Minimum Degree Reordering Algorithms: A Tutorial

Stephen Ingram*

University of British Columbia

1 Introduction: Cholesky Factorization and the Elimination Game

The problem of matrix inversion is central to many applications of Numerical Linear Algebra. When the matrix to invert is dense, little can be done to avoid the costly $O(n^3)$ process of Gaussian Elimination. When the matrix is symmetric, one can use the Cholesky Factorization to reduce the work of inversion (still $O(n^3)$, but with a smaller coefficient). When the matrix is both sparse and symmetric, we have even more options. An entire universe of approximation algorithms exists to take advantage of the structure of sparse symmetric matrices, but some applications still require computing the "true" inverse. In these cases we must once again fall back on Cholesky, but now we use a variant called Sparse Cholesky and the amount of work required to invert the matrix has changed. Slight alteration in the ordering of our equations and unknowns in our sparse, symmetric problem yields vastly different Sparse Cholesky runtimes. Why?

This is the phenomenon of *fill-in*. Figure 1 shows the process at work. Each step of Cholesky eliminates a row and a column from the matrix. Nonzero elements of our row a above the diagonal lead to new nonzero elements in those rows which have nonzeros in the column a. These new nonzeros are the so called fill-in and they spell trouble for the Sparse Cholesky algorithm in the form of more floating point operations. A remarkable phenomenon is that fill-in depends on the order in which we position the elements of the matrix. Figure 2 shows the same matrix as figure 1 but with the first and last columns and rows interchanged. This second configuration creates no fill in and Sparse Cholesky finishes faster. The task now seems obvious: find the ordering of the columns and rows of the matrix that creates the least fill-in. If we could do this, then we could minimize the work to invert any sparse, symmetric matrix by reordering using permutation matrices.

Bad news arrives from the world of graph theory but to understand it, we must recast the fill-in process in the language of graphs. Any symmetric matrix corresponds to an undirected graph called the elimination graph (see figure 3). To build the elimination graph, create a vertex for every row and then for every row a that has a nonzero above the diagonal in the column b, construct an edge between the vertices a and b. The act of eliminating a row (and creating the resulting fill-in) using Cholesky corresponds to removing a vertex a and its corresponding edges, then forming a *clique* from the former neighbors of a. The fill-in corresponds

*e-mail: sfingram@cs.ubc.ca

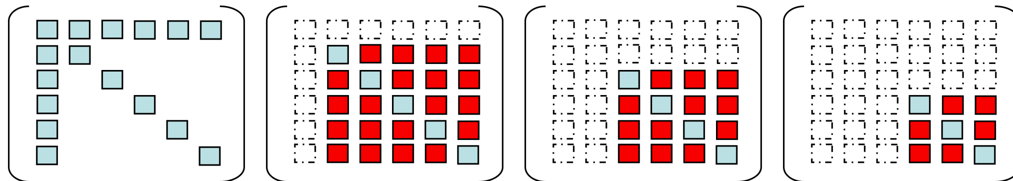

Figure 1: Four steps of elimination on a simple matrix. Fill-in is created after the first step. At far left, blue boxes represent the original nonzero elements of the matrix. In the successive images, red boxes represent the new nonzero elements generated from elimination and white boxes represent elements eliminated by Sparse Cholesky.

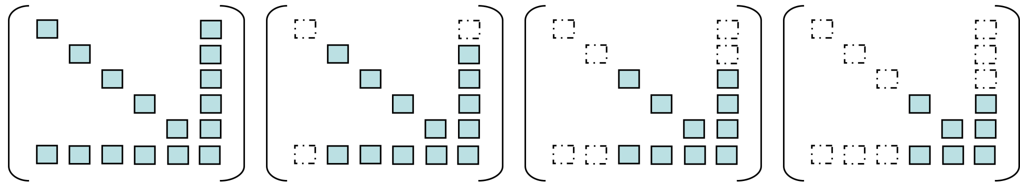

Figure 2: Similar to figure 1 except the first and last row and column have been swapped. No fill-in is generated during elimination.

to the edges created to form the clique. When we continue this graph operation until no vertices remain to eliminate, it is called the *elimination game*. Note that graph representation of the matrix models the nonzero structure and not the actual numerical answers of our matrix inversion. Thus the elimination game is a good model only for determining the work required to invert the matrix, not actually inverting the matrix.

The bad news is then this: *determining the ordering of nodes in the elimination game that results in the minimum number of new edges is NP-Hard*[4]. This implies that minimizing the work to invert any sparse, symmetric matrix is more costly than inverting any ordering of the matrix. The good news is that we can approximate this minimum using graph-based heuristics. One such heuristic is to always select the vertex with minimum degree [2]. The algorithms built around this simple heuristic are the subject of this paper.

2 Minimum Degree Reordering

It is surprising that simply eliminating the vertex with the smallest degree is so effective. It is also surprising that such a simple heuristic conceals a great deal of complexity. The first complication is in its storage requirements. The second complication comes from redundant computations.

2.1 Quotient Graphs

Using the minimum degree policy in the elimination game is trivial to implement. One glaring problem with this idea, however, is that elimination graph edges are created and stored *explicitly*. In other words, a new edge must be created for every instance of fill-in. For some problems this can mean millions or perhaps billions of newly created edges and the storage requirements of the algorithm become infeasible. To remedy this, the algorithm stays the same but *quotient graphs* replace elimination graphs.

Rather than create and store edges explicitly, quotient graphs create edges *implicitly*, eliminating the need for extra storage. The strategy is to introduce a special type of vertex. We no longer remove vertices when we eliminate them during the game. Now they are transformed into a special type of vertex called an *enode*. Two neighboring enodes are always merged. When we determine the degree of a given node, we count the set of neighbors and if a neighbor is an enode, we count its neighbors too (without double counting).

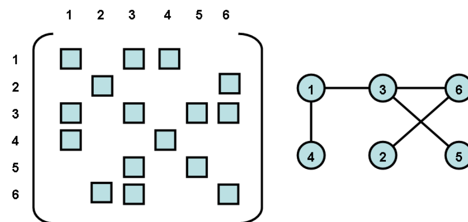

Figure 3: A symmetric matrix and its corresponding elimination graph.

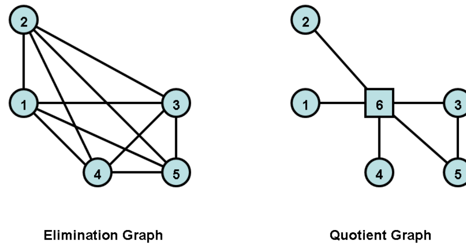

Figure 4: The elimination of vertex 6 with neighbors 1 through 5 yields the two equivalent graphs above. In the elimination graph, new edges must be constructed to make the neighbors of vertex 6 a clique. In the quotient graph, vertex 6 becomes an enode and the new edges are represented implicitly.

Compare the elimination step illustrated in figure 4 on an elimination graph and the equivalent quotient graph. The important thing to note is that the two graphs are equivalent in terms of vertex degrees, but that the quotient graph never requires more storage than the original quotient graph [1] making the algorithm's storage requirements tractable.

2.2 Indistinguishable Vertices and Mass Elimination

An interesting thing happens when two equations have identical nonzero coefficients. In the elimination graph this corresponds to two vertices that are neighbors and also have identical neighbors. These two vertices, which have the same degree, are called *indistinguishable* [2]. If you eliminate any vertex, then indistinguishable vertices remain indistinguishable. Also, if it is time to eliminate any indistinguishable vertex, then the other vertices will immediately follow next to be the vertex of minimum degree.

Rather than following a long, inevitable chain of vertex eliminations, it is more computationally efficient to merge indistinguishable vertices and eliminate them all in a single step. We accomplish this by attaching a weight to vertices. A regular vertex has a weight of size 1. If we merge two indistinguishable vertices, the resulting vertex has weight 2. We increment the weight with every new merge. These weights allow us to properly determine the degrees of a vertex. Eliminating a group of merged vertices is called *mass elimination*.

2.3 Computational Cost

Because indistinguishable vertices don't arise in every elimination game, we must run $|V|$ rounds in the worst case (where V is the set of vertices). The costliest operation of each round is the *degree-update* which calculates the degrees of each of the neighbors of the eliminated node a. The following pseudocode describes degree-update:

```
‘‘compute reachable set r’’
for each neighbor b of a
    if b == enode
        for each neighbor c of b
            add c to r if it isn’t already there
    else
        add b to r if it isn’t already there
    end
end
‘‘recalculate degrees of reachable set without overcounting’’
for each element b of r
    b.degree = 0
```

```
    for each neighbor c of b
        if c == enode
            for each neighbor d of c
                b.degree++
        else
            b.degree++
        end
    end
end
```

The nested structure of the degree calculations makes runtimes dependent on the density of the graph (and thus the underlying matrix). The worst-case requires a $O(|V|^2|E|)$ runtime [4] (E is the set of edges). Because we assume sparsity, runtimes are usually much better than this pessimistic bound.

3 Practical Minimum Degree Algorithms

The Minimum Degree algorithm is not typically used in practice because we can improve the algorithm further. Improvements come in two flavors but both focus on the degree-update. First is *Multiple Minimum Degree* which seeks to reduce the number of degree-updates. Second is *Approximate Minimum Degree* which seeks to reduce the computational cost of the degree-update.

3.1 Multiple Minimum Degree Reordering

In an elimination game, many vertices may have the minimum degree. Which one should be chosen? Numerous tie-breaking strategies exist [2], but Multiple Minimum Degree or MMD suggests choosing an independent set. That is, we choose a set of nodes who all have the minimum degree, but are not neighbors. Eliminating any of them will not effect the degree of the others and so we can, as in the case of indistinguishable nodes, eliminate them all at once. The phenomenon of a large independent set of minimum degree nodes arises frequently in large systems, especially those arising from finite differences.

The runtime bound remains the same [4], however the degree update is performed fewer times and so the runtimes are frequently much better than Minimum Degree. Another surprising consequence is that Multiple Minimum Degree reorderings usually result in fewer nonzeros.

3.2 Approximate Minimum Degree Reordering

Approximate Minimum Degree or AMD uses a heuristic to solve a heuristic. It computes only an approximation of the degree of a vertex [1]. We first assign weights to enodes defined to be the sum of the weights of the vertices adjacent to it. The approximation is a tight upper bound of the degree and results from the following heuristic: *The degree of a vertex cannot be greater than the sum of the weights of its neighbors* (both vertices and enodes). This approximation of the degree of a vertex a is tightened by subtracting a value related to the overlapping sets of vertices adjacent to the neighboring enodes of a. This minimizes overcounting the degree of a.

AMD is substantially faster than the other reordering algorithms because the approximation reduces the runtime complexity. We must still update the reachable set of vertices, however we need only examine their immediate neighbors. This results in a runtime of $O(|V||E|)$. AMD also typically yields fewer nonzeros than both MMD and MD.

4 Sample Results

All three algorithms were implemented in the prefuse visualization toolkit [3] for demonstration purposes. A drawback of the toolkit is the inefficiency of the graph data structures. Table 1 shows runtimes and nonzeros

for the prefuse implementations as well as implementations in Matlab[1]. The prefuse implementation of AMD is actually slower than MMD which should not be the case. Table 1 also illustrates the number of degree-update iterations (total number of nodes the degree-updates touched) between these two algorithms. If the slowest part of the algorithm is the degree update, then AMD makes a big difference in runtime cost. Unfortunately, this is not the case in the prefuse implementations. Finally, we compare MMD and AMD on a very large matrix. Table 2 shows a disparity in over a million nonzeros in favor of AMD.

	prefuse			Matlab	
	MD	MMD	AMD	symmmd	symamd
milliseconds	3421	2060	2504	780	310
nonzeros	70591	71048	69298	66026	65162
degree-update-iters		3805376	70575		

Table 1: Millisecond runtimes and number of nonzeros for prefuse and Matlab implementations of Minimum Degree Reorderings. All algorithms operated on a 64×64 2D Poisson matrix. Also included are the number of degree-update iterations between MMD and AMD.

	symmmd	symamd
milliseconds	3375	1188
nonzeros	10743767	9532314

Table 2: Runtimes and number of nonzeros between MMD and AMD on a very-large (262144×262144) 2D Poisson matrix.

5 Conclusion

Which algorithm should we prefer? Problems have been constructed where MMD is superior to AMD [1] in terms of nonzeros and even runtimes. However these instances are uncommon, the improvement in nonzeros is never over 9%, and the improvement in runtimes is only happens on problems with small matrices. On larger problems, AMD is shown to be an order to several orders of magnitude faster than MMD [1]. As of the writing of this paper, no heuristics have been published to determine when MMD should be used over AMD. Given these empirical observations and its low runtime complexity, AMD should be the minimum degree reordering algorithm of choice for large, sparse problems requiring an exact inverse.

References

[1] Patrick R. Amestoy, Timothy A. Davis, and Iain S. Duff. *An Approximate Minimum Degree Ordering Algorithm.* SIAM Journal on Matrix Analysis and Applications, 17 (1996), pp. 886-905.

[2] A. George and J. W. H. Liu. *The evolution of the minimum degree ordering algorithm.* SIAM Review, 31(1):1–19, 1989.

[3] J. Heer, S. K. Card, J. A. Landay, *prefuse: a toolkit for interactive information visualization.* In CHI 2005, Human Factors in Computing Systems, 2005.

[4] P. Heggernes, S. C. Eisenstatz, G. Kumfert, A. Pothen. *The computational complexity of the Minimum Degree algorithm.* Proceedings of 14th Norwegian Computer Science Conference, NIK 2001, University of Troms, Norway.

[1] ©1994-2006 by The MathWorks, Inc.