Programming Techniques
S. L. Graham Editor

Storing a Sparse Table

Robert Endre Tarjan and
Andrew Chi-Chih Yao
Stanford University

The problem of storing and searching large sparse tables is ubiquitous in computer science. The standard technique for storing such tables is hashing, but hashing has poor worst-case performance. We propose a good worst-case method for storing a static table of n entries, each an integer between 0 and $N-1$. The method requires $O(n)$ words of storage and allows $O(\log_n N)$ access time. Although our method is a little complicated to use in practice, our analysis shows why a simpler algorithm used for compressing LR parsing tables works so well.

Key Words and Phrases: Gaussian elimination, parsing, searching, sparse matrix, table compression, table lookup

CR Categories: 3.74, 4.12, 4.34, 5.25

Professor Tarjan's research was supported in part by the National Science Foundation under grant MCS75-22870-A02, the Office of Naval Research under contract N00014-76-C-0688, and a Guggenheim fellowship; Professor Yao's research was supported in part by the National Science Foundation under grant MCS-77-05313.

Authors' address: Department of Computer Science, Stanford University, Stanford, CA 94305.

1. Introduction

The following table-searching problem arises in many areas of computer science. Given a universe of N names and an initially empty table, we wish to be able to perform two operations on the table:

enter(x): Add name x (and possibly some associated information) to the table.
lookup(x): Discover whether x is present in the table, and if it is, retrieve the information associated with it.

We shall consider the *static* case of this problem, in which all the entries occur before all the lookups. We shall use a random access machine with uniform cost measure [1] as the computing model. We assume that the names are integers in the range 0 through $N-1$ and that each storage cell in the machine can store an integer of magnitude $O(N)$.

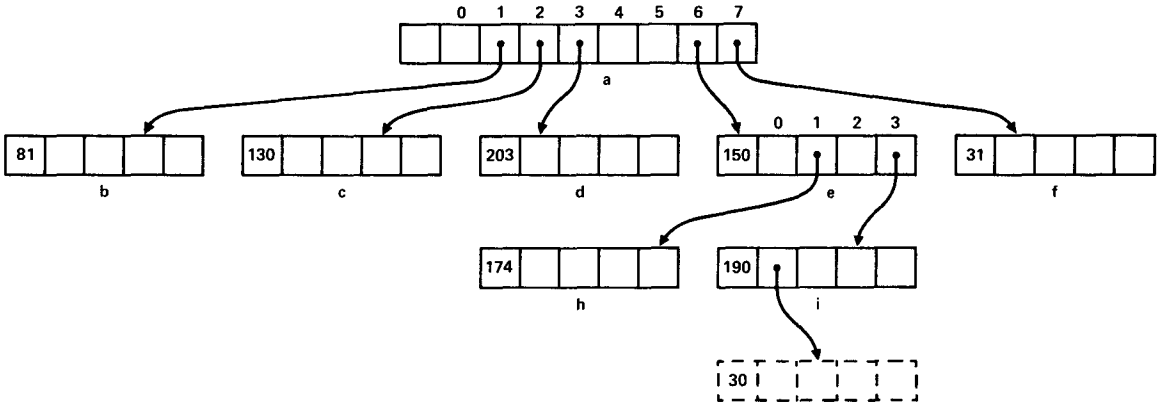
We are especially interested in tables that are sparse but not too sparse, by which we mean that N, the size of the universe of names, is bounded by a polynomial in n, the table size. Two applications which require storing and accessing such tables are LR parsing [2] and sparse Gaussian elimination [7]; in these applications the table is a sparse square matrix, and N is $O(n^2)$.

An optimum solution to the table-searching problem would be a method which requires $O(1)$ time per access and $O(n)$ words of storage. Note that the number of bits required to store a table of n entries selected from 0 through $N-1$ is $\log_2 \binom{N}{n} = \Omega(n \log_2 N)$ for $N = \Omega(n^{1+\epsilon})$. Since in our model each word can store at most $O(\log_2 N)$ bits, at least $\Omega(n)$ words are required to store the table. If we use an array of size N to store the table, each operation requires $O(1)$ time, but the storage is excessive if $n \ll N$. (Note that the solution to exercise 2.12 in Aho et al. [1] allows us to avoid initializing the array.) If we use a balanced binary tree [5] or similar structure to store the table, the storage is $O(n)$ but each operation requires $O(\log n)$ time. The best method in many practical situations is the use of a hash table [5], which requires $O(n)$ space to store the table and achieves an $O(1)$ time bound per operation on the average, though not in the worst case.

Although for most practical purposes hashing solves the table lookup problem, it is of interest to know how far the storage required for the table can be reduced while maintaining an $O(1)$ worst-case access time. We may be able to devise a method that is simpler to implement than hashing and thus better in some applications. Surprisingly little work, none of it theoretical, has been done on this problem; see Sprugnoli [6], for example.

In this paper we present a good worst-case method for storing sparse tables. Our method combines a trie structure [5] with a method for compressing tables by using double displacements. This double displacement

Fig. 1. A Trie with $m = 512$, Eight-Way Branching at the Root, Four-Way Branching Elsewhere, and Entries 3, 81, 130, 150, 174, 190, 203, 255. To insert 30, we divide 30 by 8 and repeatedly by 4, giving $30 \xrightarrow{6} 3 \xrightarrow{3} 0 \xrightarrow{0} 0$. The remainders 6 and 3 lead to node i, where we insert a new node as indicated. To delete 150, we locate it in node e, locate an external node which is a descendant of e, say h, replace 150 in e by 174, and delete node h.



method is an elaboration of a single displacement method suggested by Aho and Ullman [2] and Ziegler [8] for compressing parsing tables. Our method requires $O(n)$ words of storage and allows $O(\log_n N)$ access time. Although our method is a little complicated to use in practice, our analysis shows why the single displacement scheme, which is very practical, works so well.

The paper contains five sections. In Section 2 we review trie structures. In Section 3 we describe and analyze the double displacement method. In Section 4 we combine the results of Sections 2 and 3 to obtain our static table storage scheme. In Section 5 we discuss applications of our method and modifications to make it practical.

2. Tries for Table Storage

In this section we review a table storage method described by Knuth [5]. In addition to being a part of our algorithm, the method is interesting in its own right as a storage scheme for dynamic tables in which the total number of entries to be made is known ahead of time. Two applications of this method are to keep track of signatures when finding equivalent expressions [3] and to compute the fill-in positions for sparse Gaussian elimination [7].

To store a dynamic table, we use a trie with n-way branching at the root and k-way branching at every other node, where $k \geq 2$ is an integer whose value is selected in advance. Each node in the trie contains one table name and either n or k pointers to nodes one level deeper in the trie. (Some or all of the pointers may be null.) Figure 1 gives an example of such a data structure.

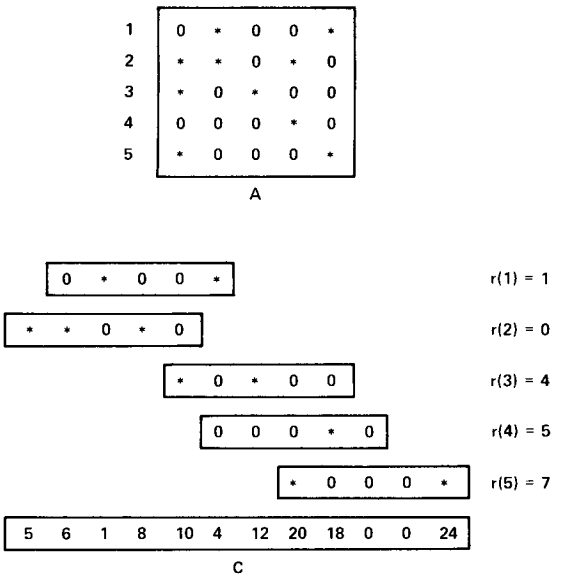
To look up a name x in the trie, we divide x by n and then repeatedly divide the resulting quotient by k. We use the successive remainders to specify a search path in the trie. For instance, to search for 190 in the trie of Figure 1, we look for 190 in the root. Not finding it, we divide 190 by 8, leaving 23 with remainder 6, which leads us to node e. Again not finding 190, we divide 23 by 4 to get 5 with remainder 3. This leads us to node i, where we find 190. To insert a name in the trie, we first search for it. The search leads to a null pointer, which we replace by a pointer to a new node containing the new name. See Figure 1.

Our tries differ from those discussed by Knuth only in that we allow the root to have a higher branching factor than the other nodes; this reduces the time required by the method without increasing the space bound, but requires that we know n (at least approximately) before we begin to construct the table. It is straightforward to implement the method, and we leave the details as an exercise. Note that by choosing the branching factors to be powers of 2, we can replace division by shifting, and we can allocate space for the pointers out of a single array, avoiding initialization by using the solution to exercise 2.12 in Aho et al. [1].

The total space required by the method is $O(kn)$ in the worst case. The time required for either a lookup or an insertion is proportional to the length of the search path, which is $\lceil \log_k(N/n) \rceil$ in the worst case. On the average, assuming that the table entries are uniformly and independently distributed among 0 through $N - 1$, the method requires $O(1)$ lookup and insertion time, since it is at least as fast (ignoring constant factors) as hashing with separate chaining [5]. The method exhibits an interesting time-space tradeoff. Assume $N = O(p(n))$ where p is a polynomial. If we choose k to be a constant, the storage is $O(n)$ and the worst-case access time is $O(\log n)$. If we choose $k = n^\epsilon$ for some constant ϵ, the storage is $O(n^{1+\epsilon})$ and the access time is $O(1)$.

If we add to each trie node a list of the non-null pointers in it, then the data structure will support deletions. To delete a given table entry, we first search for the node containing it, say p. We then locate some external node q which is a descendant of p. We replace the entry in p by the entry in q and delete node q. If p itself is an external node, we merely delete p. See Figure 1. With careful implementation this method requires $O(\log_k(N/n))$ time in the worst case for a deletion.

Fig. 2. Row displacements computed for an array using "first-fit-decreasing" strategy. Asterisks denote nonzeros. Each position in array C contains the position in A (if any) mapped to that position in C. Positions in A are numbered row by row starting from zero. Row displacements are computed in the order 2, 1, 3, 5, 4.



3. Table Compression by Double Displacement

Section 2 shows that by using tries the worst-case time to access a table can be decreased as much as desired, at the expense of additional storage. If the table to be stored is static, i.e., all the entries take place before all the lookups, then we can improve the method of Section 2 substantially. In this section we describe a double displacement scheme for compression of static tables.

For simplicity we shall assume that N is a perfect square, i.e., $N = m^2$ for some integer m. We shall also assume that $n \geq \max\{2, m\}$. These assumptions hold in the applications we have in mind. We can represent the table to be accessed by an $m \times m$ array A. Position (i, j) in the array corresponds to name k, where $i = \lfloor k/m \rfloor + 1$ and $j = k \bmod m + 1$. Position (i, j) contains the information associated with k if k is present in the table and contains zero if k is absent from the table.

We shall describe a method for compressing A into a one-dimensional array C with fewer positions than A by giving a mapping from positions in A to positions in C such that no two nonzeros in A are mapped to the same position in C. Our mapping is defined by a *displacement* $r(i)$ for each row i; position (i, j) in A is mapped into position $r(i) + j$ in C. The idea is to overlap the rows of A so that no two nonzeros end up in the same position. See Figure 2.

Each entry in C indicates the position in A (if any) mapped to that position in C, along with any associated information. To look up a name k, we compute $i = \lfloor k/m \rfloor + 1$ and $j = k \bmod m + 1$. If $C(r(i) + j)$ contains k, we retrieve the associated information. If not, we know k is not in the table. The access time with this method is $O(1)$; the storage required is m for the row displacements plus space proportional to the number of positions in C. Aho and Ullman [2] and Ziegler [8] advocate this scheme as a way of compressing parsing tables, but they provide no analysis.

To use this method, we need a way to find a good set of displacements. (It is NP-complete to find a set of displacements that minimizes the size of C, or even that comes within a factor of less than 2 of minimizing the size of C [4].) Ziegler suggests the following "first-fit" method: Compute the row displacements for rows 1 through m one at a time. Select as the row displacement $r(i)$ for row i the smallest value such that no nonzero in row i is mapped to the same position as any nonzero in a previous row. An even better method, also suggested by Ziegler, is to sort the rows in decreasing order by the number of nonzeros they contain and then apply the first-fit method. We shall employ this "first-fit decreasing" method. See Figure 2.

If the distribution of nonzeros among the rows of A is reasonably uniform, then the first-fit-decreasing method compresses A effectively. We can verify this empirical observation by an analysis of the method. Our intuition is that rows with only a few nonzeros do not block too many possible displacement values for other rows; it is only the rows with many nonzeros that cause problems. To quantify this phenomenon, we define $n(l)$, for $l \geq 0$, to be the total number of nonzeros in rows with more than l nonzeros. Our first theorem shows that if $n(l)/n$ decreases fast enough as l increases, then the first-fit-decreasing method works well.

THEOREM 1. *Suppose the array A has the following "harmonic decay" property:*

H: For any l, $n(l) \leq n/(l + 1)$.

Then every row displacement $r(i)$ computed for A by the first-fit-decreasing method satisfies $0 \leq r(i) \leq n$.

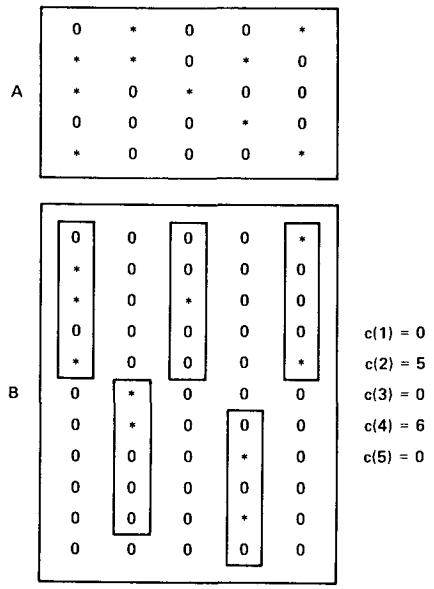
PROOF. For any row i, consider the choice of $r(i)$. Suppose $r(i)$ contains $l \geq 1$ nonzeros. By H the number of nonzeros in previous rows is at most n/l. Each such nonzero can block at most l choices for $r(i)$. Altogether at most n choices can be blocked, and so $0 \leq r(i) \leq n$. □

If the array A has harmonic decay, then by Theorem 1 the row displacement method provides $O(1)$-time table access while requiring only $n + 2m$ words of storage, not counting storage of the information associated with each name. Constructing the table requires $O(n^2 + m)$ time. The appendix contains an implementation of the first-fit-decreasing method and verifies the $O(n^2 + m)$ time bound.

It is useful to reflect a bit on the meaning of harmonic decay. If A has harmonic decay, at least half the nonzeros in A must be in rows with only a single nonzero. In addition, no row can have more than $\sqrt{n}$ nonzeros.

If the array A does not have harmonic decay, the first-fit-decreasing method does not do such a good job of compression, at least in the worst case. Since we are

Fig. 3. Column displacements computed using the first-fit method to satisfy E_j for all j. This constraint requires no rows with more than one nonzero in B_2, at most one such row in B_3 and B_4, and at most two such rows in B_5.



interested in the worst case, we shall look at a way to overcome this difficulty. What we need is a way to smooth out the distribution of nonzeros among the rows of A before we compute row displacements. To accomplish this we apply to A a set of *column displacements* $c(j)$, mapping each position (i, j) into a new position $(i + c(j), j)$. This transforms A into a new array B with an increased number of rows (namely, $\max_j c(j) + m$) but with the same number of columns. See Figure 3. Our expectation is that if we choose an appropriate set of column displacements, then the compression achieved when we apply the first-fit-decreasing method to B will more than compensate for the expansion caused by transforming A into B.

We need a criterion for choosing the column displacements. We use an *exponential decay* condition defined as follows. Let B_j be the array consisting of the first j shifted columns of A. Let n_j be the total number of nonzeros in B_j. Let $n_j(i)$ for $i \geq 0$ be the total number of nonzeros in B_j that appear in rows of B_j containing more than i nonzeros.

E_j: For any $i \geq 0$, $n_j(i) \leq n_j/2^{i(2-n_j/n)}$.

We choose the column displacements to satisfy E_j for all j by employing the first-fit method as follows: Compute the displacements for columns 1 through m one at a time. Select as the column displacement $c(j)$ for column j the smallest value such that B_j satisfies E_j. See Figure 3.

It is difficult to motivate the exponential decay condition without doing a careful analysis. We content ourselves here with a few observations. If $B = B_m$ satisfies E_m, then $n(\lfloor \log_2 n \rfloor) \leq n/2^{\lfloor \log_2 n \rfloor} < 2$, which means no row of B contains more than $\lfloor \log_2 n \rfloor$ nonzeros (recall $n \geq 2$), and $n_j(i) \leq n_j/2^{i(2-n_j/n)}$ for $i = \lfloor \log_2 n \rfloor$ implies $n_j(i) \leq n_j/2^{i(2-n_j/n)}$ for all $i > \lfloor \log_2 n \rfloor$. Furthermore $n(i) \leq n/2^i \leq n/(i+1)$ for $i \geq 0$, and B has a decay rate much faster than harmonic. We seem to be able to obtain this exponential decay just as easily as harmonic decay, at least theoretically in the worst case. In practice it may be beneficial to weaken the exponential decay condition E_j and to employ the first-fit-decreasing method to satisfy it, but these changes do not seem to improve our analysis.

We now bound the expansion caused by using column displacements.

THEOREM 2. *The set of column displacements $c(j)$ computed by the first-fit method to satisfy E_j for all j is such that $0 \leq c(j) \leq 4n \log_2 \log_2 n + 9.5n$ for $1 \leq j \leq m$.*

PROOF. For any column j, consider the situation when $c(j)$ is chosen. In order for a possible choice of $c(j)$ to violate E_j, there must be some i in the range $0 \leq i \leq \lfloor \log_2 n \rfloor$ such that $n_j(i) > n_j/2^{i(2-n_j/n)}$. Since E_{j-1} holds, $n_{j-1}(i) \leq n_{j-1}/2^{i(2-n_{j-1}/n)}$. Each row of B_j with exactly i nonzeros in the first $j-1$ columns and an additional nonzero in column j contributes $i+1$ to $n_j(i) - n_{j-1}(i)$. Each row of B_j with more than i nonzeros in the first $j-1$ columns and an additional nonzero in column j contributes 1 to $n_j(i) - n_{j-1}(i)$. Thus there must be more than $(n_j/2^{i(2-n_j/n)} - n_{j-1}/2^{i(2-n_{j-1}/n)})/(i+1)$ rows in B_j with more than $i-1$ nonzeros in the first $j-1$ columns and an additional nonzero in column j. We can bound this quantity by

$$(n_j/2^{i(2-n_j/n)} - n_{j-1}/2^{i(2-n_{j-1}/n)})/(i+1)$$

$$= (n_{j-1}/2^{i(2-n_{j-1}/n)}) \left(\frac{n_j 2^{i(n_j - n_{j-1})/n}}{n_{j-1}} - 1 \right) \Big/ (i+1)$$

$$\geq (n_{j-1}/2^{i(2-n_{j-1}/n)})(2^{i(n_j - n_{j-1})/n} - 1)/(i+1)$$

since $n_j/n_{j-1} \geq 1$,

$$\geq (n_{j-1}/2^{i(2-n_{j-1}/n)})(i(n_j - n_{j-1})(\ln 2)/n)/(i+1)$$

since $e^x \geq 1 + x$,

$$\geq (i\, n_{j-1}(n_j - n_{j-1}) \ln 2)/(2^{i(2-n_{j-1}/n)} n(i+1)).$$

Consider the set of ordered pairs whose first element is a row of B_{j-1} with more than $i-1$ nonzeros and whose second element is a nonzero of column j. There are at most $n_{j-1}(i-1)(n_j - n_{j-1})/i$ such pairs. Each choice of $c(j)$ for which $n_j(i) > n_j/2^{i(2-n_j/n)}$ accounts for more than $(i\, n_{j-1}(n_j - n_{j-1}) \ln 2)/(2^{i(2-n_{j-1}/n)} n(i+1))$ distinct pairs. Thus the number of choices of $c(j)$ for which $n_j(i) > n_j/2^{i(2-n_j/n)}$ is bounded by

$$\frac{n_{j-1}(i-1)(n_j - n_{j-1}) 2^{i(2-n_{j-1}/n)} n(i+1)}{i^2 n_{j-1}(n_j - n_{j-1}) \ln 2}$$

$$\leq \frac{n_{j-1} 2^{i(2-n_{j-1}/n)} n(i+1)}{2^{(i-1)(2-n_{j-1}/n)} i^2 n_{j-1} \ln 2}$$

by E_{j-1}

$$\leq \frac{2^{(2-n_{j-1}/n)}n(i+1)}{i^2 \ln 2} \leq (4 \log_2 e)n(i+1)/i^2.$$

Summing over i, we find that at most

$$\sum_{i=1}^{\lfloor \log_2 n \rfloor} (4 \log_2 e)n(i+1)/i^2$$
$$\leq (4 \log_2 e)n(\ln \log_2 n + 1 + \pi^2/6)$$
$$\leq 4n \log_2 \log_2 n + 9.5\, n$$

choices of $c(j)$ are blocked, and $0 \leq c(j) \leq 4n \log_2 \log_2 n + 9.5\, n$. □

It is not hard to implement the first-fit method so that it computes column displacements to satisfy E_j for all j in $O(n^2 + m)$ time. We leave the details to the reader.

By combining row and column displacements, we obtain the following table storage scheme.

Table Construction

Step 1. Construct a set of column displacements $c(j)$ for array A by using the first-fit method to satisfy E_j for all j. Compute the transformed array B.

Step 2. Construct a set of row displacements $r(i)$ for B by using the first-fit-decreasing method. Construct the transformed array C.

Table Lookup

Let k be the name to be accessed. Compute $i = \lfloor k/m \rfloor + 1$, $j = k \bmod m + 1$, and $k^* = r(i + c(j)) + j$. If $C(k^*)$ contains k, retrieve the associated information. If not, k is not in the table.

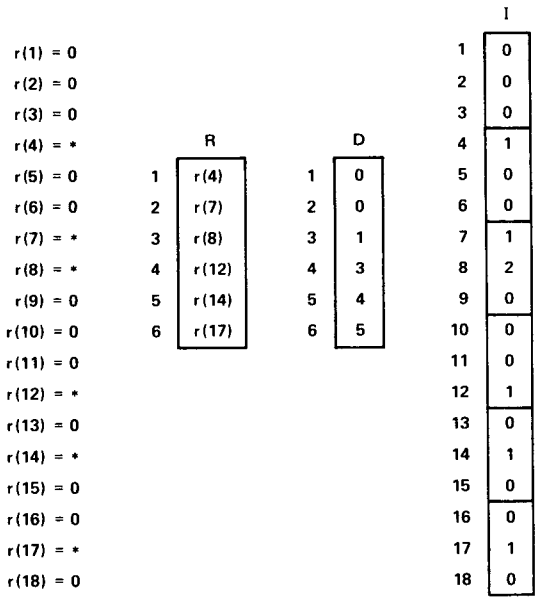
With this method, the access time is $O(1)$, the storage is m for the column displacements plus $4n \log_2 \log_2 n + m + 9.5\, n$ for the row displacements (by Theorem 2) plus $n + m$ for C (by Theorem 1), not including space required to store the information associated with each name. The total space is thus $4n \log_2 \log_2 n + 3m + 10.5\, n$ words. The time required to construct the storage scheme is $O(n^2 + m)$.

4. A Static Table Storage Scheme

By incorporating one more idea into our method, we can reduce its storage to $O(n)$. So far we have not used the ability to pack several small numbers into a single word of storage; we employ this ability now. The double displacement method requires nonlinear storage only for the row displacements, which comprise a table of $\lfloor 4n \log_2 \log_2 n + m + 9.5n \rfloor$ entries, at most n of which are nonzero. Let $d = \lceil 4 \log_2 \log_2 n + m/n + 9.5 \rceil \leq \lceil 4 \log_2 n \log_2 n + 10.5 \rceil$ (recall $n \geq m$).

We store the nonzero row displacements $r(i)$ in an array R of size n, in increasing order on i. That is, $R(1)$ contains the first nonzero $r(i)$, $R(2)$ the next nonzero $r(i)$, and so on. To access the array R, we use a directory consisting of an array D of size n containing *base addresses* and an array I of size nd containing *increments*. To define D and I, we divide r into sections of length d. For each section, there is a corresponding entry in D that gives the location in R of the last nonzero row displacement preceding the section; this is the base address of

Fig. 4. Row displacement directory with $n = 6$, $d = 3$. Asterisks denote nonzero row displacements.



the section. For each nonzero row $r(i)$, $I(i)$ specifies the offset of the location of $r(i)$ in R from the base address of the section containing $r(i)$. More formally, arrays D and I are defined as follows. For any $1 \leq j \leq n$, $D(j)$ is the position in R containing the nonzero row displacement $r(i)$ of maximum i such that $i \leq (j-1)d$. If there is no such nonzero row displacement, $D(j) = 0$. For any $1 \leq i \leq nd$, $I(i) = 0$ if $r(i) = 0$; otherwise $I(i) = p - D(\lceil i/d \rceil)$, where p is the position in R where $r(i)$ is stored. See Figure 4.

We access the row displacements as follows. To look up $r(i)$ for some $1 \leq i \leq nd$, we examine $I(i)$. If $I(i) = 0$, we return $r(i) = 0$. If $I(i) \neq 0$, we return $r(i) = R(D(\lceil i/d \rceil) + I(i))$. This computation requires $O(1)$ time. Thus the double displacement scheme with a row displacement directory has $O(1)$ lookup time.

To compute the storage required by this method, note that each increment stored in I is an integer in the range O to d, where $d \leq \lceil 4 \log_2 \log_2 n + 10.5 \rceil$. By packing I densely, we can store the entire array in $\lceil (nd \lceil \log_2 (d+1) \rceil)/b \rceil$ words of storage, where b is the number of bits per word. Since by our assumptions $b \geq \log_2 n$, the storage required for I is $o(n)$, and the total storage for the double displacement scheme with row displacement directory is $3n + 2m + o(n)$ words, not including space required to store the information associated with each name.

We can combine this method with the trie structure of Section 2 to obtain a static table storage scheme good for arbitrary values of n and N. Our first step is to construct a trie as in Section 2 with n-way branching at all the nodes. The trie has depth at most $\lceil \log_n N \rceil - 1$ and contains n^2 pointers, of which only $n - 1$ are nonnull. We can regard the pointers in this trie as consisting of a table of $n - 1$ entries selected from n^2 possible

names; $\lceil \log_n N \rceil$ lookups of pointers in this table are required to look up an entry in the original table. We use the modified double displacement scheme with $m = n$ to store the pointer table. We thus obtain a method which requires $O(n)$ words of storage and allows $O(\log_n N)$ access time.

5. Remarks

There are several possible applications of our table storage schemes. The trie structure of Section 2 can be used to keep track of fill-in when carrying out sparse Gaussian elimination [7] and to keep track of signatures when finding equivalent expressions [3]. The displacement methods of Sections 3 and 4 can be used to store LR parsing tables [2, 8] and to carry out the numeric factorization step in sparse Gaussian elimination [7].

Although we have not studied the practicality of our methods and have not attempted to minimize the constants in our storage bounds, we believe that appropriate variants of the methods are simple enough to compete with hashing in some situations. Experiments are needed to determine the best way to choose row and column displacements in practice. Our approach can be combined with other tricks to obtain additional table compression. For instance, a common technique in storing LR parsing tables is to overlap identical rows in the parsing table; this idea can be incorporated into our methods. Our intention here has not been to advocate a specific table storage scheme, but rather to show that the efficiency of displacement methods has a theoretical basis, and that the theory can suggest directions in which to look for practical algorithms.

We conclude with a few comments about our theoretical results. The algorithms we have discussed make use of array storage; it seems that they cannot be implemented using only list structures as storage. Thus they indicate a difference in power between random access machines and pointer machines. They also suggest a time-space tradeoff for the table storage problem in the dynamic case. Whether such a time-space tradeoff exists is a question deserving further study.

Appendix: An Implementation of the First-Fit Decreasing Method

Let A be an $m \times m$ array. The following algorithm is a straightforward implementation of the first-fit-decreasing method to choose row displacements $r(i)$ so that no two nonzeros appear in the same column. Input to the algorithm is a list of the nonzero positions in A.

First-Fit Decreasing Algorithm

```
Step 1. for i := 1 until m do
            count(i) := 0; list(i) := ∅ od;
        for each nonzero position (i, j) do
            add one to count(i); put j in list(i) od;
Step 2. for l := 0 until n do bucket(l) := ∅ od;
        for i := 1 until m do put i in bucket(count(i)) od;
Step 3. for k := 0 until n + m do entry(k) := false od;
        for l := n step − 1 until 0 do
            for each i in bucket(l) do
                r(i) := 0;
    check overlap: for each j in list(i) do
                if entry(r(i) + j) then
                    r(i) := r(i) + 1; go to check overlap fi od;
                for each j in list(i) do
                    entry(r(i) + j) := true od od od;
```

After step 1, $list(i)$ is a list of the nonzero columns in row i and $count(i)$ is a count of these nonzeros. Step 2 is a radix sort of the rows by their number of nonzeros. The initialization in step 3 assumes that A has harmonic decay, which is the main case in which we are interested. If A does not have harmonic decay, more space must be allocated for *entry*.

THEOREM 3. *If A has harmonic decay, then the first-fit-decreasing algorithm requires $O(n^2 + m)$ time to compute row displacements for A.*

PROOF. Steps 1 and 2 and the initialization in step 3 require $O(n + m)$ time. For $1 \le i \le m$, let row i contain l_i nonzeros. Then the time to compute the displacement for row i is $O(nl_i)$, and the total time to compute row displacements is $O(\sum_{i=1}^{m} nl_i + m) = O(n^2 + m)$. □

Acknowledgments. Our thanks to Y. Shiloach, for extensive discussions which contributed greatly to the ideas presented here, and to M. Brown, S. Graham, and the referees for constructive suggestions which improved the presentation considerably.

Received August 1978; revised June 1979

References

1. Aho, A.V., Hopcroft, J.E., and Ullman, J.D. *The Design and Analysis of Computer Algorithm.* Addison-Wesley, Reading, Mass., 1974.
2. Aho, A.V., and Ullman, J.D. *Principles of Compiler Design.* Addison-Wesley, Reading, Mass., 1977.
3. Downey, P.J., Sethi, R., and Tarjan, R.E. Variations on the common subexpression problem. To appear in *J. ACM.*
4. Even, S., Lichtenstein, D.I., and Shiloach, Y. Remarks on Ziegler's method for matrix compression. Unpublished manuscript, 1977.
5. Knuth, D.E. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973.
6. Sprugnoli, R. Perfect hashing functions: A single probe retrieving method for static sets. *Comm. ACM 20,* 11 (Nov. 1977), 841–850.
7. Tarjan, R.E. Graph theory and Gaussian elimination. In *Sparse Matrix Computations,* J.R. Bunch and D.J. Rose, Eds., Academic Press, New York, 1976, pp. 3–22.
8. Ziegler, S.F. Smaller faster table driven parser. Unpublished manuscript, Madison Academic Comptg. Ctr., U. of Wisconsin, Madison, Wisconsin, 1977.