Research Report 303

Matricial Space-Economy with Constant Access-Time

Ida Pu, Alan Gibbons

RR303

We describe a particularly simple and practical algorithm for economic storage of arrays without recourse to the intricacies of perfect hash functions. This is done while retaining constant access time. The algorithm performs usefully over a range of values of x/n where the notional input array contains n entries with x non-zero elements. For x less than or equal to n/k and one particular setting of the array parameter, we show that the average storage required is about $n\,(k+30)/10k$. For sparse arrays this becomes $n/10$.

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

March 1996

Matricial Space-Economy with Constant Access-Time

(DRAFT)

Ida Pu and Alan Gibbons *
Department of Computer Science, University of Warwick, UK

Abstract

We describe a particularly simple and practical algorithm for economic storage of arrays without recourse to the intricacies of perfect hash functions. This is done while retaining constant access time. The algorithm performs usefully over a range of values of x/n where the notional input array contains n entries with x non-zero elements. For $x \leq n/k$ and one particular setting of the array parameter, we show that the average storage required is about $n(k+30)/10k$, where n is the number of locations of the notional input array. For sparse arrays this drops to about $n/10$.

1 Introduction

In many programming applications involving sparse matrices, there is the potential for space economy in matricial storage. For example, we could store just the non-empty (or non-zero) elements along with their original matricial indices. It would then be a trivial matter to find (that is, *access*) any matricial element, given its indices, in logarithmic sequential time using standard binary search. The penalty to be paid for the space-economy of such an algorithm is the loss of constant access-time. In this paper we describe a particularly simple and practical algorithm to achieve space-economy and at the same time retain constant access-time. Moreover, our deterministic sequential algorithm does not have recourse to the intricacies of perfect hash functions [1]. Apart from [1], we have been unable to find any reference to the problem addressed by this paper. The algorithm treats the input as though it was a one-dimensional array. Trivial changes would readily yield an algorithm for matrices of arbitrary dimensions.
In section 2 we describe our simple algorithm. In section 3 we obtain an exact expression for the average storage requirements of the algorithm and we quantify this for a particular value of an algorithmic parameter. The final section is a summary of our results and conclusions.

*fax: +44 1203 525714, e-mail: amg@dcs.warwick.ac.uk. Partially supported by the EC ALCOM-IT programme contract number 20244

2 The algorithm

Imagine that n input items occupy the locations of a notional array INPUT[1 .. n] and that there are x non-zero elements. Our algorithm constructs a compressed array, COMP, and an auxiliary array, AUX[1..t, 1..2]. The arrays COMP and AUX occupy available storage space whereas the elements of INPUT are merely sequentially scanned by the input device. The elements of COMP include the non-zero elements of INPUT and the elements of AUX allow any element of INPUT to be deduced in constant time. The array INPUT is imagined to be divided from the left into equal sized segments of length s with (if s is not a divisor of n) a final smaller segment of length $n - s\lfloor n/s \rfloor$. We define the t in terms of s, $t = \lceil n/s \rceil$. In general any such segment of INPUT will contain a number of leading and trailing zeros (or *empty* elements). We delete these and construct COMP by concatenating the truncated segments. Figure 1 shows an example for $n = 21$ and $s = 4$. Here the array COMP has length 6.

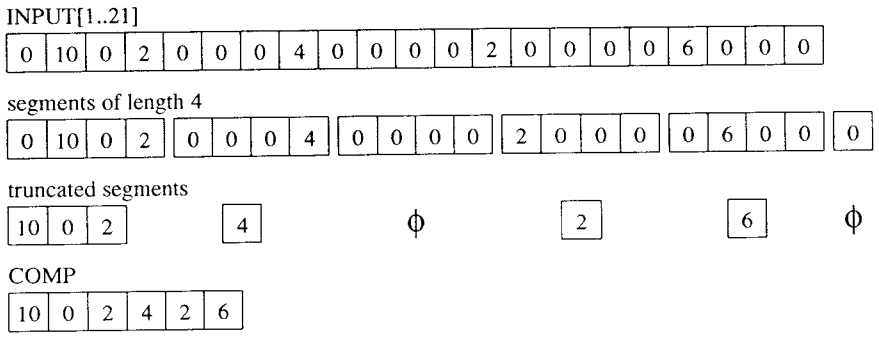

Figure 1: The construction of COMP

In general we might expect the length of COMP to be much less than the length of INPUT especially, for example, when t is close to x. In this case many untruncated segments will, on average, contain only empty elements or exactly one significant element. In both cases segment truncation does not force the storage of non-significant elements of INPUT in COMP.

The potential for space-economy to be had from storing COMP rather than INPUT can be achieved while retaining constant access-time as we now see. Given j we are required to deduce INPUT[j] from COMP by making use of the array AUX which we now define. For $1 \leq i \leq t$, AUX[i, 1] is the address of the first element of the ith truncated segment in the array COMP (if the truncated segment is empty, then it is the address of where such an element *would* have appeared) and AUX[i, 2] is the number of leading zeros in the ith untruncated segment. If the segment contains only zeros, then they are counted as leading zeros. Figure 2 shows AUX for the example of figure 1.

The algorithm FindINPUT[j] described below returns the value of the jth element in the imagined array INPUT given the arrays COMP and AUX.

In Algorithm FindINPUT[j], h is the number of the segment from the left containing INPUT[j], and d is the address of INPUT[j] in the uncompressed array *relative to*

1	4	5	5	6	7
1	3	4	0	1	1

Figure 2: The matrix AUX for our example

beginning of the segment containing INPUT[j]. Note AUX[$h+1,1$] – AUX[$h,1$] is the length of the truncated segment and the length of the AUX array is $m+1$ in this implementation, where m is the number of segments.
For each segment of length s, d, the offset of the the distance from INPUT[j] to the beginning of the the uncompressed segment containing INPUT[j], can only be in three areas, i.e. $0 \leq d \leq nhz$, where nhz is *the number of the leading zeros*, or $nhz < d < nhz+nt$ where nt is the *the length of the truncated segment* or $nhz + nt \leq d \leq s$. The number of leading zeros in the segment is stored in AUX[$h,2$] and the length of the truncated segment is actually the difference between the address of COMP for the first non-zero element of the segment and of the following segment, i.e. AUX[$h+1,1$] – AUX[$h,1$]. All this is illustrated in figure 3.

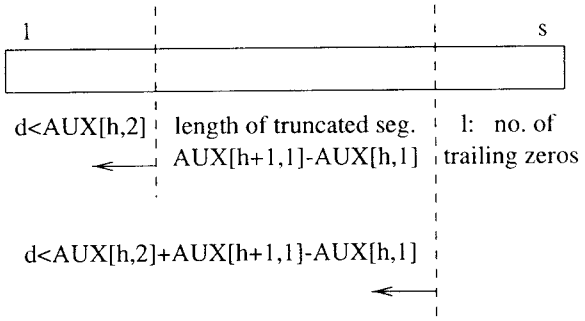

Figure 3:

The algorithm may be implemented in different ways. We assume a block structured style in which arrays may be dynamically declared. The implementation would be more elegantly expressed in a language which allowed for dynamic extension of arrays. Within our self-imposed style, we need to read in the data twice, although this can be avoided by an alternative style.
Algorithm Construct reads in the data sequence and returns the compressed array COMP and auxiliary array AUX, where AUX[$1..m+1,1$] contains the addresses in COMP of the first non-zero element of each segment and AUX[$1..m,2$] stores the number of leading zeros in each segment. AUX[$m+1,1$] stores the next address to the last address of COMP.
Line 2 inputs the length of the input sequences and the length of the segments Line 3 computes the number of segments. Line 4 dynamically declares the auxiliary array AUX.
The algorithm Construct deals with the data sequence by segments. In each segment (lines 6 – 21), it stores the address of the first non-zero element in a segment

in AUX[1..$m+1,1$] (Line 8), In line $9-21$ k counts the leading zeros and l counts the trailing zeros. The length of the truncated segment is q (Line 19) and p is the address in COMP for the first non-zero element of the next segment (Line 20).
Line 25 dynamically declares the array COMP. Lines $26-37$ read the data at the second time. This enters the truncated segments into COMP. That programme body in which the input is to be manipulated is imagined to be contained between Lines $37-38$. This is within the block in which the required arrays have scope.

Algorithm FindINPUT[j]
Input: j
Output: INPUT[j]
1 **begin**
2 $h = \lceil \frac{j}{s} \rceil$, $d = (j-1)$ **mod** s
3 **if** $d <$ AUX$(h,2)$ **or** $d \geq$ AUX$(h,2)$ + AUX$(h+1,1)$ − AUX$(h,1)$
4 **then** INPUT$[j] = 0$
5 **else** INPUT$[j]$ = COMP[AUX$(h,1) + d -$ AUX$(h,2)$]
6 **end**

Algorithm Construct
Input: input data sequence
Output: array COMP and AUX
1 **begin**
2 **read** n, s
3 $m = \lceil \frac{n}{s} \rceil$
4 **array** AUX[1..$(m+1)$, 1..2]
5 $p \leftarrow 1$
6 **for** $i = 1$ **to** m **do**
7 **begin**
8 AUX$[i,1] \leftarrow p$, $k \leftarrow l \leftarrow 0$
9 **for** $j = 1$ **to** s **do**
10 **begin**
11 *isHead0* ← **true**
12 **read** input($i.e.$INPUT[$(i-1)s+j$])
13 **if** (input = 0 **and** *isHead0*) **then** $k \leftarrow k+1$
14 **else** *isHead0* ← false
15 **if** (input = 0 **and** ¬*isHead0*) **then** $l = l+1$
16 **else** $l \leftarrow 0$
17 **end**
18 AUX$[i,2] \leftarrow k$
19 $q \leftarrow s-(k+l)$
20 $p \leftarrow p+q$
21 **end**
22 AUX$[m+1,1] \leftarrow p$
23 **end**
24 **begin**
25 **array** COMP[1..$p-1$]
26 $p \leftarrow 1$, **restart read**
27 **for** $i = 1$ **to** m **do**

```
28    begin
29    for j = 1 to AUX[i, 2] do read input
30    for j = 1 to AUX[i + 1, 1] − AUX[i, 1] do
31        begin
32        read input
33        COMP[p] ← input
34        p ← p + 1
35        end
36    for j = 1 to s − AUX[i, 2] − AUX[i + 1, 1] + AUX[i, 1] do read input
37    end
38 ...
   Programme body employing calls FindINPUT[j]
   ...
39 end
```

3 Expected Space Economy

In section 3.1 we obtain an exact algebraic expression for the expected length of the array COMP. Then in section 3.2, for a particular choice of the variable s, we derive a simple expression for this average which nevertheless provides a close approximation. This is confirmed by a comparison between plots of the exact and approximate expressions. The particular choice of $s = n/x$ is shown to provide an algorithm with good space economy for sparse matrices.

3.1 The average length of the array COMP

We prove three Lemmas which ultimately provide an exact expression for the average length of the array COMP. Given a one-dimensional array segment of length s containing k non-zero elements, by $\mathrm{A}(s,k)$ we denote the *average* length of the reduced segment obtained by removing those segment locations containing leading and trailing zeros. The first Lemma derives an expression for $\mathrm{A}(s,k)$.

Lemma 1 *For* $1 \leq k \leq s$,

$$A(s,k) = s\left(\frac{k-1}{k+1}\right) + \left(\frac{2k}{k+1}\right)$$

Proof Clearly $\mathrm{A}(s,k)\binom{s}{k} = Sum(s,k)$ where $Sum(s,k)$ is the sum of the lengths of reduced segments of all possible distributions of the non-zero elements over the unreduced segment. Now

$$Sum(s,k) = \sum_{i=0}^{s-k}(k+i)(s-(k+i)+1)\binom{k+i-2}{k-2} \tag{1}$$

because a reduced segment of length $(k+i)$ can reside at $(s-(k+i)+1)$ positions in the segment and in any one position there may be $\binom{k+i-2}{k-2}$ segments of this length. We simplify this expression by first taking the sum variable i completely into the binomial coefficients as follows:

$$(k+i)(s-(k+i)+1)\binom{k+i-2}{k-2} = s(k+i)\binom{k+i-2}{k-2} - (k+i)(k+i-1)\binom{k+i-2}{k-2} \quad (2)$$

but $$(k+i)(k+i-1)\binom{k+i-2}{k-2} = k(k-1)\binom{k+i}{k} \quad (3)$$

and $$(k+i)\binom{k+i-2}{k-2} = (k+i-1)\binom{k+i-2}{k-2} + \binom{k+i-2}{k-2} = (k-1)\binom{k+i-1}{k-1} + \binom{k+i-2}{k-2} \quad (4)$$

Combining 2, 3 and 4 with 1:

$$Sum(s,k) = s(k-1)\sum_{i=0}^{s-k}\binom{k+i-1}{k-1} + s\sum_{i=0}^{s-k}\binom{k+i-2}{k-2} - k(k-1)\sum_{i=0}^{s-k}\binom{k+i}{k} \quad (5)$$

We use the standard results:

$$\sum_{r=0}^{y}\binom{r}{l} = \binom{y+1}{l+1} \text{ and } \binom{b}{a} = 0, \text{if } b < a \quad (6)$$

To simplify 5, we note using 6:

$$\sum_{i=0}^{s-k}\binom{k+i-1}{k-1} = \sum_{j=k-1}^{s-1}\binom{j}{k-1} = \sum_{j=0}^{s-1}\binom{j}{k-1} = \binom{s}{k} \quad (7)$$

Similarly: $$\sum_{i=0}^{s-k}\binom{k+i-2}{k-2} = \binom{s-1}{k-1} \text{ and } \sum_{i=0}^{s-k}\binom{k+i}{k} = \binom{s+1}{k+1} \quad (8)$$

From 7, 8 and 5:

$$Sum(s,k) = s(s-1)\binom{s}{k} + s\binom{s-1}{k-1} - k(k-1)\binom{s+1}{k+1} = \binom{s}{k}\frac{ks+2k-s}{k+1}$$

Thus $A(s,k) = s\left(\frac{k-1}{k+1}\right) + \left(\frac{2k}{k+1}\right)$ which is what was to be proved. Notice that the proof is only valid for $s \leq k \leq 2$ because equation 1 was written down on this basis. However, we notice that the formula gives $A(s,k) = 1$ which happens to be valid. If $k = 0$ then clearly $A(s,k) = 0$, whereas the formula gives $(-s)$. Compensation must therefore be made for this fact in later derivations of formulas. □

The formula of the previous Lemma has been numerically checked for a number of cases. For example, it is easy to check from first principles that $A(5,3) = 4$ which is what the formula also gives.

We now employ result of Lemma 1 to find an exact expression for $Ave(n,x,s)$ which denotes the *average* length that the array COMP would attain given that INPUT[1..n], x and s are as defined in the previous section.

Lemma 2 *If s is a divisor of n, then*

$$Ave(n,x,s) = \frac{n}{s}\left[\left((s+2) - 2\frac{n+1}{x+1}\right) + \alpha\left(s + 2\frac{n-s-x}{x+1}\right)\right]$$

where $\alpha = \prod_{i=0}^{x-1}(1 - \frac{s}{n-i}) = \binom{n-s}{x}/\binom{n}{x}$.

Proof Clearly $Ave(n,x,s)\binom{n}{x} = Sum(n,x,s)$ where $Sum(n,x,s)$ is the sum of the lengths over reduced segments over all possible distributions of the non-zero elements over the array INPUT[1..n]. We first show that

$$Sum(n,x,s) = \frac{n}{s}\sum_{k=1}^{\min(x,s)} \binom{s}{k}\binom{n-s}{x-k}A(s,k) \tag{9}$$

If s is a divisor of n then there are exactly n/s segments. We consider how the x non-zero elements are partitioned these segments. Consider the contribution to $Sum(n,x,s)$ from all distributions which place k_i non-zero elements in the ith segment, $1 \le i \le \frac{n}{s}$. Writing $m = \frac{n}{s}$, this contribution is given by:

$$\binom{s}{k_1}\binom{s}{k_2}\cdots\binom{s}{k_m}(A(s,k_1) + A(s,k_2) + \cdots + A(s,k_m)) \tag{10}$$

because, for each fixed internal arrangement of k_i non-zero elements in the ith segment all possible internal arrangements of k_j non-zero elements in all the jth segments ($i \neq j$) occur. We obtain equation 9 by summing expression 10 over all possible partitions and using the identity:

$$\sum_{\substack{b_1 + b_2 + \cdots + b_j = b \\ 0 \le b_i \le \min(x,s), 1 \le i \le j}} \binom{s}{b_1}\binom{s}{b_2}\cdots\binom{s}{b_j} = \binom{js}{b}$$

Now substituting for $A(s,k)$ from Lemma 1, and taking account of the fact (noted at the end of the proof of Lemma 1) that the expression for $A(s,0)$ needs to be compensated for, we obtain:

$$\begin{aligned} Sum(n,x,s)/m &= \sum_{k=0}^{\min(x,s)} \binom{s}{k}\binom{n-s}{x-k}\left((s+2) - 2\frac{s+1}{k+1}\right) + s\binom{n-s}{x} \\ &= (s+2)\binom{n}{x} - 2(s+1)\sum_{k=0}^{\min(x,s)} \binom{s}{k}\binom{n-s}{x-k}\frac{1}{k+1} + s\binom{n-s}{x} \end{aligned} \tag{11}$$

But

$$\binom{s}{k}\frac{1}{k+1} = \frac{1}{s+1}\binom{s+1}{k+1} \tag{12}$$

and

$$\sum_{k=0}^{\min(x,s)} \binom{s+1}{k+1}\binom{n-s}{x-k} = \sum_{j=1}^{\min(x,s)+1} \binom{s+1}{j}\binom{n-s}{(x+1)-j} - \binom{s+1}{0}\binom{n-s}{x+1} = \binom{n+1}{x+1} - \binom{n-s}{x+1} \quad (13)$$

So $$Sum(n,x,s)/m = (s+2)\binom{n}{x} + s\binom{n-s}{x} - 2(\binom{n+1}{x+1} - \binom{n-s}{x+1})$$

now

$$\binom{n+1}{x+1} = \frac{n+1}{x+1}\binom{n}{x} \quad \text{and} \quad \binom{n-s}{x} = \binom{n}{x}\prod_{i=0}^{x-1}(1-\frac{s}{n-i})$$

and so we finally obtain

$$Sum(n,x,s)/(m\binom{n}{x}) = ((s+2) - 2\frac{n+1}{x+1}) + (s + 2\frac{n-s-x}{x+1})\prod_{i=0}^{x-1}(1-\frac{s}{n-i})$$

from which the result follows immediately. □

The formula of Lemma 2 has been independently checked for a number of specific cases. Note that we obtain $Ave(n,1,s) = 1$, $Ave(n,x,2) = x$ and $Ave(n,2,n/2) = \frac{n+19}{12} + \frac{1}{4(n-1)}$ which are easily proved independently.

Space permitting, we could now have combined the two previous Lemmas to find an expression for $Ave(n,x,s)$ which is not subject to the restriction that s be a divisor of n. This is readily done by treating the array INPUT as consisting of two parts, the first of length $s\lfloor n/s \rfloor$ and the second of length $n - s\lfloor n/s \rfloor$. The first may then handled through Lemma 2 and second through Lemma 1 and combined to give the desired result. However, the result is sufficiently similar to that of Lemma 2 for us to be content with that in the space available. In any case Lemma 2 is sufficient for the algorithmic analysis provided in section 3.2. For completeness the following Lemma does provide a bound on $Ave(n,x,s)$ with no restrictions on the parameters.

Lemma 3 *For any x,s such that $0 \le x, s \le n$,*

$$Ave(n,x,s) \le \lceil \frac{n}{s} \rceil \left[\left((s+2) - 2\frac{s\lceil \frac{n}{s} \rceil + 1}{x+1} \right) + \alpha \left(s + 2\frac{s\lceil \frac{n}{s} \rceil - s - x}{x+1} \right) \right]$$

where $\alpha = \prod_{i=0}^{x-1}(1 - \frac{s}{s\lceil \frac{n}{s} \rceil - i}) = \binom{s\lceil \frac{n}{s} \rceil - s}{x} / \binom{s\lceil \frac{n}{s} \rceil}{x}$.

Proof We observe that:

$$Ave(n,x,s) \le Ave(s\lceil \frac{n}{s} \rceil, x, s)$$

If s is a divisor of n, then equation holds. Otherwise we note that for each arrangement of non-zero elements over INPUT[1..n] in which there are at least two such elements in the final segment, there are arrangements over INPUT[1..$s\lceil \frac{n}{s} \rceil$] for which all non-zero elements in the first $\lfloor \frac{n}{s} \rfloor$ segments are identically arranged but over the final segment may be arranged to produce larger reduced segments. Now $s\lceil \frac{n}{s} \rceil$ is divisible by s and so we can apply Lemma 2 to Ave($s\lceil \frac{n}{s} \rceil, x, s$) and the theorem follows. □

3.2 Quantifying the storage requirements

Here we show how setting $s = n/x$ provides an algorithm with effective space economy. For technical clarity we shall assume the formula of Lemma 2 although it should be clear that the arguments used here would obtain essentially similar results if s was not restricted to be a divisor of n.
Note that whatever the values of n and x, if $s \leq 2$ the algorithm will give a length of x for the array COMP which is optimal. This follows both from Lemma 2 and from the observation that no non-zero elements of the input will be written to COMP for such small values of s. On the average, the total storage required by the algorithm is, within a small additive constant, given by $Ave(n, x, s) + 2\frac{n}{s}$. The second term is what the array AUX needs and for $s = n/s$ is $2x$. Of course, x provides a natural lower bound for storage required for the problem considered in this paper.
Rewriting the formula of Lemma 2 and replacing s with n/x, we obtain:

$$Ave(n, x, s) = \frac{2xn}{x+1} - (1-\alpha)\left(\frac{3x-1}{x+1}n - \frac{2x^2}{x+1}\right) \tag{14}$$

We need an upper bound on α to get an upper bound for $Ave(n, x, s)$. Here $\alpha = \prod_{i=0}^{x-1}(1 - \frac{n/x}{n-i}) < (1 - \frac{1}{x})^x < e^{-1}$. If $x > 35$ then, within 1%, α is closely approximated by e^{-1}. For such values of x we further take $\frac{3x-1}{x+1} \approx 3$, $\frac{2x^2}{x+1} \approx 2x$ and $\frac{2x}{x+1} \approx 2$. It then follows that:

$$Ave(n, x, s) \approx 2n - (1 - e^{-1})(3n - 2x) = 0.1037n + 1.2642x \tag{15}$$

where we have taken $e = 2.71828$.
Taking $s = n/x$, the above analysis shows that we have an algorithm which uses (taking the requirements of $Ave(n, x, n/s)$ and AUX together)) about $\frac{n}{10} + 3x$ space. This is confirmed by Figures 4 and 5. Figure 4 plots $Ave(n, x, n/s)/n$ against x for various values of n using the *exact* expression for $Ave(n, x, n/s)$ given by Lemma 2. Figure 5 provides the same plots using the formula of equation 15 for $Ave(n, x, n/s)$. The similarity between these figures is remarkable, particularly for low values of x. This provides additional confidence for our claim that we have described, for sparse matrices (where $n \gg x$), an algorithm which on average uses about 10% of the space requirement of the input data.

4 Summary and concluding remarks

We have described a particularly simple and practical algorithm for economic storage of arrays without recourse to the intricacies of perfect hash functions. This was done at the express intention of retaining constant access time. The algorithm performs usefully over a range of values of x/n. For $x \leq n/k$, the average storage required is about $n(k + 30)/10k$, where n is the number of locations of the notional input array. For sparse arrays this drops to about $n/10$.

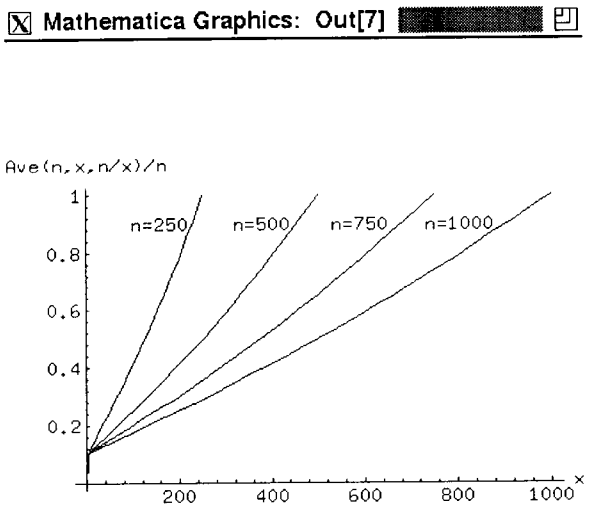

Figure 4: Plots obtained using the exact expression for Ave(n,x,n/x)

We obtained an exact expression for the average storage requirements of the algorithm and we quantified this for a particular value of the parameter s, namely when $s = n/x$. It is not at all clear that this particular choice provides the best performance for the algorithm although this choice is certainly useful. Further investigations are necessary in this regard. We would also like to know what the distribution of the length of COMP over x looks like. Whether or not it is sharply peaked around the average value is not known.

Although economic, the average storage space required by the algorithm is seemingly a fraction of n. It would be preferable if this was replaced by a constant multiple of x which provides a natural lower bound. It is unclear whether an algorithm as simple and as practical as ours can achieve this. It certainly seems unlikely within the approach taken in this paper. This is because there are distributions of the non-zero elements which force a dependence of the length of COMP on n even if additional strategies are allowed. One such strategy might employ the sliding of the imposed grid along the notional array INPUT so as to minimise the length of COMP for any particular input. This strategy cannot avoid the dependence of the length of COMP on n because, for example, the non-zero elements may be uniformly distributed at separating distances of $s/2$. For $s = n/x$, this guarantees that each unreduced segment occupied by two elements contributes $n/2x$ elements to COMP. Other choices of s might avoid this.

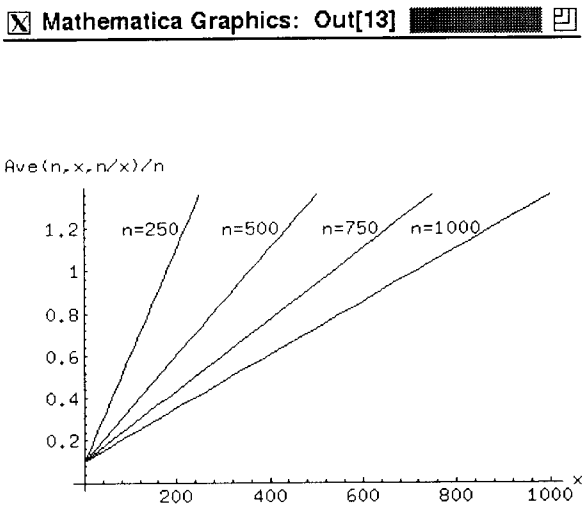

Figure 5: Plots obtained using the approximate expression for Ave(n,x,n/x)

5 Acknowledgements

We thank S. Muthukrishnan whose conversation (perhaps unwittingly) initiated the enquiries of this paper.

References

[1] Fredman and Komlos and Szemered, "Storing a Sparse Table with O(1) Worst Case Access Time", *Proc. 23th IEEE Symposium on Foundations of Computer Science (FOCS)*, 1983, 165-169.