SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 21(1), 35–49 (JANUARY 1991)

A Letter-oriented Perfect Hashing Scheme Based upon Sparse Table Compression

CHIN-CHEN CHANG
Institute of Computer Science and Information Engineering, National Chung Cheng University, Chiayi, Taiwan 62107, R.O.C.

AND

TZONG-CHEN WU
Institute of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan 30025, R.O.C.

SUMMARY

In this paper, a new letter-oriented perfect hashing scheme based on Ziegler's row displacement method is presented. A unique *n*-tuple from a given set of static letter-oriented key words can be extracted by a heuristic algorithm. Then the extracted distinct *n*-tuples are associated with a 0/1 sparse matrix. Using a sparse matrix compression technique, a perfect hashing function on the key words is then constructed.

KEY WORDS Perfect hashing functions Letter-oriented hashing Sparse table compression Row displacement method

INTRODUCTION

Hashing is a fast addressing technique for directly accessing data in the memory space. Given a set of keys, one can retrieve the information associated with a key in a very short time through a preconstructed hashing function on keys. A perfect hashing function is defined as a one-to-one mapping from the set of keys into an address space.[1] There are several methods which have been proposed for constructing perfect hashing functions.[1–11] By a sparse table, we mean a table in which the number of non-zero elements, as opposed to the zero elements, which are useless or vacant, is much less than the size of the table itself. Also, there are many schemes that can be used to compress a sparse table into a linear array to yield more efficient memory usage.[12–19] Some of them are frequently used for storing the parsing tables in compiler design.[16–19]

Consider the sets of static letter-oriented keys. We present here a new scheme for constructing perfect hashing functions based on Ziegler's row displacement method.[17] One can extract a unique *n*-tuple on the keys by a heuristic algorithm for a set of letter-oriented key words. The extracted *n*-tuples are associated with a 0/1 sparse matrix. We first apply a sparse matrix compression technique to obtain a more compact matrix, and then decompose the compressed matrix into a set of triangular-like row vectors. A row displacement method is employed to compress these decom-

0038–0644/91/010035–15$07.50

Received 2 May 1990
Revised 20 June 1990

posed row vectors into a more condensed linear array. The displacements of all decomposed row vectors can be determined by applying Ziegler's row displacement method. Then a perfect hashing function for this set of letter-oriented keys is constructed. In the following sections, we will give detailed descriptions of the construction of perfect hashing functions on sets of static letter-oriented keys. Some discussions about practical implementations of our scheme are also presented.

SPARSE TABLE COMPRESSION

There are several known displacement methods that have been proposed for storing sparse tables.[15–19] Among them, many methods can be used as the bases of constructing perfect hashing functions. Since Ziegler's row displacement[17] is simple, we adopt it to form our perfect hashing functions. Therefore, we give a brief description of Ziegler's approach in this section. Reviews of some other compression methods for static sparse tables are included in References 15, 16 and 19.

A sparse matrix with non-zero elements in it is given. For convenience, a matrix is regarded as a set of row vectors or a set of column vectors. Ziegler proposed a row displacement method to compress all row vectors of the matrix into a linear array such that all non-zero elements are placed overlapped in the linear array without conflict to yield more efficient memory usage. By applying Ziegler's row displacement method, the element at the position (i,j) of the matrix can be directly stored at the location $\mathrm{BASE}(i)+j$ in memory, where $\mathrm{BASE}(i)$ is referred to as the row displacement of the ith row vector. Figure 1 shows a graphical illustration of the row displacement method. Furthermore, Ziegler also gave a suggestion that sorting all the row vectors of the matrix in descending order of the number of non-zero elements in them before placing these row vectors into the linear array will obtain better results. It is known that the problem of computing the optimal displacements of row vectors is NP-complete.[12, 20] Tarjan and Yao[12] also pointed out that Ziegler's method, which is sometimes referred to as a first-fit decreasing method, yields excellent results in practice.

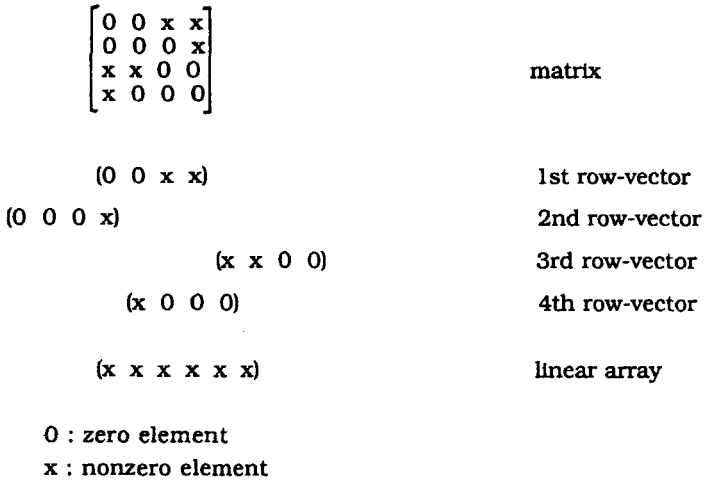


Figure 1. A graphical illustration of the row displacement method

Let the sorted row vectors of a compressed matrix be LL and the maximal length of row vector be m. Ziegler's row displacement method for placing row vectors into a linear array is described as follows:

Step 1. Allocate a set of m free linear locations, denoted by S, in memory.
Step 2. Get a row vector D from LL.
Step 3. Starting from the head of S, search for a free location from which the non-zero elements of D can be fully placed without conflicting with the non-zero elements of the previously placed row vectors in S. If such a location is not found then allocate m more contiguous locations from memory; append them to S and repeat this step until such a location is found.
Step 4. Place D overlapped into S.
Step 5. Repeat from Step 2 until no row vector remains in LL.

OUR SCHEME

This section presents a new perfect hashing scheme for sets of letter-oriented keys. First, we assume that a unique n-tuple has been obtained on the set of keys by some artificial rules. Then we map the extracted distinct n-tuples to a set of entries of a 0/1 sparse matrix M, where non-zero elements in M are represented as 1s; others are represented as 0s. Throughout this section, we use a simple example to explain our scheme in finding perfect hashing functions. A general model of our method is also given in the last part of this section.

Consider the case of the twelve months' identifiers in English listed as below:

JANUARY	MAY	SEPTEMBER
FEBRUARY	JUNE	OCTOBER
MARCH	JULY	NOVEMBER
APRIL	AUGUST	DECEMBER

By extracting the second and the third letters, the twelve distinct extracted pairs (2-tuples) are listed as following:

(A, N)	(A, Y)	(E, P)
(E, B)	(U, N)	(C, T)
(A, R)	(U, L)	(O, V)
(P, R)	(U, G)	(E, C)

Then we produce a 26 × 26 0/1 matrix M associated with the above twelve pairs (see Figure 2). For the sake of readability, the 0s in M are represented as dots.

The reader may notice that the matrix M is rather sparse. Now the problem of our hashing scheme turns out to be how to compress the matrix M of Figure 2 into a more condensed linear array such that the storage used by all extracted n-tuples, i.e. the 1s in the matrix M, is as small as possible. Here we present a straightforward algorithm for compressing a sparse matrix into a more compact one. The compressed matrix is used as the basis of our hashing scheme. First, we shall define some functions which are used later for describing our algorithm more clearly:

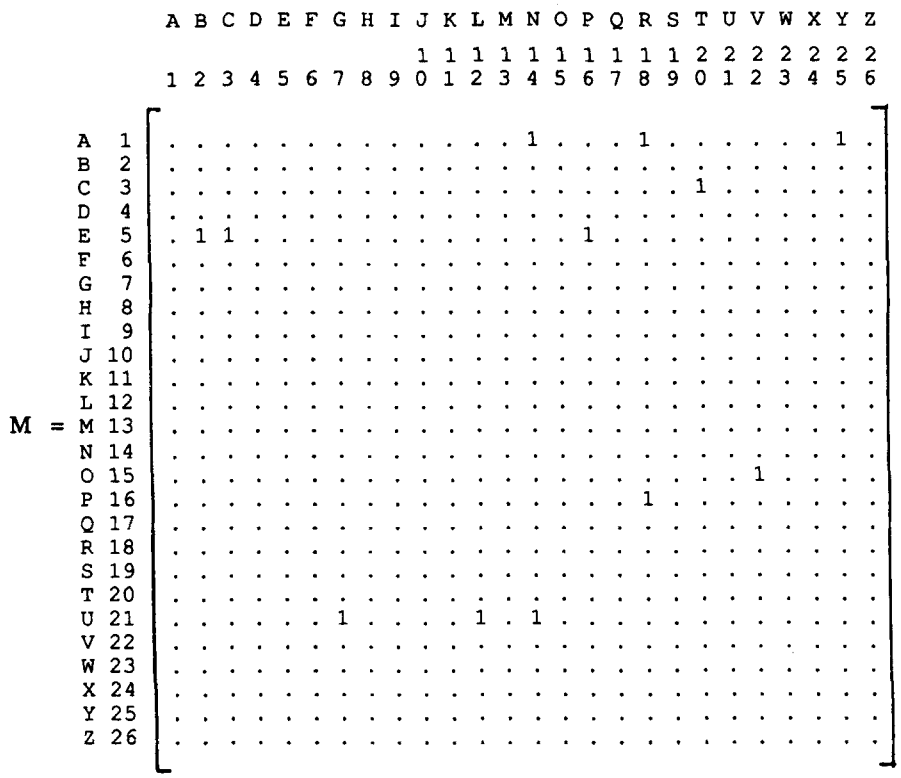


Figure 2

CheckRow(M,*i*,*j*)

For the matrix M, if both the *i*th and the *j*th row vectors have 1s at the same position then return FALSE; otherwise return TRUE.

CheckCol(M,*i*,*j*)

For the matrix M, if both the *i*th and *j*th column vectors have 1s at the same position then return FALSE; otherwise return TRUE.

MoveRow(M,*i*,*j*)

For the matrix M, move the *j*th row vector to the *i*th row vector and set all elements of the *j*th row vector to 0.

MoveCol(M,*i*,*j*)

For the matrix M, move the *j*th column vector to the *i*th column vector and set all elements of the *j*th column vector to 0.

MergeRow(M,i,j)

For the matrix M, place all the 1s of the jth row vector into the same position of the ith row vector and set all elements of the jth row vector to 0.

MergeCol(M,i,j)

For the matrix M, place all the 1s of the jth column vector into the same positions of the ith column vector and set all elements of the jth column vector to 0.

Example 1

Let the matrix A be

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

CheckRow(A,1,2) will return FALSE, because both the first and the second row vectors have 1s at the third position in A. Similarly, CheckCol(A,1,2) will return TRUE. MergeRow(A,1,3) will make the first row vector [0 1 1] and MergeCol(A,2,3) will make the second column vector

$$\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

Again, for the original matrix A, MoveRow(A,1,2) will make A

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

Then MoveCol(A,1,2) will make A

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

Consider a sparse matrix M such as that associated with the example of the twelve months. Let ROW(i) be the index of the row vector into which the ith row vector is merged; let COL(i) be the index of the column vector into which the ith column

vector is merged. The algorithm to produce a more compact form of the sparse matrix M is stated as follows.

Algorithm COMPRESS-MATRIX

Input

An $m \times n$ 0/1 matrix M.

Output

1. A more compact 0/1 matrix CM with p rows and q columns.
2. The number of rows of CM, p.
3. The number of columns of CM, q.
4. ROW(i) and COL(j), for i=1, 2, ..., m and j=1, 2, ..., n.

Step 1 [initialization]

```
For i=1 to m do RF(i) := TRUE;
For j=1 to n do CF(i) := TRUE;
(* RF and CF mean row flag and column flag for indicating if the row and column
   are available for check, respectively. *)
```

Step 2 [merge rows]

```
p := 1;
For i=1 to m do
  If RF(i) = TRUE then
  Begin
    For j=i+1 to m do
      If RF(j) = TRUE and CheckRow(M,i,j) = TRUE then
      Begin
        MergeRow(M,i,j);
        RF(j) := FALSE;
        ROW(j) := p
      End;
    ROW(i) := p;
    MoveRow(M,p,i);
    p := p+1
  End;
p := p-1; (* the number of rows of CM *)
```

Step 3 [merge columns]

```
q := 1;
For i=1 to n do
```

```
    If CF(i) = TRUE then
    Begin
      For j=i+1 to n do
        If CF(j) = TRUE and CheckCol(M,i,j) = TRUE then
        Begin
          MergeCol(M,i,j);
           CF(j) := FALSE;
           COL(j) := q
        End;
      COL(i) := q;
      MoveCol(M,q,i);
      q := q+1
    End;
  q := q-1; (* the number of columns of CM *)
```

Step 4 [output results]

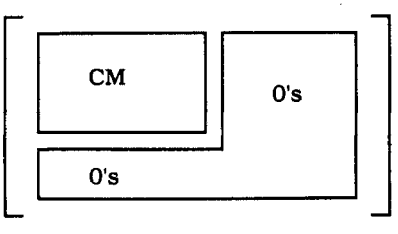
```
Output CM, p, and q;
For i=1 to m do Output ROW(i);
For j=1 to n do Output COL(j);
```

A graphical illustration of the result produced by the above algorithm is shown in Figure 3.

When a more compact matrix CM is obtained, we decompose it into two triangular-like parts, U and L, as shown in Figure 4. Let p be the position (i,j) of CM, if $j \geq \lfloor q\ i/p \rfloor$ then it is in U; otherwise it is in L. That is, some original row vectors are decomposed into two row vectors, one is in U and the other is in L. Now, we define the indices of the decomposed row vectors which will be used for determining the displacements of these row vectors in a contiguous linear array. Let $R_i^{(U)}$ be a row vector in U and $R_i^{(L)}$ be a row vector in L which both are contained in the ith original row vector of CM. Define the index of $R_i^{(U)}$ to be i and define the index of $R_i^{(L)}$ to be $(i - \lceil 2p / q \rceil + p - 1)$. Thus, CM can be decomposed into $(2p - \lceil 2p / q \rceil + 1)$ row vectors in total.

Reconsider the 0/1 sparse matrix M produced by the example of the twelve months. After executing the algorithm COMPRESS-MATRIX, we obtain a compressed 2×8 matrix as follows:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$



M

Figure 3. Results of the algorithm COMPRESS-MATRIX

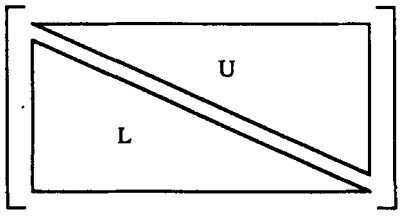


Figure 4. Triangular-like decomposition of the CM *matrix*

The values of ROW (i) and COL (i) are listed in Table I. Figure 5 depicts the triangular-like decomposition of CM with the index of each decomposed row vector to U and L.

Once the matrix CM is produced, the algorithm for determining the displacements of the decomposed row vectors in CM to a contiguous linear array is given as follows.

Algorithm DETERMINE-DISPLACEMENTS

Input

All decomposed row vectors in the matrix CM.

Output

Displacements of all the decomposed row vectors, BASE(i).

Table I. ROW(i) and COL(i) for the matrix M

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
ROW(i)	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	1	1	1	1	2	1	1	1	1	1
COL(i)	1	1	2	1	1	1	1	1	1	1	1	2	1	3	1	4	1	5	1	6	1	7	1	1	8	1

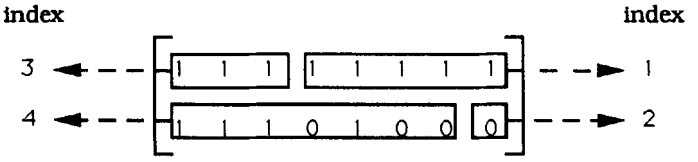


Figure 5. Triangular-like decomposition with indices of CM

Step 1

(a) Record the number of 1s and the index of each row vector.

(b) Sort all row vectors on the number of 1s in their descending order.

(c) Let LL be the list of the sorted decomposed row vectors.

Step 2

(a) Get a row vector from LL with index i.

(b) Apply Ziegler's first-fit decreasing method to place this vector into a linear array.

Step 3

(a) Record the position k at which the first element of the vector is located.

(b) Set BASE(i) = $k-1$.

Step 4

Repeat from Step 2 until no decomposed row vector remains in LL.

By executing the algorithm DETERMINE-DISPLACEMENTS, all displacements of the decomposed row vectors of the matrix CM, i.e. BASE(i), for the twelve months are listed as in Table II.

Let the extracted n-tuple be mapped to the position (r, c) of the original matrix M. The compressed matrix CM has two rows and eight columns. When all values of ROW(i), COL(i) and BASE(i) are determined, by transforming the position of each non-zero element in the matrix M to the location in the linear array applied by Ziegler's row displacement method, we obtain a perfect hashing function for the twelve months in English as below:

$$h(r,c) = \begin{cases} \text{BASE(ROW}(r)) + \text{COL}(c) - 4 * \text{ROW(r)} + 1, \text{ if COL(c)} \geq 4 * \text{ROW(r)} \\ \\ \text{BASE(ROW}(r) + 2) + \text{COL}(c), \text{ otherwise} \end{cases}$$

Table II. BASE(i)s of the CM matrix

i	1	2	3	4
BASE(i)	0	0	10	5

For instance, for the key JANUARY in our example of the twelve months, the extracted character tuple is (A, N) which is expressed as (1,14). From Table I, we obtain ROW(1)=1 and COL(14)=3. Since COL(14) < 4 * ROW(1), the hashing value is computed as

$$h(1,14) = \text{BASE}(\text{ROW}(1) + 2) + \text{COL}(1)$$
$$= \text{BASE}(3) + 3 = 10 + 3 = 13$$

For the key OCTOBER, the extracted character tuple is (C, T) which is expressed as (3,20). Again, ROW(3)=1 and COL(20)=6 by Table I. Since COL(20) $\geq \lfloor 8 * \text{ROW}(3) / 2 \rfloor$, the hashing value is computed as

$$h(3,20) = \text{BASE}(\text{ROW}(3)) + \text{COL}(20) - 4 * \text{ROW}(3) + 1$$
$$= \text{BASE}(1) + 6 - 4 + 1 = 0 + 3 = 3$$

We give the general model of our scheme for constructing perfect hashing functions on sets of letter-oriented keys in the following. For convenience, we use $\bar{x}$ to denote the lexical order of x. For example, $\bar{A}$ is 1, $\bar{B}$ is 2, $\bar{Z}$ is 26 and so on.

Given a set of letter-oriented keys k_i, for i=1,2, ..., N. Let $E_i=(a_{i,1}, a_{i,2}, \ldots, a_{i,n})$ be an extracted n-tuple from k_i. Assume that all N extracted n-tuples are distinct. Let ω be the cardinality of the set of characters appeared in all extracted n- tuples. For instance, if the characters that appeared are the letters from A to Z then ω is 26. From all the E_is, an $s \times t$ 0/1 sparse matrix M is produced, and the corresponding entry (r, c) of M is 1, where $s = \omega^{\lfloor n/2 \rfloor}$, $t = \omega^{n - \lfloor n/2 \rfloor}$,

$$r = \sum_{j=1}^{\lfloor n/2 \rfloor} (\omega^{j-1} \bar{a}_{i,j}) \text{ and } c = \sum_{j=1}^{n - \lfloor n/2 \rfloor} (\omega^{j-1} \bar{a}_{i,\lfloor n/2 \rfloor + j})$$

Let CM be a matrix with p rows and q columns produced by the algorithm COMPRESS-MATRIX. A perfect hashing function on the given N keys is defined as

$$h(r,c) = \begin{cases} \text{BASE}(\text{ROW}(r)) + \text{COL}(c) - \lfloor q * \text{ROW}(r) / p \rfloor + 1, \text{ if COL}(c) \\ \geq \lfloor q * \text{ROW}(r) / p \rfloor \\ \text{BASE}(\text{ROW}(r) - \lceil 2 * p / q \rceil + p + 1) + \text{COL}(c), \text{ otherwise} \end{cases} \quad (1)$$

where the ROW(i) and COL(i) are determined by the algorithm COMPRESS-MATRIX, the BASE(i)s are determined by the algorithm DETERMINE-DISPLACEMENTS, p is the number of rows of CM and q is the number of rows of CM.

One may ensure the correctness of formula (1) by transforming the position of a non-zero element in a matrix to the location in a linear array.[8] The following example illustrates how to map a set of keys with distinct extracted n-tuples to a 0/1 matrix.

Example 2

Let ω be 26. Assume that a three-tuple is used to map the set of keys to a matrix distinctly. Then a 26 × 676 matrix M is produced since $s = 26$ and $t = 676$. Suppose

that one of the three-tuples is (A, C, E). Since $r = 1$ and $c = 26\times3 + 5 = 82$, the corresponding entry (1, 82) of M will be set to 1. In the same way, assume that a four-tuple is used to map the set of keys to a matrix distinctly. Then a 676×676 matrix M is produced since $s = 676$ and $t = 676$. Suppose that one of the four-tuples is (A, C, E, F). Since $r = 26 + 3 = 29$ and $c = 26\times5 + 6 = 136$, the corresponding entry (29, 136) will set to 1.

In general, the algorithm for constructing a perfect hashing function for a set of N keys is stated as follows.

Algorithm CONSTRUCT-PHF

Input

A set of N keys.

Output

p, q, ROW(i)s, COL(i)s and BASE(i)s such that formula (1) is a perfect hashing function.

Step 1

Extract N distinct n-tuples on keys artificially.

Step 2

Produce an $s \times t$ 0/1 matrix M associated with all distinct extracted n-tuples, where $s=\omega\lfloor n/2\rfloor$, $t=\omega^{n-\lfloor n/2\rfloor}$ and ω is the cardinality of the set of characters appeared in all extracted n-tuples.

Step 3

Call the algorithm COMPRESS-MATRIX to compress the matrix M into a compact matrix CM with p rows and q columns and determine the values of ROW(i) and COL(i).

Step 4

Decompose CM into two triangular-like parts U and L as shown in Figure 2. Record the decomposed row vectors with their indices.

Step 5

Call the algorithm DETERMINE-DISPLACEMENTS to find BASE(i).

Step 6

Output p, q, ROW(i), COL(i) and BASE(i).

Here, we give two examples to explain how the algorithm CONSTRUCT-PHF works.

Example 3

Consider the CDC PASCAL reserved words listed as below:

AND	ARRAY	BEGIN	CASE	CONST	DIV
DO	DOWNTO	ELSE	END	FILE	FOR
FUNCTION	GOTO	IF	IN	LABEL	MOD
NIL	NOT	OF	OR	OTHERWISE	PACKED
PROCEDURE	PROGRAM	RECORD	REPEAT	SEGMENT	SET
THEN	TO	TYPE	UNTIL	VALUE	VAR
WHILE	WITH				

For Step 1, let (a, b) be the extracted two-tuple of each reserved word k by the following rules:

1. If length(k) ≤ 3 then a is the first character of k and b is the last character of k.
2. If length(k) > 3 then a is the first character of k and b is the fourth character of k.

Thus, there are 38 distinct extracted two-tuples as below:

(A, D)	(A, A)	(B, I)	(C, E)	(C, S)	(D, V)
(D, O)	(D, N)	(E, E)	(E, D)	(F, E)	(F, R)
(F, C)	(G, O)	(I, F)	(I, N)	(L, E)	(M, D)
(N, L)	(N, T)	(O, F)	(O, R)	(O, E)	(P, K)
(P, C)	(P, G)	(R, O)	(R, E)	(S, M)	(S, T)
(T, N)	(T, O)	(T, E)	(U, I)	(V, U)	(V, R)
(W, L)	(W, T)				

By executing Step 3 to compress the original 26×26 0/1 matrix M produced in Step 2, we obtain a 7×15 compressed matrix CM for which p is 7 and q is 15, and parameters ROW(i) and COL(i). By executing Step 4 and Step 5, the parameters BASE(i) are determined. The compressed matrix CM, the parameters ROW(i), COL(i) and BASE(i) are shown in Figure 6.

Example 4

Reconsider Example 3. For instance, for the key AND, the extracted two-tuple is (A, D), which corresponds to (1, 4). Thus the hashing value is computed as

$$h(1, 4) = \text{BASE}(\text{ROW}(1) - \lceil 2\times 7 / 15 \rceil + 15 + 1)) + \text{COL}(4)$$
$$= \text{BASE}(1 - 0 + 16) + 3 = \text{BASE}(17) + 3 = 21 + 3 = 24.$$

$$CM = \begin{bmatrix} 1&1&1&1&1&1&1&1&1&1&1&1&1&1 \\ 1&1&1&1&1&1&0&1&1&1&0&0&1&0&0 \\ 0&1&1&1&0&0&0&0&0&0&1&0&0&0&0 \\ 0&0&0&1&0&0&0&0&0&0&0&0&0&0&0 \\ 1&0&0&1&0&0&0&0&0&0&1&0&0&0&0 \\ 0&0&0&1&0&0&0&0&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0&1&1&0&0&0&0&0 \end{bmatrix}$$

(a) the compressed matrix CM

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
ROW(i)	1	1	1	1	2	3	2	1	2	1	1	4	3	1	2	1	1	6	2	7	2	1	2	1	1	1
COL(i)	1	1	2	3	4	1	5	2	6	1	7	8	5	9	10	1	1	11	12	13	14	15	1	1	1	1

(b) the ROW(i) and COL(i) s

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
BASE(i)	0	14	12	0	17	0	0	22	24	26	30	31	29	27

(c) the BASE(i) s

Figure 6. Results for CDC PASCAL reserved words

DISCUSSION

Recently, Sager[10] proposed an efficient minimal prefect hashing scheme. Later, Fox *et al.*[7] presented another effective scheme. Sager's algorithm applied his scheme on sets up to 512 words, whereas Fox *et al.*, using an improvement of Sager's algorithm, formed a minimal perfect hashing function for up to 1000 words; however the size of the set of key words that can be hashed by our near minimal perfect hashing scheme depends on the length of the extracted unique n-tuples and the available memory in a practical implementation.

For the space requirements, the two methods proposed individually by Sager and Fox *et al.* require only two words of storage per word hashed. In our method, for the 38 reserved words of the CDC PASCAL programming language, we need $26 \times 2 + 14 = 66$ words, or about 1·7 words of storage per word hashed, where $26 \times 2 = 52$ words are used for ROW(i) and COL(i) and 14 words are used for BASE(i). In general, the number of storage per word required, NSPW, by our method is

$$\text{NSPW} = \frac{\text{number of ROW}(i)\text{s} + \text{number of COL}(i)\text{s} + \text{number of BASE}(i)\text{s}}{N}$$

$$\leq \frac{\omega\lfloor n/2 \rfloor + \omega^{n-\lfloor n/2 \rfloor} + 2\omega\lfloor n/2 \rfloor}{N} \leq \frac{4\omega\lceil n/2 \rceil}{N}$$

or about

$$O\left(\frac{4\omega\lceil n/2\rceil}{N}\right)$$

where ω is the cardinality of the set of characters appearing in all extracted n-tuples, N denotes the number of key words hashed and n is the tuple length. Note that the magnitude of n highly depends on the intelligence of the above-mentioned algorithm for extracting distinct n-tuples. Thus, the space needed by our method is dominated by the extracted n-tuples.

Since the time spent in finding a perfect hashing function for a set of N keys is based on the time complexity of the algorithm DETERMINE-DISPLACEMENTS, we analyse the time required to execute the algorithm. Let the compressed matrix CM have p rows and q columns. We have $N/pq = \rho$, the compression rate, where $0 < \rho \leq 1$. That is, $N = \rho pq$. The time complexity of the algorithm DETERMINE-DISPLACEMENTS is

$$O\left(q\sum_{i=1}^{p} iq\right) = O(p^2q^2) = O\left(\frac{N^2}{\rho^2}\right).$$

Thus, our method has a worst-case time complexity of $O(N^2/\rho^2)$.

CONCLUSIONS

We have presented a near minimal perfect hashing scheme for letter-oriented sets of key words. Our scheme uses Ziegler's row displacement compression technique for producing the parameters of hashing functions. Furthermore, two advantages are achieved:

1. The extracted distinct n-tuples can be represented by a 0/1 matrix and it is suitable for bit-string operations during the construction of hashing functions.
2. The computation of the hashing value for addressing a key is simple.

However, for the space requirement of our method, the sizes of ROW(i) and COL(i) fully depend on the length of the extracted n-tuples from keys and the size of BASE(i) heavily depends on the compactness of the compressed matrix resulted from the adopted scheme for matrix compression. There are still many good methods for static sparse matrix compression. For example, the method proposed by Dürre[15] based on Ziegler's row displacement method was applied well with large dictionaries. It is worth while investigating further the choice of a more suitable compression method as a good basis for a perfect hashing scheme on large word sets. Up to now, researchers have proposed many perfect hashing schemes using extracted n-tuples.[2-11] They all used trial and error to find the needed n-tuples. How to find a good heuristic algorithm to extract a unique n-tuple for an arbitrary list of word sets with the least amount of required time still remains open.

ACKNOWLEDGEMENTS

The authors would like to thank the referees for their very useful comments which improved the presentation of this paper.

REFERENCES

1. R. Sprugnoli, 'Perfect hashing functions: a single probe retrieving method for static sets', *CACM*, **20**, (11), 841–850 (1977).
2. C. C. Chang, C. Y. Chen and J. K. Jan, 'On the design of a machine-independent perfect hashing scheme', to appear in *The Computer Journal* (1990).
3. C. C. Chang and R. C. T. Lee, 'A letter-oriented minimal perfect hashing scheme', *The Computer Journal*, **29**, (3), 277–281 (1986).
4. C. C. Chang and J. C. Shieh, 'On the design of letter-oriented minimal perfect hashing functions', *Journal of the Chinese Institute of Engineers*, **8**, (3), 285–297 (1985).
5. R. J. Chichelli, 'Minimal perfect hashing functions made simple', *CACM*, **23**, (1), 17–19 (1980).
6. C. R. Cook and R. R. Oldehoeft, 'A letter oriented minimal perfect hashing function', *SIGPLAN Notices*, **17**, (9), 18–27 (1982).
7. E. A. Fox, Q. F. Chen, L. S. Heath and S. Datta, 'A more cost effective algorithm for finding minimal perfect hashing functions', *ACM Conference Proceedings*, 1989, pp. 114–122.
8. J. K. Jan and C. C. Chang, 'Addressing for letter-oriented keys', *Journal of the Chinese Institute of Engineers*, **11**, (3), 279–284 (1988).
9. G. Jasechke and G. Osterburg, 'On Chichelli's minimal perfect functions method', *CACM*, **23**, (12), 728–729 (1980).
10. T. J. Sager, 'A polynomial time generator for minimal perfect hash functions', *CACM*, **28**, (5), 523–532 (1985).
11. M. D. Brain and A. L. Tharp, 'Near-perfect hashing of large word sets', *Software—Practice and Experience*, **19**, (10), 967–978 (1989).
12. R. E. Tarjan and A. C. Yao, 'Storing a sparse table', *CACM*, **21**, (11), 606–611 (1977).
13. A. C. Yao, 'Should table be sorted?', *JACM*, **28**, (3), 615–628 (1981).
14. M. L. Fredman, J. Komlos and E. Szemerdi, 'Storing a sparse table with $O(1)$ worst case access time', *JACM*, **31**, (3), 538–544 (1984).
15. K. Dürre, 'Storing static tries', in U. Pape (ed.), *Graphtheoretic Concepts in Computer Science*, Universitäts-Verlag, Rudolf Trauner, Linz, 1984, pp. 125–134.
16. K. Dürre and G. Fels, 'Efficiency of sparse matrix storage techniques', in U. Pape (ed.), *Discrete Structures and Algorithms*, Hanser-Verlag, 1980, pp. 209–221.
17. S. F. Ziegler, 'Smaller faster table driven parser', Madison Academic Computing Center, University of Wisconsin, Madison, Wisconsin, 1977.
18. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley Publishing Co., Reading, Mass., 1977.
19. P. Dencker, K. Dürre and J. Heuft, 'Optimization of parser tables for portable compilers', *ACM Transactions on Programming Languages and Systems*, **6**, (4), 546–572 (1984).
20. M. R. Garey and D. S. Johnson, *Computers and Intractability—A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, San Francisco, 1979.