangular $\mathbf{U}$ exists. Then, the *elimination tree* $T(\mathbf{A})$ of an unsymmetric $\mathbf{A}$ is defined by Eisenstat and Liu [6, 7] 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 (see Theorem 3.3 of [6])**.** *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 [7] propose Algorithm 1 to compute the parent pointers (1) and hence to construct the elimination tree.

```
1: for vertex k = 1 to n do
2:    Find the strong 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) = ∞
```

Algorithm 1: eTree

The strong components of a digraph with m edges and n vertices can be found in $\Theta(n + m)$ time [12]. Hence, the worst-case time complexity of the algorithm eTree is $\Theta(mn)$ for a graph containing n vertices and m edges. Eisenstat and Liu reduce the practical running time of the algorithm by using quotient graphs [7] 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 $\Theta(mn)$.

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

The essence of the proposed algorithm lies in the following theorem of Eisenstat and Liu [7, Theorem 3.2].

Theorem 2. (see [7, 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 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 proposed algorithm is based on an algorithm of Tarjan [13]. Tarjan's original algorithm constructs a tree whose leaves correspond to the vertices of a given graph and applies a recursion to the edge set, whereas the proposed elimination tree algorithm constructs a tree whose vertices correspond to the vertices of the given graph by applying a recursion to the vertex set.

2.1 The algorithm

Let $\mathbf{A}$ be an irreducible, unsymmetric, $n \times n$ matrix. 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.

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 becomes 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, 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 11 is responsible for setting the parent pointers of the vertices in these subtrees where the root of each subtree is 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 use the root, i.e., the highest numbered vertex in V_ℓ.

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).

Consider the sample matrix from [7] 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 line 14, yielding the subtree shown in Fig. 1(c)

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

(i) G is strongly connected,

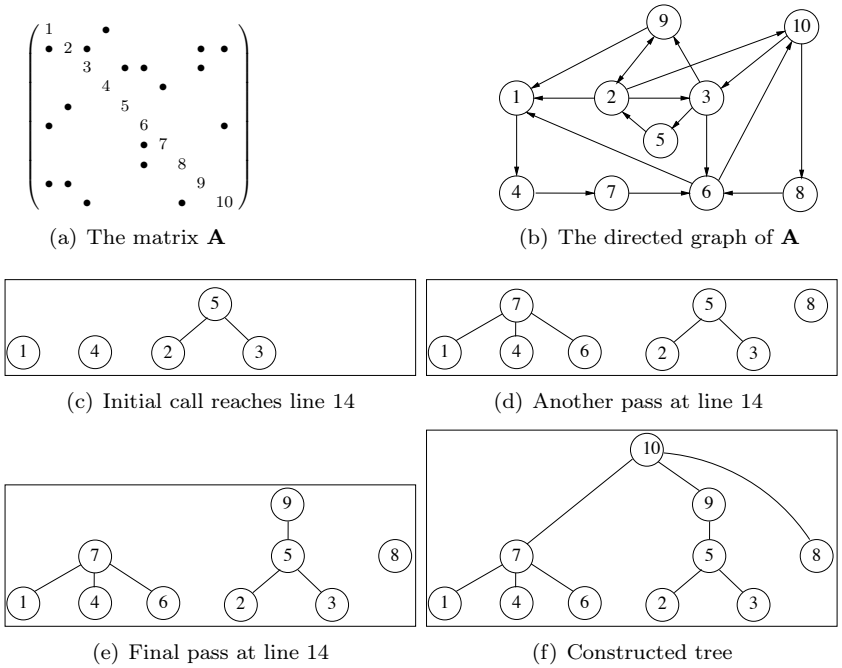
(ii) G_s is acyclic,

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

Output PARENT pointers are set.

1: **if** $s = \eta - 1$ **then**
2: **for** $j = 1$ **to** $\eta - 1$ **do**
3: PARENT$(i_j) = i_\eta$
4: **else**
5: $k = \lceil (s + \eta)/2 \rceil$
6: Let p be the number of strong components of G_k
7: **for** $\ell = 1$ **to** p **do**
8: Let $C_\ell = (V_\ell, E_\ell)$ be the ℓth strong component of G_k
9: **if** $|V_\ell| > 1$ **then**
10: $s_\ell = |\{i_j : i_j \in V_\ell, j \leq s\}|$ ▶ the first s vertices form a dag
11: UET(C_ℓ, s_ℓ) ▶ Set parents for the vertices in each str. comp.
12: Let $\Pi = \{V_1, \ldots, V_p, i_{k+1}, i_{k+2}, \ldots, i_\eta\}$ be a partition of V
13: Let $G^\Pi = (V^\Pi = \{r_1, \ldots, r_p, i_{k+1}, i_{k+2}, \ldots, i_\eta\}, E^\Pi)$ be the quotient graph induced by Π where V_ℓ is represented by the highest numbered vertex $r_\ell \in V_\ell$ for $\ell = 1, \ldots, p$
14: UET$(G^\Pi = (V^\Pi, E^\Pi), p)$ ▶ G^Π is strongly connected, G^Π_p is acyclic

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



(a) The matrix **A**

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

(c) Initial call reaches line 14

(d) Another pass at line 14

(e) Final pass at line 14

(f) Constructed tree

Figure 1: Tracing the algorithm on the sample matrix.

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 11, yielding the subtrees shown in Fig. 1(d). Then the algorithm reaches again to line 14 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 Complexity analysis

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 for an $n \times n$ matrix with m off-diagonal nonzero etnries. 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 11, or in the call at line 14 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 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 $\mathcal{O}(m \log n)$.

2.3 Implementation details

Our implementation of `UET` allocates $10n + m$ space for a strongly connected graph before calling the recursive function (this does not include the parent pointers), and overwrites the input matrix, essentially requiring $11n+2m$ space. The recursive function itself allocates $2n + p$ extra space before the recursive calls on the p strong components (at line 11) 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 was 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.

The proposed algorithm requires an irreducible matrix $\mathbf{A}$ (or a strongly con-

nected graph) at the initial call. One can therefore detect the irreducible blocks of a general **A** and call the proposed algorithm to construct the elimination tree of each block separately and build the elimination tree (forest in this case) of the initial matrix **A**. Alternatively, one can append a zero-free row and column to any input matrix, and create an irreducible matrix, say **B**. The new row/column will be the root of the elimination tree of **B**. By removing the newly added vertex from the elimination tree of **B**, one can obtain the elimination tree of **A**. We found it convenient to not to add those edges, but assume their presence and handle them by the if check at line 1 of Algorithm 2.

One further detail we need to mention is that the body of the **else** statement at line 4 handles the case where $p = k$ differently. If we have $p = k$, 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. This technique does not affect the theoretical complexity of the algorithm, as it is equivalent to calling the algorithm recursively with $s = k$.

3 Experimental results

We have implemented the proposed elimination tree construction algorithm in C and compared it with a Fortran implementation of the algorithm of Eisenstat and Liu [7]. In the sequel, `UET` refers to the proposed algorithm with the modification discussed in the last paragraph of Section 2.3, and `EL` refers to the algorithm of 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.

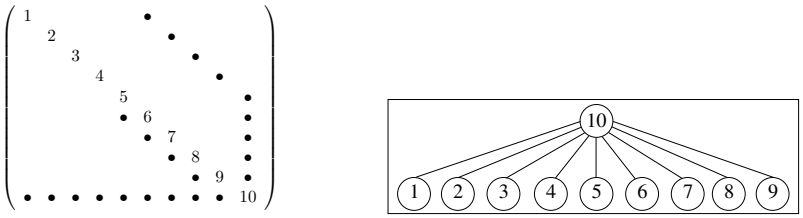
3.1 Constructed examples

There are matrices on which the algorithm `EL` runs in time $\Theta(mn)$. For a given positive integer k, let A be a matrix of size $n = 2k$ whose nonzero set is the following:

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

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 $\Theta(mn) = \Theta(n^2)$ time. The proposed algorithm, without the modification presented at the last paragraph



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

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

n	m	`etree`	`EL`	`UET`
100000	249997	0.005	27.509	0.073
200000	499997	0.010	125.064	0.147
300000	749997	0.014	269.573	0.223

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

of Section 2.3, 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 finds $s = 9$ 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`, the implementation with the modification presented at the last paragraph of Section 2.3, 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 symmetric 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` on these instances. As expected, both algorithms are much slower than the linear time `etree` algorithm.

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 [2]. 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

	`dmperm`	`MC64`
AMD	84	84
MeTiS	89	90

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

	`dmperm` diagonal				`MC64` diagonal			
	AMD		MeTiS		AMD		MeTiS	
matrix	`EL`	`UET`	`EL`	`UET`	`EL`	`UET`	`EL`	`UET`
pre2	0.387	0.825	0.333	1.038	0.478	0.869	0.320	1.000
Hamrle3	11.239	2.017	6.358	2.377	17.493	2.240	13.016	2.439
largebasis	0.212	0.991	0.222	1.051	0.191	1.010	0.237	1.047
t2em	0.232	1.094	0.249	1.406	0.232	1.090	0.251	1.396
tmt_unsym	0.244	1.130	0.255	1.383	0.244	1.139	0.251	1.398
ch7-8-b5	80.421	0.297	3.173	0.238	90.689	0.310	2.852	0.247
m133-b3	140.486	0.459	7.333	0.334	285.167	0.459	11.183	0.392
shar_te2-b3	134.650	0.450	19.268	0.391	297.392	0.462	4.953	0.324
atmosmodd	0.467	1.827	0.534	2.574	0.465	1.825	0.537	2.604
atmosmodj	0.465	1.839	0.534	2.593	0.464	1.832	0.534	2.590
atmosmodl	0.543	2.230	0.623	3.129	0.543	2.220	0.622	3.124
atmosmodm	0.543	2.237	0.622	3.062	0.543	2.255	0.623	3.157
memchip	0.804	3.627	0.818	4.198	0.805	3.559	0.814	4.241
circuit5M_dc	0.968	4.678	0.979	4.949	0.959	4.627	0.978	4.836

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.

use `MC64` [3] 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 [5] and permuted 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 [9] (using MeshPart toolbox [8]) 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 times for each algorithm. On the other hand, we report the median of ten running times for each algorithm when MeTiS was used, since the expected behavior in this case is the median one.

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 14 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 (in increasing order of ids at the UFL collection) on which this latter case occurred. In this table, in only 16 out of 56 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 a zero-free diagonal and m off-diagonal nonzeros in $\mathcal{O}(m \log n)$ time. The previously known algorithm [7] has the worst-case time complexity of $\Theta(mn)$.

We have implemented the proposed algorithm and compared it with the algorithm of Eisenstat and Liu [7]. Both algorithms are found to be fast in general. On 88 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 of 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 of the algorithm `EL`, 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.

Bibliography

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

[2] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, 2011.

[3] I. S. Duff and J. 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.

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

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

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

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

[8] J. R. Gilbert, G. L. Miller, and S.-H. Teng. Geometric mesh partitioning: Implementation and experiments. *SIAM Journal on Scientific Computing*, 19(6):2091–2110, 1998.

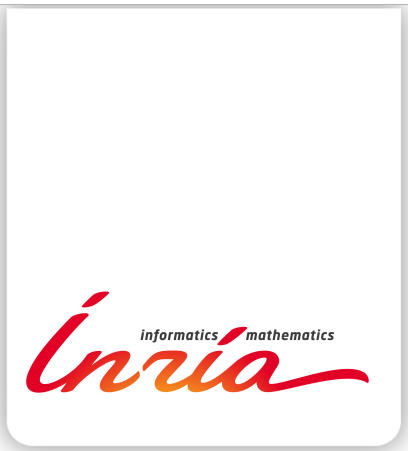
[9] G. Karypis and V. Kumar. *MeTiS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 4.0*. University of Minnesota, Department of Comp. Sci. and Eng., Army HPC Research Center, Minneapolis, 1998.

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

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

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

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



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399