An AMD Preprocessor For Matrices With Some Dense Rows and Columns.

Joseph L. Carmen

Department of Computer and Information Sciences and Engineering
University of Florida, Gainesville, FL 32611

Abstract

A preprocessor for the Approximate Minimum Degree ordering algorithm (AMD) is presented. The primary goal of preprocessing is to reduce the ordering time of AMD on large sparse matrices with some very dense rows and columns. We show that with preprocessing, matrices that normally require large amounts of ordering time can be done significantly faster.

Keywords and Phrases.
approximate minimum degree ordering algorithm, sparse matrices, preprocessing algorithms.

1 Introduction

1.1 description. *The Approximate Minimum Degree Ordering Algorithm* (AMD) [1] for preordering a sparse symmetric positive definite matrix **M** prior to numerical factorization is used for a wide variety of scientific and engineering applications. AMD finds a permutation **P** so that the Cholesky factorization of $PMP^T = LL^T$ requires less work and storage (nonzeroes in **L**) than the Cholesky factorization of $M = LL^T$. The permuted matrix PMP^T from AMD normally takes much less time to compute the numerical factorization. One important exception that this paper addresses is when **M** contains one or more dense (or nearly dense) rows and columns.

If a matrix is not sparse, and simultaneously not extremely dense, it can be partitioned into four submatrices as shown in figure 1.

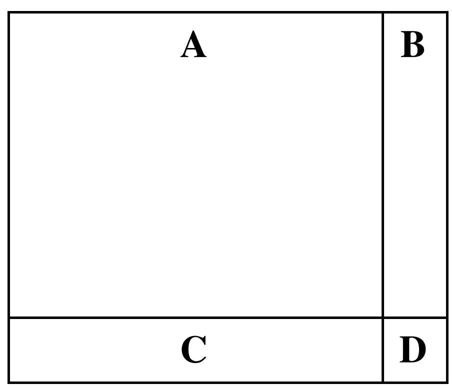

Figure 1: Partitioning a matrix

Where submatrix **A** is sparse and submatrices **B, C,** and **D** are dense (or nearly so). After partitioning, AMD rapidly orders submatrix **A**, and **D** is ordered by sorting the rows and columns according to the number of nonzeroes in **D**. Once both the submatrices **A** and **D** are ordered, then the user has the ordering for the entire matrix. For matrices with some dense rows this is much faster than preordering the entire matrix, and, occasionally, yields a better ordering.

1.2 Approximate Minimum Degree Algorithm. AMD uses a graph theoretical approach to represent a matrix. The nodes of the undirected graph are the diagonals of the matrix and the edges are the off-diagonal nonzeros. Thus, if $a_{ij} \neq 0$ and $i \neq j$ then there exists an edge from node *i* to node *j*. AMD selects nodes from the graph to produce an ordering such that the degree of each node selected is minimized. The degree of a node is the number of nodes that are adjacent to it. The degree represents the number of nonzeroes in a row. The maximum degree of a node is *n*-1, where *n* is the order of the matrix.

Once AMD selects a node, it eliminates the selected node from the graph. However, to eliminate the node from the graph, edges are added to the neighbors of the node so that the neighbors become a clique. In the formation of a clique, each neighboring node must be checked to see if it has an edge to the other neighboring nodes. If a neighboring node is dense (or fully connected), i.e. has *n*-1 edges, then ,in the worst case, *n*-1 comparisons are made. Since this process must be done for each node to be eliminated, i.e. *n* times, there will be at least *n*(*n*-1) comparisons.

2 The AMD Preprocessor Algorithm.

2.1 Introduction. The primary purpose of the preprocessor algorithm is reduce the ordering time of AMD by partitioning matrix **M** so that the partitioned matrix in figure 1 is formed. Appendix A contains the AMDpre algorithm (AMDpre).

The key variable in AMDpre is the value of the threshold used to determine when a row or column is dense and is to be placed in the submatrix **D**. A good value for this threshold is the square root of *n*. For most matrices with some dense rows and columns, this value for the threshold will yield the best ordering time than a higher value for the threshold. If a neighboring node *i* has at most $\sqrt{n}$ edges, then ,in the worst case, $\sqrt{n}$ comparisons are made. Since there are at most $\sqrt{n}$ nodes adjacent to node *i* there will be at least $\sqrt{n} * \sqrt{n}$ or n comparisons. The algorithm is written so the user can pass in a single value, *z*, that can increase or decrease the value of the threshold in relationship to the square root of *n*.

An alternative value for the threshold is to select those rows whose densities are significantly higher than the average density. Using this approach for linear programming problems, such as GUPTA1 (see table 1), a better ordering will result as explained in [4]. The set of matrices obtained from Gupta contained a few rows that are approximately two or three times more dense than the average row density. The permutation using this method usually yields a better flop count and lower fill-in from factorization for linear programming problems, however, it will not be as fast as using the square root of *n* for the threshold.

In either case, ordering the dense nodes by row density normally does not improve the flop count or fill-in. An alternative to this approach is to order the Schur complement ($D - CA^{-1}B$) instead of **D** [5].

2.2 Buckets. In section 1 of AMDpre, a bucket sort is used to sort all the nodes according to their row densities, or degrees, that are greater than the value of the threshold. The node is simply placed in a bucket corresponding to its degree. The primary function of section 1 is to select the initial nodes for the submatrix **D** (see figure 1). The rest of the nodes are considered sparse and are placed in submartix **A**.

The primary function of section 2 of the algorithm, is to reduce the size of submatrix **D** made in section 1. AMDpre selects a node from the bucket with the highest number. There may be more than one node in a bucket. The nodes in the bucket are in a singly linked list. AMDpre uses a control variable to indicate that there are more nodes remaining in the current bucket.

When a node in the list is selected, it is deleted from the list, essentially removing it from the bucket, and it is checked to see if its degree was changed. This is a simple matter of comparing its value in the *len* array with the bucket number it was in. The *len* array in AMDpre is used to keep a current record of the density of each node. If the value of the *len* array is less than the bucket number, then the degree of that node must have been changed. If the degree is below the value of the threshold then the node is placed into the submatrix **A**. Otherwise, the node is placed in the bucket equal to it current degree. It is very possible for a node to be moved from one bucket to another bucket many times before it is permanently placed in either submatrix **A** or submatrix **D.** If the density of the node equals the bucket number then the node is placed in submatrix **D**. AMDpre then reduces the degree of all the adjacent nodes by 1.

2.3 Mapping function. Once the nodes for submatrix **D** have been selected and ordered, a record is made of what nodes are in **A** and in **D.** This is the primary function of section 3. This is called the mapping. This mapping is needed because all the numbers in submatrix **A** have to be sequentially renumbered. For example, given a matrix of size 5 and AMDpre finds that node 2 is dense. It is then placed in **D**. The nodes for submatrix **A** will be 1, 3, 4, and 5. The node for submatrix **D** is 2. The nodes in **A** will need to be renumbered. So the mapping is: 1 = 1, 2 = 3, 3 = 4, and 4 = 5. The 2 in **D** is mapped to 5.

2.4 Sending the submatrix. Section 4 eliminates the nodes in submatrix **D** from the *iw* array. The *pe* array, which contains pointers to the *iw* array, is updated and the *len* array is also updated to reflect only the nodes that are in submatrix **A.** AMD uses the *iw* array to construct the graph of the matrix. Section 5 saves the original value of *n* then calls AMD with the new values of *n, iw(n), pe(n), and len(n).*

2.5 Returning the permutation. After AMD returns the ordering, section 6 restores the original numbering to the submatrix **A.** Section 7 makes the inverse permutation of the ordering and returns both the permutation and its inverse to the calling program.

2.6 A general AMDpre run-time analysis. A casual glance at the algorithm may cause one to think that section 2 may be the longest running function. On average section 4 will take the longest. The run time of AMDpre is $O(|A|)$, where $|A|$ is the number of non-zeroes in the matrix.

Section 4 contains a for-loop looping n times. On each loop, AMDpre looks through a part of the node adjacency list. The list has the length of *iwlen*, where part of that array contains all the non-zeroes in the matrix. The total non-zeroes in the list is *pfree* -1. The part of the list section 4 looks at during a single loop is the nodes that are adjacent to the current node. AMDpre will only look at an entry in the list once in the entire program. So, if it looked at a node's adjacency list in section 2 it will not be looking at that same node's list in section 4. Therefore, the run time of section 4 is $O(|A|)$.

3 Sparse matrices Run time analysis.

3.1 Sparse Matrices used. The matrices in Table 1 are a sampling of 520 matrices used during the testing phase of AMDpre. Only the matrices in Table 1 will be used in this paper.

Matrix	Matrix order	number of nonzeros	description
BCSSTK29	13992	619488	Structural engineering
BCSSTK30	28924	2043492	Structural engineering
BCSSTK31	35588	1181416	Structural engineering
BCSSTK32	44609	2014701	Structural engineering
CT20STIF	52329	2698463	Structural engineering
CRYSTK03	24696	1751178	Structural engineering
EX11	16614	1096948	Test matrix from FIDAP
EX19	12005	259879	Test matrix from FIDAP
NASASRB	54870	2677324	Shuttle Rocket Booster
AF23560	23560	484256	Linear programming problem
LHR34	35152	764014	Light hydrocarbon recovery
BBMAT	38744	1771722	2D airfoil exact Jacobian
VENKAT50	62425	1780217	Structural engineering
APPU	14000	1853104	NASA test random matrix
WANG3	26064	177168	Discretized electron continuity
GUPTA1	31802	2164210	Linear programming problem
GUPTA2	62064	4248286	Linear programming problem
GUPTA3	16783	9323427	Linear programming problem

Table 1: Selected sparse matrices.

3.2 AMD and AMDpre ordering time for sparse matrices. Table 3 lists the ordering times for the selected matrices. Column 3 is the ordering time of AMD without any attempt at preprocessing, and column 4 is AMDpre. For most of the matrices no preprocessing is necessary

Matrix	Matrix order	AMD time	AMDpre time	Rows torn
BCSSTK29	13992	.19	.19	0
BCSSTK30	28924	**.47**	.62	96
BCSSTK31	35588	.71	.71	0
BCSSTK32	44609	**.64**	.80	4
CT20STIF	52329	.91	.91	0
CRYSTK03	24696	.44	.44	0
EX11	16614	.41	.41	0
EX19	12005	.16	.16	0
NASASRB	54870	**.98**	1.19	12
AF23560	23560	1.19	1.20	0
LHR34	35152	2.22	2.22	0
BBMAT	38744	3.49	3.50	0
VENKAT50	62425	.59	.60	0
APPU	14000	10.03	**4.56**	6497
WANG3	26064	.77	.77	0
GUPTA1	31802	563.40	**.72**	676
GUPTA2	62064	1655.07	**4.29**	1193
GUPTA3	16783	147.66	**.83**	1347

Table 3: Sparse matrices ordering times (in seconds)

For the matrix BCSSTK30, 96 dense rows where removed. Note, however, that this removal did not yield a positive change in run time. This is because those 96 rows have degrees that were not significantly higher than the average density. The average density for BCSSTK30 is 69.7 with a standard deviation of 31.7. While is possible to change the value of the threshold, getting a better ordering time for BCSSTK30 is not. This is because those 96 rows each have densities that are close to the threshold and not significantly high enough to decrease AMD ordering time.

The NASASRB matrix has a different distribution. The NASASRB has an average density of 47.8 with a standard deviation of 8.5. The square root of n for NASASRB is 234 which is 21 standard deviations from the average. Removing 12 rows from 54870, or .0002 percent, caused an increase in the ordering time. Compared to the size of this matrix these 12 rows are not very dense and, therefore, this matrix needs no preprocessing.

One of the more interesting matrices in table 3 is the APPU, a random matrix with a relatively normal distribution and a very high average density. The average density for the APPU is 255.6 which is more than twice the square root of n, which is 118. This is a good example of what the preprocessor does. Since the average density is above the threshold, the AMDpre removed 6497 rows, nearly half the rows, with a 54 percent reduction of ordering time.

All three Gupta matrices are difficult to order without AMDpre. It took AMD 28 minutes to order GUPTA2. AMDpre, on the other hand, removed only 2 percent of the rows (1193) and cut the ordering time to just over 4 seconds! The ordering time reductions of the other two Gupta matrices are almost as dramatic. All three Gupta matrices have some dense rows that are several times more dense than the average.

Although statistics can tell you quite a bit about a matrix, statistics cannot accurately predict the ordering time for a given matrix. As the next section will show, just adding one dense row, increasing the average density by just 1, can cause the ordering time for a sparse matrix to increase dramatically. In other words, knowing the average density and distribution of the rows and columns in a matrix will not indicate the existence of a very dense row or column.

Matrix	Matrix order	AMD time	AMDpre time	Rows torn
BCSSTK29	13993	1.35	.24	1
BCSSTK30	28925	5.04	.62	97
BCSSTK31	35589	13.98	.81	1
BCSSTK32	44610	14.24	.80	5
CT20STIF	52330	17.02	1.12	1
CRYSTK03	24697	2.21	.58	1
EX11	16615	1.62	.50	1
EX19	12006	2.99	.17	1
NASASRB	54871	15.88	1.21	13
AF23560	23561	21.21	1.68	1
LHR34	35153	49.70	2.30	1
BBMAT	38745	23.27	4.69	1
VENKAT50	62426	21.12	.88	1
WANG3	26065	48.02	1.06	1

Table 4: Sparse matrices ordering times (in seconds) with a dense row and column added.

3.3 AMD and AMDpre ordering time for matrices with a dense row and column added. Nearly 100 matrices, of the 520 matrices used during testing, were also tested by adding a dense row and column. This was done in part to see what effect a dense row and column has on AMD ordering time and to test the effectiveness of AMDpre. The matrices of table 3 each had a dense row and column added. AMDpre had no problem detecting the dense row, or rows, and no problem partitioning the matrix. In general, the resulting submatrix **A** is the same as the original matrix from table 3, and the submatrix **D** is just the dense node, or nodes. It was not necessary to add a dense row and column to APPU or the Gupta matrices because those were already dense enough.

Comparing column 3 of both table 3 and 4, the ordering time increased by an average factor of 18.1. This is the effect one dense row has on the AMD and is convincing evidence for the use of AMDpre.

Comparing column 4 of both table 3 and 4, only two ordering times stayed the same, BCSSKT30 and BCSSTK32. The rest of the times increased an average of 29.8 percent. This increase is roughly the time it took AMDpre to partition the matrix.

Comparing columns 3 and 4 of table 4, the average reduction in ordering time is 88.7 percent. The largest reduction in run time is the matrix LHR34, which went from 49.7 sec to 2.3 sec. This is a 95.4 percent reduction. Also note, from table 3, it took 2.22 sec for AMDpre which means that AMDpre consumed only .08 seconds as a result of the addition of one dense row. The largest reduction in percent change is the BCSSTK29 which is 97.8 percent. The smallest reduction in percent change is EX11 with a reduction of 69.1 percent.

As pointed out in section 1, and as evidenced by table 4, the AMD utilizes up to 97 percent of its computing time dealing with just one dense row. It is completely obvious that preordering is necessary in these matrices. However, in many cases, knowing the distribution of the row densities before hand is impossible. As seen in table 3, a matrix may have a row density greater than the threshold and not need preprocessing. AMDpre will yield positive results if there is one or more extremely dense rows and columns (as in table 4) or many slightly dense rows and columns (as with the Gupta matrices).

Matrix	AMD			AMDpre			
	time	fill-in	flops	time	fillin	flops	rows torn
BCSSTK29	.19	1.66	395.1	.19	1.66	395.1	0
BCSSTK30	**.47**	**3.76**	**899.7**	.62	3.99	1091.5	96
BCSSTK31	.71	5.46	2799.5	.71	5.46	2799.5	0
BCSSTK32	**.64**	5.06	1051.0	.80	**5.06**	**1050.5**	4
CT20STIF	.91	11.10	7983.7	.91	11.10	7983.7	0
CRYSTK03	.44	11.60	9851.2	.44	11.60	9851.2	0
EX11	.41	5.87	3322.2	.41	5.87	3322.2	0
EX19	.16	.29	10.7	.16	.29	10.7	0
NASASRB	**.98**	**11.86**	**4692.7**	1.19	11.92	4771.2	12
AF23560	1.19	4.07	1262.1	1.20	4.07	1262.1	0
LHR34	2.22	2.86	500.7	2.22	2.86	500.7	0
BBMAT	3.49	22.52	20410.	3.50	22.52	20410.	0
VENKAT50	.59	5.79	1144.3	.60	5.79	1144.3	0
APPU	10.03	87.61	768.4	**4.56**	**87.61**	**767.4**	6497
WANG3	.77	5.57	5207.8	.77	5.57	5207.8	0
GUPTA1	563.40	**2.02**	**300.1**	**.72**	2.05	301.1	676
GUPTA2	1655.1	**5.83**	1875.1	**4.29**	6.24	**1850.0**	1193
GUPTA3	147.66	5.70	2652.3	**.83**	**5.59**	**2602.6**	1347

Table 5: Sparse matrices resulting fillins and flops (in millions) and ordering time (in seconds).

4 Floating point operations (flops) resulting from AMD and AMDpre preordering.

4.1 Fill-in. In this section, the ordering is analyzed. When Gaussian elimination or Cholesky decomposition is performed, new nonzeroes will be created called fillins. The primary goal of AMD is, in the best case, to minimize the fillin. Unfortunately, this problem is NP-complete [1].

There are many reasons why fillin is not desirable. Foremost is storage allocation. The amount of memory is finite and fillin maybe very large. Allocating sufficient memory beforehand may not be possible. Another reason is the computer time required to do the factorization increases rapidity with increase in fillin. The computer time is measured in floating point operations or flops.

4.2 Comparing AMD and AMDpre. In table 5, columns 2 and 5 are from table 3, columns 3 and 6 are the resulting fillin, columns 4 and 7 are the floating point operations per second required for Choleski decomposition, and the last column is the total rows removed by AMDpre.

In majority of the matrices tested and the selected matrices in table 4, there is not a significant change fillin. AMDpre on the average does not adversely effect the fillin. Matrices BCSSTK30 and NASASRB are examples of significant fillin caused by preprocessing. With matrix BCSSTK32 more time was spend, but a slightly better fillin was achieved. In all three Gupta matrices, the ordering time was better, however, fillin was not improved.

5 Conclusion.

We have described a preprocessing algorithm based on the technique of partitioning the matrix prior to finding a permutation matrix. With the speed of many computers available today and the increase in memory used by the cpu, many large sparse matrices that used to take days to preorder now only take seconds. AMD is primarily concerned with sparse matrices and is not designed to process a matrix with one or more very dense rows and columns in a timely manner. Preprocessing, or filtering out, these rows and columns now allows AMD to preorder a larger number of matrices than with out preprocessing.

Appendix A. The AMDpre algorithm.

section 1. make hash table

```
degree = Z * √n
head(1 .. n) = 0; next(1 .. n) = 0
for i = 1 to n do
        deg = len(i)
        if (deg > dense) then
                ** place into hash bucket
                if (head(deg) = 0)
                        head(deg) = i
                else
                        number = head(deg)
                        while (next(number) != 0) do
                                number = next(number)
                        end while
                end if
        end if
end for
```

section 2. construct submatrix B

```
lastnode = n
current = n
dense = dense + 1
while (current >= dense) do
        top = 0
        ** find last node in the bucket
        number = head(current)
        node = number
        while (number > 0) do
                pnum = node
                node = number
                number = next(node)
                top = 1
        end while
```

```
        ** remove node from list
        if (top = 1) then next(pnum) = o
        deg = len(node)
        if (deg >= dense) then
                if (deg < current) then
                        ** re-insert in lower hash bucket
                        if (head(deg) = 0)
                                head(deg) = i
                        else
                                number = head(deg)
                                while (next(number) != 0) do
                                        number = next(number)
                                end while
                        end if
                else
                        ** place node in submatrix B
                        last(lastnode) = node
                        lastnode = lastnode + 1
                        len(node) = 2*n
                        pnum = pe(node + 1) - 1
                        if (node = n) then pnum = pfree - 1
                        ** decrement adjacent nodes by 1
                        for i = pe(node) to pnum do
                                len(iw(i)) = len(iw(i)) - 1
                        end for
                end if
        end if
        if (top == 0) then current = current - 1
end while

section 3. make the mapping index array
lastnode = n
current = 1
for i = 1 to n do
```

```
        if (len(i) < dense) then
                ** nodes in submatrix A
                elen(i) = current
                mapping(current) = i
                current = current + 1
        else
                ** nodes in submatrix B
                elen(i) = lastnode
                mapping(lastnode) = i
                lastnode = lastnode + 1
        end if
end for
```

section 4. construct submatrix A (including non-zeroes in A).

```
current = 1
node = 1
for i = 1 to n
        if (elen(i) < lastnode)
                ** construct the new iw array
                pnum = pe(i)
                pe(node) = current
                nextnum = pe(node + 1) - 1
                if (i = n) then nextnum = pfree - 1
                for j = pnum to nextnum do
                        if (elen(iw(j) < lastnode) then
                                iw(current) = elen(iw(j))
                                current = current + 1
                        end if
                end for
        ** place new degree length in len
        len(node) = current - pe(node)
        node = node + 1
end for
```

section 5. send submatrix A to the AMD

```
ntemp = n
pfree = current
n = lastnode

call AMD
```

section 6. map nodes in A to original matrix values

```
n = ntemp
for i = 1 to lastnode do
        last(i) = mapping(last(i))
end for
```

section 7. make inverse permutation array

```
for i = 1 to n do
        elen(last(i)) = i
end for
end program
```

Appendix B. References

1. Timothy A. Davis, Patrick Amestoy, and Iain Duff. *An approximate minimum degree ordering algorithm*. Technical Report TR-94-039, Computer and Information Sciences Department, University Of Florida, Gainesville, FL, 1994.

2. I.S. Duff, A.M. Erisman, and J.K. Reid, *Direct methods for Sparse,* London: Oxford Univ. Press, 1986.

3. A. George and J.W.H. Liu, *A fast implementation of the minimum degree ordering algorithm using quotient graphs,* ACM Transtions on Mathematical Software, 6 (1980), pp. 337-358.

4. Anshul Gupta. *Fast and effective algorithms for graph partitioning and sparse matrix ordering.* IBM Research Report RC 20496, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1996.

5. Gene H. Golub and Charles F. Van Loan, *Matrix Computations,* second edition, The Johns Hopkins University Press, Baltimore, Maryland, 1989.