Old Dominion University
ODU Digital Commons

Computer Science Theses & Dissertations | Computer Science

Spring 2000

An Object-Oriented Algorithmic Laboratory for Ordering Sparse Matrices

Gary Karl Kumfert
Old Dominion University

Follow this and additional works at: https://digitalcommons.odu.edu/computerscience_etds

Part of the Computer Sciences Commons, and the Mathematics Commons

Recommended Citation
Kumfert, Gary K.. "An Object-Oriented Algorithmic Laboratory for Ordering Sparse Matrices" (2000). Doctor of Philosophy (PhD), dissertation, Computer Science, Old Dominion University, DOI: 10.25777/n4ss-xc64
https://digitalcommons.odu.edu/computerscience_etds/108

This Dissertation is brought to you for free and open access by the Computer Science at ODU Digital Commons. It has been accepted for inclusion in Computer Science Theses & Dissertations by an authorized administrator of ODU Digital Commons. For more information, please contact digitalcommons@odu.edu.

AN OBJECT-ORIENTED ALGORITHMIC LABORATORY

FOR ORDERING SPARSE MATRICES

by

Gary Karl Kumfert
B.S. May 1993 Old Dominion University

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirement for the Degree of

DOCTOR OF PHILOSOPHY

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
May 2000

Approved by:

Alex Pothen (Director)

Chet Grosch (Member)

David Keyes (Member)

Stephen Olariu (Member)

Virginia Torczon (Member)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ABSTRACT

AN OBJECT-ORIENTED ALGORITHMIC LABORATORY
FOR ORDERING SPARSE MATRICES

Gary Karl Kumfert
Old Dominion University, 2000
Advisor: Dr. Alex Pothen

We focus on two known NP-hard problems that have applications in sparse matrix computations: the envelope/wavefront reduction problem and the fill reduction problem. Envelope/wavefront reducing orderings have a wide range of applications including profile and frontal solvers, incomplete factorization preconditioning, graph reordering for cache performance, gene sequencing, and spatial databases. Fill reducing orderings are generally limited to — but an inextricable part of — sparse matrix factorization.

Our major contribution to this field is the design of new and improved heuristics for these NP-hard problems and their efficient implementation in a robust, cross-platform, object-oriented software package. In this body of research, we (1) examine current ordering algorithms, analyze their asymptotic complexity, and characterize their behavior in model problems, (2) introduce new and improved algorithms that address deficiencies found in previous heuristics, (3) implement an object-oriented library of these algorithms in a robust, modular fashion without significant loss of efficiency, and (4) extend our algorithms and software to address both generalized and constrained problems. We stress that the major contribution is the algorithms *and* the implementation; the whole being greater than the sum of its parts.

The initial motivation for implementing our algorithms in object-oriented software was to manage the inherent complexity. During our research came the realization that the object-oriented implementation enabled new possibilities for augmented algorithms that would not have been as natural to generalize from a procedural implementation. Some extensions are constructed from a family of related algorithmic components, thereby creating a poly-algorithm that can adapt its strategy to the properties of the specific problem instance dynamically. Other algorithms are tailored for special constraints by aggregating algorithmic components and having them collaboratively generate the global ordering.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Our software laboratory, "Spindle," implements state-of-the-art ordering algorithms for sparse matrices and graphs. We have used it to examine and augment the behavior of existing algorithms and test new ones. Its 40,000+ lines of C++ code includes a base library test drivers, sample applications, and interfaces to C, C++, Matlab, and PETSc. Spindle is freely available and can be built on a variety of UNIX platforms as well as WindowsNT.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

To my wife, Wen.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ACKNOWLEDGEMENTS

There are many people who have helped me in my research and contributed to the successful completion of this dissertation.

I extend many, many thanks to my committee members for their patience and hours of guidance on my research and editing of this manuscript. It has been much improved because of their feedback.

The untiring efforts of my advisor, Alex Pothen, deserve special recognition. He has been my mentor, friend, task master, guru, and role model. I very much enjoyed working under him and am grateful for his support.

I am also very grateful for the moral and political support of David Keyes; also a teacher, mentor, and friend. He has gone to bat for me on more than one occasion, and always been a supporter of my research and my activism.

Special thanks also goes to two of my office mates, colleagues, and friends Florin Dobrian and David Hysom. I have enjoyed and appreciated the constant interaction with them about algorithms, linear algebra, object-oriented design, politics, economics, religion, mountain climbing, stress management, and photography.

My thanks also to Bruce Hendrickson who was very supportive of my work from early on. He provided lots of support when we were using Chaco for spectral orderings.

I learned a lot, both individually and collectively, from the PETSc Team — Satish Balay, Lois Curfmann-McInnes, Barry Smith, and Bill Gropp — at Argonne National Labs. I also learned a lot of software tricks digging though their code.

Thanks also goes to Jennifer Scott at Rutherford Appleton Labs who offered a lot of help and support on my early Sloan work.

I also greatly benefitted from impromptu meetings with Joseph Liu, Tim Davis and Cleve Ashcraft at various conferences and through email. They were all very helpful in explaining the finer points of their

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

various implementations. Cleve deserves special thanks for also voluntarily reviewing this manuscript and providing very useful feedback.

Special mention goes to the love and support I recieved from my family: my wife, Wen Kumfert; my parents, Carolyn Carr and Gerhard Kumfert; and my brother Kirk Kumfert. I know a copy of this dissertation will sit in Kirk's classroom, just to show his students that matrices are indeed used beyond high school math classes.

Thanks also goes to the federal government, which funded most of my graduate studies. My research was supported by National Science Foundation grants CCR-9412698 and DMS-9807172, and by a GAANN fellowship from the Department of Education. The final editing was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract no. W–7405–Eng–48.

Finally, I wish to acknowledge two teachers from Philadelphia Public Schools that have significantly influenced me at an early age.

I wish to pay tribute to my calculus teacher at Northeast Public High School, the late Jerry Kramer. He made me work hard, and made me want to succeed. His untimely death was a terrible loss. I would not have started my career in Mathematics were it not for his influence.

I am also very grateful for Marilyn Melstein — my first computer science teacher and still a friend to this day. She did more than teach us how to program in BASIC on Apple II's and II+'s. She encouraged creativity and problem solving with a passion. I still remember to this day one particular assignment that I completed using nested loops; she called it an "elegant solution." In a way, I have been striving for that ever since.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

DISCLAIMER

The final editing of this document occurred while the author was employed by an agency of the United States Government under the management of the University of California. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of the authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

UCRL-LR-136934

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE OF CONTENTS

Page

LIST OF TABLES x

LIST OF FIGURES xi

LIST OF ALGORITHMS xiii

Section

1. INTRODUCTION 1

2. BACKGROUND 3
   - SPARSE MATRIX COMPUTATIONS 3
   - OBJECT-ORIENTED PROGRAMMING 10

3. ALGORITHMS 16
   - DIAMETER OF A GRAPH 16
   - ENVELOPE/WAVEFRONT REDUCTION 21
   - FILL REDUCTION 59
   - SUMMARY 91

4. SOFTWARE 92
   - RESOURCES USED 92
   - DESIGN 93
   - FEATURES 114
   - BALANCING FEATURES WITH PERFORMANCE 123
   - SUMMARY 126

5. RESULTS 128
   - ENVELOPE/WAVEFRONT REDUCTION 128
   - POLYMORPHIC MINIMUM FILL 134

6. EXTENSIONS 150
   - UNSYMMETRIC ORDERINGS 150
   - CONSTRAINED ORDERINGS 157
   - SUMMARY 160

7. CONCLUSION 162

BIBLIOGRAPHY 164

INDEX 172

VITA 175

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

LIST OF TABLES

Table — Page

1. Results of two problems on a CRAY-J90 using MA42. 55
2. Convergence of preconditioned CG on `body.y-5` and `bcsstk17`. 57
3. Several Greedy Fill-Reducing Heuristics. 67
4. Transition table for the `DataStruct` class 97
5. Transition table for the `SpindleAlgorithm` class 106
6. Eighteen test problems for wavefront reduction. 129
7. Mean square Wavefront sizes for various algorithms relative to RCM. 130
8. Maximum wavefront sizes relative to the RCM algorithm. 131
9. Envelope sizes relative to RCM. 132
10. Bandwidths relative to RCM. 132
11. CPU times relative to the RCM algorithm. 133
12. Test Set for Fill Reducing Orderings. 135
13. Comparison of MMD implementations: GENMMD and SPOOLES vs. Spindle 137
14. Comparison of AMD Implementations: AMDBAR and SPOOLES vs. Spindle 138
15. Storage requirements for factors using various fill-reducing orderings. 141
16. Work requirements for factors using various fill-reducing orderings. 142
17. CPU time to generate various fill-reducing orderings. 143
18. Comparison of Spindle's greedy algorithms vs. reported nested dissection algorithms. . . 149

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

LIST OF FIGURES

Figure Page

1. Different Graph Models of a Sparse Matrix. . . . 4
2. A "black box" formulation of a sparse direct solver. . . . 9
3. Example of row-widths and wavefronts 25
4. Structure of bcsstk30 with four orderings. . . . 26
5. Domains of a Vertical Ordering on rectangular 5-point grid. . . . 29
6. Domains of a Diagonal Ordering on rectangular 5-point grid. . . . 30
7. Domains of a Vertical Ordering on rectangular 9-point grid. . . . 32
8. Domains of a Diagonal Ordering on rectangular 9-point grid. . . . 34
9. The Sloan algorithm in progress. . . . 43
10. Envelope parameters of barth5 as a function of the weights W_1 and W_2. . . . 45
11. Envelope parameters of finance512 as a function of the weights W_1 and W_2. . . . 46
12. Relative timing performance of RCM, ArraySloan, and HeapSloan algorithms. . . . 50
13. Convergence of body.y-5 for various orderings using IC(0) preconditioned CG. . . . 56
14. Convergence of bcsstk17 for various orderings using IC(2) preconditioned CG. . . . 56
15. Examples of factorization and fill. . . . 60
16. Example of a quotient graph. . . . 63
17. Quotient Graph Transformation in Detail. . . . 65
18. State Diagram of a Quotient Graph. . . . 71
19. The Quotient Graph while eliminating a 16×16 torus. . . . 85
20. Interface to the ClassMetaData class. . . . 94
21. Interface to the SpindleBaseClass class. . . . 95
22. Interface to the DataStructure class. . . . 96
23. Example of PermutationMap 102
24. Object persistence of PermutationMap: part 1. . . . 103
25. Object persistence of PermutationMap: part 2. . . . 104
26. Interface to the SpindleAlgorithm class. . . . 105

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Figure Page

27. Example: Algorithms in ac'ion. 111

28. A C-like function that directly accesses a Graph's data 112

29. A C++ global function using Graph::adj_iterator 113

30. Example: cyclic dependency between classes. 115

31. Example: removing cyclic dependencies by ecalation. 115

32. Inheritance hierarchy for multiple matrix formats. 119

33. The Strategy Pattern. 120

34. An augmented Strategy Pattern. 121

35. Interaction of initializing the ordering. 122

36. Interaction of the ordering. 123

37. Interaction of eliminating vertices during ordering. 124

38. Supernode compression and elimination in fill-reducing orderings 144

39. Details of pds10 when dynamically changing from MMD to AMD 146

40. Details of ken13 when dynamically changing from MMD to AMD 147

41. An example square, unsymmetric matrix. 151

42. Example of LU factorization. 155

43. Constrained wavefront reduction example. 159

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

LIST OF ALGORITHMS

Algorithm Page

1. The Pseudo-Diameter Algorithm. 19

2. The Sloan algorithm for a vertex-weighted graph. 42

3. The Multiple Minimum Degree algorithm defined in terms of a Quotient Graph. 66

4. The most expensive loop in the MD and MMD update process. 75

5. The most expensive loop in the AMD update process 79

6. Computing degree using the reachable set iterator. 125

7. Computing degree without using the reachable set iterators. 126

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1. INTRODUCTION

There are two major ordering problems that are addressed in this thesis, motivated primarily by their application to sparse matrix computations in scientific computing. Both of these problems are known to be NP-hard and have had several heuristics vie for dominance in each case. These algorithms are designed to work on large problems, often testing the amount of storage available on the computer.

For the envelope/wavefront reduction problem, we were successful in enhancing a combinatorial heuristic in several ways. We achieved a significant reduction in the asymptotic complexity and show a corresponding reduction in actual running time. We also identified a clear dichotomy in how the algorithm behaved with different problem instances and therefore could further improve the quality of the result. Finally, we were able to adapt the algorithm to refine an existing ordering, which gives rise to an interesting algebraic-combinatorial hybrid.

The fill reduction problem has received a great deal more attention with many papers and algorithms already published. Interestingly, while most of the algorithms are closely related, there is no single piece of software that implements all of these algorithms. We provide an entire suite of greedy fill-reducing ordering algorithms. We also present asymptotic complexity bounds for the minimum degree (MD), multiple minimum degree (MMD), and approximate minimum degree (AMD) algorithms. Our implementation is the first known implementation using advanced object-oriented techniques such as polymorphism. The execution time of our implementation using these object-oriented techniques is within a small constant of traditional Fortran implementations, but far more flexible.

Our interest in developing new algorithms therefore extends to their efficient implementation. Significant attention is therefore expended on the design and implementation of our object-oriented software, Spindle. We use Spindle to experiment with, and extend our knowledge in, ordering problems. Special attention is paid to the trade-offs between the use of elegant, advanced software techniques and achieving high performance retain efficiency comparable with procedurally based codes implemented in C and Fortran77.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

There is a critical relationship between the "computer science" and "software engineering" — the algorithms and the implementation — that should not be dismissed lightly. Advanced algorithms, which are essential for good performance, are increasingly complex and can greatly benefit from good object-oriented implementations. In return, a well-defined framework of algorithmic components is significantly more flexible and extensible than a collection of algorithmic "black boxes."

Most computations involving a large, sparse systems of equations are sensitive to the ordering of the equations and the unknowns. For some instances, such as direct factorization, the impact of ordering this system is well understood. Other instances, such as incomplete factorization preconditioning, the effect of the ordering has been observed but is not well understood or even well characterized. Ordering these systems for specific computations is often vital to make the computation efficient, or even tractable. A poor ordering of the matrix in a sparse direct factorization can inflate the asymptotic complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n^3)$. The ordering of the matrix in incomplete factor preconditioned Krylov solver can accelerate or even prohibit convergence.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2. BACKGROUND

The information required to frame this research in context is provided here. Our research spans the domains of sparse numerical linear algebra and object-oriented software engineering. Section 2.1 discusses relevant sparse matrix topics, primarily graph models and sparse factorization. Section 2.2 enumerates the software design principles we consider in the process of building the object-oriented software.

2.1 Sparse Matrix Computations

We often describe solving the system of equations

$$Ax = b$$

where A is large and sparse. Interestingly, the "sparse" qualifier is never actually defined. An informal working definition — generally attributed to J. H. Wilkinson [40] — is that the matrix A has enough zero entries to make storing only the non-zero entries and their explicit indices more efficient than that storing the zeros explicitly and foregoing the indices. Assuming 8 byte double-precision floating point numbers (16 for complex numbers) and 4 byte integers for the indexing, only one half of the n^2 entries in an $n \times n$ matrix would need to be zero.

In practice, a much stronger statement of sparsity can be made — especially in matrices arising from finite differences and finite element methods. These matrices are so sparse that there are only $\mathcal{O}(n)$ non-zero entries.

2.1.1 Graph Models

The placement of the nonzeros in the matrix, called the *nonzero structure*, determines many aspects of sparse matrix computations. Graphs are a common abstraction to model the nonzero structure of a sparse matrix. In this section, we list three graph models, some of their salient characteristics, and in what

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

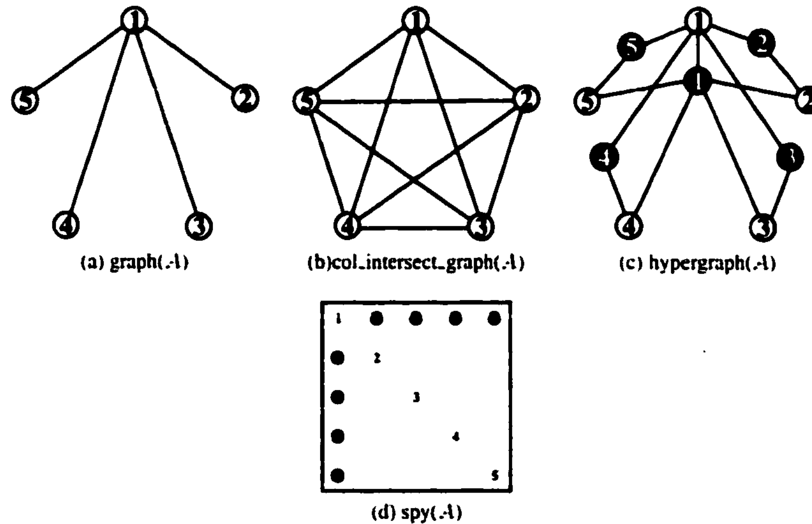


FIG. 1. *Different Graphs Models of a Sparse Matrix. (a) adjacency graph, (b) column intersection graph, and (c) hypergraph of the same sparse matrix (d). Note that for the hypergraph, the hyperedges are represented as black circles with white labels for the hubs.*

contexts each model is most appropriate. Figure 1 presents all three models for a simple matrix to further illustrate their differences.

2.1.1.a Adjacency Graph

By far the most common graph model employed is that of an undirected graph on a symmetric sparse matrix, called the *adjacency graph*. The name may sound somewhat redundant until one realizes that an undirected graph is commonly stored represented as an adjacency matrix. The adjacency graph is so pervasive that it is commonly referred to as *the* graph of a matrix, even though other graph models exist. Before discussing the construction of this model, we need to introduce the concept of *structural symmetry* and its relation the normal concept of a symmetric matrix.

Definition 2.1 *Consider a sparse $n \times n$ matrix $A = [a_{ij}]$. The matrix is said to be* structurally symmetric *if and only if for every $a_{ij} \neq 0$ in A there exists $a_{ji} \neq 0$.*

Remark 2.1 *If a matrix is symmetric, it must also be structurally symmetric.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Definition 2.2 *Consider a sparse, structurally symmetric $n \times n$ matrix $A = [a_{ij}]$. The* graph of the matrix, *$G(A) = (V, E)$, where V is a set of n vertices corresponding to the n rows/columns of A, and E is a set of edges (i, j) where $(i, j) \in E \Leftrightarrow a_{ij} \neq 0$.*

The adjacency graph is the most common graph model employed in ordering and partitioning codes, despite its known deficiencies [42, 43]. The most immediate problem is that it is restricted to symmetric matrices. It is not uncommon to find software that will add explicit nonzeros to a sparse, square, unsymmetric matrices to make them structurally symmetric. Furthermore self edges induced by nonzero diagonal elements in the matrix, if any, are dropped. Thus, an unsymmetric matrix permutation derived from a symmetric graph may have anomalies that are caused by the implicit removal of any diagonal nonzeros. It should also be noted that partitioning codes typically use this model, and typically try to minimize edge cuts, though this does not accurately translate into communication costs in a matrix-vector multiply in parallel [15].

It is common to have weights attached to the vertices, edges, or both in a graph. Rarely are these weights directly related to the nonzero values in the original matrix. More often these weights are integer values that are used for combinatorial purposes.

2.1.1.b Column Intersection Graph

The column intersection graph can be generated for any rectangular matrix. It has been used to generate a column permutation for unsymmetric fill-reducing orderings. The construction of the column intersection graph is as follows.

Definition 2.3 *Consider a sparse $m \times n$ matrix $A = [a_{ij}]$. The* column intersection graph of the matrix, *$G_c(A) = (V_c, E_c)$, where V_c is a set of n vertices corresponding to the n columns of A, and E_c is a set of edges (i, j) where $(i, j) \in E_c \Leftrightarrow a_{ki} \neq 0$ and $a_{kj} \neq 0$ for some row k.*

While this heuristic has been used in commercial applications, it is imperfect. The mapping from rectangular matrices is onto, but not one-to-one. That is to say that the column intersection graph does not directly translate back into the nonzero structure of a unique matrix. In Figure 1(b) the column

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

intersection graph is completely connected, even though the original matrix is sparse. Clearly, we could add any number of additional nonzeros to the original matrix in Figure 1(d) without changing the column intersection graph.

There is a relationship between a column intersection graph and an adjacency graph as given in the following theorem.

Observation 2.1 *For any sparse rectangular matrix $A = [a_{ij}]$, the column intersection graph is identical to the adjacency graph of $A^T A$.*

2.1.1.c Hypergraph

The hypergraph model is the least commonly implemented, but may be the most robust model of all. It has applications particularly in unsymmetric wavefront and partitioning applications, though no known codes using it currently exist.

Definition 2.4 *Consider a sparse $m \times n$ matrix $A = [a_{ij}]$. The* hypergraph of the matrix, $G_h(A) = (V_h, E_h)$, *where V_h is a set of n vertices corresponding to the n columns of A, and E_h is a set of hyperedges $(i, j, k, \dots)$ corresponding to the m columns of A. Each vertex j is a member of hyperedge i if an only if $a_{ij} \neq 0$.*

2.1.1.d Additional Graph Theory

Graph compression is an important technique that is critical for achieving the best performance in modern codes. To explain the rationale behind graph compression, we must first define *indistinguishable vertices*.

Definition 2.5 *Given an undirected graph $G = (V, E)$, two vertices $v, w \in V$ are* indistinguishable *if and only if there exists an edge $(v, w) \in E$ and $\{\text{adj}(v) \cup v\} = \{\text{adj}(w) \cup w\}$.*

Graph compression is the practice of finding sets of indistinguishable vertices and replacing them with a single supervertex. This supervertex has a weight equal to the number of constituent indistinguishable vertices. If there is an edge between two supervertices i and j in the compressed graph, it logically

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

represents vwgt(i) × vwgt(j) edges in the original graph, where vwgt() is the vertex weight. Therefore, it is only necessary to store an array of length $|V|$ for the vertex weights of the compressed graph — edge weights are computed on demand.

Graph compression can drastically reduce the storage required for, and the time spent indexing into, sparse data structure. In most cases, an ordering can be performed on the compressed graph with the understanding that members of a supervertex are numbered sequentially. Working with the compressed graph can even improve the quality of the result, particularly with some new fill-reducing heuristics.

This compression technique applies to sparse matrices, but it is not exactly the same. In our ordering algorithms, we are concerned only with the structure of the matrix, not the values contained therein. Sparse matrices, however, have values associated with each nonzero entry that must be preserved. In sparse matrices, the indistinguishable vertices form dense blocks — called *inodes* in PETSc [7, 8], and *supernodes* in SuperLU [56]. This reduces time and space for sparse indexing and increases flop rates and cache performance by operating on dense blocks within a sparse matrix.

Definition 2.6 *A* clique, *C, in a graph $G = (V, E)$ is a set of vertices $C \subset V$ such that for every i and j in C, there exists an edge (i, j) in E.*

Observation 2.2 *Each set of indistinguishable vertices forms a clique, but not every clique forms an indistinguishable set of vertices.*

For the rest of this manuscript, a graph is assumed to be undirected and may or may not be compressed. Any uncompressed graph without explicit vertex or edge weights is assumed to have all vertex and edge weights equal to one.

If x is a vertex in the graph, then we define its adjacency set adj(x) as the set of all vertices that share an edge with x. If X is a set of vertices in a graph, then its adjacency set

$$\text{adj}(X) = \left(\bigcup_{v \in X} \text{adj}(v) \right) \setminus X \quad .$$

The degree of a vertex is typically defined for unweighted graphs as $\deg(x) = |\text{adj}(x)|$. For compressed graphs, each edge represents a connection between every vertex in the first supervertex to every vertex

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

in the second. Therefore the weight of each edge in a compressed graph is implicitly the product of the vertex weights on either end. Thus, the degree for a compressed graph is

$$\deg(x) = \mathrm{vwgt}(x) * \sum_{v \in \mathrm{adj}(x)} \mathrm{vwgt}(v).$$

One characteristic of all these graph representations is that it does not have an implicit ordering in the same way that a matrix does. To capture this on a graph, we must label the vertices. Computing a reordering of the matrix is then reduced to the problem of relabeling the vertices in the graph. This will cause a change of placement of nonzeros in the matrix, but not the number of them.

The structure of a symmetric matrix A can be altered by performing symmetric (identical row and column) permutations. For unsymmetric matrices, ordering algorithms can either make the matrix structurally symmetric and continue, or use a model that supports unsymmetric matrices and generate separate row and column permutations.

It is also interesting to note that all of the graph models presented have undirected edges (or hyperedges). This is probably due to analytical ease more than necessity. There is, in fact, a very natural directed graph model for sparse unsymmetric factorization, but its deployment in actual codes is far from widespread.

2.1.2 Sparse Direct Solvers

To solve a large, sparse symmetric system of equations $Ax = b$ directly, the matrix A is decomposed into the product of LDU where L is a lower triangular matrix, D is diagonal, and U is upper triangular. The modified system of equations $LDUx = b$ can then be solved quickly through a series of triangular backsolves and diagonal scalings. If the original matrix is symmetric, then $U = L^{\mathrm{T}}$, and the storage and arithmetic required to form the factorization is halved.

If A is a sparse matrix, the LU factors often are significantly less sparse. Since sparsity of the factors is determined by the nonzero structure of the original matrix and the elimination order, it is prudent to consider permuting the system to $(P_rAP_c)(P_cx) = P_rb$ using row and column permutation matrices,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

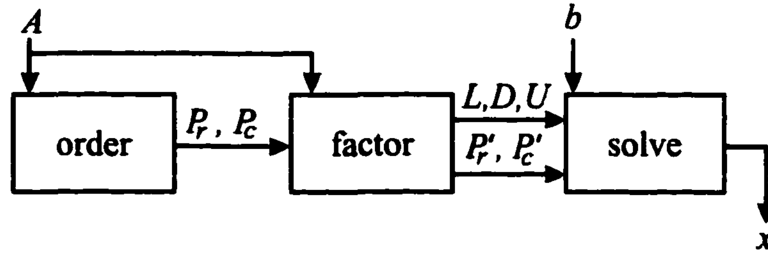


FIG. 2. *A "black box" formulation of a sparse direct solver.*

P_r, P_c, that reduces the storage and arithmetic work required for the process. If the original matrix A is symmetric, then $P_r = P_c$ to preserve symmetry.

If A is symmetric and positive definite, then the factorization is numerically stable for any symmetric permutation. Otherwise, it is still common to permute the system to reduce work and storage. Only during the factorization itself is care taken to maintain numeric stability, which can degrade the amount of work and storage saved.

The direct solution of a sparse symmetric linear system of equations can be described in three lines, corresponding to the three main computational steps: order, factor, and solve.

$$(P_r, P_c) = \text{order}(A) \quad (2.1)$$

$$(L, D, U, P_r', P_c') = \text{factor}(A, P_r, P_c) \quad (2.2)$$

$$x = \text{solve}(L, D, U, P_r', P_c', b) \quad (2.3)$$

where A is a sparse, symmetric, matrix, L is the lower triangular factorization, D is a diagonal scaling matrix, P_r and P_c are row and column permutations. P_r' and P_c' are the permutations after being modified by the factorization process for numerical stability, b is the known vector, and x is the unknown solution. Equivalently, we show a "black box" scheme in Fig. 2.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.2 Object-Oriented Programming

There is more to computer science than just transforming a mathematical algorithm to a working piece of software. There is an art to getting the software to be usable, flexible, to withstand years of use on multiple platforms. Software lasts far longer than hardware; it must because it is more difficult and expensive to build.

We chose to follow object-oriented paradigms in implementing our software because we firmly believed in several guiding principles in software design: interface simplicity, complexity management, flexibility, extensibility, safety, and efficiency. Object-oriented software does not automatically follow these principles, but they were easier to adhere to when implemented with object-oriented techniques than without.

Implementing these ordering algorithms using object-oriented software and following these principles as a fundamental part of the design has dramatically increased the amount of work and time spent in this research. However, we are confident that the "value added" by providing a robust and usable piece of software is in itself a new and significant contribution. It is only after having the software tool that we were able to get new insights and develop novel solutions. This research has convinced us that object-oriented programming not only produces better and more usable code, but also provides tools to solve problems in ways that are not natural otherwise.

2.2.1 Interface Simplicity

One of the cornerstones of providing a simple interface is to get the abstraction correct. We want to provide components that users are familiar with, and allow them to use the components in an intuitive way. The computations are formulated in terms of things like sparse matrices, vectors and permutations, and in terms of algorithms like orderings, factorizations, and solvers. Accordingly, we strive to provide abstractions for such entities with *minimal* interfaces to avoid code-bloat, and user confusion.

Previous ordering codes, primarily those implemented in C or Fortran are coded for generality and highly tuned for performance, often with little invested in interface simplicity. Users are commonly required to memorize data layouts for complicated data structures, handle storage requirements explicitly,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and use extensive combinations of codes and flags in the parameter lists of function calls. Thus the engineer or scientist using the software must think in terms of the software implementation and not in terms of the problem itself.

2.2.2 Complexity management

Even with very clean abstractions at the individual entity level, complexity can creep in as these pieces are assembled into larger, more complicated components. By aggregating objects, we stress the abstraction and its minimal interface. If the interface is too simple, the objects lack generality. If the interface is too complex, the objects lose usability. To manage the complexity of large collections of objects, we depend on encapsulation and layers of design.

Memory allocation should only be a concern for the simplest objects: character strings, vectors, etc. Higher level objects such as sparse matrices, ordering algorithms, and solvers should defer such matters to their constituent lower-level objects.

In Fortran77, for example, a sparse matrix is commonly represented as a collection several arrays. Algorithms are implemented as simple subroutines which read from and write to these arrays. Since there is no support for abstract data types in Fortran77, these subroutines tend to have long argument lists with several arguments per abstract data structure. Additionally, since Fortran 77 lacks dynamic memory allocation, users of the subroutine commonly must produce additional work arrays to be used internally by the subroutine. This forces users of high-level routines to be constantly aware of low-level details which increases the complexity of the software.

2.2.3 Flexibility

The key to achieving flexibility is loose coupling. We design separate abstractions for structural entities such as sparse matrices, permutations, and vectors on one side and for ordering and factorization algorithms one the other side. One can and should expect to swap in different ordering or factorization objects in the solver much in the same way that components in a stereo system can be swapped in and out. Once the stereo has been configured, it can be set to perform different tasks by pressing different

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

buttons. Advanced users can even use on-screen programming to perform more arcane functions. So too with flexible software. Ordering and factorization are performed only once in a series of systems with the same coefficient matrix but with different right hand sides, however the triangular solves must be repeated for every system in the series. A similar situation occurs with iterative refinement, where triangular solves must be repeated for a single run of the ordering and factorization algorithms.

Swapping components does not happen only with data structures and algorithms. In general, we want to be able to swap smaller components within a larger one. For instance, factorization is usually composed of a couple distinct phases (symbolic factorization and numerical factorization). Positive definite solvers and indefinite solvers differ only in the numerical factorization where the latter must pay attention to numerical stability. Therefore this is the only component we should swap out. By splitting a factorization algorithm in this way we provide the possibility of performing only symbolic work, for those who are not interested in the numerical results.

The challenge is that flexibility and simplicity are at odds with one another. The simplest interface to a solver would be just a black box that one throws a coefficient matrix and a right hand side in one end and produces the solution out the other end. While very easy to use, it would not be at all flexible. On the other hand, a very flexible implementation can expose too many details, making a steeper learning curve for users. We try to provide multiple entry points to our code and let the user decide which one is apropriate for their needs.

While little of this part of the design philosophy is radically new, we find that it is also not generally established practice either. This style of software is harder to implement correctly because it provides multiple possible execution paths. It resembles more the event-driven software of windowing programs than a simple subroutine that marches though its prescribed execution path.

2.2.4 Extensibility

Whereas flexibility allows us to push different buttons and interchange components, extensibility allows others to create new components and alter the effects of certain buttons. The best techniques we found

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

for ensuring extensibility in our codes were by enforcing decoupling, providing robust interfaces, and pointing out specific places for polymorphism.

Extensibility is not an automatic feature of a program written in an object-oriented language. Rather, it is a disciplined choice early in the design. In our implementations, we have very explicit points where the code was designed to be extended. Our intent is to keep the package open to other ordering algorithms and better heuristics as they become available.

2.2.5 Safety

When we talk about safety here, we are concerned with two major issues: protecting the user from making programming mistakes with components from our codes (compile time errors), and providing meaningful error handling when errors are detected at run-time. Earlier we argued that the simplicity of the interface increases the usability of the software, we add here that usability is further increased by safety.

Compile time safety is heavily dependent on features of the programming language. Any strongly typed language can adequately prevent users from putting square pegs in round holes. That is, we can prevent users from passing a vector as an argument that should really be a matrix.

Run time errors are more difficult to handle. Structural entities such as sparse matrices and permutations are inputs for factorization algorithms. When such an algorithm is run, it not only should detect if the inputs are valid, but also that they correspond to each other.

The biggest difficulty about error handling is that while we, the library writers know very well *how* to detect the error conditions when they occur, we must leave it to the user of the library to determine *what* action should be taken. Error handling is an important and all too often overlooked part of writing applications of any significant size. Because we are writing a multi-language based application and provide multiple interfaces (Fortran77, C, Matlab, C++) we immediately rejected using C++ exception handling.

Instead, we have made the more complicated classes self-aware. They are responsible for performing self-diagnostics to confirm that they are in a valid state. If any instances are not in a valid state, they are responsible for being able to report what errors were detected upon request.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.2.6 Efficiency

An efficient piece of software is one that makes judicious use of resources. Previous direct solver packages tend to be highly efficient, using compact data structures and algorithms that are intimately intertwined in the code. Decoupling the data structures from the algorithms and requiring them to interact through a high-level interface can add significant computational overhead. The compromise between flexibility and efficiency is determined by these interfaces.

Consider the means by which an algorithmic object accesses the data stored in a structural object, say a factorization operating on a sparse matrix. Correct object-oriented design requires full encapsulation of the matrix, meaning that the factorization algorithm must not be aware of the internal representation of the sparse matrix. The most general way to retrieve values of elements within a matrix is to query each pair of row and column indices. For a sparse $n \times n$ matrix, this means making n^2 queries for only $\mathcal{O}(n)$ data.

In practice, sparse matrix algorithms must take advantage of the storage format as an essential optimization. This is often done in object-oriented libraries like PETSc by "data-structure neutral" programming. This is accomplished by providing an abstract base class for a matrix or vector, and deriving a concrete implementations for each data-layout: row-major compressed sparse, column-major AIJ triples, blocked, etc. Then each concrete derived class must implement its own basic linear algebra functions. Given t types of matrices and n basic linear algebra subroutines for each, these libraries provide $t \times n$ virtual functions.

Our goal is very narrow: provide an ordering code that is as fast as any other software written in any other language, but is more usable, flexible, and extensible because we used object-oriented design and advanced programming paradigms. Our algorithms are far more complicated than a matrix-vector multiply and providing a set of algorithms for each possible representation of the matrix is not feasible. Even if we did general algorithms that operate on matrices regardless of their layout, we could not make any guarantees about performance. In short, we are more concerned about adding new algorithms, not adding more matrix formats. Spindle's ordering algorithms apply to general, sparse, symmetric matrices

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

that must be laid out in a specific way for efficient computations. We provide enough tools to convert to Spindle's graph and matrix classes.

Finally, we put some restrictions on the object-oriented techniques we made use of. For example, we do not define operator overloading for our matrices and vectors because our contribution is not in providing a general matrix class, but in providing a suite of ordering algorithms. We eschew the use of multiple inheritance, although we have found one occasion where its use is very helpful and so have used it there.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3. ALGORITHMS

This chapter presents three problems and applicable algorithms. The graph diameter problem and the pseudo-diameter algorithm are discussed in Section 3.1. The envelope/wavefront reduction problem and our Fast Sloan Algorithm is discussed at length in Section 3.2. Finally, we motivate the fill reduction problem and discuss a suite of greedy fill reducing ordering algorithms in Section 3.3.

3.1 Diameter of a Graph

Finding the diameter of a graph is not an ordering problem *per se*, but it is a necessary first step for many ordering algorithms such as the Reverse Cuthill-McKee (RCM) [17], Gibbs-Poole-Stockmeyer (GPS) [38, 55], Gibbs-King (GK) [37, 51], and Sloan [26, 52, 76] algorithms. In Section 3.1.1 we define the diameter of a graph and show that it requires $O(|V| * |E|)$ time, which is more expensive than the ordering algorithms themselves. In practice, a heuristic that computes a *pseudo-diameter*, or approximate diameter, is used employed. We step through a modern implementation of the pseudo-diameter algorithm in Section 3.1.2.

3.1.1 Definitions and Concepts

We begin by defining what we mean by the *diameter* of a graph. To do so, we need to also introduce the notion of *distance* between two vertices in the graph.

Definition 3.1 $\text{dist}_G(u, v)$, *the* distance *between two vertices* u, v *in a graph* G, *is the length of the shortest path between vertices* u *and* v *in* G.

Definition 3.2 *The* diameter of a graph G, *is a path from* s *to* e *such that* $\text{dist}_G(s, e)$ *is the maximum of all distances between any nodes in* G.

$$\text{length_of_diameter}(G) = \max_{u,v \in V(G)} \text{dist}(u, v) \quad .$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

To find the distance from one vertex in a graph to all other vertices, simply perform a breadth-first-search starting at that vertex, called the root. It is well known that the time spent in a breadth-first-search is $\mathcal{O}(|V| + |E|)$[16, pg. 472] where the initialization overhead is $\mathcal{O}(|V|)$ and the search itself is $\mathcal{O}(|E|)$. The last vertex visited by the breadth-first-search is also the farthest from the root. To find the diameter of a graph, one needs consider the farthest vertex from every possible root. Therefore a breadth-first-search needs to be performed for each vertex in the graph. Since the initialization overhead need only incurred once, computing the diameter takes $\mathcal{O}(|V| + |V| * |E|)$ or simply $\mathcal{O}(|V| * |E|)$ time.

The start and end vertices of a diameter of a graph are not necessarily unique. Given a specific pair of start and end vertices in a graph, the shortest path between them need not be unique either. While technically the diameter of a graph includes all the intermediate vertices in the path between the start and end vertices, we are only interested in these two. Therefore, we often talk about a pseudo-diameter algorithm selecting two vertices (the start and end vertices) instead of generating an entire path through the graph.

Before presenting the pseudo-diameter algorithm in detail, it is necessary to introduce a few additional concepts, all of them centering around the level structure of a graph.

Definition 3.3 *The* level structure *of a graph G is a sequence of* level sets $L_0, L_1, L_2, \ldots, L_h$ *where*

1. *all vertices in L_0 are adjacent only to vertices in L_0 or L_1*
2. *all vertices in L_h are adjacent only to vertices in L_h or L_{h-1}*
3. *all vertices in L_i, where $i \in [1, h-1]$ are adjacent only to vertices in L_{i-1}, L_i, or L_{i+1}.*

A level structure of a graph can be generated easily by a simple breadth-first-search (BFS) from the root vertex. Whereas a BFS simply visits vertices in a certain order, we must add the concept of level sets, or distinct levels to use a BFS to implement a pseudo-diameter algorithm. With the separation of distinct levels in a BFS, we also get the following useful concepts.

Definition 3.4 *The* height *of a level structure is the number of levels sets in the structure.*

Remark 3.1 *The length of a pseudo-diameter is the same as the height of a level structure rooted at either s or e.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Definition 3.5 *The* width *of a level structure is the maximum number of vertices in of any of its level sets.*

3.1.2 The Pseudo-Diameter Algorithm In Detail

The pseudo-diameter algorithm is presented as Algorithm 1. As presented, it takes a graph and a *shrinking strategy* as input. We will discuss the role of the shrinking strategy shortly. When completed, the pseudo-diameter algorithm returns two vertices s and e as the start and end vertices of the pseudo-diameter. The pseudo-diameter computation uses two BFS engines (line 1). The `forwardBFS` always uses the current start vertex as the root. The `reverseBFS` object uses candidates for the end vertex as the root. Initially, the start node is chosen to be any vertex of smallest degree (line 2) and the end node is unknown (line 4). Now the algorithm enters the main outer loop which does not exit until a suitable end node has been determined and all candidates have been exhausted. For each iteration of the outer loop, we perform the forward breadth-first-search (line 5), set the current diameter as the height of the level structure (equivalently, the distance from the last node in the BFS to the root), and get the set of all vertices that are in the farthest level set, called the *candidate set* (line 7). At this point in our discussion, we will skip lines 8, 11, 13–14 and 23–24 as they are all optimizations to improve performance. They will be revisited after the fundamentals are explained. For each candidate for end vertex in the candidate set (line 10), we do a reverse breadth-first-search. We are particularly interested in the candidate whose reverse breadth-first-search has the minimum width, so we initialize the local variable `min_width` to an arbitrarily large number (line 9).

If we find a candidate that has a narrower level structure than the forward breadth-first-search (line 15), then we make this candidate vertex the new start vertex (line 16) and restart the algorithm. The `break` in line 17, affects only the inner loop (lines 10–21) and jumps to line 22, since e is still undetermined, the outer loop (lines 4–22) starts a new iteration.

If the reverse breadth-first-search does not have more levels than the forward breadth-first-search, then it must have at least as many. Furthermore, if it is narrower than the most narrow reverse breadth-first-search so far (line 19), then we've found a new minimum width (line 20), and the candidate is chosen as the end vertex (line 21).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ALGORITHM 1 (The Pseudo-Diameter Algorithm.)

```
     [ s, e ] = PseudoDiameter(const Graph* g, ShrinkingStrategy* strategy)
     {
         // Create two breadth-first-search engines
 1.      BFS forwardBFS( g ), reverseBFS( g );

         // Initialize start and end vertices of pseudo-diameter
 2.      Graph::vertex s = g→vertex_of_minimum_degree();
 3.      Graph::vertex e = -1; // -1 is flag for non vertex

 4.      do // while e == -1

             // do BFS starting at start node 's'
 5.          forwardBFS.execute_at( s );

             // get candidateSet of end nodes
 6.          int diameter = forwardBFS.height();
 7.          Graph::vertexSet candidateSet =
                 forwardBFS.vertices_at_level( diameter );

             // shrink candidateSet to a manageable number
 8.          stategy→shrink( candidateSet );

 9.          int min_width = MAX_INT;
10.          for each candidate in candidateSet {

                 // do BFS from each candidate
                 // (abort if width() > 'min_width')
11.              reverseBFS.short_circuit_at( min_width );
12.              reverseBFS.execute_at( candidate );

                 // determine if candidate is appropriate as 'e'
13.              if ( reverseBFS.has_short_circuited() ) {
                     // reverseBFS is wider than a previous reverseBFS with e
14.                  continue ; // do nothing, skip this candidate
15.              } else if ( reverseBFS.height() > diameter &&
                         reverseBFS.width() < min_width ) {
                     // reverseBFS is better than the forwardBFS
                     // reset algorithm with candidate as new s
16.                  s = candidate;
17.                  e = -1;
18.                  break;
19.              } else if ( reverseBFS.width() < min_width ) {
                     // reverseBFS is narrower than any others
                     // make this new end node
20.                  min_width = reverseBFS.width();
21.                  e = candidate;
                 }
             } // end foreach
22.      } while ( e == -1 );

         // swap s & e if the reverseBFS is narrower than forwardBFS
23.      if ( forwardBFS.width() > reverseBFS.width() ) {
24.          return [ e, s ];
         }
25.      return [ s, e ];
     } // end function
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Now to revisit the optimizations. The most important is the shrinking strategy (line 8). Instead of performing a reverse breadth-first-search on all vertices that are farthest away from the start vertex, it is much faster to only try a select subset. Various heuristics can be applied such as: sorting the candidates by degree and choosing half of that set [76], choosing a single vertex of each degree [26], and sorting the candidate set by vertex degree and choosing the first five vertices that are not adjacent to any previously chosen vertex [72].

Another useful optimization is the short-circuiting mechanism (lines 11, 13–14). Since we know that we will not be accepting any candidate whose level structure is wider than the one we currently have, we can abort the breadth-first-search as soon a level set that is sufficiently large is detected. First, we enable the mechanism at line 11. Recall that if this is the first time through the inner loop, `min_width` is arbitrarily large, so we know it will not be triggered. At line 13, we test if the reverse breadth-first-search has triggered. If so, we know this candidate will be rejected and we continue immediately to the next candidate (line 14).

Finally we have our start and end vertices (line 23), but it is possible for the reverse breadth-first-search to be narrower than the forward breadth-first-search. If this is the case, then we simply return the start and end vertex pair reversed (line 24). Otherwise, we return them as is (line 25).

3.1.3 History

A modern implementation of the pseudo-diameter algorithm has become significantly faster and more complex in the last thirty years. Gibbs, Poole, and Stockmeyer [38] observed that at the end of a pseudo-diameter computation, there are two level structures available: one from the start node and one from the end node. They observed that some post-processing of these two level structures can obtain a third level structure whose width is usually less than the other two. George and Liu [31] recommended terminating the construction of any level set as soon as the width exceeded that of the narrowest level set found so far. Lewis [55] recommended that candidate vertices be sorted by degree, since pseudoperipheral vertices tend to have low degree.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Sloan [76] incorporated both of these modifications into his algorithm. He also observed that vertices with high degrees are often not selected as potential start or end vertices. He therefore introduced a the first shrinking strategy that took only the first half of candidate vertices sorted by degree. Duff, Reid, and Scott [26] and later Reid and Scott [72] have introduced more aggressive shrinking strategies. In the latter work, they restrict the candidate set to no more than five in order of increasing degree and omitting any vertes that is a neighbor of a vertex already considered.

By being so restrictive, the algorithm generates level structures for a minimal number of vertices and can greatly improve its execution time. Furthermore, Reid and Scott [72] point out that the width of the level sets from the start and end vertices are not necessarily the same. If the latter is more narrow, it is advantageous to switch the start and end vertices.

3.2 Envelope/Wavefront Reduction

The envelope/wavefront reduction problem[1] is a classic graph ordering problem that has many applications that extend far beyond numerical linear algebra or sparse matrix computations.

We first introduce relevant definitions and notation (Section 3.2.1) and then derive formulae for computing the envelope and wavefront for certain model problems (Section 3.2.2). We review the history of algorithmic development for this important problem (Section 3.2.3) and discuss our enhancements to the algorithm (Section 3.2.4). Our Fast Sloan Algorithm is discussed in detail (Section 3.2.5) and then its asymptotic complexity is analyzed (Section 3.2.6). Next we discuss our related algorithm of Sloan Refinement (Section 3.2.7). We end this section with a brief overview of some promising applications (Section 3.2.8) and a brief examination of the impact our published research [52] in this problem has had.

3.2.1 Definitions and Notation

For the purposes of this discussion, we will restrict ourselves to structurally symmetric matrices (see Definition 2.1, page 4). This is a classic restriction in the literature and greatly simplifies the presentation of new material. Later, we will show how this class of algorithms can be generalized to the unsymmetric

[1] also called *skyline* or *profile* reduction

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

case. For the sake of simplicity, we will assume that all the diagonal elements of A are nonzero. While this is certainly the case for symmetric, positive definite matrices, it is not true in general. However, since these algorithms center on the undirected general graph model — which does not capture self-edges — this is not an unreasonable assumption.

Definition 3.6 *Consider a large, sparse, structurally symmetric matrix, A. The* envelope *of the matrix A, is defined as the set of all matrix entries a_{ij} between and including the first nonzero of the row, up to and excluding the diagonal.*

$$\mathrm{env}(A) = \{a_{i,j} \ni: f_i(A) \leq j < i,\ 1 \leq i \leq n\}.$$

where $f_i(A)$ is the column of the first nonzero entry in the ith row of A.

The envelope of a symmetric matrix is easily visualized. Picture a sparse, structurally symmetric matrix, then remove the upper triangle (including the diagonal) and the leading zero elements in each row. The remaining elements (whether nonzero or zero) are in the envelope of the matrix.

For analysis, we also introduce the *row width* of a sparse, structurally symmetric matrix. We will use this concept to define the *bandwidth* of a matrix and show the relation between the bandwidth and the profile.

Definition 3.7 *Consider a large, sparse, structurally symmetric matrix, A. The* row width *of the ith row, $\mathrm{rw}_i(A)$, is the difference between i and the column index of the first nonzero element on the ith row, or equivalently,*

$$\mathrm{rw}_i(A) = \max_{j\,:\,a_{ij} \neq 0} (i - j).$$

Definition 3.8 *Given a large, sparse structurally symmetric matrix, A, the* bandwidth *of A is defined as the max row width:*

$$\mathrm{bw}(A) = \max_{1 \leq i \leq n} \mathrm{rw}_i(A).$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The bandwidth of a matrix is very important for a class of solvers called *banded solvers* that store matrices in a banded format. As we are interested in general sparse matrices, we have little use for this specialized format. It is worth noting, however, that bandwidth reducing orderings, such as Reverse Cuthill McKee (RCM) [17, 59], are critically dependent on the pseudo-diameter algorithm. In fact, most of the time spent doing an RCM ordering is actually spent doing the pseudo-diameter computation. Although we've done no algorithmic work in the bandwidth problem, we provide an RCM implementation in our software since (1) we had a first-rate pseudo-diameter implementation (2) the rest was easy to implement and (3) most other RCM implementations have not kept current with the faster (and better quality) pseudo-diameter codes.

We now show that there is a relationship between the bandwidth reduction problem and the wavefront reduction problem: they can both be stated in terms of the row width.

Remark 3.2 *Given a large, sparse structurally symmetric matrix, A, the size of the envelope of A is the sum of the row widths:*

$$|\mathrm{env}(A)| = \sum_{i=1}^{n} \mathrm{rw}_i(A).$$

Therefore, we can state the bandwidth reduction problem in terms of minimizing the *maximum* row width, and we can state the envelope reduction in terms of minimizing the *sum* of the row widths.

While we have now adequately defined the envelope reduction problem, we have yet to define the analogous wavefront reduction problem. We define the wavefront below and then show how it too can be defined in terms of row width.

Consider the ith step of Cholesky factorization where only the lower triangle of A is stored. When factoring the ith column, there may be contributions from previously factored columns that need to be accumulated. Formally, any k^{th} row ($k \geq i$) is called *active* if there exists a previously factored column ℓ ($\ell \leq i$) such that $a_{k\ell} \neq 0$. The set of all active equations for each column i of A is called the *wavefront* of A, $\mathrm{wf}_i(A)$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In terms of the envelope, the ith wavefront is the set of rows i^{th} column that are within the envelope of the matrix, including the ith row itself. We can also define the ith wavefront in terms of the general undirected graph of A. In the graph of A, the ith wavefront consists of the vertex i together with the set of vertices adjacent to the vertices numbered from 1 to i. Formally, the ith wavefront is

$$\mathrm{wf}_i(A) = v_i \cup \mathrm{adj}\left(\{v_1, v_2, \ldots, v_i\}\right).$$

The vector of n wavefronts is often summarized into scalar values, such as the *maximum wavefront* and *mean-square wavefront* as defined below:

$$\mathrm{maxwf}(A) = \max_{1 \le i \le n} \{|\mathrm{wf}_i(A)|\}. \quad (3.1)$$

$$\mathrm{mswf}(A) = \frac{1}{n} \sum_{i=1}^{n} |\mathrm{wf}_i(A)|^2. \quad (3.2)$$

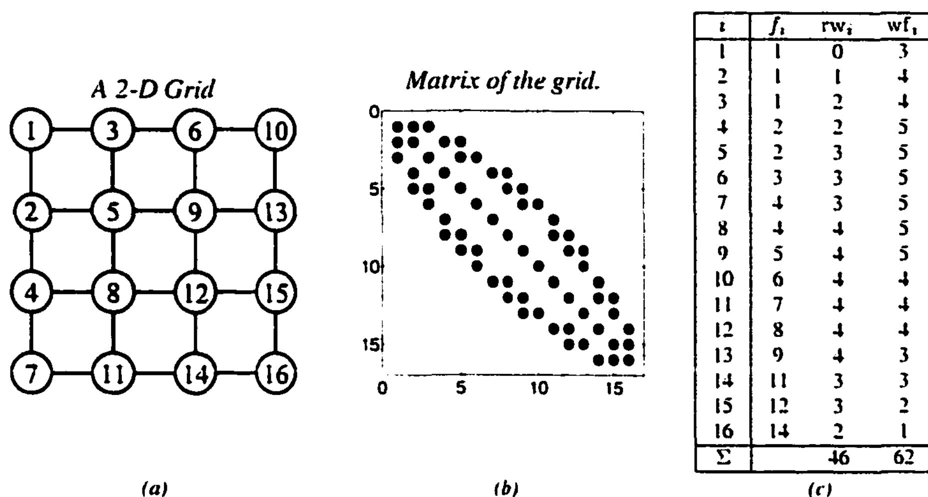
There are a class of solvers called frontal solvers whose performance is determined largely by these wavefront characteristics. The maximum wavefront size measures the maximum storage needed for a frontal solver during factorization, while the mean square wavefront measures the number of floating point operations in the factorization. Duff, Reid, and Erisman [22] discuss the application of wavefront reducing orderings to frontal factorization.

Finally, we need to show the relationship between the envelope and the wavefront.

Observation 3.1 *The sum of the wavefronts of a matrix equals the size of the envelope, plus n.*

$$\sum_{i=1}^{n} |\mathrm{wf}_i(A)| = n + \sum_{i=1}^{n} \mathrm{rw}_i(A) = n + |\mathrm{env}(A)|.$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



i	f_i	rw_i	wf_i
1	1	0	3
2	1	1	4
3	1	2	4
4	2	2	5
5	2	3	5
6	3	3	5
7	4	3	5
8	4	4	5
9	5	4	5
10	6	4	4
11	7	4	4
12	8	4	4
13	9	4	3
14	11	3	3
15	12	3	2
16	14	2	1
Σ		46	62

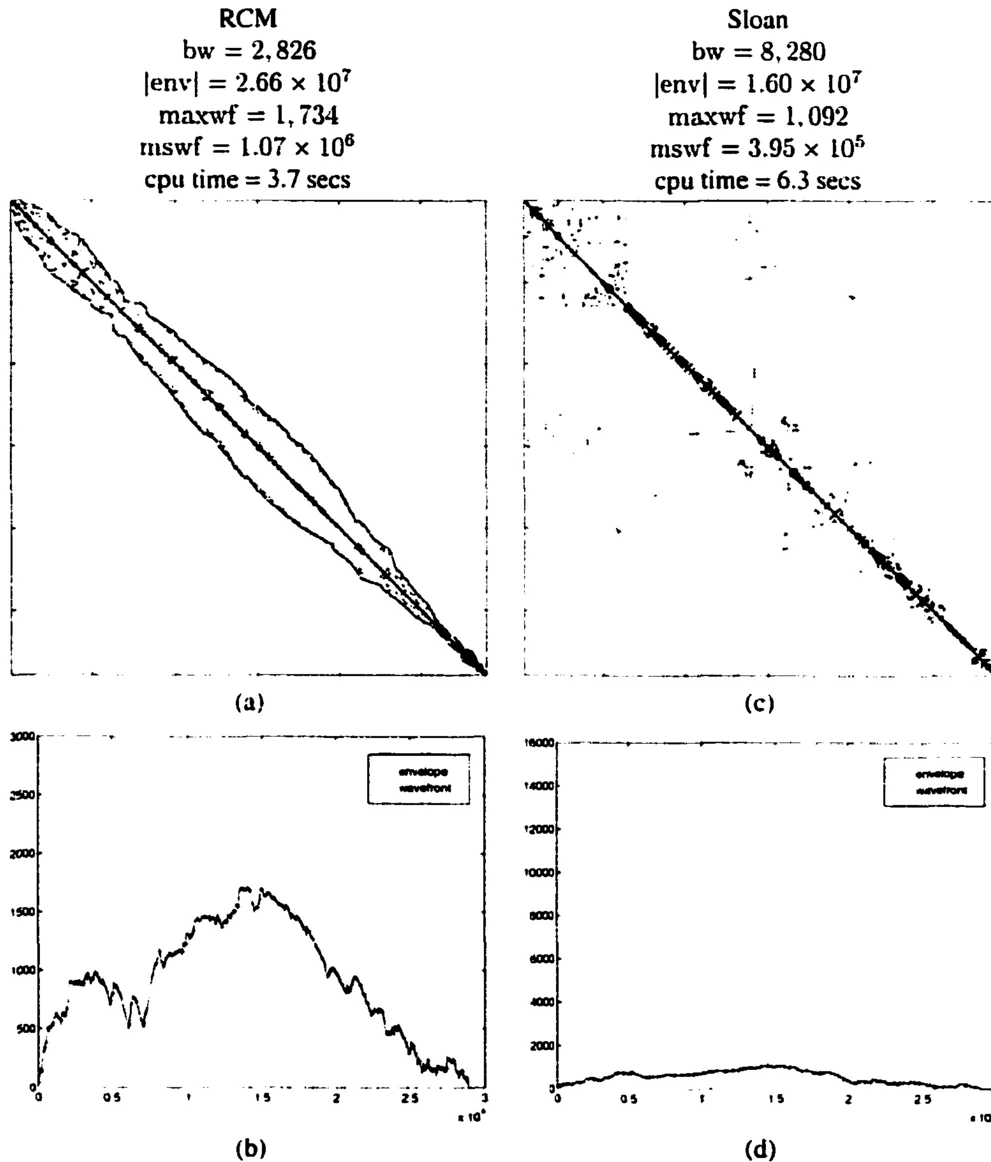
FIG. 3. *Example of row-widths and wavefronts. A two dimensional mesh and its vertex ordering are shown in (a), the structure of the associated matrix is in (b), and a table of pertinent data is in (c).*

We now provide a simple example to illustrate these characteristics we have defined. Figure 3(a) shows a small two-dimensional grid and Figure 3(b) shows the structure of its associated matrix A. Figure 3(c) is a table showing the row-widths and wavefronts of the matrix A. From this table, we can compute the parameters $\text{esize}(A) = 46$, $\text{bw}(A) = 4$, $\text{maxwf}(A) = 5$, and $\text{mswf}(A) \approx 16.4$.

All of these parameters are sensitive to the nonzero structure and the ordering of the matrix. If we numbered the vertices in Figure 3 in a spiral fashion beginning with vertex one and numbering from the outside towards the inside, the permuted matrix A' yields $\text{esize}(A') = 59$, $\text{bw}(A') = 11$, $\text{maxwf}(A') = 7$, and $\text{mswf}(A') \approx 24.8$.

We further illustrate the influence of ordering on envelope and wavefront for a real problem, and using four real-world orderings. Figure 4, illustrates just how different these orderings are. For the RCM, Sloan, spectral, and hybrid (spectral with a modified Sloan refinement) orderings we show the nonzero structure of a matrix, plot the row widths and wavefront sizes, and tabulate some relevant data. Note that the area under the wavefront curve is actually larger than the area under the row width curve. However, the row width varies so wildly, it appears as if we had colored in the area underneath it when, in fact, we only plotted the top of it, like we did the wavefront. The wavefront, by comparison, is much less volatile.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



Continued on next page . . .

FIG. 4. *Structure of* bcsstk30 *with four orderings. (a) is the nonzero structure using the RCM ordering. (b) is a plot of the row widths and wavefront sizes for the RCM ordering. (c) is the nonzero structure for the Sloan ordering. (d) is the plot of the row widths and wavefront sizes for the Sloan ordering.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

... Continued from previous page.

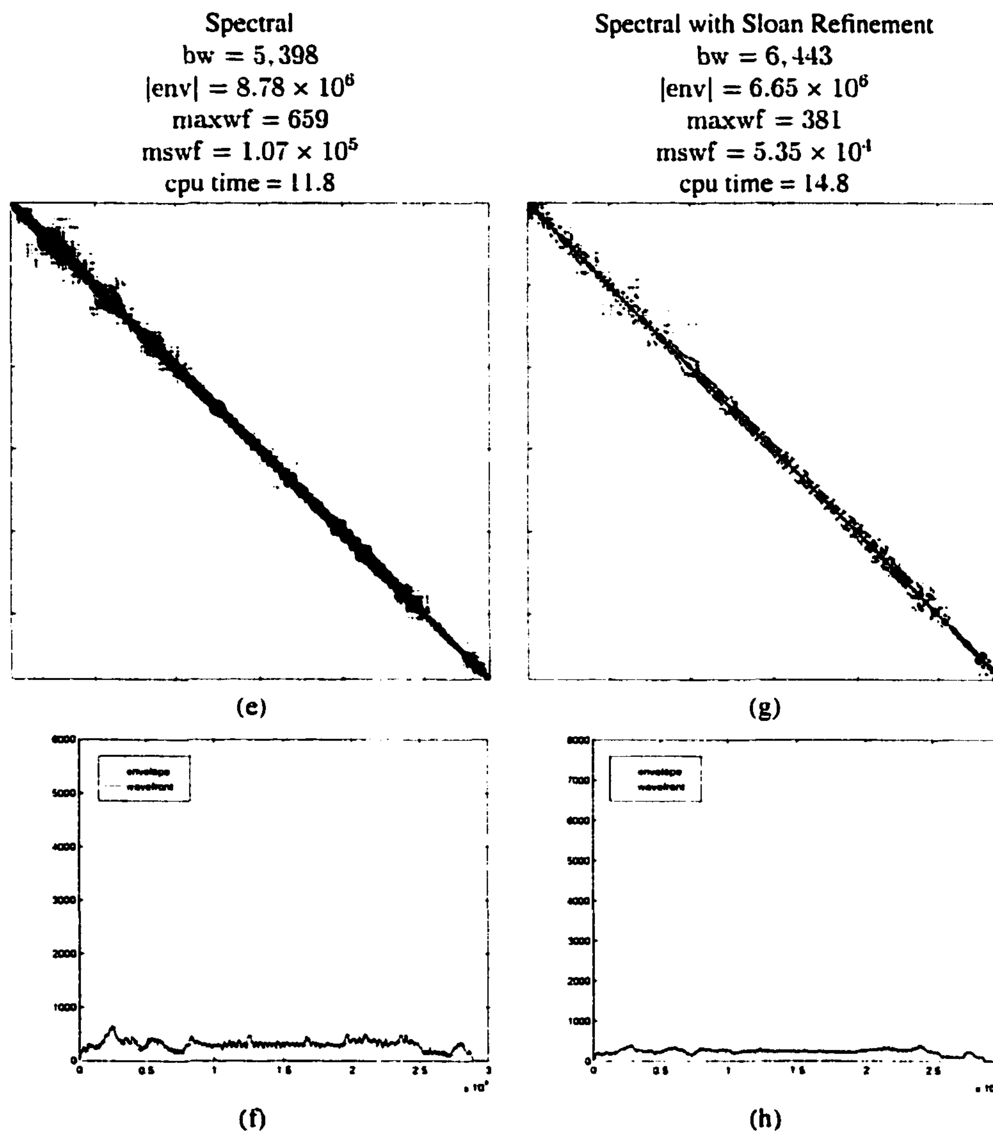


FIGURE 4 (Continued): (e) is the nonzero structure using the spectral ordering. (f) is a plot of the row widths and wavefront sizes for the spectral ordering. (g) is the nonzero structure for the Spectral ordering after Sloan refinement. (h) is the plot of the row widths and wavefront sizes for the spectral ordering after Sloan refinement.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.2.2 Model Problems

It is interesting to note that all of these envelope and wavefront characteristics can be computed for a wide range of model problems and orderings. Here we compute them for three types of rectangular grids, each with two different orderings. For the purposes of this discussion, we assume an $m \times n$ grid with $m \leq n$ without loss of generality. The grid shown in Figure 3(a) is an example of a grid using a *five point stencil* — meaning, that each node depends on itself and its four neighbors (north, south, east and west). A *seven point* stencil includes either northeast and southwest or northwest and southeast neighbors, and a *nine point* stencil contains all the above.

In this analysis we consider two orderings; *vertical* and *diagonal*. In describing both orderings it is necessary to visualize the grid oriented horizontally. In a vertical ordering, the vertices are numbered by columns; top to bottom, left to right. A diagonal ordering is the optimal ordering for square, five point grids. In our case, the diagonal ordering always starts from the node on the top left (northwest) corner of the grid, and numbers them immediately below and to the right. This pattern is repeated, numbering vertices along diagonals running from the bottom left up and to the right. For some rectangular grids, the detail about direction along the diagonal makes some difference.

To compute all of these parameters, we need to compute the sum of the wavefront ($\sum |\mathrm{wf}_i(A)|$) and sum of wavefront squared ($\sum |\mathrm{wf}_i(A)|^2$). For all of these problems, we break up the mesh into different parts and analyze them individually. Then, we sum the parts back together at the end. We use the following well-known identities in our analysis.

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}. \tag{3.3}$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}, \tag{3.4}$$

$$\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}. \tag{3.5}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

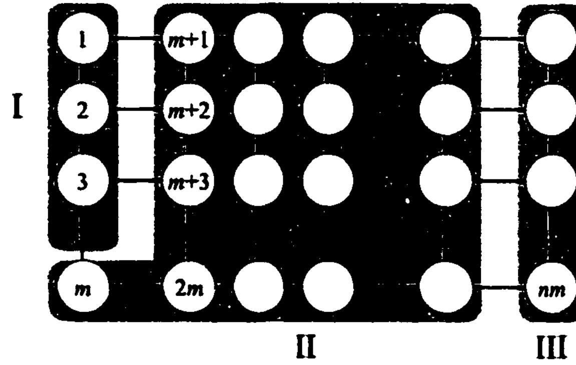


FIG. 5. *Domains of a Vertical Ordering on rectangular 5-point grid.*

3.2.2.a Vertical Ordering on 5-point Mesh

We begin with the simplest mesh, and the simplest ordering. Figure 5 shows how we break up the mesh into different domains. Domain (I) consists of the first $m - 1$ vertices. the first vertex has a wavefront of 3 (two adjacent vertices, plus itself). Each subsequent vertex adds two new vertices to the wavefront, while removing the previously numbered node. This continues for the first $m - 1$ vertices. The mth vertex does not change the size of the wavefront, which is $m + 1$. Domain (II) consists of the mth vertex, and the following $m(n - 2)$ vertices. Throughout this domain, the size of the wavefront is uniformly $m + 1$. The final m vertices in domain (III), are all in the wavefront when the last vertex of domain (II) is numbered. Numbering each vertex in this domain effectively reduces the wavefront by 1.

Therefore, we can write the size of the wavefront for this problem explicitly as

$$|\text{wf}_i(R_{5v})| = \begin{cases} i + 2 & i < m \\ m + 1 & m \leq i \leq m(n-1) \\ mn - i + 1 & m(n-1) < i \leq mn \end{cases} , \tag{3.6}$$

where R_{5v} stands for a matrix based on a rectangular mesh with a 5-point stencil ordered vertically. These equations can be easily checked with the square 5-point mesh shown in Figure 3. Using Equation 3.6, we

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

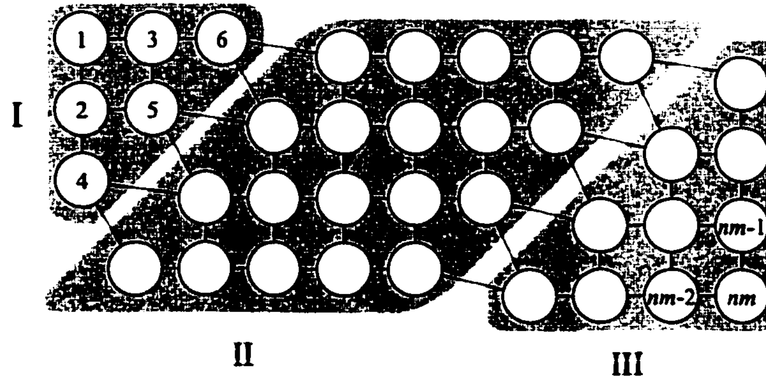


FIG. 6. *Domains of a Diagonal Ordering on rectangular 5-point grid.*

can compute the sum of the wavefront sizes and the sum of the wavefront squared,

$$\begin{aligned}\sum |\mathrm{wf}_i(R_{5v})| &= \left[\sum_{i=1}^{m-1}(i+2)\right] + \left[\sum_{i=m}^{m(n-1)}(m+1)\right] + \left[\sum_{i=m(n-1)+1}^{mn}(mn-i+1)\right] \\ &= \left[\sum_{i=1}^{m} i - m + 2(m-1)\right] + \left[(m+1)(mn-2m+1)\right] + \left[\sum_{i=1}^{m} i\right] \\ &= n(m^2+m) - m^2 + m - 1,\end{aligned}$$

and

$$\begin{aligned}\sum |\mathrm{wf}_i(R_{5v})|^2 &= \left[\sum_{i=1}^{m-1}(i+2)^2\right] + \left[\sum_{i=m}^{m(n-1)}(m+1)^2\right] + \left[\sum_{i=m(n-1)+1}^{mn}(mn-i+1)^2\right] \\ &= \left[\frac{m(m-1)(2m-1)}{6} + 4\frac{m(m-1)}{2} + 4(m-1)\right] \\ &\quad + \left[(m+1)^2(mn-2m+1)\right] + \left[\sum_{i=1}^{m} i^2\right] \\ &= n(m^3+2m^2+m) - \frac{4}{3}m^3 - m^2 + \frac{7}{3}m - 3.\end{aligned}$$

3.2.2.b Diagonal Ordering on 5-point Mesh

We are now ready to apply the same technique for a diagonal ordering. In Figure 6, we show a 4×9 mesh, since drawing one that it arbitrarily sized is difficult with a diagonal ordering. We also show in Figure 6 how the separation into domains is different. In this case, domain (I) consists of the first $m(m-1)/2$ vertices. the first vertex has a wavefront of 3, the next two have a wavefront of 4, the next three, 5, and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

so on for the rest of the domain. Like before, domain (II) has a wavefront that is uniformly $m + 1$, but it is smaller, containing only $m(n - m)$ vertices. The final $m(m + 1)/2$ vertices in domain (III) have an unusual pattern of wavefronts that requires some investigation. The first $m - 1$ vertices of this last domain have a wavefront of $m + 1$, the next $m - 1$ have a wavefront of m, then $m - 2$ vertices, each with a wavefront of $m - 1$ and so on. The last two vertices have wavefronts of 2 and 1, respectively.

While it is not so hard to describe the wavefronts of each vertex using the diagonal ordering. It is difficult to provide a closed-form equation like we did for the vertical ordering of a 5-point grid. Luckily, we are interested in the sum of the wavefronts and the sum of the squares, which can be written in a simple form. For the domain (I), we can show that the sum of the $m(m - 1)/2$ wavefronts is

$$\sum_{i=1}^{m(m-1)/2} |\text{wf}_i(R_{5d})| = \sum_{i=1}^{m-1} i(i+2).$$

Similarly, we can show that the sum of the $m(m + 1)/2$ wavefronts for domain (III) is equal to

$$\sum_{i=nm-m(m+1)/2+1}^{nm} |\text{wf}_i(R_{5d})| = (m-1)(m+1) + \sum_{i=1}^{m-1} i(i+1) + 1.$$

Thus, the sum of the wavefronts is

$$\begin{aligned}
\sum_{i=1}^{n} |\text{wf}_i(R_{5d})| &= \left[\sum_{i=1}^{m-1} i(i+2)\right] + \left[m(n-m)(m+1)\right] \\
&\quad + \left[(m-1)(m+1) + \sum_{i=1}^{m-1} i(i+1) + 1\right] \\
&= 2\sum_{i=1}^{m-1} i^2 + 3\sum_{i=1}^{m-1} i + n(m^2+m) - m^3 - m^2 + m^2 \\
&= n(m^2+m) - \frac{1}{3}m^3 + \frac{1}{2}m^2 - \frac{7}{6}m,
\end{aligned}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

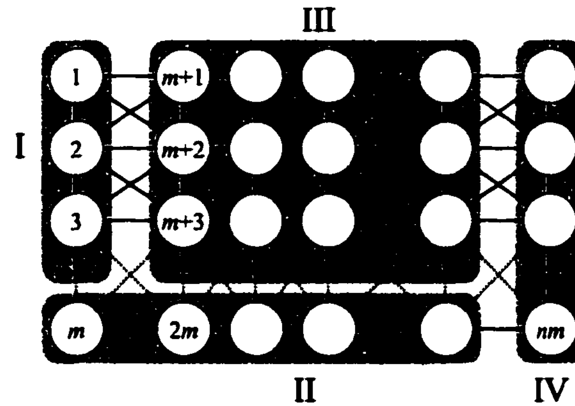


FIG. 7. *Domains of a Vertical Ordering on rectangular 9-point grid.*

and the sum of the wavefront squared is

$$\begin{aligned}
\sum_{i=1}^{n} |\mathrm{wf}_i(R_{5d})|^2 &= \left[\sum_{i=1}^{m-1} i(i+2)^2\right] + \left[m(n-m)(m+1)^2\right] \\
&\quad + \left[(m-1)(m+1)^2 + \sum_{i=1}^{m-1} i(i+1)^2 + 1\right] \\
&= nm(m+1)^2 + 2\sum_{i=1}^{m-1} i^3 + 6\sum_{i=0}^{m-1} i^2 + 5\sum_{i=1}^{m-1} i + (m+1)^2(-m^2+m-1) + 1 \\
&= nm(m+1)^2 - \frac{m}{2}(m^3+5).
\end{aligned}$$

3.2.2.c Vertical Ordering on 9-point Mesh

Next we present the computations for a vertical ordering on a rectangular grid using a 9 point mesh. The construction is much the same as for a 5 point mesh. For domain (I) the first vertex starts with a wavefront of four and each subsequent vertex up to the $m - 1$st adds an additional 2 vertices, and subtracts the previous vertex for a net increase of 1 from the previous wavefront size. The mth vertex, and every whole multiple up to $n - 1$, belongs to domain (II); each of which having a wavefront of $m + 1$. The vertices in domain (III) each have a wavefront of $m + 2$. Domain (IV) is exactly like domain (III) of the vertically ordered 5-point mesh. Every vertex starts off in the wavefront and as they are numbered all their predecessors are removed. Like the vertical ordering of a 5 point grid, we can easily write an explicit

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

equation for the size of the wavefronts for a vertically ordered 9 point grid.

$$|\mathrm{wf}_i(R_{9v})| = \begin{cases} i+3 & i < m \\ m+1 & m \le i \le m(n-1), \quad i \mod m = 0 \\ m+2 & m \le i \le m(n-1), \quad i \mod m \ne 0 \\ mn - i + 1 & m(n-1) < i \le mn \end{cases} . \tag{3.7}$$

Using Equation 3.7, we can compute the sum of the wavefront sizes and the sum of the squares,

$$\begin{aligned} \sum |\mathrm{wf}_i(R_{9v})| &= \left[\sum_{i=1}^{m-1}(i+3)\right] + \left[\sum_{i=1}^{n-1}(m+1)\right] + \left[\sum_{i=m}^{(m-1)(n-1)}(m+2)\right] \\ &\quad + \left[\sum_{i=m(n-1)+1}^{mn}(mn-i+1)\right] \\ &= \left[\sum_{i=1}^{m-1} i + 3(m-1)\right] + \left[(m+1)(n-1)\right] + \left[(m+2)(m-1)(n-2)\right] \\ &\quad + \left[\sum_{i=1}^{m} i\right] \\ &= \left[\frac{m(m-1)}{2} + 3(m-1)\right] + \left[(m+1)(n-1)\right] + \left[(m+2)(m-1)(n-2)\right] \\ &\quad + \left[\frac{m(m+1)}{2}\right] \\ &= n(m^2 + 2m - 1) - m^2 - m. \end{aligned}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

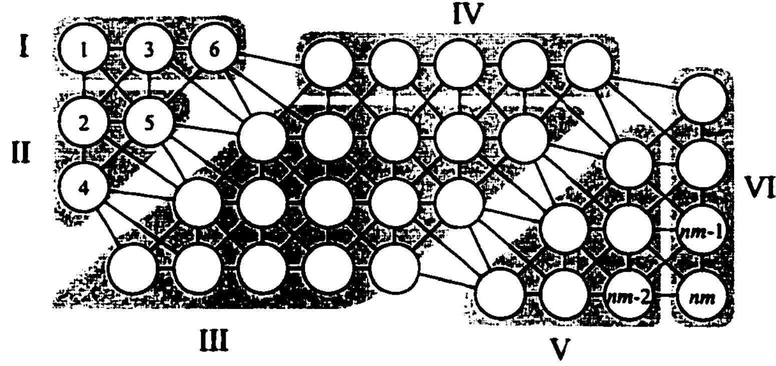


FIG. 8. *Domains of a Diagonal Ordering on rectangular 9-point grid.*

and

$$
\begin{aligned}
\sum |\mathrm{wf}_i(R_{9v})|^2 &= \left[\sum_{i=1}^{m-1}(i+3)^2\right] + \left[\sum_{i=1}^{n-1}(m+1)^2\right] \\
&+ \left[\sum_{i=m}^{(m-1)(n-1)}(m+2)^2\right] + \left[\sum_{i=m(n-1)+1}^{mn}(mn-i+1)^2\right] \\
&= \left[\sum_{i=1}^{m-1} i^2 + 6\sum_{i=1}^{m-1} i + 9(m-1)\right] + \left[(m+1)^2(n-1)\right] \\
&+ \left[(m+2)^2(m-1)(n-2)\right] + \left[\sum_{i=1}^{m} i^2\right] \\
&= \left[\frac{2m^3-3m^2+m}{6} + 6\frac{m(m-1)}{2} + 9(m-1)\right] + \left[(m+1)^2(n-1)\right] \\
&+ \left[(m+2)^2(m-1)(n-2)\right] + \left[\frac{2m^3+3m^2+m}{6}\right] \\
&= n(m^3+4m^2+2m-3) - \frac{1}{3}(4m^3+12m^2-13m+6).
\end{aligned}
$$

3.2.2.d Diagonal Ordering on 9-point Mesh

Finally, we present a diagonal ordering on a 9-point stencil. Figure 8 shows a 4×9 rectangular grid with a 9-point stencil. It is slightly deformed to put additional space between vertices of different domains. This time, we identify six different domains. Furthermore, all vertices in one domain are not exhausted before beginning the second. We start with domains (I) and (II). The first vertex has a wavefront of four vertices. Each time we jump from domain (I) to domain (II), the size of the wavefront increases by one. Similarly, each time we jump from domain (II) domain (I) the wavefront increases by one. Therefore all

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the wavefront sizes in (I) are even numbers and all the wavefront sizes in domain (II) are odd. The sum of the wavefronts for domain (I) is simply $\sum_{i=1}^{m-1}(2i+2)$. The sum of the wavefronts for domain (II) is $\sum_{i=1}^{m-2} i(2i+3)$. All $(m-n)(m-1)$ vertices in domain (III) have a wavefront of $2m-1$ vertices and all $n-m$ vertices in domain (IV) have a wavefront of $2m$ vertices. Domains (V) and (VI) interplay similarly as (I) and (II) did, only in this case every time the ordering switches domains, the wavefront reduces by one. The sum of the wavefronts for domain (V) is $\sum_{i=1}^{m-1} i(2i+1)$. The sum of the wavefronts for domain (VI) is $1+\sum_{i=1}^{m-1} 2i$.

Putting this all together we get

$$\begin{aligned}
\sum |\mathrm{wf}_i(R_{9d})| &= \left[\sum_{i=1}^{m-1}(2i+2)\right] + \left[\sum_{i=1}^{m-2} i(2i+3)\right] + \left[(m-1)(n-m)(2m-1)\right] \\
&\quad + \left[(n-m)(2m)\right] + \left[\sum_{i=1}^{m-1} i(2i+1)\right] + \left[1+\sum_{i=1}^{m-1} 2i\right] \\
&= 4\sum_{i=1}^{m-1} i^2 + 8\sum_{i=1}^{m-1} i + \sum_{i=1}^{m-1} 2 - (m-1)[2(m-1)-3] \\
&\quad + (n-m)[(m-1)(2m-1)+2m] + 1 \\
&= n(m^2-m+1) - \frac{2}{3}m^3 + m^2 - \frac{4}{3}m.
\end{aligned}$$

and

$$\begin{aligned}
\sum |\mathrm{wf}_i(R_{9d})|^2 &= \left[\sum_{i=1}^{m-1}(2i+2)^2\right] + \left[\sum_{i=1}^{m-2} i(2i+3)^2\right] \\
&\quad + \left[(m-1)(n-m)(2m-1)^2 + (n-m)(2m)^2\right] \\
&\quad + \left[1+\sum_{i=1}^{m-1} i(2i+1)^2\right] + \left[1+\sum_{i=1}^{m-1}(2i)^2\right] \\
&= 8\sum_{i=1}^{m-1} i^3 + 24\sum_{i=1}^{m-1} i^2 + 18\sum_{i=1}^{m-1} i + 4(m-1) - (m-1)[2(m-1)+3]^2 \\
&\quad + (n-m)[(m-1)(2m-1)^2 + m^2] + 1 \\
&= n(4m^3 - 4m^2 + 5m - 1) - 2m^4 + 4m^3 - 6m^2 + 3m - 2.
\end{aligned}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.2.2.e Summary

Using Observation 3.1 to relate envelope size to the sum of wavefronts, Equation 3.2 which defines the mean square wavefront, and all the results for our model problems, we can define the esize() and mswf() for all of our model problems.

$$
\begin{aligned}
\text{esize}(R_{5v}) &= n(m^2) - m^2 + m - 1 \\
\text{mswf}(R_{5v}) &= m^2 + 2m + 1 - \frac{1}{n}(\frac{4}{3}m^2 + m - \frac{7}{3} + \frac{3}{m}) \\
\text{esize}(R_{5d}) &= n(m^2) - \frac{1}{3}m^3 + \frac{1}{2}m^2 - \frac{7}{6}m \\
\text{mswf}(R_{5d}) &= m^2 + 2m + 1 - \frac{1}{2n}(m^3 + 5) \\
\text{esize}(R_{9v}) &= n(m^2 + m - 1) - m^2 \\
\text{mswf}(R_{9v}) &= m^2 + 4m + 2 - \frac{3}{m} - \frac{1}{n}(\frac{4}{3}m^2 + 4m - \frac{13}{3} + \frac{2}{m}) \\
\text{esize}(R_{9d}) &= n(2m^2 - 2m + 1) - \frac{2}{3}m^3 + m^2 - \frac{4}{3}m \\
\text{mswf}(R_{9d}) &= 4m^2 - 4m + 5 - \frac{1}{m} - \frac{1}{n}(2m^3 - 4m^2 + 6m - 3 + \frac{2}{m})
\end{aligned}
$$

The behavior of the seven point stencil is, either like the five point or the nine point depending on the relationship between the ordering and the extra pair of neighbors. For example in a vertical ordering (i.e., from top to bottom, left to right), if the stencil connects Northeast and Southwest neighbors, the seven point stencil behaves exactly like the five point grid. Similarly, if the diagonal ordering orders along the direction of the diagonal edges, it behaves like a five point grid. However, if the diagonal edges are oriented the other way, then the results are the same as a nine point stencil.

The most striking result is that the diagonal ordering is worse than the vertical ordering for 9 point stencils. Criticism about spectral orderings' tendency to order rectangular grids with a nine point stencil along short columns instead of along the diagonal was an instigating factor into this investigation. We found the results to be most enlightening.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.2.3 History

There is an interesting history of work done in the envelope/wavefront reducing orderings. Much of the work was inspired by frontal solvers which depended critically on the wavefront being minimized. Early algorithms to reduce the envelope include the King [51], Gibbs-Poole-Stockmeyer [38], and Gibbs-King [37] algorithms. Currently, the best two heuristics for the envelope/wavefront reduction problem are a combinatorial algorithm by Sloan [76] and an algebraic algorithm called the *spectral* ordering [9]. We present a brief history of each of these algorithms in preparation for describing our contributions to the envelope/wavefront reduction problem.

3.2.3.a King, Gibbs-Poole-Stockmeyer, and Gibbs-King

King [51] wrote one of the earliest algorithms for envelope reduction. His work was an essentially greedy implementation to minimize the wavefront. King's algorithm breaks ties by choosing the vertex that was active for the longest period of time (equivalently, the active node with the lowest numbered neighbor). King was also the first to discuss the importance of what Sloan calls *preactive nodes*. King's algorithm did not have a method for selecting a starting node. Instead he suggests that the user choose several and keep the ordering with the best characteristics.

Gibbs, Poole, and Stockmeyer [38] tried to simultaneously minimize the bandwidth and the envelope of a sparse matrix. To accomplish this, they concentrated primarily on the pseudo-diameter algorithm. Specifically, they perform some post-processing on the level sets generated at both ends of the pseudo-diameter to generate a third level set (possibly with multiple vertices in level set L_0) with a reduced max level set, and therefore resulting in a smaller bandwidth. There is an interesting survey article by Gibbs, Poole, and Stockmeyer [39] including many other envelope reducing ordering algorithms of the time.

The Gibbs-King [37] algorithm is a combination of the Gibbs-Poole-Stockmeyer level structure technique combined with numbering vertices on each level according to the King criterion. It is generally regarded to produce the the best quality ordering of the three algorithms, though it is the slowest [55].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.2.3.b Sloan Ordering

Sloan [76] originally introduced his envelope/wavefront reducing ordering algorithm for undirected and unweighted graphs. The idea of Sloan's algorithm is to number vertices from one endpoint of an approximate diameter in the graph, choosing the next vertex to number from among the neighbors of currently numbered vertices and their neighbors. A vertex of maximum priority is chosen from this eligible subset of vertices; the priority of a vertex has a "local" term that attempts to reduce the incremental increase in the wavefront, and a "global" term that reflects its distance from a second endpoint of the approximate diameter.

Duff, Reid, and Scott [26] have extended this algorithm to weighted graphs obtained from finite element meshes, and have used these orderings for frontal factorization methods. The weighted implementation is faster for finite element meshes when several vertices have common adjacency relationships. They have also described variants of the Sloan algorithm that work directly with the elements (rather than the nodes of the elements). The Sloan algorithm is a remarkable advance over previously available algorithms such as RCM [17], Gibbs-Poole-Stockmeyer [38, 55], and Gibbs-King [37, 51] algorithms since it computes smaller envelope and wavefront sizes.

3.2.3.c Spectral Ordering

Unlike the rest of the algorithms that are combinatorial in nature, the spectral algorithm is algebraic, and hence its good envelope-reduction properties are intriguing. Barnard, Pothen, and Simon [9] described this spectral algorithm that associates a Laplacian matrix with the given symmetric matrix, computes an eigenvector corresponding to the smallest positive Laplacian eigenvalue. This eigenvector is also called the *Fiedler vector* in recognition of the pioneering work of Miroslav Fiedler on the spectral properties of the Laplacian [28, 29]. The final permutation is obtained by sorting the components of the Fiedler vector in monotonically increasing or decreasing order.

The spectral algorithm has been examined for a wide range of applications. Juvan and Mohar [48, 49] have considered spectral methods for minimizing the p-sum problem (for $p \geq 1$), and Paulino et al. [66, 67] have applied spectral orderings to minimize envelope sizes. Additionally, spectral methods have been

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

applied successfully in areas such as graph partitioning [45, 68, 69], the seriation problem [6], and DNA sequencing [41].

George and Pothen [36] analyzed the algorithm theoretically, by considering a related problem called the 2-sum problem. They showed that minimizing the 2-sum over all permutations is equivalent to a quadratic assignment problem, in which the trace of a product of matrices is minimized over the set of permutation matrices. This problem is NP-complete; however, lower bounds for the 2-sum could be obtained by minimizing over the set of orthogonal and doubly stochastic matrices. (Permutation matrices satisfy the additional property that their elements are nonnegative; this property is relaxed to obtain a lower bound.) This technique gave tight lower bounds for the 2-sum for many finite-element problems, showing that the 2-sums from the spectral ordering were nearly optimal (within a few percent typically). They also showed that the permutation matrix closest to the orthogonal matrix attaining the lower bound is obtained (to first order) by permuting the second Laplacian eigenvector in monotonic order. This justifies the spectral algorithm for minimizing the 2-sum. These authors also showed that a family of graphs with small (n^{γ}) separators has small mean square wavefront (at most $O(n^{1+\gamma})$), where n is the number of vertices in the graph, and the exponent $\gamma \geq 1/2$ determines the separator size.

While we were working on the envelope/wavefront reduction problem by improving Sloan's algorithm, there was independent work by Boman and Hendrickson [13] on the same problem, but using multi-level heuristics. While their work was an improvement over the original Sloan algorithm, it was not as successful as our improved algorithm.

3.2.4 Contribution

For the most part, we follow Sloan [76], and Duff, Reid and Scott [26] in our work on the Sloan algorithm. Our new contributions are the following:

- We show that the use of a heap instead of an array to maintain the priorities of vertices leads to a lower time complexity, and an implementation that is about four times faster on our test problems. Sloan had implemented both versions, preferring the array over the heap for the smaller problems

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

he worked with, and had reported results only for the former. Duff, Reid, and Scott had followed Sloan in this choice.

- Our implementation of the Sloan algorithm for vertex-weighted graphs mimics what the algorithm would do on the corresponding unweighted graph, unlike the Duff, Reid, and Scott implementation. Hence we define the key parameters in the algorithm differently, and this results in no degradation in quality when ordering compressed graphs.
- We examine the weights of the two terms in the priority function to show that our test problems fall into two classes with different asymptotic behaviors of their envelope parameters; by choosing different weights for these two classes, we reduce the wavefront sizes obtained from the Sloan algorithm, on the average, to 60% of the original Sloan algorithm on a set of eighteen test problems.
- Together, the enhancements above enable the Sloan algorithm to compute small envelope and wavefront sizes fast—the time it needs is in general between two to five times that of the much simpler RCM algorithm.
- The analysis of the spectral algorithm suggests that while spectral orderings may also reduce related quantities such as the envelope size and the work in an envelope factorization, they might be improved further by post-processing with a combinatorial reordering algorithm. We introduce a variant of the Sloan algorithm as a post-processing step; creating a spectral/sloan or algebraic/combinatoric hybrid algorithm.

3.2.5 Fast Sloan Ordering

In this section we consider a weighted graph on a set of multi-vertices and edges, with integer weights on the multi-vertices. We think of the weighted graph as being derived from an unweighted graph, and the weight of a multi-vertex as the number of vertices of the unweighted graph that it represents. The weighted graphs in our applications are obtained from finite element meshes, where neighboring vertices with the same adjacency structures are "condensed" together to form multi-vertices. The weighted graph could potentially have fewer vertices and many fewer edges than the original unweighted graph in many

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

finite element problems. Duff, Reid, and Scott [26] call the weighted graph the supervariable connectivity graph. Ashcraft [3] refers to it as the compressed graph, and has used it to speed up the minimum-degree algorithm, and Wang [79] used it for an efficient nested dissection algorithm.

Sloan's algorithm [76] is a graph traversal algorithm that has two parts. The first part is the pseudo-diameter algorithm (see Section 3.1) that selects a start vertex s and an end vertex e. The second part then numbers the vertices, beginning from s, and chooses the next vertex to number from a set of eligible vertices by means of a priority function. Roughly, the priority of a vertex has a dynamic and static component: the dynamic component favors a vertex that increases the current wavefront the least, while the static part favors vertices at the greatest distance from the end vertex e. The computation-intensive part of the algorithm is maintaining the priorities of the eligible vertices correctly as vertices are numbered. As each vertex is numbered all unnumbered neighbors are updated.

3.2.5.a Eligible Vertices

At each step of the algorithm, vertices are in one of four mutually exclusive states. Any vertex that has already been numbered in the algorithm is a *numbered* vertex. *Active* vertices are unnumbered vertices that are adjacent to some numbered vertex. Vertices that are adjacent to active vertices but are neither active nor numbered are called *preactive* vertices. All other vertices are *inactive*. Initially all vertices are inactive, except for s, which is preactive. Figure 9 shows a small mesh being numbered by the sloan algorithm. Note the four different states.

At any step k, the sum of the sizes of the active verticesis exactly the size of the wavefront at that step for the reordered matrix, $\text{wf}_k(PAP^T)$, where P is the current permutation. Active and preactive vertices comprise the set of vertices *eligible* to be numbered in future steps.

An eligible vertex with the maximum priority is chosen to be numbered next. The priority function of a vertex i has two components: the increase in the wavefront size if the vertex was numbered next, $\text{incr}(i)$, and the distance from the end vertex, $\text{dist}(i, e)$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ALGORITHM 2 (The Sloan algorithm for a vertex-weighted graph.)

```
    [ Permutation result ] = SloanOrder( const Graph * G,
                                         const Graph::vertex s, const Graph::vertex e,
                                         const int w1, const int w2 )
    {
        // Initialization
1.      int n = G→size() ; // number of vertices
2.      int normfact = ⌊ dist(s,e) / max_{v∈G} deg(v) ⌋ ; // normalization factor
3.      int priority [ n ] ; // priority of each vertex in G
4.      statusType status[ n ] ; // status of each vertex in G
        // status[ i ] is one of 'inactive', 'preactive', 'active' or 'numbered'
5.      for i = 1 to n {
6.          status[ i ] = inactive ;
7.          priority[ i ] = -w1 * normfact * deg( i ) + w2 * dist( i, e ) ;
        }
8.      status[ s ] = preactive ;

        // Main Loop
9.      for k = 1 to n {
10.         i = // vertex such that 'priority[i]' is max of all active or preactive vertices
11.         for each j ∈ adj( i ) {
12.             if ( ( status[ i ] == preactive ) and
                    ( status[ j ] == inactive or preactive ) ) {
                    // 'j' becomes active, 'i' is numbered
13.                 priority[ j ] += ( G→vwgt(i) + G→vwgt(j) ) * normfact * w1 ;
14.                 status[ j ] = active ;
15.                 far_neighbors( G, j ) ;
16.             } else if ( ( status[ i ] == preactive ) and ( status[ j ] == active ) ) {
                    // 'i' moves from preactive to numbered
17.                 priority[ j ] += G→vwgt( i ) * normfact * w1;
18.             } else if ( ( status[ i ] == active ) and ( status[ j ] == preactive ) ) {
                    // 'j' moves from preactive to active
19.                 priority[ j ] += G→vwgt( j ) * normfact * w1 ;
20.                 status[ j ] = active ;
21.                 far_neighbors( G, j ) ;
                } // end if elseif...
            } // end for each...
22.         result.new2old[ k ] = i;
23.         result.old2new[ i ] = k;
24.         status[ i ] = numbered ;
        } // end for k = 1 to n ...
    } // end SloanOrder

    void far_neighbors( Graph* G, Graph::vertex j )
    {
        // 'j' is newly active, update any interested neighbors
25.     for each ℓ ∈ adj( j ) {
26.         if ( status[ ℓ ] == inactive ) {
                status[ ℓ ] = preactive ;
            }
27.         priority[ ℓ ] += G→vwgt( j ) * normfact * w1 ;
        } // end for each ...
    } // end far_neighbors
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

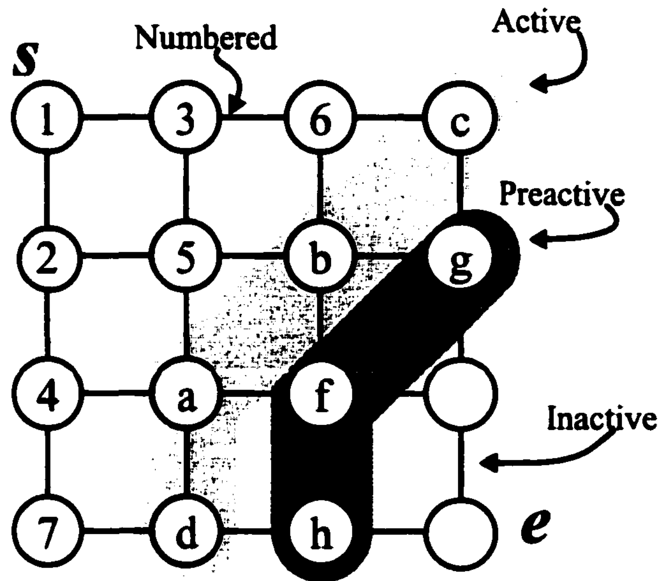


FIG. 9. *The Sloan algorithm in progress.*

3.2.5.b The Priority Function

Our implementation of the weighted Sloan algorithm on the weighted graph mimics what the original Sloan algorithm would do on an unweighted graph, and thus we define the degrees of the vertices and incr(i) differently from Duff, Reid, and Scott [26]

We denote by size(i) the integer weight of a multi-vertex i. The degree of the multi-vertex i, deg(i), is the sum of the sizes of its neighboring multi-vertices. Let the current degree of a vertex i, cdeg(i), denote the sum of the sizes of the neighbors of i among preactive or inactive vertices. It can be computed by subtracting from the degree of i the sum of the sizes of its neighbors that are numbered or active. When an eligible vertex is assigned the next available number, its preactive or inactive neighbors move into the wavefront. Thus

$$\text{incr}(i) = \begin{cases} \text{cdeg}(i) + \text{size}(i), & \text{if } i \text{ is preactive} \\ \text{cdeg}(i), & \text{if } i \text{ is active} \end{cases}.$$

The size(i) term for a preactive vertex i accounts for the inclusion of i into the wavefront. (Recall that the definition of the wavefront includes the diagonal element.) Initially, incr(i) is deg(i) + size(i) since nothing is in the wavefront yet.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The second component of the priority function, dist(i, e), measures the distance of a vertex i from the end vertex e. This component encourages the numbering of vertices that are very far from e even at the expense of a larger wavefront at the current step. This component is easily computed for all i by a breadth first search rooted at e. We show cdeg() incr() and dist() for all eligible vertices in Figure 9.

Denote by $P(i)$ the priority of an eligible vertex i during a step of the algorithm. The priority function used by Sloan, and Duff, Reid and Scott is a linear combination of two components

$$P(i) = -W_1 * \mathrm{incr}(i) + W_2 * \mathrm{dist}(i, e),$$

where W_1 and W_2 are positive integer weights. At each step, the algorithm numbers next an eligible vertex i that maximizes this priority function.

The value of incr(i) ranges from 0 to $(\Delta + 1)$ (where Δ is the maximum degree of the unweighted graph G), while dist(i, e) ranges from 0 to the diameter of the graph G. We felt it desirable for the two terms in the priority function to have the same range so that we could work with normalized weights W_1 and W_2. Hence we use the priority function

$$P(i) = -W_1 * \lfloor(\mathrm{dist}(s, e)/\Delta)\rfloor * \mathrm{incr}(i) + W_2 * \mathrm{dist}(i, e).$$

If the pseudo-diameter is less than the maximum degree, we set their ratio to one. We discuss the choice of the weights later in this section.

3.2.5.c The Algorithm

We present in Algorithm 2 our version of the weighted Sloan algorithm. This modified Sloan algorithm requires fewer accesses into the data structures representing the graph (or matrix) than the original Sloan algorithm [76]. The priority updating in the algorithm ensures that incr(j) is correctly maintained as vertices become active or preactive. When a vertex i is numbered, its neighbors and possibly their neighbors need to be examined. Vertex i must be active or preactive, since it is eligible to be numbered. We illustrate the updating of the priorities for only the first case in the algorithm, since the others can be obtained

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

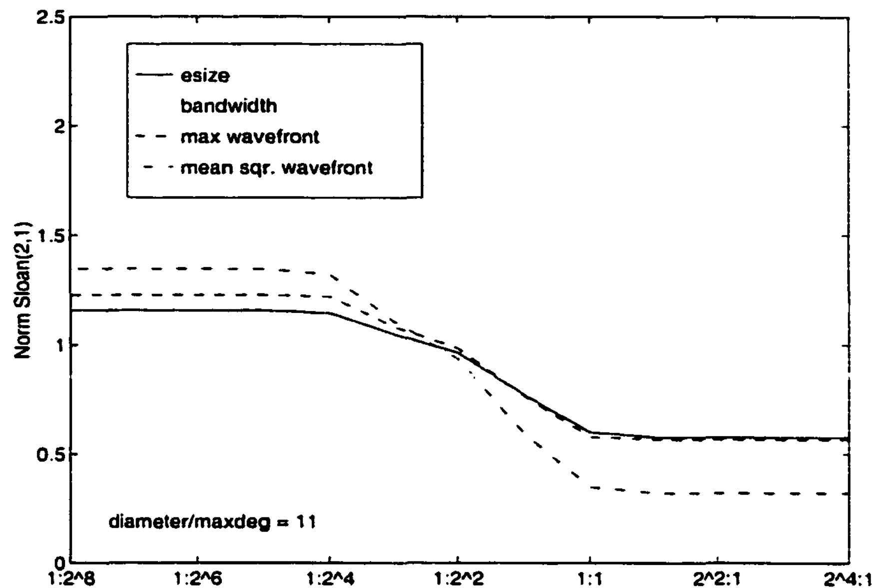


FIG. 10. *Envelope parameters of* `barth5` *as a function of the weights* W_1 *and* W_2.

similarly. Consider the case when i is preactive and j is inactive or preactive. The multi-vertex i moves from being preactive to numbered, and hence moves out of the wavefront, decreasing $\mathrm{incr}(j)$ by $\mathrm{size}(i)$, and thereby increases $P(j)$ by $W_1 * \lfloor(\mathrm{dist}(s,e)/\Delta)\rfloor * \mathrm{size}(i)$. Further, since j becomes active and is now included in the wavefront, it does not contribute in the future to $\mathrm{incr}(j)$, and hence $P(j)$ increases by $W_1 * \lfloor(\mathrm{dist}(s,e)/\Delta)\rfloor * \mathrm{size}(j)$.

3.2.5.d The Choice of Weights

Sloan [76], and Duff, Reid and Scott [26] recommend the unnormalized weights $W_1 = 2$, $W_2 = 1$. We studied the influence of the normalized weights W_1 and W_2 on the envelope parameters, and found, to our initial surprise, that the problems we tested fell into two classes.

The first class is exemplified by the `barth5` problem, whose envelope parameters are plotted for various choice of weights in Figure 10. The value of each envelope parameter is scaled with respect to the value obtained with the unnormalized weights $W_1 = 1$ and $W_2 = 2$ in the Sloan algorithm. Thus this and the next Figures reveal the improvements obtained by normalizing the weights in the Sloan algorithm.

The envelope parameters are plotted at successive points on the x-axis corresponding to changing the weight W_1 or W_2 by a factor of two. The ratio of the pseudo-diameter to maximum degree is 10 for

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

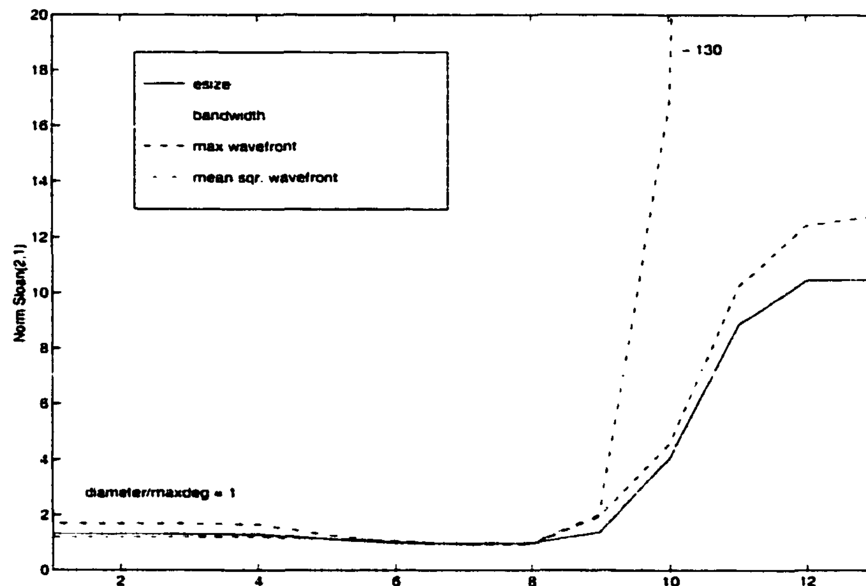


FIG. 11. *Envelope parameters of* `finance512` *as a function of the weights* W_1 *and* W_2.

this problem, and here large values of W_1 lead to the smallest envelope size and wavefront sizes. The normalized weights $W_1 = 2$ and $W_2 = 1$ suffice to obtain these values; note the asymptotic behavior of the envelope parameters. The bandwidth has a contrarian behavior to the rest of the parameters, and thus high values of W_2 lead to small bandwidths for these problems.

The second class is exemplified by the `finance512` problem, whose envelope parameters are plotted for various choice of weights in Figure 11. Again, the value of each parameter is scaled by the value obtained by the Sloan algorithm with unnormalized weights $W_1 = 2$, $W_2 = 1$. The ratio of the pseudo-diameter to maximum degree is 1. Here high values of W_2 lead to small envelope parameters. Note that the bandwidth follows the same trend as the rest of the envelope parameters, unlike the first class.

A user needs to experiment with the weights to obtain a near-optimal value of an envelope parameter for a new problem, since one does not know a priori which of the two classes it belongs to. Fortunately, small integer weights suffice to get good results in our experiments, and hence a set of good weights can be selected automatically by computing the envelope parameters with a few different weights.

The results tabulated in Section 5.1 show that it is possible to reduce the mean square wavefront by choosing one normalized set of weights for each problem in Class 1, and another for each problem in Class 2, rather than the unnormalized weights ($W_1 = 2$, $W_2 = 1$) used by Sloan, and Duff, Reid and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Scott. The weights we have used are $W_1 = 8, W_2 = 1$ for Class 1 problems, and $W_1 = 1, W_2 = 2$ for problems in Class 2. An automatic procedure could compute the envelope parameters for a few sets of weights, and then choose the ordering with the smaller values.

There are two limiting cases of the Sloan algorithm.

When $W_1 = 0, W_2 \neq 0$, then the distance from the end vertex e determines the ordering, and the Sloan algorithm behaves similarly to the Gibbs-Poole-Stockmeyer algorithm [38]. The primary difference between the two is that Sloan's algorithm does not use the post-processing to improve the width of the level sets. Preactive nodes will never be numbered directly in this case. This is different from the case when W_1 is nonzero and W_2 is much larger than W_1. Here, the first term still plays a role in reducing the envelope parameters. This case is similar in character to the Gibbs-King algorithm, excepting again for the differences in level sets. The values of envelope parameters obtained when the ratio W_2/W_1 is 2^{16} are significantly smaller than the values obtained when $W_1 = 0$ and $W_2 \neq 0$. Only neighbors and second-order neighbors of the numbered vertices are eligible to numbered at any step, and among these vertices the first term serves to reduce the local increase in the wavefront when W_1 is nonzero.

The second limiting case, when $W_2 = 0, W_1 \neq 0$, corresponds to a greedy algorithm in which vertices are always numbered to reduce the local increase in wavefront. This greedy algorithm does particularly poorly on Class 2 problems. This case is conceptually similar to King's algorithm for reducing wavefront [51].

The two classes of problems differ in the importance of the first, "local", term that controls the incremental increase in the wavefront relative to the second, "global", term that emphasizes the numbering of vertices far from the end-vertex. When the first term is more important in determining the envelope parameters, the problem belongs to Class 1, and when the second term is more important, it belongs to Class 2. We have observed that the first class of problems represent simpler meshes: e.g., discretization of the space surrounding a body, such as an airfoil in the case of `barth5`. The problems in the second class arise from finite element meshes of complex three-dimensional geometrical objects, such as automobile frames. The `finance512` problem is a linear program consisting of several subgraphs joined together

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

by a binary tree interconnection. In these problems, it is important to explore several "directions" in the graph simultaneously to obtain small envelope parameters.

The bandwidth is smaller when larger weights are given to the second term, for both classes of problems. This is to be expected, since to reduce the bandwidth, we need to decrease, over all edges, the maximum deviation between the numbers of the endpoints of an edge.

The first term in the priority function emphasizes vertices with fewer neighbors—a greedy strategy to reduce total wavefront by picking the minimum available at each step. The second term emphasizes the need to number vertices farther away from the end vertex, injecting some measure of global information into the method. Sometimes taking a hit early on (by numbering a vertex with many neighbors, hence increasing the wavefront size significantly) may benefit in the long run.

Refering again to Figure 9, if both weights W_1 and W_2 are set to one, then the two highest priority vertices at this step are "a" and "c". Whichever is numbered, the other vertex and "b" will have the highest priority at the beginning of the next step.

3.2.5.e Effect of Preactive Nodes

The execution time of the Sloan algorithm is dominated by the size of the priority queue it must maintain for all active and preactive vertices. At first glance, it would seem that the distinction between preactive and inactive serves only to properly handle the start vertex and initialize the main loop.

While it is true that for many cases a preactive vertex is never numbered — with the mandatory exception of the start vertex, it is also true that for many problems preactive nodes are chosen frequently enough to make their exclusion detrimental to the algorithm. We use a simple example to illustrate why this is so.

Consider a star-shaped graph with n vertices, where $n - 1$ vertices (called *points*) have but a single edge to the nth vertex (called the *hub*). In this case, the length of the diameter is two, with any two points sufficing as the start and end vertices.

The Sloan algorithm will naturally number the start vertex, and the wavefront will include the start vertex and the hub. The algorithm then numbers all the other points, and each one having a wavefront

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of that point and the hub. When all has been numbered except the hub and the end vertex, they can be numbered in any order without affecting the wavefront.

Consider now a modified Sloan algorithm where nodes are not considered for numbering until they are active. This hypothetical algorithm would number the start vertex, with the start vertex and the hub in the wavefront. Then, since the only active vertex is the hub, this hypothetical algorithm would be forced to number it next. The wavefront at this stage would be $n - 1$, from the newly numbered hub and the remaining $n - 2$ points in the star. For each step thereafter, the size of the wavefront would reduce by one.

Given this illustration, it is clear to see the function that preactive nodes can play. It allows the algorithm to avoid vertices of high degree until a large enough number of its low degree neighbors have been numbered. In the case of a $n - 1$ point star, omiting preactive nodes would increase the maximum wavefront from two to $n - 1$.

3.2.5.f The Accelerated Implementation

In the Sloan algorithm, the vertices eligible for numbering are kept in a priority queue. Sloan [76] implemented the priority queue both as an unordered list in an array and as a binary heap, and found that the array implementation was faster for his test problems (all with less than 3,000 vertices). Hence he reported results from the array implementation only. Duff, Reid, and Scott [26] have followed Sloan in using the array implementation for the priority queue in the Harwell library routine MC40.

This is an unfortunate accident because the heap implementation is provably better. In [52] we proved that the heap implementation runs in $O(n \log n)$ time, while the array implementation requires $O(n^{1.5})$ time for two-dimensional problems, and $O(n^{5/3})$ time for three-dimensional problems. In practice, we found even the smallest problems ran faster with a heap.

This difference in running time requirements is experimentally observed as well. In Figure 12 we compare the times taken by the array and heap implementations of the Sloan algorithm relative to our implementation of the RCM algorithm. The RCM algorithm uses a fast pseudo-diameter algorithm described by Duff, Reid, and Scott [26]. In a response to our research, Reid and Scott [72] abandoned MC40

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

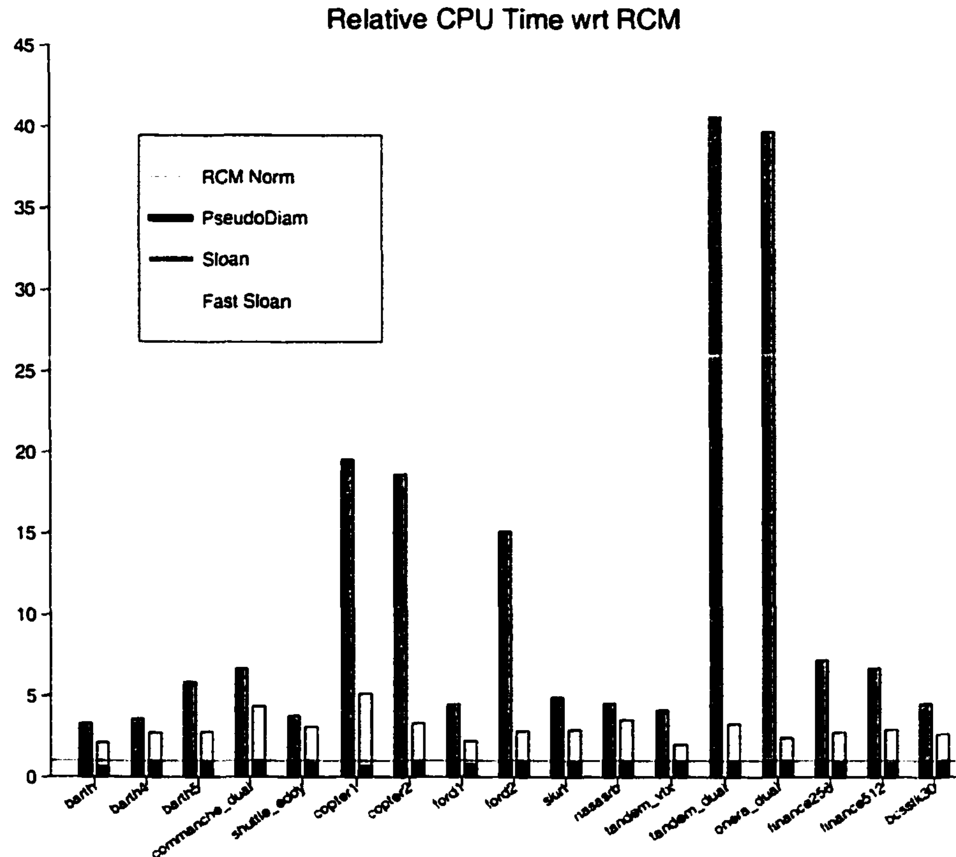


FIG. 12. *Relative timing performance of RCM, ArraySloan, and HeapSloan algorithms.*

and implemented MC60 and MC61 according to most of our research. Nevertheless, they assert that the array implementation is faster for small problems, (less that 5000 vertices).

For the eighteen matrices in Table 6, the mean time of the ArraySloan was 11.3 times that of RCM, while the median time was 8.2 that of RCM. However, the mean cost of the HeapSloan was only 2.5 times of RCM, with the median cost only 2.3. The greatest improvements are seen for the problems with greater numbers of vertices or with higher average degrees.

We have also computed the times taken by MC40B to order these problems, and found them to be comparable to the times reported here for the ArraySloan implementation, inspite of the different programming languages used (Fortran for MC40B and C for ours.)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

We emphasize that this change in the data structure for the priority queue has no significant influence on the quality of the envelope parameters computed by the algorithm. Minor differences might be seen due to different tie-breaking strategies.

3.2.6 Complexity Analysis

In this Section we analyze the computational complexity of the Sloan algorithm using both heap and array implementations. The analysis has the interesting feature that the time complexity depends on the maximum wavefront size, a quantity related to the mean square wavefront that the algorithm is seeking to reduce. Nevertheless, it is possible to get *a priori* complexity bounds for problems with good separators. The results clearly show the overwhelming superiority of the heap implementation; an analysis of the complexity of the Sloan algorithm is not available in earlier published work.

The major computational difference lies in the implementation of the priority queue (see Section 3.2.5.f). We call these two implementations *ArraySloan* and *HeapSloan* according to the data structure used to implement the priority queue.

For the array implementation, the queue operations `insert()`, `increment_priority()`, and `delete()` are all $O(1)$ operations, but the `max_priority()` operation (finding the vertex with the maximum priority) is $O(m)$, where m is the size of the queue. All operations on the binary heap are $O(\log m)$ except `max_priority()`, which is $O(1)$.

To continue with our analysis, we examine Algorithm 2 on page 42. It is immediately clear that the function `far_neighbors()` (lines 25—27) is $O(\deg(j))$ for ArraySloan. We can easily bound this by $\Delta = \max_{1 \le i \le n}(\deg(i))$. For HeapSloan, `far_neighbors()` for HeapSloan is $O(\Delta * \log m)$, where m is the maximum size of the priority queue. This is because the priority updates require reheaping.

The Sloan function (lines 1–24) has three loops: the initialization loop (lines 5–7), the outer ordering loop (lines 9–24), and the inner ordering loop (lines 11–21). The initialization loop is the same for either implementation, and is easily seen to require $O(|E|)$ time.

Consider now the ArraySloan implementation. For each step of the outermost loop starting at line 9, it must find and remove the vertex of maximum priority, requiring $O(m)$ time. The inner loop is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

executed at most Δ times. The worst case for the inner loop is when the priority is incremented and the `far_neighbors` routine is called, and this requires $O(\Delta)$ time. Thus the worst case running time for the ordering loop is $O(|V| * (m + \Delta^2))$. For the entire algorithm it is $O(|V| * (m + \Delta^2) + |E|)$.

For the HeapSloan implementation, at each step of the outermost loop starting at line 9, the algorithm must delete the vertex of maximum priority, and then rebuild the heap; this takes $O(\log m)$ time. The inner loop is executed at most Δ times. The worst case for the inner loop is when the priority is incremented and the `far_neighbors` function is called. This time is $O(\Delta * \log m)$. The worst case time complexity for the ordering loop of HeapSloan is thus $O(|V| * \Delta^2 * \log m)$. For the entire algorithm it is $O(|V| * \Delta^2 * \log m + |E|)$.

These bounds can be simplified further. The maximum size of the queue can be bounded by the smaller of (1) the product of the maximum wavefront of the reordered graph and the maximum degree, and (2) the number of vertices n. Then the complexity of ArraySloan is $\mathcal{O}(|V| * \Delta * \mathrm{maxwf})$, while the complexity of HeapSloan is $\mathcal{O}(|V| * \Delta^2 * \log(\mathrm{maxwf} * \Delta))$. If we consider degree-bounded graphs, as finite element or finite difference meshes tend to be, then the ArraySloan implementation has time complexity $\mathcal{O}(|V| * \mathrm{maxwf} + |E|)$, while the HeapSloan implementation has $\mathcal{O}(|V| * \log(\mathrm{maxwf}) + |E|)$.

These bounds have the unsatisfactory property that they depend on the maximum wavefront, a quantity that the algorithm seeks to compute and to reduce. To eliminate this dependence, we will restrict ourselves to important classes of finite element meshes with good separators.

The class of d-dimensional overlap graphs (where $d \geq 2$) whose degrees are bounded includes finite element graphs with bounded aspect ratios embedded in d dimensions and all planar graphs [62]. Overlap graphs have $O(n^{(d-1)/d})$ separators that split the graph into two parts with the ratio of their sizes at most $(d+1)/(d+2)$. Hence the maximum wavefront can be bounded by $O(n^{(d-1)/d})$ for a modified nested dissection ordering that orders one part first, then the separator, and finally the second part.

It is interesting to note that the wavefront at any step also forms a separator. If we assume that the maximum wavefront attained by the Sloan ordering is bounded by the size of the separator, then we can conclude that the HeapSloan implementation requires $O(n \log n)$ time while the ArraySloan implementation requires $O(n^{(2d-1)/d})$ time for a d-dimensional overlap graph. For a planar mesh ($d = 2$), the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ArraySloan implementation requires $O(n^{3/2})$-time, while for a three dimensional mesh with bounded aspect ratios ($d = 3$), its time complexity is $O(n^{5/3})$.

3.2.7 Sloan Refinement

The hybrid algorithm consists of two steps: first compute the spectral ordering; then use a modification of the Sloan algorithm to refine the ordering locally.

To change the Sloan algorithm from one that computes an ordering from scratch to one that refines a given ordering, we need to modify the selection of start and end nodes, and the priority function. We use *input ordering* in this section to describe the ordering of the matrix immediately before the Sloan refinement is performed. In our implementation, this input ordering is the spectral ordering, though the refining algorithm can work with any input ordering.

The Sloan algorithm requires a start node to begin numbering from, and an end node to compute the priority function. We choose the start node s to be the first node and the end node e to be the last node in the input ordering. Hence the burden of finding a good set of endpoints is placed on the spectral method. Experience suggests that this is where it should be. The spectral method seems to have a broader and more consistent view than the local diameter heuristic. This feature alone yields improved envelope parameters over the Sloan algorithm for most of our test problems.

The priority function is

$$P(i) = -W_1 * \lfloor (n/\Delta) \rfloor * \text{incr}(i) + W_2 * \text{dist}(i, e) - W_3 * i.$$

The first two terms are similar to the priority function of the Sloan algorithm, except that the normalization factor has n, the number of vertices in the numerator, rather than the pseudo-diameter. The latter is not computed in this context, and this choice makes the first and third term range from 1 to n.

This function is sensitive to the initial ordering through the addition of a third weight, W_3. For $W_3 > 0$, higher priority is given to lower numbered vertices in the input ordering. Conversely, for $W_3 < 0$, priority is given to higher numbered vertices. This effectively performs the refinement on the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

reverse input ordering, provided s and e are also reversed. There is some redundancy between weighting the distance from the end in terms of the number of hops ($\text{dist}(i, e)$) *and* the distance from the end in terms of the input ordering (i).

Selection of the nodes s and e and the new priority function are the only algorithmic modifications made to the Sloan algorithm. The node selection, node promotion, and priority updating scheme (see Fig. 2), are unchanged.

The normalization factor in the first term of the priority function makes the initial influence of the first and third terms roughly equal in magnitude when W_1 and W_3 are both equal to 1. The weight W_2 is usually set to one. This makes it a very weak parameter in the whole algorithm, but small improvements result when its influence is nonzero. If the component of the Fiedler vector with the largest absolute value has the negative sign, we set $W_3 = -1$ and swap s and e. Otherwise, we set $W_3 = 1$ and use the nondecreasing ordering of the Fiedler vector.

For Class 1 problems, higher values of W_1 can lead to improvements in the envelope parameters over the choice of $W_1 = 1$, even though it is slight in most cases. For Class 2 problems, use of $W_1 = 1$, $W_2 = W_3 = 2$ can lead to improvements as well.

3.2.8 Applications

This section discusses preliminary evidence demonstrating the applicability of the orderings we generated. In Section 3.2.8.a we describe how a reduction in mean square wavefront directly translates into a greater reduction in cpu-time in a frontal factorization. We also discuss the impact of these orderings on incomplete Cholesky (IC) preconditioned iterative solvers in Section 3.2.8.b. Finally in Section 3.2.8.c we list other areas where the envelope/wavefront reduction either has been applied or shows promise of being applied successfully.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE 1
Results of two problems on a CRAY-J90 using MA42. Times reported are in seconds.

		Sun SPARC20 Ordering Time	Cray-J90 Frontal Solve Time	Flops
bcsstk30	Initial	0	1106	8.7e+10
	RCM	3.7	1649	1.4e+11
	Sloan	6.1	989	7.5e+10
	Spectral	11.9	188	1.1e+10
	Hybrid	14.6	205	1.1e+10
skirt	Initial	0	2427	2.1e+11
	RCM	5.0	2233	1.9e+11
	Sloan	8.4	1754	1.4e+11
	Spectral	18.6	979	7.6e+10
	Hybrid	22.6	980	7.3e+10

3.2.8.a Frontal Methods

The work in a Cholesky factorization algorithm is

$$\text{work}(A) = \frac{1}{2}\sum_{i=1}^{n} |\text{wf}_i(A)| \left(|\text{wf}_i(A)| + 3 \right).$$

Hence a reduction in the mean-square wavefront leads to fewer flops during Cholesky factorization. Duff, Reid, and Scott [26] have reported that Sloan orderings lead to faster frontal factorization times than RCM orderings. Barnard, Pothen and Simon [9] have reported similar results when spectral orderings are used.

Two problems were run by Dr. Jennifer Scott on a single processor of a Cray-J90 using the Harwell frontal factorization code MA42. The matrix values were generated randomly. (The orderings used were obtained earlier than the results reported in Appendix A; however, these results suffice to show the general trends.) The results in Table 1 show a general correlation between mean square wavefronts (proportional to flops) and factorization times. The spectral ordering enables the factorization to be computed about 5.2 times faster than the Sloan ordering for the `bcsstk30` problem; this ratio is 1.8 for the `skirt` problem. The hybrid does not improve factorization times over the spectral ordering for these problems.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

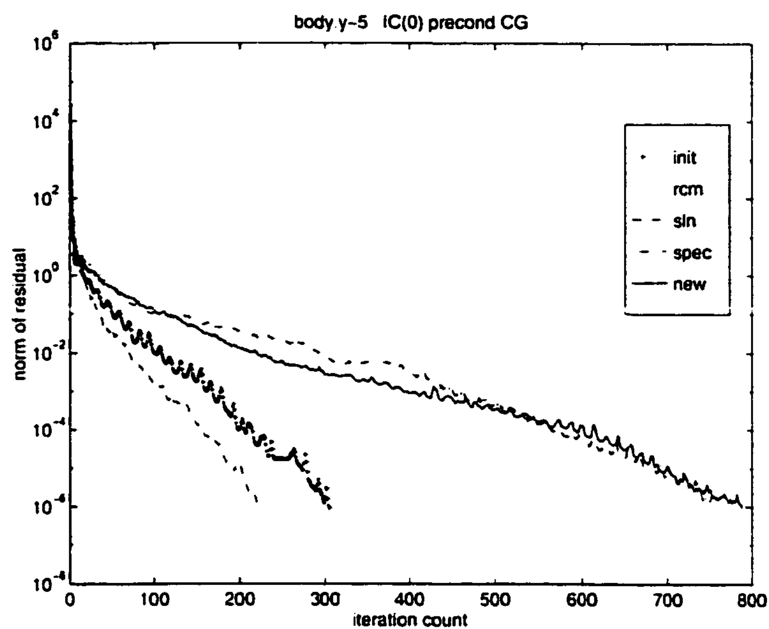


FIG. 13. *Convergence of* `body.y-5` *for various orderings using IC(0) preconditioned CG.*

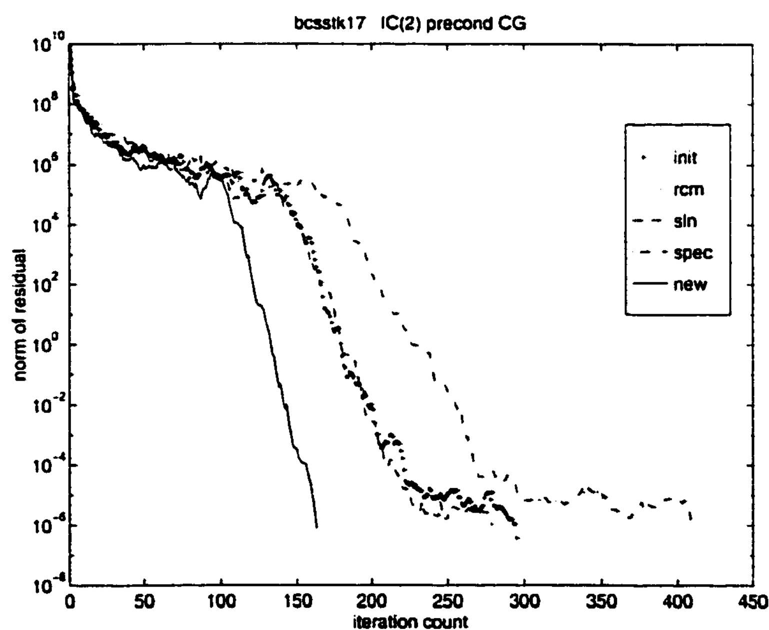


FIG. 14. *Convergence of* `bcsstk17` *for various orderings using IC(2) preconditioned CG.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE 2
Convergence of preconditioned CG on body.y-5 *and* bcsstk17.

		Ordering			
		RCM	Sloan	Spectral	Hybrid
body.y-5	$\|\|R\|\|_F$	3,608	2,598	9,166	7,276
$\|V\| = 18,589$	nnz(L)	73,721	73,721	73,721	73,721
$\|E\| = 55,132$	iteration count	756	497	1,203	1,009
Level 0	cpu time	1,103	726	1,715	1,405
	flops	6.8e+08	4.5e+08	1.1e+09	9.1e+08
Level 2	$\|\|R\|\|_F$	1,430	885	988	501
	nnz(L)	128,854	126,141	128,121	126,319
	iteration count	457	231	356	265
	cpu time	726	376	564	422
	flops	5.1e+08	2.6e+08	4.0e+08	2.9e+08
bcsstk17	$\|\|R\|\|_F$	6.5e+08	6.5e+08	7.3e+08	1.9e+09
$\|V\| = 10,974$	nnz(L)	470,304	473,017	486,524	474,935
$\|E\| = 208,838$	iteration count	422	323	320	179
Level 2	cpu time	1131	894	871	503
	flops	1.1e+09	9.5e+08	9.5e+08	5.2e+08

3.2.8.b Incomplete Cholesky Preconditioning

In this section we report preliminary experiments on the influence of our orderings on preconditioned conjugate gradients (CG). We precondition CG with an Incomplete Cholesky factorization (IC(k)) that controls k, the level of the fill introduced.

Since the envelope is small, we confine fill to a limited number of positions, and hope to capture more of the character of the problem with fewer levels of fill. However, a tighter envelope is only one of the factors that affect convergence. For instance, orderings must respect numerical anisotropy for fast convergence.

Our preliminary results have been mixed. In Table 2 we show information pertaining to two problems that are representative of our data. It is worth noting how strongly the norm of the remainder matrix for a given ordering is a predictor of iteration counts. The body.y-5 problem shows that the Sloan ordering can be very effective in reducing the iteration count. This problem is a 2-dimensional mesh with an aspect ratio of 10^{-5}. In the case of poor aspect ratios, a weighted Laplacian should be more appropriate for computing the spectral ordering, but we defer this topic for future research. Duff and Meurant [24]

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

indicate that ordering becomes more significant when the problem becomes more difficult (discontinuous coefficients, anisotropy, etc.).

Another problem from the Harwell-Boeing collection `bcsstk17` did not converge quickly for levels of fill below two, indicating that it is a difficult problem. The rate of convergence at two levels of fill shows that the new ordering reduces the iteration count by almost half that of its closest competitor. Since envelope reduction concentrates fill, it is possible that the benefits of the hybrid ordering are maximized when more than one level of fill is allowed.

3.2.8.c Other Promising Applications

The envelope/wavefront reducing ordering problem also has applications that extend beyond the immediate domain of sparse matrix computations.

Work has been done in mapping genomics problems to the envelope/wavefront reduction problem [6]. In this problem, long strands of DNA are randomly cut to a manageable size and sequenced. Then the collection sequences must be reassembled by finding the greatest amount of overlap in the known sequences. Spectral methods are particularly useful in this context because they can tolerate some amount of error in the system.

Sloan's algorithm has also shown some promise in optimizing sparse kernels for cache performance [80, 70]. The wavefront reducing ordering produced by the Sloan algorithm can increases temporal locality of the data in some sparse matrix computations.

Along a similar vein, these orderings may be useful in spatial databases, particularly for ordering the large amounts of data on disk to reduce paging when performing a query. Currently proposed algorithms, far from ideal [75], are space filling curves (primarily row order, Peano-Hilbert, and Morton/Z-order). Space filling curves have the property that they enumerate points in n-dimensional space such that if any two points are close to each other in the enumeration, then they lie close to each other in space. The converse, which is what is needed in this context, is decidedly not true. Heuristics for the envelope/wavefront reduction problem have this distance preserving metric when ordering the vertices in a graph — essentially mapping from a higher dimensional space into a one dimensional space.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.3 Fill Reduction

Sections 3.3.1–3.3.2 and 3.3.4 describe how fill is created and provide graph models for visualizing and implementing the heuristics. Each of these three sections uses Figure 15 to reinforce the concepts presented. Section 3.3.3 reviews some known optimality results for the fill reduction problem. Section 3.3.5 provides a brief introduction to the minimum degree algorithm and its derivations. We discuss our complexity analysis of MD and MMD in Section 3.3.6, and show what this means in terms of runtime for some model problems in Section 3.3.7.

3.3.1 Definition of Fill

Direct methods rely on factoring the symmetric matrix A into the product LDL^T, where L is a lower triangular matrix, and D is a diagonal matrix. The factor L is computed using Cholesky factorization — a symmetric variant of Gaussian elimination. The factor L has nonzeros in all the same positions[2] as the lower triangle of the original matrix A, plus additional nonzeros in positions that were zero in A, but induced by the factorization. These additional nonzeros are *fill* elements. The presence of fill increases both the storage required to hold the factor and the computational work required to generate it. The amount of fill created depends entirely on the nonzero structure of the matrix, and the order in which the rows/columns are factored.

An example is provided in the first column of Figure 15, showing non-zeros in original positions of A as "×" and fill elements as "•". This example incurs two fill elements. The order in which the factorization takes place, greatly influences the amount of fill. The matrix 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. 15, the elimination order $\{2, 6, 1, 3, 4, 5\}$ produces only one fill element. This is the minimum amount of fill for this example.

[2] No "accidental" cancellations will occur during factorization if the numerical values in A are algebraic indeterminates.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

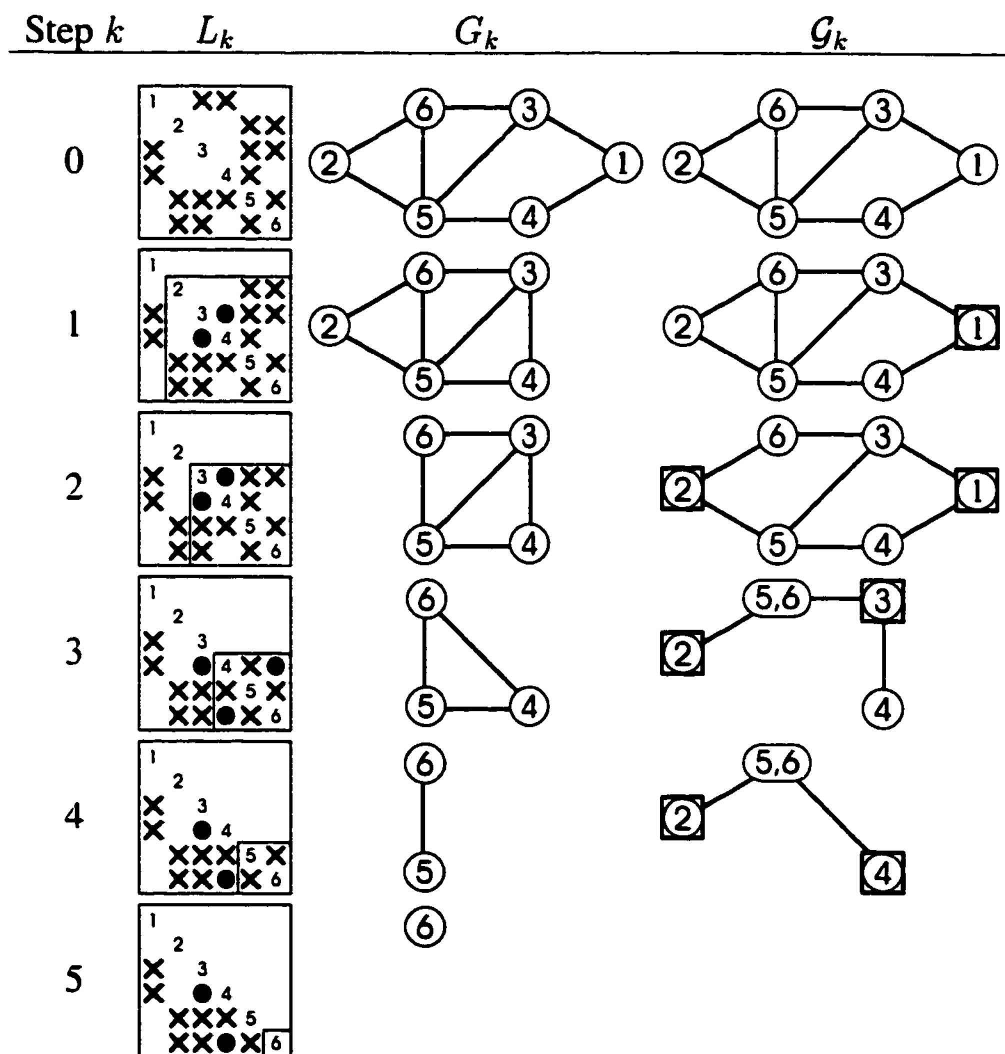


FIG. 15. *Examples of factorization and fill. For each step, k, in the factorization, there is the nonzero structure of the factor, L_k, the associated elimination graph, G_k, and the quotient graph $\mathcal{G}_k$. The elimination graph consists of vertices and edges. The quotient graph has edges and two kinds of vertices, supernodes (represented by ovals) and enodes (represented by boxed ovals).*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.3.2 The Elimination Graph

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

The graph model of symmetric Gaussian elimination was first introduced by Parter [65]. 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, $G_0 = G(A)$. At each step k, let v_k be the vertex corresponding to the kth column of 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. 15 illustrates the graph model of elimination.

Although the elimination graph is an important conceptual tool, it is not used in modern implementations because the amount of storage required to represent it can grow during elimination. When a vertex v_k is removed with degree d there are potentially $d(d-1)/2$ edges to add. Therefore the storage required for an elimination graph can grow like the size of the factor, and cannot be predetermined. In practice a *quotient graph* is employed to implicitly represent an elimination graph in the same storage as $G(A)$.

3.3.3 Known Optimality Results

The Fill Reduction problem is a known NP-complete problem [81] that is of fundamental importance in scientific computing. As such, this problem has garnered a great deal of attention and a large number of ordering heuristics. Performance guarantees, however, are harder to find.

For ordering algorithms based on Minimum Degree, there is a negative result by Berman and Schnitger [10] where for a torus T_n with $n = k^2$ vertices where $k/4$ is a power of 3, there exists a Multiple Minimum Degree (MMD) ordering that produces a factor with $n^{\log_3 4}$ fill and $n^{1.5 \log_3 4}$ arithmetic. The lower bounds for a T_n torus with $n = k^2$ vertices is $\Theta(n \lg n)$ fill and $\Theta(n^{1.5})$ arithmetic. Determining the worst possible performance for MMD on a torus remains an open problem.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

For Nested Dissection (ND) algorithms, Agrawal, Klein and Ravi [1] have proven that their polynomial nested dissection algorithm guarantees a factor of size $\mathcal{O}(\min(\sqrt{d}\log^4 n, m^{\frac{1}{4}}\log^{3.5} n))$ and requiring $\mathcal{O}(d\log^6 n)$ operations to calculate, where d is the maximum number of non-zero elements in any row or column of the $n \times n$ coefficient matrix. They also show that this result is within a factor of $\mathcal{O}(\sqrt{d}\log^4 n)$ of the optimum for fill and a factor of $\mathcal{O}(d\log^6 n)$ for operation count. By using a minimum node (non-polynomial) separator algorithm, they prove that there exists a nested dissection ordering with the tighter bounds of $\mathcal{O}(\sqrt{d}\log^2 n)$ size factor requiring $\mathcal{O}(d\log^4 n)$ operations to calculate, where d is the maximum number of non-zero elements in any row or column of the general, symmetric $n \times n$ coefficient matrix.

Tighter bounds have been found for restricted classes of graphs such as trees [47], planar graphs [57], and overlap graphs [62]. All of these arguments for ND orderings hinge on guarantees of the partition imbalance and separator size.

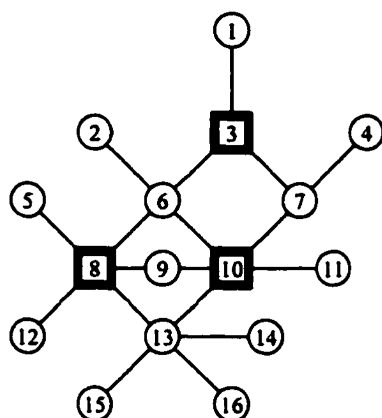
Blair, Heggernes, and Telle [11] consider the problem of taking an arbitrary filled graph G^+ of an original graph G and obtaining a graph M that is both a minimal filled graph of G and a subgraph of G^+. They report an $\mathcal{O}(f(e+f))$ algorithm that solves this problem and computes the corresponding elimination ordering, where e is the number of original edges ($e = |E(G)|$) and f is the number of fill edges ($|E(G^+)| - |E(G)|$).

3.3.4 The Quotient Graph

A quotient graph is an implicit representation of an elimination graph. It is designed specifically for greedy fill-reducing orderings such as MMD [32, 33]. Eliminating nodes does not require explicit addition of fill edges to the quotient graph, but it does impose an extra level of indirection to compute certain quantities such as vertex degree. Thus, the quotient graph is immune to the effects of fill. This is vital since the amount of fill is known only after the ordering step. The quotient graph has two distinct kinds of vertices: *supernodes* and *enodes*[3]. A supernode represents a set of one or more uneliminated columns of A. Similarly, an enode represents a set of one or more eliminated columns of A. The initial graph, $\mathcal{G}_0$,

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

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



node	reachable set	size of reachable set
1	6 7	2
2	6	1
3	N/A	
4	7	1
5	6 9 13 12	4
6	1 2 5 7 9 11 12 13	8
7	1 6 9 11 13	5
8	N/A	
9	5 6 7 11 12 13	6
10	N/A	
11	6 7 9 13	4
12	5 6 9 12	4
13	5 6 7 9 12	5
14	13	1
15	13	1
16	13	1

FIG. 16. *Example of a quotient graph. The nodes are represented as circles, and enodes as boxes. The* reachable set *of a node is the union of the set of adjacent nodes and the set of all nodes adjacent to adjacent enodes.*

consists entirely of supernodes and no enodes; further, each supernode contains one column. Edges are constructed in the same manner as in the elimination graph. The initial quotient graph, $\mathcal{G}_0$, is identical to the initial elimination graph, G_0, which can be seen in Figure 15.

When a supernode is eliminated at some step, it is not removed from the quotient graph; instead, the supernode becomes an enode. Enodes are important because they indirectly represent the fill edges in the elimination graph[4]. To demonstrate how, we first 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. The number of enodes in the path can be zero. We will say that a supernode j is *reachable* from a supernode i in $\mathcal{G}_k$ if there exists a *reachable path* from i to j. Similarly i is reachable from j since the path is undirected. Since the number of enodes in the path can be zero, adjacency in $\mathcal{G}_k$ implies reachability in $\mathcal{G}_k$. Therefore if two supernodes i, j are reachable in the quotient graph $\mathcal{G}_k$, then the corresponding vertices i, j in the elimination graph G_k are adjacent in G_k.

The *reachable set* of a vertex is simply all vertices that are reachable from a vertex, not including itself. Figure 16 shows a sample quotient graph with the reachable sets and their sizes for each node.

[4]Where d is the degree of the vertex in the elimination graph.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

For the purposes of this discussion we leave the reachable set of an enode undefined. Conceptually the reachable set of a supernode s in the quotient graph is identical to the adjacency set of s in the elimination graph. Thus the quotient graph is an implicit representation of an elimination graph.

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 at the beginning of the algorithm, before any supernodes are eliminated using graph compression [3]. 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 also be removed in certain instances during factorization. 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 shared edge can be safely removed as it is redundant. This not only improves storage and speed, but allows tighter approximations to supernode degree as well.

Going once more to Fig. 15, 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 actually removed an edge between supernodes 5 and 6. This was done because the edge was redundant; supernode 5 is reachable

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

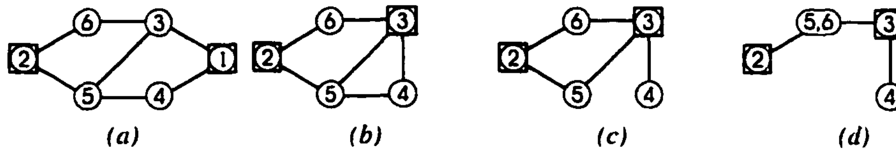


FIG. 17. *Quotient Graph Transformation in Detail. In this figure we show the transformation of a quotient graph between steps 2 and 3 of Figure 15. We start with an updated quotient graph (a). Next, we eliminate supernode 3, which causes it to form an enode that absorbs all adjacent enodes (b). Now the edge between supernodes 4 and 5 is redundant since they are reachable through the new enode 3, so the edge is removed (c). Finally, supernodes 5 and 6 are indistinguishable and merged into a single supernode.*

from 6 through enode 2. The transformation after eliminating supernode 3 involves several steps which are shown in Figure 17. First, supernode 3 becomes an enode and absorbs enode 1 (including its edge to supernode 4). Now enode 3 is adjacent to supernodes 4, 5 and 6. Next, the 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 $\{5,6\}$. The rest of the transformations are not as complicated. When supernode 4 is eliminated, it becomes an enode and absorbs enode 3. Finally supernode $\{5,6\}$ is eliminated. The relative order between columns 5 and 6 at this point (and between any columns within a supernode) has no effect on fill.

3.3.5 Greedy Fill-Reducing Heuristics

The simplest heuristic for computing a greedy fill-reducing ordering is to repeatedly select and remove a supernode from the quotient graph having minimum degree. One could implement a greedy heuristic that computed the minimum fill induced by eliminating each supernode, but it is too expensive in practice [73]. The degree of a supernode puts an upper bound on the amount of fill that could be induced by its elimination.

Since updating the quotient graph and recomputing vertex degrees is so expensive, a common optimization is to use Minimum Degree with a lazy update. This is possible because when a supernode is eliminated, only its neighbors have been changed. Therefore, if one can find a large, independent set of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ALGORITHM 3 (The Multiple Minimum Deg .e algorithm defined in terms of a Quotient Graph.)

1. $k \leftarrow 0$
2. **while** ($k < n$) {
3.   Let m be the minimum known degree, $\deg(x)$, of all $x \in \mathcal{G}_k$.
4.   **while** m is still the minimum known degree of all $x \in \mathcal{G}_k$ {
5.   Choose supernode x_k such that $\deg(x_k) = m$
6.   **for all** of the p columns represented by supernode x_k {
7.   Number columns $(k+1) \ldots (k+p)$.
        }
8.   Form enode e_k from supernode x_k and all adjacent enodes.
9.   **for all** supernodes x adjacent to e_k {
10.   Label $\deg(x)$ as "unknown."
        }
11. $k \leftarrow k + p$
    }
12.   **for all** supernodes x where $\deg(x)$ is unknown {
13.   Update lists of adjacent supernodes and enodes of x.
14.   Check for various QuotientGraph optimizations.
15.   Compute $\deg(x)$.
    }
}

supernodes of minimal degree, they can all be eliminated before updating the quotient graph. This is the Multiple Minimum Degree Algorithm [58]. We show the Multiple Minimum Degree algorithm defined in terms of a quotient graph in Fig. 3.

Another recent optimization is to compute only the approximate degree of a supernode instead of its exact degree. This Approximate Minimum Degree (AMD) algorithm [2] can do faster degree computations, but disallows multiple elimination in order to obtain tighter bounds on the approximate degrees. The MMD algorithm in Algorithm 3 could be changed to an AMD algorithm with the following modifications: 1) the inner while loop in line 4 would be executed exactly once for each iteration of the outer loop and 2) the quotient graph optimizations and degree computations would be different (faster).

These two algorithms, MMD and AMD are the most different of all the fill-reducing orderings implemented in Spindle. All the others use either the MMD or AMD subsystems and simply replace the degree computation with a more robust (and expensive) heuristic such as "minimum increase in neighbor degree" or "modified minimum deficiency". Table 3 shows a list of algorithms and references for further study.

There are a list of standard features in modern implementations that requires introduction. For more detailed information, refer to the survey paper by George and Liu [34].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE 3
Several Greedy Fill-Reducing Heuristics.

Abbreviation	Algorithm Name	Primary Reference
MMD	Multiple Minimum Degree	Liu [58]
AMD	Approximate Minimum Degree	Amestoy, Davis and Duff [2]
MMDF	Modified Minimum Deficiency	Ng and Raghavan [64]
MMMD	Modified Multiple Minimum Degree	Ng and Raghavan [64]
AMF	Approximate Minimum Fill	Rothberg [73]
AMMF	Approximate Minimum Mean Local Fill	Rothberg and Eisenstat [74]
AMIND	Approximate Minimum Increase in Neighbor Degree	Rothberg and Eisenstat [74]

- **Multiple Elimination** [58]. Lazy update of the graph. When a vertex is eliminated from the elimination graph, the entire graph does not change, only the neighbors of the newly eliminated vertex. Multiple elimination eliminates an independent set of vertices of minimal degree before updating the elimination graph.

- **Mass Elimination** [35]. When eliminating a vertex v_k, there is often a subset of vertices adjacent to v_k that can be eliminated immediately after v_k with no additional fill, and saving additional elimination graph updates.

- **Indistinguishable Nodes**. Related to mass elimination. If at some step, k, in the elimination process two vertices $v_i, v_j \in G^k$ are adjacent, but otherwise have identical adjacency lists, we call them *indistinguishable nodes* (see also Definition 2.5). Furthermore, if two nodes are indistinguishable in G^k, they remain indistinguishable in G^{k+1}. Since they are indistinguishable, their degrees are identical, saving a degree computation for one of them. Furthermore, once one node is eliminated, the other indistinguishable nodes can be eliminated immediately afterward via mass elimination.

- Taken together, mass elimination and indistinguishable nodes offer an important advantage. We can group nodes into *supernodes* and need to compute the degree and update the elimination graph only once for each supernode. The size of each supernode must be stored to accurately compute the degree.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- **Incomplete Degree Update** [27]. Given two nodes, v_i, v_j in G^k, where the set of adjacent vertices of v_i is properly contained by v_j then v_j is said to be *outmatched* by v_i. The outmatched vertex need not have its degree computed again until v_i is eliminated.

- **Quotient Graph** [32]. Emulates an elimination graph in a fixed amount of storage through an extra level of indirection through *enodes*. Quotient graphs can have the same interface as an elimination graph, but must handle internal data differently.

  Instead of removing an eliminated vertex, a quotient graph relabels the vertex as an enode. Any edges between two vertices that are members of the same enode are redundant and can be removed.

  To compute the degree of a vertex in the elimination graph, a quotient graph must compute the size of its *reachable set*. The reachable set of a vertex in a quotient graph is simply the size of the set of all vertices directly adjacent through an edge, or a members of a common enode.

- **Element Absorption** [25]. Element absorption is to enodes, what indistinguishable nodes are to vertices. Since a quotient graph provides a compact representation of an elimination graph by storing enodes instead of fill edges, one can merge any two adjacent enodes into a superenode.

- **External Degree** [58]. Given a minimum degree algorithm which takes advantage of supernodes, the degree of a supernode need not include its own weight. The intuition is that all the vertices in a supernode are already an enode and so will not induce any new fill.

- **Precompression** [3]. Additional savings can be made if the graph is compressed *before* performing the minimum degree ordering. Precompressing the graph is a standard technique employed by many ordering and partitioning algorithms, the earliest reference to this technique is Duff, Reid, and Scott [26].

- **Approximate Update** [2, 40]. Instead of computing the degree of each supernode exactly, simply compute an upper bound. This can be done in a multiple elimination setting, but it double counts any supernode that is reachable through two different elimination paths [40]. Amestoy, Davis, and Duff [2] were able to tighten this bound by double counting only supernodes that are reachable

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

through two different elimination paths, neither of which being though the most recently formed enode. This added restriction also prevents the code from using multiple elimination.

- **Alternatives to Minimizing Vertex Degree** [64, 73, 74]. Most recent work focused on improving the quality of the ordering by eliminating supernodes on a more accurate basis than degree. Many of these modifications are also cited in Table 3.

$$
\begin{aligned}
\text{AMFscore}_k(i) &= \frac{(d^2-d)-(c^2-c)}{2},\\
\text{AMMFscore}_k(i) &= \frac{(d^2-d)-(q^2-q)}{2*\text{wgt}_k(i)},\\
\text{AMINDscore}_k(i) &= \frac{(d^2-d)-(q^2-q)}{2} - d*\text{wgt}_k(i),\\
\text{MMDFscore}_k(i) &= d^2-d-p-2*d*\text{wgt}_k(i),\\
\text{MMMDscore}_k(i) &= 2d - \max_{e\in\text{eadj}(i)} \text{wgt}(e).
\end{aligned}
$$

where

$$
\begin{aligned}
d &= \text{deg}_k(i) - \text{wgt}_k(i),\\
c &= \max_{e\in\text{eadj}_k(i)} |e| - \text{wgt}_k(i),\\
q &= e_k - \text{wgt}_k(i),\\
p &= \sum_{v\in\text{sadj}(\text{eadj}(i))\setminus\text{sadj}(k)} \text{wgt}(v)^2 - \text{wgt}(v).
\end{aligned}
$$

3.3.6 Asymptotic Complexity Analysis

Modern greedy fill reducing ordering algorithms have very sophisticated implementations. They are, therefore, difficult to analyze for asymptotic complexity. Some work has been done for AMD [2], but until now, MD and MMD have escaped analysis.

In this section, we analyze the most expensive loops in MD, MMD and AMD. While the degree of a vertex *can* be computed in these loops, there is also significant work being done to update the quotient

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

graph. The prevailing notion holds that the degree computation is the most expensive part of the algorithm. The truth is that the degree computation is asymptotically of the same order as the quotient graph update. When these two processes are totally divorced from each other, it is the quotient graph update that requires more time than the degree computation because it includes several lower order terms as well.

First, we cover some basic properties that are needed for the analysis. Then we derive the asymptotic bounds which are somewhat complicated. To give these bounds more meaning, we compute them for certain model problems in Section 3.3.7.

3.3.6.a Properties of a Quotient Graph

The quotient graph is a powerful and complicated data structure. It has many properties that need explanation and quantification before we can attempt a thorough analysis of the MD, MMD, and AMD algorithms.

Consider the state diagram in Figure 18 which shows the lifecycle of a quotient graph. In this discussion, any transition along an "eliminate_supernode" edge is an *elimination step*, or simply a step. Any series of transitions from a valid state to a semi-valid state and then back to a valid state is a *stage* of elimination. A stage can involve multiple elimination steps. In the case of single elimination, such as MD or AMD, then each stage has exactly one elimination step.

When the quotient graph is in a "valid" state, the internal adjacency lists of each supernode and enode is known, and the priorities of each supernode can be established by traversing these adjacency lists.

When any supernode is eliminated, the quotient graph enters into a "semi-valid" state. Upon elimination, the reachable set of the eliminated supernode becomes the supernode adjacency list of the newly formed enode. Although the eliminated supernode has become an enode and its adjacency lists have been updated, the adjacency lists of all its neighbors have not been touched. Indeed, one could not even accurately compute the degree of any of these supernodes until their adjacency lists are updated, hence they become ineligible for elimination for the remainder of this elimination stage. In multiple elimination algorithms, subsequent supernodes in the quotient graph can be eliminated as long as they were not rendered ineligible by a neighbor's elimination earlier in this stage.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

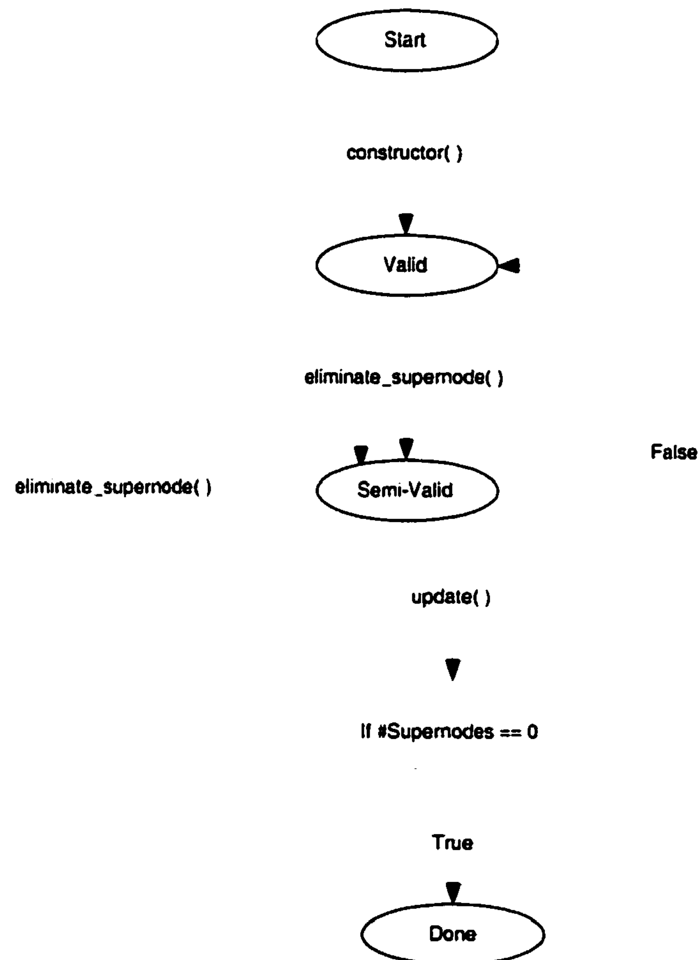


FIG. 18. *State Diagram of a Quotient Graph.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

When the quotient graph is updated, it actually updates the adjacency lists of all the ineligible supernodes and performs additional optimizations such as supernode amalgamation, outmatching, etc. All the remaining supernodes have their adjacency list completely defined and become eligible for elimination. Therefore the quotient graph returns to a valid state.

When the quotient graph has no remaining supernodes after updating, then the elimination order is completely defined and can be harvested from the otherwise empty graph class.

In this section, we use i for any node in the quotient graph (supernode or enode), k for the supernode eliminated at the kth step, s to denote supernodes in general, e for enodes in general, and r for supernodes in the reachable set of k. Thus, one can infer that in the elimination ordering π; $\pi(e) < k < \pi(r), \pi(s)$. To keep our illustrations simple and clear, we chose the input ordering of A to be the elimination ordering (as in Figure 15). Although this is certainly never the case in practice, by making $\pi(i) = i$ for our examples, we can drop the $\pi(\cdot)$ notation entirely for the rest of this document.

Careful distinctions must be made between *adjacency* in the quotient graph ($\text{adj}_{\mathcal{G}}(i)$), and adjacency in the corresponding elimination graph ($\text{adj}_G(i)$ or simply $\text{adj}(i)$). Adjacency in the quotient graph is the union of the *supernode adjacency* ($\text{sadj}(i)$), and *enode adjacency* ($\text{eadj}(i)$). Since supernodes and enodes are unique to the quotient graph, the $\mathcal{G}$ can be omitted. Whenever we are talking about the adjacency at a particular timestep k, we will use the graph subscripted at the kth step (G_k or $\mathcal{G}_k$).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

At the time that the quotient graph is transitioning to a valid state and k is the most recently eliminated supernode, the following are invariant.

$$\mathrm{reach}_{\mathcal{G}_k}(s) = \mathrm{adj}_{G_k}(s), \tag{3.8}$$

$$\mathrm{adj}_{\mathcal{G}}(i) = \mathrm{sadj}_{\mathcal{G}}(i) \cup \mathrm{eadj}_{\mathcal{G}}(i), \tag{3.9}$$

$$|\mathrm{adj}_{\mathcal{G}_k}(s)| \leq |\mathrm{adj}_{\mathcal{G}_0}(s)|, \tag{3.10}$$

$$|\mathrm{eadj}_{\mathcal{G}}(e)| = 0, \tag{3.11}$$

$$|\mathrm{sadj}_{\mathcal{G}_k}(e)| \leq |\mathrm{sadj}_{\mathcal{G}_{\pi(e)}}(e)|, \tag{3.12}$$

$$|\mathrm{adj}_{\mathcal{G}}(s)| \leq |A_{*,s}|, \tag{3.13}$$

$$|\mathrm{sadj}_{\mathcal{G}}(e)| \leq |L_{*,e}|. \tag{3.14}$$

Equation 3.8 restates that the reachable set of a quotient graph is the same as the adjacency set in the elimination graph. Equation 3.9 is the definition of adjacency on a quotient graph.

Equation 3.10 states that the size of the adjacency lists of any supernode s for all timesteps until it is eliminated is a strictly non-increasing function. This is true because there is no quotient graph transformation that involves adding edges to supernodes. Edges can change type from supernode adjacency to an enode adjacency when the adjacent supernode is eliminated. Edges are removed in any one of three cases: (1) because indistinguishable supernodes are amalgamated, (2) because the edges become redundant when adjacent supernodes are adjacent to a common enode, or (3) because the edges that used to point to some enodes are no longer valid because the enodes were absorbed by a newly formed enode. This newly formed enode is also adjacent, though it is likely to be listed (incorrectly) as an adjacent supernode.

We can make a similar assertion about the the size of the adjacency set of an enode. For the case of enodes, however, we can be more precise. Recall that when an enode is formed, it absorbs all of its adjacent enodes. Therefore, the enode adjacency of an enode is always zero as we show in Equation 3.11. We assert in Equation 3.12 that the size of the supernode adjacency list of an enode is also bounded by its size when the supernode is created. After an enode is formed, there are no quotient graph transformations

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

that add additional edges to an existing enode. Edges are only removed from enodes when indistinguishable supernodes are detected and amalgamated. Enodes themselves can of course be absorbed into larger enodes, but that only happens when a new enode is created.;

Equation 3.13 states that the adjacency list of a supernode in the quotient graph is bounded by the corresponding column in A. If we prohibit precompression, $\mathcal{G}_0(s) = A_{*,s}$ for all s. This becomes a inequality when we allow precompression, and it holds for all timesteps until the supernode is eliminated by virtue of Equation 3.10.

Equation 3.11 holds for all timesteps k after the enode e is formed until some timestep r when a reachable node of e is eliminated and e ceases to exist. If we disable supernode amalgamation and precompression it is easy to show that $|\text{sadj}_{\mathcal{G}_s}(e)| = |L_{*,e}|$. Thus the inequality in Equation 3.11 holds when using amalgamation, precompression, and Equation 3.12.

Lines 13–15 in Algorithm 3 show the update phase for the quotient graph and the reprioritizing of all the vertices adjacent to recently eliminated vertices in the elimination graph. If the algorithm is a single elimination scheme such as MD or AMD, then this set of vertices is exactly the reachable set of the most recently formed enode in the quotient graph. If the algorithm is a multiple elimination scheme, this set of updated vertices will be the union of all reachable sets of all enodes formed in the last elimination stage.

The way that the quotient graph updates itself and reprioritizes supernodes differs between approximate degree updates (which are necessarily single elimination algorithms) and exact degree updates (which may be single or multiple elimination schemes). To bound the performance of these algorithms, we show the parts of the quotient graph update subroutines that are the most expensive asymptotically. They have also been confirmed to be most expensive experimentally.

3.3.6.b Minimum Degree

We start with simplest case, a single elimination exact minimum degree ordering (called Minimum Degree). The update set in line 1 of Algorithm 4 is exactly the reachable set of the supernode that was eliminated since the last update. We start with a quotient graph that is in a state where an enode has already been formed by eliminating a supernode and amalgamating any adjacent enodes. All supernodes

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ALGORITHM 4 (The most expensive loop in the MD and MMD update process.)

```
1.  for each supernode r in the update set {
2.      timestamp ⟵ nextStamp() ;                // get value larger than any in visited
3.      visited[ r ] ⟵ timestamp ;               // prevent visiting self
4.      degree[ r ] ⟵ 0 ;                        // no self weight, external degree
5.      for each enode e in eadj(r) {

            // consider principal enodes, removing non-principal ones
6.          while ( e ≠ parent( e ) ) {          // while e is not principal
7.              eadj(r) ⟵ eadj(r) \ e ;          //   remove e from eadj(r)
8.              e ⟵ parent( e ) ;                //   advance to parent enode
            }
9.          eadj(r) ⟵ eadj(r) ∪ e ;              // write back principal enode

            // visit each adj snode once, and add to degree
10.         if ( visited[ e ] < timestamp ) {
                // if not yet visited . . .
11.             visited[ e ] ⟵ timestamp ;       //   it is now
12.             for each snode s in sadj(e) {
13.                 if ( visited[ s ] < timestamp ) {
14.                     visited[ s ] ⟵ timestamp ;
15.                     degree[ r ] ⟵ degree[ r ] + weight[ s ] ;
                    }
                }
            }

        }
16.     for each snode s in sadj(r) {
            // visit each principal unvisited adj snode once
17.         if ( ( visited[ s ] < timestamp ) and ( parent( s ) = s ) ) {
18.             visited[ s ] ⟵ timestamp ;
19.             degree[ r ] ⟵ degree + weight[ s ] ;
20.         } else {                             // if s is already visited or not principal
21.             sadj(r) ⟵ sadj(r) \ s ;          //   remove it
            }
            // NOTE: if s is non-principal, its parent is already in sadj(r)
        }
    }
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

reachable to the old supernode, however, have yet to be updated. The same holds true for any supernodes that were adjacent to one or more of the amalgamated enodes. This process includes: (1) removing any old adjacent enodes that have been merged into the newest one, (2) removing adjacent snodes that are already reachable though an enode, and (3) removing the newest enode from the old snode list, and computing the external degree of the node.

We proceed now with a line by line examination of the algorithm. The outermost loop (lines 1–22) iterates over the set of all supernodes in the quotient graph that need to be updated. Since all adjacency lists are unordered, a timestamping mechanism is used to prevent double visiting. This involves using a value `timestamp` (line 2) that is larger than any value found in the array `visited` (line 3). Since no index is used at the same time for enodes and supernodes, we can use the `visited` array for both without fear of collisions.

Now looping over all the adjacent enodes (lines 5–15), we perform two tasks. First we remove any non-principal enodes in the adjacency lists and replace them with the principal enode (lines 6–9). To see why this is necessary, consider a three node quotient graph with two supernodes i, j and one enode e. Furthermore assume there are two edges (i, e) and (e, j). Now we eliminate supernode i, which we transform into an enode that absorbs the adjacency list of e. Now it is time to update. The new enode e has an edge to supernode j, but j has an edge to a now non-existent enode e. To correct this, j must replace this with an edge to the enode that absorbed e, namely j.

Each time we produce a principal enode adjacent to a supernode in the update set, we insure that we examine its list of adjacent supernodes only once (lines 10–12). The purpose of this is twofold. First, we want to timestamp all supernodes reachable through an enode (line 14) to strip off redundant edges later (line 21). Second, we want to compute the size of the reachable set of supernode r (line 15) which is also the degree of r in the elimination graph. Note that we marked the updated supernode as visited in line 3 to prevent revisiting it through an adjacent enode.

The final inner loop (lines 16–21) iterates over the set of adjacent supernodes of each updated supernode. If the adjacent supernode has not already been visited through an enode (lines 10–11), if it is not really a newly formed enode (which would have been visited and flagged in line 11), and if it is a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

principal supernode, then we mark it as visited and add its weight to the degree of the updated supernode. Otherwise, the supernode found can be discarded.

Note that there is no need to follow parent pointers with compressed supernodes like there is with amalgamated enodes. Recall that supernodes are only compressed if they are indistinguishable, meaning that, among other things, they have identical neighbors. Therefore if i is a supernode that is being updated and it has in its adjacency list an edge to a supernode j that is compressed into another supernode k, then we know that j and k must both have an edge to i and that i must also have an edge to k. The edge to j can then simply be discarded.

Working from the innermost loops out, lines 12–15 are $\mathcal{O}(|\mathrm{sadj}(e)|)$. The cost of the loop that advances through parent pointers of enodes (lines 6–8) can be restricted to one by not forming the elimination trees explicitly during the elimination, but afterwards. In this case a minimal representation called the *front tree* [58] will keep this loop from repeating more than a constant number of iterations. The encompassing loop over all principal enodes (lines 5–15) is therefore $\mathcal{O}(\sum_{e\in \mathrm{eadj}(r)} |\mathrm{sadj}(e)|)$. The subsequent loop (lines 16–21) is $\mathcal{O}(|\mathrm{sadj}(r)|)$.

The time taken for the MD algorithm is therefore

$$\mathcal{O}\left(\sum_{k=1}^{n_p} \sum_{r\in \mathrm{sadj}(k)} \left(|\mathrm{sadj}(r)| + \sum_{e\in \mathrm{eadj}(r)} |\mathrm{sadj}(e)|\right)\right), \tag{3.15}$$

where n_p is the number of principal supernodes eliminated. Except for pathological cases in which no supernodes are indistinguishable throughout the elimination, $n_p << n$.

3.3.6.c Multiple Minimum Degree

The time taken for the MMD algorithm is related to MD. We can use the same algorithm and analysis presented from Figure 4. The difference is that the update is performed only after a set of independent supernodes of low (or minimum) degree has been eliminated. Hence the complexity is

$$\mathcal{O}\left(\sum_{j=1}^{n_h} \sum_{r\in \cup_{k\in K_j} \mathrm{sadj}(k)} \sum_{e\in \mathrm{eadj}(r)} |\mathrm{sadj}(e)|\right), \tag{3.16}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

where n_h is the height of the front tree (which is related to a supernodal elimination forest), K_j is the set of newly eliminated nodes at the jth step, and r is the reachable set of the newly eliminated snodes.

Comparing Equations 3.15 and 3.16, we can see why MMD should outperform MD. Even though both are stuck with a triply nested loop (lines 12–15, Figure 4), MMD updates less often. Unless the resulting elimination forest is a simple path, $n_h < n_p$. The second sum in Equation 3.16 is the union of all supernodes reachable from any supernode eliminated at that stage.

3.3.6.d Approximate Minimum Degree

The Approximate Minimum Degree (AMD) ordering avoids the triply nested loop inherent in exact degree computations. It does this by computing an upper bound on the size of the reachable set. Assume the weight of an enode is defined to be the sum of the weights of its adjacent supernodes. The size of the reachable set of any vertex could never be more than the sum of the weights of its adjacent supernodes and enodes. Indeed, it could be quite less [40]. What makes the bound so loose is the fact that the same supernode could be reachable through several different enodes, yet should only be counted once. Amestoy, Davis, and Duff [2] tightened this bound by making all supernodes reachable through the most recently formed enode counted only once. All other supernodes that do not share an edge with the most recently formed enode may be double counted. The downside is that the quotient graph needs to be updated after every elimination, thereby preventing AMD from employing multiple elimination. However, since the only nodes that need to be updated are the ones adjacent to the most recently formed enode, the approximation does well in practice.

The AMD algorithm uses intermediate values that are essentially set differences. Instead of summing the weights of all adjacent supernodes and enodes, AMD sums the weights of all supernodes and the "set-diffs" of all enodes.

$$\text{setDiff}[\, e \,] = \begin{cases} \sum_{s \in \text{sadj}(e)} \text{weight}[\, s \,] & \text{if } e = k, \\ \sum_{s \in \{\text{sadj}(e) \setminus \text{sadj}(k)\}} \text{weight}[s] & \text{if } e \neq k, \end{cases} \tag{3.17}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ALGORITHM 5 (The most expensive loop in the AMD update process)

```
    // 1. compute the setDiff's
1.  timestamp = nextStamp() ;                    // get value larger than any in visited
2.  k = most recent eliminated node ;
3.  for each supernode r in sadj(k) {
4.      if ( parent( r ) ≠ r ) {                 // If r is not principal ...
5.          sadj(k) ⟵ sadj(k) \ r ;             //   remove it
        } else if ( visited[ r ] < timestamp ) {  // else if r has not been visited ...
6.          visited[ r ] ⟵ timestamp ;          //   it is now.
7.          for each enode e in eadj(r) {

                // consider principal enodes, removing non-principal ones
8.              while ( e ≠ parent( e ) ) {      // while e is not principal
9.                  eadj(r) ⟵ eadj(r) \ e ;     //   remove e from eadj(r)
10.                 e ⟵ parent( e ) ;           //   advance to parent enode
                }
11.             eadj(r) ⟵ eadj(r) ∪ e ;         // write back principal enode

                // compute setdiff for enodes
12.             if ( e ≠ k ) {
13.                 if ( visited[ e ] < timestamp ) {
14.                     visited[ e ] ⟵ timestamp ;
15.                     setDiff[ e ] ⟵ weight[ e ] − weight[ r ] ;
                    } else {
16.                     setDiff[ e ] ⟵ setDiff[ e ] − weight[ r ] ;
                    }
                }
            }
        }
    }

    // 2. compute degree and hashvalues for indistinguishable node detection
17. for each supernode r in sadj(k) {
18.     approx_degree[ r ] ⟵ weight[ k ] − weight[ r ] ;
19.     hashval ⟵ 0 ;
20.     for each enode e in eadj(r) {
21.         if ( e ≠ k ) {
22.             if ( setDiff[ e ] = 0 ) {
23.                 weight[ e ] = 0 ;
24.                 eadj(r) ⟵ eadj(r) \ e ;
25.             } else if ( setDiff[ e ] > 0 ) {
26.                 approx_degree[ r ] ⟵ approx_degree[ r ] + setDiff[ e ] ;
27.                 hashval ⟵ hashval + e ;
                }
            }
        }
28.     for each supernode s in sadj(r) {
29.         if ( visited[ s ] < timestamp ) {
30.             approx_degree[ r ] ⟵ approx_degree[ r ] + weight[ s ] ;
31.             hashval ⟵ hashval + s ;
            }
        }
32.     sorter.insert( hashval, r ) ;
    }
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

where k is the most recently formed enode. This is still an upper bound on the actual degree of a supernode, but it is useful in practice.

The quotient graph update for the approximate degree algorithms is very different than for exact degree. We present the most expensive parts of the update procedure in Algorithm 5. As we did with MD and MMD, we will go through a line by line explanation of this algorithm, then compute its asymptotic complexity.

We start by generating a timestamp for this update, and noting the most recently formed enode (lines 1–2). The first main loop (lines 3–16) computes the setDiff parameter for all enodes adjacent to a node that needs updating. It also removes non-principal supernodes (lines 4–5) and enodes (lines 8–11). The first time an enode that is not the most recently formed one is encountered (lines 12–13), it is timestamped and its setDiff is initialized to its weight minus the weight of the snode that is adjacent to both it and the most recently formed enode (line 15). Every time we revisit that enode through another principal supernode, we subtract the weight of that supernode from its setDiff (line 16).

Now at line 17 of Algorithm 5, the setDiff values have been computed, and we are set to compute the approximate degrees of all the supernodes adjacent to the newest enode, k (lines 17–34). In this loop, we also take the liberty of computing a hash value that will be used later for detecting indistinguishable supernodes. The degree is initialized to the weight of the newest enode minus the weight of the supernode to be updated (line 18). The supernode is necessarily adjacent to the enode otherwise it would not need updating. Furthermore, we are actually computing the *external degree*, which does not include the weight of the vertex itself. The hash value is simply the sum of all adjacent supernodes and enodes, so it is initialized to zero (line 9). As with exact degree updates, we are testing for indistinguishability in the quotient graph, which is not as strong as indistinguishability in the elimination graph.

The first of the two inner loops (lines 20–28) iterates over the enodes adjacent to the supernode we're updating. If the enode is an edge back to the most recently formed, we have already added its contribution to the degree, and can safely skip it (lines 21–22). If the setDiff for this enode is non-negative, then we add its contribution to the degree and the hash value (lines 23–25). If, however, the setDiff is zero, then

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

this enode has no adjacent supernodes that are not also adjacent through the newest enode. In this case the enode itself can be removed.

This feature, which is unique to the AMD implementations, may be helpful in limited cases. For example, look back at Figure 15 on page 60. Looking at step 3, we see that vertices 4, 5 and 6 are indistinguishable in G_3, but the corresponding supernodes in $\mathcal{G}_3$ are not. This is because enode 2 is adjacent to supernodes 5 and 6. With the AMD algorithm as presented in Algorithm 5, the quotient graph would remove enode 2 and supernodes 4, 5, and 6 would be detected as indistinguishable.

Continuing with our discussion of the quotient graph update for approximate degree algorithms in Algorithm 5, we iterate over the adjacent supernodes and add their contributions to the degree and hash value (lines 29–32). Then the supernode-hash value pair is inserted into a sorting mechanism for later use (line 33) and the approximate degree is stored as well (line 34).

As to the complexity of the algorithm, we will examine only the second loop of Algorithm 5 (lines 17–34). The first inner loop (lines 20–28) is obviously $\mathcal{O}(|\text{eadj}(r)|)$, and the second $\mathcal{O}(|\text{sadj}(r)|)$. Using Equation 3.9, we can show that one iteration of the outer loop runs in $\mathcal{O}(|\text{adj}_{\mathcal{G}_k}(r)|)$ time. The time taken for the AMD algorithm is therefore

$$\mathcal{O}\left(\sum_{k=1}^{n_p} \sum_{r \in \text{sadj}(k)} |\text{adj}_{\mathcal{G}}(r)|\right). \tag{3.18}$$

3.3.7 Model Problems

The most complicated aspects about combinatorial ordering algorithms such as the minimum priority family and Sloan [52] is that the asymptotic complexity depends on the quality of the result. These minimum priority algorithms are very sensitive to tie breaking, and the model problems start will all nodes having the same degree. Therefore, we must assume a "perfect" tie-breaking strategy that MD, MMD, and AMDall follow.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.3.7.a Ring

Assume the graph of a matrix is a simple ring of $n = 2^t$ vertices. Then at any time, the $|\text{adj}_G(i)|$ of any supernode or enode in the quotient graph is at most two. Therefore, we can bound the supernode adjacency and enode adjacency by two. We assume the elimination order where we circle the ring and eliminate every second supernode encountered. This ordering is the best case for multiple elimination. We also note that the ring is a pathological case where no supernodes are indistinguishable at any time in the ordering.

$$\begin{aligned}\text{MD(ring)} &= \left(\sum_{k=1}^{n} \sum_{r \in \text{sadj}(k)} \sum_{e \in \text{eadj}(r)} |\text{sadj}(e)|\right) \\ &\leq \left(\sum_{k=1}^{n} 2 \times 2 \times 2\right) = \mathcal{O}(8n) = \mathcal{O}(n).\end{aligned}$$

$$\begin{aligned}\text{MMD(ring)} &= \left(\sum_{j=1}^{n_h} \sum_{r \in \cup_{k \in K_j} \text{sadj}(k)} \sum_{e \in \text{eadj}(r)} |\text{sadj}(e)|\right) \\ &\leq \left(\sum_{j=1}^{n_h} \sum_{r \in \cup_{k \in K_j} \text{sadj}(k)} 4\right) \leq (4n) = \mathcal{O}(n).\end{aligned}$$

$$\begin{aligned}\text{AMD(ring)} &= \left(\sum_{k=1}^{n} \sum_{r \in \text{sadj}(k)} |\text{adj}_{\mathcal{G}_k}(r)|\right) \\ &\leq \left(\sum_{k=1}^{n} \sum_{r \in \text{sadj}(k)} 2\right) \\ &\leq \left(\sum_{k=1}^{n} 4\right) \leq (4n) = \mathcal{O}(n).\end{aligned}$$

Note that the complexity is linear in all cases. This is because the supernode and enode adjacencies are bounded by constants. MD is the most expensive, as expected, and MMD is faster because it only

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

updates a supernode after both its left and right neighbors have been eliminated. AMD, on the other hand, is also faster than MD in this case because

$$\begin{aligned} |\mathrm{adj}_{\mathcal{G}_k}(r)| &= |\mathrm{eadj}(r) \cup \mathrm{sadj}(r)| = 2 \\ &< \sum_{e \in \mathrm{eadj}(r)} |\mathrm{sadj}(e)| = 4. \end{aligned}$$

3.3.7.b Torus

Assume a graph of a standard five-point stencil on a torus with $n = 2^{2l}$ vertices. Using an optimal tie-breaking strategy, the adjacency set of a supernode is bounded by four, but the same cannot be said for the supernode adjacency of an enode. Luckily, when using supernode amalgamation, we can bound the principal supernode adjacency of an enode by eight in this case.

To illustrate why the supernode adjacency of an enode in a torus is bounded by eight when using an optimal tie-breaking strategy, we show a series of modified views of quotient graphs in Figure 19. In this figure, supernodes are represented as filled circles, enodes as empty squares, and edges are not drawn. Instead, the edges are implied by a supernode being "near to" an enode. Lines in this modified view denote the extent of an enode's reach. Therefore in Figure 19 we draw an enode in space, bounded by a lines. All supernodes that are on the boundary line, are adjacent to that enode in the quotient graph.

Figure 19.1 represents the situation after the first mass of elimination of MMD, in which half of the supernodes have been eliminated. At this stage, each enode has exactly four adjacent supernodes. Each supernode has an *external degree* (sum of the weights of the reachable set, minus self weight) of eight. Remember that for 2-D illustration of a torus, the boundaries wrap around. After the second mass elimination, as shown in Figure 19.2, another full fourth of the remaining supernodes are eliminated. Now each enode has eight adjacent supernodes. The external degree of the supernodes now depends on whether they are adjacent to two enodes, or four. Each enode is similar to a finite-element panel with the supernode either being on the corner or side of the panel. The external degree for supernodes on the side of the panel (adjacent to two enodes) in Figure 19.2 is 11, the external degree for corner supernodes is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

higher at 15. Figure 19.3a shows the graph after a maximal independent set of side supernodes has been eliminated. It would appear that the size of the sadj of the enodes has increased beyond eight.

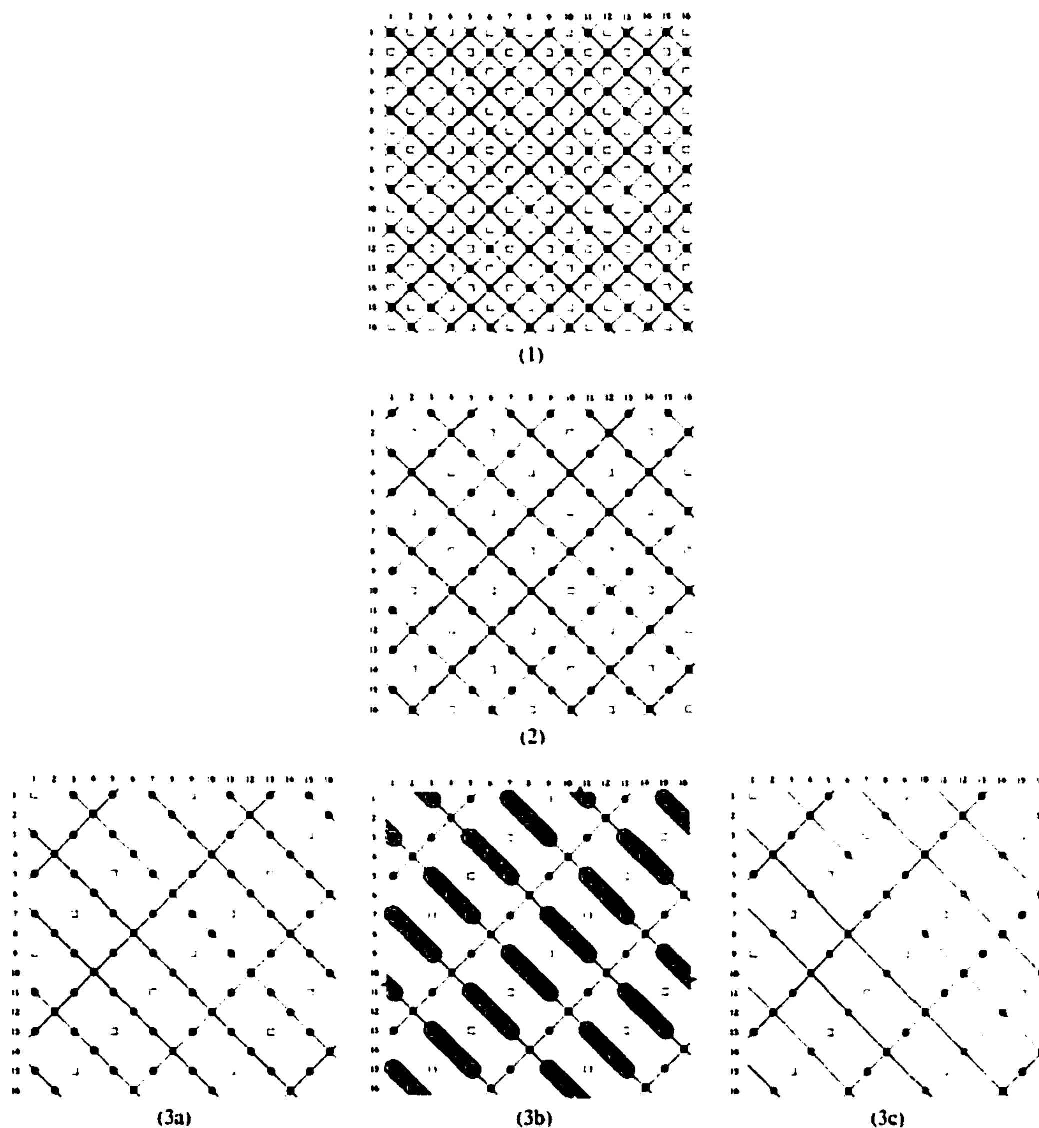
Consider the enode at coordinates (5,5) in Figure 19.3a. It has three supernodes located at coordinates (5,4), (6,5), and (7,6) that it shares exclusively with the enode at (7,3). At this point, these three supernodes are indistinguishable from each other (Figure 19.3b) and can be compressed into a single representative supernode (Figure 19.3c). After compression, we see that each enode in Figure 19.3c is adjacent to no more than eight supernodes. Now we compute the external degree of all the supernodes. The corner supernodes have external degree 31 (14 uncompressed supernodes, plus 6 compressed supernodes of weight 3, minus self weight). The uncompressed edge supernodes have external degree of 19 (8 uncompressed supernodes, plus four compressed supernodes of weight 3, minus self weight). The compressed supernodes have the minimum external degree of 13 (10 uncompressed supernodes, plus two compressed supernodes of weight 3, minus self weight of 3) and so a maximal independent set of them is chosen for the next mass elimination.

The process repeats in Figures 19.4a–19.5c and will continue to do so until there are only two enodes left (Figure 19.6). At this point, every supernode remaining is indistinguishable from every other, so they are all compressed into one supernode and eliminated in the very next step.

A similar analytical tactic was used by Berman and Schnitger [10] to prove bounds on the size of the factor when using a minimum degree algorithm. Whereas we chose an optimal tie-breaking strategy for computing bounds for execution time of the algorithm, they chose one that grew the cliques in the factor to be very large. Because of the quotient graph and supernode amalgamation, the asymptotic complexity of the ordering algorithms is affected more by the number of adjacent cliques than the size of these cliques.

Now that we have verified that for a quotient graph of a 2-D torus with an optimal tie-breaking strategy $|\text{sadj}(e)| \leq 8$ and $|\text{eadj}(s)| \leq 4$, we are ready to compute the bounds for MD, MMD and AMD.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



Continued on next page . . .

FIG. 19. *The Quotient Graph while eliminating a 16 × 16 torus. These are modified views of the quotient graph where lines denote boundaries of an enode's connectivity. (1) After the first mass elimination, (2) after the second mass elimination, (3a) after the third, (3b) highlighting indistinguishable nodes, (3c) after indistinguishable nodes are compressed into a supernode.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

... Continued from previous page.

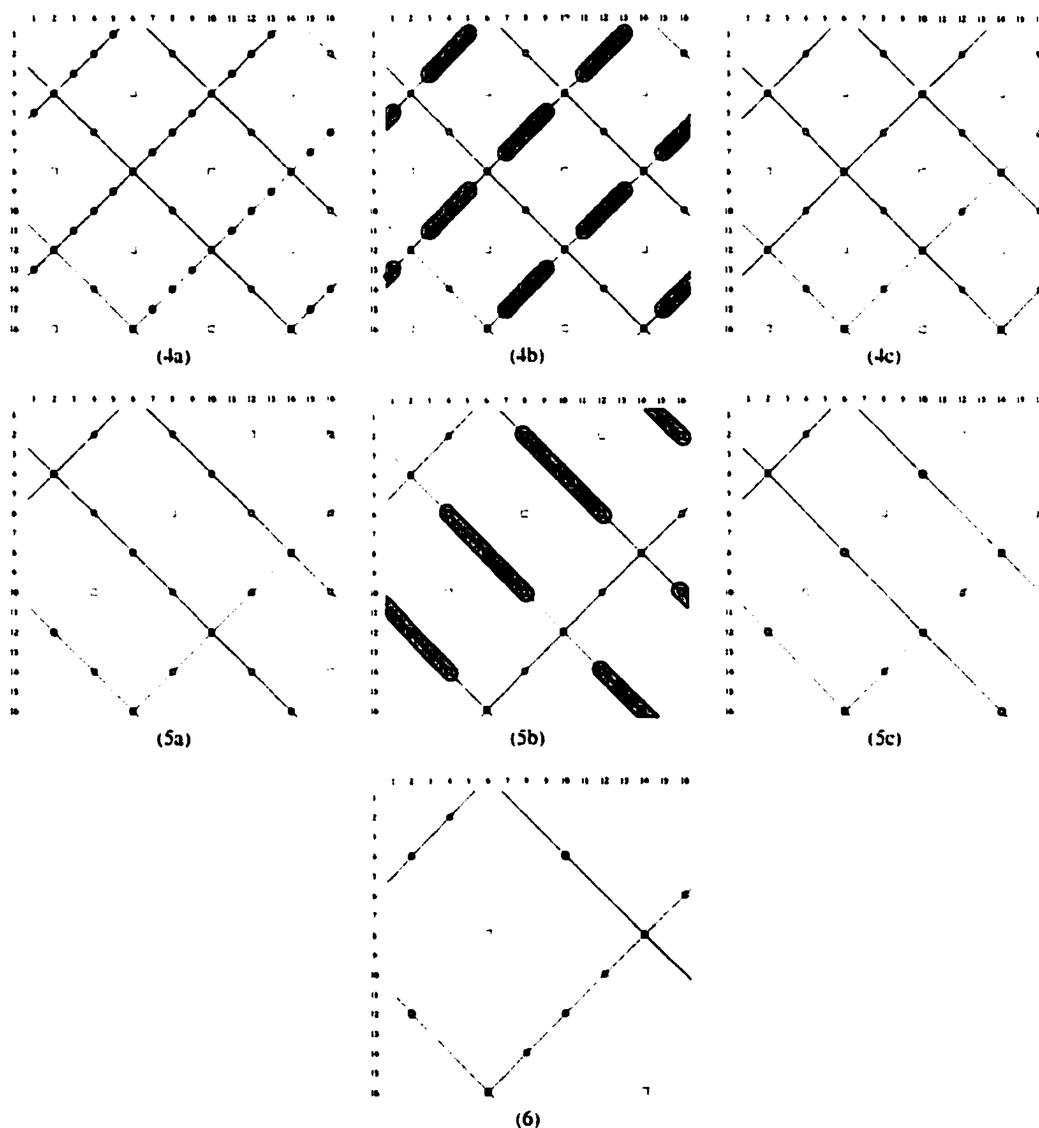


FIGURE 19 (Continued): Similarly for (4a), (4b), (4c), (5a), (5b) and (5c). At (6), there are only two enodes remaining, all supernodes are adjacent to both, so they are all compressed to a single supernode.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$
\begin{aligned}
\text{MD(torus)} &= \left(\sum_{k=1}^{n} \sum_{r \in \text{sadj}(k)} \sum_{e \in \text{eadj}(r)} |\text{sadj}(e)| \right) \\
&\leq \left(\sum_{k=1}^{n} \sum_{r \in \text{sadj}(k)} \sum_{e \in \text{eadj}(r)} 8 \right) \\
&\leq \left(\sum_{k=1}^{n} \sum_{r \in \text{sadj}(k)} 4 \times 8 \right) \\
&\leq \left(\sum_{k=1}^{n} 8 \times 4 \times 8 \right) \\
&\leq (n \times 8 \times 4 \times 8) = 256n = \mathcal{O}(n).
\end{aligned}
$$

$$
\begin{aligned}
\text{MMD(torus)} &= \left(\sum_{j=1}^{h} \sum_{r \in \cup_{k \in K_j} \text{sadj}(k)} \sum_{e \in \text{eadj}(r)} |\text{sadj}(e)| \right) \\
&\leq \left(\sum_{j=1}^{h} \sum_{r \in \cup_{k \in K_j} \text{sadj}(k)} \sum_{e \in \text{eadj}(r)} 8 \right) \\
&\leq \left(\sum_{j=1}^{h} \sum_{r \in \cup_{k \in K_j} \text{sadj}(k)} 4 \times 8 \right) \\
&\leq \left(32 \times \left(\frac{n}{2} + \frac{3}{8}n + \frac{3}{16}n + \frac{3}{32}n + \sum_{j=4}^{h} |r \in \cup_{k \in K_j} \text{sadj}(k)| \right) \right) \\
&= \left(32 \times \left(\frac{n}{2} + 3n - \frac{3n}{4} \right) \right) \\
&= (32 \times \frac{11}{4} n) = 88n = \mathcal{O}(n).
\end{aligned}
$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$\begin{aligned}
\text{AMD(torus)} &= \left(\sum_{k=1}^{n} \sum_{r \in \text{sadj}(k)} |\text{adj}_{\mathcal{G}_k}(r)|\right) \\
&= \left(\sum_{k=1}^{n} \sum_{r \in \text{sadj}(k)} |\text{eadj}(r) \cup \text{sadj}(r)|\right) \\
&\leq \left(\sum_{k=1}^{n} \sum_{r \in \text{sadj}(k)} 4\right) \\
&\leq \left(\sum_{k=1}^{n} 8 \times 4\right) = (32n) = \mathcal{O}(n).
\end{aligned}$$

Now consider the elimination of the first $n/2$ supernodes in the quotient graph of a torus. Then we know

$$\text{MD}_{\frac{n}{2}}(torus) \leq \left(\sum_{k=1}^{n/2} \sum_{r \in \text{sadj}(k)} \left(\sum_{e \in \text{eadj}(r)} |\text{sadj}(e)|\right)\right). \tag{3.19}$$

Since we also know that for the first $n/2$ supernodes the size of the supernode adjacency and enode adjacency is exactly 4, we can simplify Equation 3.19 to

$$MD_{\frac{n}{2}}(torus) \leq \frac{n}{2} \times 4 \times 4 \times 4 = 32n. \tag{3.20}$$

For multiple minimum degree, since the first $n/2$ supernodes form an independent set, they can all be eliminated before the first quotient graph update. Therefore the outer sum in Equation 3.16 has but a single term.

$$\begin{aligned}
\text{MMD}_{\frac{n}{2}} &\leq \left(\sum_{j=1}^{1} \sum_{r \in \cup_{k \in K_j} \text{sadj}(k)} \sum_{e \in \text{eadj}(r)} |\text{sadj}(e)|\right) & (3.21) \\
&\leq 1 \times \frac{n}{2} \times 4 \times 4 = 8n. & (3.22)
\end{aligned}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Finally, the running time for the first $n/2$ supernodes for AMDis

$$\mathrm{AMD}_{\frac{n}{2}}(torus) \leq \left(\sum_{k=1}^{n/2} \sum_{r \in \mathrm{sadj}(k)} |\mathrm{adj}_{\mathcal{G}_k}(r)|\right) \quad (3.23)$$

$$\leq \frac{n}{2} \times 4 \times 4 = 8n. \quad (3.24)$$

3.3.8 Simplification

For all our model problems, we see that the bounds are linear. This is mainly due to the fact that the models were chosen specifically because we could bound the size of the supernode and enode adjacencies by small constants. We suspect that this is true for whole classes of graphs such as planar graphs, though there is no known proof at this time to support our conjecture that quotient graphs preserves planarity.

Before discussing these results, we want to simplify the notation in Equations 3.15, 3.16, and 3.18 which are reproduced below.

$$\mathrm{MD} = \mathcal{O}\left(\sum_{k=1}^{n_p} \sum_{r \in \mathrm{sadj}(k)} \left(|\mathrm{sadj}(r)| + \sum_{e \in \mathrm{eadj}(r)} |\mathrm{sadj}(e)|\right)\right),$$

$$\mathrm{MMD} = \mathcal{O}\left(\sum_{j=1}^{n_h} \sum_{r \in \cup_{k \in K_j} \mathrm{sadj}(k)} \sum_{e \in \mathrm{eadj}(r)} |\mathrm{sadj}(e)|\right),$$

$$\mathrm{AMD} = \mathcal{O}\left(\sum_{k=1}^{n_p} \sum_{r \in \mathrm{sadj}(k)} |\mathrm{adj}_{\mathcal{G}_k}(r)|\right).$$

We consider first the AMDbound. The term inside the sums $\mathrm{adj}_{\mathcal{G}_k}(r)$ term represents the cost per reachable node updated. Using Equations 3.10 and 3.13, we know that $|\mathrm{adj}_{\mathcal{G}_k}(r)| \leq |A_{*,r}|$. Furthermore we observe that (ignoring the effects of precompression and supernode amalgamation) the double sum $\sum_{k=1}^{n_p} \sum_{r \in \mathrm{sadj}(k)}$ simply sums all nonzero entries $L_{k,r}$. Therefore, we can reverse the order of the summation show that

$$\sum_{k=1}^{n_p} \sum_{r \in \mathrm{sadj}(k)} \leq \sum_{r=1}^{n} |L_{*,r}|.$$

This equation becomes a strict equality if precompression and supernode amalgamation are disabled. This is also captured in the observation (again assuming no precompression or supernode amalgamation) that

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

each reachable supernode r is updated a number of times equal to the number of nonzeros in $L_{r,\bullet}$. A simplified bound for AMDcan be written as

$$AMD = \mathcal{O}\left(\sum_{r=1}^{n} |L_{r,\bullet}||A_{\bullet,r}|\right).$$

This is identical to the bound reported by Amestoy, Davis, and Duff [2].

Unfortunately, applying this same technique does not remove all of the summations from the bounds for MD and MMD. In fact, this technique makes little sense for MMD since several vertices adjacent to r can be eliminated with r only being updated once. From the matrix point of view, each reachable supernode is updated at most the number of nonzeros in $L_{r,\bullet}$, and most likely much less so.

It is obvious that MMD is faster than MD and that AMDis faster than MD. The relationship between MMD and AMDtakes some explaining. When there are no independent sets of vertices of minimal degree, MMD can perform as badly as MD. However, every time $L_{\bullet,K_j}$ has two or more entries on the same row, MMD performs only one update. Under optimal conditions MMD approaches AMD. When examining the elimination of the first $n/2$ supernodes in the torus, we were able to show that MMD is as fast as AMD. We will show experimentally later, that the first few iterations MMD can actually go faster than AMD, most likely because AMD has a larger hidden constant in the asymptotic analysis.

3.3.9 Contribution

The fill reduction problem is so ingrained in sparse matrix computations, that every person who ever solved a sparse linear system of equations probably used an algorithm like MMD, or some equivalent. The user base is easily in the tens of thousands. This problem has been actively research for the past 20 years. The number of papers published on this subject is easily in the hundreds.

We are the first to prove a detailed complexity analysis for MD, MMD, and AMD. We are also the first to point out that, contrary to what is in literature, the quotient graph update is more time consuming than computing the degrees of the supernodes.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In Section 4.3.3 we show how this insight into the differing characteristics of MMD and AMD lead us to implement a fill reducing poly algorithm. In Section 5.2.3 we show some preliminary results from this polymorphic algorithm.

3.4 Summary

It is one thing to implement an algorithm and demonstrate that it works well. It is quite another to analyze it and know how it works and why it works well. In our research, we prefer the latter.

For the Sloan algorithm, we were able to reduce the asymptotic complexity from $\mathcal{O}(n^{5/3})$ for 3-D meshes with good separators to $\mathcal{O}(n \log n)$. We non-dimensionalized the parameter space and identified a previously unreported behavior of the algorithm: the class-1/class-2 phenomenon. We were also able to generalize the algorithm from a reordering algorithm to one that refined an existing ordering.

Greedy fill reducing ordering algorithms have been thoroughly examined for over 20 years. Yet, we were the first to analyze the complexity of these algorithms and definitively rank their asymptotic behavior. We also correct the "common wisdom" that computing the degree is the most time-consuming part of the algorithm: analytically as well as experimentally. Our research also shows that MMD may be asymptotically slower than AMD, though there are cases — such as the beginning of the elimination — where MMD is faster in practice. This vein of research leads us to develop polyalgorithms that switch the exact algorithm dynamically.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4. SOFTWARE

We were initially attracted to object-oriented software from personal experience; our procedural codes failed to scale well with increasing layers of complexity. All of our work in the wavefront reduction problem was originally done in ANSI C [52]. Subsequently, we retrained ourselves in object-oriented programming, C++, and Java, then abandoned all the previous code and started from scratch.

Our firm foundation in state-of-art algorithms put us on track to build first class software, but that result is not guaranteed. The transformation from algorithm to software is neither automated nor easy, especially considering the high goals we set for ourselves back in Section 2.2.

We start by listing what resources were used in developing this software in Section 4.1 The basic inheritance hierarchy is explained in Section 4.2. Section 4.3 highlights some of the more attractive (and complex) features of the software. We learned some hard, but practical, lessons in building this software. Some of these are discussed in Section 4.4.

4.1 Resources Used

Our primary computing platform was Sun workstations running Solaris 2.6. Recently we have had increasing access to WindowsNT and Linux computers. Because we skirted on the cutting edge with the Standard Template Library (STL), most of the early code was developed using the GNU C++ compiler and the Silicon Graphics implementation of STL[1]. Later, we moved to the EGCS compiler[2] which eventually included STL. STL is now a part of the standard C++ standard library [78].

Managing the codebase is done with a collection of shell scripts, perl scripts, an extensive makefile setup inspired by PETSc, and RCS. Jam/MR[3] was investigated as an alternative to makefiles, but was then rejected because it lacked stability at the time. Debugging was done with Purify[4] and an excellent free graphical front end to GDB called DDD[5].

---

[1] `http://www.sgi.com/Tech/STL.`

[2] EGCS was originally an experimental offshoot from the GNU compiler. The EGCS compiler has now superseded the GNU compiler and is officially GNU C++ version 2.95. See `http://egcs.cygnus.com` for more information.

[3] `http://www.perforce.com/jam/jam.html`

[4] `http://www.rational.com/products/purify_unix`

[5] `http://www.cs-tu.bs.de/softech/ddd`

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Unit and regression testing was done with a lot of print statements and the UNIX `diff` command. It was semi-automated by shell scripts and makefiles. Documentation was generated using `doc++`[6], a free package that creates LaTeX or HTML documentation from commands imbedded in header comments.

Spindle was designed and implemented without any object modeling tools or an integrated development environment. With the exception of the underlying operating system the only non-free software used was Purify. While there are several free or "open source" tools available, they are not all high quality pieces of software. A carpenter can build a house with just a hammer and a saw, but it goes much faster with power tools. The same applies to software development and the piecemeal way in which a collection of free development tools was assembled.

4.2 Design

The very first partition in our problem space was to make a distinction between data structures and algorithms. Data structures are a structured collection of information that can be valid, invalid, stored on disk, queried or modified. Algorithms were computational engines that were hooked up to inputs, configured to perform a service, run, queried for results, and possibly reset for another run. An algorithm does not contain any data of its own, and though it certainly has state while the software is running, it has no need preserve its state after its execution is complete.

The glue that allows us to attach data structures to algorithms are iterator classes. The interaction between data structures, algorithms, and iterators in our code is very similar to the way it is done in the Standard Template Library (STL) [63, 77].

We will discuss data structure, algorithm, and iterator classes shortly, but first we need to discuss the abstract base class of all data structure and algorithm classes: the `SpindleBaseClass`.

4.2.1 Base Class and Meta Data

At the highest level of the inheritance hierarchy is an abstract base class called `SpindleBaseClass`. Being the parent class of both the `Algorithm` and `DataStructure` classes, this class has no real

[6] `http://www.zib.de/Visual/software/doc++`

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
class ClassMetaData
{
  public:
    const char * className ;
    const size_t classSize ;
    int nCreated ;
    int nDestroyed ;
    ClassMetaData * parentMetaData ;
    SpindleBaseClass * ( *pfn_CreateInstance() ) ;
    bool isDerivedFrom( const ClassMetaData * p ) const ;
};
```

FIG. 20. *Interface to the `ClassMetaData` class.*

equivalent in the physical world. Its use is limited to providing services for purely software related reasons: instance counting, unique identification number service, access to class meta data[7], providing a safe runtime cast down the inheritance hierarchy[8], instantiation by name, and other functionality to be used later to support object persistence. Some of these features have been outdated by recent additions to the C++ standard, but since compiler acceptance is slow and piecemeal we have not been eager to eject these features entirely from the code.

We begin with the interface to the `ClassMetaData` class in Figure 20. Exactly one instance of this class exists for each child of `SpindleBaseClass`. In turn, `SpindleBaseClass` has a virtual function that returns a pointer to its `ClassMetaData` (see Figure 21). Therefore, for any child of `SpindleBaseClass` — without knowing its exact type — we can access a string representation of its exact type, the size of the concrete type, the number of instances of that type created and destroyed, access to its parent meta data, and access to the default constructor for that type.

The `SpindleBaseClass` provides a minimal interface that generates unique numbers for each instance, standardizes error information[9], provides meta data, dumps state information (to be used only as last resort), provides safe dynamic casting, and allows empty instances to be created on the fly.

---

[7]This was prior to standard C++ defining Run Time Type Infomation (RTTI).
[8]This was prior to standard C++ defining `dynamic_cast<T>`.
[9]This is an experimental feature.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
class SpindleBaseClass
{
  protected:
    const int idNumber ;
    SpindleError * error;
  public:
    int queryIDNumber() const;
    const SpindleError * queryError() const;
    virtual const ClassMetaData * getMetaData() const ;
    virtual void dump( FILE * fp ) const ;
    static SpindleBaseClass * dynamicCast( SpindleBaseClass * object ) ;
    static const SpindleBaseClass *
      dynamicCast( const SpindleBaseClass * object ) ;
    static SpindleBaseClass * createInstance();
};
```

FIG. 21. *Interface to the* `SpindleBaseClass` *class.*

The last method in Figure 21 is not so interesting by itself. But, when it is accessed through the `ClassMetaData::pfn_createInstance` and a registry class indexes all instances of `Class-MetaData` by string name, we can effectively instantiate by name at run time. This infrastructure was designed with the intent of moving to persistent objects in parallel computing. Unfortunately, we have not had the opportunity to actively exploit this capability in a parallel setting.

4.2.2 Data Structures and Object Persistence

The data structures implemented in Spindle are limited to bucket sorters, forests (collections of trees), graphs, heaps, maps, and matrices. Some are not particularly complex, less than 500 lines of code. Others are exceedingly complex; the `QuotientGraph` class is well over 2500 lines of code. All of them inherit (directly or indirectly) from the `DataStructure` class.

In this section, we will explain the features of the `DataStructure` class, list all the classes that inherit from it, show the sourcecode for a small driver that exercises the functionality inherited from `SpindleBaseClass`, and a program that demonstrates Spindle's persistence mechanism.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
class DataStructure :  public SpindleBaseClass
{
  protected:
    enum { INVALID, EMPTY, UNKNOWN, VALID } currentState ;

  public:
    virtual void reset() ;
    virtual void validate() ;
    bool isValid() const { return currentState == VALID ; }
    static SpindleBaseClass * createInstance() const ;
    virtual void loadObject( SpindleArchive& ar ) ;
    virtual void storeObject( SpindleArchive& ar ) const ;
};
```

FIG. 22. *Interface to the* `DataStructure` *class.*

4.2.2.a The `DataStructure` Class

We show the interface to the `DataStructure` class in Figure 22. This class provides two major services to all its descendants: it completes the services needed to implement object persistence, and it defines a four-state scheme and transitions between those states that all data structures obey. Table 4 shows the transition table between states.

Spindle's data structures are built on the premise of split-phase construction. It does not assume that a data structure (or any class for that matter) is ready for use once the constructor is completed. Instead, it allows the data structure to be incrementally defined. Users can set the size of a `Graph`, then provide the adjacency lists, then add the vertex weights, etc.

The second phase of this construction is a call to `validate()`. Exactly what constitutes a valid state depends on the data structure itself. The `PermutationMap` class for instance, need only verify that all the integers from 0 to $n-1$ are represented in a permutation vector of length n. Our `Graph` needs to check that the adjacency list of each vertex is sorted and that there are no duplicated edges.

Once a data structure is in a known valid state, that state is intended to be maintained throughout the rest of the execution. Spindle provides a large number of `const` member functions that access the data structure without changing its state. Many non-`const` functions will preserve the known valid state of the data structure as long as no error was encountered.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE 4

Transition table for the `DataStruct` class

Method Invoked	Initial State	Final State	Comment
default constructor[a]	N/A	EMPTY	
constructor with arguments[b]	N/A	VALID	if method succeeds
		INVALID	if method fails
reset()	EMPTY	EMPTY	
	UNKNOWN	EMPTY	
	VALID	EMPTY	
	INVALID	EMPTY	
setX(X& x)[c]	EMPTY	UNKNOWN	if method succeeds
		INVALID	if method fails
	UNKNOWN	UNKNOWN	if method succeeds
		INVALID	if method fails
	VALID	UNKNOWN	if method succeeds
		INVALID	if method fails
	INVALID	INVALID	Error. Must reset() from INVALID
validate()	EMPTY	EMPTY	Error. Nothing to validate.
	UNKNOWN	VALID	if method succeeds
		INVALID	if method fails
	VALID	VALID	Error. Already known VALID
	INVALID	INVALID	Error. Must reset() from INVALID
other non-const methods	EMPTY	UNKNOWN	
	UNKNOWN	UNKNOWN	
	VALID	UNKNOWN	
	INVALID	INVALID	Error. Must reset() from INVALID
other const methods[d]	EMPTY	EMPTY	
	UNKNOWN	UNKNOWN	
	VALID	VALID	
	INVALID	INVALID	Error. Must reset() from INVALID

[a] Invokes reset().
[b] Typically provided for convenience. Invokes reset(), set . . . () for each argument, and then validate().
[c] Where "X" is the type of the argument begin supplied.
[d] Remember, const member functions should not change the state of the object.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.2.2.b Descendants of DataStructure

Here we list, briefly classes that either descend from DataStructure or are otherwise strongly related in some way. The list of classes is grouped according to a functionality.

- **Maps.** Generally mapping from the set of integers $[0 \ldots n]$ to another set of integers $[0 \ldots m]$. All maps provide two representations, one for the forward mapping and one for the reverse mapping.
  - PermutationMap. Encapsulates a one-to-one and onto mapping. Provides old2new and/or new2old.
  - CompressionMap. Encapsulates an onto, but not one-to-one mapping. Provides forward and reverse mappings fine2coarse and coarse2fine.
  - ScatterMap Encapsulates a one-to-one, but not onto mapping. Provides loc2glob and/or glob2loc (local to global and global to local).
- **Forests** — A collection of 1 or more trees.
  - GenericForest. Provides a generic collection of trees. Can be created from simple array of parent pointers. Allows grafting and pruning of trees. Provides a generic iterator as well as preorder and postorder iterators that are full STL input_iterators.
  - EliminationForest. Inherits from GenericForest and implements a data structure commonly used in direct methods. It can be created from a Graph or a (Graph, PermutationMap) pair.
- **Graph classes**
  - GraphBase. More like a parent struct than a parent class, this defines a generic layout for adjacency lists in all graphs. Base classes like this are used to break cyclic dependencies between different classes.
  - Graph. Simple adjacency graph. May or may not have vertex weights and edge weights. Provides const random access iterators over the adjacency list of each vertex. Inherits from DataStructure and GraphBase.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- `QuotientGraph`. Custom data structure for generating fill reducing orderings.

- **Matrix classes**
  - `MatrixBase`. More like a parent `struct` than a parent class, this defines a generic layout for sparse matrices. Base classes like this are used to break cyclic dependencies between different classes.
  - `Matrix`. General sparse matrix. Has very little functionality since Spindle is essentially a library of graph algorithms. Inherits from `DataStructure` and `MatrixBase`.

- **Misc**
  - `BucketSorter`. Sorts m items by n buckets, where $m > n$. Augmented to support removal of any item in any bucket by name in $\mathcal{O}(1)$ time.
  - `BinaryHeap`. Augmented to locate any item in the heap by name in $\mathcal{O}(1)$ time.

- **Utils** — These classes are a actually just a collection of static methods. They allow us to group related functions, and keep from weighing down data structure classes unnecessarily. Utils do not inherit from `DataStructure` because they have no state of their own and cannot even be instantiated.
  - `MapUtils`. A collection of static member functions to do map manipulations such as: creating an uncompressed permutation from a `PermutationMap` on a compressed `Graph`.
  - `MatrixUtils`. A collection of Matrix manipulation utilities such as: `isSymmetric`, `isTriangular`, `makeDiagonalsExplicit`, `makeTranspose`, etc.

It is interesting to note that even something as conceptually simple as a sparse matrix is implemented as a collection of classes; in this case, six (`SpindleBaseClass`, `MetaData`, `DataStructure`, `MatrixBase`, `Matrix`, and `MatrixUtils`) not including iterators. This was not how the software was designed originally. It grew into this structure for very specific reasons. The reasoning behind the separation between `Matrix` and `MatrixBase` is discussed in Section 4.3.1. The reason for the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

separation between Matrix and MatrixUtils was that the latter is rarely used. We put it in a separate class to cut down on complexity and feature bloat.

Utility classes like this are especially useful when the operation bridges two or more data structures that otherwise should be independent of each other. For instance, the MapUtils class above can take a PermutationMap of a compressed graph and its associated CompressionMap to compute a permutation on the larger uncompressed graph. Logically, we could have made this functionality a method of either map class, but then that class would be dependent on the other. Instead we created a separate utilities class to handle map transformations.

4.2.2.c Example: Using a PermutationMap

In Figure 23 we show a program that exercises some of the functionality that the PermutationMap class inherits from SpindleBaseClass. This simple program does not perform any particularly useful function other than to provide a starting point for other programs.

One item we have not described yet is the SpindleSystem class (line 2). This is a complicated class that performs useful functions for the computer scientist. It directs where trace, warning, and error messages are directed. It also manages a registry of all objects derived from SpindleBaseClass and their meta data. *Spindle* is designed to allow only one instance of the registry to be created and it is guaranteed to be initialized in any compilation unit in which SpindleSystem is included. The mechanism for this is similar to the way that cin and cout are instances of istream and ostream, and are instantiated wherever iostream.h is included in the source code [60, 61]. In line 10, we direct the registry to print out all ClassMetaData classes to standard out just before its destruction.

Moving on with our explanation of Figure 23, we create a PermutationMap in lines 11–17. We get the meta data from the registry in line 18, even though we could just as easily get it directly from the class instance. In line 22, we create a second PermutationMap instance, though it is never validated.

At the end of the code, when the SpindleSystem is being cleaned up, it destroys the registry which, in turn, prints out the ClassMetaData for the SpindleBaseClass, DataStructure,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and PermutationMap classes, their respective sizes, the number of instances created and destroyed, and the name of their parent classes.

4.2.2.d Example: PermutationMap as a Persistent Object

Far more interesting than showing how to access the meta data of an object is to demonstrate its persistence. In Figures 24 and 25, we have two programs. The first program creates a permutation, displays it to the screen, and then packs it into a SpindleArchive, which in this case saves it to a file. The second program opens the file and attaches an SpindleArchive class to it. Then we extract some class from the archive through a pointer to SpindleBaseClass. The second program safely casts the pointer back to its concrete type, prints out the internal state of the permutation, and then prints out the contents of the class registry. The outputs of the two programs should be identical.

In principle, the archive could be attached to any type of data stream: a pipe to another process, a TCP/IP socket, an MPI communicator, etc. This framework allows us to construct instances without knowing their exact type *a priori*. It also allows easy migration of data structures in and out of processes.

We intended to use this extensively as we grew into parallel applications in the future. Unfortunately, we have not had enough opportunity to explore this fully at this time. Furthermore, aspects of this framework have been usurped by new features in the C++ standard as well as component technologies such as EJB, DCOM, and CORBA. Whether or not this functionality should be maintained through later revisions is unclear at this time.

4.2.3 Algorithms as Engines

Typically new object-oriented programmers convert their data structures into classes and bundle all of the associated functions into class methods. In principle this is a good start, but it is not a blanket solution for all cases. We have stumbled across many a debate as to whether the ordering algorithm belongs as a method of the Matrix class or the PermutationMap class. In fact, it belongs in neither.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
 1.  #include <iostream>
 2.  #include "spindle/SpindleSystem.h"
 3.  #include "spindle/PermutationMap.h"

 4.  using namespace std;
 5.  #ifdef SPINDLE_USE_NAMESPACES
 6.  using namespace SPINDLE_NAMESPACE;
 7.  #endif

 8.  #define REGISTRY SpindleSystem::registry()

 9.  main() {
10.    REGISTRY.setDumpOnDestroy( stdout );

11.    const int perm_array[] = {3, 4, 2, 0, 1};
12.    PermutationMap perm( 5 );
13.    perm.getOld2New().import( perm_array, 5 );
14.    perm.validate();
15.    if ( ! perm.isValid() ) {
16.       cerr << "Permutation not valid!" << endl;
17.       exit( -1 );
       }

18.    const ClassMetaData* metaData = REGISTRY.findClass("PermutationMap");

19.    cout << "Size of PermutationMap class    = "
            << sizeof(PermutationMap) << endl;
20.    cout << "According to the registry it is = "
            << metaData->classSize << endl;
21.    cout << "This perm object is serial #    = "
            << perm.queryIDNumber() << endl;

22.    PermutationMap perm2( 8 );
23.    cout << "perm2 is " << ( ( perm2.isValid() ) ? "" : "not " )
            << "valid." << endl;
24.    cout << "The second PermutationMap object has serial #"
            << perm2.queryIDNumber() << endl;

     }
```

FIG. 23. *Example of* `PermutationMap`

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
 1.  #include <iostream>
 2.  #include "spindle/SpindleSystem.h"
 3.  #include "spindle/PermutationMap.h"
 4.  #include "spindle/SpindleArchive.h"
 5.  #include "spindle/SpindleFile.h"

 6.  using namespace std ;
 7.  #ifdef SPINDLE_USE_NAMESPACES
 8.  using namespace SPINDLE_NAMESPACE;
 7.  #endif

 8.  const int perm_array[] = { 3, 4, 2, 0, 1 };

 9.  main() {

       // create a new permutation
12.    PermutationMap * perm = new PermutationMap(5);
13.    perm->getNew2Old().import( perm_array, 5 );
14.    perm->validate();
15.    if ( ! perm->isValid() ) {
16.      cerr << "Error validating original perm." << endl;
17.      exit( -1 );
       }

18.    perm->dump( stdout ); // display its contents

19.    SpindleFile outputFile( "temp.out", "w"); // create a file
20.    SpindleArchive archive( &outputFile,      // create an archive
               SpindleArchive::store | SpindleArchive::noByteSwap );

21.    archive << perm ;  // set the object into the archive.

22.    delete perm ;      // delete the original object
     }
```

FIG. 24. *Object persistence of* `PermutationMap`*, part I. This piece of software produces an instance of a* `PermutationMap` *class, and puts it in an archive.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
 1.  #include <iostream>
 2.  #include "spindle/SpindleSystem.h"
 3.  #include "spindle/PermutationMap.h"
 4.  #include "spindle/SpindleArchive.h"
 5.  #include "spindle/SpindleFile.h"

 6.  using namespace std ;
 7.  #ifdef SPINDLE_USE_NAMESPACES
 8.  using namespace SPINDLE_NAMESPACE;
 7.  #endif

 8.  const int perm_array[] = { 3, 4, 2, 0, 1 };

 9.  main() {
10.    SpindleBaseClass * object = 0 ;
11.    SpindleFile inputFile( "temp.out", "r"); // create file
12.    SpindleArchive archive( &inputFile,      // create archive
          SpindleArchive::load | SpindleArchive::noByteSwap );

13.    archive >> object; // extract an object from the archive

       // check if we retrieved the object
14.    if ( object == 0 ) {
15.      cerr << "Did not retrieve object." << endl ;
16.      exit( -1 );
       }

       // check if it can be safely cast as a PermutationMap
17.    PermutationMap* perm =  PermutationMap::dynamicCast( object );
18.    if ( perm == 0 ) {
19.      cerr << "Dynamic cast failed." << endl ;
20.      exit( -1 );
       }

       // check if the content of the PermutationMap is valid.
21.    perm->validate();
22.    if ( !perm->isValid() ) {
23.      cerr << "Validation of new perm failed." << endl ;
24.      exit( -1 );
       }

25.    perm->dump( stdout ) ; // print out contents,
       // (should be same as first run)
26.    delete perm;
     }
```

FIG. 25. *Object persistence of* `PermutationMap`*: part 2. This program finds the archive file created by the previous program, extracts the objects contained therein, identifies it, and uses it.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
class SpindleAlgorithm :  public SpindleBaseClass
{
  protected:
    enum { INVALID, EMPTY, READY, DONE } currentState ;

  public:
    virtual void reset() ;
    virtual void execute() ;
    bool isValid() const { return currentState != INVALID ; }
    bool isEmpty() const { return currentState == EMPTY ; }
    bool isReady() const { return currentState == READY ; }
    bool isDone() const { return currentState == DONE ; }
};
```

FIG. 26. *Interface to the* `SpindleAlgorithm` *class.*

There are many ordering algorithms, many of them very complex. It is one thing to group simple transformations, such as matrix transpose, into a utilities class; quite another for a complicated algorithm like Sloan or minimum degree.

These algorithms have states of their own. They can be executed several times with varying parameters over the same data. There can be all kinds of special cases and complexity that can be hidden from the user inside an intelligent class. There can also be methods to open up all kinds of details and nuances that only advanced users might be interested in.

Complicated algorithms are classes in their own right. We will soon see that just as some data structures are implemented by combining several classes, so to are our algorithm classes.

4.2.3.a The `SpindleAlgorithm` Class

The `SpindleAlgorithm` class is an ancestor of all heavy-duty algorithm classes in *Spindle*. Like the `DataStructure` class, the `SpindleAlgorithm` class defines a four-state scheme that all algorithms adhere to, although their semantics are slightly different.

The four states defined by `SpindleAlgorithm` are: EMPTY, READY, DONE, and `INVALID`. Data structures allow a split-phase setup and then work very hard to insure they remain in a valid state. Algorithms also allow incremental specifications until they have sufficient inputs to generate an output. Even

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE 5

Transition table for the `SpindleAlgorithm` class

Method Invoked	Initial State	Final State	Comment
default constructor[a]	N/A	EMPTY	
other constructor[b]	N/A	EMPTY	if not enough to execute
		READY	if enough info to execute
		INVALID	if method fails
reset()	EMPTY	EMPTY	
	READY	EMPTY	
	DONE	EMPTY	
	INVALID	EMPTY	
setX(X& x)[c] enableX()	EMPTY	EMPTY	if still not enough to execute
		READY	if now has enough to execute
		INVALID	if method fails
	READY	READY	if method succeeds
		INVALID	if method fails
	DONE	READY	if method succeeds
		INVALID	if method fails
	INVALID	INVALID	Error. Must reset() from INVALID
execute()	EMPTY	EMPTY	Error. No data to execute on.
	READY	VALID	if algorithm succeeds
		INVALID	if algorithm fails
	DONE	DONE	Error. Already known VALID
	INVALID	INVALID	Error. Must reset() from INVALID
other non-const methods	EMPTY	EMPTY	if still not enough to execute
		READY	if now has enough to execute
		INVALID	if method fails
	READY	READY	
	DONE	READY	if can execute again
		DONE	if cannot execute again
		INVALID	if method fails
	INVALID	INVALID	Error. Must reset() from INVALID
other const methods[d]	EMPTY	EMPTY	
	READY	READY	
	DONE	DONE	
	INVALID	INVALID	Error. Must reset() from INVALID

[a] Invokes reset().
[b] Typically provided for convenience. Invokes reset(), and set... () for each argument.
[c] Where "X" is the type of the argument begin supplied.
[d] Remember, const member functions should not change the state of the object.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

when they can generate an output, it does not necessarily mean that they should. There might, after all, be additional specifications or details to be set and we do not want to restrict the user to have to enter them in any particular order. When the user is ready, a call to `execute()` will signal the `SpindleAlgorithm` to execute.

If the run is a success, then it is up to the user to ask for whatever output the algorithm is prepared to generate. After running a MMD ordering, one user may ask for a `PermutationMap` object, another may have use for the `EliminationForest`, and yet another may be interested in diagnostic statistics. As long as the algorithm is in its `DONE` state, all these outputs are available upon request. Or a user may set a new parameter thereby restoring the algorithm to the `READY` state. At this point, the algorithm can be executed on the modified input.

In short, these algorithm classes become computational engines or services. The ability to construct an empty instance, set parameters and inputs individually, execute, and examine the results upon request, are very similar in flavor to event driven graphical user interfaces and even CGI driven web pages. In these web pages, the user typically requests an empty page, makes selections, types entries, sets input, then presses a button typically labeled "Submit". All the inputs are then packaged and sent to the server which parses them, performs some operation, and then presents the user with some feedback as to whether the transaction was successful or not, and whether the user would like to examine the results in any greater detail.

The concept of algorithms as engines that provide services is a point of view echoed by the component based software community. We did not happen across component based technology until very recently. Nevertheless, we feel that the similarities are not purely coincidental.

4.2.3.b Descendants of `SpindleAlgorithm`

As *Spindle* is a library of ordering algorithms, it is not surprising to find that most of the largest and most complicated classes are the ordering algorithm classes. There are also supporting classes that perform tasks required by the ordering algorithms. We list all the descendants of `SpindleAlgorithm` below:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- **Support**. Though not really ordering algorithms themselves, these classes provide important services to the ordering algorithms.
  - BreadthFirstSearch. Given a graph and a starting point, this algorithm traverses the graph by generating level structures. This implementation can abort the algorithm if the width of level structure is beyond the user specified tolerance (implements short-circuiting strategy for PseudoDiameter). This class allows specification of multiple root nodes should the user prefer a breadth-first-search from a set of vertices. Alternatively, this implementation also can be restricted to consider only vertices in the same subdomain of a partitioned graph.
  - GraphCompressor. Given an adjacency graph, this class will compute the number of vertices in the compressed graph. Then, if the user requests, it will generate a compressed, vertex weighted Graph class and CompressionMap from the uncompressed vertices to the compressed vertices. This class is used by the OrderingAlgorithm class.
  - PseudoDiameter. This class implements the algorithm discussed in Section 3.1. Must be run for each connected component of the graph. It can be set to observe domain restrictions on a partitioned graph. It also relies on a separate class derived from ShinkingStrategy to restrict the number of candidates examined. We explain this in detail in Section 4.3.2.
  - SymbolicFactorization. This descendant of SpindleAlgorithm implements the first half of the factoring operation in Figure 2. Given a Graph or a Graph and either a PermutationMap or an EliminationForest, this class computes the EliminationForest (if not supplied) and then generates a PermutationMap that is a proper post order of the EliminationForest. It then calculates the amount of storage required to store the factor as well as the amount of work required to compute it.
- **Ordering Algorithms.**
  - OrderingAlgorithm. Derived from SpindleAlgorithm, this class is the parent of all the other ordering classes. It provides a uniform interface for all ordering algorithms. It also handles precompression of the input graph and mapping the permutation of the compressed

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

graph back to the uncompressed graph. Therefore, all derived classes need only concentrate on the compressed graph. Users can control when graph compression is actually used by setting a tolerance for the ratio of vertices in the compressed graph to vertices in the original graph. When this ratio is close to one, there may be little or no benefit to actually creating a second graph instance.

- `RCMEngine`. Derived from `OrderingAlgorithm`, this class will automatically run the RCM algorithm on each connected component of the input graph. It can also be set to restrict the ordering within subdomains of a partitioned system.

- `SloanEngine`. Also derived from `OrderingAlgorithm`, this class will automatically compute a Sloan ordering on each connected component of the input graph. By default, it will perform two orderings, one with "Class 1" and one with "Class 2" weighting strategies (see Section 3.2.5.d), and automatically choose the best result for each connected component. Alternatively, the user can specify what class of weights should be used, whether they be normalized or not, individual weights, or even the global function to employ other than the one provided; the distance from the endpoint.

  Like RCM, it can also be set to restrict the ordering within subdomains of a partitioned system. This algorithm class can also be set to refine an existing ordering whether it be an explicit permutation or the implicit ordering of the input graph. It also allows the user to explicitly set the start node, end node, or both in lieu of the `PseudoDiameter` class. Furthermore, whole sets of start and/or end nodes can be specified. There is also a mechanism to ensure that end vertices are guaranteed to be numbered last.

- `MinPriorityEngine`. This class inherits from `OrderingAlgorithm` and implements a collection of greedy, fill-reducing ordering algorithms. It allows single or multiple elimination with exact updates, or single elimination with approximate updates. As these algorithms tend to be sensitive to input orderings, this class allows the user to randomize the graph before performing the ordering, or set a specific permutation. It sets a tolerance for when heavily connected vertices are removed and numbered last, even before the ordering begins. It can even

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

recreate a bug in GENMMD where vertices adjacent to an enode have their degree artificially inflated by one (not recommended).

4.2.3.c Algorithms in Action

Figure 27 is a minimal driver for the different ordering algorithms in *Spindle*. Lines 1–3 simply lay out the arrays that will be used to create our `Graph` class (line 11).

The `OptionDatabase` class on line 5 is a convenient class to handle configuration details. It defines an "option" as a flag and zero or more associated arguments. The flag and arguments are all stored as strings, but the `OptionDatabase` provides methods that safely extract ints, floats and the like from the string representation. In line 6, we pass it the command line arguments, but we skip `argv[0]` since its always the name of the program being executed and does not provide any useful information to the class.

The `OptionDatabase` class also can be queried if options exist. We can see from line 6 of Figure 27 that the query can be a string of logically or'ed possibilities. Lines 7–10 simply print out useage information if its queried, or if no minimize option is specified.

In lines 11–13, we create a graph class to perform an ordering on and insure it is valid. In lines 14–20 we create an instance of an ordering algorithm based on the user's commandline option and pass the newly created object to a pointer to the parent class `OrderingAlgorithm`. If the user did not provide a valid argument on the command line, lines 20–22 print an error message and exit.

The rest of the code in Figure 27 is almost self explanatory. We give the ordering algorithm a graph to operate on (line 24), run the algorithm (line 26), gain access to the resulting permutation (line 28) and print its contents to the screen.

4.2.4 Iterators

Separating the data structures from algorithms is extremely beneficial in creating modular code. Creating a usable interface whereby modules can interact efficiently is difficult. The best compromise we found was to use iterator classes. The algorithms assume that iterators have a certain interface for traversing a set of data, and the data structures implement iterators specific to their implementation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    // includes go here
    #include ...

    //
 1. const int nvtxs = 9 ;
 2. const int adjhead[] = {0,2,5,7,11,14,16,19,21};
 3. const int adjlist[] = {1,3,0,2,4,1,5,1,3,5,7,2,4,8,3,7,4,6,8,5,7};

 4. int main( int argc, const char * argv[] ) {

 5.   OptionDatabase db;
 6.   db.loadCommandLineOptions( argc - 1 , argv + 1 );
 7.   if ( db.hasOption( "h|help|?" ) || ( !db.hasOption("minimize") ) ) {
 8.     cout << "Usage :" << argv[ 0 ]
             << " -minimize [ fill | env |  wf | bw ]" << endl ;
 9.     exit( 0 );
      }

10.   Graph mesh3x3( 9, adjhead, adjlist );
11.   mesh3x3.validate();
12.   assert( mesh3x3.isValid() );

13.   OrderingAlgorithm * order;
14.   if ( db.hasOption( "minimize", "fill" ) ) {
15.     order = new MinPriorityEngine() ;
16.   } else if ( db.hasOption( "minimize", "env|wf" ) ) {
17.     order = new SloanEngine() ;
18.   } else if ( db.hasOption( "minimize", "bw" ) ) {
19.     order = new RCMEngine() ;
20.   } else {
21.     cout << "Error: Invalid argument to minimize"
             << db.getOption( "minimize" ) << endl;
22.     exit( -1 );
      }
23.   assert( order->isEmpty() );

24.   order->setGraph( & mesh3x3 );
25.   assert( order->isReady() );

26.   order->execute();
27.   assert( order->isDone() );

28.   const PermutationMap * result = order->getPermutation();
29.   assert( result->isValid() );
30.   result->dump( stdout );

31.   delete order;
32.   return 0;
    }
```

FIG. 27. *Example: Algorithms in action. This is a minimal use of the ordering algorithms. They have a lot of options to trigger before invoking* `execute()`.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
bool isAdjacent( const Graph& g, int i, int j ) const
{
  for( int k = g.adjHead[i]; k < g.adjHead[i+1]; ++k ) {
    if ( g.adjList[k] == j ) {
      return true;
    }
  }
  return false;
}
```

FIG. 28. *A C-like function that directly accesses a Graph's data*

It is important to question if iterator classes are really necessary in our case. The algorithms in Spindle operate on a graph or a matrix, they are not as general as algorithms provided in STL [77] like `sort()` which operates on vectors, lists, and deques. On the other hand, we are designing for flexibility. While Spindle in its original incarnation performs orderings on just `Graph` classes, we wanted to allow for more classes later. Indeed we see indications of this happening in Section 6. Although this iterator paradigm was successful in many cases, we also discuss where they were misapplied and dragged performance significantly in Section 4.4.1.

4.2.4.a Definition of an Iterator.

An *iterator* is closely associated with a particular container class, usually a "friend" class, that enables traversing items in the container without revealing its underlying storage mechanism.

Assume, for example, that the list of all edges in a Graph are in the array `adjList[]` and that a second array `adjHead[]` stores the beginning index into `adjList[]` for each vertex in the graph. Then to check if vertex `i` is adjacent to vertex `j`, we could simply run through the arrays as in Figure 28.

This design has several flaws. The function assumes the layout of data in the `Graph` class and accesses it directly. Consider now a different approach where the `Graph` class creates an iterator. Conventionally, the iterator classes mimic the functionality of a pointer accessing an array. The dereference operator, `operator*()`, is overloaded to access the current item in the container, and the increment operator, `operator++()`), advances the iterator to the next item in the container. Rewriting our function in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
bool isAdjacent( const Graph& g, int i, int j ) const
{
  for( Graph::adj_iterator it = g.adj_begin(i);
       it != g.adj_end(i); ++it ) {
    if ( *it == j ) {
      return true;
    }
  }
  return false;
}
```

FIG. 29. *A C++ global function using Graph::adj_iterator*

Figure 28, we define `Graph::adj_begin(int)` to create an iterator pointing to the beginning of the adjacency list of a specified vertex and `Graph::adj_end(int)` to return an iterator pointing to the end of the list[10]. This new implementation is presented in Figure 29.

The benefit of the iterator approach is that the function `isAdjacent()` no longer assumes how the data inside `Graph` is laid out. If it is indeed sequential as it was in the previous example, then the iterator could be simply a `typedef`'ed `int*`. However, the adjacency lists could be stored in a red-black tree for faster insertions and deletions. In this case, the example using iterators still applies since it assumes a suitable iterator class is provided.

4.2.4.b Application of Iterators.

Iterators provide a kind of "compile-time" polymorphism. They allow a level of abstraction between the data structure and the algorithm, but the concrete implementation is determined at compile time. This allows the compiler to inline function calls (often through several levels) and get very good performance[11].

There were a few difficulties in applying this technique to our problems. The most complicated aspect was that all STL containers are one-dimensional constructs. Most of our data structures — matrices and graphs — are two dimensional. This was not a serious problem since we, as programmers, tend to

[10] Actually, it points to one past the end—a standard C++ convention.
[11] C++ cannot inline virtual functions.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

linearize things anyway. In the example of iterators before, for instance, we simply used iterator to iterate over the set of vertices adjacent to a particular vertex.

4.3 Features

The features provided in Spindle are a direct result of the design principles we enumerated in Section 2.2. In this section, we give examples of particularly useful features that were implemented for usability, flexibility, and extensibility in the code.

4.3.1 Multiple File Formats: A case for multiple inheritance

An important problem that we often run into is sharing problems with other researchers. Whenever we agree to generate some solutions for a client (either academia or industry) we often find that we must adapt our code to a new file format. There have been attempts to standardize sparse matrix file formats, most notably the Harwell-Boeing Format [23], and the Matrix Market format [12]. However, it is unreasonable to expect clients to restrict themselves to a small choice of formats. Understanding the nature of this problem and applying object-oriented techniques are good exercises in preparation for the harder problems ahead.

The easiest way to handle sparse matrix I/O is to have a matrix class with two member functions: one to write a particular format and another to read that format. This is a simple solution, but it has a scalability problem. First, as the number of formats increase, the number of member functions grows and the matrix class becomes more and more cumbersome. Second, if separate matrix classes are needed then all of the I/O functions must be replicated for each class.

4.3.1.a The Chicken and Egg Problem

One could reasonably create a `Matrix` class and a `MatrixFile` class for each matrix file format. Unfortunately the resulting problem is determining which creates which. One would expect to create a matrix from a file, but it also makes sense to create a file from the matrix. See, for example, the class header files in Figure 30.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
// Matrix.h
#include "MatrixFile.h"

class Matrix
{
  public:
    Matrix( MatrixFile& );
    //...
};
```

```
// MatrixFile.h
#include "Matrix.h"

class MatrixFile
{
  public:
    MatrixFile( Matrix& );
    //...
};
```

FIG. 30. *Example: cyclic dependency between classes.*

Such a design induces a cyclic dependency between the two objects, which is bad. Cyclic dependencies can have a dramatic effect on the cost of maintaining code, especially if there are concrete classes inheriting from the cyclic dependency [54, pg. 224]. This is exactly the case here, since the intention is to abstract away the differences between different file formats.

A solution is to escalate the commonality between the two classes. This has the advantage that the dependencies are made acyclic, the downside is that an additional class is introduced for purely "computer science" reasons that has no physical counterpart. We will call this class `MatrixBase` which is the direct ancestor of both the `Matrix` and `MatrixFile` classes. The latter two are shown in their improved form in Figure 31.

Now we can derive various matrix file formats from `MatrixFile` independent from the internal computer representation of the Matrix. We will show later that the benefits compound when considering matrix to graph and graph to matrix conversions.

```
// Matrix.h (second try)
#include "MatrixBase.h"

class Matrix
   :  public MatrixBase
{
  public:
    Matrix( MatrixBase& );
    //...
};
```

```
// MatrixFile.h (second try)
#include "MatrixBase.h"

class MatrixFile
   :  public MatrixBase
{
  public:
    MatrixFile( MatrixBase& );
    //...
};
```

FIG. 31. *Example: removing cyclic dependencies by ecalation.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.3.1.b Navigating Layers of Abstraction

It is important to understand that abstractions involved around the construct we call a "matrix" come from different levels and have different purposes. To define a class `Matrix` and possibly many subclasses, care must be taken to capture the abstraction correctly. It is hard to give a formula for designing a set of classes to implement an abstract concept. However, when the abstraction is captured just right, using it in the code is natural and intuitive. Good designs often open new possibilities that had not been considered.

For the matrix object, we have identified at least two dimensions of abstraction that are essentially independent, one from the mathematical point of view, one from the computer science point of view.

Along the first dimension, the mathematical one, a matrix can be sparse, dense, banded, triangular, symmetric, rectangular, real or complex, rank deficient or have full rank, etc. From a mathematical point of view, all of these words describe a property of the matrix.

From a computer science point of view, there are different ways that these 2-D constructs are mapped out into computer memory which is iself one dimensional. Primarily, matrices must be set in either row-major or column-major order, though diagonally arranged data has been used in some cases. For sparse matrices, indices can start counting from zero or one. Layout is further complicated by block structures, graph compression, etc.

The critical question is: in all the specifications of matrix listed above, which ones are specializations of a matrix and which ones are attributes? The answer to this question directs which concepts are implemented by subclassing and which are implemented as fields inside the class.

The answer also depends on how the class(es) will be used. Rarely will a programmer find a need to implement separate classes for full rank and rank deficient matrices, but it is also not obvious that a programmer must implement sparse and dense matrices as separate classes either. Matlab uses the same structure for sparse and dense matrices and allows conversion between the two. On the other hand, PETSc has both sparse and dense matrices subclassed from their abstract `Mat` base class.

A third dimension of complexity comes from matrix file formats, which can be either a text or binary file, and more generally, a pipe, socket connection, or other forms of I/O streams. In particular, even if the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

matrix is symmetric, and the storage is explicit, it may still be implicit — meaning only the lower/upper triangle is stored — to conserve disk-space/banwidth/etc.

4.3.1.c Final Layout

Our major concern was to have a flexible extensible system for getting matrices in many different formats in and out of our programs at runtime. We discuss in this section how we finally organized our solution. The inheritance hierarchy of the subsystem is shown in Fig. 32.

First we made non-abstract base classes `GraphBase` and `MatrixBase` which define a general layout for the data. From these, we derive `Graph` and `Matrix` classes which provide the public accessor/mutator functions, each provide constructors from both `GraphBase` and `MatrixBase`. Furthermore, `Graph` and `Matrix` classes also inherit from the `DataStructure` class which gives it generic data structure state, error reporting functionality, and all the other features described in Section 4.2.2. This way, both can construct from each other without the cyclic dependencies.

The final important piece before fitting together the entire puzzle is a `DataStream` class. This abstract base class has no ancestors and does all of its I/O using the C style `FILE` pointers. We chose this C-style of I/O because, although it lacks the type-safety of C++ style `iostream`, it does allow us to do I/O through files, pipes, and sockets. These features have unfortunately not been included in the C++ standard.

If we try to open a file with a "`.gz`" suffix, the file object inherits from the `DataStream` class the functionality to open a `FILE` pointer that is in fact a pipe to the output of gunzip[12]. The `DataStream` class is therefore responsible for opening and closing the file, uncompressing if necessary, opening or closing the pipe, or the socket, etc. but is an abstract class because it doesn't know what to do with the `FILE` once it's initialized. This class also provides the error handling services that are typical with file I/O.

To understand how all these partial classes come together to do I/O for a sparse matrix format, consider adding a new format to the library, a Matrix Market file. To be able to read this format, we create a class

[12] The "`.gz`" suffix indicates a file that is compressed with the GNU zip utility (`gzip`) and can be uncompressed by its complement, `gunzip`.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

MatrixMarketFile which inherits from MatrixBase and DataStream. This new class needs to implement two constructors based on MatrixBase or GraphBase and two virtual functions: read(FILE *) and write(FILE *) (in practice, it also implements many more accessor/modifier methods specific to the Matrix Market format). Now, we can read a Matrix Market file, and create instances of either Graph or Matrix (or any other class that uses a MatrixBase in its constructor). Furthermore, from any class that inherits from MatrixBase or GraphBase we can write the object in Matrix Market format. A graph-based file format, for instance Chaco [44] can be created using a similar inheritance hierarchy based on GraphBase.

4.3.2 Extensibile Pseudo-Diameter

As we saw in Section 3.1.3 even simple algorithms like the pseudo-diameter computation are under constant improvement. Good software needs to be aware of this fact and allow for it to be easily adapted — lest it become obsolete.

To implement the many different shrinking strategies and allow for the subsequent development of new ones, we created a separate homomorphic inheritance hierarchy just to implement shrinking strategies. The abstract base class ShrinkingStrategy, defines two vectors of (vertex, degree) pairs, a function that takes an array of vertices and a const Graph pointer and packs the (vertex,degree) pairs into one of the vectors, and a pure virtual function that copies a subset of (vertex, degree) pairs from the first vector to the second. Concrete classes derived from ShrinkingStrategy, must define exactly how (vertex, degree) pairs from the first vector are chosen for the second.

Spindle's PseudoDiameter class has a pointer to the abstract base class ShrinkingStrategy and has no knowledge which derived class is actually being used. In fact, we were able to implement the more aggressive shrinking strategies of Reid and Scott [72] without modifying any of the existing code.

It turns out that this design is an instance of the Strategy Pattern [30, pg. 315]. We show the relationships between the classes in Figure 33 This important design pattern offers several benefits. It provides a more efficient and more extensible alternative to long chains of conditional statements for selecting

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

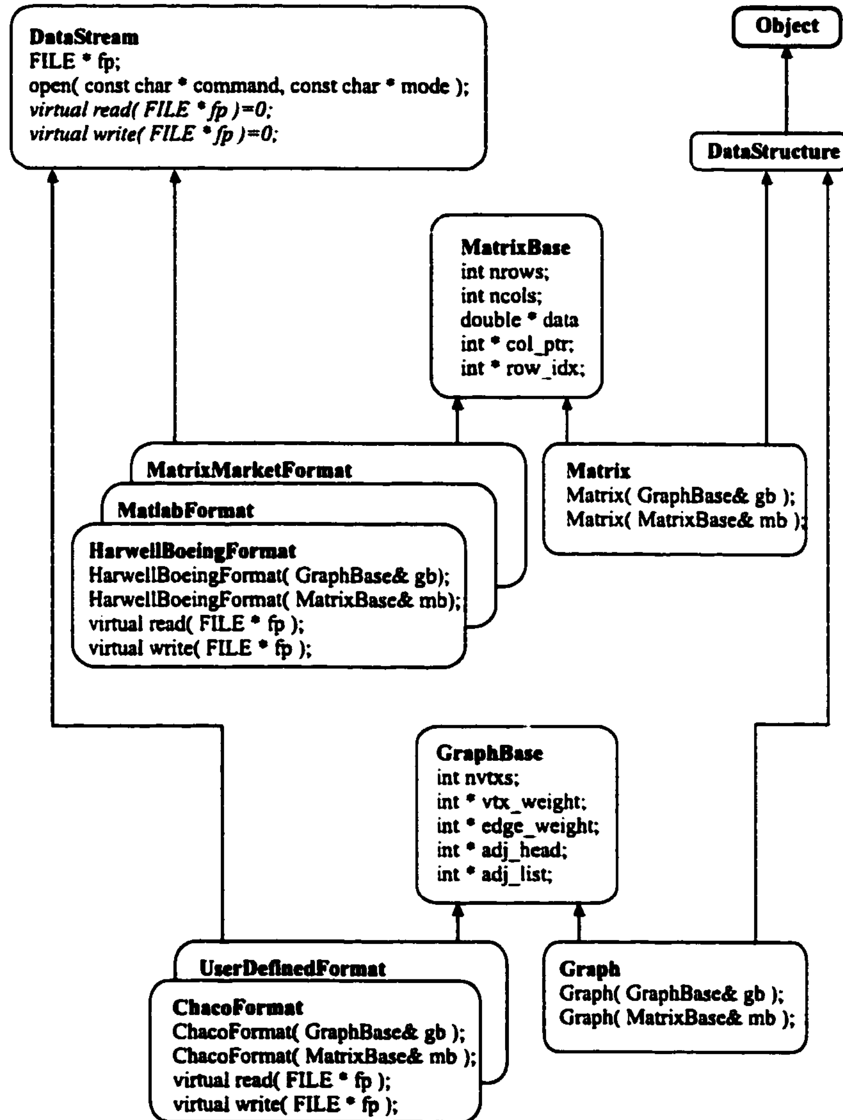


FIG. 32. *Inheritance hierarchy for multiple matrix formats. A fragment of the inheritance hierarchy highlighting how multiple file formats are implemented and extended. To add an additional format, simply inherit from* `DataStream` *and one of* `GraphBase` *or* `MatrixBase`*. Then implement the two pure virtual methods inherited from* `DataStream`*.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

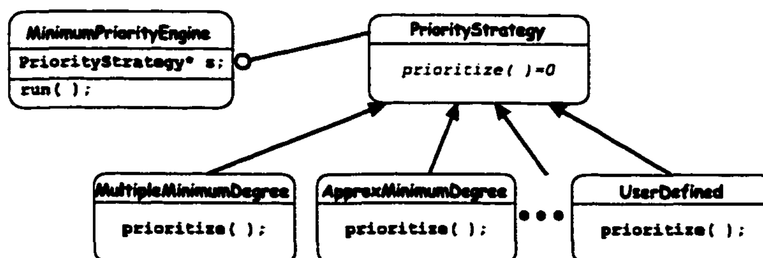


FIG. 33. *The Strategy Pattern. Applied in this case to shrinking strategies for the pseudo-diameter algorithm.*

desired behavior. It allows new strategies to be implemented without changing any code in the `Pseudo-Diameter` class. It is also more attractive than overriding a member function of the `PseudoDiameter` class directly because of the its overall complexity.

4.3.3 Polymorphic Fill-Reducing Orderings

One example where object-oriented implementation had substantial payoffs in terms of extensibility was in our ability to construct polymorphic fill reducing orderings. Recall from Table 3 that there are several different types of greedy algorithms, many of which are quite recent. In fact, there is no known library containing all of these algorithms, besides Spindle. While some of these heuristics are related, others — particularly MMD and AMD — are radically different in the ways that degree is computed, the underlying graph is updated, and what optimizations are allowed and disallowed. Fundamentally, MMD allows lazy update of the quotient graph by allowing multiple vertices to be removed between each update. AMD doesn't require as much work per graph update, but the graph must be updated after every node is eliminated.

We wanted to use the Strategy Pattern [30, pg. 315] again as we did in Section 4.3.2, but this time the design was more complicated. The `PseudoDiameter` class did not know or care about the way the set of candidate vertices was reduced. The `MinPriorityEngine` did not know the formula used to compute the vertices' priority, but it does need to know if the strategy allows the lazy update mechanism of MMD or if it needs a single elimination scheme to compute the tight approximate degrees.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

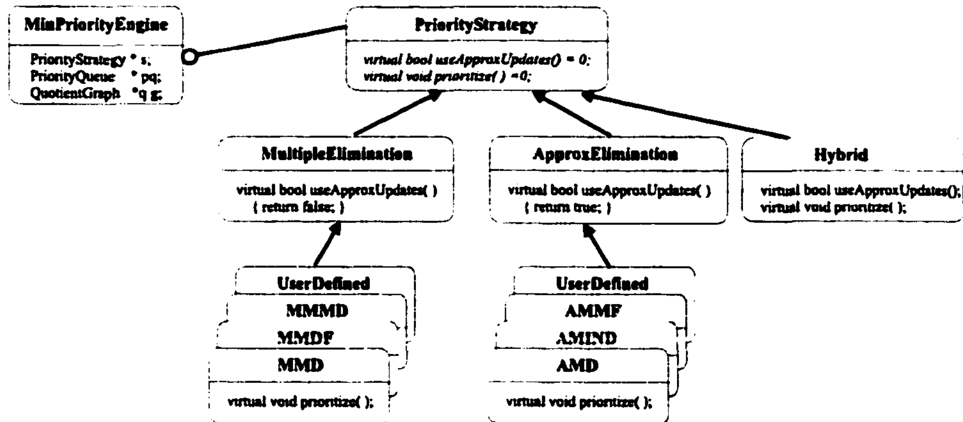


FIG. 34. *An augmented Strategy Pattern. Used to implement a family of greedy, fill reducing algorithms.*

We created a complete framework for the entire family of minimum-degree like algorithms, but it required an additional virtual function. See Figure 34 for the inheritance and composition relationships. In this arrangement, the class `MinimumPriorityEngine` (which we will call `Engine` for short) is an algorithm that is given a graph, repeatedly selects the node of minimum priority, eliminates it from the graph, and then updates the graph adding appropriate fill edges when necessary. The catch is that it has no idea how to determine the priority of the vertices. It must rely on a `PriorityStrategy` class (`Strategy` for short), or more specifically, a specialized descendant of the `strategy`.

We listed the benefits of the Strategy Pattern earlier in Section 4.3.2, but there are potential drawbacks for using this pattern. There is an increased number of classes in the library, one for each ordering algorithm. This is not a major concern, though users should be insulated from this by reasonable defaults being provided. Another possible concern is the communication overhead. The calling interface must be identical for all the strategies, though individual types may not need all the information provided. There is a potential for algorithmic overhead in the decoupling between Engine and Strategy. In our case, the engine could query the strategy once for each vertex that needs to be evaluated, though the virtual-function call overhead would become high. Alternatively, the engine might request all vertices in the remaining graph G_k to be re-prioritized after each node is eliminated. This may result in too much work being done inside the Strategy. Luckily, with all these algorithms the only nodes whose priority changes are the ones

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

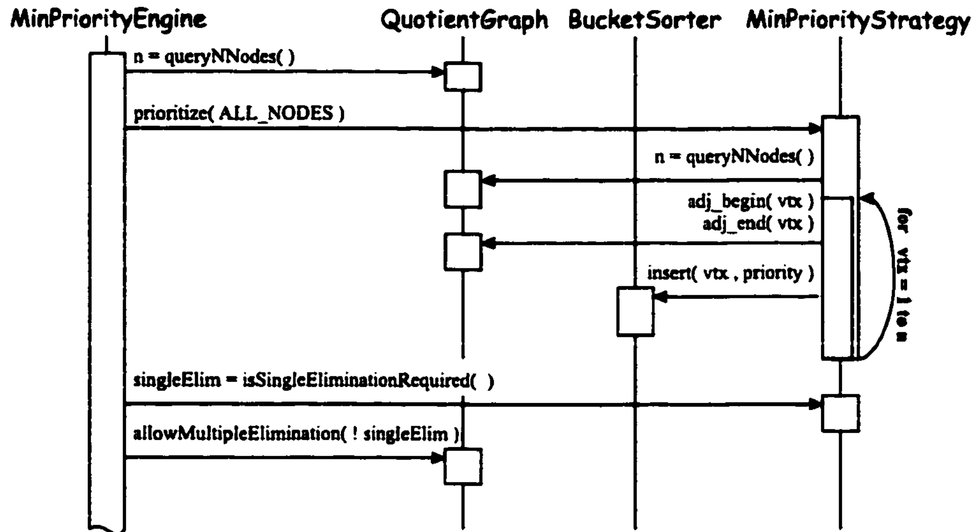


FIG. 35. *Interaction of initializing the ordering.*

adjacent to the most recently eliminated node. The `QuotientGraph` keeps track of this information since it must prevent a node with an unknown (or invalid) priority from being eliminated. In other words, a vertex cannot be eliminated if any of its neighbors have been eliminated since the last quotient graph update.

For the entire framework to implement MMD, a class must be derived from the `Strategy` abstract base class and override the pure virtual member function `computePriority`. The `Engine` is responsible for maintaining the graph and a priority queue of vertices. It selects the vertex of minimum priority, removes it from the queue and eliminates it from the `QuotientGraph`. The priority of all the neighbors of the most recently eliminated node is changed, so they too are removed from the priority queue for the time being. When there are no longer any vertices in the priority queue of the same minimum degree as the first vertex eliminated from this stage, the `Engine` updates the graph, and gives a list of all vertices adjacent to newly eliminated ones to the `MMDStrategy` class. This class, in turn, computes the new priority of the vertices and inserts them into the priority queue.

To make this setup efficient, we use a `BucketSorter` class to implement the priority queue and a `QuotientGraph` class to implement the series of graphs during elimination. The interaction of these four major objects is shown in Fig. 36. We hide the details of how single elimination and multiple

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

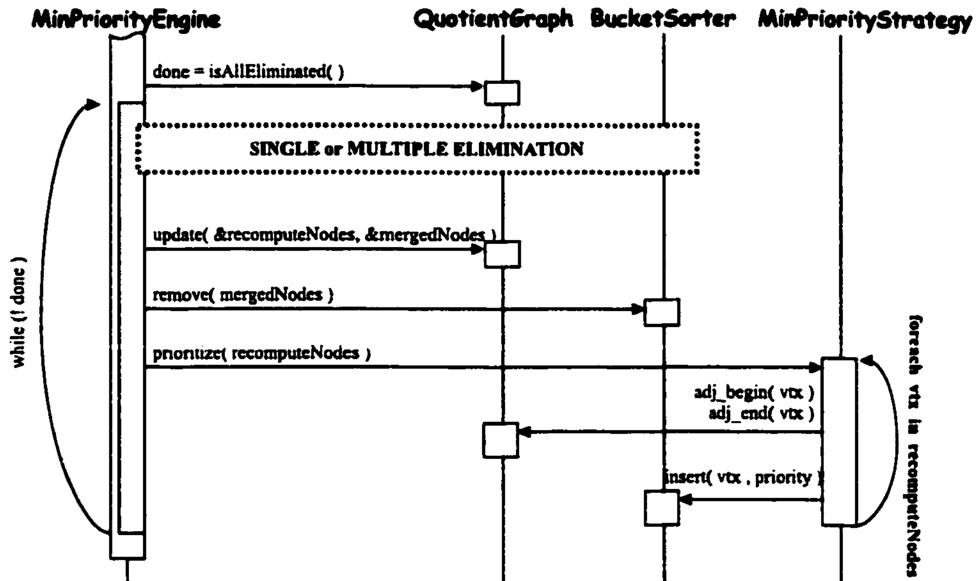


FIG. 36. *Interaction of the ordering.*

elimination are handled. This too is determined by a simple query to the `Strategy` class. When the `QuotientGraph` is updated, it performs various types of compression which may remove additional vertices or modify the list of vertices that need their priority recomputed. When it calls the `Strategy` to compute the priorities, it provides a const reference to the `QuotientGraph` for it to explore the data-structure without fear of side-effects, and the `BucketSorter` to insert the vertices in.

We mention that infomation from the `Strategy` must also propagate to the `QuotientGraph` as it is required to behave in slightly different ways when updating for single elimination ordering algorithms (e.g., AMD) and multiple elimination algorithms (e.g., MMD). Thus the `Engine` must query the `Strategy` what type is required and set the `QuotientGraph` to behave accordingly. This is handled in the first phase of the `execute()` function that is overridden from the `Engine`'s parent class, `SpindleAlgorithm`. The interaction of the objects in this phase is shown in Fig. 35.

4.4 Balancing Features with Performance

There is a very real danger of adding too many features into a piece of software. Just because some feature *can* be added, does not imply that it *should* be added. Excessively feature-laden software can

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

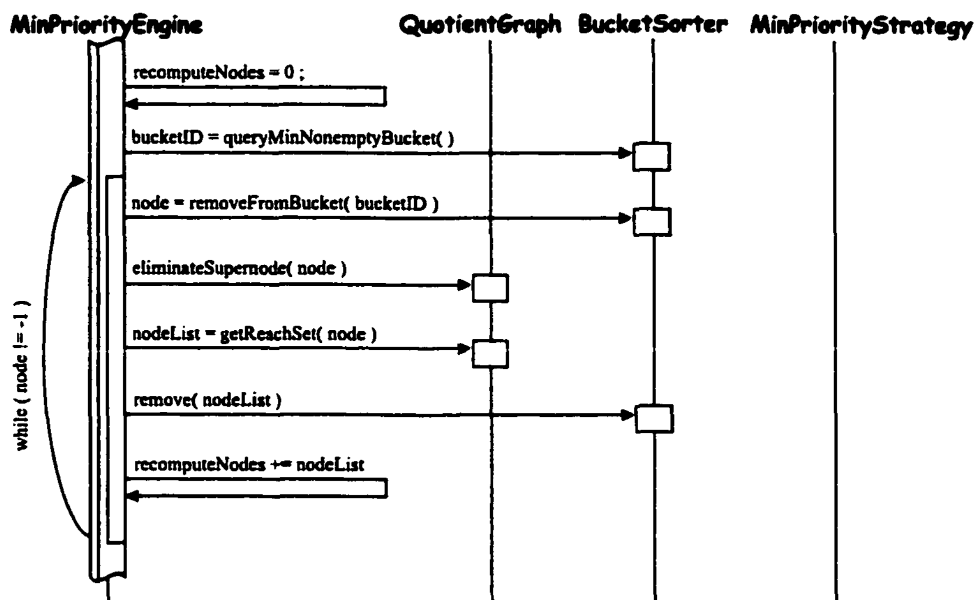


FIG. 37. *Interaction of eliminating vertices during ordering.*

actually inhibit flexibility and extensibility as the components become to large and unwieldy for the user to shape to their specific purposes. Feature-laden software also tends to become excessively complex and increasingly inefficient.

4.4.1 Judicious Application of Iterators

One disappointing endeavor was to provide an iterator class to traverse the *reachable set* of the `QuotientGraph` class. Although we were successful in implementing such a class, its performance was so poor, that its general use was abandoned. This caused an increase in the difficulty of implementing the various minimum priority strategies. Here we explain why this idea looked good on paper, why it didn't work well in practice, and why this problem is unavoidable.

Ideally, one would like to provide a class that iterates over the reachable set so that the priority computation can be implemented cleanly. In the sample code below, this class is `typedef`'ed inside the `QuotientGraph` class as `reach_iterator`. We add the additional detail that there may be a weight associated with each vertex, so degree computation sums the weights of the nodes in the reachable set.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ALGORITHM 6 (Computing degree using the reachable set iterator.)

```
void MinimumDegree::prioritize( const QGraph& g, List& l,
                                PriorityQueue& pq )
{
  for( List::iterator it = l.begin(); it != l.end(); ++it ) {
    int i = *it;
    int degree = 0;
    for( QGraph::reach_iterator j = g.reach_begin(i);
      j != g.reach_end(i); ++j ) {
      degree += g.getNodeWeight( *j )
    }
    pq.insert( degree, i );
  }
}
```

In fact, we were able to implement such an interface. But there is a hidden overhead that causes this implementation to be too expensive. The definition of a reachable set is the union of several non-disjoint subsets. Therefore the iterator must test at each iteration if there are any more items in the current set, if there are any more sets, and some internal mechanism to prevent double visiting the same node in different sets. Referring again to Fig. 16 note that the reachable set need not be traversed in sorted order (as presented here), but it cannot allow the same vertex to be counted twice through two different enodes. Furthermore the reachable set does not include the node itself. The most effective way to prevent "double-visiting" is to maintain an array of length equal to the number of nodes in the graph and flag the appropriate entry when a node is "visited."

The `ReachSetIter` class requires privileged access to the `QuotientGraph` class though friendship. Since the `QuotientGraph` has two adjacency lists per node, the iterator over the reachable set is a bit more complicated. Most of the details are not difficult. However the increment operator becomes excessively tedious. The problem is that the increment operator must re-determine its state at each call ... Is it already at the end? Are there more nodes in the current list? Are there more enodes in the enode list? Once there is a next node located, has it been marked? If so go back to the beginning.

There is a way to evaluate the reachable set manually by iterating over sets of *adjacent* nodes and enodes of the quotient graph manually. This is shown in Algorithm 7 which is functionally equivalent to the code in Algorithm 6.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ALGORITHM 7 (Computing degree without using the reachable set iterators.)

```
void MinimumDegree::prioritize(const QGraph& g, List& l,
                                 PriorityQueue& pq )
{
  for( List::iterator it = l.begin(); it != l.end(); ++it ) {
    int i = *it;
    int degree = 0;
    int my_stamp = nextStamp();        // get new timestamp
    g.visited[ i ] = my_stamp;         // Mark myself visited
    for( QGraph::enode_iterator e = g.enode_begin(i);
      e != g.enode_end(i); ++e ) {
      int enode = *e;                  // for all adjacent enodes
      for( QGraph::node_iterator j = g.node_begin(e);
        j != g.enode_end(e); ++j ) {
        int adj = *j;                  // for all adjacent nodes
        if ( visited[ adj ] ¡ my_stamp ) {
          // if not already visited, mark it and add to degree
          visited[ adj ] = my_stamp;
          degree += g.getNodeWeight( adj );
        }
      }
    }
    pq.insert( degree, i );
  }
}
```

The lesson learned here is to be judicious in the use of fancy techniques. The coding benefits of using a reachable set iterator are far outweighed by the speed increase of manually running through adjacency lists. Note that the latter scheme makes the critical assumption that every node has a "self-edge" to itself in the list of adjacent enodes. This convenient assumption also increases coupling between the `QuotientGraph` class and the descendants of the `MinimumPriorityStrategy` class.

4.5 Summary

There is no conflict between object-oriented design and scientific computing. It is true that some object oriented techniques lead to performance losses but a careful design leads to scientific computing software that is much easier to understand and use. For successful implementations, it is crucial to understand which techniques are appropriate for what circumstances. Neither encapsulation nor inheritance necessarily slows down programs, unless a virtual function is used.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Good design requires tradeoffs. There is no perfect solution unless we are talking about pedagogical problems in books. Real life applications have many constraints, most of the time conflicting.

Decoupling is a perfect example. It can introduce overheads since objects have to communicate through well defined interfaces. On the other hand, decoupling localizes potential code changes and increases flexibility. All our algorithms are aware that our `Graph` class is implemented as a sorted adjacency list[13]. We could have implemented our algorithms to not assume unordered adjacency lists, but this would have impacted the performance of some of our algorithms. The tradeoffs between flexibility and efficiency are determined by the interface.

[13]The adjacency lists of a `QuotientGraph` are an exception, they are not sorted.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5. RESULTS

This section tabulates all the experimental results we obtained by running our software. Results on the envelope/wavefront reduction problem are presented first in Section 5.1, followed by results for the fill reducing heuristics in Section 5.2.

5.1 Envelope/Wavefront Reduction

We describe in Section 5.1.1 how we chose the computational parameters in the hybrid algorithm. In Section 5.1.2 we discuss the relative reductions in envelope size and wavefront of eighteen test problems obtained from RCM, Sloan, spectral, and hybrid algorithms.

5.1.1 Spectral Orderings from Chaco

We use the SymmLQ/RQI option in Chaco [44] to obtain the Fiedler vector, which is sorted to produce the spectral ordering. Chaco takes a multilevel approach, coarsening the grid until it has less than some user specified number of vertices (1000 seems to be sufficient). Then it computes the Fiedler vector on the coarse grid, orthogonalizing only for eigenvectors corresponding to small eigenvalues. Then the coarse grid is refined back to the original grid and the eigenvector is refined using Rayleigh Quotient Iteration (RQI). This refinement is the dominant cost of the whole process. During the coarsening, we compute generalized eigenvectors of the weighted Laplacians of the coarse graphs from the equation $Ax = \lambda Dx$, where D is the diagonal matrix of vertex weights. This feature, obtained by turning on the parameter MAKE_VWGTS, speeds up the eigenvector computation substantially.

Two other parameters, EIGEN_TOLERANCE and COARSE_NLEVEL_RQI, control how accurately eigenvectors are computed and how many levels of graph refinement occur before the approximate eigenvector is refined using RQI, respectively. We set the value of EIGEN_TOLERANCE to 10^{-3}, and it was very effective in reducing cpu-time. Even in the case where this tolerance induces misconvergences, the spectral ordering is still good and the hybrid ordering even better for most problems. The

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE 6

Eighteen test problems for wavefront reduction. For the three problems that compressed well, their compressed versions are also shown.

	Problem	$\|V\|$	$\|E\|$	class	Comment
1	barth	6,691	19,748	1	
2	barth4	6,019	17,473	1	2-D CFD problems
3	barth5	15,606	45,878	1	
4	shuttle_eddy	10,429	46,585	1	
5	copter1	17,222	96,921	1	
6	copter2	55,476	352,238	1	
7	ford1	18,728	41,424	2	3-D structural problems
8	ford2	100,196	222,246	2	
9	skirt	45,361	1,268,228	2	
0	nasasrb	54,870	1,311,227	2	
11	commanche_dual	7,920	11,880	1	
12	tandem_vtx	18,454	117,448	1	3-D CFD problems
13	tandem_dual	84,069	183,212	1	
14	onera_dual	85,567	116,817	1	
15	bcsstk30	28,924	1,007,284	2	3-D stiffness matrix
16	pds10	16,558	66,550	1	
17	finance256	37,376	130,560	2	linear programs
18	finance512	74,752	261,120	2	
19	c_skirt	14,944	160,461	2	compressed skirt
20	c_nasasrb	24,953	275,796	2	compressed nasasrb
21	c_bcsstk30	9,289	111,442	2	compressed bcsstk30

COARSE_NLEVEL_RQI parameter didn't have much effect, so we used the program's default value of 2.

5.1.2 Comparison of Algorithms

We consider five ordering algorithms RCM, Sloan with unnormalized weights $W_1 = 2$, $W_2 = 1$, Sloan with normalized weights ($W_1 = 8$, $W_2 = 1$ for problems in Class 1, and $W_1 = 1$, $W_2 = 2$ for problems in Class 2), spectral, and hybrid (normalized weights $W_1 = W_2 = W_3 = 1$ for Class 1 problems, $W_1 = 1$, $W_2 = W_3 = 2$ for Class 2 problems). When we refer to the Sloan algorithm without mentioning the weights, we mean the algorithm with normalized weights. We have compared the quality and time requirements of these algorithms on eighteen problems (see Table 6). The problems are chosen to represent a variety of application areas: structural analysis, fluid dynamics, and linear programs from stochastic optimization and multicommodity flows. The complete set of results are shown in Tables 7–11. The values for all the orderings are normalized to RCM.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE 7
Mean square Wavefront sizes for various algorithms relative to RCM. The numbers in parentheses after the values for the normalized Sloan algorithm show the class of each problem (See Section 3).

	Problem	mswf RCM	normalized by RCM Sloan	FastSloan	spectral	hybrid
1	barth	1.26e4	.48	.43	.43	.30
2	barth4	1.61e4	.40	.21	.20	.15
3	barth5	5.08e4	.56	.18	.18	.14
4	shuttle	5.84e3	.60	.60	1.0	.65
5	copter1	2.84e5	.71	.45	.74	.53
6	copter2	2.26e6	.39	.27	.28	.16
7	ford1	2.65e4	.67	.67	.48	.39
8	ford2	3.74e5	.51	.51	.44	.33
9	skirt	1.11e6	.57	.50	.44	.37
0	nasasrb	1.65e5	.74	.75	.99	.71
11	commanche_dual	6.73e3	.60	.34	.37	.23
12	tandem_vtx	8.28e5	.16	.12	.14	.10
13	tandem_dual	1.96e6	53	.28	.14	.11
14	onera_dual	4.86e6	.44	.21	.09	.07
15	bcsstk30	1.07e6	.37	.30	.10	.05
16	pds10	3.66e6	.20	.13	.75	.15
17	finance256	9.38e5	.04	.04	.07	.04
18	finance512	5.79e5	.05	.06	.14	.05
19	c_skirt			.46	.51	.39
20	c_nasasrb			.68	1.8	.75
21	c_bcsstk30			.26	.13	.06

Initially we discuss the results on the uncompressed graphs, since most of the graphs in our test collection did not gain much from compression. We discuss later in this section the three problems that exhibited good gains from compression.

The envelope parameters and times reported in the tables are normalized with respect to the values obtained from RCM. For the Sloan algorithm, two sets of values are reported: the first is from the unnormalized weights $W_1 = 2$, $W_2 = 1$, and the second from the normalized weights for Class 1 and Class 2 problems. The normalized Sloan algorithm is labeled by the column FastSloan in Table 7. The results for the compressed problems are indicated by the last three rows.

The Sloan algorithm with the normalized weights reduces the mean-square wavefront on average to 23% of that of RCM; when unnormalized weights are used in the Sloan algorithm, the mean square wavefront is 36% of that of RCM. (Henceforth, a performance figure should be interpreted to be the average value for the problems in the test collection; we shall not state this explicitly.) The hybrid reduces mean-square wavefront to 14% of that of RCM, and to 60% of that of (normalized) Sloan. The hybrid

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE 8
Maximum wavefront sizes relative to the RCM algorithm.

		maxwf	normalized by RCM			
	Problem	RCM	Sloan	FastSloan	spectral	hybrid
1	barth	164	.66	.65	.64	.53
2	barth4	204	.60	.42	.37	.34
3	barth5	351	.77	.44	.42	.39
4	shuttle	167	.85	.66	1.30	.67
5	copter1	797	.84	.58	.65	.57
6	copter2	2,447	.58	.49	.43	.32
7	ford1	223	.86	.86	.96	.78
8	ford2	884	.74	.78	.91	.76
9	skirt	1,745	.65	.84	.65	.57
0	nasasrb	840	.73	.91	1.20	.86
11	commanche_dual	150	.83	.55	.55	.44
12	tandem_vtx	1,489	.38	.30	.29	.25
13	tandem_dual	2,008	.72	.55	.34	.30
14	onera_dual	3,096	.67	.45	.34	.30
15	bcsstk30	1,734	.63	.64	.38	.22
16	pds10	2,996	.48	.40	1.00	.28
17	finance256	1,437	.22	.22	.30	.21
18	finance512	879	.28	.32	.85	.49
19	c_skirt			.67	.68	.54
20	c_nasasrb			.71	2.3	.78
21	c_bcsstk30			.52	.40	.23

algorithm computes the smallest mean square wavefront for all but three of the eighteen problems. Note that even for the problems where the spectral algorithm does poorly relative to the Sloan algorithm, the post-processing enables the hybrid algorithm to compute relatively small wavefronts. In general, the spectral and Sloan algorithms tend to vie for second place with RCM finishing fourth.

These algorithms also yield smaller maximum wavefront sizes than RCM. The normalized Sloan algorithm yields values about 52% of RCM, while the hybrid computes values about 38% of RCM. Thus these algorithms lead to reduced storage requirements for frontal factorization methods.

The results for the envelope size are similar. The hybrid, on average, reduces the envelope size to 37% of that of the RCM ordering, and to 73% of that of the normalized Sloan algorithm.

The Sloan, spectral, and the hybrid algorithms all reduce the wavefront size and envelope size at the expense of increased bandwidth. This is expected for the Sloan algorithm since Figures 10 and 11 show that the weights yielding small wavefront sizes are quite different from the weights for small bandwidth. It is also not surprising for the spectral and the hybrid algorithms since their objective functions, 2-sum (for spectral, see [36]) and wave front size (for the hybrid) differ from the bandwidth.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE 9
Envelope sizes relative to RCM.

	Problem	$\|env(A)\|$ RCM	Sloan	FastSloan	spectral	hybrid
			normalized by RCM			
1	barth	7.01e5	.69	.66	.66	.55
2	barth4	7.03e5	.64	.47	.46	.40
3	barth5	3.26e6	.75	.43	.44	.39
4	shuttle	7.09e5	.81	.82	1.00	.85
5	copter1	8.62e6	.84	.68	.89	.74
6	copter2	7.55e7	.63	.53	.56	.43
7	ford1	2.90e6	.81	.80	.68	.61
8	ford2	5.72e7	.71	.71	.65	.56
9	skirt	4.42e7	.77	.72	.70	.63
0	nasasrb	2.06e7	.89	.88	.99	.87
11	commanche_dual	5.90e5	.73	.59	.61	.47
12	tandem_vtx	1.53e7	.42	.37	.40	.34
13	tandem_dual	1.22e8	.72	.54	.39	.34
14	onera_dual	1.71e8	.66	.46	.31	.27
15	bcsstk30	2.66e7	.60	.53	.33	.25
16	pds10	2.95e7	.41	.34	.82	.38
17	finance256	3.26e7	.20	.22	.28	.20
18	finance512	5.55e7	.21	.25	.34	.20
19	c_skirt			.70	.74	.65
20	c_nasasrb			.86	1.10	.89
21	c_bcsstk30			.52	.38	.26

TABLE 10
Bandwidths relative to RCM.

	Problem	bandwidth RCM	Sloan	FastSloan	spectral	hybrid
			normalized by RCM			
1	barth	199	2.93	4.53	1.76	4.15
2	barth4	218	5.02	7.04	2.64	7.39
3	barth5	373	3.44	8.91	1.96	5.19
4	shuttle	238	3.50	3.39	2.66	4.05
5	copter1	932	3.80	7.34	1.02	7.82
6	copter2	2,975	4.05	11.4	1.89	8.39
7	ford1	258	7.67	6.91	12.0	12.0
8	ford2	963	7.06	12.1	5.75	8.04
9	skirt	2,070	9.37	3.66	2.13	2.15
0	nasasrb	881	5.82	5.83	4.17	5.57
11	commanche_dual	155	9.94	15.9	2.52	8.15
12	tandem_vtx	1,847	2.35	3.56	1.39	2.29
13	tandem_dual	2,199	3.55	9.07	2.92	4.72
14	onera_dual	3,478	8.93	11.3	2.08	3.19
15	bcsstk30	2,826	5.60	5.11	1.91	2.28
16	pds10	4,235	3.59	3.77	1.87	3.58
17	finance256	2,014	4.41	4.11	2.49	2.44
18	finance512	1,306	3.26	2.88	2.84	2.38
19	c_skirt			6.07	3.19	3.16
20	c_nasasrb			5.81	6.83	4.72
21	c_bcsstk30			4.02	2.05	2.03

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE 11
CPU times relative to the RCM algorithm.

	Problem	bandwidth RCM	normalized by RCM fast sloan	spectral	hybrid
1	barth	.13	1.9	10.	11.
2	barth4	.05	3.4	18.	20.
3	barth5	.16	2.7	19.	21.
4	shuttle	.12	2.7	15.	17.
5	copter1	.13	4.7	25.	28.
6	copter2	.88	3.0	18.	20.
7	ford1	.30	1.7	12.	13.
8	ford2	1.1	2.7	19.	21.
9	skirt	5.0	1.7	3.7	4.5
0	nasasrb	3.3	2.3	8.5	9.7
11	commanche_dual	.07	2.1	19.	19.
12	tandem_vtx	.27	2.7	14.	16.
13	tandem_dual	1.4	2.2	14.	15.
14	onera_dual	1.2	2.3	15.	15.
15	bcsstk30	3.7	1.7	3.2	4.0
16	pds10	.35	2.1	36.	37.
17	finance256	.51	2.4	16.	18.
18	finance512	1.0	2.3	17.	18.
19	c_skirt		.33	.69	.91
20	c_nasasrb		.49	1.8	2.3
21	c_bcsstk30		.34	.56	.74

On these test problems, our efficient implementation of the Sloan algorithm requires on average only 2.1 times that of the time taken by the RCM algorithm. The hybrid algorithm requires about 5.0 times the time taken by the Sloan algorithm on the average. This ratio is always greater than one, since the hybrid algorithm uses second step of the Sloan algorithm (numbering the vertices) to refine the spectral ordering, and the eigenvector computation is much more expensive than the first step of the Sloan algorithm (the pseudo-diameter computation). We believe that these time requirements are small for the applications that we consider: preconditioned iterative methods and frontal solvers.

5.1.2.a Gains from Compressed Graphs

As discussed in Section 3.2.4, the use of the supervariable connectivity graph [26] (called the compressed graph by Ashcraft [3]) can lead to further gain in the execution times of the algorithms. Only three of the problems, `skirt`, `nasasrb`, `bcsstk30`, compressed well. This is because many of the multi-component finite element problems in our test set had only one node representing the multiple degrees of freedom at that node. The compression feature is an important part of many software packages for

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

solving PDE's, since it results in reduced running times and storage overheads, and our results also show impressive gains from compression.

Three problems in our test set compressed well: `skirt`, `nasasrb`, and `bcsstk30`. Results for these problems are shown in the last three rows of each table. The numbers of multivertices and edges in the compressed graphs are also shown. For these three problems, compression speeds up the Sloan algorithm on average by a factor of nearly 5, and the hybrid algorithm by a factor of 4.6.

Compression improves the quality of the Sloan algorithm for these three problems, and does not have much impact on the hybrid algorithm. This improved quality of the compressed Sloan algorithm follows from our choice of parameters in the compressed algorithm to correspond exactly to their values in the uncompressed graph. However, on `nasasrb`, the spectral envelope parameters deteriorate upon compression. We do not know the reason for this, but it could be due to the poorer quality of the eigenvector computed for the weighted problem. In any case, the compressed hybrid algorithm recoups most of this deterioration.

5.2 Polymorphic Minimum Fill

In Section 5.2.1 we compare the quality and runtime of *Spindle* against other implementations that are publicly available. We compare different heuristics derived from `PriorityStrategy` in Section 5.2.2. In Section 5.2.3 we examine how these different heuristics behave in more detail by switching the heuristic in the middle of the computation and examining how the behavior changes. Finally in Section 5.2.4, we react to recent work by Bornstein [14], who reports that state-of-art minimum degree algorithms perform poorly compared to state-of-art nested dissection algorithms.

Our test set is listed in Table 12. The matrices are ordered in increasing size of the factor using MMD ordering. Each number reported is the average of 11 runs with different seeds in the random number generator. All the software was compiled with the EGCS compilers [1] which is a publicly available experimental branch of the GNU compilers [2].

---

[1] http://www.cygnus.com
[2] http://www.gnu.org

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE 12

Test Set for Fill Reducing Orderings. The test set is sorted by increasing work to factor for Spindle's MMD.

	Problem	$\|V\|$	$\|E\|$
1	commanche	7,920	11,880
2	barth4	6,019	17,473
3	barth	6,691	19,748
4	bcsstk34	588	10,415
5	ford1	18,728	41,424
6	ken13	28,632	66,586
7	barth5	15,606	45,878
8	shuttle_eddy	10,429	46,585
9	bcsstk38	8,032	173,714
10	bcsstk18	11,948	68,571
11	bcsstk23	3,134	21,022
12	bcsstk16	4,884	142,747
13	bcsstk15	3,948	56,934
14	bcsstk17	10,974	208,838
15	pwt	36,519	144,794
16	ford2	100,196	222,246
17	crystk01	4,875	155,508
18	bcsstk35	30,237	709,963
19	msc10848	10,848	609,465
20	bcsstk37	25,503	557,737
21	msc23052	23,052	565,881
22	bcsstk36	23,052	560,044
23	bcsstk30	28,924	1,007,284
24	tandem_vtx	18,454	117,448
25	pds10	16,558	66,550
26	bcsstk32	44,609	985,046
27	struct3	53,570	560,062
28	copter1	17,222	96,921
29	bcsstk33	8,738	291,583
30	struct4	4,350	116,724
31	bcsstk31	35,588	572,914
32	crystk02	13,965	477,309
33	nasasrb	54,870	1,311,227
34	skirt	45,361	1,268,228
35	tandem_dual	94,069	183,212
36	onera_dual	85,567	166,817
37	copter2	55,476	352,238
38	crystk03	24,696	863,241
39	3dtube	45,330	1,584,144
40	cfd1	70,656	878,854
41	gearbox	153,746	4,463,329
42	cfd2	123,440	1,482,229

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.2.1 Comparing Implementations

While these algorithms are conceptually simple, they are very challenging to implement and to optimize. There are a few publicly available implementations of MMD and AMD. The original implementation of MMD by Liu [58] is called GENMMD. One of the original implementations of AMD is AMDBAR by Amestoy, Davis and Duff [2]. This is a publicly released version, there is another version that is included with the Harwell Sparse Library. The newer algorithms by Ng and Raghavan [64] and Rothberg and Eisenstat [74] were reported by modifying each of the two implementations above; neither group has released their modified versions.

A third implementation of multiple minimum degree that we compare against is written by Ashcraft and extracted from a much larger body of software. The component we use, MSMD (Multi-Stage Minimum Degree) is designed to work in multiple stages — commonly as the degenerate case in a nested dissection ordering. It was originally released in SMOOTH [5] and later in a larger project with several contributers called SPOOLES [4]. This implementation by Ashcraft is the closest to our own in spirit. It is an object based design, though implemented in C. None of the implementations mentioned above implement all the heuristics provided in Spindle, though SPOOLES does have MMD and a variety of alternate settings that, in effect, implement AMD with weaker approximations.

We compare the performance of GENMMD, AMDBAR, SPOOLES, and Spindle in Tables 13 and 14. Table 13 focuses on relative performance of Multiple Minimum Degree (MMD) implementations. All numbers are normalized by the corresponding value of Spindle's MMD. We list the relative size of the factor (storage), the amount of computational work required to compute the factor (work), and the time required to compute the ordering. This information is tabulated for Approximate Minimum Degree (AMD) in Table 14.

GENMMD is over 500 lines Fortran77 code with four subroutines, and 30+ goto statements. It is optimized in terms of speed as well as memory requirements. The same array is used to maintain several distinct linked lists at the same time. For SPOOLES, we set it to do multiple elimination, precompress the graph, do compression of 2-adjacent supernodes at each elimination step, and exact quotient graph

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE 13

Compararison of MMD implementations: GENMMD and SPOOLES vs. Spindle. For each implementation, we present the size of the factor, the amount of work required to compute the numerical values in the factor, and the time taken to generate the ordering.

	problem	GENMMD/Spindle			SPOOLES/Spindle		
		size	work	time	size	work	time
1	commanche	1.00	.97	.23	1.03	1.04	.62
2	barth4	.99	.98	.27	1.01	1.00	.77
3	barth	.99	.97	.31	1.01	99	.72
4	bcsstk34	1.00	1.01	.75	1.04	1.08	1.50
5	ford1	1.00	.99	.37	1.02	1.04	.70
6	ken13	1.00	1.00	1.57	1.02	1.01	1.46
7	barth	1.00	1.01	.34	1.00	.97	.74
8	shuttle_eddy	1.01	1.04	.37	1.01	1.01	.78
9	bcsstk38	1.03	1.07	1.02	1.03	1.09	2.36
10	bcsstk18	1.01	1.02	.34	1.06	1.16	.58
11	bcsstk23	.99	.96	.39	1.01	1.01	.39
12	bcsstk16	.98	.96	.98	1.00	.99	3.37
13	bcsstk15	1.00	1.00	.37	.99	.97	.50
14	bcsstk17	1.03	1.06	.89	1.06	1.16	2.57
15	pwt	1.00	1.01	.53	1.03	1.07	.85
16	ford2	1.01	1.05	.67	1.06	1.17	.90
17	crystk01	.97	.93	.97	.98	.95	3.40
18	bcsstk35	1.01	1.00	2.14	1.01	99	7.16
19	msc10848	1.02	1.04	1.54	1.02	1.03	6.18
20	bcsstk37	1.00	.99	1.41	1.02	1.03	5.09
21	msc23052	1.01	1.02	1.90	1.02	1.05	8.13
22	bcsstk36	1.02	1.02	1.88	1.02	1.05	7.64
23	bcsstk30	1.03	1.07	1.45	1.04	1.08	3.46
24	tandem_vtx	1.00	.99	.39	1.03	1.09	.65
25	pds10	.99	.99	.78	1.04	1.02	.07
26	bcsstk32	1.00	.96	1.80	1.00	.98	4.01
27	struct3	1.00	.99	.92	1.05	1.09	1.38
28	copter1	1.00	.99	.31	.97	.95	.46
29	bcsstk33	.99	.97	.57	1.01	1.02	1.51
30	struct4	1.05	1.11	.22	1.04	1.10	.21
31	bcsstk31	1.04	1.07	1.00	1.08	1.19	1.86
32	crystk02	.98	.95	1.15	.98	.95	3.41
33	nasasrb	1.00	1.01	1.66	1.14	1.52	2.55
34	skirt	1.01	1.01	1.74	1.04	1.07	3.70
35	tandem_dual	.99	.95	.86	1.05	1.09	.70
36	onera_dual	1.00	1.00	.84	1.05	1.08	.68
37	copter2	1.01	1.01	.60	1.01	1.02	.64
38	crystk03	1.00	1.00	1.16	1.00	1.00	3.11
39	3dtube	1.00	1.02	1.61	1.00	1.01	3.25
40	cfd1	1.00	1.00	.42	.98	.94	.41
41	gearbox	1.01	1.02	2.67	1.01	1.03	3.50
42	cfd2	1.00	.99	.57	.99	.98	.48
geometric mean		1.00	1.00	.77	1.02	1.08	1.36
mean		1.00	1.01	.95	1.02	1.08	2.20

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE 14

Comparison of AMD Implementations: AMDBAR and SPOOLES vs. Spindle. For each implementation, we present the size of the factor, the amount of work required to compute the numerical values in the factor, and the time taken to generate the ordering.

	problem	AMDBAR/Spindle			SPOOLES/Spindle		
		size	work	time	size	work	time
1	commanche	1.00	.99	.10	1.05	1.09	.34
2	barth4	.99	.98	.14	1.02	1.02	.47
3	barth	1.00	1.01	.12	1.02	1.02	.41
4	bcsstk34	.99	.98	.33	1.03	1.07	1.44
5	ford1	1.00	1.02	.18	1.03	1.06	.40
6	ken13	1.00	1.01	.30	1.00	.96	2.52
7	barth5	1.00	1.00	.19	1.01	1.00	.46
8	shuttle_eddy	1.00	1.00	.18	1.02	1.02	.56
9	bcsstk38	1.01	1.01	.34	1.01	1.02	2.03
10	bcsstk18	1.00	1.00	.21	1.46	2.08	.65
11	bcsstk23	1.00	1.01	.24	1.03	1.07	.73
12	bcsstk16	.99	.98	.34	1.01	1.01	2.16
13	bcsstk15	1.01	1.03	.30	1.01	1.03	1.14
14	bcsstk17	1.01	1.04	.32	1.15	1.30	1.50
15	pwt	.99	.96	.37	1.02	1.00	.57
16	ford2	.99	.96	41.46	1.04	1.07	.52
17	crystk01	.96	.91	.40	.99	.98	2.14
18	bcsstk35	.99	.88	.54	1.00	.99	2.38
19	msc10848	1.01	1.04	.59	1.00	.99	1.46
20	bcsstk37	1.00	.99	.52	.99	.98	2.14
21	msc23052	1.00	1.00	.53	1.01	1.03	2.87
22	bcsstk36	1.01	1.04	.51	1.00	1.01	2.59
23	bcsstk30	1.01	1.02	.59	1.02	1.04	2.80
24	tandem_vtx	1.01	1.03	.32	1.04	1.09	.70
25	pds10	.99	.98	.38	1.01	1.00	1.04
26	bcsstk32	1.00	1.00	.59	1.01	1.01	2.00
27	struct3	1.00	1.02	.45	1.04	1.07	.92
28	copter1	1.01	1.03	.26	1.01	1.03	.58
29	bcsstk33	1.00	.99	.39	1.00	1.01	2.06
30	struct4	.99	.96	.44	1.04	1.07	1.91
31	bcsstk31	1.01	1.02	.54	1.02	1.00	1.48
32	crystk02	.97	.94	.52	.99	.98	2.68
33	nasasrb	1.00	1.00	.66	1.03	1.09	1.87
34	skirt	1.01	1.01	.65	1.04	1.08	2.23
35	tandem_dual	1.00	.99	.42	1.05	1.08	.58
36	onera_dual	1.02	1.04	.38	1.06	1.10	.54
37	copter2	1.02	1.05	.49	1.03	1.06	.72
38	crystk03	.98	.95	.58	.99	.98	2.41
39	3dtube	.98	.96	.79	.98	.95	2.50
40	cfd1	.99	.97	.71	.99	.96	1.07
41	gearbox	1.01	1.03	25.84	1.03	1.10	2.27
42	cfd2	1.01	1.02	17.91	1.00	.99	1.78
geometric mean		1.00	1.00	.50	1.03	1.05	1.24
mean		1.00	1.00	2.41	1.03	1.06	1.54

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

updates. The quality of the result between GENMMD, SPOOLES, and Spindle is very close. The difference between them could easily be discrepancies in how tie breaking is handled inside each code. In every case where Spindle's MMD is faster than GENMMD, it is because Spindle's MMD precompresses the graph and GENMMD does not. Spindle is generally slower than GENMMD and for this test set SPOOLES is slightly slower than Spindle's MMD. An interesting exception is for the problem number 17, pds10. Both the GENMMD and Spindle's MMD implementations take a significant amount of time to compute the ordering, especially considering the size of the problem, but SPOOLES seems to not suffer. We will investigate this problem in some detail in Section 5.2.3.

AMDBAR is also over 500 lines of Fortran77 code with 30+ goto statements and is organized into a single subroutine. Because AMD can do more aggressive graph compression at each elimination step, precompression is not an issue and AMDBAR soundly beats Spindle's AMD in execution time. There are, however, three exceptional cases where AMDBAR runs significantly slower than Spindle's AMD: ford2, gearbox, and cfd2. Looking at the unnormalized data and the size of the problems involved, it is clear that AMDBAR is taking an inordinate amount of time to compute the orderings for these problems. We have not taken the time to understand why AMDBAR has difficulty in these instances. Because of the structure of the AMDBAR code, common analysis techniques, such as running a basic block profiler is not immediately helpful. In terms of quality, because AMD does not rely on independent sets of vertices it appears to not be as sensitive to tie breaking. The quality of the results in the AMDBAR and Spindle's AMD implementations are very close. To set SPOOLES on an equal footing, we set it to do single elimination, do approximate quotient graph updates, and aggressive graph compressions at each step. This is the closest setting to an AMD ordering, the quality of the results for SPOOLES at these settings is slightly inferior to AMDBAR and Spindle's AMD.

5.2.2 Different Algorithms within Spindle

The following three Tables 15–17 compare the storage, work and execution time for a collection of different heuristics derived from the `PriorityStrategy` class. We chose to compare the MMD and AMD implementations as well as the algorithms proposed by Rothberg and Eisenstat [74] Minimum Mean Fill

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(MMF) and Minimum Increase in Neighbor Degree (MIND), both the MMD based and AMD based versions. The data for the different heuristics is normalized by the results of the MMD ordering.

For this test set, we see that the quality of the results from MMD and AMD are very close. We also see that AMMF is better than MMF but that AMIND is slightly worse than MIND. This is consistent with results reported by Rothberg and Eisenstat [74]. Comparing the times in Table 17 we see that AMD has a definite advantage over MMD and that the overheads in computing fill or increase in neighbor degree is on par with the results reported in [74].

Considering Table 17 in detail, ken13, shows that multiple elimination strategies tend to perform 2.5-3 times faster than corresponding approximate elimination strategies. Conversely for pds10, the multiple elimination strategies take a large amount of time and the corresponding approximate elimination strategies are over 25 times faster. Interestingly, ken13 and pds10 are not very different in size and both come from multi-commodity flow problems.

To better illustrate the behaviors of these algorithms on these problems, see Figure 38. For each plot, the bottom line is the cumulative number of supernodes that are eliminated, the top line is the sum of the first line plus the number remaining supernodes in the quotient graph. The bottom line is strictly increasing since there is always at least one supernode eliminated at each step. The top line is non-increasing; decreasing only when indistinguishable supernodes are detected and removed. The horizontal axis is cpu time since each elimination stage has a different update cost.

Figures 38a and 38b show the pds10 problem for MMD and AMD respectively. Figures 38c and 38d show the same for the ken13 test matrix. In both cases, MMD is faster in the beginning, however, we can see in the case of pds10 that the rate of convergence between the two lines slows drastically where AMD does not.

To investigate why these algorithms behave so differently on these two different problems, we exercise an additional feature from using the Strategy Pattern in our design.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE 15
Storage requirements for factors using various fill-reducing orderings.

	Problem	size MMD	normalized by MMD AMD	MMF	AMMF	MIND	AMIND	MMMD
1	commanche	76,127	.99	.97	.98	.97	.98	.98
2	barth4	115,987	.99	.95	.94	.94	.95	.96
3	barth	111,781	.98	.95	.97	.98	.98	.98
4	bcsstk34	46,573	.98	.91	.87	.88	.84	.88
5	ford1	315,851	.99	.98	.98	.97	.98	.98
6	ken13	344,355	1.02	1.00	1.01	1.02	1.01	.98
7	barth5	374,285	.99	.94	.94	.95	.94	.99
8	shuttle_eddy	381,568	1.00	.93	.92	.93	.94	.95
9	bcsstk38	733,484	1.00	1.02	.93	.98	.96	.95
10	bcsstk18	632,545	1.00	.96	.90	.92	.92	.92
11	bcsstk23	462,048	.98	.88	.84	.87	.88	.93
12	bcsstk16	754,298	.98	.96	.94	.86	.91	1.00
13	bcsstk15	655,977	1.00	.96	.89	.92	.90	.96
14	bcsstk17	1,106,613	1.00	.97	.95	.96	1.00	.93
15	pwt	1,757,057	1.01	.97	.96	.96	.97	.96
16	ford2	2,408,562	1.01	.93	.94	.93	.95	.97
17	crystk01	1,089,378	1.01	.98	.99	.87	.94	.96
18	bcsstk35	2,730,331	1.00	.98	.99	.98	1.00	.90
19	msc10848	1,979,471	1.00	1.13	1.08	.97	.97	.98
20	bcsstk37	2,820,230	1.00	.98	.96	.98	.99	1.00
21	msc23052	2,725,581	1.00	1.00	.99	.99	.99	.97
22	bcsstk36	2,737,589	1.00	.99	.99	.98	.99	.98
23	bcsstk30	3,739,015	1.00	.94	.93	.91	.93	.98
24	tandem_vtx	2,634,154	.98	.79	.79	.84	.85	.95
25	pds10	1,627,376	.97	.97	.96	.94	.94	.89
26	bcsstk32	5,215,869	.98	.95	.93	.94	.95	1.00
27	struct3	5,342,841	.99	1.08	.94	.96	.96	.95
28	copter1	2,478,886	.97	.84	.82	.88	.85	.87
29	bcsstk33	2,666,854	.99	.94	.90	.85	.96	.95
30	struct4	2,237,851	1.01	.84	.78	.81	.83	.91
31	bcsstk31	5,134,592	1.01	.91	.85	.92	.88	.85
32	crystk02	6,170,998	1.00	.87	.90	.81	.92	.97
33	nasasrb	12,515,804	.99	.93	.84	.95	.92	.83
34	skirt	10,807,462	.99	.95	.93	.89	.91	.93
35	tandem_dual	11,400,057	.98	.75	.75	.80	.81	.85
36	onera_dual	11,046,392	.98	.74	.76	.79	.81	.86
37	copter2	14,095,311	.98	.74	.74	.80	.80	.85
38	crystk03	14,140,693	1.00	.86	.88	.79	.89	.81
39	3dtube	31,845,622	1.01	.87	.90	.85	.89	.86
40	cfd1	39,970,236	1.00	.72	.73	.75	.81	.81
41	gearbox	52,908,796	.99	.91	.92	.87	.91	.89
42	cfd2	91,232,664	.99	.73	.76	.76	.81	.79
geometric mean			.99	.92	.90	.90	.92	.92
mean			.99	.92	.90	.90	.92	.93

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE 16
Work requirements for factors using various fill-reducing orderings.

	Problem	work × 10^6 MMD	normalized by MMD AMD	MMF	AMMF	MIND	AMIND	MMDF
1	commanche	1.67	.98	.90	.91	.89	.91	.91
2	barth4	4.00	.97	.83	.82	.81	.83	.88
3	barth	4.36	.95	.84	.88	.90	.91	.90
4	bcsstk34	4.77	.95	.78	.71	.73	.66	.74
5	ford1	15.7	.97	.92	.92	.91	.93	.91
6	ken13	16.0	1.06	.97	1.00	1.03	.99	.94
7	barth5	19.0	.98	.83	.82	.84	.82	.88
8	shuttle_eddy	25.3	1.00	.80	.77	.80	.83	.88
9	bcsstk38	117.	1.01	1.00	.78	.93	.88	.98
10	bcsstk18	125.	1.02	.90	.76	.82	.83	.90
11	bcsstk23	147.	.95	.73	.65	.74	.75	.83
12	bcsstk16	153.	.96	.87	.84	.70	.80	.84
13	bcsstk15	169.	.98	.85	.71	.82	.79	.86
14	bcsstk17	188.	1.01	.88	.81	.89	.97	.90
15	pwt	222.	1.06	.91	.89	.92	.93	.92
16	ford2	290.	1.06	.79	.81	.81	.84	.84
17	crystk01	342.	1.03	.89	.92	.72	.85	.80
18	bcsstk35	393.	1.01	.91	.92	.91	.94	.96
19	msc10848	534.	.99	1.34	1.17	.96	.94	.96
20	bcsstk37	556.	1.00	.90	.87	.96	.94	.91
21	msc23052	598.	1.02	.98	.92	.92	.93	.91
22	bcsstk36	612.	1.00	.93	.94	.90	.92	.90
23	bcsstk30	887.	1.00	.81	.77	.78	.81	.86
24	tandem_vtx	958.	.95	.56	.56	.68	.69	.77
25	pds10	1,040.	.93	.93	.92	.88	.88	.99
26	bcsstk32	1,120.	.94	.82	.78	.81	.83	.82
27	struct3	1,240.	.99	1.26	.83	.92	.91	.89
28	copter1	1,310.	.94	.64	.61	.77	.71	.82
29	bcsstk33	1,330.	.98	.80	.72	.67	.84	.73
30	struct4	1,750.	1.03	.68	.58	.66	.69	.72
31	bcsstk31	2,380.	1.04	.81	.67	.81	.72	.92
32	crystk02	4,430.	.99	.70	.75	.63	.83	.67
33	nasasrb	5,150.	.98	.91	.66	.95	.82	.84
34	skirt	5,560.	.97	.86	.82	.75	.78	.83
35	tandem_dual	8,330.	.93	.50	.50	.60	.62	.70
36	onera_dual	9,590.	.95	.49	.54	.60	.62	.71
37	copter2	12,400.	.95	.53	.54	.64	.65	.74
38	crystk03	13,600.	1.00	.67	.72	.62	.76	.65
39	3dtube	42,200.	1.02	.77	.80	.75	.80	.77
40	cfd1	47,900.	1.01	.47	.49	.56	.65	.68
41	gearbox	57,200.	.98	.74	.79	.72	.79	.77
42	cfd2	185,000.	.96	.53	.57	.58	.67	.65
geometric mean			.99	.80	.76	.78	.81	.83
mean			.99	.82	.77	.79	.82	.83

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE 17

CPU time to generate various fill-reducing orderings.

	Problem	time (secs) MMD	normalized by MMD					
			AMD	MMF	AMMF	MIND	AMIND	MMMD
1	commanche	.21	2.46	1.28	2.60	1.21	2.53	1.09
2	barth4	.15	2.01	1.29	2.16	1.23	2.23	1.17
3	barth	.18	2.23	1.34	2.40	1.28	2.41	1.20
4	bcsstk34	.02	.90	1.45	1.15	1.10	.80	1.30
5	ford1	.56	2.04	1.27	2.16	1.19	2.15	1.11
6	ken13	1.28	2.61	1.16	2.87	1.13	3.05	1.10
7	barth5	.44	1.88	1.26	2.00	1.21	2.03	1.10
8	shuttle_eddy	.28	1.74	1.27	1.87	1.17	1.86	1.07
9	bcsstk38	.31	.65	1.94	.77	1.28	.82	1.20
10	bcsstk18	.79	.95	1.80	1.01	1.21	1.01	1.09
11	bcsstk23	.41	.47	2.04	.61	1.17	.52	1.05
12	bcsstk16	.12	.97	2.15	1.55	1.19	1.10	1.15
13	bcsstk15	.42	.46	2.54	.77	1.19	.60	1.06
14	bcsstk17	.28	1.10	1.62	1.35	1.28	1.35	1.14
15	pwt	1.29	1.55	1.34	1.64	1.19	1.65	1.05
16	ford2	3.97	1.74	1.20	1.84	1.15	1.81	1.12
17	crystk01	.13	.87	2.20	1.47	1.06	1.20	.94
18	bcsstk35	.36	1.34	1.39	1.54	1.41	1.66	1.20
19	msc10848	.20	1.06	2.71	1.23	1.31	1.28	1.13
20	bcsstk37	.35	1.27	1.65	1.45	1.53	1.61	1.30
21	msc23052	.23	1.32	1.27	1.54	1.41	1.76	1.18
22	bcsstk36	.23	1.42	1.32	1.58	1.41	1.75	1.16
23	bcsstk30	.66	.98	1.66	1.07	1.33	1.18	1.15
24	tandem_vtx	1.18	1.02	1.37	1.20	1.20	1.18	1.10
25	pds10	48.65	.05	1.01	.02	1.43	.05	1.22
26	bcsstk32	.90	1.09	1.70	1.25	1.36	1.33	1.17
27	struct3	2.03	1.07	6.09	.97	1.23	1.06	1.18
28	copter1	1.26	1.03	1.56	1.38	1.11	1.18	1.02
29	bcsstk33	.56	.49	4.20	.88	1.16	.66	1.09
30	struct4	2.54	.13	4.50	.33	.98	.18	.93
31	bcsstk31	1.35	.85	1.71	.97	1.27	1.05	1.13
32	crystk02	.45	.73	1.67	1.20	1.02	1.08	.91
33	nasasrb	1.78	.84	2.16	1.05	1.53	1.18	1.23
34	skirt	1.44	.79	2.34	.99	1.36	1.09	1.08
35	tandem_dual	6.09	1.24	1.20	1.35	1.13	1.33	1.06
36	onera_dual	5.66	1.28	1.20	1.34	1.13	1.34	1.06
37	copter2	5.47	.92	1.25	1.05	1.13	1.05	1.02
38	crystk03	.93	.71	1.54	1.04	1.02	1.05	.89
39	3dtube	1.81	.74	1.29	1.05	1.12	1.20	.92
40	cfd1	14.46	.40	1.22	.50	.96	.54	.88
41	gearbox	5.39	.79	2.59	.96	1.30	1.17	1.03
42	cfd2	21.36	.47	1.17	.59	1.00	.64	.90
geometric mean			.92	1.69	1.13	1.21	1.12	1.09
mean			1.11	1.85	1.30	1.22	1.31	1.09

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

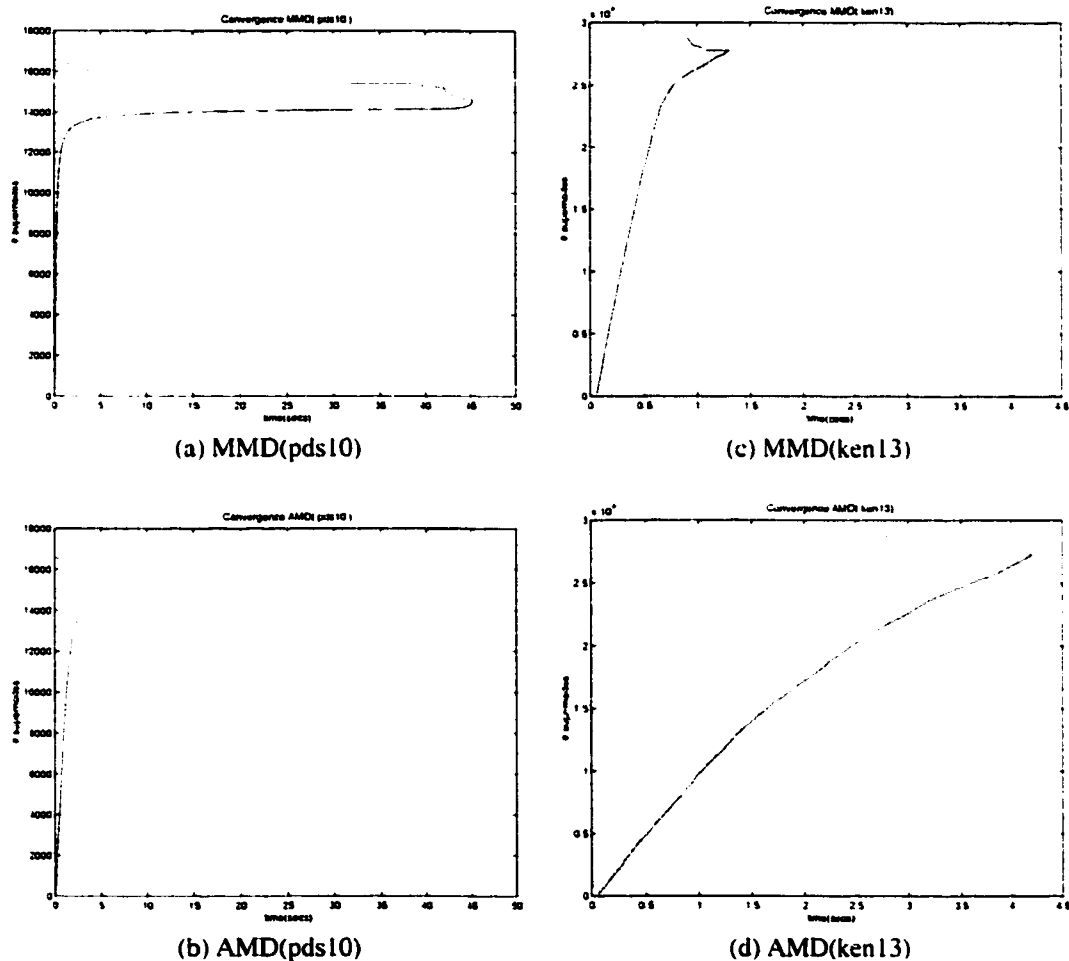


FIG. 38. *Supernode compression and elimination in fill-reducing orderings. Plotted are the cumulative number of eliminated supernodes (bottom line) and total supernodes either eliminated or outstanding (top line) vs. execution time. The two lines converge when all principal supernodes that exist have been eliminated.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.2.3 Switching Algorithms Mid-Stream

Because our implementation is object-oriented and because we paid careful attention to decomposition of duties between the objects, we have some additional flexibility in our system that we can exercise. The `MinPriority` algorithm class uses the services of a `PriorityStrategy`, but never knows exactly which one it is using, so the exact type of strategy can be switched in the middle of the ordering computation.

For this to be successful, the `MinPriority` and `QuotientGraph` classes must be amenable to dynamically changing some of their characteristics as well. The exact type of `PriorityStrategy` must communicate to the `MinPriority` algorithm whether to do single or multiple elimination and it must notify the `QuotientGraph` class whether to do approximate updates, or exact updates.

We created a `Switcher` class that implements MMD until a certain condition is met. After that condition is met, it changes its behavior to AMD. For the purposes of this experiment, the conditions required to switch from MMD to AMD was simply the percentage of original nodes that were eliminated from the quotient graph. In Figures 39 and 40 we show how pds10 and ken13, respectively, performed when switching from MMD to AMD at various points in the computation.

In Figure 39 we show experiments in which the switch between MMD and AMD is changed in 10% increments; we also zoom in at the 84-87% range where there is an abrupt transition. Careful examination reveals that MMD is stalling when eliminating the last two to three thousand supernodes in the quotient graph. AMD, however not only eliminates the remaining quotient graph faster, it also does so by eliminating far less supernodes.

As the elimination progresses, not only does the quotient graph have far fewer vertices, it also becomes increasingly connected. For pds10, the last 2000 supernodes are almost, but not completely connected. This means that MMD cannot find large independent sets of vertices — which is crucial to its efficient execution. However, there are a lot of indistinguishable nodes that are available and MMD is not finding them. This is because MMD does a lazy update and has a larger set of supernodes to handle at each quotient graph update. Therefore it is too expensive to do an exhaustive search of all possible pairs of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

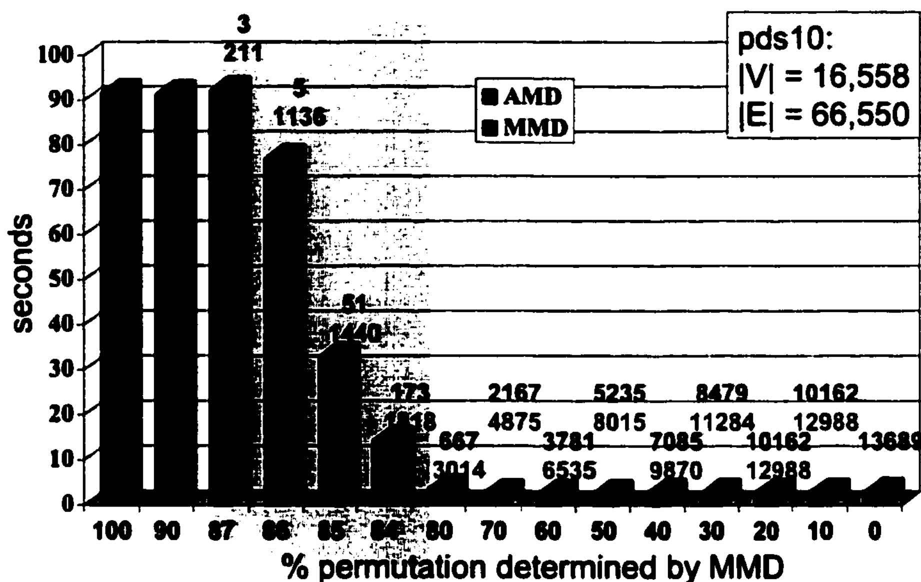


FIG. 39. *Details of pds10 when dynamically changing from MMD to AMD. Each bar represents a different run of the software with CPU time in the vertical direction. Each run starts with MMD and then switches to AMD when a certain percentage of the total permutation is determined. This percentage is shown on the horizontal axis. After the switch, AMD is used to complete the ordering. There are two numbers above each bar. The lower of the two is the number of supernodes in the quotient graph at the time MMD algorithm cedes control to AMD. The upper number is the number of supernode eliminations performed by AMD to complete the ordering. These two numbers are not necessarily the same by virtue of supernode amalgamation, though the upper number can be no greater than the lower one.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

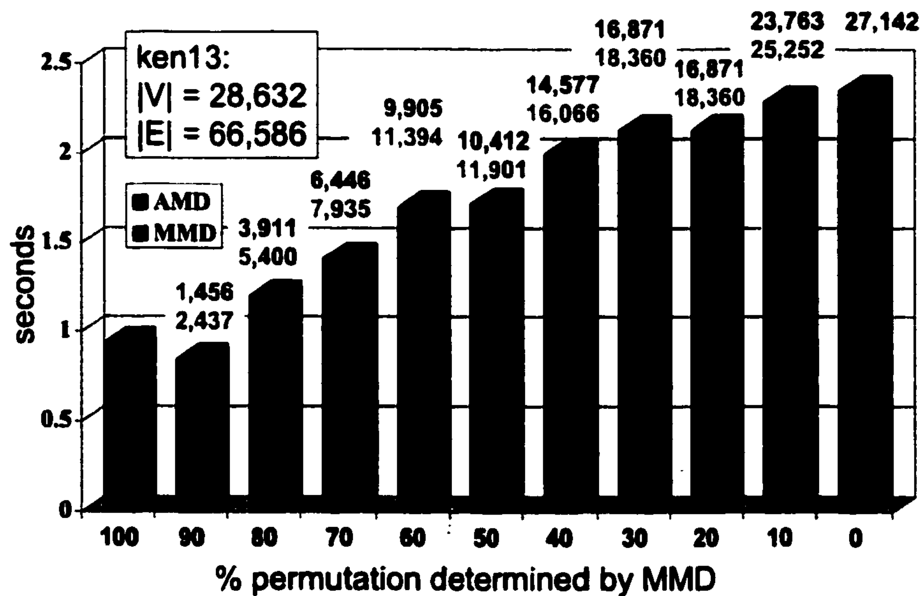


FIG. 40. *Details of ken13 when dynamically changing from MMD to AMD. at different points in the ordering. Each bar represents a different run of the software with CPU time in the vertical direction. Each run starts with MMD and then switches to AMD when a certain percentage of the total permutation is determined. This percentage is shown on the horizontal axis. After the switch to AMD, the algorithms completes the ordering. There are two numbers above each bar. The lower of the two is the number of supernodes in the quotient graph at the time MMD algorithm cedes control to AMD. The upper number is the number of supernode eliminations performed by AMD to complete the ordering. These two numbers are not necessarily the same by virtue of supernode amalgamation, though the upper number can be no greater than the lower one.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

indistinguishable supernodes. What it does instead is check all so-called "two adjacent" supernodes. These are supernodes that are adjacent to exactly two enodes and no other supernodes.

AMD, on the other hand, has a much smaller set of supernodes to deal with at every graph update, and since it is a single elimination scheme all the supernodes are reachable to each other through the most recently created enode. AMD then can afford an exhaustive search for indistinguishable nodes. This strategy pays off in the end because AMD can compress approximately the last 2000 supernodes into just under 200.

For ken13 in Figure 40 the picture is quite different. Cursory examination in this case shows that AMD's aggressive graph compression is not paying off, and in fact is much slower than MMD. Because AMD is a single elimination scheme there are many more quotient graph updates to perform, one for each eliminated supernode. MMD can reduce the quotient graph from 26 to 18 thousand supernodes with just two quotient graph updates.

We note that in both ken13 and pds10 the optimum is neither MMD or AMD, but some combination thereof. The intuition is that initially, when the quotient graph is very sparse and there are large independent sets of vertices to be had, MMD is better to use. Later in the elimination process, however, the quotient graph becomes much denser and there may not be as many sets of independent vertices to eliminate at once. AMD is better suited for this case especially with its more aggressive compression abilities. Of course, switching on the percentage of the total permutation that is completed is a crude control. It is quite possible that a more sophisticated property of the quotient graph, or some logic built into the `MinPriority` object that analyzes its behavior over the last few iterations would provide a much better reason to switch from a multiple elimination method to an approximate one.

It might seem more natural and a better design to make the `Switcher` class more generic by simply giving it two references to `PriorityStrategy` and letting it switch from any one strategy to another by forwarding the messages it receives to the current strategy. In this case, however, when the strategy switches it would be necessary to purge the `PriorityQueue` and recompute all the priorities of all the outstanding supernodes. This would incur some additional overhead, but would require no foreseeable changes in the architecture of the software. Currently, there is no need to purge and recompute for all

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE 18

Comparison of Spindle's greedy algorithms vs. reported nested dissection algorithms. All numbers are normalized by Spindle's MMD ordering.

	problem	size AMD	size AMMF	size Metis	size BEND	work AMD	work AMMF	work Metis	work BEND
8	shuttle_eddy	1.00	.92	.96	.86	1.00	.77	.91	.71
15	pwt	1.01	.96	.79	.85	1.06	.89	.51	.63
18	bcsstk35	1.00	.99	1.15	1.02	1.01	.92	1.29	1.00
20	bcsstk37	1.00	.96	1.12	.95	1.00	.87	1.24	.86
22	bcsstk36	1.00	.99	1.11	.93	1.00	.94	1.15	.79
23	bcsstk30	1.00	.93	1.19	1.04	1.00	.77	1.33	1.06
26	bcsstk32	.98	.93	1.10	.97	.94	.78	1.15	.86
27	struct3	.99	.94	.86	.83	.99	.83	.62	.57
29	bcsstk33	.99	.90	.87	.70	.98	.72	.67	.44
31	bcsstk31	1.01	.85	.86	.81	1.04	.67	.50	.49
33	nasasrb	.99	.84	.85	.78	.98	.66	.69	.55
39	3dtube	1.01	.90	.58	.56	1.02	.80	.29	.30
40	cfd1	1.00	.73	.57	.56	1.01	.49	.37	.28
41	gearbox	.99	.92	.72	.72	.98	.79	.41	.38
42	cfd2	.99	.76	.43	.43	.96	.57	.19	.16
geometric mean		1.00	.90	.84	.78	1.00	.75	.65	.54
mean		1.00	.90	.88	.80	1.00	.76	.76	.60

outstanding supernodes at the moment the strategy switches because AMD uses upper bounds on the degree that MMD uses. Any supernodes in the priority queue at that time are more precise than AMD would compute anyway, so there is no problem leaving them in.

5.2.4 Comparison against Nested Dissection

Recently, attention has been redirected to divide-and-conquer type fill-reducing algorithms using various forms of nested dissection. In his Ph.D. thesis, Bornstein [14] presented a comparison of AMD [2], Metis [50], BEND [46], for a set of matrices. In Table 18 we reproduce matrices that are common to both Bornstein's and our test set.

Even compared with the newest greedy fill-reducing algorithms, divide-and-conquer implementations provide superior permutations to reduce storage and work requirements. The cost to perform these nested dissection orderings, remains larger than the greedy algorithms, both in terms of cpu time and storage. The latter is due to the fact that many nested-dissection codes use multi-level strategies that requires significant amounts of memory.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

6. EXTENSIONS

One of the most encouraging signs of a well-engineered code is its adaptability to new circumstances. Indeed, extensibility was a primary design goal that we stated back in Section 2.2. In our work, we have extended the code in directions that we had not originally planned, but where circumstances (or opportunity) had presented itself. We pay particular attention to how well the software adapted, and how much modification was required. Our efforts run the gamut of software evolution; from deriving a new sub-class and plugging it in, to modifying or rewriting certain objects. Obviously the less original code that was modified, the better — and more successful — for us.

In Section 6.1 we extend our symmetric ordering algorithms to provide orderings for general sparse matrices. This section introduces additional notation for unsymmetric matrices, a discussion of some established practices, a column-oriented minimum fill algorithm (Section 6.1.1) and a novel generalization for the Sloan algorithm (Section 6.1.2).

In Section 6.2 we introduce some ordering algorithms with additional constraints. First we discuss a multi-stage, greedy fill reducing ordering (Section 6.2.1). This algorithm is a critical part of any nested dissection ordering where the graph that has already been partitioned, and now the subdomains are sufficiently small to do a greedy fill reducing ordering on each subdomain, respecting partition boundaries. Then we discuss a block-wavefront reducing algorithm (Section 6.2.2) where we want to minimize the global wavefront, but are subject to partition constraints where the partitions themselves are ordered and all vertices on a certain partition must be exhausted before proceeding to another.

6.1 Unsymmetric Orderings

Until this point, we have always made the assumption that we are dealing with a large, sparse, structurally symmetric matrix A, and wanted a single permutation P to symmetrically permute the system PAP^{T}. Let us now assume that the matrix is not structurally symmetric, indeed it may not even be square. In this case, we would require separate row and column permutations P_r, P_c.

In practice, unsymmetric orderings are avoided by making the problem symmetric.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

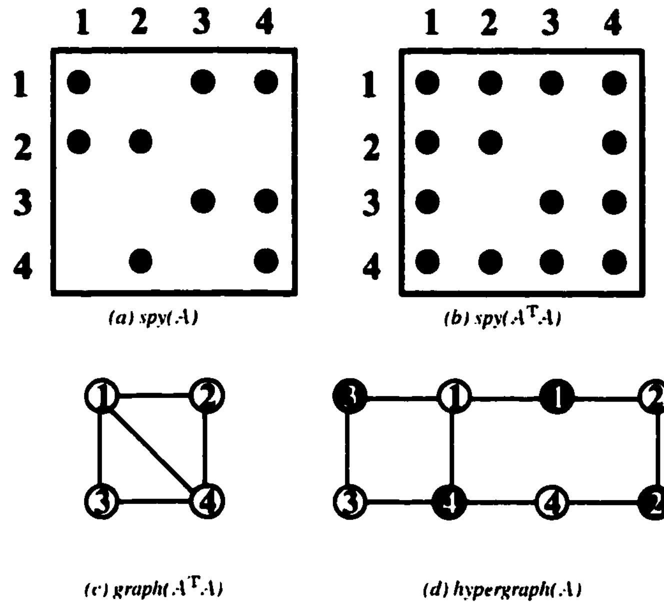


FIG. 41. *An example square, unsymmetric matrix. (a) Nonzero structure of the square, unsymmetric matrix. (b) Nonzero structure of $A^T A$. (c) Its column intersection graph. (d) Its hypergraph (white circles for columns, black circles for rows).*

For square matrices that are "almost" symmetric, one can perform a symmetric ordering on the sparse structure of $A + A^T$. The resulting permutation is applied to both the rows and columns of A. This process makes the ordering algorithm form a symmetric permutation that is an upper bound for the underlying unsymmetric one. Matlab's `symrcm()` function actually operates on the nonzero structure of $A + A^T$ [40].

For rectangular problems where A can have more rows than columns, one can perform a symmetric ordering on the sparsity structure of $A^T A$, which is symmetric but decidedly less sparse. The resulting permutation can then only be applied as a column permutation. This is effectively what is done in Matlab's `colmmd()` function [40] though the product, $A^T A$, is not explicitly formed.

For Sections 6.1.1–6.1.2 we will consider an example unsymmetric matrix shown in Figure 41(a).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

6.1.1 Column Minimum Fill

To adapt *Spindle* to perform a column minimim fill ordering for unsymmetric problems, we have a few choices.

We could derive a `ColIntersectionGraph` class from our generic `Graph` class and make its constructor form the graph of A^TA explicitly. Then we simply pass the graph to the existing `MinPriorityEngine` class and associate apparatus and return the result.

Alternatively, we can take an approach similar to Davis and Duff [18, 19] where we extend the `QuotientGraph` class to implicitly represent the graph of A^TA. To see how this second technique works, consider the hypergraph model in Figure 41(d). Instead of visualizing the black circles as the hub of a hyperedge, interpret them as enodes in a quotient graph. Once these initial conditions have been set, the regular `MinPriorityEngine` will produce an elimination order that we can use as the column fill-reducing ordering.

Although this second approach is more complicated to implement, there are considerable storage savings by using the implicit representation. This representation requires $\mathcal{O}(\text{nnz}(A))$ storage. Worst case for the explicit `ColIntersectionGraph` is $\mathcal{O}(\text{nnz}^2(A))$ (See Figure 1). Thus, we consider the implicit approach to be superior, even though setting up its initial conditions will require modifications to existing code.

The critical problem is constructing an appropriate `QuotientGraph`. The current implementation assumes that it is always constructed from a graph of all supernodes and no enodes. The simplest solution is to create a `HyperGraph` class, and pass it (via a pointer to `Graph`) to the existing constructor of a `QuotientGraph` class. With this solution, the `QuotientGraph` class (having no knowledge of hypergraphs) will interpret both vertices and hyperedge hubs as supernodes. Then, since no hyperedge hubs-turned-supernodes are adjacent, they form an independent set which can all be eliminated before updating the quotient graph. After updating the quotient graph, the quotient graph is in the initial state we want that implicitly represents A^TA. The problem is that the quotient graph will incorrectly report the number of supernodes, the number of eliminations, etc. since from its point of view the initial number of supernodes included the hyperedge-hubs. A second option is adding a feature to the `QuotientGraph`

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

class to recognize enodes that exist *a priori*. This can probably be done via some additional constructor using a `HyperGraph` or some similarly distinctive graph from an unsymmetric problem. We propose modifying the original `QuotientGraph` because its internal state is too complex to easily change its behavior consistently in a derived class.

Depending on how successful we are in encapsulating the details of the symmetry/unsymmetry in the `QuotientGraph`, we may need to fine-tune some details in the `MinPriorityEngine`. This is especially so if we build the unsymmetric quotient graph in the piecemeal fashion we first described. We are confident if we take our time with the `QuotientGraph`, the ordering engine can remain untouched. This also makes sense from very broad perspective since algorithm is the same whether the underlying matrix is symmetric or not.

6.1.2 Unsymmetric Sloan

Unlike the fill reduction problem where current state-of-art simply performs a column ordering, we can provide a wavefront-reducing row and column ordering for unsymmetric matrices. Furthermore, we can do so without modifying the `SloanEngine` at all.

Before progressing further, we need to define the unsymmetric generalizations of some terms we introduced in Section 3.2.1.

We start by generalizing the definition of a row width for the lower triangle of a general sparse square matrix.

Definition 6.1 *Consider a large, sparse, square matrix, $\mathcal{A}$. The* row width *for the ith row,* $\mathrm{rw}_i(\mathcal{A})$, *is the difference between i and the column index of the first nonzero entry of the row, or the diagonal (whichever comes first).*

$$\mathrm{rw}_i(\mathcal{A}) = \max_{j:a_{ij}\neq 0 \text{ or } j=i}(i-j).$$

We introduce a similar concept for the upper triangle of a general sparse, square matrix.

Definition 6.2 *Consider a large, sparse, square matrix, $\mathcal{A}$. The* column height *for the ith column,* $\mathrm{ch}_i(\mathcal{A})$, *is the difference between i and the row index of the first nonzero entry of the column, or the diagonal*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(whichever comes first).

$$\mathrm{ch}_j(A) = \max_{i: a_{ij} \neq 0 \text{ or } i=j} (j - i).$$

Remark 6.1 *For any sparse, square matrix, A,*

$$\mathrm{ch}_i(A) = \mathrm{rw}_i(A^T).$$

We now define terms like *bandwidth* and *envelope* for square, unsymmetric matrices.

Definition 6.3 *Given a large, sparse, square, unsymmetric matrix, A, the* bandwidth *of A is the sum of the maximum row width and the maximum column height*

$$\mathrm{bw}(A) = \max_{1 \leq i \leq n} \mathrm{rw}_i(A) + \max_{1 \leq j \leq n} \mathrm{ch}_j(A).$$

Definition 6.4 *Given a large, sparse, square, unsymmetric matrix, A, the envelope of A is the sum of the envelopes of both $L + L^T$ and the transpose of $U + U^T$.*

Remark 6.2 *Given a large, sparse, square, unsymmetric matrix, A, the size of the envelope of A is the sum of the row widths and column heights:*

$$|\mathrm{env}(A)| = \sum_{i=1}^{n} \mathrm{rw}_i(A) + \sum_{j=1}^{n} \mathrm{ch}_j(A).$$

These generalized terms seem to have the unfortunate side effect of doubling the values when applied to symmetric matrices. In fact, we have implicitly halved the terms for the symmetric case since the underlying system is symmetric and we can store, factor, and work with them in half the space. Matrices that are only structurally symmetric can only sometimes take advantage of this economy so the application of these terms depends on the context.

We need to define the wavefront of an unsymmetric matrix. In Section 3.2.1 we defined wavefront of a symmetric matrix through Cholesky factorization. For the unsymmetric matrix, we need to first understand *LU* factorization, of which Cholesky is a symmetric variant.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

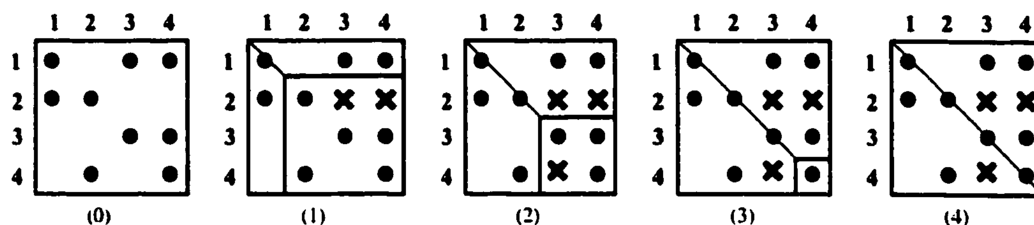


FIG. 42. *Example of LU factorization. The circles represent original nonzero entries, the crosses represent fill entries.*

We provide an example of LU factorization of the unsymmetric matrix in Figure 42. This is on the same unsymmetric matrix we first showed in Figure 41. The LU factorization decomposes a square matrix A into a lower triangular matrix L, and an upper triangular matrix U such that the product of L and U is equal to A[1]. In this particular example, we form the factors L and U in place. For detailed information about LU factorization, see Li [56].

Here, when factoring the ith row and column, we consider *active rows and columns*. An active row for column i is any of the last $n - i$ rows that has a nonzero entry in some k^{th} column where $k < i$. Similarly, an active column for row i is any of the last $n - i$ columns that has a nonzero entry in some k^{th} row where $k < i$. When factoring the ith row and column of an unsymmetric matrix, the number of active rows and columns (including both the ith row and column) is called the ith *wavefront* of A, $\mathrm{wf}_i(A)$.

Observation 6.1 *The sum of the wavefronts of a sparse, square, unsymmetric matrix equals the size of the envelope, plus 2n.*

$$\begin{aligned}\sum_{i=1}^{n} |\mathrm{wf}_i(A)| &= n + \sum_{i=1}^{n} \mathrm{rw}_i(A) + n + \sum_{i=1}^{n} \mathrm{ch}_j(A) \\ &= 2n + |\mathrm{env}(A)|\end{aligned}$$

Definition 6.5 *The ith* row wavefront *of a sparse, square, unsymmetric matrix A, $\mathrm{wf}_{r,i}(A)$ is the contribution to the wavefront by all the active columns on row i and row i itself. Similarly the ith* column wavefront, $\mathrm{wf}_{c,i}(A)$ *is the contribution to the wavefront by all the active rows on column i and column i itself.*

[1] We will assume for simplicity that the factorization is numerically stable and no pivoting is required.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Observation 6.2 *Given a sparse, square symmetric matrix A, let A_L be the lower triangle of A and A_U be the upper triangle of A. Then*

$$\mathrm{wf}_{r,i}(A) = \mathrm{wf}_i(A_L + A_L^{\mathrm{T}}),$$

and

$$\mathrm{wf}_{c,i}(A) = \mathrm{wf}_i(A_U + A_U^{\mathrm{T}}).$$

Now consider that we want to perform a wavefront reducing ordering on an unsymmetric matrix. An interesting way to accomplish this is to construct the hypergraph of the matrix. Then convert all the hyperedge hubs to vertices themselves, producing a graph of $2n$ vertices, where half of the vertices correspond to vertices in the hypergraph (and columns in the matrix) and the other half of the vertices correspond to hyperedge hubs in the hypergraph (and rows in the matrix). In this new graph of $2n$ vertices, we want to number adjacent pairs of vertices such that the number of unnumbered neighbors of all numbered vertices are minimized.

The number of as yet unnumbered vertices adjacent to a numbered vertex (plus two for the two just numbered) equals the wavefront at that step. Furthermore, the number of such vertices that originate from a vertex in the hypergraph (plus one) equals the column wavefront and the number of such vertices that originate from a hyperedge hub in the hypergraph (plus one) equals the row wavefront. Lastly, since the graph induced by this hypergraph is bipartite and we restrict ourselves to numbering adjacent pairs, we are guaranteed to generate the row and column permutations evenly and that there will be nonzeros along the diagonal of the permuted system.

Instead of writing a new algorithm to detect and removed mached pairs, we can simply run the regular `SloanEngine` the graph of $2n$ vertices. Once we get a permutation vector of length $2n$, we examine the order in which vertices are numbered. If the first vertex numbered corresponds to a vertex in the original hypergraph, we put that into one queue. If it corresponds to a hyperedge hub in the original hypergraph, we put it into another queue. Then we repeat with the rest of the vertices in the graph the Sloan algorithm numbered. When we are done, the order that the hypergraph vertices appear in the queue becomes the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

column permutation and the order that hyperedge hubs appear in the second queue determines the row permutation.

Although this new application of Sloan doesn't exactly satisfy the constraint of numbering adjacent pairs, we expect it to work well in practice. There is no need to modify exiting code, simply derive a `HyperGraph` from the `Graph` class, and add a driver to separate the larger permutation into the row and column permutations. Since the `HyperGraph` is also bipartite it is immune to the graph compression that the `SloanEngine` inherits from `OrderingAlgorithm`.

6.2 Constrained Orderings

Ordering and partitioning problems are strongly related. An ordering problem can be seen as an n-way partition. With these constrained ordering problems, the graph is already partitioned into subdomains. To preserve the partitioning, the ordering algorithm must order each subdomain completely before moving to another, while still minimizing some global property.

6.2.1 Multi-Stage Fill-Reducing Orderings

Nested dissection ordering is a divide-and-conquer approach to the fill reduction problem. It recursively selects a vertex separator that, upon removal, separates the graph into two independent subgraphs. By numbering the vertices in the separator last, it ensures that no fill edges can occur between the two subgraphs. This dissection then recurs on each subdomain until they become sufficiently small. At this point, there is a large collection of small, independent subgraphs that are yet unnumbered. These are typically ordered with MMD. Additionally, performing a MMD ordering on the vertex separators can also improve the overall reduction in fill.

Metis [50] as well as many other partitioning codes use GENMMD or some equivalent to order their subdomains. However, this requires actually generating all of these subdomains as independent graphs, running the algorithm, and then combining the results. One exception is SPOOLES [4], which has its own Multi-Stage Minimum Degree (MSMD) implementation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

We now investigate how we could add this feature to our existing code. At first glance, it would seem that that all we need to do is derive a new class from `MinPriorityEngine`, add an array to store the subdomain each vertex is in, and override the `execute()` method to produce the appropriate ordering. Although the entire `MinPriorityEngine` class is over 500 lines of code, the body for the `execute()` method is less than a page. There is, however, one problem: supernode amalgamation. It could happen that during elimination two supernodes from two different domains become indistinguishable. In this case, we need to prevent them from being amalgamated as this would violate the partition restrictions.

6.2.2 Block Wavefront Orderings

Running a wavefront reducing ordering in subdomains of a partitioned system can have many applications, including incomplete factorization preconditioning, and optimizing for cache performance on the deep memory heirarchies of current processors.

Our `SloanEngine` class has the ability already to handle many more general situations than a simple Sloan algorithm can handle. It is currently one of the most mature components in Spindle. Not only can it perform orderings that respect partitioning restrictions, it can take partition boundaries into account. This required the `PseudoDiameter` and `BreadthFirstSearch` classes to be augmented as well.

For instance, the `BreadthFirstSearch` class can start from a root node, or a set of root nodes. It can restrict its search to only those vertices in the same domain as the root node (or nodes). An added feature is that it can loosen this restriction to include nodes that are adjacent to nodes in the same domain.

The `PseudoDiameter` class can take advantage of these partition restrictions and select a pair of vertices that are the farthest apart in their own subdomain.

In a partitioned ordering, we may not know where we want to start numbering vertices in a particular domain, but we want to ensure that all the boundary vertices are numbered last. This can be accomplished with the `SloanEngine` class. We simply mark all the boundary nodes as end nodes. The `SloanEngine` uses a `BreadthFirstSearch` object (which can start from multiple roots) to generate the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

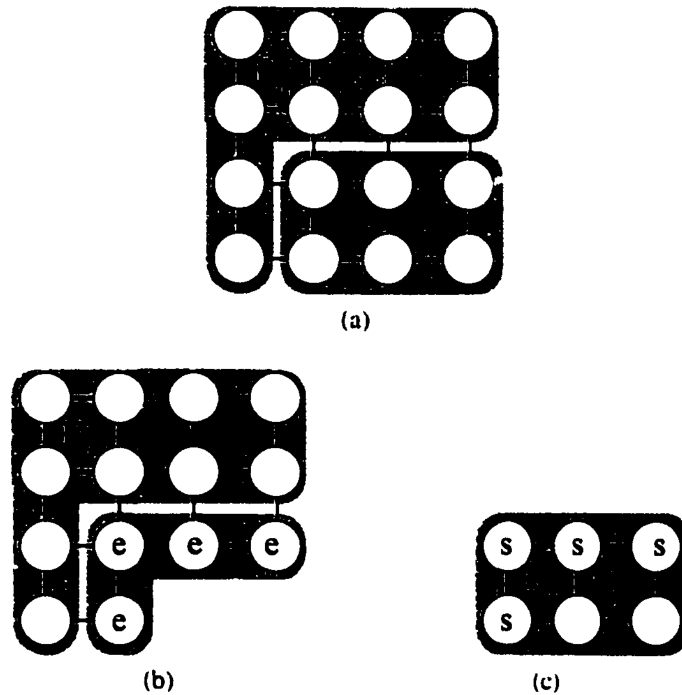


FIG. 43. *Constrained wavefront reduction example. The partitioned graph (a) is broken into two graphs: (b) and (c), which can be ordered independently.*

global component of the priority function, and to find start nodes. Then the ordering object can artificially deflate the initial priority of all the end vertices so low that all other vertices are guaranteed to be numbered before they are.

Another interesting application for these enhanced ordering engines comes from an interface problem in finite elements [71]. We have a mass with linear properties across most of the domain, but there are non-linear physics occuring at one face where it is interacting with another body. This gives rise to a symmetric block system

$$A = \begin{bmatrix} K_{11} & K_{12} \\ K_{12}^T & K_{22} \end{bmatrix}$$

where K_{11} is the linear domain, K_{22} is the non-linear interface. The task now is to generate local orderings on K_{11} and K_{22} that reducing the global wavefront.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Consider the simple example shown in Figure 43(a). After numbering all the vertices in K_{11} the boundary nodes of K_{22} form the wavefront. Since we know this wavefront at one point, we can create two Sloan ordering engines to compute the orderings before and after that wavefront.

The first graph is made up of all the vertices from the K_{11} block of the matrix and boundary vertices of the K_{22} block (see Figure 43(b)). These boundary vertices are set as the end nodes of the wavefront reducing ordering and we set their priority such that we are guaranteed they are numbered last. Then we run the `SloanEngine` on this subgraph, and examine the order in which all the vertices in K_{11} were numbered.

The second graph is strictly made up of all the vertices in the K_{22} block. The boundary vertices are labeled as start nodes, but we do not need to force them to be numbered first. The reason is that once a vertex is in the wavefront, it never leaves until its numbered. We take the order in which these vertices were numbered, and append that to the end of the ordering for the K_{11} block.

Because of the maturity of the `SloanEngine` class, we did not need to make any modifications to the class itself. We needed only to build a simple driver to construct the two subgraphs, orchestrate the two `SloanEngine` classes, and rebuild the total ordering from the two permutations on the subgraphs.

6.3 Summary

The possibilities for further extension abound. Even more important is understanding how extensible the current design is. Now that we have explained "what" changes we made, we examine "how" the changes were made and where the design failed to be extensible.

The easiest example of extensibility is where it was intentionally designed into the code: specifically the `ShrinkingStrategy` hierarchy for the `PseudoDiameter` algorithic class (Section 4.3.2), and the `PriorityStrategy` hierarchy for the `MinPriorityEngine` algorithmic class (Section 4.3.1). However, achieving extensibility where it was expected, and achieving a generally extensible code are two different matters. Here, we will focus on the latter.

For both the unsymmetric Sloan and the column minimum fill algorithms, there were no changes needed to any algorithmic component. Additional classes `HyperGraph` and `ColIntersectGraph`

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

were derived from the `Graph` class, and the `QuotientGraph` needed some modification to handle a special case of initial enodes. New drivers were implemented to extract the results of the algorithm class and reformat it to a solution for an unsymmetric problem. We felt this to be good cases of software extensibility.

For the partitioned fill reducing ordering, we needed rewrite parts of the `MinPriorityEngine` algorithm which orchestrates the interaction between its three main member objects. Adding this functionality was not difficult, but it would have been better if we could simply derive an enhanced algorithmic class. But we are inclined to believe that this special case is probably an intrinsic part of a general algorithmic object. Looking back on the rest of Spindle, we find that several algorithmic objects, including `BreadthFirstSearch`, `PseudoDiameter`, `RCMEngine` and `SloanEngine` all have partition restrictions built-in to the base class. In this case, we should have identified the trend and built partition restrictions into the `MinPriorityEngine` at the beginning. The `QuotientGraph` also needed to be modified to prevent amalgamating indistinguishable supernodes that resided on different subdomains. This was not technically difficult.

Our solution to the block-wavefront reducing ordering algorithm is more of an example of code reuse and flexibility than extensibilty in the strict sense that we defined in Section 2.2. Here we created a collection of `SloanEngine` classes and set them up to do independent problems. The code we wrote to divide the problem, and recombine the partial results from the `SloanEngine` classes is all new. The algorithmic classes themselves are not exhibiting a new behavior, but are being used in a different way.

As we said before, achieving extensibility where it was expected, and achieving a generally extensible code are two very different things. With Spindle, we feel that we have accomplished something in the middle. We are generally pleased with the extensions that we have been able to incorporate efficiently. We also acknowledge that there are practical limits to how far the code can be extended. The extensions presented in this chapter are, after all, only a small subset of all the extensions possible; and were chosen because we felt they had the highest probability of success.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

7. CONCLUSION

Better tools promote better science. And improvments in science allow better tools. While this symbiotic relationship is very obvious in areas such as bio-technology, chemistry, astronomy, and physics, it is often overlooked — or underestimated — in computer science. It is not enough to have either cutting edge algorithms or state-of-the-art software engineering. Today's research requires both.

We spent time analyzing and understanding heuristics for two well-known NP-hard problems. We augmented existing algorithms and in some cases designed new algorithms. We carefully implemented these algorithms in efficient and robust object-oriented software. Then, we used these new tools in extending our knowledge; solving additional problems in new and interesting ways.

The envelope/wavefront reduction problem and its more general form as a sequencing problem has a wide variety of applications: scientific computing, cache performance tuning, spatial databases, and genomics to name a few. Our work has produced asymptotically faster algorithms that reduce the envelope and wavefront better than any other known heuristic. The flexible implementation allows us to generate these orderings on a variety of architectures and integrate with larger pieces of software such as PETSc and Matlab. It also allows us to solve related problems of constrained envelope/wavefront reduction and unsymmetric envelope/wavefront reduction with minimal additions to the existing code.

The fill reduction problem is a classic problem in sparse matrix factorization that has been researched and improved upon for well over 20 years. Our research includes a comprehensive complexity analysis for a family of related heuristics. Our software implements the broadest range of these heuristics for any known implementation, free or commercial. We use the analysis and the software to find weaknesses in the current heuristics and demonstrate a novel polymorphic algorithm that has the potential to adapt dynamically as the ordering progresses. Furthermore, we were able to implement all of this flexible, object-oriented software to execute within a small constant of native Fortran77 code.

There are many additional research projects that readily extend from Spindle. On the algorithmic side, the next logical problem to implement would be a full partitioning package. This could be used on its own or in conjunction with Sloan or MinPriority for various nested dissection orderings. On the software

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

side, because sparse matrix reordering is a service provided to much larger solver codes, and because of Spindle's inherent object-oriented nature, extending Spindle's implementation to include distributed component technologies is tempting.

It is our sincere desire to continue to develop and maintain Spindle for some time to come. It provides many useful services for a wide range of applications and we hope to have it bundled within complete solver packages in the near future. Spindle has proven a useful framework for algorithmic research and a significant contribution to sparse matrix computations.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

REFERENCES

[1] A. AGRAWAL, P. KLEIN, AND R. RAVI, *Cutting down on fill using nested dissection: Provably good elimination orderings.*, in Graph Theory and Sparse Matrix Computation, A. George, J. R. Gilbert, and J. W. H. Liu, eds., vol. 56 of The IMA Volumes in Mathematics and its Applications, Springer-Verlag, 1993, pp. 31–55.

[2] P. AMESTOY, T. A. DAVIS, AND I. S. DUFF, *An approximate minimum degree ordering algorithm*, SIAM J. Mat. Anal. & Appl., 17 (1996), pp. 886–905.

[3] C. ASHCRAFT, *Compressed graphs and the minimum degree algorithm*, SIAM J. Sci. Comput., 16 (1995), pp. 1404–1411.

[4] C. ASHCRAFT AND R. GRIMES, *SPOOLES: An object-oriented sparse matrix library*, in Proceedings of the Ninth SIAM Conference on Parallel Processing, 1999. http://www.netlib.org/linalg/spooles.

[5] C. ASHCRAFT AND J. W. H. LIU, *SMOOTH: A software package for ordering sparse matrices*, November 1996. http://www.cs.yorku.ca/~joseph/SMOOTH.html.

[6] J. E. ATKINS, E. G. BOMAN, AND B. HENDRICKSON, *A spectral algorithm for the consecutive ones problem*, SIAM J. Comput., 28 (1996), pp. 326–337.

[7] S. BALAY, W. D. GROPP, L. C. MCINNES, AND B. F. SMITH, *Petsc: The portable extensible toolkit for scientific computing*. http://www.mcs.anl.gov/petsc.

[8] ——, *Efficient Management of Parallelism in Object-Oriented Numerical Software Libraries*, Birkhauser Press, 1997, pp. 163–202.

[9] S. T. BARNARD, A. POTHEN, AND H. D. SIMON, *A spectral algorithm for envelope reduction of sparse matrices*, J. Numerical Linear Algebra with Applications, 2 (1995), pp. 317–334.

[10] P. BERMAN AND G. SCHNITGER, *On the performance of the minimum degree ordering for Gaussian elimination*, SIAM J. Mat. Anal. & Appl., 11 (1990), pp. 83–88.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[11] J. R. S. BLAIR, P. HEGGERNES, AND J. A. TELLE, *A practical algorithm for making filled graphs minimal*, Theoretical Computer Science A, (2000). To Appear.

[12] R. BOISVERT, R. POZO, K. REMINGTON, R. BARRETT, AND J. DONGARRA, *Matrix market: a web resource for test matrix collections*, in Numerical Software: Assessment and Enhancement, R. Boisvert, ed., Chapman and Hall, London, 1997, pp. 125–137.

[13] E. BOMAN AND B. HENDRICKSON, *Multilevel envelope reduction*, Tech. Report SCCM-96-14, Stanford University, 1996.

[14] C. BORNSTEIN, *Parallelizing and De-parallelizing Elimination Orders*, PhD thesis, Carnegie Mellon University, 1998.

[15] U. V. ÇATALYÜREK AND C. AYKANAT, *Decomposing irregularly sparse matrices for parallel matrix-vector multiplications*, in Lecture Notes in Computer Science, vol. 117, Springer-Verlag, 1996.

[16] T. H. CORMEN, C. E. LEISERSON, AND R. L. RIVEST, *Introduction to Algorithms*, McGraw Hill, 1992.

[17] E. H. CUTHILL, AND J. MCKEE, *Reducing the bandwidth of sparse symmetric matrices*, in Proceed. 24th Nat. Conf. Assoc. Comp. Mach., ACM Publications, 1969, pp. 157–172.

[18] T. A. DAVIS AND I. S. DUFF, *An unsymmetric-pattern multifrontal method for sparse LU factorization*, SIAM J. Mat. Anal. & Appl., 19 (1997), pp. 140–158.

[19] ——, *A combined unifrontal/multifrontal method for unsymmetric sparse matrices*, ACM Trans. on Math. Software, 25 (1999), pp. 1–19.

[20] F. DOBRIAN, G. KUMFERT, AND A. POTHEN, *Object-oriented design for sparse direct solvers*, in Computing in Object-Oriented Parallel Environments, D. C. et. al., ed., vol. 1505 of Lecture Notes in Computer Science, Springer-Verlag, December 1998, pp. 207–214.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[21] ——, *Sparse direct solvers using object-oriented techniques*, in Advances in Software Tools for Scientific Computing, H. P. Langtangen, A. M. Bruaset, and E. Quak, eds., Lecture Notes in Computational Science and Engineering, Springer–Verlag, 1999, pp. 89–131.

[22] I. S. DUFF, A. M. ERISMAN, AND J. K. REID, *Direct Methods for Sparse Matrices*, Clarendon Press, Oxford, 1986.

[23] I. S. DUFF, R. G. GRIMES, AND J. G. LEWIS, *Users' Guide for the Harwell-Boeing Sparse Matrix Collection*, Oct 1992.

[24] I. S. DUFF AND G. A. MEURANT, *The effect of ordering on preconditioned conjugate gradients*, Tech. Report CSS 221, AERE Harwell, Oxon OX11 ORA, United Kingdom, September 1988.

[25] I. S. DUFF AND J. K. REID, *The multifrontal solution of indefinite sparse symmetric linear equations*, ACM Trans. on Math. Software, 9 (1983), pp. 302–325.

[26] I. S. DUFF, J. K. REID, AND J. A. SCOTT, *The use of profile reduction algorithms with a frontal code*, Int. J. for Num. Meths. in Eng., 28 (1989), pp. 2555–2568.

[27] S. C. EISENSTAT, M. C. GURSKY, M. H. SCHULTZ, AND A. H. SHERMAN, *The Yale sparse matrix package I: the symmetric codes*, Internatl. J. Numer. Meths. Engr., 18 (1982), pp. 1145–1151.

[28] M. FIEDLER, *Algebraic connectivity of graphs*, Czechoslovak Math. J., 23 (1973), pp. 298–305.

[29] ——, *A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory*, Czechoslovak Math. J., 25 (1975), pp. 619–633.

[30] E. GAMMA, R. HELM, R. JOHNSON, AND J. VLISSIDES, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Professional Computing Series, Addison Wesley Longman, 1995.

[31] A. GEORGE AND J. W. H. LIU, *An implementation of a pseudoperipheral node finder.*, ACM Trans. on Math. Software, 5 (1979), pp. 284–295.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[32] ——, *A fast implementation of the minimum degree algorithm using quotient graphs*, ACM Trans. on Math. Software, 6 (1980), pp. 337–358.

[33] ——, *A minimal storage impelmentation of the minimum degree algorithm*, SIAM J. Num. Anal., 17 (1980), pp. 282–299.

[34] ——, *The evolution of the minimum degree algorithm*, SIAM Rev., 31 (1989), pp. 1–19.

[35] A. GEORGE AND D. R. MCINTYRE, *On the application of the minimum degree algorithm to finite element systems*, SIAM J. Num. Anal., 15 (1978), pp. 90–111.

[36] A. GEORGE AND A. POTHEN, *Analysis of the spectral approach to envelope reduction via a quadratic assignment formulation*. SIAM J. Mat. Anal. & Appl., 1997.

[37] N. E. GIBBS, *Algorithm 509: A hybrid profile reduction algorithm*, ACM Trans. on Math. Software, 2 (1976), pp. 378–387.

[38] N. E. GIBBS, W. G. POOLE, JR., AND P. K. STOCKMEYER, *An algorithm for reducing the bandwidth and profile of a sparse matrix*, SIAM J. Num. Anal., 13 (1976), pp. 236–249.

[39] ——, *A comparison of several bandwidth and profile reduction algorithms*, ACM Trans. on Math. Software, 2 (1976), pp. 322–330.

[40] J. R. GILBERT, C. MOLER, AND R. SCHREIBER, *Sparse matrices in MATLAB: design and implementation*, SIAM J. Mat. Anal. & Appl., 11 (1992), pp. 333–356.

[41] D. S. GREENBERG AND S. C. ISTRAIL, *Physical mapping with* STS *hybridization:, opportunities and limits*, tech. report, Sandia National Labs, 1994.

[42] B. HENDRICKSON, *Graph partitioning and parallel solvers: Has the Emperor no clothes?*, in Proceedings from Irregular '98, vol. 1457 of Lecture Notes in Computer Science, Springer-Verlag, 1998, pp. 218–225.

[43] B. HENDRICKSON AND T. G. KOLDA, *Partitioning rectangular and structurally nonsymmetric sparse matrices for parallel processing*. To appear in SIAM J. Sci. Comput.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[44] B. HENDRICKSON AND R. LELAND, *The Chaco user's guide: Version 2.0*, Tech. Report SAND94-2692, Sandia National Laboratories, Albuquerque, NM 87815, 1994.

[45] ——, *An improved spectral graph partitioning algorithm for mapping parallel computations*, SIAM J. Sci. Comput., 16 (1995), pp. 452–469.

[46] B. HENDRICKSON AND E. ROTHBERG, *Effective sparse matrix ordering: Just around the BEND*, in Eighth SIAM Conference on Parallel Processing for Scientific Computing, March 1997.

[47] C. JORDAN, *Sur les assemblages de lignes*, J. Reine Angew. Math, 70 (1969), pp. 185–190.

[48] M. JUVAN AND B. MOHAR, *Laplace eigenvalues and bandwidth-type invariants of graphs*. Preprint, Department of Mathematics, University of Ljubljana, Jadranska 19, 61 111, Lubljana, Slovenia, 1990.

[49] ——, *Optimal linear labelings and eigenvalues of graphs*, Discr. Appl. Math., 36 (1992), pp. 153–168.

[50] G. KARYPIS AND V. KUMAR, *MeTiS: An unstructured graph partitioning and sparse matrix ordering system*. `http://www.cs.umn.edu/~karypis/metis/metis.html`.

[51] I. P. KING, *An automatic reordering scheme for simultaneous equations derived from network systems*, Intl. J. Num. Meths. Engr., 2 (1970), pp. 523–533.

[52] G. KUMFERT AND A. POTHEN, *Two improved algorithms for envelope and wavefront reduction*, BIT, 37 (1997), pp. 559–590.

[53] ——, *An object-oriented collection of minimum degree algorithms*, in Computing in Object-Oriented Parallel Environments, D. C. et. al., ed., vol. 1505 of Lecture Notes in Computer Science, Springer, December 1998, pp. 207–214.

[54] J. LAKOS, *Large-Scale C++ Software Design*, Professional Computing Series, Addison-Wesley, 1996.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[55] J. G. LEWIS, *Implementations of the Gibbs-Poole-Stockmeyer and Gibbs-King algorithms*, ACM Trans. on Math. Soft., 8 (1982), pp. 180 – 189.

[56] X. S. LI, *Sparse Gaussian Elimination on High Performance Computers*, PhD thesis, University of California at Berkeley, 1996.

[57] R. J. LIPTON AND R. E. TARJAN, *A separator theorem for planar graphs*, SIAM J. Appl. Math., 36 (1979), pp. 177–189.

[58] J. W. H. LIU, *Modification of the minimum-degree algorithm by multiple elimination*, ACM Trans. on Math. Software, 11 (1985), pp. 141–153.

[59] J. W. H. LIU AND A. H. SHERMAN, *Comparative analysis of the Cuthill-Mckee and the reverse Cuthill-Mckee ordering algorithms for sparse matrices*, SIAM J. Num. Anal., 13 (1976), pp. 198–213.

[60] S. MEYERS, *More Effective C++: 35 New Ways to Improve your Programs and Designs*, Addison-Wesley, 1996.

[61] ——, *Effective C++: 50 Specific Ways to Improve your Programs and Designs*, Addison-Wesley, 2nd ed., 1998.

[62] G. L. MILLER, S.-H. TENG, W. THURSTON, AND S. A. VAVASIS, *Automatic mesh partitioning*, in Graph Theory and Sparse Matrix Computation, A. George, J. R. Gilbert, and J. W. H. Liu, eds., Springer Verlag, 1993, pp. 57–84. The IMA Volumes in Mathematics and its Applications, Volume 56.

[63] D. R. MUSSER AND A. SAINI, *STL Tutorial and Reference Guide*, Addison-Wesley, 1996.

[64] E. G. NG AND P. RAGHAVAN, *Performance of greedy ordering heuristics for sparse Cholesky factorization*. Submitted to SIAM J. Mat. Anal. & Appl., 1997.

[65] S. PARTER, *The use of planar graphs in Gaussian Elimination*, SIAM Rev., 3 (1961), pp. 364–369.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[66] G. H. PAULINO, I. F. M. MENEZES, M. GATTASS, AND S. MUKHERJEE, *Node and element resequencing using the Laplacian of a finite element graph, Part I*, International Journal for Numerical Methods in Engineering, 37 (1994), pp. 1511–1530.

[67] ——, *Node and element resequencing using the Laplacian of a finite element graph, Part II*, International Journal for Numerical Methods in Engineering, 37 (1994), pp. 1531–1555.

[68] A. POTHEN, H. D. SIMON, AND K. P. LIOU, *Partitioning sparse matrices with eigenvectors of graphs*, SIAM J. Mat. Anal. & Appl., 11 (1990), pp. 430–452.

[69] A. POTHEN, H. D. SIMON, AND L. WANG, *Spectral nested dissection*, Tech. Report CS-92-01, Computer Science, Pennsylvania State University, University Park, PA, 1992. Also NASA Ames Research Center Report RNR-092-003.

[70] A. POTHEN, S. YE, AND J. FU, *Enhancing the cache performance of irregular computations by reordering data accesses*. (in preparation), 2000.

[71] L. V. QUOC AND J. R. O'LEARY, *Automatic node resequencing with constraints*, Computers and Structures, 18 (1984), pp. 55–69.

[72] J. K. REID AND J. A. SCOTT, *Ordering symmetric sparse matrices for small profile and wavefront*, Tech. Report RAL-1998-016, Rutherford Appleton Laboratory, 1998.

[73] E. ROTHBERG, *Ordering sparse matrices using approximate minimum local fill*. Preprint,Silicon Graphics Inc., Mountain View CA, April 1996.

[74] E. ROTHBERG AND S. C. EISENSTAT, *Node selection strategies for bottom-up sparse matrix ordering*, SIAM J. Mat. Anal. & Appl., 19 (1998), pp. 682–695.

[75] S. SHEKHAR, S. CHAWLA, S. RAVADA, A. FETTERER, X. LIU, AND C.-T. LU, *Spatial databases: Accomplishments and research needs*, IEEE Trans. on Knowledge and Data Eng., 11 (1999), pp. 45–55.

[76] S. W. SLOAN, *An algorithm for profile and wavefront reduction of sparse matrices*, International Journal for Numerical Methods in Engineering, 23 (1986), pp. 239–251.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[77] A. STEPANOV AND M. LEE, *The Standard Template Library*, Palo Alto, CA 94304, Oct. 1995.

[78] B. STROUSTRUP, *The C++ Annotated Reference Manual*, Addison-Wesley, 3 ed., 1997.

[79] L. WANG, *Spectral Nested Dissection*, PhD thesis, The Pennsylvania State University, August 1994.

[80] W. WATTHAYU, *Cache-friendly algorithms in scientific computing*, master's thesis, Old Dominion University, 1999.

[81] M. YANNAKAKIS, *Computing the minimum fill-in is NP-complete*, SIAM J. Alg. & Disc. Meth., (1981), pp. 77–79.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

INDEX

active columns, **154**
active rows, **154**
adjacency graph, *see* graph, adjacency-
algebraic
 ordering algorithms, 38
  combinatorial hybrid, 1, 40
`Algorithm` class, 92
AMD, 1, **67**, **77–80**, 65–90, 119, 122, 135–148
 implementations, *see* AMDBAR, SPOOLES, or Spindle
AMDBAR, **135**, 137, 138
 algorithm, *see* AMD
AMF, **67**
 priority function, **69**
AMIND, **67**
 priority function, **69**
AMMF, **67**
 priority function, **69**
anisotropy, 55, 57
Approximate Minimum Degree, *see* AMD
Approximate Minimum Fill, *see* AMF
Approximate Minimum Increase in Neighbor Degree, *see* AMIND
Approximate Minimum Mean Local Fill, *see* AMMF
ArraySloan, 49–52

bandwidth, 22, 23, 37, 46, 48, 130–132
 reduction, 23, **23**
 symmetric, **22**
bandwith
 unsymmetric, **153**
BEND, 148
BFS, *see* search, breadth-first
`BinaryHeap` class, **98**
breadth-first-search, *see* search, breadth-first
`BreadthFirstSearch` class, **107**, 157, 160
`BucketSorter` class, **98**, 121, 122

CFD, *see* computational fluid dynamics
CG, *see* conjugate gradient methods
Chaco, 117, 127
Cholesky, *see* factorization, Cholesky
`ClassMetaData` class, 93, 94, 99
clique, **7**
`ColIntersectGraph` class, 159
`ColIntersectionGraph` class, 151
column height
 unsymmetric, **152**
column intersection graph, *see* graph, column intersection-combinatorial
 ordering algorithms, 1, 37, 81
  algebraic hybrid, 1, 40
 refinement algorithm, 40
Common Object Request Broker Architecture, *see* CORBA
`CompressionMap` class, **97**, 99, 107
computational fluid dynamics
 problems, 128
conjugate gradient methods, 55, 56
CORBA, 100

`DataStream` class, 116, 118
`DataStruct` class, 96
`DataStructure` class, 92, 94, 95, 97–99, 104, 116
DCOM, 100
diameter
 pseudo, *see* pseudo-diameter
Distributed Component Object Model, *see* DCOM

EJB, 100
`EliminationForest` class, **97**, 106, 107
Enterprise Java Beans, *see* EJB
envelope, **22**
 unsymmetric, **153**

factorization
 Cholesky, 23, 54, 55, **60**, 153
  incomplete, 54
 LU, 153, **153**, **154**
 symmetric, *see* factorization, Cholesky
 unsymmetric, *see* factorization, LU
factorization, Cholesky, incomplete, 55–57
Fiedler vector, **38**, 54, 127, **127**
finite-element, 39, 83

Gaussian elimination
 variants
  symmetric, *see* factorization, Cholesky
`GenericForest` class, **97**
GENMMD, **135**, 136, 138
 algorithm, *see* MMD
Gibbs-King, *see* GK
Gibbs-Poole-Stockmeyer, *see* GPS
GK, 16
GPS, 16
`Graph` class, 95, 97, **97**, 98, 107, 109, 111, 112, 116, 117, 125, 151, 155, 159
graph
 adjacency-, **4–5**

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

bipartite-, 155, 156
column intersection-, 4, **5–6**, 150
compression, **6–7**
diameter, **16**
directed, 8
distance, **16**, 17, 18, 38, 41, 44, 47, 53, 108
hyper-, 4, **6**, 150, 151, 155
index, 53, 58
of matrix, **4**
quotient
unsymmetric, 152
undirected, 4, 6, 7, 22, 24, 38, 60
undirected edges, 8
unweighted, 38
`GraphBase` class, 97, **97**, 116–118
`GraphCompressor` class, **107**

Harwell Sparse Library, 49, 55, 135
Harwell-Boeing Sparse Matrix Collection, 57, 113
HeapSloan, 49–52
`HyperGraph` class, 151, 155, 156, 159
hypergraph, *see* graph, hyper-

IC, *see* factorization, Cholesky, incomplete
inodes, 7

level set, **17**
level structure, **17**
height, **17**
width, **18**
LU, *see* factorization, LU

`MapUtils` class, **98**, 99
Matlab, 13, 115, 150, 161
matrices
rectangular, 150
structurally symmetric-, 153
`Matrix` class, **98**, 99, **100**, 113, 114, 116, 117
matrix
format
column-major, 115
row-major, 115
graph of, **4**
unsymmetric, 5, 6
sparse, **3**
storage
column-major, 14
row-major, 14
structure, **3**
Matrix Market Library, 113, 116, 117
`MatrixBase` class, **98**, 114, 116–118
`MatrixFile` class, 113, 114
`MatrixMarketFile` class, 116

`MatrixUtils` class, **98**, 99
MC40, 49, 50
MC60, 50
MD, 1
Message Passing Interface, *see* MPI
`MetaData` class, 98
Metis, 148
`MinimumPriorityEngine` class, 119
`MinimumPriorityStrategy` class, 124, 125
`MinPriorityEngine` class, **108**, 119, 151, 152, 156, 159, 160
MMD, 1, **67**, 65–90, 119, 122, 135–148
implementations, *see* GENMMD, SPOOLES, or Spindle
MMDF, **67**
priority function, **69**
MMMD, **67**
priority function, **69**
Modified Minimum Deficiency, *see* MMDF
Modified Multiple Minimum Degree, *see* MMMD
MPI, 100
Multiple Minimum Degree, *see* MMD

node, *see* vertex
NP-complete, 39, 61
NP-hard, 1, 161

operating systems
Linux, 91
Solaris, 91
WindowsNT, 91
`OptionDatabase` class, **109**
ordering
fill-reducing, 1, 5, 7, **58–90**, 108, 151
results, 133–148
wavefront-reduction, **21–58**
`OrderingAlgorithm` class, **107**, 108, 109, 156

path
undirected, 62
permutation
column, 150–152
unsymmetric, 149–156
`PermutationMap` class, 95, 97, **97**, 98–103, 106, 107
PETSc, 7, 14, 91, 115, 161
Portable Extensible Toolkit for Scientific Computing, *see* PETSc
preactive nodes, 37
preconditioning

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

incomplete Cholesky, *see* factorization, Cholesky, incomplete
PriorityQueue class, 147
PriorityStrategy class, 120, 133, 138, 144, 147, 159
pseudo-diameter, 16–21
PseudoDiameter class, **107**, 108, 117, 119, 157, 159, 160

QuotientGraph class, 94, **98**, 120–124, 126, 144, 151, 152, 159, 160

RCM, 16, 23, 25, 26, 38, 40, 49, 50, 55, 57, 108, 127–132
RCMEngine class, **108**, 160
Reverse Cuthill-McKee, *see* RCM
row width, **22**
unsymmetric, **152**

ScatterMap class, **97**
search
breadth-first-, 17, 18, 20
asymptotic complexity, **17**
ShinkingStrategy class, 107
short-circuit, 20
shrinking strategy, 18, **20**, 21, 117
ShrinkingStrategy class, 117, **117**, 159
Sloan, 16, 26, **38**, 127–133
enhanced, 21
implementation
array based, *see* ArraySloan
Harwell Sparse Library, *see* MC40 or MC60
heap based, *see* HeapSloan
normalized weights, 128–130
refinement, 21
Sloan's Algorithm
aymptotic complexity, 51–52
Sloan:Fast-, 129
SloanEngine class, **108**, 152, 155–160
sort
bucket, 98
sparse
matrix, **3**
spectral ordering, **38–39**
Spindle, iii, 1, 14, 15, 66, 92, 94, 95, 99, 104, 106, 109, 111, 113, 117, 119, 133–135, 138, 150, 157, 160–162
SpindleAlgorithm class, 104–107, 122
SpindleArchive class, 100
SpindleBaseClass class, 92–94, 98–100
SpindleSystem class, 99
SPOOLES, **135**, 136–138
Standard Template Library, *see* STL
STL, 91
Strategy Pattern, 117, 119
supernode, *see* vertex
Switcher class, 144, 147
SymbolicFactorization class, **107**

TCP/IP, 100

vertex
degree
approximate, 65
exact, *see* vertex, degree
distance, *see* graph, distance
indistinguishable, **6**, **64**, 65, 67, 68, 73, 74, 76, 79–81, 83, 84, 139, 144, 147, 157, 160

wavefront
column, **154**
reduction
blockwise, 149, 157–160
row, **154**, 155
unsymmetric, **154**

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

VITA

Gary Karl Kumfert was born and (with the exception of two years) raised in Philadelphia, Pennsylvania. In August 1989, fresh out of high school, he moved to Norfolk, Virginia with a full tuition academic scholarship to Old Dominion University. During his undergraduate career, he was recognized for distinguished service to the university and the community, and for demonstrated leadership among his peers. He earned a Bachelors of Science in Applied Mathematics with a minor in Computer Science in Spring 1993.

Having maintained a 4.0 GPA in his computer science classes as an undergraduate, Gary was approached by the Computer Science Department and offered an assistantship in the Ph.D. program. He started in Fall 1993. During his time at Old Dominion University, Gary has done an internship at ICASE at NASA Langley, served as President of the Graduate Student Association, and was awarded a GAANN Fellowship (Graduate Assistantships in Areas of National Need) funded by the Department of Education. The Department of Computer Science can be contacted directly by mail or telephone, or email.

Department of Computer Science
Old Dominion University
Norfolk, VA 23529
(757) 683-3915

Immediately after his dissertation defense in August 1999, Gary joined the Center for Applied Scientific Computing (CASC) in Lawrence Livermore National Laboratory.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.