

# On the Existence of Nonsymmetric Matrices with Perfect Elimination Orderings \*

Tzu-Yi Chen

Department of Math and Computer Science  
 Pomona College  
 610 N. College Ave.  
 Claremont, CA 91711  
 tzuyi@cs.pomona.edu

Melissa Egan<sup>†</sup>

Department of Computer Science  
 Naval Postgraduate School  
 833 Dyer Rd.  
 Monterey, CA 93943  
 mkegan@nps.edu

## Abstract

Permuting the rows and columns of a sparse matrix can dramatically reduce the memory requirements of a subsequently computed LU factorization. A perfect elimination matrix is one whose rows and columns can be permuted so that its LU factors require no additional space on top of that required by the original matrix. An implementation of an  $O(n^3)$  algorithm, presented in [6], for determining whether a matrix is perfect elimination is described. Running this code on 180 matrices from an assortment of application areas shows that 19 of them are perfect elimination and that almost half contain at least 40% eliminable entries.

**Keywords:** sparse nonsymmetric matrices, perfect elimination orderings, direct methods

## 1 Introduction

Applications ranging from computational fluid dynamics to circuit simulation depend on methods for solving systems of linear equations  $Ax = b$ . The input to the method should be an  $n \times n$  nonsingular matrix  $A$  and a vector  $b$ ; the output is then the solution vector  $x$ . Direct methods, which are generally considered more robust than the alternative of iterative methods, begin by computing the LU factorization of  $A$  (i.e., lower and upper triangular matrices  $L$  and  $U$  such that  $LU = A$ ). They then find  $x$  by efficiently solving two triangular systems of equations: first  $Ly = b$  for  $y$  and then  $Ux = y$  for  $x$ .

In practice, the matrix  $A$  is often both large and *sparse*. In other words, not only is  $n$  large, but in addition so many of the matrix entries have value 0.0 that significant space can be saved by storing only the values of the nonzero entries and information about their locations in the matrix. Unfortunately, even when  $A$  is sparse, its  $L$  and  $U$  factors may be dense. Hence, just because  $A$  is sparse and requires

limited space to store does not necessarily mean that  $L$  and  $U$  require similarly little space. For example, consider the matrix  $A$  in Figure 1. The  $n \times n$  matrix has  $3n - 2$  nonzeros (i.e.,  $\text{nnz}(A) = 3n - 2$ ), yet  $\text{nnz}(L + U) = n^2$ .

$$\begin{bmatrix} \times & \times & \times & \times \\ & \ddots & & \\ \times & & \times & \\ \times & & & \times \\ \times & & & & \times \end{bmatrix} = \begin{bmatrix} \times & & & & \\ \vdots & \ddots & & & \\ \times & \times & \times & & \\ \times & \times & \times & \times & \\ \times & \times & \times & \times & \times \end{bmatrix} \times \begin{bmatrix} \times & \times & \times & \times \\ & \times & \times & \times \\ & & \ddots & \vdots \\ & & & \times \end{bmatrix}$$

Figure 1: A sparse matrix with dense LU factors. Only elements marked by a  $\times$  have nonzero values.

Elements which are nonzero in  $L + U$ , but zero in  $A$ , are referred to as *fill*. Excessive fill can render a direct method impractical if the user lacks the computer memory necessary to store the  $L$  and  $U$  factors. Fortunately, permuting the rows and columns of  $A$  before computing its LU factorization can greatly reduce the fill. For example, consider Figure 2, which shows the effect of reverse ordering the rows and columns of the matrix in Figure 1. The factorization now introduces no fill (i.e.,  $\text{nnz}(A) = \text{nnz}(L + U)$ ).

$$\begin{bmatrix} \times & & \times \\ & \times & \times \\ & & \ddots & \times \\ \times & \times & \times & \times \end{bmatrix} = \begin{bmatrix} \times & & & \\ & \times & & \\ & & \ddots & \\ \times & \times & \times & \times \end{bmatrix} \times \begin{bmatrix} \times & & \times \\ & \times & \times \\ & & \ddots & \times \\ & & & \times \end{bmatrix}$$

Figure 2: A matrix whose LU factorization introduces no fill. This is the same matrix as that in Figure 1, but with the rows and columns in reverse order.

Because the matrix in Figure 1 (and Figure 2) can be permuted to completely eliminate fill, it is called a *perfect elimination* matrix. Similarly, the associated permutation is called a *perfect elimination ordering*. Throughout most of this paper we are concerned only with the nonzero structure of matrices, and not with the values of their nonzero entries. Therefore, we make the assumption that there is no numerical cancellation, and hence no unexpected introduction of 0 elements in the factors.

Given an arbitrary matrix, there are known algorithms for determining whether it is a perfect elimination matrix. If the matrix is structurally symmetric (i.e., the nonzero

\*In the Proceedings of the 5th Grace Hopper Celebration of Women in Computing, October 2004, Chicago, IL.

<sup>†</sup>Work done while a postbaccalaureate researcher at Pomona College.

pattern of  $A$  is the same as that of  $A^T$ ), and the rows and columns must be permuted in the same way, then there are algorithms whose running time is linear in  $n$  and the number of nonzeros in  $A$  that determine whether  $A$  is perfect elimination [7, 10, 11, 12, 13]. If the rows and columns can be permuted independently, as one often has the freedom to do with a nonsymmetric matrix, an  $O(n^5)$  algorithm is described in [7], and an  $O(n^3)$  algorithm is described in [6].

The question addressed in this paper is whether nonsymmetric matrices from actual applications have nonsymmetric perfect elimination orderings. The surprising answer is “yes.” After summarizing the known algorithms for determining whether a nonsymmetric matrix is perfect elimination, this paper describes, to the best of our knowledge, the first implementation of the algorithm in [6]. The code is run on 180 matrices that come from a wide range of practical applications, resulting in the discovery that over 10% are, in fact, perfect elimination. Furthermore, of the ones that are not, many have a large number of entries that can be eliminated without introducing fill. Finally, possibilities for future work are discussed.

## 2 Algorithms

The known algorithms for determining whether a nonsymmetric matrix is perfect elimination rely on Theorem 1 from [8]. Although Theorem 1 is stated in terms of bipartite graphs, references such as [7] discuss the natural correspondence between bipartite graphs and  $(0, 1)$  matrices (i.e., matrices whose nonzero entries all have value 1). Throughout the remainder of this paper  $A_1$  is used to refer to the matrix with the same nonzero structure as  $A$ , but with nonzero values set to 1.

**Theorem 1** *If  $e$  is a bisimplicial edge of a perfect elimination bipartite graph  $G$ , then  $G - e$  is also a perfect elimination bipartite graph.*

In terms of matrices, Theorem 1 says that if an entry of a perfect elimination matrix can be used as a pivot without introducing fill, then the rest of the matrix can still be permuted in a way that generates no fill. In practice, the  $a_{ij}$  element in the matrix  $A$  can be used as a pivot without introducing fill if  $a_{ij} \neq 0$  and if, in  $A_1$ , every row with an element in column  $j$  majorizes row  $i$ . Informally, in a  $(0, 1)$  matrix, row  $i_1$  majorizes row  $i_2$  if row  $i_1$  has a nonzero element in at least every column that row  $i_2$  does.

### 2.1 An $O(n^5)$ algorithm

The authors of [8] noted that Theorem 1 immediately suggests an algorithm for determining whether a matrix has a perfect elimination ordering: repeatedly search for elements which create no fill when used as pivots, then delete them by zeroing out the row and column containing that element. The matrix is perfect elimination if and only if this can be repeated until the matrix is empty.

In [6], it is noted that the running time of this algorithm is  $O(n^5)$ , where  $n$  is the dimension of the matrix. In each of  $n$  iterations up to  $n^2$  elements are checked for eliminability; each check requires looking at up to  $n^2$  elements.

### 2.2 An $O(n^3)$ algorithm

In [6], Goh and Rotem prove the following lemma and use it to improve on the algorithm in [8].

**Lemma 1** *Let  $M = (m_{ij})$  be an  $n \times n$   $(0, 1)$  matrix representing a bipartite graph  $G = (X \cup Y, E)$ . Let  $l_i$  be the number of rows in  $M$  that majorize row  $i$  and  $s_j$  the sum of entries in column  $j$  of  $M$ . Then  $m_{ij} = 1$  and  $l_i = s_j$  if and only if the edge  $x_i y_j$  is a bisimplicial edge of  $G$ .*

In other words, if  $M = A_1$ , then using the nonzero element  $a_{ij}$  as a pivot creates no fill if and only if  $s_j$ , the number of nonzeros in column  $j$ , equals  $l_i$ , the number of rows that majorize row  $i$  in  $M$ . The reasoning is as follows: clearly  $s_j \geq l_i$ , since  $m_{ij}$  is nonzero and therefore any row that majorizes row  $i$  must have a nonzero value in column  $j$ . Furthermore,  $s_j \leq l_i$  since if  $s_j > l_i$ , then there is some row  $k$  such that  $m_{kj} \neq 0$ , yet for which  $m_{kr} = 0$  when  $m_{ir} \neq 0$ . This would mean using  $m_{ij}$  as a pivot would create fill by placing a nonzero value in  $m_{kr}$ . This shows that if  $m_{ij}$  creates no fill as a pivot, then  $s_j$  must equal  $l_i$ .

In Figure 3 an adapted version of the pseudocode in [6] is presented. The initial values of  $s_j$  are easily computed. The algorithm calculates the value of  $l_i$  for all  $i$  by first computing  $Q = A_1 A_1^T$ . Now  $q_{ij}$  is the number of columns in  $A$  with a nonzero entry in both row  $i$  and row  $j$ . This means  $q_{ii}$  is the number of nonzero entries in row  $i$ , so  $q_{ij} = q_{ji}$  if and only if row  $j$  majorizes row  $i$ . Hence  $l_i$  is the number of elements in row  $i$  of  $Q$  that are equal to  $q_{ii}$ . Note that  $M^{ij}$  denotes the matrix  $M$  with all entries in row  $i$  and column  $j$  set to 0.

In the worst case this is an  $O(n^3)$  algorithm. The matrix  $Q$  is computed only once, and is then updated in each iteration. The  $s$ -values and matrix  $M$  must also be updated in each iteration, and the  $l$ -values must be recalculated as each entry is deleted. The updates take  $O(n^2)$  time and are repeated up to  $n$  times, making the total running time of the algorithm  $O(n^3)$ .

## 3 Implementation

Now we turn to the implementation of the  $O(n^3)$  algorithm described above. We first explain how the sparse matrices given as input are stored, and then summarize our implementation and subsequent optimization decisions.

### 3.1 Storing sparse matrices

As noted in the introduction, given a sparse  $n \times n$  matrix, users generally do not store the values of all  $n^2$  elements in a two-dimensional array. Instead, they store only the values of the nonzero elements, together with information about where they are located in the matrix. The simplest storage format might be the *triplet* format, in which an entry of the form  $(i, j, v)$  means that  $a_{ij} = v$ .

In practice more compact representations are used. A standard format is the column-compressed format, sometimes referred to as the Harwell-Boeing format [9]. In this format a matrix is represented by three arrays. The first is `nzval`, which stores the nonzero values in the matrix, beginning with those in the first column, then those in the second column, and so on. The second is `rowind`, where `rowind[i]` contains the row index of the element whose value is stored in `nzval[i]`. The last is `colptr`, where `colptr[j]` gives the index of the first element in `rowind` that is in column  $j$ . Note that `colptr` is of size `n+1`, and that the value of the last element must be the total number of nonzero elements in the matrix. As the algorithm assumes the input is a  $(0, 1)$  matrix, the implementation uses only the `rowind` and `colptr` arrays.

```

PerfectElimination( $A$ )
1    $M = A_1$ 
2   isPerfectElimination  $\leftarrow$  true
3   compute  $Q = MM^T$ 
4   for  $j \leftarrow 1$  to  $n$  do
5      $s_j \leftarrow \sum_{i=1}^n m_{ij}$ 
6   while there exists an  $s_j \neq 0$  and isPerfectElimination do
7     for  $i \leftarrow 1$  to  $n$  do
8        $l_i \leftarrow$  the number of entries in row  $i$  of  $Q$  which are equal to  $q_{ii}$ 
9     if there is a nonzero entry  $m_{ij}$  such that  $s_j = l_i$  then
10      Compute the matrix  $D = (d_{kl})$  where  $d_{kl} = m_{kj} * m_{lj}$ ;
11       $Q \leftarrow (Q - D)^{ii}$  (*)
12      for  $k \leftarrow 1$  to  $n$  do
13         $s_k \leftarrow s_k - m_{ik}$ 
14       $s_j \leftarrow 0$ 
15       $M \leftarrow M^{ij}$  (**)
16    else isPerfectElimination  $\leftarrow$  false

```

Figure 3: Pseudocode from [6] describing an  $O(n^3)$  algorithm that determines whether a matrix is perfect elimination.

(\*)  $Q$  is now equal to  $(M^{ij})(M^{ij})^T$

(\*\*) This line is, incorrectly, omitted in [6].

For more information on triplet, column-compressed, and other common formats for storing sparse matrices, see [1].

### 3.2 Implementation and optimization details

The algorithm was implemented in C, making use of the I/O functions for matrices stored in column-compressed format provided at [2]. The process of calculating  $Q$  requires storage of size  $O(n + \text{nnz}(A) + \text{nnz}(Q))$  and takes time  $O(n^2 + n \cdot \text{nnz}(A))$ , where  $n$  is the dimension of  $A$ ,  $\text{nnz}(A)$  is the number of nonzero elements in  $A$ , and  $\text{nnz}(Q)$  is the number of nonzero elements in  $Q = AA^T$ . Once  $Q$  is created, it is updated in place. Storing the  $s$ - and  $l$ - values requires two integer arrays, each of size  $n$ .

With a few exceptions, noted here, our implementation follows the pseudocode given in Figure 3, originally from [6]. The first change is that, when calculating the  $l$ -values in line 8, we evaluate the columns of  $Q$  rather than the rows. Because  $MM^T = (MM^T)^T$ , we know  $Q$  is symmetric and so the operations are equivalent. In addition, because our matrices are stored in column-compressed format, it is more efficient to retrieve columns than rows. Note that we do not recalculate  $l$ -values for rows which have already been removed from  $Q$ .

Second, when updating  $Q$  in lines 10-11, we do not compute  $D$  and then subtract it from  $Q$ . Our code updates only those columns of  $Q$  with entries that have changed with the removal of the last pivot. More specifically, if the last element removed from  $M$  was in column  $j$ , then for each nonzero element  $m_{kj}$ , column  $k$  of  $Q$  must be updated. The nonzero elements in any given column  $c$  of  $M$  are those elements with indices between  $\text{colptr}[c]$  and  $\text{colptr}[c+1]-1$ .

Finding an eliminable element and making the updates necessary to remove it takes  $O(n + \text{nnz}(A) + \text{nnz}(Q))$  time, and is repeated up to  $n$  times. Thus the overall algorithm takes  $O(n^2 + n \cdot \text{nnz}(A) + n \cdot \text{nnz}(Q))$  time (which is  $O(n^3)$  in the worst case, when  $A$  and/or  $Q$  are dense).

## 4 Results and Analysis

We first tested the code on small sample matrices where it could be determined by hand whether each matrix was perfect elimination. We then moved on to a test suite consisting of 180 matrices from a wide variety of applications. These matrices were chosen from both the University of Florida sparse matrix collection [3] and the Matrix Market [2]. All tests were run on a 2 GHz Intel Xeon processor with 2 GBytes of memory.

### 4.1 Percent Perfect Elimination

Figure 4 shows the number of entries, as a percentage of the dimension  $n$ , that can be eliminated in each matrix without introducing fill. If the percentage is 100, then the matrix is perfect elimination. If the percentage is 50, then  $n/2$  elements can be eliminated without fill, before an element that does generate fill must be chosen.

Notice that 19 of the 180 matrices (over 10%) are in fact perfect elimination. Furthermore, these perfect elimination matrices come from a range of applications, including circuit simulation (add32), plasma physics (mhd416b), and linear programming (bp\_0). For these matrices, the perfect elimination ordering is generally not trivial: it is neither the original ordering of the matrix, nor is it the ordering returned by the popular nonsymmetric fill-reducing heuristic COLAMD [4].

Figure 4 further shows that almost half of the matrices contain 40%, or more, eliminable entries.

### 4.2 Effect of Pivoting

Although so far the emphasis has been on the nonzero structure of the matrices and not on the numerical values of their entries (i.e.,  $A_1$ , and not  $A$ ), we now change course. When factoring a nonsymmetric matrix, a technique known as *pivoting* is often needed for numerical stability.<sup>1</sup> Pivoting, dis-

<sup>1</sup>Pivoting is not needed for symmetric positive definite matrices. This is one reason the symmetric case is less complex than the nonsymmetric case.

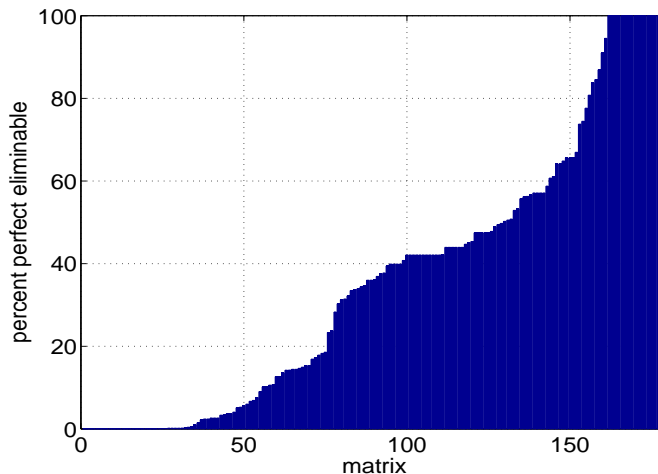


Figure 4: The number of entries, expressed as a percentage of the dimension, that can be consecutively eliminated without introducing fill.

cussed in most linear algebra textbooks, refers to moving large entries of a matrix to the diagonal during the factorization. Not pivoting can result in tiny entries on the diagonal, which is sometimes referred to as *pivot growth*. Unfortunately, while necessary for stability, pivoting means permuting the rows and/or columns of a matrix and therefore can destroy a precomputed perfect elimination ordering.

Our second set of tests looked at the pivot growth when using a perfect elimination ordering, and then looked at the amount of fill created when pivoting was used in combination with a perfect elimination ordering. More specifically, the matrices that were found to be perfect elimination were ordered using their perfect elimination ordering, and then factored using the package SuperLU [5]. We computed the factorizations both with and without partial pivoting and looked at both the pivot growth and at the number of nonzeros in the  $L$  and  $U$  factors.<sup>2</sup> The results are shown in Table 1.

In Table 1, when there are several perfect elimination matrices that belong to the same family of matrices (e.g., olm100, olm500, olm1000, and olm5000), only one is included. As expected for these perfect elimination matrices, without partial pivoting the number of nonzeros in  $L + U$  equals the number of nonzeros in  $A$ ; however, the pivot growth could occasionally be poor (as with the olm5000 matrix). With partial pivoting the pivot growth is generally improved, though sometimes at the cost of increased fill. Two facts stand out. First, except for the olm5000 matrix, the pivot growth is not terrible even without pivoting. Second, introducing partial pivoting generally does not increase the fill by very much, if at all.

To further emphasize the importance of using a fill-reducing ordering, we note that if no fill-reducing ordering whatsoever is used, and partial pivoting is allowed for stability, the number of nonzeros in  $L + U$  can be significantly larger than the number of nonzeros in  $A$ . For example,  $\text{nnz}(L+U)$  is 6012 for the shl\_400 matrix and 15268664 for the add32 matrix.

For more detailed results, including the names of the matrices and the percent to which each is perfectly eliminable,

<sup>2</sup>For SuperLU to give a precise number for  $\text{nnz}(L + U)$ , set the relaxation parameter `relax` in the `sp_ienv.c` file to 1.

see the table available at:

<http://www.cs.pomona.edu/~tzuyi/PerfectElimination>.

## 5 Conclusions and Future Work

This paper presents, to the best of our knowledge, the first implementation of an algorithm for determining whether nonsymmetric matrices are perfect elimination. Furthermore, extensive tests on matrices from a range of actual applications show that a surprising number are either perfect elimination, or are nearly so.

While this is an interesting result in and of itself, it would also be interesting to adapt this code for use as a general heuristic for reducing fill for nonsymmetric matrices. For example, a simple ordering heuristic might select an element to eliminate when  $s_j \approx l_i$ , in line 9 of the pseudocode in Figure 3, rather than only when  $s_j = l_i$ . Unfortunately, preliminary results suggest that while this type of heuristic can occasionally dramatically outperform a standard heuristic such as COLAMD [4], more frequently it does worse and sometimes much worse. The general patterns seem to hold as parameters in the heuristic are varied, which suggests that outperforming a heuristic such as COLAMD may require less obvious, and more significant, changes to the current code.

## References

- [1] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the solution of linear systems: Building blocks for iterative methods*. SIAM, 1994.
- [2] R. Barrett, R. Boisvert, J. J. Dongarra, R. Lipman, B. Miller, R. Pozo, and K. Remington. Matrix Market. Available at: <http://math.nist.gov/MatrixMarket>.
- [3] T. Davis. University of Florida sparse matrix collection. NA Digest, v.92, n.42, Oct. 16, 1994 and NA Digest, v.96, n.28, Jul. 23, 1996, and NA

matrix	n(A)	nnz(A)	without pivoting		with partial pivoting	
			nnz(L+U)	recip. piv. growth	nnz(L+U)	recip. piv. growth
add32	4960	23884	23884	1.0	23884	1.0
bp_0	822	3276	3276	1.0	3276	1.0
memplus	17758	126150	126150	1.0	126678	1.0
mhd416b	416	2312	2312	1.0	2524	.881
olm5000	5000	19996	19996	.000415	34954	.6
shl_400	663	1712	1713 (*)	1.0	1713	1.0
str_0	363	2454	2454	1.0	2454	1.0
tols4000	4000	8784	8784	.999	8784	.999

Table 1: The effect of pivoting on (reciprocal) pivot growth and fill for matrices with perfect elimination orderings.

(\*) This is the number of nonzeros in  $L + U$  as reported by SuperLU [5]. Tests in Matlab show that there are, in fact, only 1712 nonzeros in the factors of the shl\_400 matrix.

Digest, v.97, n.23, Jun. 7, 1997. Available at:  
<http://www.cise.ufl.edu/research/sparse/matrices/>.

- [4] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng. A column approximate minimum degree ordering algorithm. Technical Report TR-00-005, Department of Computer and Information Science and Engineering, University of Florida, October 2000.
- [5] J. W. Demmel, J. R. Gilbert, and X. S. Li. *SuperLU users' guide*, September 1999. Available at: <http://www.nersc.gov/~xiaoye/SuperLU/>.
- [6] L. Goh and D. Rotem. Recognition of perfect elimination bipartite graphs. *Inform. Process. Lett.*, 15(4):179–182, 1982.
- [7] M. C. Golumbic. *Algorithmic graph theory and perfect graphs*. Computer science and applied mathematics. Academic Press, New York, 1980.
- [8] M. C. Golumbic and C. F. Goss. Perfect elimination and chordal bipartite graphs. *J. Graph Theory*, 2(2):155–163, 1978.
- [9] I. S. Duff and R. G. Grimes and J. G. Lewis. *Users' guide for the Harwell-Boeing sparse matrix collection*, October 1992. Available at: <http://math.nist.gov/MatrixMarket/collections/hb.html>.
- [10] G. S. Leuker. Structured breadth first search and chordal graphs. Technical Report TR-158, Princeton University, Princeton, NJ, 1974.
- [11] B. S. Panda. New linear time algorithms for generating perfect elimination orderings of chordal graphs. *Inform. Process. Lett.*, 58:111–115, 1996.
- [12] D. J. Rose and R. E. Tarjan. Algorithmic aspects of vertex elimination. In *Proceedings of the 7th Annual ACM Symposium on the Theory of Computing*, pages 245–254, 1975.
- [13] D. J. Rose, R. E. Tarjan, and G. S. Leuker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comp.*, 5:266–283, 1976.