

# On computing $Ax$ and $\pi^T A$ , when $A$ is sparse

A.J. Hoffman

*IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598, USA*

W.R. Pulleyblank

*IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598, USA*

J.A. Tomlin

*IBM Almaden Research Center, San Jose, CA 95120, USA*

*Dedicated to Ted Rivlin with admiration and friendship*

This note describes a method for calculating  $Ax$ , where  $A$  is sparse and stored by columns. The method is a practical application of a well-known theorem on edge-coloring of bipartite graphs and an algorithm originally used to prove theorems about edge-coloring. We will describe the method in the context of linear programming, where we have used and tested it, although it is clear that it is more widely applicable.

## 1. Introduction

This note is concerned with vectorizing the matrix-vector product calculation  $Ax$  for a sparse matrix  $A$  such as is typically encountered in the linear programming (LP) problem:

$$\min_x \sum_j c_j x_j,$$

subject to:

$$\begin{aligned} Ax &= b, \\ l_j &\leq x_j \leq u_j. \end{aligned}$$

The reader is assumed to have some familiarity with linear programming (see e.g. [2]). The problem of vectorizing multiplication of  $A$  on the left, i.e.  $\pi^T A$  is discussed in detail by Forrest and Tomlin [4,5], with particular reference to the critical part it plays in the simplex method. Indeed, this calculation (or some subset of it) must be carried out at each iteration of the simplex method to “price” the columns for introduction into the basis. Thus it may be carried out thousands of times on medium to large problems, and its efficiency is usually critical. Multiplication on the right,  $Ax$ , however, is only needed occasionally by the simplex method, to check on the feasibility of the current solution.

The situation is somewhat different for interior methods for LP (see e.g. [10] and [6]). There the number of iterations is much smaller, but generally involve as many or more multiplications on the right as the left. Since we often wish to follow interior point optimization by iterations of the simplex (or a simplex-like) method, we wish to preserve any data structures which facilitate multiplication on the left, but now it is also of interest to increase the efficiency of the  $Ax$  calculation. To see how this might be done, we begin by reviewing the  $\pi^T A$  procedure.

## 2. Multiplication on the left

All pricing schemes for the simplex method call for computing many sparse inner products:

$$d_j = c_j + \sum_{i|a_{ij} \neq 0} \pi_i a_{ij}.$$

Forrest and Tomlin [4] reported that on the IBM 3090 vector facility (circa 1988) this calculation vectorized at 4 cycles per nonzero  $a_{ij}$ , with a startup time of approximately 180 cycles. Scalar coding to perform the same task took approximately 18 cycles an element with a startup time of 15 cycles. The "break even" point for vectorizing the inner product is then at about 12 nonzero elements per column. Unfortunately, the average number of nonzeros per column in real world LP models is smaller than this – usually about 4 to 8. Clearly some reorganization of the calculations must be carried out to take advantage of vector processing.

The method described by Forrest and Tomlin [4] makes use of the observation that there are (almost inevitably) large numbers of columns having the same number of nonzeros, and these can be grouped together. The general approach is as follows:

- Store columns with large numbers of entries (and few instances of that number) separately as these can be vectorized in the obvious way.
- Sort the rest of the matrix so that all columns with the same number of entries are stored together in blocks.

Thus we may keep the columns with (say)  $s$  nonzeros in dense  $s \times z$  blocks:

$$\begin{pmatrix} a_{i_{11}1} & a_{i_{12}2} & a_{i_{13}3} & \cdots & a_{i_{1z}z} \\ a_{i_{21}1} & a_{i_{22}2} & a_{i_{23}3} & \cdots & a_{i_{2z}z} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{i_{s1}1} & a_{i_{s2}2} & a_{i_{s3}3} & \cdots & a_{i_{sz}z} \end{pmatrix} \quad (2.1)$$

where  $i_{kj}$  is the  $k$ th index of a nonzero in column  $j$  of the block. These indices are stored separately in an integer array:

$$\begin{pmatrix} i_{11} & i_{12} & i_{13} & \cdots & i_{1z} \\ i_{21} & i_{22} & i_{23} & \cdots & i_{2z} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ i_{s1} & i_{s2} & i_{s3} & \cdots & i_{sz} \end{pmatrix}. \quad (2.2)$$

The reduced cost vector for such a block, say DJ, may then be computed in efficient vector fashion by initializing it to the appropriate cost row values and for each  $k = 1, \dots, s$  performing the vector operations:

- (1) LOAD the vector of indices

$$\langle i_{k1}, i_{k2}, \dots, i_{kz} \rangle$$

- (2) Gather the vector of  $\pi$ -values

$$PI = \langle \pi_{i_{k1}}, \pi_{i_{k2}}, \dots, \pi_{i_{kz}} \rangle$$

using the LOAD INDIRECT facility.

- (3) Point to the  $k$ th row of the block,  $AROW = \langle a_{i_{k1}}, a_{i_{k2}}, \dots, a_{i_{kz}} \rangle$ , in memory and MULTIPLY AND ADD the element-by-element product of this vector with PI to update the reduced costs:

$$DJ \leftarrow DJ + PI * AROW$$

See IBM Corp. [7] for details of these vector instructions. Note that the operations are carried out row-wise; that is on vectors whose length is the number of columns in the block, *not* the number of rows (i.e. nonzeros per matrix column). To avoid cache misses these blocks of values and their associated indices should be stored row-wise. It is also useful to limit the number of columns in a block to be at most the length of the machine's vector registers; thus there may be several blocks corresponding to the same number of column nonzeros.

### 3. Multiplication on the right

In interior point methods for LP where  $Ax$  must be computed a significant number of times, it would be possible to simply use the same vector blocking scheme on  $A^T$  and compute  $x^T A^T$ . However, this means that yet another copy of  $A$  must be stored, for a relatively limited benefit in terms of the total computing effort required. It is therefore of some interest to ask whether the vector block form of  $A$  is useful here. There is a difficulty which must be overcome.

Consider for simplicity a matrix which can be represented by one vector block (2.1) and the slightly more general operation

$$v \leftarrow v + Ax.$$

Given  $s < z$ , we would like to again use the rows of the block as vectors, which would lead to the following procedure:

- (0) LOAD the vector X of  $x$ -values.  
 (1) For each  $k$ , LOAD the vector of indices

$$\langle i_{k1}, i_{k2}, \dots, i_{kz} \rangle$$

- (2) Gather the vector of
- $v$
- values

$$V = \langle v_{i_{k1}}, v_{i_{k2}}, \dots, v_{i_{kz}} \rangle$$

using the LOAD INDIRECT facility.

- (3) Point to the
- $k$
- th row of the block,
- $AROW = \langle a_{i_{k1}1}, a_{i_{k2}2}, \dots, a_{i_{kz}z} \rangle$
- , in memory and MULTIPLY AND ADD the element-by-element product of this vector with
- $X$
- to update the target vector:

$$V \leftarrow V + AROW * X.$$

- (4) Store the vector of updated
- $v$
- values using the STORE INDIRECT facility.

The difficulty with this procedure is that the vector of indices  $\langle i_{k1}, i_{k2}, \dots, i_{kz} \rangle$  may not have distinct entries, in which case the procedure will lead to indeterminate (but certainly incorrect) results. However, if the indices corresponding to each row of the block are distinct, the procedure will work, and quite efficiently vectorize the calculation. We therefore consider the possibility of arranging for this to happen as often as possible.

Let us begin by noting that the order of the nonzeros within any column of the block has no effect on the  $\pi^T A$  calculation and can be changed to facilitate the  $Ax$  calculation at will. Our approach is to reorder the elements within each column of the block to maximize the number of rows of the block which have all distinct non-zero (matrix) row indices. We first establish the conditions under which a block can be rearranged so that all the rows satisfy the distinct index property. This turns out to be a direct application of the König Line Coloring Theorem (see Lovász and Plummer [9], p. 37), which states that the edges of a bipartite graph can be colored with  $s$  colors, in such a way that no two adjacent edges receive the same color, if and only if no node is incident with more than  $s$  edges.

### Theorem

The entries  $i_{kj}$  in a  $s \times z$  block (2.2) can be re-ordered within each column in such a way that each row of the block has distinct entries if and only if no entry appears more than  $s$  times in the block.

### Proof

The necessity is obvious. To prove the sufficiency, construct a bipartite graph with one set of nodes  $V$  corresponding to the individual index values appearing in the block and the other set  $C$  corresponding to the columns of the block. Create an edge joining nodes  $v \in V$  and  $j \in C$  if an entry corresponding to  $v$  appears in column  $j$ . Every node in  $C$  has degree exactly  $s$  and, by hypothesis, every node in  $V$  has degree at most  $s$ . By the König Line Coloring Theorem, the edges of the graph can be colored with  $s$  colors so that no adjacent edges receiving the same color. Arbitrarily assign the  $s$  colors to the rows of the matrix. If we then permute the indices in each column so that each is in the row of the block corresponding to the color of the corresponding edge, there can be no repetition in any row of the block.  $\square$

There are several different ways of proving König's Theorem. One proof, due to Faber et al. [3] (see also Lovász and Plummer [9], p. 286) starts with any coloring of the edges of the bipartite graph using  $s$  colors, and then "corrects" defects in the coloring one after another by switching colors of edges belonging to two-colored alternating paths until a proper coloring is obtained. By "proper" we mean one for which all edges incident with each node receive different colors. This method can easily be adapted to yield an efficient algorithm for permuting the entries in each column as we require. The correctness of this algorithm also provides a direct proof of the preceding theorem.

### Algorithm

- (1) Process each column  $j$  of the block in sequence, by examining each entry  $i_{kj}$  in turn, from top to bottom. We define the *processed part* of each row  $h$  to consist of all its entries which have already been processed. If the value  $i_{kj}$  does not appear in the processed part of row  $k$ , then the processing of  $i_{kj}$  is complete and we go on to the next entry.
- (2) If the value  $i_{kj}$  already appears in the processed part of row  $k$ , attempt a simple swap by looking for an entry below  $i_{kj}$  in column  $j$  which does not appear in the processed part of row  $k$ . If such an entry exists, swap it with the one currently being considered. If none exist, carry out the following steps.
- (3) The value  $v = i_{kj}$  currently occurs at least twice in row  $k$ , once in the processed part of row  $k$ , and the entry  $i_{kj}$ . Let  $h$  be the index of the column in the processed part for which  $i_{kh} = v$ . Since  $v$  occurs at most  $s$  times in the matrix, there exists some row  $l$  (different from  $k$ ) for which the processed part does not contain the value  $v$ .
- (4) Let  $v' = i_{lh}$ . Swap the entries  $i_{kh}$  and  $i_{lh}$ . We now have a single entry with value  $v$  in the part of row  $k$  up to and including column  $j$ . If the value  $v'$  only occurs once in the processed part of row  $k$ , then we are done and we resume the scan in Step (1). Otherwise,  $v'$  occurs twice in the processed part of row  $k$ , once in column  $h$  and once in some other column  $h'$ . Set  $h = h'$  and  $v = v'$ . Repeat Step (4).

We make two remarks. First, note that when we do the sequence of swaps described in step (4), we always use the same two rows  $k$  and  $l$ . Second, we will never swap the pair of entries in a column more than once before choosing a new entry  $i_{kj}$ . One way that this can be seen is as follows:

Construct a directed graph whose nodes are the entries in the processed parts of rows  $k$  and  $l$ , plus  $i_{kj}$ . From the node corresponding to each entry in row  $l$ , construct an arc oriented towards the node for the entry in row  $k$  in the same column. Whenever we have  $i_{lj_1} = i_{kj_2}$ , for two columns  $j_1, j_2$ , we add an arc oriented from the node for  $i_{lj_1}$  to the node for  $i_{kj_2}$ . Finally add an arc from the node for  $i_{kj}$  to the node for  $i_{lh}$ , where  $h$  is the index of the other column containing  $v$ . We now have a directed graph in which every node has at most one arc directed out from it and at most one arc directed into it. Therefore it decomposes into a number of node disjoint

paths and cycles. The sequence of swaps performed in Step (4) will correspond precisely to following the arcs in the path in this graph starting with the node for  $i_{kj}$ , where each swap corresponds to moving two arcs. When we get to the end of the path, the sequence of swaps will terminate.

As a consequence, when we are processing the  $j$ th column, we will have to perform at most  $j$  swaps while carrying out step (4) for an entry  $i_{hj}$ . With some care, this can be implemented so that the worst case performance of our algorithm is  $O(sz^2)$ , where the block has  $s$  rows and  $z$  columns (assuming  $z \geq s$ ). In practice, we normally require much less time than this.

There remains the question of what to do with a block if the conditions of the theorem are *not* satisfied. One answer is to permute frequently occurring indices into the last row of the block, reducing  $s$  until the conditions are satisfied – if possible. The procedure we have implemented is as follows:

#### Heuristic

Let the count of the number of times row index  $r_k$  appears be  $c_{r_k}$ . For each column  $j = 1, \dots, z$  of the block find the index with the largest count (breaking ties by magnitude of the index), permute this index into the last row ( $s$ ), and reduce the count for that index by 1. When the pass is completed, reduce  $s$  for the block by 1 and see if the new maximum  $c_{r_k}$  is  $\leq s$ .

When this preprocessing is complete, we use the above algorithm to compute the product of  $x$  by the first  $s$  rows of  $A$ , then add in the product of  $x$  with the last rows.

#### 4. Computational procedure

To process a matrix we examine each vector block in turn. If all the row indices are unique, then all the rows of the block are trivially vectorizable and we can proceed to the next. If not, test to see whether the maximum row count is at most  $s$ . If it is not, carry out the heuristic procedure above to permute row indices with high counts to the bottom, reducing  $s$  at each pass. When the conditions of the theorem are satisfied, execute the algorithm described above.

#### 5. Computational experience

The performance of the algorithm and especially its success in reordering the vector blocks are problem dependent. Table 1 gives details of a number of test problems. The first five are standard problems from a test set well-known in the linear programming community. The last three are structured multi-product, multi-plant, multi-time-period production planning models. The data given in the table does not refer to the problems as originally specified, but rather to the problems after preprocessing; the change should not be significant in judging the virtues of our

Table 1  
Test problem characteristics.

Model	Rows	Columns	Nonzeros
BandM	185	340	1670
Degen2	381	473	3580
25FV47	715	1484	9994
PILOTS	1374	3361	40757
Degen3	1412	1727	24465
Mod41	34952	94752	210211
Mod42	40022	110475	244439
MPS4	90482	219249	502943

method. All problems were run using the IBM Optimization Subroutine Library (OSL) interior point predictor-corrector algorithm on a 3090J vector processor.

It will be seen from table 2 that the  $Ax$  calculation is performed considerably more often than the (vectorized)  $\pi^T A$  calculation and takes more time per calculation, by a factor of roughly 2 to 3. All times are expressed in seconds. Comparing

Table 2  
Standard  $Ax$  and vector  $\pi^T A$  times.

Model	Vector Block Elements	Number of $Ax$ calc	Total $Ax$ time	Number of $\pi^T A$ calc	Total $\pi^T A$ time
BandM	1272	79	0.056	46	0.018
Degen2	3005	68	0.101	40	0.028
25FV47	9300	134	0.486	79	0.099
PILOTS	32123	184	2.440	109	0.523
Degen3	12106	93	0.676	53	0.186
Mod41	210211	213	17.627	127	3.793
Mod42	244439	278	26.746	166	5.863
MPS4	502943	408	82.826	244	18.187

Table 3  
Standard and reordered  $Ax$  times.

Model	Reordered Elements	Reordering Time	New $Ax$ time	Old $Ax$ time
BandM	991	0.007	0.043	0.056
Degen2	2765	0.020	0.062	0.101
25FV47	2592	0.044	0.416	0.486
PILOTS	19587	0.216	1.662	2.440
Degen3	9726	0.089	0.462*	0.676
Mod41	124797	1.340	11.209	17.627
Mod42	146487	1.289	17.169	26.746
MPS4	397368	3.361	43.543	82.826

tables 1 and 2 we see that most of the nonzeros were placed in vector blocks for the  $\pi^T A$  calculation.

We applied our heuristic to remove row indices with too many nonzeros from the coloring algorithm. In table 3, the column headed Reordered Elements gives the number of elements which survived and were colored by the algorithm. The column headed New  $Ax$  Time does not include the time spent processing the removed rows. Comparing Tables 2 and 3 we see that the rate of success in reordering elements within the blocks to vectorize  $Ax$  varies from mediocre (only 28% for 25FV47) to high (90% for Degen2), with good results for the large models. From table 3 we see that the time for  $Ax$  calculations is indeed improved, by up to a factor of nearly two for the largest problem. Furthermore, the reordering algorithm is fast enough that in every case, even when little improvement is attained, this improvement more than compensates for the reordering time. We therefore conclude that it is a practical mechanism for production use. We expect savings to be most significant when models are large and/or highly structured.

The computational result highlighted\* for Degen3 in table 3 brings out one final minor computational point. The rounding characteristics of vector arithmetic can be different than a sequence of scalar operations. Furthermore, operations are carried out in a different order for both right and left multiplication when the vector blocks are reordered. These slight differences may result in an algorithm taking a different solution path and did indeed result in Degen3 being solved in one fewer interior point iterations, which reduces the number of  $Ax$  operations from 93 to 88. However, even when this is allowed for, the reordering still produces a significant net improvement.

## 6. Conclusion

This note has considered only the very limited problem of reordering vector blocks without in any way modifying their design for efficient  $\pi^T A$  processing. Clearly, there are more general problems which could be posed in terms of matrix storage methods which would give vector efficiency for both right and left multiplication. We have not considered them here. Similarly, the heuristic we have used to obtain blocks satisfying the conditions of the theorem is only one of a number of approaches which could have been taken.

A related problem is that of determining, for a sparse matrix, whether there exists a partition of the columns into  $k$  blocks, each containing at most  $z$  columns, so that each row has at most  $s$  nonzeros in each block. If the number of columns of  $A$  containing  $s$  nonzeros is much greater than  $s$  and  $z$ , this would be helpful, as we could then use the vector processor on our problem in pieces. However, this problem is *NP*-complete, which can be seen as follows.

Colbourn [1] showed that the problem of determining whether the edge set of a graph can be partitioned into triangles is *NP*-complete. Let  $A$  be the node-edge incidence matrix of a graph  $G = (V, E)$ . Then  $s = 2$ . Let  $z = 3$  and  $k = |E|/3$ . Then the

problem of partitioning the edge set of  $G$  into triangles is precisely our matrix partitioning problem.

## References

- [1] C.J. Colbourn, *The complexity of computing partial Latin squares*, *Discrete Applied Mathematics* 8 (1984) 25–30.
- [2] G.B. Dantzig, *Linear Programming and Extensions* (Princeton Univ. Press, Princeton, NJ, 1963).
- [3] V. Faber, A. Ehrenfeucht and H.A. Kierstead, A new method of proving theorems on chromatic index, Technical report LA-UR-82-661, Los Alamos National Laboratory, Los Alamos, NM (1982).
- [4] J.H.J. Forrest and J.A. Tomlin, Vector processing in simplex and interior methods for linear programming, *Annals of Operations Res.* 22 (1990) 71–100.
- [5] J.J.H. Forrest and J.A. Tomlin, Implementing the simplex method for the Optimization Subroutine Library, *IBM Systems J.* 31, No. 1 (1992) 11–25.
- [6] J.J.H. Forrest and J.A. Tomlin, Implementing interior point linear programming methods in the Optimization Subroutine Library, *IBM Systems J.* 31, No. 1 (1992) 26–38.
- [7] IBM Corp. *IBM System/370 Vector Operations*, Publication Number SA22-7125-3 (1988).
- [8] IBM Corp. *Optimization Subroutine Library, Guide and Reference, Release 2, Third Edition*, Publication Number SC23-0519-2 (1991).
- [9] L. Lovász and M.D. Plummer, *Matching Theory* (North-Holland, Amsterdam, 1986).
- [10] L.J. Lustig, R.E. Marsten and D.F. Shanno, On Implementing Mehrotra's Predictor-Corrector Interior Point Method for Linear Programming, *SIAM J. Optimization* 2 (1992) 435–449.