Electronic Transactions on Numerical Analysis.
Volume 22, pp. 122-145, 2006.

ISSN 1068-9613.

ETNA
Kent State University
etna@mcs.kent.edu

COMBINATORIAL ALGORITHMS FOR COMPUTING COLUMN SPACE BASES THAT HAVE SPARSE INVERSES*

ALI PINAR†, EDMOND CHOW‡, AND ALEX POTHEN§

Abstract. This paper presents a new combinatorial approach towards constructing a sparse, implicit basis for the null space of a sparse, under-determined matrix A. Our approach is to compute a column space basis of A that has a sparse inverse, which could be used to represent a null space basis in implicit form. We investigate three different algorithms for computing column space bases: two greedy algorithms implemented using graph matchings, and a third, which employs a divide and conquer strategy implemented with hypergraph partitioning followed by a matching. Our results show that for many matrices from linear programming, structural analysis, and circuit simulation, it is possible to compute column space bases having sparse inverses, contrary to conventional wisdom. The hypergraph partitioning method yields sparser basis inverses and has low computational time requirements, relative to the greedy approaches. We also discuss the complexity of selecting a column space basis when it is known that such a basis exists in block diagonal form with a given small block size.

Key words. sparse column space basis, sparse null space basis, block angular matrix, block diagonal matrix, matching, hypergraph partitioning, inverse of a basis

AMS subject classifications. 65F50, 68R10, 90C20

1. Introduction.

Many applications require a matrix Z that represents a basis for the null space of a large, sparse, under-determined matrix A. We describe new approaches for obtaining an implicit null space basis by computing a basis for the column space of A; the matrix representing the column space basis is required to have a sparse inverse. The algorithms for computing bases of the column space are based on the combinatorial concepts of matchings and hypergraph partitioning.

One context in which a null space basis is required is constrained optimization when the Karush-Kuhn-Tucker (KKT) system

$$\begin{bmatrix} G & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} x \\ \lambda \end{bmatrix} = \begin{bmatrix} -d \\ b \end{bmatrix}$$

is solved by the null space method, also called the *force* method in structural mechanics. Here G is $n \times n$, A is $m \times n$ with $m < n$, and the vectors are partitioned to conform with G and A. For discussion purposes, we assume that A has full row rank m. The null space method involves solving systems with the reduced-Hessian Z^TGZ, and thus the method is often advantageous when $n - m$ is small.

One approach for computing a null space basis Z is the following. Let P be a permutation

*Received March 18, 2005. Accepted for publication December 7, 2005. Recommended by R. Lehoucq.

†Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720 (apinar@lbl.gov). Supported by the Director, Office of Science, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract DE-AC03-76SF00098.

‡Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, L-560, Box 808, Livermore, CA 94568. The work of this author was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48. Current Address: D. E. Shaw Research and Development, 39th Floor, 120 West 45th St., New York NY 10036 (etchow@gmail.com).

§Computer Science Department and Center for Computational Science, Old Dominion University, Norfolk, VA 23529 (pothen@cs.odu.edu). The work of this author was supported by NSF grants ACI-023722 and CCF 0515218, by DOE grant DE-FC02-01ER25476, and by Lawrence Livermore National Laboratory under contract B542604.

ETNA
Kent State University
etna@mcs.kent.edu

matrix such that AP can be partitioned as $[B\ N]$ where B is nonsingular. Then, a basis Z is

$$Z = P \begin{bmatrix} -B^{-1}N \\ I_{n-m} \end{bmatrix},$$

which embeds within the basis an $n-m$-dimensional identity matrix. A basis Z of this form is called a *variable elimination basis* or a *fundamental null space basis* [21]. This null space basis can be computed explicitly, or can be represented in an implicit form in which the matrix B is factored as $B = LU$; then Z and Z^TGZ are treated implicitly as operators.

The motivation for our work is to obtain a different implicit representation of a null space basis by computing a column space basis B that has a sparse inverse. (Throughout this paper, we do not distinguish between a basis and a matrix representing that basis.) Thus to compute products involving Z, one can compute products involving B^{-1} instead of solving triangular systems with the triangular factors of B. When the matrix B is represented in factored form, computations with the reduced Hessian Z^TGZ involve multiple solutions of triangular systems using the factors of B. This can be inefficient on a parallel computer due to the high communication cost to computation cost ratio inherent in sparse triangular solution. On modern processor architectures with deep memory hierarchies, the low number of arithmetic operations relative to memory accesses in sparse triangular solution also causes it to have low performance due to high memory access costs.

Our approach in this paper is to select a basis B that has a block angular form for the column space of the matrix A. Such a basis has the advantage that the inverse matrix B^{-1} can be computed and stored explicitly, with nonzeros limited to the diagonal blocks and the coupling columns. We believe that the column space bases computed by our approach could be used to construct explicitly sparse null space bases. However, we leave computational verification of this claim for future work.

In some applications a sparse inverse for the column space basis, B^{-1}, is natural. For example, in structural analysis, multi-point constraints express a set of variables x_s called *slaves* in terms of an independent set of variables x_m called *masters*. Algebraically, this can be written as

$$x_s = B^{-1}(b - Nx_m)$$

where B is a diagonal matrix. When the constraints cannot be expressed this way, there are situations when the columns and rows of A can be permuted to obtain a block diagonal matrix B with small block sizes. This is the case when contact interfaces intersect, where each contact interface is described by a set of constraints [6]. When the slave and master variables have been identified, a Gaussian elimination procedure called the *transformation method* (equivalent to the null space method) is then used to eliminate the constraint equations. The review [3] discusses many aspects of solving saddle point problems, including the role of sparse null space basis computations in that context.

It is clear that in many other applications there is no choice of P that will give an inverse matrix B^{-1} that is sparse. In incompressible fluid dynamics, A is a discrete divergence operator, and generally, no set of m columns of A has any special properties. In PDE-constrained optimization the most natural choice of P gives B^{-1} corresponding to a PDE solve [5], generally a dense operator. Thus our work is targeted at problems whose constraints are sparse and not PDE-based.

We take two approaches to design algorithms for computing bases with sparse inverses for the column space. Both of our approaches rely on choosing the basis B in block angular form with small blocks. (The block angular form of a matrix is illustrated later in the paper in Figure 3.1; the column bordered form is the one appropriate here.) One approach

ETNA
Kent State University
etna@mcs.kent.edu

to computing a block angular basis is to use a matching in an associated bipartite graph to select columns of A to belong to it. Each column is assigned a weight, which is dynamically updated during the algorithm, that reflects how its inclusion in the current partial basis would perturb the block angular structure of the partial basis. A greedy strategy is used to choose the column to augment the partial basis at each step. Two different algorithms result from this approach, depending on whether we choose to match a particular row to some column, or choose to match a column of lowest current weight irrespective of the new row that becomes matched. The second approach is to model the matrix A as a hypergraph, and then to use hypergraph partitioning to obtain a block angular form of A from which a basis B could be obtained.

Throughout this paper, we assume that the numerical rank of a submatrix formed by a set of columns of the matrix A is equal to its structural rank, the cardinality of a maximum matching in the bipartite graph of the submatrix. These concepts are described in more detail in Section 2.3.

This paper is organized as follows. Section 2 briefly reviews some preliminary concepts necessary in this paper. In Section 3, we describe heuristic greedy algorithms for computing column space bases with sparse inverses by means of matchings in bipartite graphs. In Section 4, we discuss a top-down algorithm for computing these bases by means of hypergraph partitioning to put A into a block angular form as a first step, before applying a matching-based algorithm to the subproblems. Section 5 presents the results of numerical tests with our algorithms. We compute the number of nonzeros in the inverse of a basis for the column space B^{-1}; the structure of the inverse matrix is computed using the transitive closure of the basis matrix. We compare our results to a weighted matching algorithm that constructs a sparsest basis B for column space of A, without attempting to control the number of nonzeros in B^{-1}. In the Appendix, we provide a theoretical analysis of the combinatorial problems that arise in computing block diagonal column space bases. In some cases it is known that a block diagonal column space basis B exists with block size bounded by a given K, e.g., by using information from the physical problem [6]. In other cases, it would be useful to be able to check if a column space basis B with block size bounded by some small K exists. For $K \leq 2$, there are fast algorithms for checking the existence of such bases; for larger K, we show that the problem of finding the block diagonal column basis is NP-complete. We note that computing a sparsest null space basis is NP-complete, whether or not the basis is constrained to embed an identity matrix [7].

2. Background.

2.1. Previous Work on Null Space Bases.

We now briefly discuss earlier approaches for computing sparse null space bases, a problem that motivates our current work.

Recall that the choice of a column space basis B immediately determines a fundamental null space basis Z in terms of B and the submatrix N formed by columns of A outside the basis. A null space basis with a more general form has also been proposed in earlier work. A *triangular* null space basis has the form

$$Z = P \begin{bmatrix} -B^{-1}N \\ I \end{bmatrix} U_{n-m},$$

where U_{n-m} denotes an upper triangular matrix of order $n - m$. Since a triangular null space basis could be represented in terms of a fundamental basis by post-multiplication with the matrix U_{n-m}, at first sight it could be surprising that triangular bases could be sparser than fundamental null space bases. However, each null vector in a null space basis (i.e., a column in the basis) is obtained from a linearly dependent set of columns chosen from A.

ETNA
Kent State University
etna@mcs.kent.edu

In a triangular basis, the ith null vector is computed from a subset of columns in B and a subset of the first i columns in N; in a fundamental basis, the ith null vector is computed from a subset of columns in B and the ith column of N. Since the set of columns of A available to construct a null vector in a triangular basis is a superset of the those available in a fundamental basis, one should expect that a triangular basis could be potentially sparser than a fundamental null space basis.

An early approach for constructing sparse null space bases is called the Turnback method [4, 13, 14, 16, 23]. The Turnback method generally constructs triangular bases, and identifies linear dependences among columns of A numerically. An initial QR factorization is first used to identify $n-m$ columns of A, called *start columns*, each one of which is linearly dependent on previously factored columns in A. The start columns form the submatrix N, and the remaining columns form B. From each start column, another numerical factorization (usually LU) is then used to find a linearly dependent subset of columns of A, so that the ith dependent subset includes the ith column of N, some subset of columns in B, and some subset of the first $i-1$ columns in N. Hence the ith linearly dependent subset leads to the ith null vector in a triangular basis. This numerical approach is costly, both in storage and computational time, due to the initial QR factorization on the matrix A, and the $n-m$ LU factorizations on subsets of columns of B and N. However, for many problems in structural mechanics, the equilibrium matrix A can be permuted into a nearly banded nonzero structure, and then these costs are reasonable.

One of us has shown in his thesis [21] and subsequent publications [7, 8] that structural linear dependence among the columns of a sparse matrix A can be identified using matchings in bipartite graphs, and that matching based methods are faster than numerical methods based on matrix factorizations. Graph matching can be used in both phases of the algorithm to compute null vectors: first, to identify the linearly dependent subsets of columns of A from which null vectors could be computed; and second, to ensure linear independence of the computed null vectors. A numerical factorization (LU factorization suffices) of each linearly dependent set of columns of A is needed to compute the numerical values in the null vectors. Both triangular and fundamental null space bases have been computed this way. Gilbert and Heath [13] designed and implemented a null space basis algorithm in which the start columns are identified numerically using the Turnback method, but the null vectors are computed using a matching approach.

A different approach for constructing sparse null space bases utilizes graph partitioning to reorder A as

$$QAP = \begin{bmatrix} A_1 & & & & S_1 \\ & A_2 & & & S_2 \\ & & \ddots & & \vdots \\ & & & A_k & S_k \end{bmatrix} \tag{2.1}$$

where P and Q are permutation matrices, the A_i are rectangular submatrices with fewer rows than columns, and the block of columns with S_i as submatrices has as few columns as possible. The partitioning has the additional requirement that the rectangular submatrices A_i have full row rank. (This is a column-bordered block angular form of the matrix A; a more detailed discussion is included in Section 3.2.) A basis for the column space of each submatrix A_i can then be used to assemble a sparse fundamental null space basis for A. The requirement that the submatrices A_i have full row rank, however, is difficult to satisfy. Thus researchers have looked at the simpler case where A is an equilibrium matrix, i.e., a matrix having the structure of the node-edge adjacency matrix of an undirected graph. In this case, either physical data from the problem or purely algebraic schemes can be used to find a

ETNA
Kent State University
etna@mcs.kent.edu

partitioning that satisfies all the requirements [20, 22].

Our work differs from earlier work on computing null space bases in several respects. It differs from earlier matching based approaches for computing null bases since we use matching based methods to compute bases with sparse inverses for the column space. We also employ a divide and conquer approach based on hypergraph partitioning for this problem. Finally, the results reported in earlier work, from the mid 1980s, are on relatively small matrices with hundreds of columns; we report results on much larger matrices.

2.2. Graph-theoretic Concepts. In this subsection we provide some basic definitions of graph concepts used in the rest of the paper. A more detailed review of relevant graph theory can be found in [9].

A graph G is denoted by a pair of sets (V, E), where V is a finite set of vertices and E, the set of edges, consists of tuples (u, v) of distinct vertices, $u, v \in V$. If we assign weights to the edges, then $w(u, v)$ will denote the weight of the edge (u, v).

A *path* in a graph $G = (V, E)$ from a vertex u to a vertex u' is a sequence of vertices $p = \langle v_0, v_1, v_2, \ldots, v_k \rangle$, where $v_0 = u$, $v_k = u'$, and $(v_i, v_{i+1}) \in E$ for $0 \leq i < k$. We say a vertex u' is *reachable* from a vertex u, if there is a path from u to u' in the graph.

A *matching* M in an undirected graph $G = (V, E)$ is a subset of edges $M \subseteq E$ such that for all vertices $v \in V$ at most one edge in M is incident on v. A vertex v is *matched* in a matching M if some edge in M is incident on v and it is *unmatched* otherwise. A matching is *perfect* if all the vertices in the graph are matched. If the edge $(u, v) \in M$ is in the matching, then vertices u and v are *mates*. The *maximum (cardinality) matching problem* is the problem of finding a matching of maximum cardinality in a given graph. The *maximum weight matching problem* is the problem of finding a matching that maximizes the total weight of its edges; such a matching need not be a matching with maximum cardinality in the graph. A third variant problem is a *maximum cardinality matching with maximum weight*, which is the problem of finding among all maximum cardinality matchings the one of maximum weight. All of these matching problems are well-studied in the literature, and polynomial-time algorithms have been designed for all of them.

Matching problems and algorithms are easier on *bipartite* graphs. A graph $G = (V, E)$ is bipartite if the vertex set V can be partitioned into two sets V_1 and V_2 such that any edge (u, v) has one vertex in V_1 and one vertex in V_2. Hopcroft and Karp [15] designed an $O(|E|\sqrt{|V|})$ time algorithm for maximum cardinality bipartite matching, which is asymptotically the fastest known algorithm for this problem.

2.3. Structural Rank of a Matrix. The *structural rank* of a matrix is the maximum rank among all matrices with the same nonzero structure but different numerical values. Thus the structural rank of a matrix is an upper bound on its numerical rank, and if the numerical values are chosen to avoid numerical cancellation, the two ranks are equal.

The structural rank of a matrix is equal to the cardinality of a maximum matching in the bipartite graph associated with the matrix. The bipartite graph $G = (R, C, E)$ of a matrix $A = (a_{ij})$ has a vertex $r_i \in R$ for the ith row and a vertex $c_j \in C$ for the jth column. Each nonzero a_{ij} in the matrix corresponds to an edge (r_i, c_j) in the bipartite graph. Every edge in E has one column vertex and one row vertex for its endpoints, and thus the row vertices R and column vertices C form the bipartition of the vertex set of G. Since the structural rank of the matrix is equal to the cardinality of the maximum matching in the bipartite graph, and is an upper bound on the numerical rank, every nonsingular matrix has a perfect matching in its bipartite graph.

A row-perfect matching M in the bipartite graph G of a matrix A (i.e., a matching in which every row vertex is matched) can be used to permute columns (or rows) A to bring

ETNA
Kent State University
etna@mcs.kent.edu

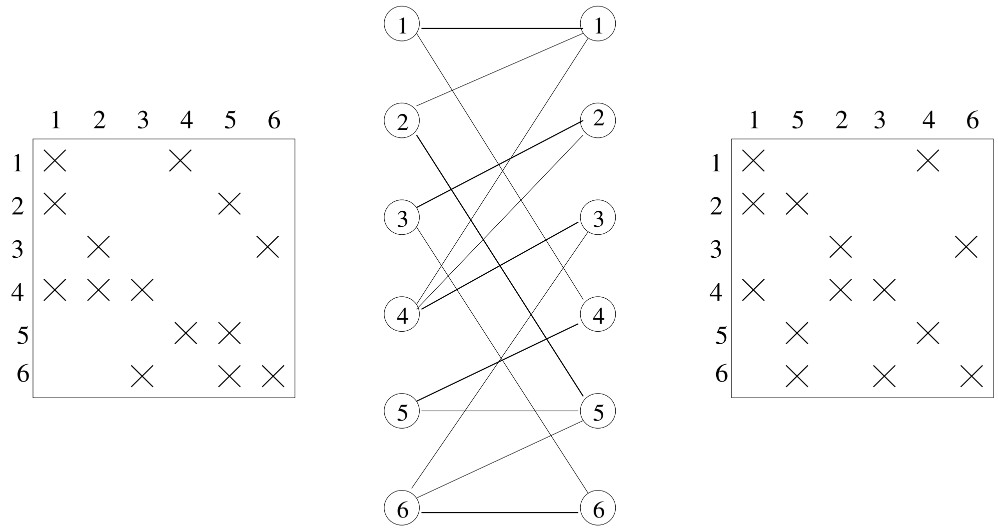

FIG. 2.1. *A square matrix, its bipartite graph, and the permuted matrix with a nonzero diagonal.*

nonzeros to the diagonal. Let A be a matrix with m rows and n columns, as in Section 1, and let the rows of A be ordered as $r_1, r_2, \ldots, r_m$. When we order the columns of the matrix as the $\langle$mate of row vertex r_1, mate of r_2, …, mate of $r_m\rangle$, with respect to some row-perfect matching, we place m nonzeros on a "diagonal" of A, corresponding to the m edges in the matching. For a square matrix, this is indeed the diagonal of the matrix, as illustrated in Figure 2.1.

Throughout this paper we assume that the structural rank of a matrix is equal to its numerical rank, an assumption that has been called the *weak Haar property* (wHP) [21]. If numerical values are assigned to the nonzeros of the matrix A randomly, then with high probability this assumption will be satisfied. Another approach is to assign algebraically independent values to the nonzeros (i.e., numbers that are not the roots of any multivariate polynomials with integer coefficients, since such roots form a set of measure zero); see Gilbert [12].

In practice, matrices from applications do not satisfy the wHP; one cause is duplicate columns in the matrix A. In null space basis computations, failure of the wHP leads to even more sparsity than predicted by a matching based method in the null space basis. In both null space basis and column space basis computations, the numerical rank of a basis with full structural rank needs to be computed using a numerical factorization.

2.4. Structure of the Inverse of a Matrix. The sparsity structure of the inverse of a square, non-singular matrix is determined by the path structure of the directed graph of the matrix. The directed graph $G = (V, E)$ of a square matrix $F = (f_{ij})$ of order m has the vertex set $\{v_1, v_2, \ldots v_m\}$, (where both the ith row and column are represented by the vertex v_i); and the edge set

$$E = \{(v_i, v_j) : i \neq j \text{ and } f_{ij} \neq 0\}.$$

We assume all diagonal entries of F are nonzero. Note that the existence of the inverse relies on the nonsingularity of F, which implies the existence of a perfect matching in the bipartite graph of F; thus the columns of F could be permuted to place m nonzeros on the diagonal.

The *transitive closure* of a directed graph $G = (V, E)$ is the directed graph $G^* =$

ETNA
Kent State University
etna@mcs.kent.edu

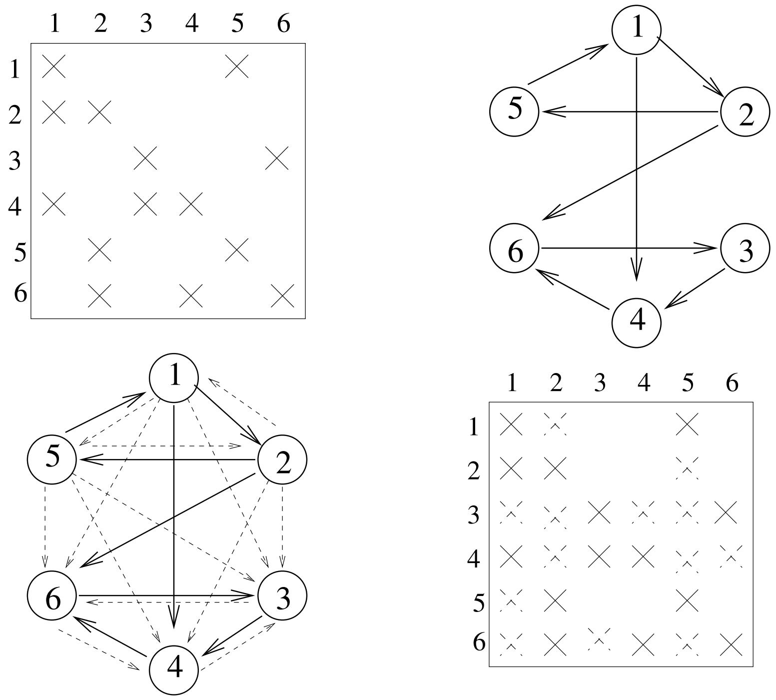

FIG. 2.2. *A matrix F (top left), the directed graph G of its transpose F^T (top right), the transitive closure of G (bottom left), and the structure of its inverse matrix F^{-1} (bottom right).*

(V, E^*), which has an edge corresponding to every directed path in G. That is,

$$E^* = \{(v_i, v_j) : \text{ if and only if } i \neq j \text{ and a directed path joins } v_i \text{ to } v_j \text{ in } G\}.$$

Gilbert [12] discusses the equivalence between the graph of F^{-1} and the transitive closure of the graph of F^T (the latter graph is equivalent to the directed graph of F with the edge directions reversed); this equivalence again assumes that the numerical values of the nonzeros in F are algebraically independent. An example is illustrated in Figure 2.2.

3. Greedy Approaches for Finding a Basis with a Sparse Inverse. The problem of selecting m columns from an $m \times n$ matrix A to form a structurally nonsingular matrix B is equivalent to finding a row-perfect matching in the bipartite graph of A. However, we need a basis that preserves its sparsity after inversion. We could try to obtain a sparse B^{-1} by making B as sparse as possible. This can be achieved by solving a weighted bipartite matching problem, with the weight of each edge equal to the degree of the column vertex it is incident on. A perfect matching in this bipartite graph finds a submatrix of m columns with the fewest nonzeros, which is structurally nonsingular. However, sparsity in B does not guarantee sparsity in its inverse. A well-known example of this is a tridiagonal matrix, which has a completely dense inverse. As discussed in Section 2.4, the structure of the inverse of a matrix F is given by the transitive closure of the directed graph of the transposed matrix F^T, and thus what is important is not the sparsity but the path structure in the directed graph of B^T. Each edge in the transitive closure of a graph corresponds to a directed path in the graph, and thus we need to choose B to minimize the number of vertices reachable by a directed path from a given vertex in the directed graph of B^T. Since a block diagonal matrix B is composed of decoupled blocks, the directed graph of B^T consists of several connected components, one

ETNA
Kent State University
etna@mcs.kent.edu

$$\begin{pmatrix} A_1 & & & \\ & A_2 & & \\ & & \ddots & \\ & & & A_4 \\ R_1 & R_2 & \dots & R_4 \end{pmatrix} \qquad \begin{pmatrix} A_1 & & & & C_1 \\ & A_2 & & & C_2 \\ & & \ddots & & \vdots \\ & & & A_4 & C_4 \end{pmatrix} \qquad \begin{pmatrix} A_1 & & & & C_1 \\ & A_2 & & & C_2 \\ & & \ddots & & \vdots \\ & & & A_4 & C_4 \\ R_1 & R_2 & \dots & R_4 & D \end{pmatrix}$$

Row bordered | Column bordered | Doubly bordered

FIG. 3.1. *Three block angular forms of matrices.*

for each block; this structure limits the number and lengths of directed paths in the directed graph of B^T. Thus a block diagonal matrix is one effective method to limit reachabilities and the number of nonzeros in B^{-1}. However, many matrices might not possess a block diagonal basis with small block sizes; such matrices might nonetheless have a block angular basis, in which a group of coupling columns is present in addition to the diagonal blocks.

Three block angular forms of a sparse matrix are shown in Fig. 3.1. The form that applies to our situation is the column bordered block angular form. In this form, the diagonal blocks are coupled by the submatrices in the last block of columns.

In this section, we present greedy techniques that find block angular bases for the column space. Our techniques rely on adding columns to a partial basis one by one. When choosing a column to add, we require that it should increase the structural rank of the partial basis by one; the column should also minimize a cost function that attempts to keep the block sizes small and the number of blocks in the partial basis large. The proposed methods call for efficient techniques for detecting whether a column increases the structural rank, for updating the block structure of the matrix as new columns are added, and for searching the space of candidate columns to add to the basis.

3.1. Feasibility. For structural nonsingularity of a matrix B, it is necessary and sufficient that B should have a perfect matching in its bipartite graph. While constructing the basis, we choose columns so that each column increases the size of the matching in the bipartite graph, and hence the structural rank of the matrix by one. We use an *augmenting path* to determine if a column increases the structural rank, a technique that has been used earlier in null-space basis computations [8].

Consider a bipartite graph $G = (R, C, E)$ and a matching M in this graph. We will use c_i to denote a column vertex belonging to the set C, and r_i to denote a row vertex belonging to the set R. An augmenting path $\langle c_0, r_0, c_1, r_1, \dots, c_k, r_k\rangle$ is a path between two unmatched vertices c_0 and r_k, whose edges alternate between matched and unmatched edges. The set of matched edges in the augmenting path, M_0, is $M_0 = \{(r_i, c_{i+1}) : 0 \le i < k\}$, with $M_0 \subseteq M$. The set of unmatched edges in the augmenting path, M_1, is $M_1 = \{(c_i, r_i) : 0 \le i \le k\}$. Note that by the definition of an augmenting path, the cardinality of M_1 is one more than that of M_0. If an augmenting path exists, then the size of the matching M can be increased by interchanging the matched and unmatched edges along the path. Hence $M' = (M \setminus M_0) \cup M_1$ is a matching whose cardinality is one greater than that of M. To determine if a column increases the size of the matching, we search for an augmenting path starting with this column. If one exists, then this column increases the size of the matching and thus is feasible. A more detailed discussion on augmenting paths can be found in [9]. Figure 3.1 illustrates an example.

3.2. Cost Function. Among the feasible columns, we want to choose one that perturbs minimally the block diagonal structure of the partial basis. Initially each row is a block by itself. When the first column is included in the basis, all of the rows in which it has nonzeros

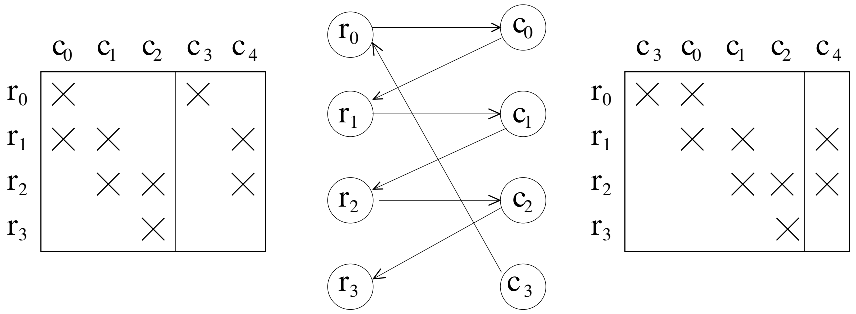

FIG. 3.2. *Checking the feasibility of a column. We check if column c_3 can be added to the current basis consisting of columns c_0, c_1, and c_2. An augmenting path that starts with c_3 and ends at r_3 is shown in the graph in the middle of the Figure. The matrix with column c_3 added to the basis is shown on the right.*

are merged into a single block.

Given a partial block diagonal basis, consider the addition of a new column to the basis. The set of rows in which the new column has nonzeros can be organized into blocks induced by the block structure of the current basis. The addition of the new column to the basis will cause these blocks to be merged into a single new block. We define the current cost of a column as the difference between the square of the new block size and the sum of squares of the sizes of the blocks it merges. More precisely, let a candidate column c have nonzeros in the current blocks 1, 2, . . ., k, and let m_i denote the current size of the ith block. If column c were to be added to the current basis, it would cause these k blocks to be merged into a single block, and the current cost of the column c would become

$$\left(\sum_{i=1}^{k} m_i\right)^2 - \sum_{i=1}^{k} m_i^2. \tag{3.1}$$

At each step, we choose a column that has the least cost among all columns not yet in the basis. The first term of the cost function in (3.1) is an upper bound on the number of nonzeros in the inverse of the merged block, and thus the cost of a column corresponds to the increase in this upper bound.

However, a basis B computed by using this cost function will not in general be block diagonal (with small block sizes), since the cost function does not look ahead of the current step in order to ensure small diagonal blocks. A more global view of the nonzero structure of the matrix A is required to identify diagonal blocks of small size in a basis B. However, the cost function creates disjoint blocks of small sizes in the partial basis and delays the merger of such blocks as long as possible until such merger becomes necessary to obtain a basis. Hence we expect to obtain a block angular structure for the basis B rather than a block diagonal structure.

The block angular structure of the basis leads to short directed path lengths for most vertices, while a few vertices corresponding to the coupling columns are involved in longer directed paths. The transitive closure of the directed graph of such a basis leads to few 'fill edges' for the many vertices in the first set, and to greater 'fill' for the few vertices in the second set. Thus this approach leads rather to sparse inverses in the column basis B.

Computing the cost of a column only requires identifying the blocks its rows belong to, and this can be implemented efficiently. However, adding a column to the basis could change its block structure, and consequently the costs of many other columns. Specifically, adding a column to the basis changes the costs of all columns that have a nonzero in a row of a block that the column is incident on. Recall that we work on matrices in which the number of columns is much larger than the number of rows, and thus updating costs of columns after each time we add a column could be time consuming.

The block structure of the basis changes dynamically as new columns are added. The efficiency of our heuristics in this section rely on effective data structures to maintain the block structure of the basis. Combinatorially this problem is equivalent to implementing disjoint set operations. Each block can be considered as a set of rows, and each column added to the basis is equivalent to a sequence of union operations on the sets of its rows. A detailed discussion on data structures for disjoint sets can be found in [9].

3.3. Searching the Space of Candidate Columns. In the previous two subsections we discussed first how to find a feasible column, and next our greedy strategy to choose a column. Now we will discuss our techniques to efficiently search for a column to add to the basis. A brute-force approach will search all columns to find a feasible column with minimum cost, and will not be efficient. Here we propose two algorithms to search for candidate columns. The first method is column based and maintains a priority queue of columns with respect to their costs, whereas the second method is row based and chooses the next column among those that can be matched to a given row. In the next two subsections we discuss these two heuristics.

3.3.1. Column-based Search. Our column-based search algorithm, as illustrated in Algorithm 1, maintains a priority queue of columns with respect to their costs. The feasibility of a column with the minimum cost is tested, and if feasible, it is added to the basis. To determine the feasibility of a column we look for an augmenting path that starts with it. If we reach an unmatched row, then the column is feasible, since it increases the structural rank by one. If there is no augmenting path that starts with this column, then it is infeasible and is discarded for the rest of the algorithm.

The critical part of this heuristic is updating the costs of columns after adding a column to the basis, and the consequent update to the block structure. Initially, we consider each row as a separate block; thus the cost of column c_i, from (3.1), is $deg(c_i)^2 - deg(c_i)$, where $deg(c_i)$ is the number of nonzeros in column c_i. After a column is added, the new block structure changes the costs of some columns. Recomputing the costs of all columns will not be efficient, and we have to restrict the updates to only those columns whose costs are changed.

The cost of a column changes only if one of the blocks it is adjacent to is merged into a bigger block. These are exactly the columns at a distance two (edges) from the column c_i in a bipartite graph representation in which each diagonal block in the current basis is represented by a block row vertex and a block column vertex. Equivalently, these columns are at a distance two from the columns in the current basis in the bipartite graph that represents the original matrix A. Based on this observation, after adding a column c_i to the basis, we generate a list of columns adjacent to rows in the new block generated by the column c_i and update their costs. Our algorithm is presented as Algorithm 1.

The most time-consuming part of the Column algorithm is updating the costs of columns after adding a column to the basis. Moreover, since we are working on matrices that have many more columns than rows, the column based heuristic has a a huge search space, and this leads to slow runtimes.

The time complexity of this algorithm is not easy to compute since the block sizes grow through the union of smaller blocks as columns are added to the basis during the algorithm. However, a worst-case bound (that is not realistic for sparse problems) is as follows. The dominant computation is updating the costs of the columns at a distance two from the columns in the current basis. The costs of all these columns can be updated in time proportional to

ETNA
Kent State University
etna@mcs.kent.edu


```
Set B = I and assign a cost to each column of A
for i = 1 to m do
    repeat
        Choose an unmatched column vertex c with the minimum cost
        Search for an augmenting path beginning at vertex c
        If no augmenting path can be found, then remove c from further consideration
    until an augmenting path is found
    Denote the final vertex on this path by r
    Replace the r-th column of B by the c-th column of A, and add r and c to the set of
    matched vertices
    Update the cost of each unmatched column of A at a distance of two from columns in
    the same block as c
end for
```

Algorithm 1: Column algorithm for computing a basis that has a sparse inverse.

$\mathrm{nnz}(A)$, the number of nonzeros in A, after a new column is added to the basis. Since there are m such column additions, the total cost of the column updates is $O(m\,\mathrm{nnz}(A))$ time.

3.3.2. Row-based Search. The row-based algorithm restricts the search space of columns whose costs need to be updated at each step to only those columns that can be reached by an augmenting path from a given row. We compute the cost of each unmatched column we reach, and choose one with minimum cost. This avoids the burden of updating column costs after each step. The row-based algorithm is presented as Algorithm 2.

```
1: Set B = I and assign a cost to each column of A
2: for i = 1 to m do
3:     Select an unmatched row vertex r
4:     Search for all augmenting paths beginning at vertex r, and denote the set of final ver-
       tices on these paths by C_r
5:     Compute the cost of every column in C_r, and select a column c of minimum cost
6:     Replace column r of B by the cth column of A, and add r and c to the set of matched
       vertices
7: end for
```

Algorithm 2: Row algorithm for computing a basis that has a sparse inverse

In this algorithm, the order in which the rows are considered is not specified (line 3 of Algorithm 2). However, the quality of the solutions depends on the order in which the rows are processed. In our experiments we used ascending and descending number of nonzeros per row, as well as random orderings.

As in the column algorithm, the time complexity of the row algorithm can be bounded by $O(m\,\mathrm{nnz}(A))$. However, we expect the row algorithm to be faster than the column algorithm, since the costs of only the columns in the set C_r needs to be computed when the row r is matched to some column.

4. A Top-down Approach. The greedy heuristics described in Section 3 work in a bottom up fashion, in which columns are added to a partial basis one by one, while trying to preserve a block angular form in the basis (and thereby sparsity in the inverse of the basis) as far as possible. In this section, we describe a top-down approach, where we remove columns from A in order to decompose the resulting matrix into multiple diagonal blocks. The idea is

ETNA
Kent State University
etna@mcs.kent.edu

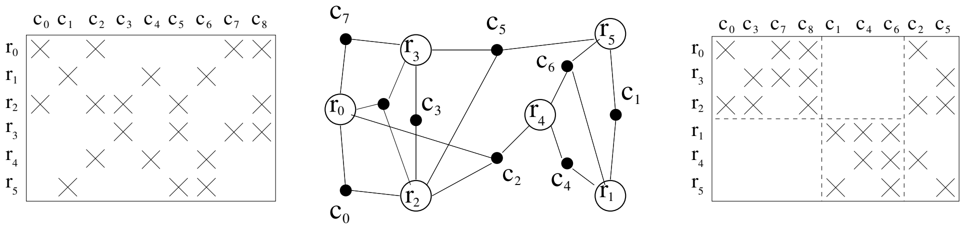

FIG. 4.1. *Permuting a matrix to block angular form with hypergraph partitioning.*

analogous to the nested dissection algorithm used to order sparse matrices to preserve sparsity during factorization. In this section, we discuss how to remove a small set of columns to decompose the residual matrix into two block diagonal submatrices. Then we propose a divide-and-conquer method that recursively applies this idea to choose a basis with small diagonal blocks.

4.1. Permuting Matrices to Block Angular Form. A block angular matrix is composed of independent blocks on the diagonal along with coupling rows and columns, as was illustrated in Figure 3.1. The block angular forms of a matrix can be exploited for parallel computation of LU and QR factorizations, and in decomposition algorithms to solve linear programming problems. Pinar et al. studied the problem of permuting a matrix to block angular form [2, 18, 19]. A thorough discussion on methods to permute matrices to block angular form can be found in [2]. Here we consider hypergraph models for the problem of computing a column bordered block angular form.

A hypergraph HG is defined by a set of vertices V, and a set of hyperedges HE, where each hyperedge is a subset of the vertices V. The nonzero structure of a matrix can be represented by a hypergraph, where each row is represented by a vertex, and each column is represented by a hyperedge. Each hyperedge representing a column contains those row vertices in which the column has nonzero elements. An example is illustrated in Figure 4.1. We call this a column hypergraph representation, since the hyperedges correspond to columns. An alternative row hypergraph representation for a matrix would represent columns by vertices and rows by hyperedges. However, in this context, a column hypergraph representation is the appropriate one.

The hypergraph partitioning problem is the problem of decomposing the vertices of the hypergraph into two or more parts, so as to minimize the number of hyperedges with vertices in different parts, while keeping the numbers of vertices in the parts roughly equal. A hyperedge whose vertices belong to more than one part is a *cut* hyperedge; a hyperedge whose vertices belong to only one part is an *internal* hyperedge. Hypergraph partitioning can be used to identify a permutation of the matrix to a block angular form. Given a partitioning of the hypergraph, we can permute the matrix so that vertices in the first (second) part define the rows in the first (second) block, and columns corresponding to internal hyperedges define the columns of the two blocks. Cut hyperedges constitute the coupling columns. By definition of hypergraph partitioning the blocks of the matrix will be block diagonal. Minimizing the cut size while partitioning the hypergraph minimizes the number of coupling columns in the block angular form of the matrix, and the size balance among parts of the hypergraph translates to balance among blocks of the matrix. Fig. 4.1 illustrates these concepts.

4.2. A Divide-and-Conquer Algorithm. The technique presented in the previous section can be used to find a block angular basis. We can partition the hypergraph so that the number of rows in each block is below a prescribed threshold. This can be done either by

ETNA
Kent State University
etna@mcs.kent.edu

determining the minimum number of parts that guarantees that the number of rows in each block is below the threshold, or by partitioning the hypergraph recursively until each block is smaller than the threshold.

Merely finding a block angular submatrix is not sufficient for our purposes. We need to extract a subset of a basis from each diagonal block, so each such submatrix should have at least as many columns as rows, and further it should be structurally non-singular. There is no guarantee that hypergraph partitioning will preserve full row rank (either structural or numerical) in the decomposed submatrices. It is impractical to enhance the hypergraph partitioner to enforce the full row rank in the blocks. We postpone handling the structural nonsingularity constraint to a post-processing phase.

We first compute the block angular form of the matrix, and then run the column algorithm from Section 3.3.1 to find a block diagonal sub-basis in each diagonal block. At the end of this step, some rows in each submatrix might remain unmatched, which means the number of columns chosen for the basis is smaller than the number of rows. In a post-processing phase, we run the row-based greedy algorithm to add columns to the basis from the set of coupling columns for structural nonsingularity. In our experiments, we have observed that only a few rows remain to be processed in the post-processing phase.

The time complexity of computing the block angular form is dominated by the complexity of the hypergraph partitioning algorithm, which needs to be recursively applied until the block sizes are small enough. Each partitioning step in the recursion can be implemented using the multilevel partitioning algorithm in time linear in the size of the hypergraph, which is $\mathrm{nnz}(A)$ (the number of nonzeros in the matrix A). Thus the complexity of the hypergraph partitioning needed to compute the block angular form is bounded by $\log m \, \mathrm{nnz}(A)$. (Currently available multilevel hypergraph partitioners, such as PaToH [24], Mondriaan [25], and hMETIS [17], use a few heuristics in the coarsening and refinement steps, which require more than linear time, in order to improve the quality of the partitions.)

5. Experimental Results. The Row, Column, and Top-down algorithms discussed in Sections 3.3.2, 3.3.1, and 4, respectively, were implemented in C and experiments were performed on a Sun Blade 100. The processor was a SPARC V9 operating at 502 MHz, with 128 MB of memory, running Solaris; the Gnu C compiler was used.

Initially we experimented with more than 105 problems from the Netlib LP test set [11]. We report results for every problem that has at least $1,000$ rows, 38 problems in all. Of these, Table 5.1 includes results for 26 problems. We report more detailed results on twelve other linear programming matrices later. Eight of these are multi-commodity flow problems from the 'ken' and 'pds' families; the other four problems are 'truss', 'cre-b', 'gosh' and 'd2q06c'. We also report results for a structural analysis problem, 'X135', and the 'pigs' matrices from animal breeding.

We experimented with additional problems from structural analysis [6], and circuit simulations (obtained from Sandia National Labs), but these had column space bases that were block diagonal matrices, with small blocks; hence these problems were not interesting for our purposes.

We report the number of rows, columns, and average number of nonzeros in a column for the matrices in Table 5.1, and also the average number of nonzeros in a column of the column basis B and its inverse B^{-1}. The problems are ordered by the average number of nonzeros in a column of the basis inverse. For the first 20 problems, the bases were computed by the Top-down algorithm, by using PaToH [24] as the hypergraph partitioner, where the the number of parts is chosen as $\lceil m/200 \rceil$, and 20% imbalance is allowed. For the last 6 problems, the results reported were computed by the Column algorithm, since it performed better than the Top-down algorithm for these problems. We see that 17 of the 26 matrices

TABLE 5.1

Sparsity in the bases and their inverses for LP matrices with at least 1000 *rows from the Netlib LP collection. Here* m *is the number of rows,* n *is the number of columns, the number of nonzeros is* $|A|$*, and the average number of nonzeros in a column is* $|A|/n$*. The results show the average number of nonzeros in a column of the basis* B *and in its inverse* B^{-1}*. The results are reported for bases computed by the Top-down algorithm for most of the problems, except for the last six. For the latter, the Column algorithm was employed.*

	m	n	$\|A\|/n$	$\|B\|/m$	$\|B^{-1}\|/m$
80bau3b	2,262	11,934	1.95	1.0	1.0
fit2p	3,000	13,525	3.71	1.0	1.0
maros-r7	3,136	9,408	15.4	1.0	1.0
osa-07	1,118	25,067	5.78	1.0	1.0
osa-14	2,337	54,797	5.79	1.0	1.0
osa-30	4,350	104,374	5.79	1.0	1.0
osa-60	10,280	243,246	5.79	1.0	1.0
sctap2	1,090	2,500	2.92	1.0	1.0
sctap3	1,480	3,340	2.92	1.0	1.0
pilot87	2,030	6,680	11.2	1.2	1.5
cre-a	3,428	7,248	2.51	1.3	1.7
pilot	1,441	4,860	9.13	1.3	1.7
cre-c	2,986	6,411	3.16	1.3	1.8
cre-d	6,476	73,948	3.33	2.0	2.5
sierra	1,227	2,735	2.93	1.9	3.6
bnl2	2,324	4,486	3.34	2.2	3.8
ship12s	1,042	2,869	2.89	2.8	4.8
ship12l	1,042	5,533	2.94	2.8	6.1
greenbea	2,389	5,598	5.55	4.3	8.0
greenbeb	2,389	5,598	5.55	4.3	8.0
ganges	1,309	1,706	4.07	2.7	9.3
woodw	1,098	8,418	4.45	3.5	11
stocfor2	2,157	3,045	3.07	2.3	21
degen3	1,503	2,604	9.77	6.9	28
stocfor3	16,675	23,541	3.09	2.3	35
dfl001	6,071	12,230	2.91	2.1	48

have fewer than 5 nonzeros in an average column of the inverse basis of the column space. For 9 of these problems, a diagonal basis B was found.

Our aim in presenting the results in Table 5.1 is to demonstrate that many linear programming constraint matrices have column space bases with sparse inverses. The 'baseline' for these results is the average number of nonzeros in a column of the original matrix A, which is what we would expect for a column space basis chosen at random. Notice that the average number of nonzeros in a column of the column space basis is often significantly lower than this baseline, and for many problems, the inverse of the column space basis does not incur much fill. We emphasize that we did not try to optimize the choice of parameters or options for the results reported above. It is possible that better choices of these values and better algorithms could lead to sparser bases for the five problems where the average number of nonzeros in a column is greater than ten. But these results should help dispel the folk wisdom that most optimization problems do not have sparse inverses for their column space bases.

We choose to report more detailed results on matrices from a subset of Netlib linear

ETNA
Kent State University
etna@mcs.kent.edu

TABLE 5.2
Additional matrices used in the experiments, showing the number of rows, m; the number of columns, n; the number of nonzeros, $|A|$; and the average number of nonzeros in a column, $|A|/n$.

	m	n	$\|A\|$	$\|A\|/n$
truss	1,000	8,806	27,836	3.2
d2q06c	2,171	5,831	33,081	5.7
X135	4,182	26,346	61,064	2.3
gosh	3,790	13,455	99,953	7.4
cre-b	7,240	77,137	260,785	3.4
ken07	2,426	3,602	8,404	2.3
ken11	14,694	21,349	49,058	2.3
ken13	28,632	42,659	97,246	2.3
ken18	105,127	154,699	358,171	2.3
pds02	2,953	7,716	16,571	2.1
pds06	9,881	29,351	63,220	2.2
pds10	16,558	49,932	107,605	2.2
pds20	33,798	108,175	232,647	2.2
pigs-m	6,119	9,397	25,013	2.7
pigs-l	17,264	28,254	75,018	2.7
pigs-v	105,882	174,193	463,303	2.7

programs (LPs) shown in Table 5.2. Problems from the 'ken' and 'pds' families, which are multi-commodity flow problems, permit us to study how the results scale as the problem sizes increase. Two problems in Table 5.2 were selected from outside the Netlib set of LPs. X135 is a matrix from structural analysis that has a block diagonal basis B with block size bounded by four [6]. The 'pigs' matrices are least squares problems (transposed) from problems in animal breeding.

We examined the influence of row orderings on the the Row algorithm in determining the sparsity of the basis inverses and the computation time. We compared the original ordering, ascending order of the number of nonzeros in a row, descending order of the same, and random orderings, but did not see consistent improvement in either quantity over the original ordering.

In the following tables, the results are reported as the average number of nonzeros per column of B, denoted by $|B|/m$; and the average number of nonzeros per column of its structural inverse, denoted by $|B^{-1}|/m$.

Table 5.3 shows the influence of block size on the sparsity of the basis inverse for the Top-down method for three problems. For our hypergraph partitioner, we chose the number of parts as $\lceil m/b \rceil$, where b is the block size. Five different maximum block sizes were used: $b =$ 50, 100, 200, 400, and 800. Ten trials were used for each block size, with each trial generating a different matrix decomposition. Note that the sparsities in the inverses are comparable to those obtained by the row method for 'X135'; better for 'ken18'; and better by an order of magnitude for the 'pigs-v' problem. When larger block sizes are allowed, the computation is faster and generally the sparsity varies less, as expected. Except for the 'pigs' matrices, the results are generally not too sensitive to the maximum block size. For the 'pigs' matrices, block sizes 50 and 100 gave poor results; a block size of 200 generally gave the best results.

Table 5.4 compares the sparsity of the bases and their inverses for the three algorithms

TABLE 5.3
The influence of block size on the Top-down method for three problems.

	block size	$\lvert B^{-1}\rvert/m$ mean	min.	std.	time (s)
X135	50	1.04	1.04	0.00	1.07
	100	1.04	1.04	0.00	0.91
	200	1.04	1.04	0.00	0.72
	400	1.04	1.04	0.00	0.56
	800	1.04	1.04	0.00	0.42
ken18	50	12.8	12.6	0.15	32.6
	100	13.3	13.1	0.13	30.4
	200	13.8	13.6	0.17	26.4
	400	13.9	13.8	0.09	22.8
	800	14.1	13.8	0.16	19.3
pigs-v	50	55.6	35.9	11.0	41.4
	100	20.6	18.1	1.64	38.0
	200	14.2	13.8	0.33	33.6
	400	16.0	15.6	0.18	28.7
	800	20.6	19.7	0.57	25.3

we have discussed thus far, the Row and Column algorithms and the Top-down algorithm. We include a fourth algorithm in the comparison, the Weighted matching algorithm (WM), which weights each column with the number of nonzeros in the column, and finds a row-perfect matching of minimum weight. This algorithm finds the sparsest column space basis B, although it does not directly control the number of nonzeros in the inverse of the basis.

The results in Table 5.4 indeed show that the Weighted matching algorithm produces the sparsest bases B, but with inverses that could be much denser than those of other methods. Note that for all methods, the average number of nonzeros in a column of the column space basis B compares favorably with the average number of nonzeros in a column of the matrix A, shown in Table 5.2.

Ranking the methods from worst to best in terms of the size of $|B^{-1}|$, we have the Weighted matching algorithm, the Row algorithm, the Top-down algorithm, and the Column algorithm. For the 'ken' and 'pds' families of problems, we note that there is a very slow degradation in the results as problem sizes increase. For the 'pigs' family of problems, the average number of nonzeros per column of the basis inverse decreases with problem size for the Top-down method.

The nonzero structures of the column space bases and their inverses, computed with the Weighted matching and the Column algorithms, are shown in Figure 5. Note that while both algorithms compute column space bases with equal numbers of nonzeros, the inverse of the basis computed by the Weighted matching algorithm has four times as many nonzeros as the one computed by the Column algorithm. Recall from Section 2.4 that the directed graph of the inverse of a matrix is the transitive closure of the directed graph corresponding to the matrix. This example demonstrates that it is not the number of nonzeros in a basis that determines the sparsity of the inverse of the basis, but rather the path structure in the directed graph representation of the basis. The Column algorithm is able to maintain short directed paths in its graph due to the objective function we employ for choosing a column to add to the basis, whereas the Weighted matching algorithm is oblivious to the path structure.

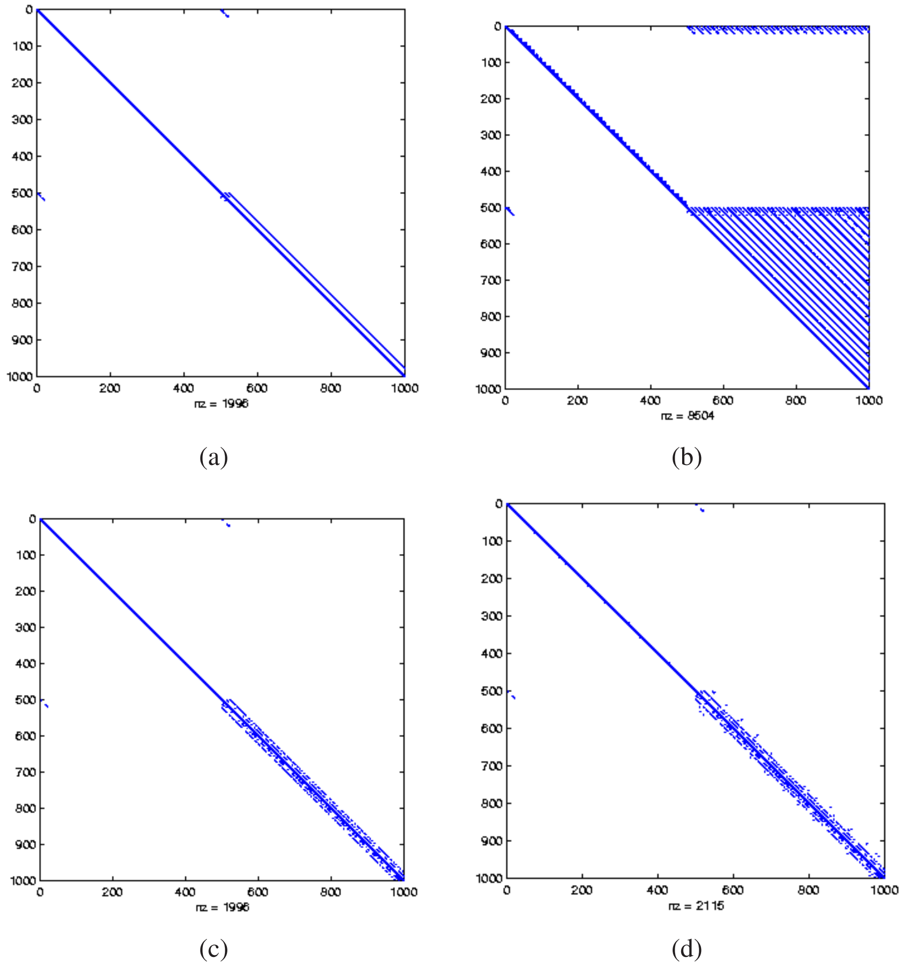

FIG. 5.1. *Comparing the nonzero structures of a column space basis and its inverse for the 'truss' matrix. Subfigure (a) corresponds to the basis matrix and (b) to the inverse of that basis matrix computed using the Weighted matching algorithm; corresponding matrices from the Column algorithm are shown in Subfigures (c) and (d). The number of nonzeros in both basis matrices is* $1,996$*; the inverse of the basis obtained by the Weighted matching algorithm has* $8,504$ *nonzeros, while the inverse of the basis computed by the Column algorithm has only* $2,115$ *nonzeros.*

Note also that if viewed as block diagonal bases, the block sizes in the bases B computed by our algorithms can be large. Instead, we compute block angular bases, in which each of the diagonal blocks exhibits a block angular structure recursively. The recursive block angular structure prevents the creation of many long directed paths in the directed graph of the partially computed basis (since such paths would case diagonal blocks of large order to merge), and thus controls the number of nonzeros in the inverse basis. Hence the sparsity in the inverse bases for the column space is a direct result of the objective function that we have employed.

Table 5.5 shows the run time requirements of the algorithms. The results show that the Column algorithm can be slow, especially for problems with large numbers of columns. The Top-down algorithm is much faster, and produces solutions of comparable sparsity in most cases. The runtimes for the Top-down algorithm are dominated by the time for hypergraph

TABLE 5.4
Number of nonzeros per column of the bases, B; and their structural inverses, denoted by B^{-1}; computed by four algorithms: Weighted matching (WM), Column algorithm (COL), Row algorithm with original ordering (ROW), and Top-down algorithm with block size 200 (TD). Each result is the average of ten runs.

	$\|B\|/m$				$\|B^{-1}\|/m$			
	WM	COL	ROW	TD	WM	COL	ROW	TD
truss	2.00	2.00	2.00	2.00	8.50	2.13	2.04	2.09
d2q06c	1.82	1.88	1.99	1.94	2.52	2.24	2.60	2.74
X135	1.04	1.04	1.04	1.04	1.60	1.04	1.04	1.04
gosh	1.53	1.55	1.76	1.92	1.94	1.93	2.22	2.47
cre-b	1.88	1.94	1.94	1.95	2.16	2.27	2.31	2.38
ken07	2.17	2.18	2.25	2.20	7.50	6.37	7.45	6.52
ken11	2.15	2.16	2.19	2.16	10.98	7.98	8.54	8.18
ken13	2.12	2.12	2.15	2.12	13.94	9.60	8.47	9.80
ken18	2.15	2.15	2.19	2.16	20.22	13.64	16.12	13.84
pds02	2.00	2.03	2.06	2.09	13.74	7.81	8.11	7.57
pds06	2.03	2.05	2.12	2.13	17.14	8.37	11.12	8.24
pds10	2.03	2.06	2.13	2.13	19.46	8.53	14.96	8.61
pds20	2.03	2.06	2.13	2.13	21.33	8.51	20.60	8.59
pigs-m	2.48	2.50	2.52	2.51	229.3	29.5	127.3	35.9
pigs-l	2.44	2.45	2.47	2.46	525.2	27.9	235.2	20.2
pigs-v	2.44	2.47	2.48	2.48	65.7	15.1	352.2	14.2

partitioning. The hypergraph partitioner PaToH [24], which we used for the experiments, has been designed to generate partitions with precise definitions of balance and metrics of partition quality. However, in this application, we need a decomposition of the matrix into smaller submatrices, with block sizes bounded for each submatrix. There is no precise definition of balance, and minimizing the cut-size is pursued only to increase the chance of obtaining diagonal blocks with full structural rank. Thus a faster hypergraph partitioner could be used in our application at the cost of increased cut sizes, such as methods based on net intersection graphs [18].

We summarize our results as follows. For generating bases that have sparse inverses, the Column algorithm and the Top-down algorithm are the best performers, especially for the larger problems in the test set. The run times of the Weighted matching algorithm are the lowest, but unfortunately, while it controls the sparsity of the column space basis, it does not control the sparsity in the inverse. The run times of the Column algorithm are high for larger problems. The Top-down algorithm combines sparse inverses in the column space bases with low run time requirements. For many of the linear programs and structural analysis problems, the inverse of a column space basis is sufficiently sparse that computing it explicitly would be a viable option. Even for the multicommodity flow problems, ‘ken’ and ‘pds’, and the animal breeding problems, where the basis inverses are not as sparse as the remaining problems considered, we believe that this approach yields computationally useful results.

Numerical Considerations. We have thus far focused on constructing structurally nonsingular bases, whereas numerical nonsingularity is essential to construct null space bases. While the structural rank is equal to the numerical rank for many sparse matrices, this equality depends on the application that generates the matrices. For instance, in our experiments

ETNA
Kent State University
etna@mcs.kent.edu

TABLE 5.5
The run times of the four algorithms in seconds.

	WM	COL	ROW	TD
truss	0.004	0.151	0.030	0.263
d2q06c	0.007	0.246	0.030	0.362
X135	0.019	1.121	0.030	0.715
gosh	0.016	1.083	0.070	1.242
cre-b	0.140	46.47	0.380	7.997
ken07	0.002	0.364	0.010	0.302
ken11	0.031	19.52	0.220	2.420
ken13	0.222	70.51	0.770	5.156
ken18	9.488	1643	64.73	26.44
pds02	0.001	0.562	0.030	0.570
pds06	0.034	7.291	0.420	2.806
pds10	0.114	19.14	1.260	5.390
pds20	0.813	77.68	5.660	14.23
pigs-m	0.009	3.846	0.120	1.085
pigs-l	0.252	39.65	0.810	3.973
pigs-v	5.319	899.6	15.26	33.60

we observed that in structural mechanics, the structurally nonsingular bases we generated were numerically nonsingular as well. On the other hand, among the LP matrices from the Netlib collection, it is common to find pairs of columns that are multiples of each other, which causes the structurally nonsingular bases to become numerically rank-deficient.

Choosing a basis for a sparse matrix is a problem that has both combinatorial and numerical aspects. However, it is mostly the nonzero structure of the basis that determines the computational costs of operating with this matrix.

In applications where the structural rank is close to the numerical rank, a structurally nonsingular basis can be used as an initial basis, and then it could be augmented by exchanging a few columns to achieve numerical nonsingularity. In applications where the structural rank is a poor approximation to numerical rank, the combinatorial phase and the numerical phase should be interleaved. The algorithms in this paper can be enhanced to achieve numerical nonsingularity in the computed bases. The Row and Column algorithms add a column to the basis if it increases the structural rank. It is easy to have the structural independence test be followed by a numerical independence test, and accept a column to add to the basis only if it increases the numerical rank also. For example, if the LU factorizations (with pivoting) of the partial bases are computed on-the-fly, these may be used for checking linear dependence. Strictly speaking, we would need rank-revealing factorizations for this purpose. However, in optimization contexts, especially at a point far from an optimum solution, LU factorization with pivoting should suffice. Such an approach has been implemented for null-space basis computations [8].

The Top-down algorithm decomposes the matrix A into smaller and disconnected submatrices by removing columns, and seeks a basis within each block. The Column algorithm is used to obtain partial bases from each diagonal block, and thus in this approach too, a numerically nonsingular subset of columns can be computed from each block; if needed, additional columns could be chosen from the coupling columns to augment the partial bases to a basis of the matrix A.

6. Conclusions and Future Work. We have designed and implemented three heuristic algorithms for constructing bases with sparse inverses for the column space of an underdetermined matrix. Our results from extensive tests with the Netlib LP (linear programs) set show that bases with sparse inverses for the column space are more common than what is generally believed. Such column space bases could be used to represent null space bases implicitly and efficiently. It would be worthwhile to investigate algorithms that could deliver explicitly sparse null space bases via the approaches considered here. In the Appendix, we show that block diagonal bases for the column space with block sizes less than three can be computed in polynomial time (if they exist), while if the block size is greater than or equal to three, the problem is NP-complete.

Appendix. Complexity. In this appendix, we investigate the complexity of selecting a column basis B with block diagonal structure. We investigate the complexity for different values of the maximum block size K. When $K = 1$, the problem reduces to finding a diagonal submatrix of A and can be solved by an optimal algorithm that requires time linear in the number of nonzeros in A, $\mathrm{nnz}(A)$. Notice that columns of such a submatrix should have exactly one nonzero. Thus the problem reduces to finding a column with a single nonzero for each row, which requires one pass over all columns to find candidate columns, and a pass over rows to verify. For $K = 2$, the problem is harder, but it still can be solved by a polynomial time algorithm as we show in the next section. However, the problem becomes NP-complete for $K > 2$, the proof of which is presented in the Section A.2.

A.1. Basis with 2×2 Blocks. In this section we show that the problem of finding a block diagonal basis where the block sizes are bounded by 2 can be reduced to the problem of finding a matching in a graph. A basis with 2×2 blocks requires matching pairs of rows with pairs of columns, so that column (row) pairs have nonzeros only at the rows (columns) they are matched to. Observe that if the pairing of rows is fixed, it will be easy to detect if there exists a column-pair that can be matched to each row-pair. However, pairing rows is nontrivial. Our algorithm has two phases. The first phase identifies all candidate row-pairs that might form the rows of a 2×2 block diagonal basis. The outcome of this phase is a graph where each row in the matrix is represented by a vertex, and candidate row-pairs are connected by edges. The second phase chooses a maximum number of row-pairs among all the candidates. Notice that each row can be part of multiple candidate row-pairs, and thus we need to choose a maximum set of mutually disjoint pairs of rows, which corresponds to a maximum matching in the graph. The following theorem formalizes the construction and the result. For clarity of presentation, we assume the number of rows in the matrix is even. We will later relax this assumption.

THEOREM A.1. *Given an $m \times n$ matrix $A = (a_{ij})$, where m is an even integer, define a graph $G = (V, E)$ so that*

- *Each row r_i in A is represented by a vertex v_i;*
- *The edge $(v_i, v_j) \in E$ if and only if there are two columns k and t, with nonzeros only in rows i and j, such that the submatrix $\begin{pmatrix} a_{ik} & a_{it} \\ a_{jk} & a_{jt} \end{pmatrix}$ is non-singular.*

Then the matrix A has a block-diagonal basis with block size at most 2 if and only if G has a perfect matching.

Note that the non-singularity of the 2×2 submatrix in the Theorem would still permit one or two (in the latter case, one from each column) of the four submatrix elements to be zero. Figure A.1 illustrates the construction of this graph.

ETNA
Kent State University
etna@mcs.kent.edu

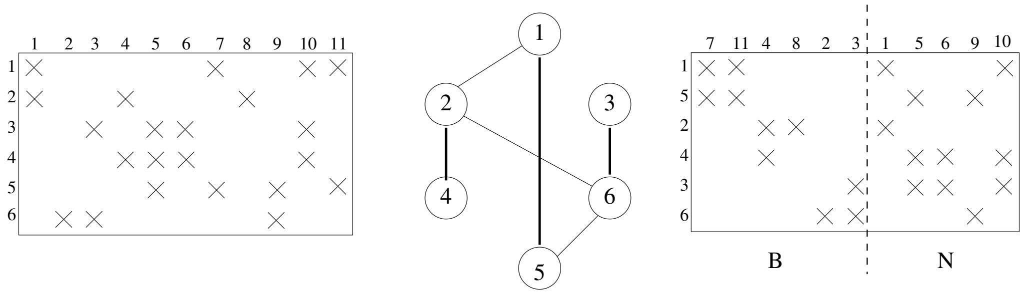

FIG. A.1. *A matrix and the graph defined in Theorem A.1.*

Proof. *Sufficiency*: A perfect matching gives $N/2$ vertex disjoint edges in G. By construction, an edge in G is defined by two columns in A, with nonzeros only in two rows such that the two columns and rows form a non-singular 2×2 submatrix. Thus a perfect matching provides m columns to form a block diagonal basis with block sizes at most 2×2.

Necessity: If there is a block diagonal basis for A with block sizes at most 2×2, then the columns of each 2×2 block will contribute an edge to G; columns with single nonzeros could be arbitrarily paired, and by the construction of G, there is an edge in G corresponding to each such pair of columns. Hence G has a matching of size $m/2$, since these edges need to be vertex-disjoint. ☐

COROLLARY A.2. *The problem of finding a* 2×2 *block diagonal basis can be reduced to the maximum matching problem.*

Proof. Theorem A.1 shows the reduction when the number of rows is even. If the number of rows is odd, we can add a pseudo-row, a pseudo-column, and a pseudo-nonzero at their intersection. This new matrix has an even number of rows, and has a block diagonal basis with blocks of size at most 2×2 if and only if the original matrix has such a basis. ☐

In some situations we might wish to construct a block diagonal basis with block sizes bounded by two, which has the maximum number of 1×1 blocks. This can be achieved by solving a weighted maximum matching problem, which gives a maximum cardinality matching with maximum sum of weights of edges in the matching. We can assign weights to edges of the graph described in Theorem A.1 so that an edge has a higher weight if it is defined by two columns, each having only a single nonzero.

A.2. Complexity for Larger Block Sizes. The reduction in the previous section does not yield polynomial solutions when $K > 2$. Remember that the first phase identifies candidate blocks for the basis, and the second phase chooses a mutually disjoint subset of these blocks. Even though the candidate blocks ($K \times K$ or smaller blocks that are nonsingular) can be identified in $O(N^K)$-time, finding a mutually disjoint subset corresponds to the hypergraph matching problem when $K \geq 3$, which is known to be NP-complete [10].

The reader will find it useful to recall the hypergraph terminology described in Section 4.1. In this section, we will show that the hypergraph matching problem can be reduced to the problem of finding a block diagonal basis, to prove the NP-hardness of our problem. The hypergraph matching problem is defined as follows.

Given a collection HE of subsets of V, and a positive integer $K \leq |HE|$, decide if HE contains K mutually disjoint sets.

The hypergraph matching problem is known to be NP-complete, even when all sets in HE have no more than 3 vertices [10]. For simplicity of presentation, our NP-completeness proof will use a reduction from finding a *perfect* matching in a 3-regular hypergraph. A

ETNA
Kent State University
etna@mcs.kent.edu

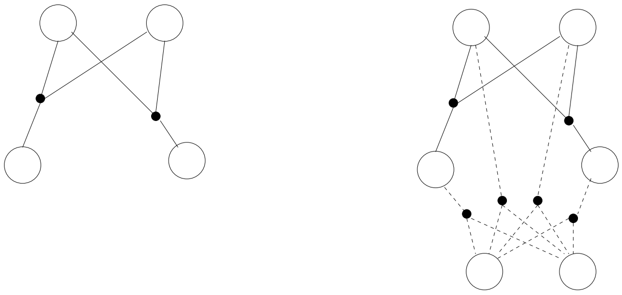

FIG. A.2. *A hypergraph HG and its extended hypergraph $HG'(1)$. Hyperedges are represented by dark circles, vertices by open circles, and lines connect each hyperedge to vertices belonging to it. In this example, $K = 1$; the extended hypergraph $HG'(1)$ has $|V| - 3K = 4 - 3 = 1$ pair of auxiliary vertices, and four hyperedges (marked with dashed lines) connecting the auxiliary vertices to all other vertices.*

hypergraph is 3-regular, if all hyperedges contain exactly three vertices. A matching is *perfect* if every vertex is adjacent to a hyperedge in the matching. Below we first show how to transform a matching problem in a hypergraph to a perfect matching problem in a related hypergraph. Without loss of generality, we assume there are no hyperedges that cover the same vertex sets (i.e., duplicated hyperedges).

DEFINITION A.3. *Given a hypergraph $HG = (V, HE)$ and a positive integer k, the extended hypergraph $HG'(k) = (V', HE')$ is defined as follows.*

- *Vertex set V' contains the original vertices in V and $(|V| - 3k)$ pairs of auxiliary vertices, i.e.,*

$$V' = V \cup \Big(\bigcup_{i=1}^{|V|-3k} \{u_i, v_i\} \Big).$$

- *The hyperedge set HE' contains the original hyperedges in HE and $|V|$ hyperedges for every pair of auxiliary vertices that connects the pair to every other vertex in V. The formal definition follows.*

$$HE' = HE \cup \Big(\bigcup_{i=1}^{|V|-3k} \cup_{w \in V} \{u_i, v_i, w\} \Big).$$

Figure A.2 illustrates an example of the extended hypergraph construction.

LEMMA A.4. *The hypergraph HG has a matching of size K if and only if the extended hypergraph $HG'(K)$ has a perfect matching.*

Proof. The main observation in the proof is that each one of $|V| - 3K$ auxiliary pairs can be matched with any vertex in V.

- *Sufficiency:* A matching of size K in HG covers $3K$ vertices in V. Each of the remaining $|V| - 3K$ vertices can be matched through a hyperedge that connects it to an unmatched auxiliary vertex pair.

- *Necessity:* The auxiliary pairs cover at most $|V| - 3K$ vertices in V, thus the remaining $3K$ vertices must be matched through hyperedges in HE.

□

THEOREM A.5. *It is NP-complete to determine if a matrix has a block-diagonal basis with block sizes bounded by three.*

Proof. We will use a reduction from the 3-regular hypergraph matching problem. As shown in Lemma A.4 a matching problem on HG can be reduced to a perfect matching problem on its extended hypergraph HG'. Observe that if HG is 3-regular, then HG' is 3-regular as well. For simplicity of presentation we will describe the reduction from the extended graph.

Given a hypergraph HG and a bound on the size of the matching, let $HG'(K) = (V, HE)$ be its extended hypergraph. Define a $|V| \times 3|HE|$ matrix A so that each vertex in V is represented by a row in A, and each hyperedge is represented by three columns with the same nonzero structure, with nonzeros in rows corresponding to the vertices of the hyperedge. This gives a 3×3 dense submatrix in A, and we can assign numerical values to make this submatrix nonsingular (e.g., assign them algebraically independent values).

We claim that the extended hypergraph $HG'(K)$ has a perfect matching if and only if A has a block diagonal basis with $3|V|$ nonzeros.

- *Sufficiency:* A perfect matching in $HG'(K)$ gives $|V|/3$ vertex disjoint hyperedges, and each hyperedge is represented by three columns that form a 3×3 nonsingular block with the rows corresponding to vertices of the hyperedge. By definition of a matching, the hyperedges are vertex disjoint, and thus the blocks are non-overlapping, which gives us a basis of $|V|/3$ blocks, each of size 3×3.
- *Necessity:* Notice that all columns of A have three nonzeros, which forces blocks of the basis of size at least three. A basis with $3|V|$ nonzeros is achieved only when the basis consists of 3×3 blocks. Such a solution requires $|V|/3$ row disjoint columns in A, which define $|V|/3$ vertex-disjoint hyperedges in $HG'(K)$.

This proves the NP-hardness of the problem. Since the correctness of a solution can be verified in polynomial time, the problem is NP-complete. ◻

Acknowledgments. We thank Erik Boman of Sandia National Labs and two anonymous reviewers for their detailed, helpful comments which have improved the paper.

REFERENCES

[1] C. ASHCRAFT. Personal communication, 2003.
[2] C. AYKANAT, A. PINAR, AND Ü.V. ÇATALYÜREK, *Permuting sparse rectangular matrices into block-diagonal form*, SIAM J. Sci. Comput., 25 (2004), pp. 1860–1879.
[3] M. BENZI, G. H. GOLUB, AND J. LIESEN, *Numerical solution of saddle point problems*, Acta Numer., 14 (2005), pp. 1–137.
[4] M. W. BERRY, M. T. HEATH, I. KANEKO, M. LAWO, R. J. PLEMMONS, AND R. C. WARD, *An algorithm to compute a sparse basis of the null space*, Numer. Math., 47 (1985), pp. 483–504.
[5] G. BIROS AND O. GHATTAS, *Parallel Lagrange-Newton-Krylov-Schur methods for PDE-constrained optimization. Part I: the Krylov-Schur solver*, SIAM J. Sci. Comput., to appear.
[6] E. CHOW, T. A. MANTEUFFEL, C. TONG, AND B. K. WALLIN, *Algebraic elimination of slide surface constraints in implicit structural analysis*, Int. J. Numer. Meth. Engg., 57 (2003), pp. 1129–1144.
[7] T. F. COLEMAN AND A. POTHEN, *The null space problem I. Complexity*, SIAM J. Alg. Disc. Meth., 7 (1986), pp. 527–537.
[8] ———, *The null space problem II. Algorithms*, SIAM J. Alg. Disc. Meth., 8 (1987), pp. 544–563.
[9] T. H. CORMEN, C. E. LEISERSON, AND R. L. RIVEST, *Introduction to Algorithms*, MIT Press and McGraw-Hill, Cambridge, MA, 1990.
[10] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability*, W.H. Freeman and Company, New York, N.Y., 1979.
[11] D. M. GAY, *Electronic mail distribution of linear programming test problems*. Mathematical Programming Society COAL Newsletter, 1985.
[12] J. R. GILBERT, *Predicting structure in sparse matrix computations*, SIAM J. Matrix Anal. Appl., 15 (1994), pp. 62–79.
[13] J. R. GILBERT AND M. T. HEATH, *Computing a sparse basis for the null space*, SIAM J. Alg. Disc. Meth., 8 (1987), pp. 446–459.

[14] M. T. HEATH, R. J. PLEMMONS, AND R. C. WARD, *Sparse orthogonal schemes for structural optimization using the force method*, SIAM J. Sci. Comput., 5 (1984), pp. 514–532.

[15] J. E. HOPCROFT AND R. M. KARP, *An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs*, SIAM J. Comput., 2 (1973), pp. 225–231.

[16] I. KANEKO, M. LAWO, AND G. THIERAUF, *On computational procedures for the force method*, Int. J. Numer. Meth. Engg., 18 (1982), pp. 1469–1495.

[17] G. KARYPIS AND V. KUMAR, *Multilevel k-way hypergraph partitioning*, in Proceedings of 36th ACM/IEEE Conference on Design Automation, ACM Press, 1999, pp. 343–348.

[18] A. PINAR AND C. AYKANAT, *An effective graph model to decompose linear programs for parallel solution*, Lecture Notes In Computer Science, 1184 (1996), pp. 592–601.

[19] A. PINAR, Ü.V. ÇATALYÜREK, C. AYKANAT, AND M. C. PINAR, *Decomposing linear programs for parallel solution*, Lecture Notes In Computer Science, 1041 (1995), pp. 473–482.

[20] R. J. PLEMMONS AND R. E. WHITE, *Substructuring methods for computing the nullspace of equilibrium matrices*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 1–22.

[21] A. POTHEN, *Sparse Null Bases and Marriage Theorems*, PhD thesis, Cornell University, Ithaca, New York, 1984.

[22] J. M. STERN AND S. A. VAVASIS, *Nested dissection for sparse nullspace bases*, SIAM J. Matrix Anal. Appl., 14 (1993), pp. 766–775.

[23] A. TOPCU, *A contribution to the systematic analysis of finite element structures through the force method*, PhD thesis, University of Essen, Essen, Germany, 1979. (in German).

[24] Ü.V. ÇATALYÜREK AND C. AYKANAT, *PaToH: A Multilevel Hypergraph Partitioning Tool for Decomposing Sparse Matrices and Partitioning VLSI Circuits*, Tech. Report BU–CE–9915, Department of Computer Engineering and Information Science, Bilkent University, Turkey, 1999.

[25] B. VASTENHOUW AND R. H. BISSELING, *A two-dimensional data distribution method for parallel sparse matrix-vector multiplication*, SIAM Rev., 47 (2005), pp. 67–95.