EXACT PREDICTION OF QR FILL-IN BY ROW-MERGE TREES[1]

SUELY OLIVEIRA[2]

Abstract. Row-merge trees for forming the QR factorization of a sparse matrix A are closely related to elimination trees for the Cholesky factorization of $A^T A$. Row-merge trees predict the exact fill-in (assuming no numerical cancellation) provided A satisfies the strong Hall property, but over-estimates the fill-in in general. However, here a fast and simple post-processing step for row-merge trees is presented that predicts the exact fill-in for sparse QR factorization using Householder reflectors, for general matrices.

Key words. row-merge trees, elimination trees, QR factorization

1. Introduction. Matrix factorizations of sparse matrices typically result in creating further nonzero entries, or *fill-in*. If this fill-in can be accurately predicted in advance, then the factorization can be performed in less time, as the additional memory needed can be allocated once in advance. Notice that fill-in can be reduced with some matrix reordering algorithms. After that, the algorithms presented in this paper can be used. The algorithms discussed here do not affect the amount of fill-in but predict the fill-in before the factorization. Performance comparisons between our algorithms and Dulmage–Mendelsohn decomposition [22, 21] will be presented at the end of the paper, but notice that even though our new algorithms perform well in these comparisons, Dulmage–Mendelsohn decomposition can be used to predict fill-in and to achieve other goals as well (such as re-ordering rows and columns to give a block-triangular form), see for example [17, 22, 21]. Also notice that the algorithms in [21] can compute the fill-in in time $O(n\,nz(A))$ if no column permutation allowed, where $nz(A)$ is the number of nonzeros in A. The elimination tree algorithms presented in this paper appear to be significantly faster than this. Perhaps what is most significant about the algorithms presented here is that they are developed by modifying the construction of elimination trees or row-merge trees and may give new insight into the graphical prediction of fill-in. Furthermore, elimination trees [15] can be used to compute symbolic factorizations, to direct the factorization of matrices in multifrontal methods [15, 16] and also to assist in parallelizing sparse systems solvers. The QR fill-in problem considered in this paper is similar to the sparse Gaussian elimination with partial pivoting considered in [7], in the sense that the fill-in for QR factorization is the union of the fill-in for Gaussian elimination with partial pivoting, over all pivot sequences.

The elimination tree of a symmetric positive definite matrix B is readily defined in terms of its Cholesky factors: if $B = R^T R$ with R upper triangular, then $ET(B)$ is the forest with nodes $\{1, 2, \ldots, n\}$ where node p is the parent of node j if $p = \min\{\, i \mid i > j,\ r_{ji} \neq 0 \,\}$. That is, p is the first row/column to be updated at the jth stage of the Cholesky factorization of B. The elimination tree for Cholesky factorizations can be efficiently computed from the original matrix B; indeed, it can be computed in $O(nz(B)\,\alpha(nz(B)))$ time, where $nz(B)$ is the number of nonzeros of the $n \times n$ matrix B, and $\alpha(\cdot)$ is the inverse Ackerman function of Tarjan [24]. Whenever $j \geq i$, if $b_{jk} \neq 0$ and i is an ancestor of k in the elimination tree, then $r_{ij} \neq 0$ [15]. Furthermore, there are no other nonzeros in R. Note that each node is regarded as an ancestor of

[1]This research was partially supported by NSF grant DMS-9720607.

[2]Department of Computer Science and Department of Mathematics, The University of Iowa, Iowa City, IA 52242-1419.

itself. Unless there is numerical cancellation, $ET(B)$ does not depend on the numerical values in B. The elimination tree is sometimes called the *elimination forest* since it is not necessarily connected.

The graph of an $n \times n$ symmetric matrix B (denoted $G(B)$) is the graph on n nodes $\{1, 2, \ldots, n\}$ with an edge connecting nodes i and j (denoted $i \sim j$) if and only if $b_{ij} \neq 0$ and $i \neq j$; nodes are labeled by the row or column number. Unsymmetric and rectangular matrices are represented by bipartite graphs. For a symmetric positive definite matrix B, unless there is numerical cancellation in the Cholesky factorization $B = R^T R$, the symbolic factorization is the graph $G(R + R^T)$, and is denoted $G^+(B)$. Forming $G^+(B)$ can be defined in graph theoretic terms: $i \sim j$ in $G^+(B)$ if and only if there is a path from i to j in $G(B)$ whose nodes are all less than $\min(i, j)$.

For the QR factorization of an $m \times n$ matrix A it is more appropriate to consider the *bipartite graph* of A ($BG(A)$) whose nodes consist of the the rows $\mathcal{R} = \{ r_1, r_2, \ldots, r_m \}$ and columns $\mathcal{C} = \{ c_1, c_2, \ldots, c_n \}$ (which are disjoint sets) and $r_i \sim c_j$ if and only if $a_{ij} \neq 0$. The symbolic QR factorization of A can be computed similarly to the way the Cholesky factorization of a symmetric positive definite matrix is computed, and is based on the *row-merge tree of* A, which we define in section 1.2. We consider QR factorizations obtained by Householder orthogonalization. Each jth step of a Householder orthogonalization of A is achieved by multiplying matrix $A^{(j)}$ by a Householder reflector $H_j = I - \beta_j w_j w_j^T$ [12]. The matrices H_j are constructed to zero all the column entries below the jth diagonal of matrix $A^{(j)}$. We then set $A^{(j+1)} = H_j A^{(j)}$. At the nth step we are finally left with an upper triangular matrix $R = A^{(n)}$. The $Q = H_1 H_2 \ldots H_n$ factor is usually represented by the matrix W of Householder vectors (column j of W is the w_j vector used to construct the Householder reflector H_j) and the vector β. Thus, the main problem is to determine the sparsity structures of W and R. In this paper, the fill-in for the QR factorization will be determined from the graphs and related row-merge trees mentioned above, for any full-rank matrix.

1.1. Hall matrices and graphs. A bipartite graph $G = (V_1, V_2, E)$ is a *Hall graph* if for every set $N \subset V_2$, there are no fewer nodes (in V_1) adjacent to N, than are in N. For example, if $N = V_2$, then the set of nodes adjacent to N must be no smaller than N; thus $|V_1| \geq |V_2|$. The bipartite graph G is *strong Hall* if for all $N \subset V_2$ such that $N \neq V_2$, the set of nodes adjacent to N has size at least $|N| + 1$.

A matrix A is said to be a Hall matrix (resp. a strong Hall matrix) if $BG(A)$ is a Hall graph (resp. strong Hall graph). Consider V_1 to be $\mathcal{R}$ and V_2 to be $\mathcal{C}$. If A is a full-rank matrix with $m \geq n$, then it is a Hall matrix [9, Cor. 2.4]. In addition, if A is a strong Hall matrix then the fill pattern of the Cholesky factor R of the symbolic product $A^T A$ is equal to the fill pattern for the symbolic R factor in the QR factorization of A [1]. Problems arise for matrices A that are not strong Hall matrices. Consider, for example,

$$A = \begin{bmatrix} \times & \times & \times & \ldots & \times \\ & \times & & & \\ & & \times & & \\ & & & \ddots & \\ & & & & \times \end{bmatrix}. \tag{1}$$

which is a matrix with nonzeros in the first row and diagonal only. This matrix is a Hall matrix, but not a strong Hall matrix. The QR factorization of A is trivial.

for $i = 1, \ldots, m$: $S(u_i) \leftarrow \{\, c_j \mid a_{ij} \neq 0 \,\} \cup \{c_i\}$
for $j = 1, \ldots, n$: $S(p_j) \leftarrow S(u_j)$
for $j = 1, \ldots, n$: $S(p_j) \leftarrow \bigcup_{x \in \mathcal{U} \cup \mathcal{P} : c_j \in S(x)} S(x) \bigcap \{c_{j+1}, \ldots, c_n\}$

FIG. 1. *Symbolic QR factorization*

However, forming $A^T A$ symbolically gives a dense matrix (due to the dense row in A). Thus from the nonzero structure of $A^T A$, it appears that the R in the QR factorization is dense, although it clearly is not. On the other hand,

$$A' = \begin{bmatrix} \times & \times & \times & \ldots & \times \\ \times & \times & \times & \ldots & \times \\ & \times & & & \\ & & \times & & \\ & & & \ddots & \\ & & & & \times \end{bmatrix} \tag{2}$$

is a strong Hall matrix, and the R in the QR factorization *is* dense. As we will show in this paper distinguishing these cases can be done by post-processing the row-merge tree. A paper which finds the fill-in structure of Q and R is [13]. Their methods are based on matchings and Hall matrices properties, which differ from the methods used here. In this paper, the fill-in for the QR factorization for non-strong Hall matrices will be determined from the graphs and related row-merge trees.

1.2. QR factorizations and row-merge trees. Hereafter, we assume that A is an $m \times n$ Hall matrix with $m \geq n$, and the rows and columns have been matched so that $a_{ii} \neq 0$ for all i. As mentioned before, if A is not a Hall matrix, it cannot be full rank.

The appropriate generalization of elimination trees to QR factorizations is the *row-merge tree* [14, 20] of a non-symmetric matrix A. This can be derived from the crude symbolic QR factorization of A; Let A be $m \times n$, with $m \geq n$. Let $\mathcal{U} = \{u_1, u_2, \ldots, u_m\}$ denote the *unprocessed* rows of A, and $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$ the processed rows of A (existing rows of R, after the QR factorization has been performed). Define $index(u_l) = l$ and $index(p_k) = k$. Define $S(x)$ to be the set of nonzero entries in x, specifically consider $S(u_i)$ to be the set of nonzero entries columns in the ith row *before* the factorization algorithm is started, plus the diagonal column (vertex number for the diagonal column). The set $S(p_j)$ is built through the last loop in Figure 1 pseudocode. It gives an upper bound on the set of nonzero entries in row j ($j \leq m$) after the jth step of Householder QR factorization, less the entries before the $(j+1)$th column. We define the set $\mathcal{R}_j \subset \mathcal{R}$ to be set of rows x which have a nonzero in the jth column ($c_j \in S(x)$) at the jth step of the Householder QR with $index(x) \geq j$. At the jth step of this loop, the union is over all rows in $\mathcal{U}$ or in $\mathcal{P}$ such that the row x is in $\mathcal{R}_j$. The set of nonzero entries of row p_j ($S(p_j)$) is then defined as this union intersected with the set of posterior columns (i.e $c_{j+1}, \ldots, c_n$). Unless there is numerical cancellation, if any row in $\mathcal{R}_j$ has a nonzero in column $k > j$, after performing the jth step, all rows in this set will have a nonzero entry in column k.

The symbolic QR factorization described in Figure 1, gives an upper bound on the sparsity patterns of the matrix R: $r_{ij} \neq 0$ only if $i = j$ or $c_j \in S(p_i)$. It is not exact

```
for k = 1, 2, ..., n
    for each i < k where b_ik ≠ 0
        s ← top of tree containing i
        if s ≠ k
            parent(s) ← k
```

FIG. 2. *Liu's algorithm for computing* ET(B)

in general, since it assumes $|\mathcal{R}_j| > 1$ for all j; in other words a row with a nonzero in column j (sparsity pattern $S(p_j)$) is assumed to participate in succeeding stages of the QR factorization, even if it is the only row with a nonzero in column j. The algorithm of Figure 1 is exact if A is a strong Hall matrix (since $|\mathcal{R}_j| > 1$ for all j, in this case); however, if A is a Hall matrix but not a strong Hall matrix, then the fill-in predicted by Figure 1 can greatly over-estimate the true fill-in. In addition, the cost of this algorithm is similar to the cost of actually performing the QR factorization, so row-merge tree have been developed [14].

Note that if A has nonzeros on the diagonal, then the first line of Figure 1 can be replaced by: **for** $i = 1, \ldots, m$: $S(u_i) \leftarrow \{\, c_j \mid a_{ij} \neq 0 \,\}$. Non-zeros on the diagonal can usually be achieved by computing a maximal matching of columns to rows; in fact, for Hall matrices the maximal matching will also be a complete matching (that is, every column will be matched to a row). Consequently, for Hall matrices, swapping the row matched to column j with row j for all j will put nonzeros in all of the diagonal entries without creating additional fill-in.

The *row-merge tree* on nodes $\mathcal{R} \cup \mathcal{C}$ can then be defined by the condition that c_j is the parent node of r_i if $j = \min\{\, k \mid c_k \in S(u_i) \,\}$, and the parent of node c_k is c_j where $j = \min\{\, l \mid c_l \in S(p_k) \,\}$. Hereafter we denote the *row-merge tree* of A as $RM(A)$.

Related to these graphs is the *column intersection graph* of A [9]. Providing there is no numerical cancellation in forming $A^T A$, the column intersection graph $CIG(A)$ is just $G(A^T A)$. More formally, the column intersection graph of A, is defined as the graph on the nodes $\{c_1, \ldots, c_n\}$ where $c_k \sim c_l$ (c_k is connected to c_l) if there is a j where both a_{jk} and a_{jl} are nonzero. Liu [14] showed that the elimination tree of $CIG(A)$ ($ET(CIG(A))$) is equal to $RM(A)$ minus all nodes in $\mathcal{R}$, and edges incident to $\mathcal{R}$. Computationally, $ET(CIG(A))$ is a useful tool because it can be computed directly from A without needing to construct $CIG(A)$ or $A^T A$, by using the algorithm in Figure 3. We also refer the reader to the algorithm of [11, Fig. 2] which achieves the same task and is similar in nature. Another paper where this task is achieved is [3, 4] using an unrelated algorithm.

The algorithm in Figure 3 is a modification of Liu's algorithm for the symmetric case [15], which is shown in Figure 2. Note that the line "$s \leftarrow$ top of tree containing j" can be computed using a union-find data structure with path compression in amortized time $O(\log(n))$ time [2, p. 449], or in amortized time $O(\alpha(nz(A), n))$ where α is an inverse of the Ackermann function if the method also uses the union-by-rank heuristic [2, pp. 450–457]. However, the union-by-rank heuristic requires additional work since the representative of a subset is determined by the algorithm, and is not necessarily the root of a component of the currently-constructed elimination tree or forest.

Notice that the elimination tree $T = ET(CIG(A))$ is the smallest tree or forest T

```
for k = 1, 2, ..., n
    for each i such that a_ik ≠ 0
        j ← max{ r < k | a_ir ≠ 0 }
        s ← top of tree containing j
        if s ≠ k
            parent(s) ← k
```

FIG. 3. *Algorithm for constructing* ET(CIG(A))

```
for j = 1, 2, ..., n:   [ count(j) ← 0 ]
for i = 1, 2, ..., m:   [ k ← min{ s | a_is ≠ 0 }; count(k) ← count(k) + 1 ]
    (Note: if the set is empty, jump to the end of the loop.)
for j = 1, 2, ..., n:   [ count(j) ← count(j) + ( Σ_{x:x col. child of j} count(x) ) − 1 ]
for j = 1, 2, ..., n:   if count(j) = 0 then [ parent(j) ← null ]
```

FIG. 4. *Post-processing step*

that satisfies the relations "l is an ancestor of k" whenever $l > k$ and $a_{jk}, a_{jl} \neq 0$ for some j. The row-merge tree $RM(A)$ can be formed from T by adding $\mathcal{R}$ to the set of nodes, and adding edges from r_i to c_k if $k = \min\{\, s \mid a_{is} \neq 0 \,\}$. For an arbitrary tree or forest T, let $RM(T)$ denote the tree or forest formed by adding $\mathcal{R}$ to the set of nodes, and adding edges from r_i to c_k where $k = \min\{\, s \mid a_{is} \neq 0 \,\}$.

Given the elimination tree $T = ET(CIG(A))$, the fill-in can be estimated as follows [15]: Column j of R has nonzeros in rows r_k where c_k is an ancestor of some row r_i in $RM(T)$ where $a_{ij} \neq 0$ and $k \leq j$. Row i of W has nonzeros in columns c_k where c_k is an ancestor of r_i in $RM(T)$ and $k \leq i$. The estimated fill-in based on the $ET(CIG(A))$ is the same as the fill-in predicted by the crude symbolic QR factorization algorithm in Figure 1.

The fill-in for the QR factorization can be computed from $ET(CIG(A))$ and A with just $O(n)$ storage overhead and $O(nz(R) + nz(W) + n)$ time; just to count the number of nonzeros needs just $O(1)$ additional storage and $O(nz(R) + nz(W) + n)$ time.

2. Post-processing the row-merge tree. If $|\mathcal{R}_k| > 1$ for all k, then the assumptions underlying the algorithm in Figure 1 are correct and the methods of the previous paragraph for computing the sparsity patterns of R and W give the correct fill-in. However, if $|\mathcal{R}_k| = 1$ for some k, then we must have $\mathcal{R}_k = \{r_k\}$. In this case, the elimination tree T must be post-processed to give the correct fill-in.

In this section we introduce a post-processing algorithm which should be applied when $|\mathcal{R}_k| = 1$ for some k. This algorithm should be performed after constructing $ET(CIG(A))$ and adding the row nodes to construct the row-merge tree of A as described in the previous section. This post-processed row-merge tree predicts fill-in at least as well as the original row-merge tree does. How to predict fill-in given a row-merge tree was described near the end of the last section. We have to thread the tree so that the children of a given node can be found in $O(1)$ time per child node. Threading a tree of n nodes (defined by a *parent* array) requires just $O(n)$ operations and $O(n)$ storage. The post-processing step for an $m \times n$ matrix A takes just $O(m+n)$ operations, including the tree threading step.

The post-processing algorithm is shown in Figure 4. Note that the final computed value of $count(j)$ is the number of rows that participate in the jth stage of the QR factorization minus one. It is also the number of row nodes in the row-merge tree rooted at c_j minus the number of column nodes in this tree. The first loop in Figure 4 initializes the array $count(j)$ for all n columns. The second loop finds out for each row i what is the column number of the first nonzero appearance in that row, and adds 1 to the count for that column. In other words, $count(k)$ states how many rows have the first nonzero in column k (the number of row children of k). For example (1) in Section 1 we have: $count(1) = count(2) = count(3) = count(4) = 1$ after the second loop. The third loop finds the number of row descendents of node j minus the number of its column descendents (including itself). The final values of $count(j)$ are shown for a couple of example matrices in Figure 5. If the value of $count(j)$ is equal to zero than that node is disconnected from the tree above. For the example matrices A and A' (equations (1) and (2)), the row merge trees and the effect of post-processing is shown in Figure 5.

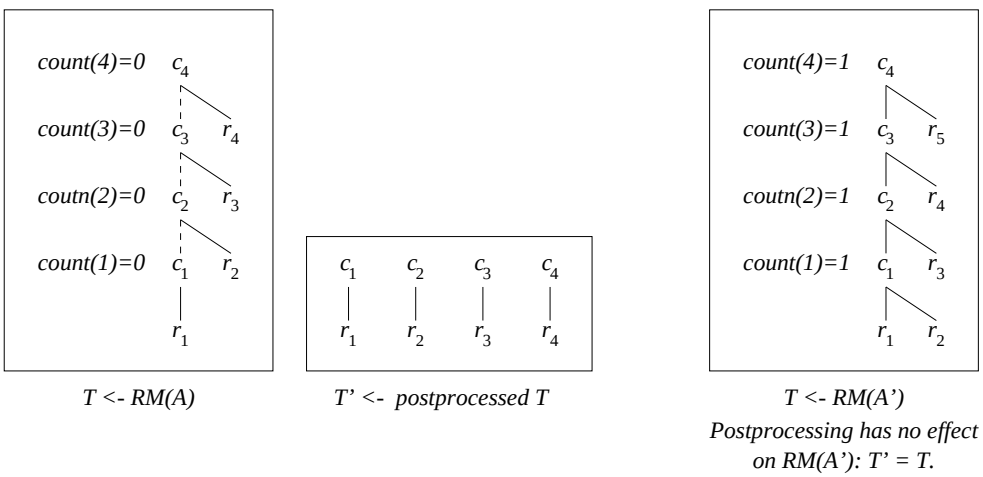

FIG. 5. *Row merge trees and the effects of post-processing on example matrices A and A'*

The basis for this post-processing algorithm is a more refined understanding of the symbolic Householder QR factorization shown in Figure 1. As mentioned earlier that algorithm is expensive and previous approaches based on row merge trees exit. Our new algorithms show that we can do better by postprocessing the row merge trees. The set $S(p_j)$ is the set of nonzero entries in row j after the jth step of the QR algorithm, minus the c_j entry. Note that the jth row no longer participates in the succeeding steps of the QR factorization after the jth step. If only row j has a nonzero in column j at the jth stage of the QR factorization, then no Householder operation needs to be performed, and thus $S(p_j)$ is irrelevant to the calculation of $S(p_l)$ for $l > j$. This fact is ignored by the symbolic QR factorization in Figure 1.

If we re-consider the algorithm in Figure 1 with this in mind, then for $j = 1, 2, \ldots, n$ there should be a count of the number of rows in $\mathcal{R}_j$ (which is $count(j)+1$). If $|\mathcal{R}_j| \leq 1$, then the jth step of the QR factorization is trivial, and $S(p_j)$ is irrelevant for future steps in the factorization. If we add one to $count(j)$ at the end of the algorithm in Figure 4, we obtain the number of rows participating in the jth stage of the factorization. This explains why we should disconnect the tree when $count(j) = 0$,

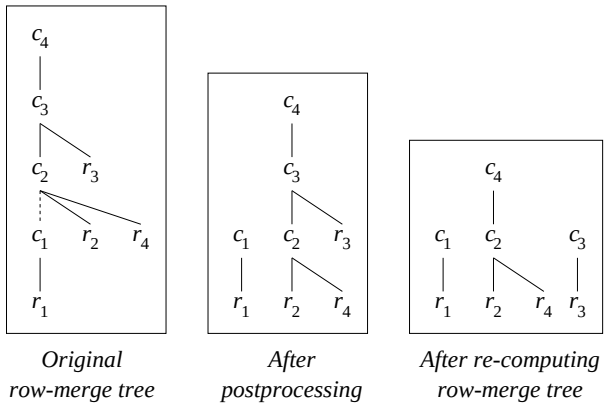

FIG. 6. *Original, post-processed and re-computed elimination trees for A''*

creating the post-processed tree T'. Now we have new row merge trees from which we can obtain improved estimates of the fill-in.

After post-processing, $nz(W)$ can be computed from $ET(CIG(A))$ and A in $O(n)$ time by the formula

$$nz(W) = \sum_{j=1}^{n} (count(j) + 1). \tag{3}$$

If we postprocess the elimination tree T' again, we do not change the value of the *count* array and have no affect on $nz(W)$. However, this post-processed elimination tree T' can be used to compute a new elimination tree T'' which will give more information about $nz(W)$. This re-processing creates a new elimination tree or forest T''. To illustrate the necessity of re-processing the post-processed elimination tree, consider the matrix

$$A'' = \begin{bmatrix} \times & \times & \times & \times \\ & \times & & \\ & & \times & \\ & \times & & \times \end{bmatrix}. \tag{4}$$

The steps of the re-processing algorithm are shown in Figure 7. The algorithm mirrors the earlier algorithm in Figure 3 for computing $ET(CIG(A))$, except that T' is used to filter the construction of the elimination tree. This means that two nodes are connected in T'' they must be connected in T'. The elimination tree $ET(CIG(A''))$ and the results of post-processing and re-processing the elimination tree using the algorithm of Figure 7 are shown in Figure 6.

To explain how the algorithm proceeds for the this example, start by looking at T' (the tree obtained after the post-processing scheme is applied). The generated tree T'' by the re-processing algorithm, is similar to the original tree T', until the second loop reaches the value $k = 3$. When $k = 3$ ($p = 1$, $j = 2$, and $s = 2$) the statement $parent(s) \leftarrow k$ of algorithm 7 is not executed, since k, s and p do not belong to the same component of T'. Nevertheless during the next iteration of this loop, when $k = 4$, that same statement will connect node 2 to its new parent, node 4.

The post-processing and subsequent re-processing steps can be applied recursively until post-processing does not alter the elimination tree or forest (that is, $count(j) > 0$

```
for k = 1, 2, ..., n:   parent(j) ← NULL
for k = 1, 2, ..., n
    for each i such that a_ik ≠ 0
        p ← min{ q | a_iq ≠ 0 }
        j ← max{ r < k | a_ir ≠ 0 }
        s ← top of tree containing j
        if k, s, and p all belong to the same component of T' and s ≠ k
            parent(s) ← k
```

FIG. 7. *Re-processing step which constructs T'' based on T'*

for all j). Define $T'' = ET^*(A, T)$ to be the elimination forest that results from the following two step process: 1) applying post-processing to the elimination tree T and then 2) re-processing the resulting forest T' to produce T'', using algorithms in Figures 4 and 7, respectively.

Since $ET^*(A, T)$ is either unchanged ($T'' = T$) or more disconnected than T, recursive applications of $T^{(k+1)} \leftarrow ET^*(A, T^{(k)})$ will terminate after at most n iterations. In practice, the number of iterations needed to obtain $T^{(k+1)} = T^{(k)}$ is very small. Let T^* be this final elimination forest where $T^* = ET^*(A, T^*)$. Then for this forest, $count(j) > 0$ for all nodes j that are not root nodes. Thus, the predicted sparsity structure of the rows in each tree of T^* is correct, and the sparsity structure predicted by T^* is the exact fill-in pattern for the Householder QR factorization.

Determining if two nodes are in a common connected component of a tree or forest T (as done in the core of loop k) can be performed in $O(1)$ time after an *ancestor* array is constructed containing the top-most ancestor of each node in T, which can be done in $O(n \log n)$ time where n is the number of nodes of T.

3. Computational results. The following operations were implemented: computing the elimination tree of the column intersection graph $ET(CIG(A))$ (including threading the trees), the post-processing algorithms of Figures 4 and 7, a maximal matching algorithm [6] for bipartite graphs, and the Dulmage–Mendelsohn decomposition [22]. Note that only path compression was used for the union-find data structures in the construction of the elimination trees [2, p. 447], and not union by rank. This is the approach used for constructing $ET(CIG(A))$ in the recent SuperLU code under development by Demmel *et al.*, which is referred to in [5]. The implementation of the maximal matching algorithm follows the recommendations of [6]. These algorithms were implemented in terms of the Meschach matrix library in C [23]. Our algorithms predicted the exact fill-in for all test matrices, as expected. Our test matrices included all rectangular matrices plus some square matrices from the Harwell–Boeing collection. Matrices which had more columns than rows were transposed before performing any calculations. No column or row re-ordering was performed on these matrices, since we aim solely to analyze the performance of our fill-in prediction algorithms.

Other test problems were 10×10 matrices with nonzeros only in the first row and the diagonal. Our new routines correctly deduced that the number of nonzero entries for R, which for this matrix has just 19 nonzeros, using the post-processed $ET(CIG(A))$ tree (i.e., no fill-in), while the unprocessed tree predicted 55 nonzeros (i.e., complete fill-in, as were expected from $A^T A$). Also, matrix A'' of equation (4) was used as a test matrix, and the results confirmed expectations. Our routines were compiled and run on a Hewlett–Packard Visualize C3000 with standard optimization

switches set on.

The basic data for the matrices are listed in Table 1, and the timings for the operations are listed in Table 2. In Table 1, $\#nz$ is the number of nonzeros of the original matrix, $nz(R)$ is the number of nonzeros of R in the QR factorization, and $nz(W)$ is the number of nonzeros in the matrix of Householder vectors that represents Q. In Table 2, $ETCIG1$ refers to the time to compute the elimination tree of the column intersection graph of the matrix, $ETCIG2$ is the amount of additional time needed for all subsequent computations of $ET^*(A,T)$ in order to compute the final elimination tree, T^*. The columns "thread" and "post-proc" give the time needed to thread the elimination tree, and to perform the post-processing of the algorithm in Figure 4, respectively.

The Dulmage–Mendelsohn decomposition gives an alternative way of predicting the exact fill-in of QR factorization. The decomposition itself is a re-ordering of the rows and columns of the matrix so that the matrix is put into block upper triangular form with strong Hall diagonal blocks. The columns "coarse" and "fine" of Table 2 give the times needed for the coarse and fine phases of the Dulmage–Mendelsohn decomposition [22] after the computation of a maximal matching. Finally, the column "MM" gives the time needed to compute a maximal matching, which is an essential step in the Dulmage–Mendelsohn algorithm. Note that the times given do not include the times for computing the fill-in for the QR factorization using either approach.

When we add the times $ETCIG1$ and $ETCIG2$ and compare with the sum of the times for the coarse and fine phases of the Dulmage–Mendelsohn method, we always get smaller times for the sum of elimination tree times ($ETCIG1$ plus $ETCIG2$). Notice that after the Dulmage-Mendelsohn decomposition, there are still other steps for the prediction of the QR fill-in which are not taken into account. Our results show that our new algorithms based on post-processed row-merge trees are usually faster than methods based on Dulmage–Mendelsohn approaches.

Acknowledgements. I would like to thank David Stewart for proofreading the paper, Alex Pothen and Esmond Ng for interesting conversations, and finally the referees for comments which improved the paper.

REFERENCES

[1] T. F. Coleman, A. Edenbrandt, and J. R. Gilbert, *Predicting fill for sparse orthogonal factorization*, J. Assoc. Comp. Mach., 33 (1986), pp. 517–532.

[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press/McGrall-Hill, Cambridge, MA/New York, Toronto, 1990.

[3] T. A. Davis and W. W. Hager, *Modifying a sparse Cholesky factorization*, SIAM J. Matrix Anal. Appl., 20 (1999), pp. 606–627 (electronic).

[4] ———, *Multiple-rank modifications of a sparse Cholesky factorization*, Tech. Report TR-99-006, Department of Computer and Information Science and Engineering, University of Florida, 1999.

[5] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu, *A supernodal approach to sparse partial pivoting*, SIAM J. Matrix Anal. Appl., 20 (1999), pp. 720–755 (electronic).

[6] I. S. Duff and T. Wiberg, *Remarks on implementations of $O(n^{1/2}\tau)$ assignment algorithms*, ACM Trans. Math. Software, 14 (1988), pp. 267–287.

[7] A. George and E. Ng, *Symbolic factorization for sparse Gaussian elimination with partial pivoting*, SIAM J. Sci. Stat. Comp., 8 (1987), pp. 877–898.

[8] A. George and E. Ng, *Symbolic factorization for sparse Gaussian elimination with partial pivoting*, SIAM J. Sci. Statist. Comput., 8 (1987), pp. 877–898.

[9] J. R. Gilbert and E. G. Ng, *Predicting structure in nonsymmetric sparse matrix factorizations*, in Graph Theory and Sparse Matrix Computation, A. George, J. R. Gilbert, and

J. W. H. Liu, eds., vol. 56 of IMA Volumes in Mathematics and its Applications, New York, Berlin, Heidelberg, 1993, Institute for Mathematics and its Applications, Springer-Verlag, pp. 107–139.

[10] J. R. Gilbert and E. G. Ng, *Predicting structure in nonsymmetric sparse matrix factorizations*, in Graph theory and sparse matrix computation, Springer, New York, 1993, pp. 107–139.

[11] J. R. Gilbert, X. S. Li, E. Ng, and B. Peyton, *Predicting the structure of sparse orthogonal factors.* in preparation.

[12] G. Golub and C. V. Loan, *Matrix Computations*, Johns Hopkins University Press, Baltimore, Maryland, 3rd ed., 1996.

[13] D. R. Hare, C. R. Johnson, D. D. Olesky, and P. van den Driessche, *Sparsity analysis of the QR factorization*, SIAM J. Matrix Anal. Appl., 14 (1993), pp. 655–669.

[14] J. W. H. Liu, *On general row merging schemes for sparse Givens transformations*, SIAM J. Sci. Stat. Computing, 7 (1986), pp. 1190–1211.

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

[16] ———, *The multifrontal method for sparse matrix solution: theory and practice*, SIAM Review, 34 (1992), pp. 82–109.

[17] E. G. Ng and B. W. Peyton, *Some results on structure prediction in sparse QR factorization*, SIAM J. Matrix Anal. Appl., 17 (1996), pp. 443–459.

[18] S. Oliveira, *A new parallel chasing algorithm for transforming arrowhead matrices to tridiagonal form*, Mathematics of Computation, 67 (1998), pp. 221–235.

[19] ———, *Reprocessing a postprocessed elimination tree to obtain exact sparsity prediction in QR factorization*, (1999). The University of Iowa TR-127.

[20] P. E. Plassmann, *Sparse Jacobian estimation and factorization on a multiprocessor*, in Large-Scale Numerical Optimization, T. F. Coleman and Y. Li, eds., SIAM Proceedings, Philadelphia, PA, 1990, SIAM Publ., pp. 152–179.

[21] A. Pothen, *Predicting the structure of sparse orthogonal factors*, Linear Algebra Appl., 194 (1993), pp. 183–203.

[22] A. Pothen and C.-J. Fan, *Computing the block triangular form of a sparse matrix*, ACM Trans. Math. Software, 16 (1990), pp. 303–324.

[23] D. E. Stewart and Z. Leyk, *Meschach: Matrix Computations in C*, Australian National University, Canberra, 1994. Proceedings of the CMA, #32.

[24] R. E. Tarjan, *Data Structures and Network Algorithms*, CBMS Regional Conference Series, SIAM Publ., 1983.

Name	$m \times n$	$\#nz$	$nz(R)$	$nz(W)$
impcol_b	59×59	312	877	671
watson1	58×59	340	1711	696
watson2	66×67	409	2211	1233
impcol_c	137×137	411	3835	3233
ash219	219×85	438	1238	7367
impcol_a	207×207	572	3615	2216
ash331	331×104	662	1026	13984
watson3	124×125	780	7750	3840
lop163	163×163	935	3251	2046
fs183_1	183×183	1069	15889	14440
ash608	608×188	1216	2970	43637
impcol_e	225×225	1308	6603	5594
impcol_d	425×425	1339	21626	13677
abb313	313×176	1557	6794	22928
ash958	958×292	1916	4516	104314
1138_bus	1138×1138	2596	99137	62572
fs680_1	680×680	2646	204152	203518
mcca	180×180	2659	5882	1730
watson4	467×468	2869	109278	88452
wm1	207×277	2909	18222	22776
wm2	207×260	2942	19879	23120
wm3	207×260	2948	19925	23121
bcspwr07	1612×1612	3718	66519	43260
bcspwr08	1624×1624	3837	87029	54749
bcspwr09	1723×1723	4117	122463	109684
illc1033	1033×320	4732	8756	92309
gre_1107	1107×1107	5664	328891	130060
fs760_1	760×760	5976	235707	223292
illc1850	1850×712	8758	71849	474111
watson5	1853×1854	10847	1717731	447291
bcspwr10	5300×5300	13571	2653153	2432762
zenios	2873×2873	15032	97430	94444
add20	2395×2395	17319	2565125	2178211
add32	4960×4960	23884	9381844	8687422
mcfe	765×765	24382	91277	24548
gemat12	4929×4929	33111	5407842	5071559
gemat11	4929×4929	33185	5415469	5071185
beause	497×507	44551	120727	116106
gemat1	4929×10595	47369	235707	223292
beacxc	497×506	50409	100373	110571
beaflw	497×507	53403	120855	114601
orani678	2529×2529	90158	2776652	1835474
memplus	17758×17758	126150	156383209	141158005

TABLE 1
Data for test matrices and their QR factorizations

Name	$ETCIG1$	$ETCIG2$	thread/post-proc	coarse/fine	MM
impcol_b	1.05(−4)	1.31(−4)	1.20(−5)/2.60(−5)	2.95(−4)/3.97(−4)	1.38(−4)
watson1	1.10(−4)	—	1.10(−5)/2.40(−5)	4.02(−4)/3.87(−4)	1.41(−4)
watson2	1.29(−4)	—	1.20(−5)/2.90(−5)	4.27(−4)/4.45(−4)	1.31(−4)
impcol_c	1.43(−4)	2.36(−4)	1.50(−5)/4.20(−5)	3.09(−4)/6.14(−4)	2.09(−4)
ash219	1.38(−4)	—	1.30(−5)/3.50(−5)	5.70(−4)/7.60(−4)	1.82(−4)
impcol_a	1.90(−4)	3.58(−4)	2.00(−5)/6.10(−5)	3.66(−4)/1.41(−3)	6.30(−4)
ash331	1.83(−4)	—	1.40(−5)/4.20(−5)	7.00(−4)/1.05(−3)	2.58(−4)
watson3	2.03(−4)	—	1.50(−5)/3.80(−5)	5.60(−4)/7.52(−4)	1.84(−4)
lop163	2.63(−4)	3.76(−4)	1.70(−5)/4.80(−5)	3.30(−4)/8.83(−4)	1.24(−4)
fs183_1	2.89(−4)	4.25(−4)	1.80(−5)/5.20(−5)	3.75(−4)/1.03(−3)	1.31(−4)
ash608	3.13(−4)	—	1.80(−5)/8.80(−5)	1.04(−3)/1.82(−3)	4.04(−4)
impcol_e	3.28(−4)	—	2.00(−5)/6.60(−5)	3.86(−4)/1.83(−3)	3.01(−4)
impcol_d	3.65(−4)	7.20(−4)	2.80(−5)/1.04(−4)	4.61(−4)/1.68(−3)	4.28(−4)
abb313	3.40(−4)	—	1.80(−5)/5.70(−5)	8.85(−4)/1.54(−3)	2.84(−4)
ash958	4.57(−4)	—	2.30(−5)/9.90(−5)	1.42(−3)/2.87(−3)	6.70(−4)
1138_bus	8.51(−4)	1.84(−3)	6.20(−5)/2.70(−4)	7.63(−4)/3.86(−3)	6.95(−4)
fs680_1	6.70(−4)	2.75(−4)	4.00(−5)/1.65(−4)	5.85(−4)/2.87(−3)	4.49(−4)
mcca	5.05(−4)	1.51(−3)	1.90(−5)/5.20(−5)	3.77(−4)/1.67(−3)	1.61(−4)
watson4	8.01(−4)	—	3.00(−5)/1.19(−4)	1.25(−3)/2.52(−3)	3.88(−4)
wm1	5.65(−4)	7.70(−4)	1.90(−5)/6.10(−5)	1.07(−3)/1.92(−3)	3.25(−4)
wm2	5.72(−4)	7.78(−4)	1.90(−5)/6.00(−5)	1.09(−3)/1.92(−3)	5.62(−4)
wm3	5.78(−4)	8.28(−4)	1.80(−5)/6.10(−5)	1.12(−3)/1.97(−3)	5.80(−4)
bcspwr07	1.09(−3)	—	8.10(−5)/3.74(−4)	9.13(−4)/5.45(−3)	8.23(−4)
bcspwr08	1.18(−3)	3.43(−3)	8.20(−5)/3.86(−4)	9.43(−4)/6.37(−3)	8.35(−4)
bcspwr09	1.14(−3)	2.80(−3)	9.00(−5)/4.12(−4)	9.44(−4)/5.83(−3)	8.81(−4)
illc1033	8.78(−4)	1.21(−3)	2.50(−5)/1.06(−4)	2.18(−3)/4.34(−3)	9.46(−4)
gre_1107	1.74(−3)	2.81(−3)	5.90(−5)/2.68(−4)	8.40(−4)/5.43(−3)	7.71(−4)
fs760_1	1.54(−3)	2.29(−3)	4.40(−5)/1.90(−4)	6.75(−4)/4.61(−3)	2.76(−4)
illc1850	1.72(−3)	2.50(−3)	4.10(−5)/2.22(−4)	3.65(−3)/7.92(−3)	2.67(−3)
watson5	2.50(−3)	—	9.40(−5)/4.67(−4)	4.12(−3)/9.69(−3)	1.37(−3)
bcspwr10	4.47(−3)	1.02(−2)	2.40(−4)/1.30(−3)	2.21(−3)/1.85(−2)	2.31(−3)
zenios	3.27(−3)	5.94(−3)	1.44(−4)/6.93(−4)	1.61(−3)/1.36(−2)	1.53(−3)
add20	4.95(−3)	7.25(−3)	1.14(−4)/6.05(−4)	1.60(−3)/1.45(−2)	1.51(−3)
add32	7.07(−3)	1.20(−2)	2.77(−4)/1.32(−3)	2.70(−3)/2.45(−2)	2.66(−3)
mcfe	4.53(−4)	5.66(−3)	4.30(−5)/2.71(−4)	1.33(−3)/1.47(−2)	8.28(−4)
gemat12	1.03(−2)	1.50(−2)	2.30(−4)/1.27(−3)	2.79(−3)/3.46(−2)	6.71(−3)
gemat11	9.35(−3)	1.45(−2)	2.26(−4)/1.63(−3)	2.85(−3)/3.62(−2)	6.51(−3)
beause	2.05(−2)	2.83(−2)	3.60(−5)/2.89(−4)	1.15(−2)/3.60(−2)	7.68(−3)
gemat1	1.72(−2)	—	2.52(−4)/2.29(−3)	3.07(−2)/6.82(−2)	2.71(−2)
beacxc	2.26(−2)	2.96(−2)	3.20(−5)/2.74(−4)	1.49(−2)/4.25(−2)	1.00(−2)
beaflw	2.39(−2)	3.15(−2)	3.10(−5)/2.85(−4)	1.51(−2)/4.30(−2)	1.04(−3)
orani678	4.38(−2)	5.60(−2)	1.43(−3)/1.10(−3)	8.62(−3)/1.01(−1)	7.17(−2)
memplus	8.20(−2)	1.14(−1)	1.07(−3)/7.83(−3)	2.51(−2)/3.69(−1)	1.84(−2)

TABLE 2

Timings for operations on various matrices. Entry $x.xx(yy)$ means $x.xx \times 10^{yy}$ seconds.