STORING MATRICES ON DISK FOR EFFICIENT ROW AND COLUMN RETRIEVAL

James R. Bitner

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712

TR-195 April 1982

ABSTRACT:

We study the problem of storing a matrix on disk assuming that the only accessing operation allowed is retrieving an entire row or column. The cost for retrieving a row or column is the number of different pages containing elements of that row or column. The cost of a matrix is the sum of the cost for retrieving each row and column. We give a non-obvious, non-trivial lower bound on the cost of storing a matrix. We show the bound is tight by giving one algorithm that asymptotically (for large matrices) achieves this bound if the page size is in a given set of integers, and another that achieves it if the page size is not in the set. We also analyze the asymptotic fraction of disk space wasted by these algorithms.

ACKNOWLEDGEMENTS: I would like to thank Jay Misra and Andy Sherman for their helpful discussions on this paper.

1. Introduction

This paper concerns methods of storing a matrix on disk such that the "cost" of "accessing" the matrix is minimized. Early work on this topic was done by McKellar and Coffman [1]. (Assume the disk is divided into pages which can hold $\underline{s}$ elements.) They considered two methods of matrix storage:

(1) "row" storage where each row is stored on as many pages as required. If the number of columns is not a multiple of s, the leftover space is wasted.

(2) "submatrix" storage where the matrix is broken up into submatrices of dimension $\lfloor\sqrt{s}\rfloor \times \lfloor\sqrt{s}\rfloor$, and each is stored on a separate page.

They evaluated both storage methods for performing the standard algorithms for matrix traversal, transposition, addition, multiplication and inversion and for Gaussian Elimination. Their cost measure was the number of page faults incurred. They concluded that submatrix storage was almost always much cheaper than row storage. However, Moler [2] contested this conclusion because the pivoting operation required in Gaussian Elimination drastically reduces the advantage of submatrix storage.

Fisher and Probert [3] combined the idea of submatrix storage with Strassen's technique for matrix multiplication [4] to show that multiplying two N x N matrices can be done using a total of $O(N^{2.81})$ page fetches, only a constant times the number of multiplications required. Thus, the page fetches are used very efficiently.

The previous work raises an interesting question: can we talk about a matrix storage method that minimizes the "cost" of "accessing" a matrix? Can we prove that submatrix storage is "optimal"? This is the question addressed by this paper.

Our result is to prove a non-trivial non-obvious lower bound on the cost of accessing a matrix and prove that it is achievable by demonstrating algorithms (similar to the submatrix storage method) which attain the lower bound.

Naturally, for the problem to be tractable we must adopt simple but realistic definitions for "cost" and "accessing". The cost criterion will be the same as in [3], that is, the number of distinct pages that must be fetched to do an "access". This gives an approximation to the number of page faults. An access will be defined as retrieving an arbitrary row or column of the matrix with every row and column having equal probability of being retrieved. Note that the problem is not interesting if only rows (or columns) are to be retrieved because the optimal method is trivially the row (or column) storage method.

We now define the problem precisely.

Definition: The _cost for retrieving a row or column_ is the number of different pages that must be read in from disk to obtain all elements of that row or column. The _cost of a matrix_ is defined to be the cost of retrieving each row and column, summed over all rows and columns. This is motivated by the assumption that each row or column of the matrix must be retrieved equally often. We also use the phrase "_cost of an algorithm_" to mean the cost of a matrix stored according to a given algorithm.

Notation: _m_ and _n_ will always denote respectively the number of rows and columns in the matrix. _s_ will denote the maximum number of matrix elements that can be stored on a page on disk.

Definition: Let COST(m,n,s) be the cost of a given algorithm and let $\mathrm{COST}^*(m,n,s)$ be the optimal cost. Then the algorithm is _asymptotically optimal for s_ iff

$$\lim_{m,n \to \infty} \mathrm{COST}(m,n,s)/\mathrm{COST}^*(m,n,s) = 1.$$

Note that we are interested in an algorithm which is asymptotically optimal for some _fixed s_.

Definition: Let WASTE(m,n,s) be the number of unused locations in partially full pages for a given algorithm. Then $\lim_{m,n \to \infty} \mathrm{WASTE}(m,n,s)/mn$ is the _asymptotic fraction of space wasted_ by the algorithm.

This paper has the following organization: Section 2 will give a lower bound (approximately $2mn/\sqrt{s}$) on the cost of storing a matrix. Section 3 will give an algorithm (Algorithm A) that asymptotically achieves the lower bound for all s in a specified set (see Theorem 2.2). Section 4 gives Algorithm B, which is asymptotically optimal for all s not in this set. The fact that two different strategies must be used to achieve the optimum is dictated by the form of the lower bound (see Theorem 2.1) which actually is the minimum of two quantities. One quantity is minimized by Algorithm A and the other by Algorithm B. Using the condition provided in Theorem 2.2 will allow us to easily tell which algorithm will give the best results for a given s. We also analyze the asymptotic fraction of space wasted by these algorithms. For Algorithm A, it is at most $1/\lfloor\sqrt{s}\rfloor$ (which is 4 or 5% for reasonable s), and for Algorithm B, it is zero. However, the storage scheme using Algorithm B is more complicated, requiring more calculation to determine which pages contain elements of a desired row or column.

Finally, we note that the following problem which is similar in spirit to ours has been studied in [5, 6, 7]. Given an N x N array A and a graph G, can the elements of A be mapped onto the vertices of G such that the distance between the vertices of any two adjacent elements of A is at most T? In other words, does there exist a mapping that preserves the proximity of the elements of A?

2. A Lower Bound

In this section we prove a lower bound of approximately $mn/2\sqrt{s}$ on the cost of storing a matrix. We begin by apportioning the cost of a matrix.

Definition: The contribution (to the cost) of page i is the number of rows in which an element of page i occurs plus the number of columns in which an element of page i occurs. Let element (i,j) be on page k. Then the contribution of element (i,j) is the contribution of page k divided by the number of elements on page k.

Clearly, if there are k pages

$$\text{COST} = \sum_{i=1}^{k} \text{contribution of page } i \tag{2.1}$$

$$= \sum_{i=1}^{m} \sum_{j=1}^{n} \text{contribution of element } (i,j) \tag{2.2}$$

We proceed by bounding the contribution of a page, then of an element. This will then bound the cost.

An m x n matrix can be viewed as an m x n chess board, where each element occupies one cell. In considering the cost of a page, we can assume (without loss of generality) that the elements of the page can be contained in a rectangle where each row and column of the rectangle contains at least one element of the page. This is because the row and columns of the matrix can be permuted without changing the cost of the matrix. If the rectangle has dimensions a x b, its cost is simply a + b, or, in other words, its semi-perimeter. The number of elements in the page must be less than or equal to the rectangle's area.

Thus, the problem of determining a lower bound on the contribution of a page is the same as determining the rectangle with minimum semi-perimeter having a given area (or greater). For k^2 elements the answer is clearly a $k \times k$ square (with semi-perimeter $2k$). If we have k^2+1 elements, they no longer fit into the square, and one of the edges of the rectangle must be pushed out a unit to accommodate the additional element. Clearly, the best we can do with k^2+1 elements is to use a $(k+1) \times k$ rectangle (with semi-perimeter $2k+1$, an increase of exactly one unit over the square). This is sufficient for up to k^2+k elements. At this point we again have an "exact fit". For k^2+k+1 elements, we must expand to a $(k+1) \times (k+1)$ square which can be used for up to $(k+1)^2$ elements, and so on. The function for the minimal contribution of a page with t elements is defined below.

Definition: Let $t = k^2+j$ with $1 \le j \le 2k+1$

$$\begin{array}{ll} \text{If } 1 \le j \le k, & g(t) = 2k + 1 \\ \text{If } k + 1 \le j \le 2k + 1, & g(t) = 2k + 2 \end{array}$$

Values where we have an "exact fit" are especially important, motivating the following definition:

Definition: x is a <u>square number</u> iff it is of form k^2 for some $k \ge 1$. x is a <u>rectangular number</u> iff it is of form $k^2 + k$ for some $k \ge 1$. (This notation is somewhat unfortunate; the sets of square and rectangle numbers <u>are</u> disjoint.)

Notation: <u>p</u> will always denote the largest square or rectangle number less than or equal to s.

Lemma 2.1: Let page i have t elements, then the contribution of page i is at least $g(t)$ and at most $2t$.

Proof: The upper bound is obvious since t elements can occur in at most t rows and t columns. To prove the lower bound, suppose elements of page i occur in a rows and b columns. The contribution of this page is then $a + b$. Also, since each element occurs at the intersection of one of these a rows and one of these b columns, there can be at most ab elements and hence $t \leq ab$. We obtain a lower bound on the contribution of this page by solving the following problem:

Minimize $a + b$
Subject to $ab \geq t$ and a,b integer

Let $t = k^2+j$ where $1 \leq j \leq 2k+1$. We consider two cases:

<u>Case 1</u> ($j \leq k$): $a=k$ and $b=k+1$ is a feasible solution (i.e., $ab \geq t$) with cost $2k+1$. Suppose there is a solution with lower cost. Then $a+b \leq 2k$. However $t-ab \geq k^2+j-a(2k-a)=(k-a)^2 + j > 0$. Hence $t > ab$, the solution is infeasible, and the optimal cost is $2k+1=g(t)$.

<u>Case 2</u>. ($j>k$): $a=b=k+1$ is a feasible solution with cost $2k+2$. Suppose there is another solution with lower cost. Then $a+b \leq 2k+1$. However since $t-ab \geq k^2+k+(j-k)-a(2k+1-a) = (k-a)(k-a+1) + (j-k) > 0$ since $j>k$ and the product is non-negative for all integer k. Hence $t>ab$, the solution is infeasible and the optimal cost is $2k+2=g(t)$. [ ]

Lemma 2.2: The contribution of an element is at least $\min(g(p)/p, g(s)/s)$ and at most 2, where p is the largest square or rectangle number less than or equal to s.

Proof: The upper bound is obvious from Lemma 2.1. Consider the lower bound. If an element occurs on a page with t total elements its contribution is at least $g(t)/t$ by Lemma 2.1. Hence, a lower bound of the contribution of any element is $\min_{1\le t\le s} g(t)/t$. We now show that this quantity equals $\min(g(p)/p,\ g(s)/s)$ which will prove the lemma.

Consider any j. If j is neither a square number nor a rectangle number nor s, then $g(j+1) = g(j)$ and $g(j+1)/(j+1) < g(j)/j$. Hence j does not minimize $g(j)/j$. Suppose j is a square or rectangle number, but not the largest one less than or equal to s. Let r be the next largest square or rectangle number. We consider two cases:

Case 1. (j is a square number, say k^2, and r is k^2+k): Then

$$g(j)/j - g(r)/r = 2k/k^2 - (2k+1)/(k^2+k) = 1/(k^2+k) > 0$$

Case 2. (j is a rectangle number, say k^2+k, and r is $(k+1)^2$): Then

$$g(j)/j - g(r)/r = (2k+1)/(k^2+k) - (2k+2)/(k+1)^2 = 1/(k^2+k) > 0$$

In any case, $g(j)/j > g(r)/r$. Therefore, the only possibilities for the minimum are s and p. []

This lemma has the following interpretation. We consider the page containing the element in question and try to minimize the ratio of the semi-perimeter to the number of elements enclosed. Clearly, if the page does not have an "exact fit" (and is less than s), more elements can be added to decrease the ratio. Hence we need only consider square and rectangle numbers and s. To minimize the ratio, we make the area as large as possible, so we use s or the largest square or rectangle less than or equal to s. We have to consider s in case it is close enough to the next higher square or rectangle number so that the advantage of having a larger rectangle exceeds the disadvantage of not having an "exact fit".

Theorem 2.1: The cost $m \times n$ matrix for page size s is at least $\min(g(p)/p,\ g(s)/s) * mn$ where p is the largest square or rectangle number less than or equal to s.

Proof: Since the cost of a matrix is the sum of the contributions of its elements the result follows immediately from Lemma 2.2. [ ]

The next lemma gives a simple test to determine which of $g(p)/p$ and $g(s)/s$ is smaller.

Lemma 2.3: Let $s = k^2+j$, $1\leq j\leq 2k + 1$, let p be the largest square or rectangle number less than or equal to s and suppose $s \neq p$. Then

(a) If p is a square number then $g(p)/p \leq g(s)/s$ iff $j\leq k/2$

(b) If p is a rectangle number then $g(p)/p \leq g(s)/s$ iff $j \leq k(3k+2)/(2k+1)$.

Proof:

<u>Case 1</u>. (p is a square number, say k^2, and $s = k^2 + j$; $1 \leq j \leq k - 1$)

Then $g(p)/p - g(s)/s = 2k/k^2 - (2k+1)/(k^2+j) = (2j-k)/k(k^2+j)$

<u>Case 2</u>. (p is a rectangle number, say k^2+k, and $s = k^2 + j$; $k + 1 \leq j \leq 2k$): Then

$$g(p)/p - g(s)/s = (2k+1)/(k^2+k) - (2k+2)/(k^2+j)$$
$$= [(2k+1)j - k(3k+2)]/[(k^2+k) * (k^2+j)]$$

In either case, the theorem is obviously true. []

Remark: Note that $g(p)/p \leq g(s)/s$ for about half the values of s.

Finally, we show that to prove a storage algorithm is optimal, it suffices to simply count the number of elements whose contribution is non-optimal.

Definition: For a given algorithm for storing a matrix, let <u>NO(m,n)</u> be the number of elements of an m x n matrix whose contribution is <u>non-optimal</u> (i.e., greater than min(g(p)/p, g(s)/s)). Such an element is called a <u>non-optimal element</u>.

Lemma 2.4: An algorithm is optimal if

$$\lim_{m,n \to \infty} \frac{NO(m,n)}{mn} = 0 \text{ for all } s.$$

Proof:

Consider any s, then

$$\lim_{m,n \to \infty} \frac{COST(m,n)}{COST^*(m,n)} - 1$$

$$\leq \lim_{m,n \to \infty} \frac{COST^*(m,n)+2NO(m,n)}{COST^*(m,n)} - 1$$

where to obtain COST(m,n), we charge each element the optimal contribution and charge the non-optimal elements 2 additional units (since their contribution is at most 2 by Lemma 2.2). By Theorem 2.1, $COST^*(m,n) \geq cmn$, where c is a function of s which does not depend on m or n. Therefore this limit

$$\leq \lim_{m,n \to \infty} \frac{2NO(m,n)}{cmn} = 0$$

Hence $\displaystyle \lim_{m,n \to \infty} \frac{COST(m,n)}{COST^*(m,n)} = 1.$ []

3. An Asymptotically Optimal Algorithm for $g(p)/p \leq g(s)/s$

In this section, we give an algorithm that is asymptotically optimal if $g(p)/p \leq g(s)/s$, where p is the largest square or rectangle number less than or equal to s. This algorithm causes a fraction (at most $1/\lfloor\sqrt{s}\rfloor$) of the disk space to be wasted.

The strategy we must employ is clear from the proof of Theorem 2.1; we must have the contribution of nearly all elements be $g(p)/p$. This requires partitioning into rectangles of dimension a x b where a and b are defined as follows:

Notation: Let $s = k^2 + j$ where $1 \leq j \leq 2k+1$ and let p be the largest square or rectangle number less than or equal to s. If $p=k^2$, let $\underline{a}=\underline{b}=k$. If $p=k^2+k$, let $\underline{a}=k$ and $\underline{b}=k+1$. (Note $ab=p$ and $a+b=g(p)$.) Finally, let $y=m \bmod a$ and $z = n \bmod b$.

Algorithm A:

Step 1: Partition the submatrix consisting of the first $\lfloor m/a \rfloor * a$ rows and $\lfloor n/b \rfloor * b$ columns into blocks of dimension axb. (Marked "A" in Figure 3.1.) Put each block on a different page on disk.

Step 2: If $y \neq 0$, partition the elements in the last y rows into blocks (marked B) of dimension $y \times \lfloor s/y \rfloor$. Put all the leftover elements (if any) in a separate block (marked C). (If y=0 step 2 does nothing.)

Step 3: Partition the elements in the last z columns (except those in the last y rows) into blocks (marked D) of dimension $\lfloor s/z \rfloor \times z$. (If z=0, step 3 does nothing.) Again, put all the leftover elements into a separate block (marked E).

Theorem 3.1: Algorithm A is optimal if $g(p)/p \leq g(s)/s$.

Proof: The contribution of each element in a type A block is $(a+b)/ab = g(p)/p$. Thus, the only non-optimal elements are those not in blocks of type A. There are at most $mz + ny < ma + nb$ such elements. Since a and b are constants for a given s, applying Lemma 2.4 proves the theorem. [ ]

Theorem 3.2: Asymptotically the space wasted by Algorithm A is $(s-ab)/ab$ which is at most $1/\lfloor\sqrt{s}\rfloor$.

Proof: There are mn/ab type A blocks, each with waste $s-ab$. We can ignore the non-optimal elements because the fraction of non-optimal elements goes to 0. Finally, since $s \leq (a+1)b$, $(s-ab)/ab \leq 1/a = 1/\lfloor\sqrt{s}\rfloor$. [ ]

Note that the size of $(s-ab)/ab$ depends on how close s is to the next larger square or rectangle number. In fact, if s is a square or rectangle number, the waste is 0. In any case, the waste is no more than $1/\lfloor\sqrt{s}\rfloor$ which is 4 or 5% for reasonable s.

Finally, a desirable property of Algorithm A is that it is very easy to compute which blocks need to be retrieved to obtain a given row and then assign the proper elements from each block to the appropriate positions in an array.

4. An Asymptotically Optimal Algorithm for $g(s)/s \leq g(p)/p$

We now discuss an algorithm that is asymptotically optimal if $g(s)/s \leq g(p)/p$ where p is the largest square or rectangle number less than or equal to s. Asymptotically the fraction of space wasted is zero.

The strategy of the last section will not give an optimal result here. Since $g(s)/s \leq g(p)/p$, the contribution of nearly every element must be $g(s)/s$. This requires partitioning into rectangles of dimension a x b where a and b are defined as follows:

Notation: Let $s = k^2 + j$ where $1 \leq j \leq 2k+1$. If $j \leq k$ let $\underline{a}=k$ and $\underline{b}=k+1$. Otherwise let $\underline{a}=\underline{b}=k+1$. (Note: a and b are defined differently in Section 3). We have $ab \geq s$ and $g(s)=a+b$. Finally, let $\underline{y} = m \bmod a$ and $\underline{z} = n \bmod b$.

Clearly, if s is not a square or rectangle number, there will be elements inside of each rectangle that cannot be put in that page. In order for the algorithm to be asymptotically optimal, these leftover elements must also be assigned to pages in an efficient manner. The algorithm does this by grouping all the leftover elements together and recursively applying itself to the submatrix formed of these elements.

Algorithm B:

Step 1: Partition the submatrix consisting of the first $\lfloor m/a \rfloor$ a rows and $\lfloor n/b \rfloor b$ columns (again, a and b are different from those in Section 3) into blocks (marked "A" in Figure 4.1) of dimension a x b. If $ab \neq s$

there will be leftover elements. Further partition each block so that the last s-ab elements in the last column are in a separate block (marked "B"). These blocks are grouped together to form a submatrix, and Algorithm B is recursively called to partition it. (The algorithm does not require that the submatrix it is partitioning be contiguous, only that the coordinates of the elements of submatrix can be described as a Cartesian product of indices. This is because permuting the rows and columns of a matrix will not change its cost, and hence, any such submatrix can be permuted into a contiguous submatrix.)

Step 2: If $y \neq 0$ partition the last y rows as in Algorithm A (into blocks marked "C" and "D").

Step 3: If $z \neq 0$ partition the last z rows as in Algorithm A (into blocks marked "E" and "F").

Figure 4.2 gives a complete example of a matrix stored by Algorithm B.

Theorem 4.1: Algorithm B is asymptotically optimal if $g(s)/s \leq g(p)/p$.

Proof: We obviously have

$NO(0,0) = 0$

$NO(m,n) = mz + ny + NO((ab-s)\lfloor m/a \rfloor, \lfloor n/b \rfloor)$

(which is hard to handle). Define a simpler recurrence $NO'(m,n)$ by

$NO'(0,0) = 0$

$NO'(m,n) = (m+n)b + NO'(m, \lfloor n/b \rfloor)$

Clearly, NO' is monotonically non-decreasing with respect to m and n. We now prove $NO(m,n) \leq NO'(m,n)$ by induction on m and n. Clearly $NO(m,n) \leq NO'(m,n)$ if m=0 or n=0. Consider any m,n>0. Since

$$\begin{aligned} NO(m,n) &= mz + ny + NO((ab-s)\lfloor m/a \rfloor, \lfloor n/b \rfloor) \\ &\leq (m+n)b + NO'((ab-s)\lfloor m/a \rfloor, \lfloor n/b \rfloor) \text{ by induction} \\ &\leq (m+n)b + NO'(m, \lfloor n/b \rfloor) \text{ by the monotonicity of } NO' \\ &= NO'(m,n) \end{aligned}$$

The recurrence for NO´ is easy to solve. By repeated substitution we have (we get an upper bound by ignoring the floor signs)

$$NO'(m,n) \leq (m+n)b + (m+n/b)b + \ldots + (m+n/b^k)b \text{ where } k = \log_b n$$

$$= mb \log_b n + nb \sum_{i=0}^{k} 1/b^i$$

$$\leq mb \log_b n + nb * b/(b-1)$$

Hence $\lim_{m,n \to \infty} NO(m,n)/mn \leq \lim_{m,n \to \infty} NO'(m,n)/mn$

$$\leq \lim_{m,n \to \infty} [mb \log_b n + nb^2/(b-1)]/mn = 0$$

and Lemma 2.4 is applied to prove the theorem. [ ]

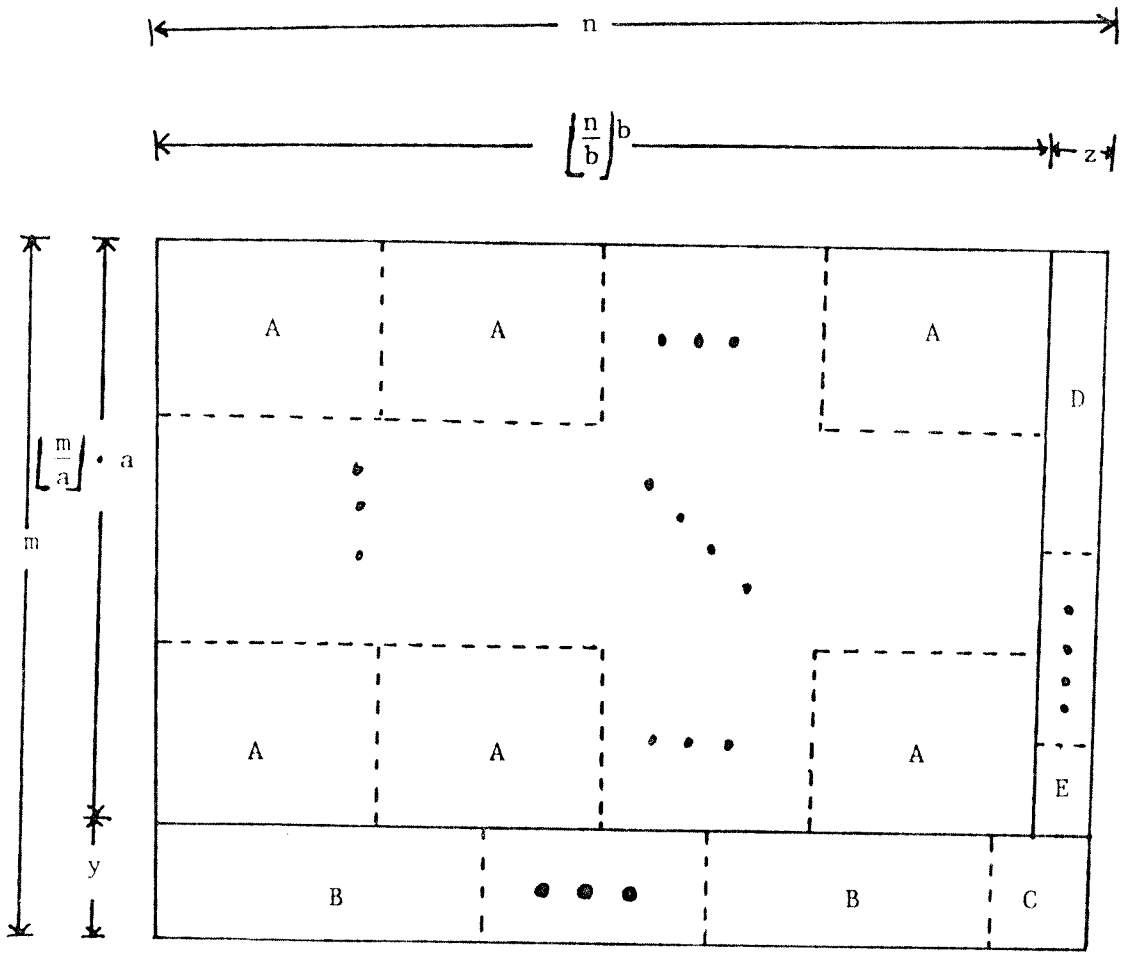
Theorem 4.2: The asymptotic fraction of space wasted by Algorithm B is zero.

Proof: Every optimal element is stored on a full page and the fraction of non-optimal elements goes to zero. [ ]

Finally, we note that an algorithm to retrieve rows and columns under this scheme is not prohibitively complicated.

5. CONCLUSION

We have shown that min(g(p)/p, g(s)/s) * mn is a lower bound for storing an m x n matrix where s is the page size and p is the largest square or rectangle number less than or equal to s. To prove the bound is tight, we exhibited two algorithms: one which is optimal if $g(p)/p \leq g(s)/s$ and the other if $g(s)/s \leq g(p)/p$. We also analyzed the space wasted by these two algorithms. One wasted almost no space and the other only a small fraction. Further, neither scheme is so complex that it cannot be implemented efficiently.



n
$\left\lfloor \frac{n}{b} \right\rfloor b$
z
m
$\left\lfloor \frac{m}{a} \right\rfloor \cdot a$
y
A
A
A
A
A
A
B
B
C
D
E

Figure 3.1

How Algorithm A partitions a matrix

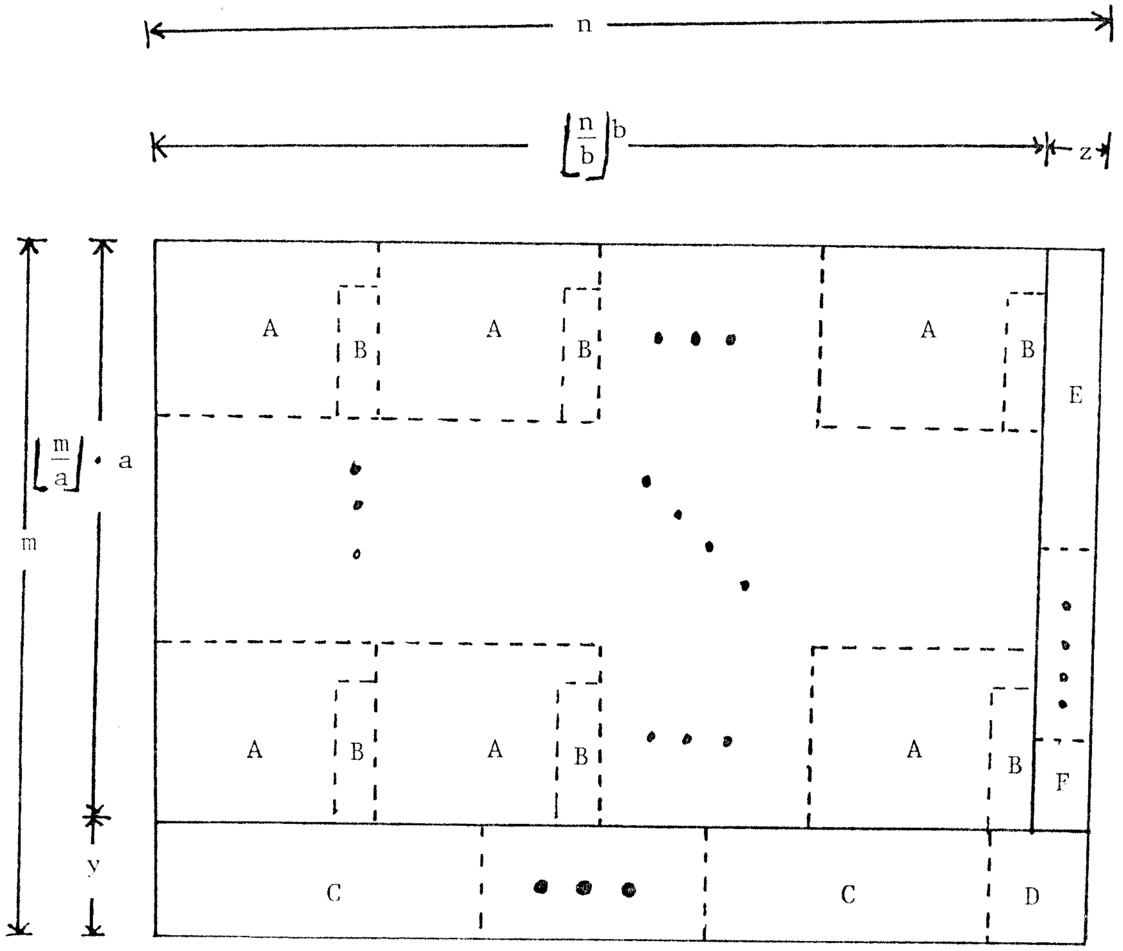

n
$\left\lfloor \frac{n}{b} \right\rfloor b$
z
$\left\lfloor \frac{m}{a} \right\rfloor \cdot a$
m
y
A
B
C
D
E
F

Figure 4.1

How Algorithm B partitions a matrix

(a and b are defined differently in Figure 3.1)

0	0	0	1	1	1	2	2	2	15	15
0	0	19	1	1	19	2	2	19	15	15
3	3	3	4	4	4	5	5	5	16	16
3	3	19	4	4	19	5	5	21	16	16
6	6	6	7	7	7	8	8	8	17	17
6	6	20	7	7	20	8	8	20	17	17
9	9	9	10	10	10	11	11	11	18	18
9	9	20	10	10	20	11	11	21	18	18
12	12	12	12	12	13	13	13	13	13	14

Figure 4.2

A 9 × 11 matrix stored using Algorithm B.

Assuming s = 5 (so a = 2, b = 3).

REFERENCES

[1] A. C. McKellar and E. G. Coffman, Organizing Matrices and Matrix Operations for Paged Memory Systems, CACM 12 (1969) 153-165.

[2] C. B. Moler, Matrix Computation with FORTRAN and Paging, CACM 15 (1972) 268-270.

[3] P. C. Fischer and R. L. Probert, A Note on Matrix Multiplication in a Paging Environment, Proc. ACM National Conf., 1976, 17-21.

[4] V. Strassen, Gaussian Elimination is Not Optimal, Numerische Mathematik 13 (1969) 354-356.

[5] A. L. Rosenberg, Preserving Proximity in Arrays, SIAM J. Comput. 4 (1975) 443-460.

[6] R. A. DeMillo, S. C. Eisenstat and R. E. Lipton, Preserving Average Proximity in Arrays, CACM 21 (1978) 228-231.

[7] D. Bollman, On Preserving Proximity in Extended Arrays, SIAM J. Comput 5, 318-323.