Consiglio Nazionale delle Ricerche

ISTITUTO DI ELABORAZIONE
DELLA INFORMAZIONE

PISA

Suboptimal solution for PLA multiple column folding

F. Luccio and M.C. Pinotti

Nota interna B4-13

Aprile 1989

SUBOPTIMAL SOLUTION FOR PLA MULTIPLE COLUMN FOLDING

F. Luccio * and M.C. Pinotti **

*Dipartimento di Informatica, Università di Pisa, Pisa, Italy.
**Istituto di Elaborazione dell'Informazione, C. N. R., Pisa, Italy.

Abstract

We study the problem of multiple column folding in the design of compact PLA's of m rows and n columns. A set of theoretical results leads to the construction of a heuristic folding algorithm, which runs in time $O(m^2n + mn^2)$, and provides suboptimal solutions.

1. Introduction

Programmable logic arrays (PLA) are effective means to implement multiple output switching functions [2], [4], [7]. A PLA has the structure of a matrix, as shown in fig. 1(a). The input variables run vertically through a section of the matrix (AND plane), which generates the product terms for the Boolean functions. These terms run horizontally through the PLA, to reach the input terminals of another section of the matrix (OR plane) where the sums of products are formed.The output variables of the PLA emerge vertically from the OR plane.

In fig. 1(a), inputs and outputs are indicated with lower and upper case letters, respectively. A *mark*, shown with a dot, in the AND plane, at the intersection of column i and row j, indicates that the variable i is present in product j. Similarly, a mark in the OR plane, in row j and column k, indicates that the product j is present in sum k.

This work has been partially supported by Ministero della Pubblica Istruzione of Italy.

In the physical design of a chip, a PLA could be directly implemented with the gates in the positions of the marks. However, such a solution cause in general a significant waste of silicon area, because most PLA matrices are very sparse. For this reason, a technique called *column folding* is used, aimed to determining a permutation of the rows of the array, such that a compaction of the columns is then possible, with the same column containing two input or two output variables (fig. 1(b)); or several input variables, or output variables (fig. 1(c)). In the two cases, we speak of *simple* or *multiple column folding*, respectively. Similarly, *row folding*, or combined *row-column folding*, may be considered [1].

Let $A=m \cdot n$ be the area of the given array, where m and n are the numbers of rows and columns, respectively. If simple column folding is used, the resulting area is $A' \geq A/2$, and this figure can go down to A/4 for combined simple row-column folding. The sparsity of the matrix can be better exploited in connection with multiple folding [1], [5]. A theoretical lower bound to the area A' is given by $m \cdot (n_i + n_o)$, where n_i (n_o) is the maximum number of marks presents in the AND (OR) portion of a row. In this case, special paths must be provided to route input and output variables from the left and right sides of the chip to the split physical columns inside the array (fig. 1(c)). If the PLA is part of a larger circuit, some constraints on the mutual positions of inputs and outputs may be imposed (*constrained folding*).

For a given PLA, the aim of folding should be to determine a folded array with minimum area (or minimum number of columns). Since this problem has been shown to be NP-hard [7], efficient heuristic algorithms must be developed. In this paper we study the problem of multiple column folding without constraints, as an extension of the approach given in [6] for simple column folding. Our theoretical approach leads to the construction of a heuristic folding algorithm, which produces interesting results on PLA design.

2. Graph theoretical interpretation of multiple column folding

A graph interpretation of multiple column folding has been first presented in [1], [3]. We follow a similar approach. For a given PLA array P, we introduce an undirected multigraph M = (V, E), that is a graph where two vertices may be connected by more than one edge (fig. 2(a)). M is called *column intersection multigraph*. Each vertex $v \in V$ corresponds to an input or output column of P and retains the *type* (t = input, or t = output) of the corresponding variable. An edge $(v_1, v_2) \in E$ is labeled by h, if the two corresponding columns have a mark on the same row h. This row is associated with *the clique h* of M, that is the set of all vertices connected by edges labelled h. If v_1 and v_2 are connected in M, or are of different types, the corresponding columns cannot be implemented on the same physical column (i.e., the two columns cannot be folded together). As a consequence, the cardinality of the maximum clique of vertices of type input (or output) is a lower bound to the number of input columns (or output columns) of the folded array.

An ordered set $v_1, ..., v_k$, $k \geq 2$, of vertices of the same type t, such that $(v_i, v_j) \notin E$ for $1 \leq i < j \leq k$, is called a *t-column folding list* (*t-CFL*, or simply *CFL* if t is implicit). Such a list will be represented on M with a set of *directed* edges from v_i to v_{i+1}, $1 \leq i \leq k-1$, called the edges of the t-CFL (fig. 2(b)). Note that M has now the two families of undirected and directed edges, with different meanings. A CFL indicates a set of columns that can be folded together in the specified order (e.g., from top to bottom on the new folded column). This order, however, puts some constraints on the permutation of the rows, that may be incompatible with another CFL.

Following [1], we now give a necessary and sufficient condition for the joint folding of several lists. Let S be a set of disjoint CFL's, and let D_S be the set of directed edges inserted in M to represent such CFL's. We say that S can be

physically implemented , if all its CFL's can be jointly folded. Let an *alternating cycle* C in M be a cycle formed by chains of edges of D_S, alternated with single undirected edges of E, where all the edges of D_S inserted in C have the same direction. We have:

Theorem 1 S can be physically implemented if and only if M contains no alternating cycles.

The proof of theorem 1 is essentially the same reported in [1], because the column intersection graph considered in the above paper has an alternating cycle if and only if our multigraph has an alternating cycle.

The number of columns of a folded array is denoted by $n_f < n$ (the rows are still m). The folded array is *optimal* if n_f is minimum. If the array is folded according to a set S of CFL's satisfying theorem 1, we have $n_f = s + u$, with $s = |S|$ and u equal to the number of unfolded columns. Note that $|D_S| = n - s - u$. An optimal folded array is therefore relative a set S which minimizes $(s + u)$, that is, maximizes the cardinality of the set D_S.

3. Structure of the folding algorithm

The heuristic algorithm presented in this paper builds the folded array row by row, and column by column. When a variable v is required for the first time, in the row currently built, a new or an available column c is assigned to v. This column is then made newly available when v is no more required. To describe this mechanism, we need some terminology.

A *new column* c is created, when a variable v is assigned to it for the first time. c assumes the type (input or output) of v, and is then called *taken*, while v is called *alive*. As soon as v is no more needed, v is *killed* and c becomes *free*. At any given step, some variables are alive, some are killed, and the remaining, called *future*

variables, are still to be considered. When a future variable is needed, it is assigned to a new column, or to a free column of the same type (which becomes again taken).

Let P be the array to be folded, and Qj denote the new array under construction, after j rows have been created. Qj is called *j-folded array*. The algorithm starts with $Q_0 = \emptyset$, and builds Q_m, the (final) folded array, according to the following computational structure:

```
for j:=1 to m do  {build Qj}
  SELECT (h); {row h is chosen from P}
  INSERT (h); {Qj is built from Qj-1}
  KILL  {the status of the columns of P and Qj is updated}
endfor
```

The procedure SELECT chooses a row h from P such that h has not been already selected. This choice is guided by the theoretical results of the next section. If they do not apply, the choice is made at random. The procedure INSERT builds Q_{j+1} by inserting in the array the selected row h. If, in the original array P, h has marks in some columns corresponding to future variables, these variables are assigned to free columns of the same type (if any). If these columns are not enough, new columns are inserted in the array. The procedure KILL kills the variables in P no more needed in the remaining rows, and sets free in Q_j the respective columns. The crucial point of the whole algorithm is then the criterion used for choosing row h in SELECT, to be discussed later.

The formal version of the algorithm, called FOLDING ALGORITHM, is presented below. It makes use of five matrices S_P, S_Q, V_R, V_C, Z (see fig. 3, relative to the array of fig. 1(a)). S_P [1:2;1:n] gives information about the columns and associated variables of P. For each column k, the first row S_P [1,k] indicates the type $t \in$ {input, output}, and the second row S_P [2,k] indicates the status $s \in$ {killed, future, alive}, where killed is denoted by -, future is denoted by 0, and alive is denoted by the number of the column of Q to which k has been

assigned. Initially each column is future (fig. 3(a)). The array S_Q [1:3;1:n] is relative to the columns of Qj. In fact S_Q [1,k], S_Q [2,k] and S_Q [3, k] contain the type t ∈ {input, output}, the status z ∈ {new, taken, free}, and the pointer to a t-CFL which will be associated to column k, respectively. The initial contents of S_Q is void.

V_R [1:m] and V_C [1:n] contain further row and column information. In fact, after step i, V_R [h] contains the number of variables relative to the row h of P, which are still future variables for Qj. When row h is selected in Qj, V_R [h] is set to '-'. V_C [k] contains the number of rows of P connected to column k, not yet built in Qj. Initially, V_R and V_C contain the number of marks in each row and in each column of P, respectively (see fig. 3(a)).

The last vector Z [1:m] gives the order in which the rows of P are selected, that is, Z[j] indicates the j-th choice. If applied to the array P of fig. 1(a), algorithm FOLDING builds the folded array Q_8 of fig. 1(c). After step 3 (i.e., after rows 1, 8, and 7 have been selected to build Q_3), the contents of the various matrices are shown in fig. 3(b).

```
FOLDING  ALGORITHM
  create VR[1:m], Z [1:m], VC[1:n], SP[1:2;1:n], SQ[1:3;1:n]
  for h := 1 to m do
      VR [h] := number of marks in the row h of P
  endfor;
  for k := 1 to n  do
      SP [1, k]) := type of column k of P;
      SP [2, k] := 0;        {0 for future}
      VC [k] := number of marks in column k of P
  endfor;
  count := 0;
  Q0 := SQ := Z := blank;     {end inizialization}
  for j := 1 to m do          {build Qj}
      Z[j] := SELECT (h);     {row h is chosen in P}
      INSERT (Z[j]);          {Qj is built from Qj-1}
      KILL                    {update columns status of P and Qj}
  endfor
end  FOLDING  ALGORITHM.
```

```
function SELECT (h : output):
   choose row h from P such that h has not been already selected;
   return h
end  SELECT.
```

```
procedure INSERT (h : input):
     V_R [h] := '-';                    {h has been selected}
     for j := 1 to n do
       if P [h,j] = dot then            {variable j appears in row h}
          begin
          V_C [j] := V_C [j] - 1;       {update V_C}
          if S_P [2, j] = 0 then
           begin
           for w:= 1 to m do            {update V_R}
            if P [w,j] = dot  and V_R [w] ≠ '-' then
            V_R [w] := V_R [w] - 1
           endfor;
           if ∃ a free column c ∈ Q_j-1 with the same type of j
            then
            insert j as the last element  of the list S_Q [3,c]
            else                        {insert new column in Q}
            count := count + 1;
            c := count;
            S_Q [1,c] := S_P [1,j];
            S_Q [3,c] := pointer to a new t-CFL
            j is inserted as the first element  of S_Q [3,c]
           endif;
           S_P [2,j] := c;                {j becomes alive}
           S_Q [2,c] := t {t for taken}
          endif;
          Q[i, S_P [2,j]] := dot
       endif
     endfor
end  INSERT.
```

```
procedure  KILL:
    for j := 1 to n do
    if V_C [j] = 0 then
     begin
     S_Q [2, S_P [2,j]] := f;           {f for column free}
     S_P [j,2] := '-';                  { - for column killed}
     VC[j] := '-';
     endif
      endfor
end  KILL.
```

The time complexity T of FOLDING is O (m(S+I+K)) where S, I and K are the complexities of SELECT, INSERT and KILL respectively. While S will be discussed later, the analysis of the program immediately shows that I is O (n(m+n)) and K is O(n), hence we have:

$$T \in O(mS+m^2n+mn^2) \qquad (1)$$

The correctness of the algorithm is also immediate, since the PLA folded array is directly built row by row, and no alternating cycle may appear (Theorem 1).

4. Theoretical results and the specification of SELECT.

The efficiency of the algorithm FOLDING relays upon a proper choice of row h in the procedure SELECT. This choice will be guided by two theorems, derived in this section, aimed to construct the array Q_{j+1} from Q_j in the best possible way. To present this point formally, denote by F_j a final (i.e., m rows) folded array derived from Q_j by folding the maximum number of the remaining columns. F_j will be called a *j-suboptimal* array. Note that F_j may not be an optimal array, since Q_j may have been built with non optimal choices. Clearly, more than one j-suboptimal array may exist. For a generic j, finding F_j is an NP hard problem, as can be immediately proved by extending the result of [7]. Our aim is then finding a heuristic solution as close to F_j as possible. The procedure will be based on the following results.

Let R(A) denote the set of rows of a subarray A ($P \supseteq A$), and let R(v,A) be the sets of rows of A where variable v has marks. For a given Q_j we have:

Theorem 2. Given $r \in R(P) - R(Q_j)$, let $v_1, ..., v_k$, $k > 0$, be the variables for which b has marks, and let $R(v_i,Q_j) \neq \emptyset$ for $1 \leq i \leq k$. Then, there exists a j-suboptimal array F_j with r in position j + 1.

Proof. Let F_j^* be a j-suboptimal array for Q_j, with row r of P built on row j+q of F_j^*, $1 \le q \le m-j$. If q=1, then $F_j = F_j^*$. If q>1, (see fig.4.a), F_j is built with a cyclic shift of the rows j+1, . . ., j+q of F_j^*, which are brought into positions j+2, . . ., j+q, j+1 (see fig. 4.b). In fact, these rows maintain their marks in the new array. Each variable of F_j^* killed (or born) in row k, $j+1 \le k < j+q$, is killed (or born) in row k+1 in F_j, while the variables in row j+q bring their marks into row j+1. The resulting array clearly is a folded array for P, with the same size of F_j^*, hence is j-suboptimal. ❑

Theorem 3. Let v be a variable of type t with $R(v,Q_j) \neq \emptyset$, and let $B = R(v,P) - R(v,Q_j) \neq \emptyset$. Let u be a variable of type t with $R(u,Q_j) = \emptyset$, such that $R(u,P) \supseteq B$. And, for any other variable w, $R(w,Q_j) = \emptyset$, let $R(w,P) \cap B = \emptyset$. Then:

Case 1. if Q_j has a free column c_f of type t, then there exists an F_j with u assigned to c_f, and all the rows of B in positions j+1, ... , j+|B| (in any order);

Case 2. if Q_j has no free columns, then there exists an F_j with u assigned to a new column, and all the rows of B in position j+1, ... ,j+|B| (in any order).

Proof. Let F_j^* be a suboptimal array for Q_j, such that F_j^* does not satisfy the stated properties for F_j. Then, F_j can be built by rearranging F_j^* as follows.

Case 1. We have two subcases, namely:

1.1 u is assigned to $c_u \neq c_f$ in F_j^* (fig. 5(a)); or

1.2 u is assigned to c_f in F_j^*, but the rows of B are different from j+1, ... , j+|B| (fig. 6(a)).

Let c_v be the column of v in F_j^*; V be the portion of c_v below v, and let $B=\{b_1, \ldots ,b_q\}$.

In case 1.1, let U be the lower portion of c_u including u, and Z be the lower portion of c_f from row j+1. We can then permute the rows by placing $b_1, \ldots ,b_q$ in

positions j+1, ... , j+q, moving (the marks of) U to c_f, moving Z to c_v from row j+q+1, and moving v to c_u (fig. 5(b)).

Note that Z may contain only future variables, hence Z does not have marks in the rows of B by hypothesis, and can start from row j+q+1 in F_j. Any other column c_x is obviously rearranged (see fig. 5).

Case 1.2, is similar, as indicated in fig. 6.

Case 2. As case 1, by letting the upper portion of column c_f to be void up to row j. ❑

Theorem 2 is applied, whenever possible, in the procedure SELECT, leading to the inclusion of the row r in Q_{j+1}. In the example of fig. 1(c), Q_3 is built from Q_2 by selecting row 7 which satisfies theorem 2. Theorem 3 is tried whenever theorem 2 does not apply. In the example of fig. 1(c), theorem 3, case 1, is applied for the selection of row 2 (construction of Q_4) with v = e, t = input, u = c, B = {2}, and c_f = first column. Note that the application of theorem 3 may cause theorem 2 to apply again.

If, starting from subarray Q_j, the algorithm FOLDING reaches a final array by all choices guided by the two theorems, a j-suboptimal array is found. Otherwise, a random row selection [8] is be done at some step j'>j, and a j'-suboptimal array is sought for.

Specifically, the function SELECT takes the following form, for the selection of row h:

```
function  SELECT  (h:  output):
 if (∃ r: r as in theorem 2)
      then h:=r
      else if (∃ v, u: v, u as in theorem 3)
              then h:=b, where b ∈ B;
              else h:=random selection
            endif
 endif;
return(h);
end  SELECT.
```

Note that, if theorem 3 applies, the rows of whole set B could be selected,to build $Q_{j+|B|}$ from Q_j, with an obvious modification of the algorithm.

It can be easily shown that, in SELECT, the existence of row r can be verified in time O (m). Furthermore the existence of columns v, u can be verified in time O (m·n) with clever programming. That is, the time complexity S of SELECT is O (m·n), and, for the overall complexity of the algorithm FOLDING we have from (1):

$$T \in (m^2n + mn^2) \qquad (2)$$

5. Conclusion and discussion

In this note we have presented some theoretical results which guide a heuristic algorithm for multiple folding of PLA's. Our algorithm builds the folded array row by row, with a time complexity comparable to the one of other known algorithms for simple folding (e.g., see [3]). We have run our algorithm on a large set of personality matrices, some of which published in literature. The results on sixteen arrays of different sizes and densities (i.e., the percent of marks in the array) are reported in Table 1. In particular, array 15 is the array of [3]. The FOLDING algorithm could reach better results with a more sophisticated criterion of row selection in function SELECT, at the expense of a higher computational complexity.

In particular theorem 3 can be extended as follows, to treat sets of variables:

Theorem 4. Let V, U be two sets of variables of the same type t, $|V|=|U|$, such that $\forall\ v \in V$: $R(v,Q_j) \neq \emptyset$ and $B(v) = R(v,P) - R(v,Q_j) \neq \emptyset$; and $\forall\ u \in U$: $R(u,Q_j) = \emptyset$. Let $B = \cup_{v \in V} B(v)$, and $\cup_{u \in U} R(u,P) \supseteq B$. And, for any other variable w, $R(w,Q_j) = \emptyset$, let $B \cap R(w,P) = \emptyset$. If Q_j has a set C of free columns of type t, with $|C| \geq |U|$, then there exists on F_j with the variables of U assigned to columns of C, and all the rows of B in position j+1, ... ,j+|B| (in any order).

The proof of theorem 4 is an extension of the one of theorem 3, and is not reported here. Although quite powerful, theorem 4 has not been implemented in the procedure SELECT for sake of simplicity.

array	original			folded		
#	m	n	d%	m	n	d%
1	15	20	10	15	5	41
2	15	15	16	15	7	35
3	15	15	26	15	12	33
4	16	20	14	16	9	31
5	16	20	13	16	9	30
6	20	20	14	20	10	29
7	20	20	13	20	8	33
8	25	15	15	25	7	33
9	25	22	13	25	10	29
10	30	29	13	30	18	20
11	31	20	14	31	13	21
12	35	25	11	35	16	17
13	42	30	13	42	23	17
14	45	40	7	45	19	15
15	52	42	12	52	18	27
16	55	50	6	55	20	15

Table 1. Results of algorithm FOLDING on twenty arrays of m rows, and n columns, with density d.

References

[1] G. De Micheli, A. Sangiovanni-Vincentelli, "Multiple constrained folding of programmable logic arrays: Theory and applications", *IEEE Trans. Computer-Aided Design*, vol. CAD-2, no. 3, pp. 151-167, July 1983.

[2] H. Fleisher and L. I. Maissel, "An introduction to array logic", *IBM J. Res. Dev.*, vol. 19, pp. 98-109, Mar. 1975.

[3] G. D. Hatchel, A. R. Newton, A. Sangiovanni-Vincentelli, "An algorithm for optimal PLA folding", *IEEE Trans. Computer-Aided Design*, vol. CAD-1, no. 2, pp. 63-76, April 1982.

[4] S. Y. Hwang, R. W. Dutton, T. Blank, "A best-first search algorithm for optimal PLA folding", *IEEE Trans. Computer-Aided Design*, vol. CAD-5, no. 3, pp 433--442, July 1986.

[5] K. Ishikawa, T. Ishitani, S. Horiguchi, "A study on Multiple Folded PLA Structure", *Atsugi Eletrical Communication Lab.*, NTT (Japan) 1984.

[6] Y. S. Kuo, T.C. Hu, "An effective algorithm for optimal PLA column folding", *Integration the VLSI journal* , 5, pp. 217-230, (1987).

[7] M. Luby, U. Vazirani, V. Vazirani and A. Sangiovanni-Vincentelli, "Some theoretical results on the optimal PLA folding problem", in *Proc. Int. Circ. and Comp. Conf.* (New York, NY), Oct. 1982, pp.165-170.

[8] D. D. Makarenko, J. Tartar, "A statistical analysis of PLA folding", *IEEE Trans. Computer-Aided Design*, vol. CAD-5, no. 1, pp. 39-51, January 1986.

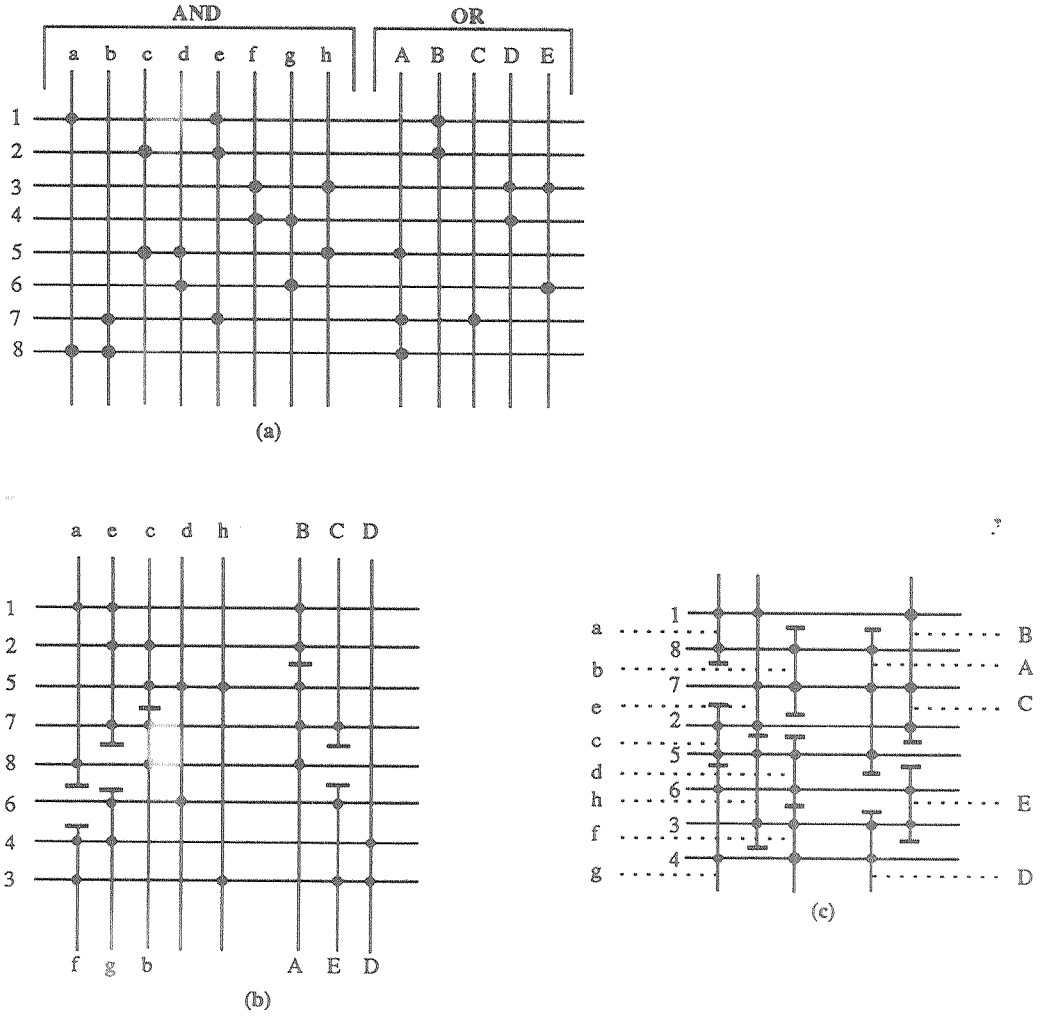


Fig. 1 (a) Example of PLA. (b) Simple column folding. (c) Multiple column folding.

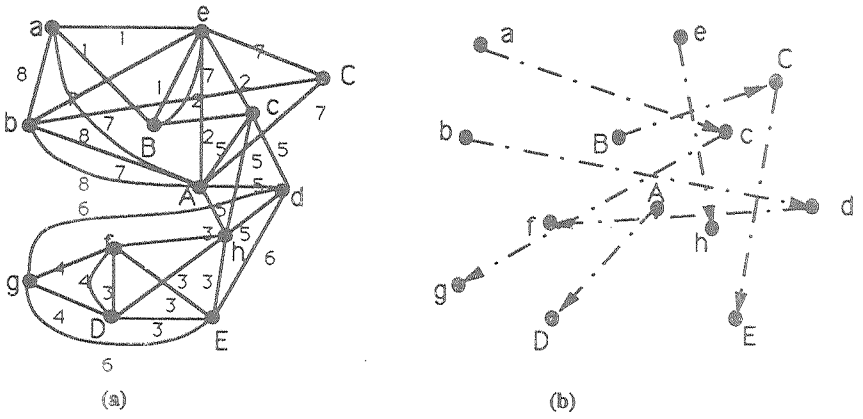
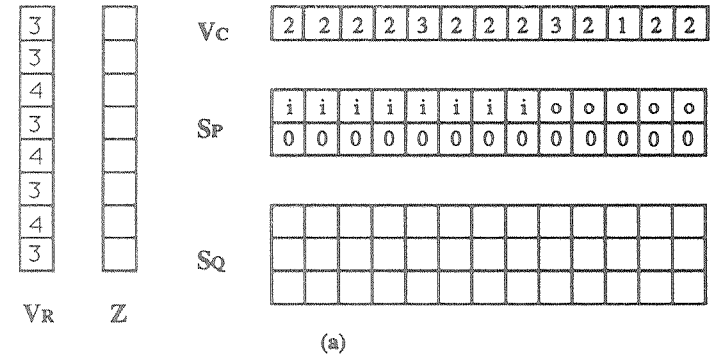


Fig.2 (a) Column intersection multigraph M of sample PLA in fig.1(a). (b) The CFL's for fig.1(c).



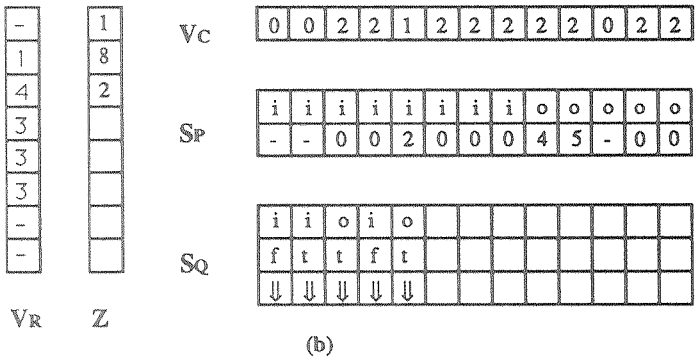
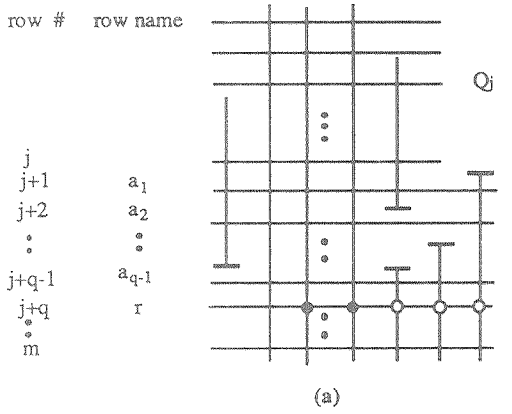


Fig.3 (a) Initial matrices contents for the array of fig.1(a).
(b) Situation after step 3 of the folding algorithm.



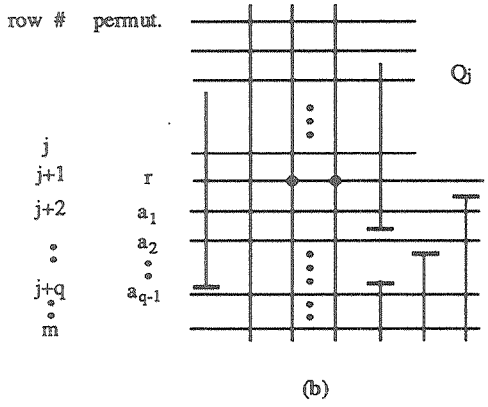


Fig. 4 Proof of Theorem 2.
(a) A j-suboptimal array F_j^* for Q_j (white dots indicate that a mark cannot be present).
(b) The array of (a) after a cyclic shift of the rows.

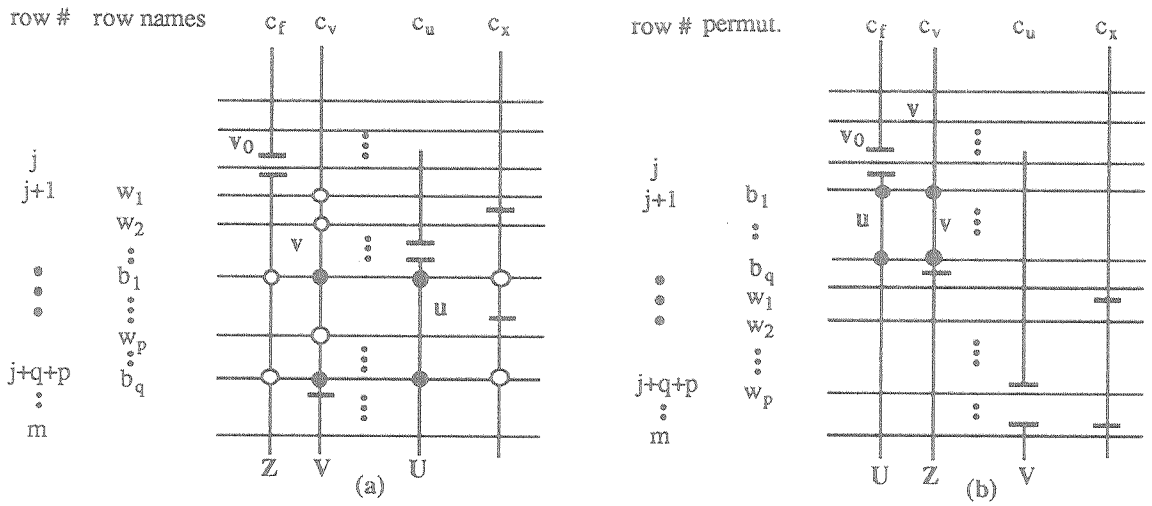


Fig. 5 Proof of Theorem 3, case 1.1 with B = {b_1,...,b_q}.
(a) A j-suboptimal array F_j *.
(b) F_j obtained rearranging F_j *.

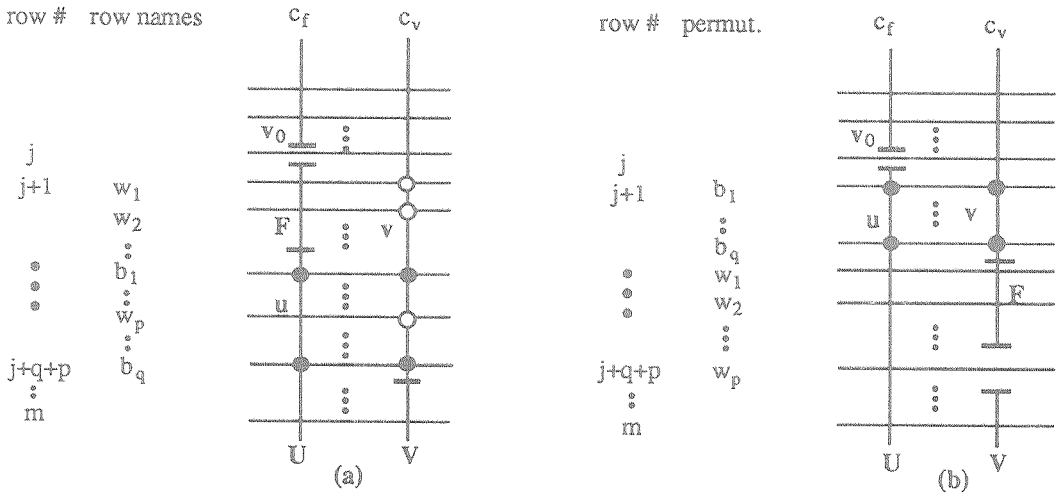


Fig. 6 Proof of Theorem 3, case 1.2.
(a) A j-suboptimal array F_j *.
(b) F_j obtained rearranging F_j *.