THE UNIVERSITY OF FLORIDA SPARSE MATRIX COLLECTION*

TIMOTHY A. DAVIS†

Abstract. The University of Florida Sparse Matrix Collection is a large, widely available, and actively growing set of sparse matrices that arise in real applications. Its matrices cover a wide spectrum of problem domains, both those arising from problems with underlying 2D or 3D geometry (structural engineering, computational fluid dynamics, model reduction, electromagnetics, semiconductor devices, thermodynamics, materials, acoustics, computer graphics/vision, robotics/kinematics, and other discretizations) and those that typically do not have such geometry (optimization, circuit simulation, networks and graphs, economic and financial modeling, theoretical and quantum chemistry, chemical process simulation, mathematics and statistics, and power networks). The collection meets a vital need that artificially-generated matrices cannot meet, and is widely used by the sparse matrix algorithms community for the development and performance evaluation of sparse matrix algorithms. The collection includes software for accessing and managing the collection, from MATLAB, Fortran, and C.

Key words. sparse matrices, performance evaluation and testing of sparse matrix algorithms.

AMS subject classifications. 65F50, 65-04.

1. Introduction. Wilkinson defined a sparse matrix as *any matrix with enough zeros that it pays to take advantage of them* [42]. Wilkinson seems to have never published this definition in writing. According to Bunch [17], Wilkinson stated this definition as early as 1969. This author has not found any printed versions of this definition earlier than those of Duff [29] and Reid [62], who do not cite Wilkinson and elaborate more fully on the definition. It may have arisen independently. What is certain is that by the early 1970's, it was the common understanding in the sparse matrix community that defining a sparse matrix as a matrix with some fraction of nonzero entries was inappropriate. Instead, it was recognized very early that sparsity is an economic issue; if you can save time and memory by exploiting the zeros, then the matrix is sparse.

Here, the University of Florida Sparse Matrix Collection is described. Section 2 gives the motivation for collecting sparse matrices from real applications and making them widely available. Section 3 describes the current state of the collection: where the matrices come from, how they are stored, and what additional auxiliary information is available. In Section 4, a suite of widely-available software packages is described for accessing and managing the collection (including software that generates matrix statistics and web pages). Examples from the many uses of the collection in previous literature, plus one new one, are given in Section 5. Section 6 concludes with instructions on how to submit new matrices to the collection.

2. Purpose of the collection. The role of sparse matrices from real applications in the development, testing, and performance evaluation of sparse matrix algorithms has long been recognized. The first established collection was started by Curtis and Reid (1970 and following), extended by Duff and Reid (1979, [36]), and then compiled into the Harwell-Boeing collection by Duff, Grimes, and Lewis (1989, [32]). This collection provides the starting point of University of Florida Sparse Matrix Collection. Since then, additional matrices have been added over the years, and

* This work was supported by the National Science Foundation under grants 9111263, 9223088, 9504974, 9803599, 0203720, and 0620286.

†Department of Computer and Information Science and Engineering, University of Florida

other collections have been made, many of which have also been incorporated into the UF Collection (such as [7, 8, 9, 12, 13, 40, 44, 54, 55, 58, 63, 64, 65, 67, 69]).

To ensure ease of access and allow for repeatable experimental design, the matrices in the collection are indexed and stored in a uniform manner, and software is available for reading the matrices, creating the matrices, and managing the collection. A simple download mechanism is provided so that a single MATLAB statement can download a matrix and load it directly into the MATLAB workspace.

2.1. Random matrices. Maintaining a sparse matrix collection is not a trivial task, nor is it trivial to download an 8GB data set (the size of the collection as of 2007 in just one of its three available data formats). It is much simpler to generate random matrices on the fly, and to use those for the testing and performance evaluation of sparse matrix algorithms. The results obtained could even be repeatable, with the creation of software that generates a repeatable and parameterized sequence of random sparse matrices.

While simple, this approach is unsuitable. Random matrices (or more precisely pseudorandom matrices) have several characteristics that limit their use for testing and performance evaluation [47].

Under modest assumptions, random n-by-n matrices with $O(n)$ nonzero entries require $O(n^3)$ time and $O(n^2)$ memory to factorize, because of catastrophic fill-in [28]. In other words, depending on the constants in the big-O notation, it is better to use a dense matrix algorithm to factorize a random sparse matrix. Random sparse matrices are not sparse, according to Wilkinson's definition. Even if sparse matrix techniques are useful, this catastrophic fill-in will tend to introduce a bias in the performance evaluation of different sparse matrix methods, towards those that can exploit factors with very large and very dense lower right submatrices. A sparse matrix algorithm should ideally operate in time proportional to the number of floating-point operations (a goal that seems deceptively easy but is actually hard to accomplish). If a poorly written sparse matrix algorithm included an $O(n^2)$ time component, even with a small constant, this poor behavior would be masked if only random sparse matrices were to be used for performance comparisons with well-written algorithms.

No sparse matrix from real applications exhibits this catastrophic-fill-in characteristic. All sparse matrices arising in real applications have some kind of structure in their nonzero pattern which can be profitably exploited by fill-reducing orderings. Although sparse matrices from different applications "behave" differently from each other (in terms of amount of fill-in from different orderings, and relative performance of different factorization methods, for example), none of them behave anything like random sparse matrices.

Random dense matrices are nonsingular (with very high probability) and tend to be very well conditioned [38]. The structural rank of a sparse matrix is the maximum number of nonzero entries that can be permuted to the diagonal (`sprank(A)` in MATLAB), or equivalently, the size of a maximum matching of the bipartite graph of the matrix. It is an upper bound on the numerical rank. In perfect arithmetic, for every fixed sparsity pattern S, the numerical rank and structural rank of a random sparse matrix with pattern S are equal, with probability one [41].[1] Random sparse matrices also have very poor graph separators when partitioned for parallel computing (which affects both direct and iterative parallel methods).

[1] See also [16], which has most of the theorems needed to show this.

Catastrophic fill-in in direct methods, and the well-conditioned property of random sparse matrices, introduces an artifical bias towards iterative methods.

Although random matrices are completely meaningless when comparing the performance of two sparse matrix algorithms, they do find some use in the testing of sparse matrix codes since they are so simple to generate on the fly. However, even this use has limitations. Backslash (`x=A\b`) in MATLAB, for example, must detect both structural and numerical rank deficiency. Many matrices arising in practice have structural and numerical ranks that differ. Code that considers this case cannot be tested with random sparse matrices. The numerical accuracy of pivoting strategies also cannot be reliably tested with random matrices [57].

2.2. Sparse matrices from real applications. Since random matrices are unsuitable, there is a need for a collection of matrices from real applications. The ideal test suite would be somehow representative of matrices that arise in practice. It should cast a broad net, so as to capture matrices from every application domain that relies on sparse matrix methods. For example, matrices arising in circuit simulation (and other network-related problems) differ greatly from matrices arising from the discretization of 2D and 3D physical domains. Computational fluid dynamics matrices differ from structural engineering matrices, and both are vastly different from matrices arising in linear programming or financial portfolio optimization. The collection should be kept up to date, since matrices of interest grow in size each year as computer memories get larger. New application domains also appear, such as eigenvalue problems arising in web connectivity matrices [49, 60] which have existed only since the mid 1990's.

Sparse matrix algorithm developers use these matrices to develop their methods, since theoretical asymptotic time/memory complexity analysis only goes so far. If there are no matrices available to developers from a given application domain, it is quite possible that when their methods are used in that domain, the performance results will be disappointing. This provides a strong motivation for computational scientists to submit their matrices to a widely available collection.

There are two strategies for constructing such a collection; any collection will tend to be driven by a combination of the two strategies. The first is the "trophy case" strategy, which seeks the unusual or interesting matrices, and perhaps a few "typical" matrices from each application area. Duplicate matrices (those with characteristics similar to matrices already in the collection) would be avoided.

The second strategy is a "statistical sampling" of the domains that generate sparse matrix problems. In this strategy, narrow application areas such as DNA electrophoresis would have few matrices in a collection, while broad and widely used applications (finite element methods for 2D and 3D physical domains, for example) would have many matrices in a collection. In this strategy, duplicates (although not exact duplicates) would be welcome. The UF Collection was constructed along this second strategy, with some "trophy case" matrices added as well. Matrices from submitters are never turned away for being near duplicates of matrices already in the collection. Some problem sets that are too small have been declined, although an application area with a range of matrix sizes from very tiny to very large is very useful. The smaller matrices are well suited for testing, and the larger ones for performance evaluation.

Both strategies introduce a source of bias. For example, whenever this author is queried by an end-user about his software (direct methods for solving sparse linear systems, not iterative methods), a request is made for one or more matrices for inclusion in the collection. Thus, application areas that are more suitable for iterative

methods may be under-represented in the collection.

One method used to avoid this bias is to rely on matrices collected by others. The UF Collection includes many sets of matrices in this form, such as those collected by Saad, who works on iterative methods for sparse linear systems [68]. In addition, not all matrices in the collection describe a sparse linear system. For example, many linear programming problems appear in widely available collections in MPS format [40, 55, 58, 63]; they have been converted into a standard matrix format (minimize $c'x$ subject to $Ax = b$ and $l \leq x \leq u$) and included in the UF collection.

2.3. Matrix generators. A matrix generator is a software package that generates matrices typical of what might arise in a real application, in any desired size. These are particularly useful for parallel sparse matrix algorithms, where the problem size should scale with the number of processors when scaled speedup is being measured. The Matrix Market includes several matrix generators, including the LAPACK test matrix generators [6], the NEP matrix generators [7, 8], and Higham's Matrix Computational Toolbox [47] (part of which appears as the `gallery` function in MATLAB).

However, there are no matrix generators in the UF Collection in its current form, for two reasons. The existing generators such as the ones described above are well-written, easy to understand, and easy to maintain. However, they generate matrices with very regular nonzero patterns. Matrices that arise in real applications tend to have a more complex structure. They require complete applications to generate them, but these software packages are dauntingly large and would be impossible for this author to maintain.

3. Description of the collection. As of January, 2007, the UF Sparse Matrix Collection consists of 1840 *problem sets*, each of which contains at least one sparse matrix (typically just one). The structure of a *problem set* is described in detail in Section 3.2. Most frequently, it represents a sparse linear system of the form $Ax = b$, where b is often provided as well as the sparse matrix A. In some cases, a problem set consists of a sequence of closely related matrices. Some problem sets represent a problem with two sparse matrices, such as a stiffness matrix and a mass matrix from a structural engineering eigenvalue problem, for example. In this case, A is the stiffness matrix, and the mass matrix appears as an auxiliary matrix in the problem set. Counting all of the matrices in these sequences, and all sparse auxiliary matrices (such as mass matrices, but excluding sparse vectors), the collection includes 2425 matrices. The web site for the collection is http://www.cise.ufl.edu/research/sparse/matrices.

It is important that a collection include a large number of matrices. For example, an experiment was recently performed on a variant of the diagonal Markowitz ordering scheme [4], using the UMFPACK data structure for representing the active submatrix [20, 22, 23]. Unlike the quotient graph used in the minimum degree ordering algorithm, the size of the data structure is not guaranteed to be limited by the size of the initial structure of the matrix being ordered. However, only a *single* matrix in the collection triggered the case where the size of the data structure during elimination exceeded the initial size (by 3%). This particular matrix was not added to the collection because it exhibited this property; it was already in the collection. This result also explains why the UMFPACK data structure requires so little integer memory space in practice.

3.1. Application areas. The UF collection is divided into 99 different matrix groups, with more groups added as new matrices are submitted to the collection. A complete list of these groups is too long to include here; details are given on the

TABLE 3.1
Partial list of problem sources

Non-Hermitian Eigenvalue Problems	Bai et al. [7, 8]
Pajek Networks	Batagelj and Mrvar [9]
Multistage stochastic financial modeling	Berger et al. [11]
The Matrix Market	Boisvert et al. [12, 13]
Univ. of Utrecht circuit simulation matrices	Bomhof and van der Vorst [14]
Harwell-Boeing Collection	Duff, Grimes, and Lewis [31, 32]
Frequency domain, nonlinear analog circuits	Feldmann et al. [39]
NETLIB Linear Programming Test Problems	Gay [40]
Symmetric sparse matrix benchmarks	Gould, Hu, and Scott [44]
Linear programming problems	Gupta [45]
Stanford/Berkeley Web Matrices	Kamvar [49]
Xyce circuit simulation matrices	Hoekstra [51]
Computer vision problems	Kemelmacher [52]
Parasol Matrices	Koster [54]
Linear programming test set	Mészáros [55]
2D and 3D semiconductor physics	Miller and Wang [56]
Linear programming test set	Mittelman [58]
QAPLIB, quadratic assignment problems	Resende [63]
Oberwolfach Model Reduction Benchmarks	Rudnyi et al. [64, 65]
SPARSKIT matrix collection	Saad [67]
Univ. of Basel Collection	Schenk [69]
MRI reconstruction	M. Bydder, UCSD; see [71]
DNA electrophoresis matrices	van Heukelum et al. [72]
Chemical engineering problems	Zitney [73, 74]

collection's web site. Each group consists of either a set of matrices from the same application and source, or a set of matrices from another sparse matrix collection. In the latter case, the group may consist of matrices from very different application areas (for example, the Harwell-Boeing collection forms a single group).

Sources (papers and web sites) for some of the matrix groups are listed in Table 3.1. The UF Collection includes many more matrices submitted to the collection from their authors which do not include a citable paper or web site. This list gives a flavor of the range of problems the collection contains.

A complete list of the problem domains for all 1840 matrices is given in Table 3.2. The table is split into three categories: problems with no underlying geometry, problems with 2D or 3D geometry, and counter-examples for which the geometry is unclear (at least 3 of them certainly have no 2D/3D geometry).

3.2. Matrix problem characteristics. Each problem set contains the following items:

- `name`: the problem group and name, such as `HB/bcsstk13` for the `bcsstk13` matrix in the Harwell-Boeing group.
- `title`: a short, one-line, descriptive title.
- `A`: an m-by-n sparse matrix; binary, integer, real, or complex.
- `id`: an integer in the range 1 to the number of problems in the collection. Once a problem is assigned a serial number, it is never changed.
- `date`: the year the matrix was created, or added to the collection (or another collection, such as the Harwell-Boeing collection) if the creation date is unknown. This has been left blank for 42 matrices in the NETLIB LP set [40], since it is unclear from the documentation when they were created or added to that collection.

TABLE 3.2
Problem domains

1140	**problems with no 2D/3D geometry**
342	linear programming
252	circuit simulation
126	optimization (excluding linear programming problems)
110	graph problems (excluding random graphs)
70	chemical process simulation
68	random graphs
67	economic/financial modeling
61	theoretical/quantum chemistry
22	least squares, statistics, mathematics
22	power networks
692	**problems with 2D/3D geometry**
271	structural engineering
159	computational fluid dynamics
84	other (various ODE's, PDE's, etc.)
38	model reduction
36	electromagnetics
34	semiconductor devices
25	thermal problems
13	materials
12	acoustics
7	computer graphics/vision
3	robotics, kinematics
8	**counter-examples**

- `author`: the name or names of the originator(s) of the matrix. This is left blank if unknown (200 matrices).
- `ed`: the name or names of the editor(s) or collector(s) of the matrix, who first included it in a widely available collection.
- `kind`: a one-line string describing the type of the problem, selected from a well-defined set and thus suitable for parsing by a program. Table 3.2 is a slightly condensed version of the list of possible values of this item. This field is always non-empty.

The following items are optionally present:

- `Zeros`: a binary matrix whose pattern gives the explicit zero entries provided by the matrix author. MATLAB drops these entries from any sparse matrix, and thus they are stored separately in that data format. In the Rutherford-Boeing and Matrix Market format, these entries are stored with `A` itself.
- `b`: the right-hand side of the linear system $Ax = b$; may be any size, sparse or dense, and real or complex.
- `x`: the supposed solution of the linear system $Ax = b$; may be any size, sparse or dense, and real or complex.
- `notes`: notes about the problem, as one or more lines of text.
- `aux`: an auxiliary structure, containing arbitrary problem dependent information. Each item in `aux` must be a matrix or a sequence of matrices (stored as a cell array in MATLAB). The matrices in `aux` can be of any type (real, complex, or character, and either full or sparse). Typical contents include cost vectors and bounds for linear programming problems, node names for network problems (such as URL's), additional matrices for model reduction and eigenvalue problems (mass matrices, for example), and 2D or 3D coordinates of the nodes of the graph of `A`.

Each problem set is held in three different formats, all of which contain the same information. In the MATLAB format, the problem set is a single MATLAB variable, stored as a `struct` and held in a single MAT-file. A problem can be downloaded directly into MATLAB with either of the statements

```
Problem = UFget (35)
Problem = UFget ('HB/bcsstk13')
```

where the problem `HB/bcsstk13` has a serial number of 35. The printed MATLAB output of these statements is given below.

```
Problem =
 title: 'SYMMETRIC STIFFNESS MATRIX, FLUID FLOW GENERALIZED EIGENVALUES'
     A: [2003x2003 double]
  name: 'HB/bcsstk13'
    id: 35
  date: '1982'
author: 'J. Lewis'
    ed: 'I. Duff, R. Grimes, J. Lewis'
  kind: 'computational fluid dynamics problem'
```

Once a problem MAT-file is downloaded, `UFget` caches it on the user's local system, so that any one matrix need only be downloaded once. A future version of Mathematica will include an interface to the collection [48].

The `UFget(35)` usage facilitates the creation of specialized subsets of the collection for the design of experiments, as lists of matrix `id`'s. An index is provided that contains statistics about each matrix, such as dimensions, number of nonzeros, symmetry, positive definiteness, structural rank, and results from three fill reducing orderings: minimum degree (AMD [1] or COLAMD [24]), nested dissection (METIS [50]), and a Dulmage-Mendelsohn permutation [30, 37, 61] followed by a best-effort ordering of each diagonal block. For example, suppose we wish to test all structurally non-singular square matrices, in order of problem size as determined by the number of nonzeros in the factors as found by AMD. In MATLAB, this is done with the following simple code, which downloads the matrices as needed.

```
index = UFget ;
list = find (index.nrows == index.ncols & index.sprank == index.nrows) ;
[ignore i] = sort (index.amd_lnz (list)) ;
list = list (i) ;
for k = list
    Problem = UFget (k)
    A = Problem.A ;
    % ... run an experiment on the matrix A
end
```

As another example, the complete experiment to determine the AMD ordering times in Section 5 required about 80 lines of MATLAB, including plotting of the results.

Each problem is also stored in the Rutherford-Boeing format [31, 33] and in the Matrix Market format [12, 13]. In both formats, each problem set is stored as one or more files in a single directory, including all meta-data about each problem set (all the items listed above in the MATLAB problem `struct`).

The Rutherford-Boeing format is a successor to the Harwell-Boeing format [31, 32]. It stores a sparse matrix in compressed column form. Unlike the Harwell-Boeing format, files in the Rutherford-Boeing format can be read using any programming language, not just Fortran. The Matrix Market format stores a matrix in triplet format (also called the coordinate format), where each line of the file includes the row

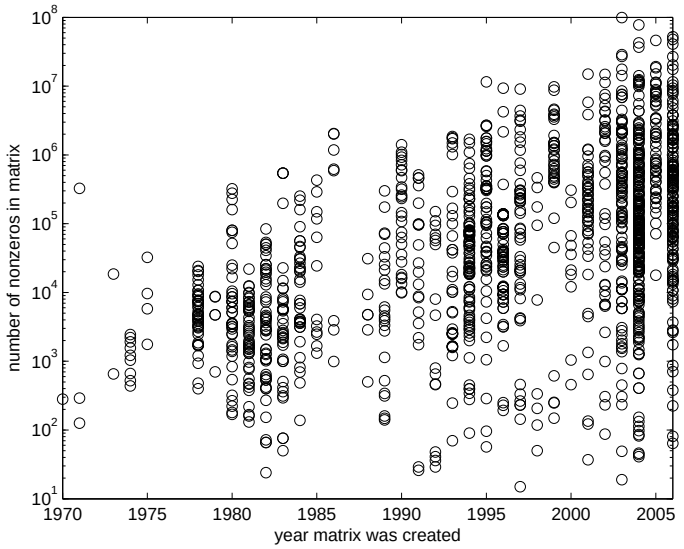

FIG. 3.1. *Matrix size, broken down by year matrix was created*

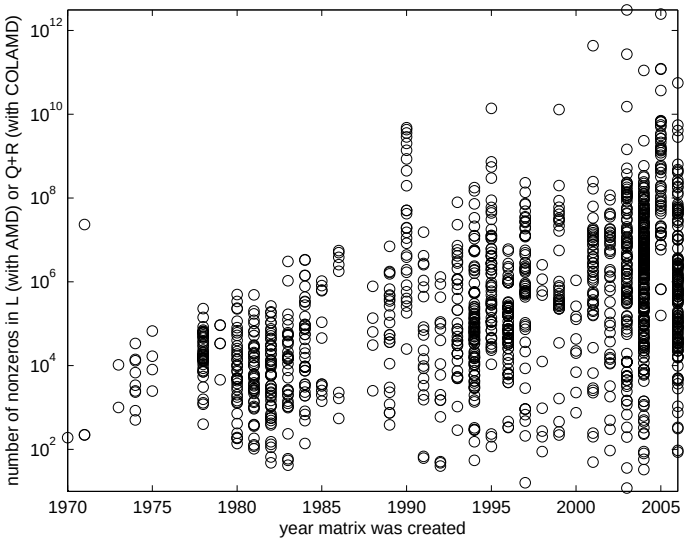

FIG. 3.2. *Matrix factor size, broken down by year matrix was created*

index, column index, and numerical value of a single entry.

3.3. Matrix statistics. The matrices in the collection come from 239 different authors and 49 different editors/collectors. The dates the matrices were created range from 1970 to 2006, with new matrices added each year (except for 1972, 1976, 1977, and 1987). Figures 3.1 and 3.2 plot the number of nonzeros in each matrix and the number of nonzeros in the factors (the Cholesky factor L of $P(A+A^T)P^T$ with AMD

if square, or the Householder-vector representation of the QR factorization of AP or A^TP with COLAMD otherwise[2]), respectively, as a function of the year the matrix was created. These plots show an exponential growth in problem sizes, similar to how computer memory sizes have grown since 1970. Note that small problems are still being actively added to the collection. This is because a matrix group often includes a range of related problems, from small test cases to the largest problems of interest.[3]

4. Software for accessing and managing the collection. All of the software for managing the collection appears in various component packages of the SuiteSparse meta-package, available at http://www.cise.ufl.edu/research/sparse.

The primary MATLAB interface to the collection is the UFget toolbox, which has already been described above. The UFcollection package is a MATLAB toolbox used to create and update the MATLAB index for the collection, to create the web pages, and to export the collection to the Matrix Market and Rutherford-Boeing formats. To accomplish these tasks, it relies on the UFget, CHOLMOD [18, 25], CSparse [21], and RBio packages. CHOLMOD is a sparse Cholesky factorization and update/downdate package. It includes software, in C, for reading and writing matrices in the Matrix Market format. These functions also include a MATLAB interface. RBio is a MATLAB toolbox written in Fortran for reading a writing sparse matrices in the Rutherford-Boeing format. The toolbox can also read all matrices in the original Harwell-Boeing format. CSparse is a software package developed in conjunction with the book *Direct Methods for Sparse Linear Systems* [21]. It is used here for creating pictures of the matrices for posting on the collection's web site, and for computing various matrix statistics.

Additional software for reading and writing Rutherford-Boeing matrices can be obtained at http://www.cerfacs.fr/algor/Softs/RB. Software for reading and writing Matrix Market files can be obtained at http://www.nist.gov/MatrixMarket.

5. Example uses of the collection. A collection such as the one described here can answer many questions about the design, analysis, and performance of sparse matrix algorithms that cannot be answered by theoretical analyses or artificially-generated matrices. Matrices obtained from (or available in) this collection have been used in a wide range of published experimental results. For a small sample, see [3, 4, 5, 10, 15, 27, 34, 35, 46, 70]. Google Scholar, as of January 2007, lists 115 papers that cite the collection explicitly, as it appeared in an NA Digest note in 1997 [19].

Essentially all research articles that include a section on the performance analysis of a sparse matrix algorithm include results on matrices from real applications, many of them matrices in the UF Collection. Articles that do not use these standard benchmarks typically use matrices that could arise in practice, such as the 5-point discrete Laplacian on a regular mesh ([66], for example).

Two examples of the kinds of questions a set of real matrices can answer are given here: the average-case time complexity of the minimum degree ordering algorithm, and the tradeoff between supernodal and non-supernodal sparse Cholesky factorization methods [18].

[2]The matrix A or A^T is used, whichever is tall and thin. Note that this overestimates the size of a linear programming problem, since those problems do not require a QR factorization.

[3]The outlier matrix in 1971 is from the Edinburgh Associative Thesaurus, www.eat.rl.ac.uk, obtained from the Pajek data set [9]. It is a graph with 23,219 nodes and 325,592 edges that was first constructed in 1971 [53].

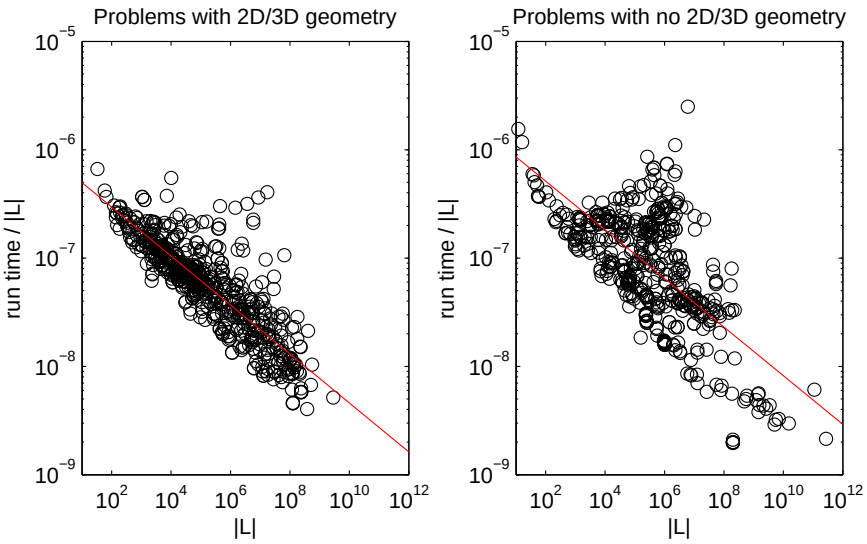

FIG. 5.1. *AMD run time over $|L|$, as a function of $|L|$*

5.1. First example: average-case run time of minimum degree. The running time of the minimum degree ordering algorithm is notoriously difficult to analyze. Under modest assumptions, a loose worst-case upper bound on the run time of the AMD variant of this method is given by

$$O\left(\sum_{k=1}^{n} |L_{k*}| \cdot |(PAP^T)_{k*}|\right), \tag{5.1}$$

where P is the permutation found by AMD, L is the Cholesky factor of PAP^T, and $|x|$ denotes the number of nonzeros in a vector or matrix [1, 2]. The bound does not consider the speedup obtained by exploiting supernodes, which has not been analyzed because the graph changes unpredictably during the elimination.

A single dense row or column of A leads to a run time of at least $\Omega(n^2)$, but AMD includes a preprocessing step that removes any row or column with degree $10\sqrt{n}$ or larger (the effect of this step is not accounted for in (5.1)). Still, a host of nearly-dense rows/columns could lead to unacceptable ordering time. Is the bound (5.1) reached in practice? What is the average-case time complexity of AMD?

Determining the theoretical average-case time complexity is beyond the scope of any analysis that has been done for this algorithm, so the best that can be done is to test the method on a set of "average" sparse matrices that arise in practice. The results of ordering $A + A^T$ (ignoring numerical cancellation in the matrix addition) on all structurally non-singular square matrices in the collection (1361 of them, excluding 3 matrices too large for this experiment) are shown in Figures 5.1 and 5.2.

Each matrix is a single circle in the plots. The results are split into two sets: matrices from problems with 2D/3D geometry, and problems without any underlying geometry. Each plot includes a best-fit line. Figure 5.1 shows that $O(|L|)$ is a very loose upper bound on the average case run time (excluding a few worst-case examples), even though $O(|L|)$ is already a much lower bound than (5.1). The best-fit line is the function $T(|L|) = O(|L|^{0.77})$ in both plots. Figure 5.2 shows that for most matrices,

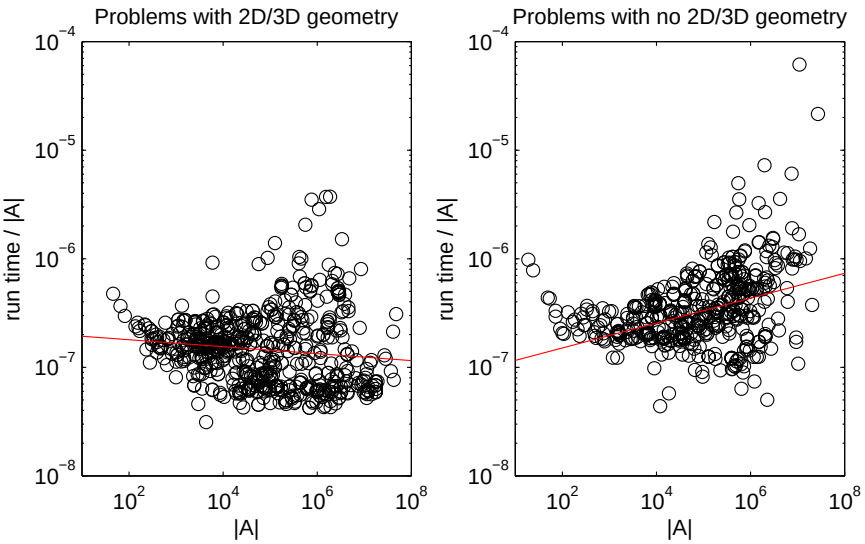

FIG. 5.2. *AMD run time over $|A|$, as a function of $|A|$*

$O(|A|)$ could be considered a reasonable estimate of the average-case time complexity of the AMD algorithm for problems with 2D/3D geometry (the slope is essentially flat; $T(|A|) = O(|A|^{0.97})$). The algorithm examines each entry in A, so it must be $\Omega(|A|)$ even in the best case. For problems with no 2D/3D geometry, the slope of the best-fit line is slightly positive, indicating a function $T(|A|) = O(|A|^{1.11})$. The outliers in this figure tend to have many nearly-dense rows and columns.

5.2. Second example: BLAS tradeoff in sparse Cholesky factorization. CHOLMOD is a sparse Cholesky factorization and update/downdate package that appears in `x=A\b` and `chol(A)` in MATLAB, when `A` is sparse and symmetric positive definite [18, 25]. It includes two sparse Cholesky factorization methods, a BLAS-based supernodal method [59] and an up-looking non-BLAS-based method [21]. The two methods are included because a BLAS-based method is slower for very sparse matrices (tridiagonal ones, for example). The dense matrix operations in the BLAS gain their performance advantage when the ratio of floating-point work to memory traffic is high. Thus, we predicted that the ratio of the number of floating-point operations over the number of nonzeros in L would be a good way to automatically select the appropriate method. Both of these terms are available from the symbolic analysis, prior to numerical factorization. A similar metric was used to compare the BLAS-based SuperLU method [26] with its non-BLAS based precursor, GPLU [43]. The primary difference is that for sparse LU factorization, the metric can only be estimated prior to numeric factorization, which limits its use as a simple method for selecting the appropriate method.

Two questions remain: how useful is this metric, and what should the cutoff value be? We tested both methods with 320 matrices from the collection: all symmetric positive definite matrices and all symmetric binary matrices with zero-free diagonals to which values were added to ensure positive-definiteness; random matrices were excluded (as of September 2006, when there were only 1377 matrices in the collection). The relative performance of the two methods is plotted versus the flops/$|L|$ ratio, as shown in Figure 5.3. These results show that the flops/$|L|$ ratio is a remarkably

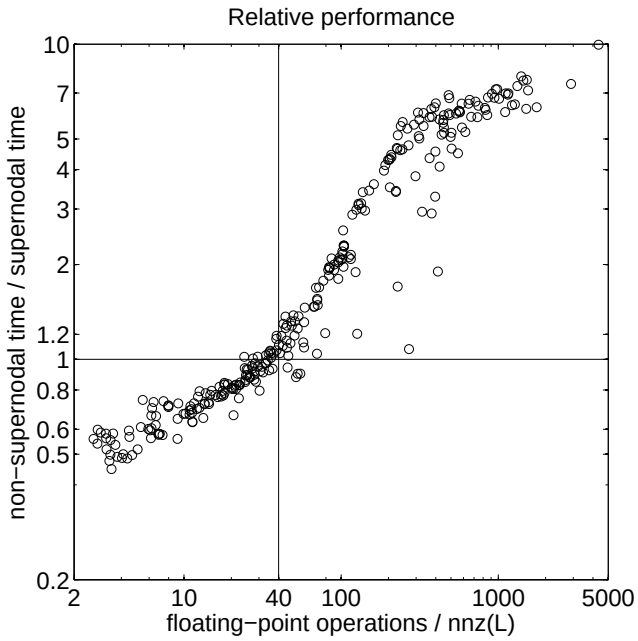

FIG. 5.3. *CHOLMOD relative supernodal and non-supernodal performance, from [18]*

accurate predictor of the relative performance of these two methods (much better than we expected). They also show that a value of 40 on a Pentium 4 is a good threshold. Even when the wrong method is selected using is approach, at most a 20% performance penalty occurs. The threshold of 40 is fairly insensitive to the architecture (it would be 30 on an AMD processor, and 35 on a Sun Sparc). It would be impossible to determine this cutoff using random matrices or a theoretical analysis.

6. Summary. A large, easily accessible, and actively growing collection of sparse matrices from real applications is crucial for the development and testing of sparse matrix algorithms. The University of Florida Sparse Matrix Collection meets this need, as the largest collection available and one of the most widely used.

Computational scientists are encouraged to submit their sparse matrices for inclusion in the collection. Matrices used in published performance evaluations of sparse matrix algorithms are of particular interest, to enable repeatable experiments by other researchers. Submit matrices to http://www.cise.ufl.edu/∼web-gfs, for username `davis`. Use a standard format for the matrix, such as a MATLAB MAT-file, a Rutherford-Boeing file, a Matrix Market file, or a list of triplets (where each line of the file contains the row index, column index, and numerical value of one entry in the matrix). Include a description of the matrix, the problem area it arises from, citations (if available), and source (in case the matrix author and submitter are different). Refer to Section 3.2 for more details about the meta-data to submit. Refer to Table 3.2 for a list of categories, and select one of them for your matrix or propose a new one.

Acknowledgments. I would like to thank John Gilbert for his comments on random sparse matrices and his feedback on a draft of this paper. I would also like to

thank the 239 matrix authors and 48 editors/collectors (excluding myself) who made this collection possible.

REFERENCES

[1] P. R. Amestoy, T. A. Davis, and I. S. Duff, *An approximate minimum degree ordering algorithm*, SIAM J. Matrix Anal. Appl., 17 (1996), pp. 886–905.

[2] ———, *Algorithm 837: AMD, an approximate minimum degree ordering algorithm*, ACM Trans. Math. Software, 30 (2004), pp. 381–388.

[3] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster, *A fully asynchronous multifrontal solver using distributed dynamic scheduling*, SIAM J. Matrix Anal. Appl., 23 (2001), pp. 15–41.

[4] P. R. Amestoy, X. Li, and E. G. Ng, *Diagonal Markowitz scheme with local symmetrization*, SIAM J. Matrix Anal. Appl., 29 (2007), pp. 228–244.

[5] P. R. Amestoy and C. Puglisi, *An unsymmetrized multifrontal LU factorization*, SIAM J. Matrix Anal. Appl., 24 (2002), pp. 553–569.

[6] E. Anderson, Z. Bai, C. H. Bischof, S. Blackford, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. C. Sorensen, *LAPACK Users' Guide*, SIAM, Philadelphia, 3rd ed., 1999.

[7] Z. Bai, D. Day, J. Demmel, and J. Dongarra, *Non-Hermitian eigenvalue problems*. http://www.cs.ucdavis.edu/∼bai/NEP.

[8] Z. Bai, D. Day, J. Demmel, and J. Dongarra, *Test matrix collection (non-Hermitian eigenvalue problems), Release 1*, tech. report, University of Kentucky, September 1996. Available at ftp://ftp.ms.uky.edu/pub/misc/bai/Collection.

[9] V. Batagelj and A. Mrvar, *Pajek networks*. http://vlado.fmf.uni-lj.si/pub/networks/data.

[10] M. Benzi and M. Tuma, *A sparse approximate inverse preconditioner for nonsymmetric linear systems*, SIAM J. Sci. Comput., 19 (1998), pp. 968–994.

[11] A. J. Berger, J. M. Mulvey, E. Rothberg, and R. J. Vanderbei, *Solving multistage stochastic programs using tree dissection*, Tech. Report SOR-95-07, Dept. of Civil Eng. and Operations Research, Princeton Univ., Princeton, NJ, June 1995.

[12] R. F. Boisvert, R. Pozo, K. Remington, R. Barrett, J. Dongarra, B. Miller, and B. Lipman, *The Matrix Market*. http://math.nist.gov/MatrixMarket.

[13] R. F. Boisvert, R. Pozo, K. Remington, R. Barrett, and J. J. Dongarra, *The Matrix Market: A web resource for test matrix collections*, in Quality of Numerical Software, Assessment and Enhancement, R. F. Boisvert, ed., Chapman & Hall, London, 1997, pp. 125–137. (`http://math.nist.gov/MatrixMarket`).

[14] C. W. Bomhof and H. A. van der Vorst, *A parallel linear system solver for circuit simulation problems*, Numer. Linear Algebra Appl., 7 (2000), pp. 649–665.

[15] I. Brainman and S. Toledo, *Nested-dissection orderings for sparse LU with partial pivoting*, SIAM J. Matrix Anal. Appl., 23 (2002), pp. 998–1012.

[16] R. K. Brayton, F. G. Gustavson, and R. A. Willoughby, *Some results on sparse matrices*, Math. Comp., 24 (1970), pp. 937–954.

[17] J. Bunch, *personal communication*, 2007.

[18] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam, *Algorithm 8xx: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate*, ACM Trans. Math. Software, (2006). (submitted).

[19] T. A. Davis, *University of Florida sparse matrix collection*. http://www.cise.ufl.edu/research/sparse, 1997. NA Digest, vol 97, no. 23.

[20] ———, *A column pre-ordering strategy for the unsymmetric-pattern multifrontal method*, ACM Trans. Math. Software, 30 (2004), pp. 165–195.

[21] ———, *Algorithm 849: A concise sparse Cholesky factorization package*, ACM Trans. Math. Software, 31 (2005), pp. 587–591.

[22] T. A. Davis and I. S. Duff, *An unsymmetric-pattern multifrontal method for sparse LU factorization*, SIAM J. Matrix Anal. Appl., 18 (1997), pp. 140–158.

[23] ———, *A combined unifrontal/multifrontal method for unsymmetric sparse matrices*, ACM Trans. Math. Software, 25 (1999), pp. 1–19.

[24] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng, *A column approximate minimum degree ordering algorithm*, ACM Trans. Math. Software, 30 (2004), pp. 353–376.

[25] T. A. Davis and W. W. Hager, *Dynamic supernodes in sparse Cholesky update/downdate and triangular solves*, ACM Trans. Math. Software, (2006). (submitted).

[26] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu, *A supernodal*

approach to sparse partial pivoting, SIAM J. Matrix Anal. Appl., 20 (1999), pp. 720–755.
[27] J. W. Demmel, J. R. Gilbert, and X. S. Li, *An asynchronous parallel supernodal algorithm for sparse Gaussian elimination*, SIAM J. Matrix Anal. Appl., 20 (1999), pp. 915–952.
[28] I. S. Duff, *Pivot selection and row ordering in Givens reductions on sparse matrices*, Computing, 13 (1974), pp. 239–248.
[29] ———, *A survey of sparse matrix research*, Proc. IEEE, 65 (1977), pp. 500–535.
[30] ———, *On algorithms for obtaining a maximum transversal*, ACM Trans. Math. Software, 7 (1981), pp. 315–330.
[31] I. S. Duff, R. G. Grimes, and J. G. Lewis, *Harwell/Boeing collection*. http://www.cse.clrc.ac.uk/nag/hb.
[32] ———, *Sparse matrix test problems*, ACM Trans. Math. Software, 15 (1989), pp. 1–14.
[33] ———, *The Rutherford-Boeing sparse matrix collection*, Tech. Report RAL-TR-97-031, Rutherford Appleton Laboratory, Oxon, UK, July 1997.
[34] I. S. Duff and J. Koster, *The design and use of algorithms for permuting large entries to the diagonal of sparse matrices*, SIAM J. Matrix Anal. Appl., 20 (1999), pp. 889–901.
[35] I. S. Duff and S. Pralet, *Strategies for scaling and pivoting for sparse symmetric indefinite problems*, SIAM J. Matrix Anal. Appl., 27 (2005), pp. 313–340.
[36] I. S. Duff and J. K. Reid, *Performance evaluation of codes for sparse matrix problems*, in Performance Evaluation of Numerical Software, L. D. Fosdick, ed., New York: North-Holland, New York, 1979, pp. 121–135.
[37] A. L. Dulmage and N. S. Mendelsohn, *Two algorithms for bipartite graphs*, J. SIAM, 11 (1963), pp. 183–194.
[38] A. Edelman, *Eigenvalues and condition numbers of random matrices*, SIAM J. Matrix Anal. Appl., 9 (1988), pp. 543–560.
[39] P. Feldmann, R. Melville, and D. Long, *Efficient frequency domain analysis of large nonlinear analog circuits*, in Proceedings of the IEEE Custom Integrated Circuits Conference, Santa Clara, CA, 1996.
[40] D. Gay, *NETLIB LP test problems*. http://www.netlib.org/lp.
[41] J. R. Gilbert, *personal communication*. See also `help sprank` in MATLAB.
[42] J. R. Gilbert, C. Moler, and R. Schreiber, *Sparse matrices in MATLAB: design and implementation*, SIAM J. Matrix Anal. Appl., 13 (1992), pp. 333–356.
[43] J. R. Gilbert and T. Peierls, *Sparse partial pivoting in time proportional to arithmetic operations*, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 862–874.
[44] N. Gould, Y. Hu, and J. Scott, *Gould/Hu/Scott collection*. ftp://ftp.numerical.rl.ac.uk/pub/matrices/symmetric.
[45] A. Gupta, *Fast and effective algorithms for graph partitioning and sparse matrix ordering*, Tech. Report RC 20496 (90799), IBM Research Division, Yorktown Heights, NY, July 1996.
[46] ———, *Improved symbolic and numerical factorization algorithms for unsymmetric sparse matrices*, SIAM J. Matrix Anal. Appl., 24 (2002), pp. 529–552.
[47] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, SIAM, Philadelphia, 2nd ed., 2002.
[48] Y. Hu. personal communication, Wolfram Research, Inc., 2007.
[49] S. Kamvar, *Stanford-Berkeley web matrices*. http://www.stanford.edu/∼sdkamvar.
[50] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., 20 (1998), pp. 359–392.
[51] E. Keiter, S. Hutchinson, R. Hoekstra, E. Rankin, T. Russo, and L. Waters, *Computational algorithms for device-circuit coupling*, Tech. Report SAND2003-0080, Sandia National Laboratories, Albequerque, NM, 2003.
[52] I. Kemelmacher, *Indexing with unkown illumination and pose*, in IEEE Conf. Computer Vision and Pattern Recognition, 2005.
[53] G. R. Kiss, C. Armstrong, R. Milroy, and J. Piper, *An associative thesaurus of English and its computer analysis*, in The Computer and Literary Studies, A.J. Aitken, R.W. Bailey, and N. Hamilton-Smith, eds., Edinburgh Univ. Press, 1973.
[54] J. Koster, *Parasol matrices*. http://www.parallab.uib.no/projects/parasol/data.
[55] C. Mészáros, *Linear programming test set*. http://www.sztaki.hu/∼meszaros/public_ftp/lptestset.
[56] J. J. H. Miller and S. Wang, *An exponentially fitted finite element method for a stationary convection-diffusion problem*, in Computatioal methods for boundary and interior layers in several dimensions, J. J. H. Miller, ed., Boole Press, Dublin, 1991, pp. 120–137.
[57] W. Miller, *The Engineering of Numerical Software*, Prentice-Hall, Englewood Cliffs, NJ, 1984.
[58] H. Mittelmann, *Linear programming test set*. http://plato.asu.edu/ftp/lptestset.

[59] E. G. Ng and B. W. Peyton, *Block sparse Cholesky algorithms on advanced uniprocessor computers*, SIAM J. Sci. Comput., 14 (1993), pp. 1034–1056.
[60] L. Page, S. Brin, R. Motwani, and T. Winograd, *The PageRank citation ranking: bringing order to the web*, tech. report, Stanford Digital Library Technologies Project, Stanford, CA, Jan. 1998.
[61] A. Pothen and C. Fan, *Computing the block triangular form of a sparse matrix*, ACM Trans. Math. Software, 16 (1990), pp. 303–324.
[62] J. K. Reid, *Sparse matrices*, in The State of the Art in Numerical Analysis, D. A. H. Jacobs, ed., New York: Academic Press, 1977, pp. 85–146.
[63] M. G. C. Resende, K. G. Ramakrishnan, and Z. Drezner, *Computing lower bounds for the quadratic assignment problem with an interior point algorithm for linear programming*, Operations Research, 43 (1995), pp. 781–791.
[64] E. B. Rudnyi, *Oberwolfach model reduction benchmarks*. http://www.imtek.uni-freiburg.de/simulation/benchmark.
[65] E. B. Rudnyi, B. van Rietbergen, and J. G. Korvink, *Model reduction for high dimensional micro-FE models*, in Proc. 3rd HPC-Europa Transnat. Access Meeting, Barcelona, 2006.
[66] A. Ruesken, *Approximation of the determinant of large sparse symmetric positive definite matrices*, SIAM J. Matrix Anal. Appl., 23 (2002), pp. 799–818.
[67] Y. Saad, *SPARSKIT collection*. http://math.nist.gov/MatrixMarket/data/SPARSKIT.
[68] ———, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Co., 1996.
[69] O. Schenk, *University of Basel collection*. http://www.computational.unibas.ch/computer_science/scicomp/matrices.
[70] J. Schulze, *Towards a tighter coupling of bottom-up and top-down sparse matrix ordering methods*, BIT, 41 (2001), pp. 800–841.
[71] H. Sedarat and D. G. Nishimura, *On the optimality of the Gridding Reconstruction Algorithm*, IEEE Trans. Medical Imaging, 19 (2000), pp. 306–317.
[72] A. van Heukelum, G. T. Barkema, and Bisseling R. H., *DNA electrophoresis studied with the cage model*, J. Comp. Phys., 180 (2002), pp. 313–326.
[73] S. E. Zitney, *Sparse matrix methods for chemical process separation calculations on supercomputers*, in Proc. Supercomputing '92, Minneapolis, MN, Nov. 1992, IEEE Computer Society Press, pp. 414–423.
[74] S. E. Zitney, J. Mallya, T. A. Davis, and M. A. Stadtherr, *Multifrontal vs. frontal techniques for chemical process simulation on supercomputers*, Comput. Chem. Eng., 20 (1996), pp. 641–646.