An Object-Oriented Collection of Minimum Degree Algorithms

Design, Implementation, and Experiences*

Gary Kumfert[1] and Alex Pothen[1,2]

[1] Department of Computer Science, Old Dominion University
[2] ICASE, NASA Langley Research Center

Abstract. The multiple minimum degree (MMD) algorithm and its variants have enjoyed 20+ years of research and progress in generating fill-reducing orderings for sparse, symmetric positive definite matrices. Although conceptually simple, efficient implementations of these algorithms are deceptively complex and highly specialized.
In this case study, we present an object-oriented library that implements several recent minimum degree-like algorithms. We discuss how object-oriented design forces us to decompose these algorithms in a different manner than earlier codes and demonstrate how this impacts the flexibility and efficiency of our C++ implementation. We compare the performance of our code against other implementations in C or Fortran.

1 Introduction

We have implemented a family of algorithms in scientific-computing — traditionally written in Fortran77 or C — using object-oriented techniques and C++. The particular family of algorithms chosen, the Multiple Minimum Degree (MMD) algorithm and its variants, is a fertile area of research and has been so for the last twenty years. Several significant advances have been published as recently as the last three years. Current implementations, unfortunately, tend to be specific to a single algorithm, are highly optimized, and are generally not readily extensible. Many are also not public domain.

Our goal was to construct an object-oriented library that provides a laboratory for creating and experimenting with these newer algorithms. In anticipation of new variations that are likely to be proposed in the future, we wanted the code to be extensible. The performance of the code must also be competitive with other implementations.

These algorithms generate permutations of large, sparse, symmetric matrices to control the work and storage required to factor that matrix. We explain the

* This work was supported by National Science Foundation grants CCR-9412698 and DMS-9807172, by a GAANN fellowship from the Department of Education, and by NASA under Contract NAS1-19480 while the second author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681-0001

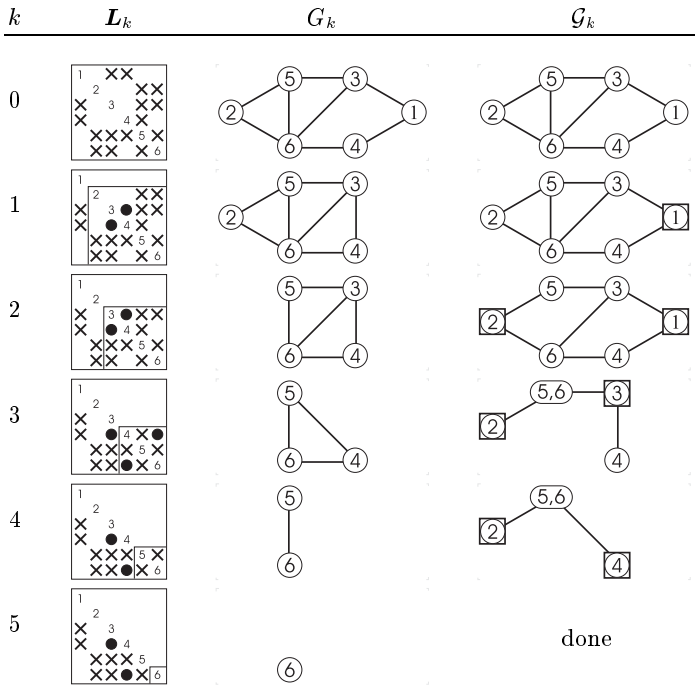

Fig. 1. Examples of factorization and fill. For each step, k, in the factorization, there is the nonzero structure of the factor, L_k, the associated elimination graph, G_k, and the quotient graph $\mathcal{G}_k$. The elimination graph consists of vertices and edges. The quotient graph has edges and two kinds of vertices, supernodes (represented by ovals) and enodes (represented by boxed ovals).

details of how work and storage for factorization of a matrix depends on the ordering in Sect. 2. This is formally stated as the *fill-minimization* problem. Also in Sect. 2, we review the Minimum Degree algorithm and its variants emphasizing recent developments. In Sect. 3 we discuss the design of our library, fleshing out the primary objects and how they interact. We present our experimental results in Sect. 4; examining the quality of the orderings obtained with our codes, and comparing the speed of our library with other implementations. The exercise has led us to new insights into the nature of these algorithms. We provide some interpretation of the experience in Sect. 5.

2 Background

2.1 Sparse Matrix Factorization

Consider a linear system of equations $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, where the coefficient matrix $\boldsymbol{A}$ is sparse, symmetric, and either positive definite or indefinite. We know $\boldsymbol{A}$ and $\boldsymbol{b}$ in

advance and must solve for $\boldsymbol{x}$. A direct method for solving this problem computes a factorization of the matrix $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{B}\boldsymbol{L}^{\mathrm{T}}$, where $\boldsymbol{L}$ is a lower triangular matrix, and $\boldsymbol{B}$ is a block diagonal matrix with 1×1 or 2×2 blocks.

The factor $\boldsymbol{L}$ is computed by setting $\boldsymbol{L}_0 = \boldsymbol{A}$ and then creating $\boldsymbol{L}_{k+1}$ by adding multiples of rows and columns of $\boldsymbol{L}_k$ to other rows and columns of $\boldsymbol{L}_k$. This implies that $\boldsymbol{L}$ has nonzeros in all the same positions[1] as $\boldsymbol{A}$ plus some nonzeros in positions that were zero in $\boldsymbol{A}$, but induced by the factorization. It is exactly these nonzeros that are called *fill* elements. The presence of fill increases both the storage and work required in the factorization.

An example matrix is provided in Fig. 1 that shows non-zeros in original positions of $\boldsymbol{A}$ as "$\times$" and fill elements as "$\bullet$". This example incurs two fill elements. The order in which the factorization takes place greatly influences the amount of fill. The matrix $\boldsymbol{A}$ is often permuted by rows and columns to reduce the number of fill elements, thereby reducing storage and flops required for factorization. Given the example in Fig. 1, the elimination order $\{2, 6, 1, 3, 4, 5\}$ produces only one fill element. This is the minimum number of fill elements for this example.

If $\boldsymbol{A}$ is positive definite, Cholesky factorization is numerically stable for any symmetric permutation of $\boldsymbol{A}$, and the fill-reducing permutation need not be modified during factorization. If $\boldsymbol{A}$ is indefinite, then the initial permutation may have to be further modified during factorization for numerical stability.

2.2 Elimination Graph

The graph G of the sparse matrix $\boldsymbol{A}$ is a graph whose vertices correspond to the columns of $\boldsymbol{A}$. We label the vertices $1, \ldots, n$, to correspond to the n columns of $\boldsymbol{A}$. An edge (i, j) connecting vertices i and j in G exists if and only if a_{ij} is nonzero. By symmetry, $a_{j,i}$ is also nonzero.

The graph model of symmetric Gaussian elimination was introduced by Parter [7]. A sequence of elimination graphs, G_k, represent the fill created in each step of the factorization. The initial elimination graph is the graph of the matrix, $G_0 = G(\boldsymbol{A})$. At each step k, let v_k be the vertex corresponding to the k^{th} column of $\boldsymbol{A}$ to be eliminated. The elimination graph at the next step, G_{k+1}, is obtained by adding edges to make all the vertices adjacent to v_k pairwise adjacent to each other, and then removing v_k and all edges incident on v_k. The inserted edges are *fill edges* in the elimination graph. This process repeats until all the vertices are removed from the elimination graph. The example in Fig. 1 illustrates the graph model of elimination. Finding an elimination order that produces the minimum amount of fill is NP-complete [10].

2.3 Ordering Heuristics

An upper bound on the fill that a vertex of degree d can create on elimination is $d(d-1)/2$. The minimum degree algorithm attempts to minimize fill by choosing

[1] No "accidental" cancellations will occur during factorization if the numerical values in $\boldsymbol{A}$ are algebraic indeterminates.

Abbreviation	Algorithm Name	Primary Reference
MMD	Multiple Minimum Degree	Liu [5]
AMD	Approximate Minimum Degree	Amestoy, Davis and Duff [1]
AMF	Approximate Minimum Fill	Rothberg [8]
AMMF	Approximate Minimum Mean Local Fill	Rothberg and Eisenstat [9]
AMIND	Approximate Minimum Increase in Neighbor Degree	Rothberg and Eisenstat [9]
MMDF	Modified Minimum Deficiency	Ng and Raghavan [6]
MMMD	Modified Multiple Minimum Degree	Ng and Raghavan [6]

Table 1. Algorithms that fit into the Minimum Priority family.

the vertex with the minimum degree in the current elimination graph, hence reducing fill by controlling this worst-case bound. In Multiple Minimum Degree (MMD), a maximal independent set of vertices of low degree are eliminated in one step to keep the cost of updating the graph low.

Many more enhancements are necessary to obtain a practically efficient implementation of MMD. A survey article by George and Liu [4] provides the details. There have been several contributions to the field since the survey. A list of algorithms that we implement in our library and references are in Table 1. Most of these adaptations increase the runtime by 5-25% but reduce the amount of arithmetic required to generate the factor by 10-25%.

2.4 The Quotient Graph

Up to this point we have been discussing the elimination graph to model fill in a minimum priority ordering. While it is an important conceptual tool, it has difficulties in implementation arising from the fact that the storage required can grow like the size of the factor and cannot be predetermined. In practice, implementations use a *quotient graph*, $\mathcal{G}$, to represent the elimination graph in no more space than that of the initial graph $G(\boldsymbol{A})$. A quotient graph can have the same interface as an elimination graph, but it must handle internal data differently, essentially through an extra level of indirection.

The quotient graph has two distinct kinds of vertices: *supernodes* and *enodes*[2] A supernode represents a set of one or more uneliminated columns of $\boldsymbol{A}$. Similarly, an enode represents a set of one or more eliminated columns of $\boldsymbol{A}$. The initial graph, $\mathcal{G}_0$, consists entirely of supernodes and no enodes; further, each supernode contains one column. Edges are constructed the same as in the elimination graph. The initial quotient graph, $\mathcal{G}_0$, is identical to the initial elimination graph, G_0.

When a supernode is eliminated at some step, it is not removed from the quotient graph; instead, the supernode becomes an enode. Enodes indirectly represent the fill edges in the elimination graph. To demonstrate how, we first

[2] Also called "eliminated supernode" or "element" elsewhere.

```
k ← 0
while k < n
    Let m be the minimum known degree, deg(x), of all x ∈ G_k.
    while m is still the minimum known degree of all x ∈ G_k
        Choose supernode x_k such that deg(x_k) = m
        for all of the p columns represented by supernode x_k:
            Number columns (k + 1)... (k + p).
        Form enode e_k from supernode x_k and all adjacent enodes.
        for all supernodes x adjacent to c_k:
            Label deg(x) as "unknown."
        k ← k + p
    for all supernodes x where deg(x) is unknown:
        Update lists of adjacent supernodes and enodes of x.
        Check for various QuotientGraph optimizations.
        Compute deg(x).
```

Fig. 2. The Multiple Minimum Degree algorithm defined in terms of a Quotient Graph.

define a *reachable path* in the quotient graph as a path $(i, e_1, e_2, \ldots e_p, j)$, where i and j are supernodes in $\mathcal{G}_k$ and $e_1, e_2, \ldots e_p$ are enodes. Note that the number of enodes in the path can be zero. We also say that a pair of supernodes i, j is *reachable* in $\mathcal{G}_k$ if there exists a *reachable path* joining i and j. Since the number of enodes in the path can be zero, adjacency in $\mathcal{G}_k$ implies reachability in $\mathcal{G}_k$. If two supernodes i, j are reachable in the quotient graph $\mathcal{G}_k$, then the corresponding two vertices i, j in the elimination graph G_k are adjacent in G_k.

In practice, the quotient graph is aggressively optimized; all non-essential enodes, supernodes, and edges are deleted. Since we are only interested in paths through enodes, if two enodes are adjacent they are amalgamated into one. So in practice, the number of enodes in all reachable paths is limited to either zero or one. Alternatively, one can state that, in practice, the *reachable set* of a supernode is the union of its adjacent supernodes and all supernodes adjacent to its adjacent enodes. This amalgamation process is one way how some enodes come to represent more than their original eliminated column.

Supernodes are also amalgamated but with a different rationale. Two supernodes are *indistinguishable* if their reachable sets (including themselves) are identical. When this occurs, all but one of the indistinguishable supernodes can be removed from the graph. The remaining supernode keeps a list of all the columns of the supernodes compressed into it. When the remaining supernode is eliminated and becomes an enode, all its columns can be eliminated together. The search for indistinguishable supernodes can be done before eliminating a single supernode using graph compression [2]. More supernodes become indistinguishable as elimination proceeds. An exhaustive search for indistinguishable supernodes during elimination is prohibitively expensive, so it is often limited to supernodes with identical adjacency sets (assuming a self-edge) instead of identical reachable sets.

Edges between supernodes can be removed as elimination proceeds. When a pair of adjacent supernodes share a common enode, they are reachable through both the shared edge and the shared enode. In this case, the edge can be safely removed. This not only improves storage and speed, but allows tighter approximations to supernode degree as well.

Going once more to Fig. 1, we consider now the quotient graph. Initially, the elimination graph and quotient graph are identical. After the elimination of column 1, we see that supernode 1 is now an enode. Note that unlike the elimination graph, no edge was added between supernodes 3 and 4 since they are reachable through enode 1. After the elimination of column 2, we have removed an edge between supernodes 5 and 6. This was done because the edge was redundant; supernode 5 is reachable from 6 through enode 2. When we eliminate column 3, supernode 3 becomes an enode, it absorbs enode 1 (including its edge to supernode 4). Now enode 3 is adjacent to supernodes 4, 5 and 6. The fill edge between supernodes 4 and 5 is redundant and can be removed. At this point 4, 5, and 6 are indistinguishable. However, since we cannot afford an exhaustive search, a quick search (by looking for identical adjacency lists) finds only supernodes 5 and 6 so they are merged to supernode $\{5,6\}$. Then supernode 4 becomes an enode and absorbs enode 3. Finally supernode $\{5,6\}$ is eliminated. The relative order between columns 5 and 6 has no effect on fill.

We show the Multiple Minimum Degree algorithm defined in terms of a quotient graph in Fig. 2. A single elimination Minimum Degree algorithm is similar, but executes the inner **while** loop only once. We point out that we have not provided an exhaustive accounting of quotient graph features and optimizations. Most of the time is spent in the last three lines Fig. 2, and often they are tightly intertwined in implementations.

3 Design

To provide a basis for comparison, we briefly discuss the design and implementation characteristics of MMD [5] and AMD [1]. Both implementations were written in Fortran77 using a procedural decomposition. They have no dynamic memory allocation and implement no abstract data types in the code besides arrays.

GENMMD is implemented in roughly 500 lines of executable source code with about 100 lines of comments. The main routine has 12 parameters in its calling sequence and uses four subroutines that roughly correspond to initialization, supernode elimination, quotient graph update/degree calculation, and finalization of the permutation vector. The code operates in a very tight footprint and will often use the same array for different data structures at the same time. The code has over 20 goto statements and can be difficult to follow.

AMD has roughly 600 lines of executable source code which almost doubles when the extensive comments are included. It is implemented as a single routine with 16 calling parameters and no subroutine calls. It is generally well structured and documented. Manually touching up our f2c conversion, we were able to

- Quotient Graph
  1. Must provide a method for extracting the Reachable Set of a vertex.
  2. Be able to eliminate supernodes on demand.
  3. Should have a separate lazy update method for multiple elimination.
  4. Should provide lists of compressed vertices that can be ignored for the rest of the ordering algorithm.
  5. Must produce an elimination tree or permutation vector after all the vertices have been eliminated.
  6. Should allow const access to current graph for various Priority Strategies.
- Bucket Sorter
  1. Must remove an item from the smallest non-empty bucket in constant time.
  2. Must insert an item-key pair in constant time.
  3. Must remove an item by name from anywhere in constant time.
- Priority Strategy
  1. Must compute the new priority for each vertex in the list.
  2. Must insert the priority-vertex pairs into the Bucket Sorter.

Fig. 3. Three most important classes in a minimum priority ordering and some of their related requirements.

easily replace the 17 goto statements with while loops, and break and continue statements. This code is part of the commercial Harwell Subroutine Library, though we report results from an earlier version shared with us.

The three major classes in our implementation are shown in a basic outline in Fig. 3. Given these classes, we can describe our fourth object; the MinimumPriorityOrdering class that is responsible for directing the interactions of these other objects. The main method of this class (excluding details, debugging statements, tests, comments, etc.) is approximately the code fragment in Fig. 4. By far the most complicated (and expensive) part of the code is line 15 of Fig. 4 where the graph update occurs.

The most elegant feature of this implementation is that the PriorityStrategy object is an abstract base class. We have implemented several derived classes, each one implementing one of the algorithms in Table 1. Each derived class involves overriding two virtual functions (one of them trivial). The classes derived from PriorityStrategy average 50 lines of code each. This is an instance of the Strategy Pattern [3].

The trickiest part is providing enough access to the QuotientGraph for the PriorityStrategy to be useful and extensible, but to provide enough protection to keep the PriorityStrategy from corrupting the rather complicated state information in the QuotientGraph.

Because we want our library to be extensible, we have to provide the PriorityStrategy class access to the QuotientGraph. But we want to protect that access so that the QuotientGraph's sensitive and complicated internal workings are abstracted away and cannot be corrupted. We provided a full-fledged iterator class, called ReachableSetIterator, that encapsulated the details of the Quotient-

```
    // Major Classes
    QuotientGraph* qgraph;
    BucketSorter* sorter;
    PriorityStrategy* priority;
    SuperNodeList* reachableSuperNodes, * mergedSuperNodes;

    // Initialization...
    ...
    // Load all vertices into sorter
1.  priority->computeAndInsert( priority::ALL_NODES, qgraph, sorter );
2.  if ( priority->requireSingleElimination() == true )
3.       maxStep = 1;
    else
4.       maxStep = graph->size();

    // Main loop
5.  while ( sorter->notEmpty() ) {
6.       int min = sorter->queryMinNonemptyBucket();
7.       int step = 0;
8.       while ( ( min == sorter->queryMinNonemptyBucket() &&
                   ( step < maxStep ) ) {
9.            int snode = sorter->removeItemFromBucket( min );
10.           qgraph->eliminateSupernode( snode );
              SuperNodeList* tempSuperNodes;
11.           tempSuperNodes = qgraph->queryReachableSet( snode );
12.           sorter->removeSuperNodes( tempSuperNodes );
13.           *reachableSuperNodes += *tempSuperNodes;
14.           ++step;
         }
15.      qgraph->update( reachableSuperNodes, mergedSuperNodes );
16.      sorter->removeSuperNodes( mergedSuperNodes );
17.      priority->computeAndInsert( reachableSuperNodes, qgraph, sorter );
18.      mergedSuperNodes->resize( 0 );
19.      reachableSuperNodes->resize( 0 );
    }
```

Fig. 4. A general Minimum Priority Algorithm using the objects described in Fig. 3

Graph from the PriorityStrategy, making the interface indistinguishable from an EliminationGraph.

Unfortunately, the overhead of using these iterators to compute the priorities was too expensive. We rewrote the PriorityStrategy classes to access the QuotientGraph at a lower level... traversing adjacency lists instead of reachable sets. This gave us the performance we needed, but had the unfortunate effect of increasing the coupling between classes. However, the ReachableSetIterator was left in the code for ease of prototyping.

Currently we have implemented a PriorityStrategy class for all of the algorithms listed in Table 1. They all compute their priority as a function of either the *external degree*, or a tight *approximate degree*, of a supernode. Computing the external degree is more expensive, but allows multiple elimination. For technical reasons, to get the approximate degree tight enough the quotient graph must be updated after every supernode is eliminated, hence all algorithms that use approximate degree are single elimination algorithms[3]. For this reason, all previous implementations are either multiple elimination codes or single elimination codes, not both. The quotient graph update is the most complicated part of the code and single elimination updates are different from multiple elimination updates.

The MinimumPriorityOrdering class queries the PriorityStrategy whether it requires quotient graph updates after each elimination or not. It then relays this information to the QuotientGraph class which has different optimized update methods for single elimination and multiple elimination. The QuotientGraph class can compute partial values for external degree or approximate degree as a side-effect of the particular update method.

Given this framework, it is possible to modify the MinimumPriorityOrdering class to switch algorithms during elimination. For example, one could use MMD at first to create a lot of enodes fast, then switch to AMD when the quotient graph becomes more tightly connected and independent sets of vertices to eliminate are small. There are other plausible combinations because different algorithms in Table 1 prefer vertices with different topological properties. It is possible that the topological properties of the optimal vertex to eliminate changes as elimination progresses.

4 Results

We compare actual execution times of our implementation to an f2c conversion of the GENMMD code by Liu [5]. This is currently among the most widely used implementations. In general, our object-oriented implementation is within a factor of 3-4 of GENMMD. We expect this to get closer to a factor of 2-3 as the code matures. We normalize the execution time of our implementation to

[3] Readers are cautioned that algorithms in Table 1 that approximate quantities other than degree could be multiple elimination algorithms. Rothberg and Eisenstat [9] have defined their algorithms using either external degree (multiple elimination) or approximate degree (single elimination).

	problem	$\|V\|$	$\|E\|$	time (seconds) GENMMD no compr.	time(normalized) C++ no compr.	C++ compr.
1.	commanche	7,920	11,880	.08	5.88	5.81
2.	barth4	6,019	17,473	.06	6.17	6.42
3.	barth	6,691	19,748	.10	5.00	5.36
4.	ford1	18,728	41,424	.30	4.57	4.69
5.	ken13	28,632	66,486	3.61	.92	.94
6.	barth5	15,606	45,878	.28	4.96	4.97
7.	shuttle_eddy	10,429	46,585	.09	9.44	9.33
8.	bcsstk18	11,948	68,571	.44	4.59	4.89
9.	bcsstk16	4,884	142,747	.16	8.19	1.74
10.	bcsstk23	3,134	21,022	.22	4.32	4.34
11.	bcsstk15	3,948	56,934	.22	4.77	4.62
12.	bcsstk17	10,974	208,838	.30	5.97	2.33
13.	pwt	36,519	144,794	.58	6.16	6.32
14.	ford2	100,196	222,246	2.44	3.84	3.90
15.	bcsstk30	28,924	1,007,284	.95	5.79	1.67
16.	tandem_vtx	18,454	117,448	.85	4.11	4.13
17.	pds10	16,558	66,550	107.81	1.24	1.16
18.	copter1	17,222	96,921	.67	6.22	6.52
19.	bcsstk31	35,588	572,914	1.50	4.83	2.58
20.	nasasrb	54,870	1,311,227	2.06	6.14	2.44
21.	skirt	45,361	1,268,228	2.03	6.38	1.72
22.	tandem_dual	94,069	183,212	4.50	3.70	3.67
23.	onera_dual	85,567	166,817	4.23	3.65	3.69
24.	copter2	55,476	352,238	3.96	4.57	4.70
		geometric mean			4.61	3.53
		median			4.90	4.24

Table 2. Relative performance of our implementation of MMD (both with and without precompression) to GENMMD. GENMMD does not have precompression. The problems are sorted in nondecreasing size of the Cholesky factor.

GENMMD and present them in Table 2. For direct comparison, pre-compressing the graph was disabled in our C++ code. We also show how our code performs with compression.

All runtimes are from a Sun UltraSPARC-5 with 64MB of main memory. The software was compiled with GNU C++ version 2.8.1 with the `-O`, and `-fno-exceptions` flags set. The list of 24 problems are sorted in nondecreasing order of the work in computing the factor with the MMD ordering. The numbers presented are the average of eleven runs with different seeds to the random number generator. Because these algorithms are extremely sensitive to tie-breaking, it is common to randomize the graph before computing the ordering.

We refer the reader to Table 3 for relative quality of orderings and execution times. As with the previous table, the data represents the average of 11 runs with different seeds in the random number generator. The relative improvement

	problem	Work MMD	Work (normalized) AMD	AMF	AMMF	AMIND	MMDF	MMMD
1.	commanche	1.76e+06	1.00	.89	.87	.87	.92	.89
2.	barth4	4.12e+06	1.00	.89	.83	.82	.86	.82
3.	barth	4.55e+06	1.02	.90	.84	.85	.91	.89
4.	ford1	1.67e+07	.98	.84	.87	.82	.89	.86
5.	ken13	1.84e+07	1.01	.89	.88	.96	.83	.87
6.	barth5	1.96e+07	1.00	.90	.81	.82	.72	.83
7.	shuttle_eddy	2.76e+07	.97	.87	.74	.74	.75	.81
8.	bcsstk18	1.37e+08	.98	.77	.78	.74	.86	.83
9.	bcsstk16	1.56e+08	1.02	.81	.84	.82	.82	.81
10.	bcsstk23	1.56e+08	.95	.79	.73	.75	.80	.81
11.	bcsstk15	1.74e+08	.97	.89	.84	.81	.84	.86
12.	bcsstk17	2.22e+08	1.10	.89	.85	.88	1.02	.89
13.	pwt	2.43e+08	1.03	.92	.87	.90	.88	.90
14.	ford2	3.19e+08	1.03	.76	.72	.70	.77	.77
15.	bcsstk30	9.12e+08	1.01	.97	.82	.79	.88	.87
16.	tandem_vtx	1.04e+09	.97	.77	.56	.66	.70	.77
17.	pds10	1.04e+09	.90	.88	.91	.87	.88	1.00
18.	copter1	1.33e+09	.96	.82	.62	.71	.79	.87
19.	bcsstk31	2.57e+09	1.00	.95	.67	.71	.94	.87
20.	nasasrb	5.47e+09	.95	.82	.70	.79	.93	.82
21.	skirt	6.04e+09	1.11	.83	.90	.76	.88	.83
22.	tandem_dual	8.54e+09	.97	.42	.51	.62	.72	.72
23.	onera_dual	9.69e+09	1.03	.70	.48	.57	.65	.71
24.	copter2	1.35e+09	.97	.73	.50	.61	.66	.69
	geometric mean		1.00	.84	.74	.77	.83	.83
	median		1.00	.85	.82	.79	.85	.83

Table 3. Comparison of quality of various priority policies. Quality of the ordering here is measured in terms of the amount of work to factor the matrix with the given ordering. Refer to Table 1 for algorithm names and references.

in the quality of the orderings over MMD is comparable with the improvements reported by other authors, even though the test sets are not identical.

We have successfully compiled and used our code on Sun Solaris workstations using both SunPRO C++ version 4.2 and GNU C++ version 2.8.1.1. The code does not work on older versions of the same compilers. We have also compiled our code on Windows NT using Microsoft Visual C++ 5.0.

5 Conclusions

Contrary to popular belief, our implementation shows that the most expensive part of these minimum priority algorithms is *not* the degree computation... it is the quotient graph update. With all other implementations—including GENMMD and AMD—the degree computation is tightly coupled with the quotient

graph update, making it impossible to separate the costs of degree computation from graph update with any of the earlier procedural implementations. The priority computation (for minimum degree) involves traversing the adjacency set of each reachable supernode after updating the graph. Updating the graph, however, involves updating the adjacency sets of each supernode and enode adjacent to each reachable supernode. This update process often requires several distinct passes.

By insisting on a flexible, extensible framework, we required more decoupling between the priority computation and graph update: between algorithm and data structure. In some cases, we had to increase the coupling between key classes to improve performance. We are generally satisfied with the performance of our code and with the value added by providing implementations of the full gamut of state-of-art algorithms. We will make the software publicly available.

Acknowledgements We thank Tim Davis and Joseph Liu for their help and insights from their implementations and experiences. We are especially grateful to Cleve Ashcraft for stimulating discussions about object-oriented design, efficiency, and programming tricks.

References

1. Patrick Amestoy, Timothy A. Davis, and Iain S. Duff. An approximate minimum degree ordering algorithm. Technical Report TR–94–039, Computer and Information Sciences Dept., University of Florida, December 1994.
2. Cleve Ashcraft. Compressed graphs and the minimum degree algorithm. *SIAM J. Sci. Comput.*, 16(6):1404–1411, 1995.
3. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional Computing Series. Addison Wesley Longman, 1995.
4. J. Alan George and Joeseph W. H. Liu. The evolution of the minimum degree algorithm. *SIAM Rev.*, 31(1):1–19, 1989.
5. Joseph W. H. Liu. Modification of the minimum-degree algorithm by multiple elimination. *ACM Trans. Math. Software*, 11:141–153, 1985.
6. Esmond G. Ng and Padma Raghavan. Performance of greedy ordering heuristics for sparse Cholesky factorization. Submitted to SIAM J. Mat. Anal. Appl., 1997.
7. S Parter. The use of planar graphs in Gaussian elimination. *SIAM Rev.*, 3:364–369, 1961.
8. Ed Rothberg. Ordering sparse matrices using approximate minimum local fill. Preprint, April 1996.
9. Ed Rothberg and Stan Eisenstat. Node selection strategies for bottom-up sparse matrix ordering. *SIAM J. Matrix Anal. Appl.*, 19(3):682–695, 1998.
10. M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM J. Algebraic and Discrete Methods*, 2:77–79, 1981.